



Database Integrity: Challenges and Solutions

by Jorge H. Doorn and Laura Rivero (eds)

ISBN: 1930708386

[Idea Group Publishing](#) © 2002 (344 pages)

Geared toward designers and professionals interested in the conceptual aspects of integrity problems in different paradigms, this text gives a thorough examination.

Table of Contents

[Database Integrity—Challenges and Solutions](#)

[Preface](#)

[Chapter I](#)

- Introduction

[Chapter II](#)

- Database Integrity—Fundamentals and Current Implementations

[Chapter III](#)

- Preserving Relationship Cardinality Constraints in Relational Schemata

[Chapter IV](#)

- Integrity Constraints in an Active Database Environment

[Chapter V](#)

- Integrity Constraints in Spatial Databases

[Chapter VI](#)

- Consistent Queries Over Databases With Integrity Constraints

[Chapter VII](#)

- Translating Advanced Integrity Checking Technology to SQL

[Chapter VIII](#)

- Functional Dependencies for Value Based Identification in Object-Oriented Databases

[Chapter IX](#)

- Integrity Issues in the Web—Beyond Distributed Databases

[Chapter X](#)

- Integrity Maintenance in Extensible Databases

[Index](#)

[List of Figures](#)

[List of Tables](#)

Database Integrity—Challenges and Solutions

Jorge H. Doorn Universidad Nacional del Centro de la Provincia de Buenos Aires,

Argentina

Laura C. Rivero Universidad Nacional del Centro de la Provincia de Buenos Aires and
Universidad Nacional de La Plata,

Argentina

Idea Group Publishing

Information Science Publishing

Hershey • London • Melbourne • Singapore • Beijing

Acquisition Editor: Mehdi Khosrowpour

Managing Editor: Jan Travers

Development Editor: Michele Rossi

Copy Editor: Nicholas Tonelli

Typesetter: LeAnn Whitcomb

Cover Design: Tedi Wingard

Printed at: Integrated Book Technology

Published in the United States of America by

Idea Group Publishing

1331 E. Chocolate Avenue

Hershey PA 17033-1117

Tel: 717-533-8845

Fax: 717-533-8661

E-mail: cust@idea-group.com

Web site: <http://www.idea-group.com>

and in the United Kingdom by

Idea Group Publishing

3 Henrietta Street

Covent Garden

London WC2E 8LU

Tel: 44 20 7240 0856

Fax: 44 20 7379 3313

Web site: <http://www.eurospan.co.uk>

Copyright © 2002 Idea Group Publishing

All rights reserved. No part of this book may be reproduced in any form or by any means, electronic or mechanical, including photocopying, without written permission from the publisher.

Library of Congress Cataloging-in-Publication Data

Doorn, Jorge H., 1946-

Database integrity: challenges and solutions/Jorge H. Doorn, Laura C. Rivero.

p. cm.

Includes bibliographical references and index.

1-930708-38-6

(cloth)

1. Database management. 2. Database security. J. Rivero, Laura C., 1956-II. Title

QA76.9.D3 .D685 2002
005.74--dc21 2001059405

eISBN 1-59140-024-4

British Cataloguing in Publication Data

A Cataloguing in Publication record for this book is available from the British Library.

About the Authors

Jorge Horacio Doorn is full professor in the Computer Science Department at the Universidad Nacional del Centro (UNCPBA), Argentina, since 1989. He has wide experience in actual industrial applications. He has been project leader in several projects and currently he is the leader of the database research team in the Computer Science and Systems Department. His research interests include compilers design and database systems.

Laura Rivero has received her BS degree in systems analysis from the Universidad Nacional del Centro (UNCPBA), Argentina, in 1979. She is a Professor in the Department of Computer Science and Systems of the UNCPBA and a doctoral student in Computer Science at the Universidad Nacional de La Plata. Her lecturing and research activities concentrate on data structures and database design and integrity.

Juan M. Ale is Professor in the Computer Science Department, Faculty of Engineering of Buenos Aires University. He holds degrees in scientific computation, systems engineering and computer sciences from Buenos Aires University. His current research interests include data mining, data warehousing and temporal databases.

Karla A. V. Borges received her B.S. degree in Civil Engineering in 1982 from PUC-MG, and her M.Sc. in Informatics and Public Administration from João Pinheiro Foundation, in 1997. She is the leader of geographic data modeling efforts for the GIS project of Belo Horizonte, Brazil, and the former head of the Urban Management Applications Department at Prodabel (Belo Horizonte's municipal information technology company). Currently, she is a Ph.D. student at the Computer Science Department of Universidade Federal de Minas Gerais. Her main interests are geographic databases, geographic data modeling, urban GIS, and ontologies.

Elena Castro received the M. Sc. in Mathematics from Universidad Complutense of Madrid in 1995. Since 1998 she has worked as assistant lecturer in the Advanced

Databases Group of the Computer Science Department at the Universidad Carlos III de Madrid. She is currently teaching Relational Databases. Her research interests include database conceptual and logical modelling, advanced database CASE environments and information engineering.

Dolores Cuadra received the M. Sc. in Mathematics from Universidad Complutense of Madrid in 1995. Since 1997 she has worked as assistant lecturer in the Advanced Databases Group in the Computer Science Department at the Universidad Carlos III de Madrid. She is currently teaching Database Design and Relational Databases. Her research interests include database conceptual and logical modelling and advanced database CASE environments.

Clodoveu A. Davis Jr. received a B.S. degree in Civil Engineering in 1985 from Universidade Federal de Minas Gerais. He also has M.Sc. and Ph.D. degrees in Computer Science, also from Universidade Federal de Minas Gerais, in 1992 and 2000, respectively. He led the team at Prodabel that conducted the implementation of GIS technology in the city of Belo Horizonte, Brazil, and coordinated several geographic application development efforts. Currently, he is a researcher at Prodabel's Development and Studies Center, and the editor of Informatica Publica, a Brazilian journal on information technology for the public sector. His main interests are urban GIS, geographic databases, map generalization, and multiple representations in GIS.

Hendrik Decker graduated in Computer Science and Mathematics at the Technical University of Munich (Germany). He did his PhD in the departments of Electrical Engineering and Computer Science at the University of Kaiserslautern (Germany). He was a researcher in the Knowledge-Bases group of the European Computer-Industry Research Centre, from 1984 to 1990. Then, he was involved in database research, development and customer consultancy at Siemens in Munich until 2001. Early in 2002, he joined the Instituto Tecnológico de Informática in Valencia (Spain) as a research project manager. More information can be found at <http://www.pms.informatik.unimuenchen.de/mitarbeiter/assoziierte/hdecker>.

Mauricio Minuto Espil is professor at the Catholic University of Argentina. He received a degree in computer science from Buenos Aires University. His current research interests include data warehousing, OLAP tools, non-monotonic reasoning and XML databases.

Paloma Martínez Fernández got a degree in Computer Science from Universidad Politécnica of Madrid in 1992. Since 1992, she has been working at the Advanced Databases Group in the Computer Science Department at Universidad Carlos III of Madrid. In 1998 she obtained the Ph.D. degree in Computer Science from Universidad Politécnica of Madrid. She is currently teaching Database Design, Advanced Databases in the Computer Science Department at the Universidad Carlos III de Madrid. She has been working in several European and National research projects about Natural Language Processing, Advanced Database Technologies, knowledge-based systems and Software Engineering.

Viviana Ferraggine has received her BS degree in systems engineering from the Universidad Nacional del Centro (UNCPBA), Argentina, in 1997. She is currently an Auxiliary Assistant with the Department of Computer Science and Systems and a master student in Computer Science at the Universidad Nacional de La Plata. Her research interests include database systems and data structures.

Sergio Greco received its laurea degree in electrical engineering from University of Calabria, Italy. Currently, he is a full professor at the faculty of Engineering at the University of Calabria. Prior of this, he was researcher at CRAI, a research consortium in Informatics and assistant professor at the University of Calabria. He was a visiting researcher at the research center of Microelectronics and Computer Center (MCC) of Austin (Texas) and at the Computer Science Department of University of California at Los Angeles. His area of research includes database theory, deductive database, logic programming, and query languages.

Hans-Joachim Klein received his diploma in computer science from the University of Saarbrücken, Germany, and his doctorate degree and the *venia legendi* from the University of Kiel, Germany. He has taught at several universities. Currently, he is a lecturer at the Department of Computer Science of the University of Kiel. His research interests include incomplete information in databases, graphical interfaces, integrity constraints for object-oriented data models, and applications of graph-theoretic and algebraic methods in crystallography.

Alberto H. F. Laender received the B.S. degree in Electrical Engineering and the M.Sc. degree in Computer Science from the Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, in 1974 and 1979, respectively, and the Ph.D. degree in Computing from the University of East Anglia, Norwich, UK, in 1984. He joined the Computer Science Department of the the Universidade Federal de Minas Gerais in 1975, where he is currently a Full Professor and the head of the Database Research Group. He was also twice the Coordinator of the Computer Science Graduate Program (1987–89 and 1993–96). In 1997, he was a visiting scientist at the Hewlett-Packard Palo Alto Laboratories. He has served as a program committee member for several international conferences on databases and Web-related topics, and was one of the program committee co-chairs of 19th International Conference on Conceptual Modeling held in Salt Lake City, Utah, in October 2000. He is also a founder member of the Brazilian Computer Society and an Editorial Board member of the *Journal of the Brazilian Computer Society* and of the *Information Systems Review*. His research interests include conceptual database modeling, database design methods, database user interfaces, semistructured data, and Web data management.

A.C. Gómez Lora is a graduate student in the Department of Computer Sciences of the University of Málaga. He received the M.S. degree in Computer Sciences from the University of Málaga in 1997 and he is currently working towards the Ph.D. degree at this University. He is currently supported by a research grant. His research interests include hybrid techniques for recursive query evaluation and optimisation in distributed systems and optimisation techniques at query evaluation time.

J.F. Aldana Montes received the Ph. D. degree from the University of Málaga, Spain, in 1998. He presently holds the rank of Assistant Professor in the Computer Sciences Department of the University of Málaga. Dr. Aldana acted as program chair for the JISBD from 1999–2001. One of his current research programs involves the study of amalgamation of databases and web technologies. His areas of interest include Evaluation and Optimisation of Recursive queries, on Datalog and SQL; distributed evaluation of XML; (semantic) optimisation of XML queries; semantic integration of information on the Web.

Carlos Nieto received the M.Sc. in Mathematics from Universidad Complutense of Madrid in 1991. He has working in several design and architecture research projects for the Spanish Public Office. Since 1997 he has worked as assistant lecturer and teacher at the Advanced Databases Group in the Computer Science Department at the Universidad Carlos III de Madrid. His research interests include conceptual modelling theories as well as specification languages.

Jochen Rasch received his diploma and doctorate degree in computer science from the University of Kiel, Germany, where he worked from 1993 to 1998 as a research and teaching assistant at the Department of Computer Science. He is currently working for SAP AG, Walldorf, on object models for Customer Relationship Management applications. His interests include object-oriented modeling of business applications, relational implementation of object-oriented data models, and value-based integrity constraints for object databases.

Ulrich Schiel holds a Master in Informatics at PUC-Rio de Janeiro, Brazil (1978) and Dr. rer. nat. at University of Stuttgart, Germany (1984). Visiting researcher at GMD-IPSI, Darmstadt-Germany (1988-1999). Since 1978 professor at Federal University of Paraíba, Brazil. Research projects in Temporal Object-Oriented Databases and Multilingual Information Indexing and Retrieval. Additional research interests in Information Systems on the Web, Ontologies and Information Systems Design.

Manuel Velasco B.Sc. (1993) and Ph.D. (1998) degree in Computer Science from Universidad Politécnica of Madrid. He is working as lecturer and researcher in the Computer Science Department at the Universidad Carlos III de Madrid. He has been working in several researching fields in Software Engineering as Reuse and Software Testing mainly, as well as in Information Science, with publications in international papers and congresses. He has participated in some projects funded by the European Union, collaborating with another universities and companies.

M.I. Yagüe del Valle received the B.S. and M.S. degrees in computer sciences from the University of Granada, Spain, in 1990 and 1992, respectively. She worked at Vigo University, Spain, as a lecturer, from 1993–1996. She joined the University of Málaga in 1996 where she presently has a permanent position as lecturer in the Computer Sciences Department. She is currently working towards the Ph.D. degree at the University of Málaga. Her research interests include the application of XML related technology to

databases, and the Semantic Web. She is currently studying the application of semantic information on query optimisation on the Web.

Ester Zumpano received the laurea degree in computer science engineering from University of Calabria, Italy. Currently, she is a Phd student at the faculty of Engineering at the University of Calabria. Her area of research includes logic programming, deductive database, database integration, and query languages.

Preface

The objective of this book is to give both innovative and classic knowledge about database integrity concepts. Chapters covering topics on several well-established research areas give the state of the art on basic database integrity issues, active databases, SQL databases and geographical databases, including integrity support in current SQL-compliant commercial systems. Chapters on novel subjects focus on specific problems on recent database paradigms.

Chapters 2 through 5 are included in the first group whereas chapters 6 to 10 conform the second group. A brief summary of each chapter is given in the proper section of the [Chapter 1](#).

[Chapter 2](#) describes from a high semantic level, the integrity problems in the real world focusing on the granularity of the involved concepts. In this sense, the first concept considered is the domain, later the relation and finally more complex restrictions. A mapping from real world constraints to database world constraints is the heart of the first part of this chapter, while the second part exemplifies how this mapping is seeing in SQL-compliant commercial products. The degree of adhesion of every product to the current SQL standard is analyzed, showing how it influences the mapping.

[Chapter 3](#) addresses a very important topic in database design that has been almost neglected in the literature. It deals with some aspects of the transformation of conceptual schematas into logical ones, such as the Entity Relationship construct: the relationship and its associated cardinality constraints.

In the active database context, [Chapter 4](#) surveys the interaction among active rules and integrity constraints from both the static point of view following the recent SQL standard and the dynamic point of view using temporal logic formalism.

[Chapter 5](#) focuses on the relationship existing between the nature of spatial information, spatial relationships, and spatial integrity constraints. The authors propose the use of OMT-G, an object-oriented data model for geographic applications, at an early stage in the specification of integrity constraints in spatial databases.

Even though integrity constraints are usually used to define constraints on data, [Chapter 6](#) illustrates their applicability in several contexts such as semantic query optimization, cooperative query answering, database integration and view update.

The main goal of [Chapter 7](#) is to arrive at a coherent technology for deriving efficient SQL triggers from declarative specifications of arbitrary integrity constraints. In this chapter, the author describes how to implement advanced *datalog* technology for integrity checking in the framework of SQL, showing how to represent and evaluate arbitrarily complex constraints in SQL without incurring major disadvantages usually associated with integrity checking.

[Chapter 8](#) focuses on the generalization of the well-known functional dependencies to object schemas, offering insights on one of the fundamental concepts of the object-oriented approach: object identity. Then the authors describe an approach to generalize functional dependencies to object functional dependencies.

[Chapter 9](#) addresses concepts of a recently introduced paradigm: the Web as the database, and its implications regarding the progressive adaptation of database techniques to Web usage. This chapter deals with different issues related to integrity and its maintenance on the Web and introduces the reader to other related and open issues, such as the query problem and query optimization on the Web.

Finally, [Chapter 10](#) introduces an approach to integrate integrity constraints to the system, as rules into a general (schema) to allow an easy way to define the semantics of a complex data model. This approach is scalable since rules systems can, at any time, be expanded to incorporate concepts of new applications.

Summing up, this book provides an exciting opportunity to understand relevant topics on integrity in databases, and to find out current trends and solutions for consistency problems in different database paradigms.

Acknowledgments

The editors would like to thank Mehdi Khosrow-Pour for the opportunity he gave us and the Idea Group Publishing staff, mainly Jan Travers, Carrie Stull and Michele Rossi, for their help in advising us how to solve multiple problems and for their guidance and professional support. We acknowledge the support of the chapter authors who also helped us, doing an invaluable job refereeing other chapters. Thanks also to the external reviewers who had provided many constructive comments:

MSc. Silvia Gordillo from LIFIA, Universidad Nacional de La Plata, BA, Argentina;

Dr. Stefano Ceri at the Dipartimento di Elettronica e Informazione, Politecnico di Milano in Milano, Italy;

Dr. Robert Laurini from LISI-Bât. Blaise Pascal INSA de Lyon and LISI-IUT Génie Informatique Université Claude Bernard Lyon I in France;

Dr. Mario Piattini at the Escuela Superior de Informática, Universidad de Castilla-La Mancha. Spain and

Eng. Guillermo Unger at the Software Division of IBM Argentina.

A special note of thanks goes to the anonymous reviewers from Sybase and Oracle.

Jorge H. Doorn

Laura C. Rivero

Chapter I: Introduction

**Jorge H. Doorn, Universidad Nacional del Centro de la Provincia de Buenos Aires,
Argentina**

**Laura C. Rivero, Universidad Nacional del Centro de la Provincia de Buenos Aires
and Universidad Nacional de La Plata,**

Argentina

**Viviana E. Ferraggine, Universidad Nacional del Centro de la Provincia de Buenos
Aires,**

Argentina

INTRODUCTION

Computers are used to manage information. This use may range from collecting a small piece of data, performing a calculation and producing an output as in an embedded micro-controller application to the processing and storage of huge amounts of complex data seen in large databases. Design software to perform such information management is a difficult task. This book is oriented to discuss problems that arise in software products with significant amounts of data. Software developers have to deal with the capture and understanding of complex requirements, the design of the architecture of the software, and the development of the planned software artifacts. Along with the software product life cycle, many other activities are carried out, such as setting the software into service, training users and adapting to a changing world. The proper requirement's elicitation of a software product is a key factor in the success of the whole process.

However, these requirements are not easy to deal with. They have different intrinsic nature and they may appear showing different faces. In many cases, some requirements are totally or partially hidden in the information collected by the software developers (Jackson, 1995).

More than one criterion may be used to characterize requirements. One that seems to be useful is to divide them into those properties related to what the software has to do, and those properties that the software should have. The first group is usually called *Functional Requirements* and the second *Non Functional Requirements*. A Functional Requirement could be, for example, "The system should present the current balance of the customer account in the screen" and a related Non Functional Requirement may be, "The customer balance should be ready in less than five seconds".

In most cases, Non Functional Requirements are harder to perceive and model than Functional Requirements. Functional Requirements are usually expressed as procedures, methods or activities related to the software behavior. On the other hand, Non Functional Requirements are expressed as rules or properties that must be satisfied in a more declarative way.

Non Functional Requirements cover different areas of the desired product such as security, performance and output quality. The data to be stored and processed by the software have properties that must be ensured. Examples of data properties are found everywhere; however, they are usually disregarded. This occurs because most data properties are obvious and everybody knows about them, but also mostly everybody forgets them. For example, there is no need to say that the age of one person is always younger than the age of his or her parents. Everybody knows it, but the Database Engine where this data is stored does not (Loucopoulos & Karakostas, 1995).

The data and procedures approach to software design have to deal with the problem of the properties of the data. The object oriented approach works with objects and relationships among them. It seems that there is no data involved in this analysis. Actually, the problem from the data properties point of view is a little worse since the attributes of involved objects have properties and maybe these properties relate one object attribute with another object attribute. This is because the object orientation watches the Universe of Discourse using a model that has an extra layer between the developer and the data properties. This disadvantage does not damage the advantages of the object-oriented approach; however, the developer has to cope with it.

Not every data property must be modeled; it must be looked at carefully to see if it is needed in the context of the scope and in the objective of the software artifact. A more analytical approach may order the data properties, taking into account their importance (Karlsson, 1996).

When a data property describes the allowed values for attributes, it is called *Domain Property*. Another kind of data properties establishes connections among different attributes; these are known as *Relationships*. When a data property carries out a semantic

that is specific to the Universe of Discourse not found in any other occurrence of the same data, it is called *Business Rule* (Ceri et al., 1997), (Codd, 1990), (Ross, 1997).

DATA PROPERTIES

Data properties should be seen from the point of view of being as independent as possible from the representation model. In this way, data properties can be analyzed from two different perspectives: from the real world and the database world. A real world data property of a given class may be mapped into a different database class of properties due to materialization issues. This depends on the database paradigm and on the adhesion of the DBMS to that paradigm.

These subjects are extensively analyzed in [Chapter 2](#). The following sections only introduce some concepts.

Domain Properties

Programming Languages and Database Engines have a set of built-in data types whose main purpose is to deal with the Domain Properties of the data to be stored. These data types are useful and have been used for decades, helping users to take care of their data processing needs.

As well as all the other Non Functional Requirements, Domain Properties are expressed in the Universe of Discourse in declarative ways. A Domain Property defines the Set of Values that the attribute may have. Sets may be defined by enumeration of the members or by abstraction. It becomes then natural to think, "... everything seems to be okay, the only problem is to map the Universe of Discourse declarative rule to one of the data type offered by the engine...." The issue here is that this mapping is not always possible. Consider the following examples:

- Age of the employee is a non-negative integer number.
- The quantity of product in stock is a non-negative real number.
- The customer name is made up of letters and a few other characters.
- The street name used in supplier's address must be an actual town street.
- The shirt colors in the price list must be one of the cloth supplier colors.

The non-negativeness condition of the employee age and the quantity in stock cannot be mapped into the built-in type systems of most of the current Database Engines. The same thing happens with the subset of characters allowed for customer name. In some Database models, these three examples can be handled with small methods or procedures that specialize the basic data type, keeping in some sense the declarative flavor of the restriction but implemented with procedural technique. The key issue here is that when

the implementation of the restriction is hidden and strongly coupled with the data itself, the mapping is almost perfect.

The street name domain cannot be defined in a computer environment in a declarative or "almost declarative" way. It must be defined by extension. In other words, a data resource holding all existent street names should be created to make possible the enforcement of this domain property. The shirt colors domain has to be defined by extension, too. Domain properties can be even worse. For example, what would happen if the shirt colors "... must be one of those chosen by the marketing department for this promotion..."?. Or simply "... must be one of those the sales manager likes..."?.

Within a certain time framework, the complete street name set may be considered unchangeable or static. Otherwise, the complete color set appears as dynamic. Both domain properties, defined by extension, are difficult to handle and dynamics are the harder ones.

Poor design may create, in some paradigms, inclusion relationships among data in the same or different tables or objects that could be confused with domain properties. These situations will be detailed in next sections.

Relationship Properties

The connection among different attributes is the source of most of the data processing richness and problems. These connections have a scope larger than domain restrictions since they involve several attributes—at least two—usually belonging to different objects or entities. When people buy articles or students attend courses, the link between them is a very important issue.

When software artifacts are involved, the links among real world things (persons, objects, activities, etc.) are present throughout the whole process of their development. Sooner or later, the links among those things become data relationships. How soon this happens depends on the software design approach. Moreover, not every piece of data of a software artifact can be traced back to real world things. Some are attributes of the relationship itself.

Some relationships are simply binary since they connect two real world things (cardinalities may be 1:1, 1: N , N : M). However, relationships may connect three or more real objects (then they are called n -ary relationships). When data and its relationships need to be persistent in any data repository, an obvious issue needs to be analyzed: how are they preserved? Some very old approaches put the linked data together to express the relationship. These approaches used to have many well-known disadvantages especially when the relationship cardinality was not 1:1. To overcome these disadvantages, duplication of data approaches were used in the past, too. For example, the data of a customer were usually attached to each of the bought products. This redundancy was also known as source of a new problem—consistency of data—since a customer might appear with two values for a given attribute due to data maintenance activities. This is a very

well-known problem of data redundancy and it is also one of the reasons that pushed towards the creation of the first database models.

A relationship whose cardinality is $N:M$ between entities or objects introduces new problems. First, there may be attributes belonging to the relationship itself and second, more than one link is needed.

The redundancy can be reduced or avoided, expressing the links among data in other ways. This implies the inclusion of a special attribute in the data not found in the real world but only used to represent the link. This attribute may be either a physical reference—telling where the related data is stored—or a logical reference—holding a key attribute that permits finding the related data. In both cases, the redundancy problem is replaced by a referential integrity problem.

To summarize, the technique used to store the data in the computer resources will create one or both of the following problems: Data Redundancy or Referential Integrity.

Since the early Hierarchical Databases, the driving idea has been "no duplicated data is allowed in the database." If the duplicated data is factorized, any other data previously attached to the factorized data know where the removed information is now placed. The way to know where such data is located is called a reference to the now missing data. In different database paradigms, this reference could be a physical pointer or a logical key reference, too.

No matter how it is referenced, the referred data should be available every time it is needed. But since the referred data is stored and processed independently from the referring data, the link may become lost. This is called the Referential Integrity problem.

Codd introduced the term *relational model* in his seminal research work (1970). In a subsequent article (1979), he presented the first published expression of entity integrity rules and referential integrity. Nowadays, the referential integrity problem is a concern of most relational commercial products that offer solutions to it.

In the object-oriented (OO) environment, objects are collected into classes and relationships are established at the class level. When these notions reach the implementation level, the problem is exactly the same. Relationships require references, which may be, again, physical or logical.

Business Rules

A business rule is an assertion that constrains or defines some aspect of the business. Every organization restrains behavior in some way. This is strongly associated with constraints that define which data may (or may not) be actualized. A business rule is a declarative sentence usually describing a correct state of a piece of data.

The process of identifying business rules is often iterative and heuristic, for rules begin as general organization statements of policy. Even if the policy is formal and specific, it is typically described in a general and informal fashion, and it often remains for the person responsible to translate it into meaningful specific statements of what to do. These statements are only sometimes originated in a given policy. More often, they arise from the day-to-day operation of the organization. These sentences are sometimes clear, sometimes (perhaps deliberately) ambiguous, and most of the time, contain more than one idea (Ross, 1997).

At the data processing level, the scope of business rules may be as small as the domain restrictions in some cases or as large as relationships in others. A business rule could be confused because of its appearance, with domain restrictions and with relationships. A business rule whose scope is only one attribute has, in fact, the same structure as a domain restriction, sharing all their properties. On the other hand, business rules involving more than one attribute belonging to the same or different entities or objects may be either similar to or rather different from the relationships.

However, business rules not always define allowed data states but allowed services connected with data states, for instance: "a given report cannot be produced if a specific data is missing". They may also establish a property involving data previously stored or new incoming data, for example: "employee salary could not be diminished."

Managing Data Properties

During the design phase of a software product, an important decision must be taken in relation with the properties of the data. They should be considered part of the data or be disconnected from the data and treated as Functional Requirements, included in what is called *Late Functional Requirements*.

From the early ages of the foundational programming languages, the properties of the available data types were considered part of them. The mechanism to enforce these properties was hidden from the user, giving a declarative flavor to them.

The programming tools have evolved for several decades, thus increasing their expressiveness by hiding more and more data properties under user defined layers. In other words, successful efforts have been made to let the developers handle some of those properties as part of the data itself.

However, the option remains open. There are still many data properties whose enforcement cannot be easily attached to the data itself. Open options always mean challenges. The challenge here is how to deal with complex data properties without turning them into Functional Requirements, burdening the developers with many data details that have to be taken into account every time they are handled.

This challenge is the core of this book. The following chapters are devoted to the study of how the Data Properties preservation problems are handled in the framework of the

different database paradigms. The problems presented in this chapter are not the only ones found in databases. Database paradigms offer solutions to previously unsolved problems but usually introduce new ones.

A tool developed in the framework of a given database paradigm is not the end of the story; on the contrary it is the beginning. A software artifact has to be developed using the tool and decisions about how to handle the preservation of data properties.

A well-conceived, implemented software artifact could be made using strategies that improve the paradigm framework and increase its semantic level. Obviously, this is not new in the computer science arena. It may be recalled that many developers have encapsulated data and procedures using only rigorous programming and file policies but only with languages without data abstraction capability. Before that, they were programmers creating well-structured programs using unstructured languages.

This book deals with the analysis of which directions to proceed, and how far. It is desirable that some of the ideas found in the next chapters become tips to improve the database paradigms, but most important, these ideas are intended to help readers deal with the design and implementation of complex data processing systems by using a database technique.

THE INTEGRITY PROBLEM IN DATABASES

One of the driving forces that stimulated the emergence of database technology was the need to guarantee the quality of the data stored. As was already mentioned, the data have properties; thus, quality of data means that all those properties are adequately represented and preserved. Efficient maintenance of data integrity has become a critical problem, since testing the validity of a large number of constraints in a large database and after each transaction is an expensive task.

Hierarchical and Network Databases

The data integrity problem in older databases only focused on relationships, which were implemented through physical links. The domain properties were considered by means of data types and business rules were completely ignored.

In network data models, referential integrity was supported through the set type construction. If member records are fixed to an owner, deletion of owner has a cascading effect. If member records need not be part of a set, the effect of deleting owner is equivalent to setting the relationship to null.

In hierarchical systems this issue was supported where a dependent child record type has total participation in the relationship with its parent record type. If a root of a tree or subtree is deleted, then so are all of its dependents (cascading actions).

Relational Databases

The term relational model is actually rather vague. It refers to a specific data model with relations as data structures, an algebra for specifying queries, and no mechanisms for expressing updates or constraints. Subsequent articles by Codd introduced the first integrity constraints for this model—namely, functional dependencies. Researchers in database theory developed a number of variations on Codd's original model, to gain a higher expressive power. This evolution was accompanied by the evolution of the integrity constraints. A rich theory for constraints has emerged, based mainly on a fundamental class of constraints called dependencies. Its main motivation is to incorporate more semantics into the relational model. The term relational model has thus come to refer to the broad class of database models that have relations as the data structure and that incorporate some or all of the query capabilities, update capabilities and integrity constraints (Abiteboul et al., 1995).

Among many other innovations, from the integrity point of view the main change introduced by the earlier relational database models has been the use of logical references or foreign keys instead of physical references. Regarding the domain properties, very few things have changed from previous models. Business rules have recently started to be considered.

Due to the misunderstanding of Codd's relational concepts by some relational vendors, the author published a textbook in 1990 that highlights the characteristics a relational system must have. In the integrity field, he introduced five types of integrity restrictions: Entity type, Referential type, Column type, Domain type and User-defined type. Entity integrity and referential integrity apply to the base relations in every relational database. Entity integrity establishes that no component in a primary key is allowed to have null values. Referential integrity points that for each distinct foreign key value, an equal value of a primary key from the same domain must exist in the database. Otherwise, the foreign key value must be null. Domain and column constraints refer to the allowed values for a given attribute or set of attributes. Integrity constraints other than those mentioned are needed for relational databases. These constraints, named user-defined integrity constraints, permit the DBA to define, in a way that can be enforced by the DBMS, many of the company regulations pertaining to the company operations (internal), and many of the government and other external regulations. Once these constraints are defined and included in the catalog, the DBMS should enforce them. Consequently, there should not be the need to depend on voluntary compliance of application programmers or end users. All integrity constraints are applied not only to keep the database in an accurate state by preventing violations of these constraints, but also to trigger specified repairing actions when specified conditions arise in the database.

The observance of Codd's principles has been evolving positively throughout the last decade. On the other hand, many ideas of other database paradigms already in the scenario (and in their own evolution process) have been borrowed by researchers and vendors, and introduced in current relational products. This fact created incomplete and extended relational engines that started to be known as postrelational databases. They are incomplete because not all the prescribed issues for the relational model are satisfied; and extended because they go further in some other areas such as active characteristics. This

mix is somehow explosive since users enforce a non-supported integrity constraint by means of provided extensions in a rather unstructured way. Triggers are one of the mechanisms that are overused.

Hazardousness is not contradictory with usefulness. Current DBMS may be used in a very productive way if the risk of using some of their extensions is understood. When a restriction cannot be expressed declaratively, triggers are very useful tools to support data integrity in a database. In actual commercial relational products implementing the SQL-92 standard, integrity constraints can alternatively be expressed as triggers, which also allow the definition of policies to repair violations. However, declarative constraints are generally preferable to explicit triggers because they are easier to manage (Rivero, Doorn & Ferragline, 2000).

The relational model has provided an excellent framework for theoretical research into fundamental aspects of data manipulation and integrity constraints. This research provides a strong foundation for the study of other models (Abiteboul, Hull & Vianu, 1995).

Dolores Cuadra et al. devote [Chapter 3](#) to the study of the transformation of conceptual schematas into logical ones in a methodological framework, focusing on a special Entity Relationship construct: the relationship and its associated cardinality constraints. The authors remind us that, concerning the logical design, the transformation process of conceptual schemata into relational schemata should be performed with an effort made to completely preserve the semantics included in the conceptual schema. Even though the final objective is to keep the semantics in the database itself and not in the applications accessing the database, sometimes a certain loss of semantics is produced, for instance, foreign key and not null options in the relational model are not enough to control ER cardinality constraints. In their chapter entitled "[*Preserving Relationship Cardinality Constraints in Relational Schemata*](#)," the authors review the relationship and cardinality constraint constructs through different methodological approaches; they analyze the transformation of conceptual n-ary relationships into the relational model following an active rules approach, and provide several practical implications as well as future research paths.

Active Databases

Supporting reactive behavior implies that a database management system may be viewed from a production rule system perspective. These production rules are well known today, in database terminology, as active rules or simply triggers.

Active rules provide reactive behavior. It is a form of computation, which is motivated by the occurrence of some event, typically a database operation, executing a reaction to that stimulus. Active rules may pose queries to the database to collect information about events and database objects and decide whether events require an action; then they may execute actions, normally any sequence of database operations (Ceri et al., 1997).

Reactive behavior is seen as an interesting and practical way to check for satisfaction and enforcement of integrity constraints. Nevertheless, integrity constraint maintenance, materialized view maintenance (especially useful in the warehousing area) and implementation of business rules are not the only areas of application of data repositories with reactive behavior. Other interesting application areas are replication of data for audit purpose, data sampling, workflow processing, scheduling, real time applications, and many others. In fact, practically all products offered today in the marketplace support complex reactive behavior on the client side.

Undesired behavioral characteristics have been observed related to production rule systems, however. For example, termination is not always guaranteed, non-determinism could be expected in the results, confluence with respect to a desired goal could be not achieved. Since triggers and declarative integrity constraint definitions may appear intermingled in a concrete application, an integrating model is needed to soften to some extent the effects of this undesirable behavior, ensuring that no matter what the nature of the rules involved is, integrity is always preserved.

Active rules and integrity constraints are related topics. Systems do not support both completely, but partially, in their kernels (see [Chapter II](#)). When a constraint must be enforced on data, if such constraint cannot be declared, it may be implemented by defining triggers. Studying the relationships between constraints and triggers from this point of view is therefore mandatory. In simple words, methods are needed to check and enforce constraints by means of triggers.

From a user point of view, reactivity is a concept related with object state evolution over time. Dynamic constraints, i.e., constraints making assertions on the evolution of object states, may be needed to control changes in the state of data objects. Dynamic constraints are mandatory in the correct design of applications, particularly for workflow processing and for the Web. Actual products support some kind of functionality in this area, allowing triggers to refer to transitions when an atomic modification operation is executed. Supporting such type of constraints by means of handcrafted triggers written by a novice, without any method in mind, may be potentially dangerous from the perspective of correctness. Formal methods guaranteeing correctness are needed for good deployment of such triggers.

Juan M. Ale and Mauricio Minuto Espil entitled their contribution: "[*Integrity Constraints in an Active Database Environment*](#)." In [Chapter IV](#) the authors survey the interaction between active rules and integrity constraints. First, they analyze the static case following the SQL-1999 Standard Committee point of view. Then, they consider the case of dynamic constraints using temporal logic formalism. This chapter also includes a comprehensive discussion of the applicability, limitations and partial solutions found when attempting to ensure the satisfaction of dynamic constraints.

Spatial Databases

Even though there is a very active research area interested in the design of robust and efficient spatial databases, it is still evident that the inability of current GIS regarding the implementation and management of spatial integrity constraints occur because of the scope of geographic applications, and special problems come up due to the locational aspects of data. A modification in a spatial database may cause simultaneous updates in a large number of records in multiple files, making it hard to manage all the environment. A very sophisticated control is required to avoid redundancy and loss of integrity. In [Chapter V](#) entitled “*Integrity Constraints in Spatial Databases*”, Karla A. V. Borges, Clodoveu A. Davis Jr., and Alberto H. F. Laender address the relationship existing between the nature of spatial information, spatial relationships, and spatial integrity constraints, and propose the use of OMT-G, an object-oriented data model for geographic applications, at an early stage in the specification of integrity constraints in spatial databases. OMT-G provides appropriate primitives for representing spatial data, supports spatial relationships and allows the specification of spatial integrity rules (topological, semantic and user integrity rules) through its spatial primitives and spatial relationship constructs. Once constraints are explicitly documented in the conceptual modeling phase, and methods to enforce the spatial integrity constraints are defined, the spatial database management system and the application must implement such constraints.

Since [Chapter V](#) does not cover integrity constraints associated with the representation of simple objects, such as constraints implicit to the geometric description of a polygon, the authors provide relevant references to research related to consistency rules associated with the representation of spatial objects.

Object-Relational Databases

Object-oriented literature typically uses the term "relationship" to mean, specifically, relationships supported by foreign keys in a relational system. ORDBMSs may use this SQL-92 oriented implementation. On the other hand, OR systems, like Illustra, provide the *references* as a natural substitute for primary key-foreign key relationships found in traditional SQL systems. These systems allow a column in a table to contain a value that is a reference to an instance of a composite type stored in another table. Conceptually, this data type is a pointer to a record of a specific type in the table. This implementation is supported by the unique object identifier (OID), which all rows in a table have. In this case, each value stored in the referencing column is an OID. To summarize, in order to support referential integrity you can use typical primary key-foreign key relationships (logical pointer) as well as a pointer implementation (reference via an OID). Whenever available, the second option is preferable, because an OID is guaranteed to be unique and never changed, while the foreign key value is not necessarily time invariant. SQL was extended with capabilities such as a function for the recovery of referenced objects, and support for other functions over the type reference engine (Stonebraker, 1996).

Various levels of support for referential integrity have been implemented in those systems. Some do not support referential integrity, leaving it as the responsibility of the user-written code. Other systems check that all references are to existing objects of the

right type (taking special care of avoiding dangling references) or keep all references up to date automatically (Connolly et al., 1999).

As in the relational paradigm, triggers are very effective in supporting data integrity in a database, especially to deal with those restrictions that cannot be expressed declaratively. ORDBMSs and OODBMSs demand a system of triggers even more flexible than the relational one.

In Chapter VI "[*Consistent Queries over Databases with Integrity Constraints*](#)," Sergio Greco and Ester Zupanó point out that even though integrity constraints are usually used to define constraints on data, nowadays they have a wide applicability in several contexts such as semantic query optimization, cooperative query answering, database integration and view updates. In this chapter, the authors illustrate recent techniques for computing consistent answers and repairs for possibly inconsistent databases. They present some preliminaries on relational databases, disjunctive deductive databases and integrity constraints and then they introduce the formal definition of repair, consistent answer and the different techniques for querying and repairing inconsistent databases, including an extension of relational algebra; the integrated relational calculus which extends relations and algebra for querying inconsistent data; techniques for merging relations based on majority criteria, for querying and repairing inconsistent data based on the concept of residuals, for querying inconsistent databases based on the definition of a logic program for defining possible repairs and a technique based on the rewriting of integrity constraints into disjunctive Datalog rules.

Hendrik Decker is the author of Chapter VII entitled "[*Translating Advanced Integrity Checking Technology to SQL Databases*](#)." The main goal of this chapter is to arrive at a coherent technology for deriving efficient SQL triggers from declarative specifications of arbitrary integrity constraints. The user may specify integrity constraints declaratively in some manner, then the triggers derived from such specifications will behave such that whenever some update event violates any of the integrity constraints, one or several of the triggers derived from that constraint are activated to enforce the constraint. In this chapter the author describes how to implement advanced *datalog* technology for integrity checking in the framework of SQL, showing how to represent and evaluate arbitrarily complex constraints in SQL without incurring major disadvantages usually associated with integrity checking in knowledge-rich applications: error-prone procedural specification and laborious maintenance of integrity constraints is avoided and the cost of evaluation is considerably reduced by an automated translation of declarative specifications to SQL triggers.

Object-Oriented Databases

Given that in an OO environment, objects are collected into classes, relationships also are established at the class level; a relationship between classes denotes a set of relationships between the objects of the respective classes. Relationships may have attributes. A model is always a compromise to achieve the right amount of expressive power while keeping

simplicity and clarity. Domain restrictions—keys and referential integrity constraints—may be used straightforwardly in OODBMSs.

Other kinds of constraints are peculiar to OODBMSs, for instance: constraints of the migration of objects between classes (roles); exclusivity constraints between classes; constraints on the definitions of subclasses; and existence dependencies. The last ones are the key to semantic integrity checking since they allow the designers to track and solve inconsistencies in an object-oriented conceptual schema (Snoeck & Dedene, 1998).

Some currently available OO systems do not provide mechanisms for the definition, management and control of integrity constraints. In these systems, it is possible to embed integrity constraints as part of the update methods associated with the objects. The introduction of a language for defining the constraints would facilitate such constraints to be defined in a declarative rather than procedural way (Bertino & Martino, 1993).

Chapter VIII by Jochen Rasch and Hans-Joachim Klein entitled "[*Functional Dependencies for Value-Based Identification in Object-Oriented Databases*](#)," focuses on the generalization of the well-known functional dependencies to object schemas. The authors offer insights on one of the fundamental concepts of the object-oriented approach: object identity. They first introduce some basic notions of the object model, including a formalization of the terms object schema and schema graph, as well as some concepts of the relational data model. Then they describe an approach to generalize functional dependencies to object functional dependencies. The chapter presents different semantics for object functional dependencies, based on a relational representation of relevant parts of states. The proposal introduced by the authors is compared to related approaches and some interesting challenges for future research are pointed out in the conclusions.

Chapter IX "[*Integrity Maintenance in Extensible Data Bases*](#)" by Ulrich Schiel, analyzes the problem of the increasing complexity in the data models required by new applications holding, for example, complex structured objects, multimedia data, and unstructured documents, each with their own semantic complexity. To allow an easy way to define the semantics of such complex data models, the author introduces an approach to define it by means of general (schema) integrity constraints integrated to the system as rules.

Distributed Databases

A distributed database managing system (DDBMS) is a database management system (DBMS) that supports characteristics of a distributed database, that is, the possibility of handling information contained in multiple locations, preserving data integrity at the same time. A DDB differs from a centralized DB mainly in that data are placed at a number of locations instead of being located in only one site. This characteristic of DDBs causes the control of data integrity to be more complex than for centralized environments. Each transaction can involve more than one location, and it is hard to keep an execution order of the instructions of the transaction to preserve data integrity. The information integrity problems discussed for a centralized DB also appear for a DDB, together with a

series of new issues. The studies carried out for transactions in centralized environments can be used as a starting point to solve the problems in distributed environments.

Some authors define a DDB as a DB that uses the same DBMS at each location of the net. Other state that when there is a distributed system using heterogeneous DBMSs in the network, it is called Heterogeneous Data Base (HDB). In general, data integrity questions are dealt with in similar ways both for DDBs and HDBs, DBMS interoperability allows for solving integrity problems.

At first glance, the Web is a huge repository of information without any structure whatsoever. Nowadays, this is changing quickly. The Web presents the highest degree of distribution, heterogeneity, and autonomy, and therefore, traditional distributed database techniques must be further extended to deal with this new environment. Chapter IX, "[*Integrity Issues in The Web: Beyond Distributed Databases*](#)" by J.F. Aldana Montes, M.I. Yagüe del Valle and A.C. Gómez Lora, focuses on introducing a new paradigm: The Web as the database, and its implications regarding integrity, i.e., the progressive adaptation of database techniques to Web usage. The authors study the different issues related to integrity and its maintenance on the Web and introduce the reader to other related and open issues, such as the query problem and query optimization on the Web.

REFERENCES

- Abiteboul, S., Hull, H., & Vianu V. (1995). *Foundations on databases*. Addison Wesley Publ. Co.
- Bertino, E. & Martino, L. (1993). *Object-oriented database systems. Concepts and architectures*. Addison Wesley.
- Ceri, S. & Fraternali, P. (1997), *Designing database applications with objects and rules: The IDEA methodology*, Addison Wesley.
- Codd, E. (1970). *A Relational Model of Data for Large Shared Data Banks*. *CACM*. 13(6): 377–387.
- Codd, E. (1979). *Extending the Data Base Relational Model to Capture More Meaning*. *ACM Trans. On Database Systems*, 4(4): 397–434.
- Codd, E. (1990) *The relational model for database management*. Version 2. Addison Wesley Publ. Co.
- Connolly, T., Begg, C. & Strachan, A. (1999) *Database systems: A practical approach to design, implementation and management*. 2nd. Edition. Addison Wesley.
- Jackson, M. (1995). *Software Requirements & Specifications. A Lexicon of Practice, principles and prejudices*. Addison Wesley, ACM Press.
- Karlsson, J. (1996). *Software Requirements Prioritizing. ICRE'96 Second International Conference on Requirements Engineering. IEEE CSP*. Los Alamitos, CA. Proceedings.
- Loucopoulos, P & Karakostas. (1995). *System Requirements Engineering*. McGraw Hill International Series in Software Engineering.
- Rivero, L., Doorn, J. & Ferraggine, V. (2000). *Inclusion Dependencies. In Developing Quality Complex Database Systems: Practices, Techniques, and Technologies*. Hershey: Idea Group Publishing.

Ross, Ronald. G. (1997). *The business rule book. Classifying, defining and modeling rules*". Database Research Group, R. Ross Editor/Publisher.
Snoeck, M. & Dedene, G. (1998). *Existence dependency: The key to semantic integrity between structural and behavioural aspects of object types. IEEE Trans. On Software Engineering*. 24 (4). April 1998.
Stonebraker, M. (1996). *Object-relational DBMSs. The next great wave*. Morgan Kauffman Publishers.

Chapter II: Database Integrity— Fundamentals and Current Implementations

**Laura C. Rivero, Universidad Nacional del Centro de la Provincia de Buenos Aires
and Universidad Nacional de La Plata,**

Argentina

**Jorge H. Doorn, Viviana E. Ferraggine, Universidad Nacional del Centro de la
Provincia de Buenos Aires,**

Argentina

In [Part I](#), this chapter surveys the state of the art of the semantic integrity constraints in some relational and object relational available database systems. In [Part II](#), it also provides an overview of the SQL standard integrity issues and describes semantic integrity support in the following DBMSs: Oracle, IBM DB2, Informix, Sybase and PostgreSQL.

The major differences and similarities among these systems are analyzed in relation to the definition, semantics and fidelity to the SQL standard prescriptions.

PART I: INTRODUCTION

This chapter is devoted to expanding the concepts presented in [Chapter I](#). One of the most important current trends in database management is the increase of the semantic content of stored data. In this way, the first step in the establishment of the database theory is the precise definition of data models, since without it the database concepts cannot be understood as regards the design, analysis and implementation of schemas, transactions, and databases (Thalheim, 1996).

Taking into account that a database is a resource shared by many applications, it is advisable to register any knowledge about data semantics in the database in such a way that there is no need to replicate it into the applications using such knowledge. This knowledge covers a large variety of fields of the Universe of Discourse (UoD) "under

the form" of rules, which can be grouped in the following families: rules about the valid values of particular items of data; rules describing the way the data are associated with one another (interdata connections); and rules about the actions that should be performed when a specific event shows up (business or enterprise rules). Generally, the first two kinds of rules are included under the denomination "[integrity constraints](#)." However, the concept of rules is preferable since, in general, the distinction between constraints and business rules is not clear. From an operational point of view, all rules can be treated as active requirements since the system must verify that user manipulations leave the data in an allowed state. If the execution of a proposed transaction leads to a constraint violation, the system either aborts the transaction or executes repairing actions, clearly revealing a reactive nature.

In the database world, rules are relevant concepts to describe a piece of active requirements. Rules define the intended structural and behavioral properties of objects involved in a database application, and they can be specified in several ways. At procedural and production levels, rules clearly exhibit a reactive structure. At the conceptual level, some rules already have an active form while some others do not, but all of them involve active requirements (Van den Berghe, 1999).

When the database engine automatically enforces rules like these, stored data become more "active," thus acquiring a richer level of semantic content (Chamberlin, 1998). In other words, database constraints can be regarded as a language to specify the semantics of data.

Most database systems provide some support for integrity constraints. For example, current commercial database systems (especially RDBMSs) enforce only a small set of constraints, mainly because of the performance overhead associated with update operations. In this manner, in RDBMSs and ORDBMSs some restrictions related to the valid values of a particular column (typing constraints) can be directly represented at schema definition time using the facilities the language (usually SQL) offers for the data definition (DDL). Others are expressed and enforced by mechanisms such as check conditions, assertions and triggers in RDBMSs or specific methods in OODBMSs. The best approach to implement semantic integrity constraints requires a formal specification method to define assertions and a set of enforcement algorithms to guarantee database consistency relative to these assertions.

With respect to the rules related to valid values and data associations, it should be pointed out that since there are many different restrictions over data, many different classes of constraints are generated. The different database paradigms—relational, object relational or others—were not conceived with the integrity vision as the primary objective, then they have a weak semantic approach to this subject. In the specific field of RDBMSs a database can be viewed as a collection of tuples. Tuples are a very poor media to express semantic qualities so additional semantic features must be specified in another way. The specification of such features depends on the choice of a DBMS, being the level of support of current relational products uneven from system to system.

On the other hand, updates to data items may also be constrained by business rules governing the real world changes. These changes are represented by updates in the database world. Some DBMSs provide more facilities than others do for defining enterprise constraints. In most systems, there is no support for some or all of the enterprise constraints and it will be necessary to include the constraints into the applications (Connolly, Begg & Strachan, 1999) or specific purpose programs.

This chapter is devoted to examining the state of the art of the semantic integrity constraints in some (object) relational-available DBMSs, also provides an overview of the SQL standard integrity issues and a comparison of the semantic integrity support in Oracle, IBM DB2, Informix, Sybase and PostgreSQL. The main differences and matches among these systems are analyzed in relation with the definition, semantics, and fidelity to the SQL standard prescriptions.

This chapter has been structured as follows: In the section "[Integrity Constraints](#)" a constraint classification is presented. In "[The SQL Standard Facilities](#)" section, the diverse integrity features proposed in the SQL-99 standard are discussed in detail. [Part II](#) begins with the presentation of the reference systems, and the description of a motivating example. The review of integrity issues in mentioned reference systems is developed in the section "[Integrity Constraints in Current Database Management Systems](#)." Finally, this chapter ends with some concluding remarks.

INTEGRITY CONSTRAINTS

During the conceptual modeling phase, the designer captures and describes both the relevant actors and resources playing in the UofD and the semantic links among them, producing a connected network of object and relationship types. These types are defined by their components (attributes) and assertions on the valid values and their behavior. By enriching the conceptual schema with a complete set of such assertions, which should be enforced dynamically and continuously, the database designer depicts consistent states at design time (Codd, 1990). Integrity enforcement efficiency is influenced by the complexity of the assertion set, by the structure of the database repository and by the device that controls and drives database actualizations. Semantic data control ensures the maintenance of database consistency by rejecting update transactions that lead to inconsistent states or by activating specific actions on the database state to compensate the effect of the previous transaction. In this context, the task to ensure the fulfillment of the integrity requirements is a well-known problem and target of current research. Since most of the current relational DBMS systems fail to provide adequate support for the integrity maintenance, this activity becomes a DBA programmer's responsibility.

As was briefly introduced in [Chapter I](#) the data properties are seen from a point of view as much independent as possible from the representation model. In this way, data properties can be analyzed from two different perspectives: from the real world and from the database world. A real-world data property of a given class may be mapped into a different database class of properties due to materialization issues. Doing so depends on the database paradigm and on the adhesion of the DBMS to that paradigm.

According to this approach, when a data property describes the allowed values of an attribute or a set of attributes in the real world, it is called *Domain Restriction* in the database world. The semantic connections among objects and among the properties of a given object are called *Relationships* in the real world and they become a different kind of link in the database world. The restrictions over these links and the links themselves are materialized in the database world by means of restrictions over attributes pertaining to the same or different objects.

The identification of actors or things using a unique inherent or an artificial attribute (surrogate key) is a need of the real world. Since data processing emphasizes this need, in a database context, this necessity leads to the selection of primary keys.

Finally, when a data property carries out a semantic that is specific of the UofD, modeling the reaction to events or stimuli generated in the real world, it is called *Business Rule*. Some business rules may be expressed as domain restrictions and some others look like relationships but most of them must be expressed in a more complex way (Ceri et al., 1997; Codd, 1990; Ross, 1997). At data processing level, the scope of business rules may be as limited as domain restrictions or as extensive as relationships; in other cases they are completely different. A business rule whose scope is only one attribute has, in fact, the same structure as a domain restriction and it shares all their properties. On the other hand, business rules involving more than one attribute belonging to the same or different entities or objects may be either similar to or rather different from the relationships.

[Table 1](#) shows a general and succinct definition of the restrictions according to both points of view and [Figure 1](#) depicts the mapping between both worlds. Boldfaced arrows represent common situations and narrow ones stand for not so frequent mappings. The borders among the kinds of rules in the database world depend on the context in which the assertion is made. In such way, for instance, some constraint having the appearance of a SQL domain restriction could be materialized as a general restriction in an actual engine.

Table 1: Real world restrictions and their correlates in the database world			
REAL WORLD RULES		DATABASE WORLD RULES	
NAME	DEFINITION	DEFINITION	NAME
DOMAIN	Allowed values for data items	Attribute values constrained by basic types or specializations of them	DOMAIN RESTRICTION
RELATIONSHIP	Semantic connections among real things (inter-object) and/or among the identifier/ descriptor data items of real things (intra-object)	Inclusion dependencies, referential integrity constraints and functional dependencies	OBJECT RESTRICTION

Table 1: Real world restrictions and their correlates in the database world			
<i>REAL WORLD RULES</i>		<i>DATABASE WORLD RULES</i>	
<i>NAME</i>	<i>DEFINITION</i>	<i>DEFINITION</i>	<i>NAME</i>
<i>BUSINESS RULE</i>	Specific semantic characteristics of the UofD	Complex assertions combining columns from arbitrary combinations of base tables	<i>GENERAL RESTRICTION</i>

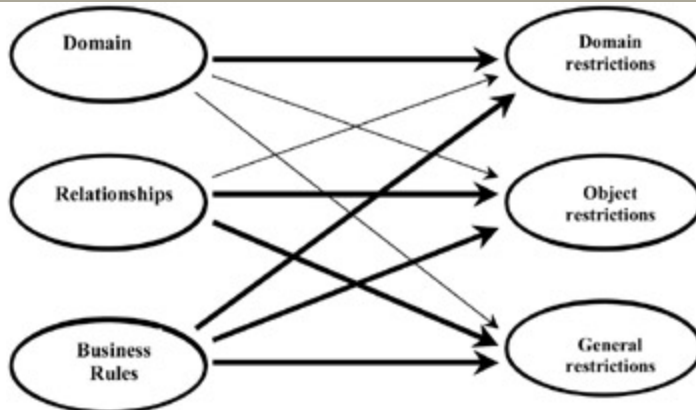


Figure 1: Restriction Mapping

Obviously, there are semantic gaps between the real world and the database world columns of [Table 1](#). In the real world a domain may be—and frequently is—compound such as it happens with: the employee address (composed by the number, street name, city name and zip code); the room number in a hotel (composed of the floor number and the room number), etc. In the database world, these domains may be specified as a unit or through their components. Only domains defined as a unit may become Domain Restrictions in the database world. More complex approaches must be used to preserve database compound domains. For example, not all possible combinations of numbers and streets represent valid addresses of a given city, but usually the specification of the allowed values is hard to define unless the components are specified separately. As a counterexample, the hotel room domain can be precisely specified when all floors have the same number of rooms.

The gap between Relationships and Object Restrictions is less evident since two different problems of the real world are enclosed in a single notion: connections among different real objects and connections among data items of a single object. On the other hand, Object Restrictions naturally fulfill both approaches.

Finally, Business Rules are an open family of restrictions while General Restrictions collect all issues not supported by the previous two kinds.

Examples:

- i. Consider a medical center. In this context, a Patient can be identified via his/her Patient Id (PI) or via his/her Social Security Number (SSN). Suppose a business rule of this information system which requires that each Patient must have a SSN or a PI or both. In a real system, a trigger such as the one depicted in section "Integrity Constraints in Current Commercial Products" below implements this constraint. This example corresponds to the arrow
- ii. In the same context, suppose a Patient can ask for, at most, three Medical Services. This is a property of the relationship between the patient and the Medical Service entities but, due to limitations in the implementation of these facilities, in current systems this restriction must be expressed as a trigger or via a piece of code in an application program
- iii. A relationship among the identifier and descriptive attributes represents a functional dependency, which is specified via a PRIMARY KEY clause
- iv. Finally, suppose a domain defined over a set of prime numbers less than 10,000. This domain property can be specified in three ways: via a Domain Restriction, enumerating all possible instances an attribute may have (1); via an inclusion dependency if the set of prime numbers is materialized as a one-column table (2); or via a coded generating algorithm (3).

Domain Restrictions

As was already defined, a domain restriction defines the set of values that an attribute may have. These sets may be defined by enumeration of the members or by intension and they are associated with a specifically-defined domain. And in relational systems, they apply to every column in every base table that is defined over that domain. Examples of this class are null and default restrictions, value restrictions, enumerated and scope restrictions.

Within the SQL2 standard, domain restrictions are unfortunately considered as a basic type with additional restrictions specified by extension or by intension. This is a version, limited in one sense and extended in another, of the user defined data types in programming languages. It is limited because it can only use a basic type and no other built types, such as Cartesian product. On the other hand, it is an extended version because all possible values are restricted in a more flexible and expressive way than in those cases. SQL3 overcomes this limitation.

Domain definitions are useful when several tables contain identical column definitions. In this way, the domain is defined just once and is used wherever it is needed. Definition of domains must be carefully specified to avoid contradictory constraints. To specify a column domain, the definition of such domain must be provided. A new problem then arises: the integrity of the integrity rule set. This is usually called the metaintegrity problem (Zaniolo et al., 1997). Every property of the integrity rule set applies to the metaintegrity rules. Their unique characteristic is that metaintegrity rule objects are integrity rules. Available commercial products present an insufficient coverage of this area.

When these restrictions are not generalized as domains, i.e., when they are associated with a specific column in a specific table, they are usually called *column constraints* (Codd, 1990).

In SQL, domain restrictions can be expressed in the specification of a table (CREATE TABLE sentence) or via a domain definition (CREATE DOMAIN sentence). Additionally, domain definitions can be altered or dropped through the clauses ALTER DOMAIN and DROP DOMAIN.

The syntax of these clauses can be found in the section "[The SQL Standard Facilities](#)" and examples are provided in the section "Integrity Constraints in Current Commercial Products".

Object Restrictions

As was previously mentioned, relationship constraints allow the characterization of intra- and inter-object relationships. In the relational context, the most relevant relationships, which connect attributes that describe and/or identify the entities, are the functional dependencies. They may be tagged as primary or secondary according to the structure of their left term. When the left term is the primary key of the relation, the dependency is a primary one.

In other cases, they are secondary dependencies. This type of dependency is not currently supported.

On the other hand, relationships between real-world things (actors and resources) also become relationships in the database world, having a scope larger than domain

restrictions since they involve several attributes, at least two, usually belonging to different objects or entities.

The manner that these connections appear in the database world depends on the way the objects have been represented. In a relational context, relations are used to model the real world, i.e., entities, their descriptions and the relationships among entities. A database designed "strictly" adhering to a methodology only produces relationships between properly designed classes of entities. On the contrary, ad-hoc refinements of the logical schema without concern for the corresponding conceptual design usually lead to the modeling of another kind of relationships. As in this case the entities were not properly designed, i.e., the schema holds hidden entities, the relationships among them and other objects are also misrepresented. A hidden entity is one that has not been made explicit as a relation in the schema, but it conceptually exists in the real world.

It should be noticed that even though relationships are symmetric, a designator and a designated relationship term could be distinguished. These components are usually named left and right-hand side of the relationship, respectively. When the right side is properly modeled, the real-world relationship is represented as a key-based inclusion dependency (usually named referential integrity restrictions) in the database world. In this case, the attribute or a set of attributes—which materializes the reference and pertains to the left term—is named the foreign key and the connection is based either on the primary key or on an alternate key of the right side table. On the other hand if the right side is a hidden entity, a non-key (pure) inclusion dependency represents the relationship, and its terms have no special names.

Systems adhering to the SQL standard allow the specification of referential integrity using the FOREIGN KEY clause. Non-key inclusion dependencies are almost completely disregarded by actual systems, obliging the users to manage them via special-case code or triggers. SQL offers an extension of the well-known FOREIGN KEY clause adding PARTIAL and FULL MATCH options. These concepts are detailed in the section '[The SQL Standard Facilities](#)'.

Most relationships are simply binary since they connect two real world things (cardinalities may be 1:1, 1:N, N:M). However, relationships may connect three or more real objects (then they are called n-ary relationships). In this case, they may be converted into a set of binary relationships.

A main issue strongly related to the relationships is the "referential action." The referential action is formed by the set of operations that is necessary to perform in order to maintain the relationships in a proper way. In other words, if a designated object is deleted or changed, what actions are to be performed in order to preserve the database integrity? One option may be: delete (change) all related objects, and the objects related to these ones and so on. Other options are: do not permit the deletion (update) if there are objects designing it; destroy the link between the objects nullifying the reference in the designator; or label the link as invalid replacing it by a default value. These options are named cascade, restricted (no action), set null and set default, respectively. They are fully

described in the following section and are the standard actions. However, some authors have proposed another kind of actions (forgive, label as an exception, etc.) that can be implemented in combination with the standard ones and with prompts to allow the users to execute specific actions (Etzion, 1993). Current systems partially support only standard actions.

General Restrictions

A business rule constrains or defines some aspect of the business. Their name proceeds from the fact that they perform part of the business management, modeling the reaction to events which occur in the real world with tangible side effects on the database state (Ceri & Fraternali, 1997). However, business rules sometimes do not define allowed data states but allowed actions connected with data states ("a given report cannot be produced if a specific data is missing") and/or establish a property involving data previously stored or new incoming data ("employee salary could not be diminished").

Even though SQL provides the CHECK and the CREATE ASSERTION clauses, they are usually insufficient to represent the richness of real world rules. For this reason, current systems offer an additional facility: procedures that are actively invoked under update operations. These procedures are called triggers.

Because of the trigger reactive behavior, the system reaction is not limited to the typical rollback of the offending transaction (abort rules). Looked from a higher level of abstraction, integrity restrictions can be specified within the context of a maintenance integrity policy. In this context, the action to be performed over a database state if a constraint is violated is specified and thus the rules become repairing.

Triggers are procedures that are implicitly invoked under the occurrence of certain pre-established events, generally data updates of a specific table. They are the most-used tool to materialize business rules and their needed reactive behavior.

Considered as active rules, triggers should materialize all the concepts those rules exhibit. Active rules fit in the Event-Condition-Action paradigm. In this way, the *consideration* of a trigger—the concept relative to the event that activates it—can be immediate, deferred, or detached. *Immediate consideration* can occur BEFORE the activating event, AFTER the event, or INSTEAD OF the event. *Deferred consideration* can occur, for instance, at the end of a transaction or after user-defined commands. Finally, *detached consideration* happens in the context of a separate transaction^[1] *Execution* of the action is the concept relative to the condition consideration. It can be immediate, deferred, or detached as well. *Immediate execution* implies the action execution to immediately follow the condition consideration; it is the most used option. *Deferred execution* postpones the executions of the actions until the end of the transaction, and finally, *detached execution* happens in the context of a separate transaction after the rule is considered.

In the following sections, additional features for the commercial product triggers implementation, and several examples are provided.

As a concluding remark of this section note that, given the importance of standard compliance, all vendors have tried to produce systems as close as possible to the preliminary standard document, disregarding some of its most exotic features, but documents left a number of open issues, which have been "closed" by vendors in different ways.

[1] Naturally, this issue concerns concepts such as isolation levels, concurrency control and others related to them. They constitute another perspective of the database integrity problem.

THE SQL STANDARD FACILITIES

The following sections have been framed taking into account the SQL-92 standard and preliminary documents of SQL3.

Lately the main commercial database engines have adhered—in larger or smaller degrees—to the standard of SQL known in the literature as SQL2 or SQL-92 (Date & Darwen, 1997) and throughout the last years, to some characteristics published in the preliminary and the final document of SQL3 standard, also known as SQL-99 (SQL99-1, 1999; SQL99-2, 1999). What is called SQL in this chapter is at least SQL2 and when needed, additional characteristics from SQL3 are explicitly included.

These standards have been divided into three levels: Full SQL, Intermediate SQL, and Entry SQL. The former is the complete standard, the second is a subset of it, and the latter is a subset of the second level. Current products implement Entry SQL facilities together with some extra characteristics from the second and/or the third level. Since integrity restrictions have a precise format whose complete specification is provided in the full level, whenever one of such restrictions is needed, it must be specified in an ad-hoc way according to the features each DBMS provides.

In this context, an integrity constraint is seen as a conditional expression required to evaluate TRUE. SQL provides a broad variety of methods to implement integrity constraints. The overall restriction specified in SQL for a particular database can be interpreted as the logical conjunction (AND) of all particular restrictions.

Within the standard guidelines, when a user tries to incorporate a new restriction, the database state should be checked in order to verify if it satisfies this restriction. In case the new restriction is violated, it should not be included in the database catalog. All restrictions have a name given by the user or automatically provided by the system. These names are important in two senses. On one hand, they help the user to perform the database application debugging, to disable or drop an integrity check; or to find a constraint to change the checking model. On the other hand, they are essential for the DBMS for the identification and management of the schema components into the metadatabase. These characteristics are explained in the corresponding sections.

Each conceptual schema developed under the SQL standard prescriptions contains a set of definitions related to the relevant concepts of the UoFD. In this way, domains, base tables, views, restrictions, privileges, and any other object that can be represented in SQL have their respective definitions (Date & Darwen, 1997).^{[2][3]}

Domain Restrictions

As was previously expressed, domains can be specified via a declarative clause or inside the definition of the columns of a table. The SQL clause used to define domains is **CREATE DOMAIN**. The basic **CREATE DOMAIN** clause must specify the domain name, the basic data type it constrains and optionally the default value an instance of this domain can accept and the restriction on the values the domain can contain. The syntax is:

```
CREATE DOMAIN domain-name [ AS ] data-type
[DEFAULT default-option] [CHECK (search-condition)]
```

Default-option and *search-condition* are detailed below.

If the domain definition needs to be modified, the **ALTER DOMAIN** clause should be used; whenever a domain definition is no longer necessary, the proper sentence is **DROP DOMAIN**.

```
ALTER DOMAIN <domain-name> <domain-alteration-action>
DROP DOMAIN <domain-name> { RESTRICT | CASCADE }
```

The *domain-alteration-action* may be the modification of the default value of a domain, the addition of a *column-constraint-definition* or the elimination of a constraint over that domain.

At this point, a metaintegrity subject arises. Since the domain definitions and the table definitions are related, the alteration or dropping of a domain definition produces the propagation of this action over the related components. When **RESTRICT** is specified, the delete operation succeeds if, and only if, it is not referenced in any column definition in any table, view or integrity restriction definition. If **CASCADE** is specified, **DROP DOMAIN** always succeeds since all references from column tables, views or constraints will be dropped, too, and those column definitions will be altered with the **DEFAULT** value or the constraint definition if it corresponds.

Domains are useful when several tables contain identical column definitions.

Another way to materialize domain restrictions is by including them into the column of a table definition. In the **CREATE TABLE** sentence, the user specifies the name of the table and its components following this syntax:

```
CREATE TABLE table-name (table-element-list);
```

Table-element-list is defined as a list of *table-element*, separated by commas. A *table-element* may be

```
column-definition | constraint-definition
```

whereas a *column-definition* is

```
column-name { data-type | domain-name }
[ DEFAULT default-option ]
[ column-constraint-definition ... ]
[ collate-clause ]
```

A *column-constraint-definition* is

```
[ constraint-name-definition ] column-constraint
[ constraint-attributes ]
```

And a *column-constraint*:

```
Not-null-definition | unique-constraint-definition |
referential-constraint-definition | check-constraint-
definition
```

On the other hand, a *constraint-definition* is specified as

```
[ constraint-name-definition ] table-constraint
[ constraint-attributes ]
```

where a table-constraint may be a *unique-constraint-definition*, a *referential-constraint-definition* or a *check-constraint-definition*. They are defined in their corresponding sections.

Modifications to the table definition can be expressed by means of the statement

```
ALTER TABLE table-name
[column-alteration-action | table-constraint-
alteration-action]
```

Using this sentence the user can alter column definitions and integrity restriction definitions. Related to the former case, it is possible to perform the addition of columns, the modification of the definition of a column or the elimination of a column. The elimination of a column fails if it is the unique column in a table definition or if the RESTRICT modality has been specified to perform the deletion and this column is yet referenced. Finally, if the table definition is no longer necessary, the user can express

```
DROP TABLE table-name {RESTRICT | CASCADE}
```

Default Option

Default values can be defined both in the domain definition and in the table definition. The default value can be NULL, a literal or some function provided by SQL. When a new default definition is added to the definition of a domain, it is automatically applied to all columns defined over such domain. On the other hand, when a default definition is eliminated, the default value is copied in the definition of all related columns. If a column defined over a domain has its own default value, it is preserved when a default definition is dropped or added to the domain definition.

Search-Condition

Constraints over domains can be defined as part of a domain definition, as part of a table definition or via a general definition. A constraint defined in a domain definition is expressed as:

```
[CONSTRAINT constraint-name ] CHECK (search-condition)
```

The *search-condition* includes logical combinations of simple expressions which, in turn, can be a BETWEEN comparison, a LIKE condition, an IN condition, a MATCH condition, a NULL condition, a table lookup, etc. Several examples are provided in the following sections.

In some situations, the user needs integrity constraints to be checked immediately, i.e., after each SQL statement has been executed. In other cases, the user needs the checking at the transaction commit. To accomplish those behaviors, the user can define a constraint as INITIALLY IMMEDIATE (default) or INITIALLY DEFERRED, respectively. In the first case an additional meta-restriction qualifying IMMEDIATE can be expressed using [NOT] DEFERRABLE. It indicates whether the option can be changed within the context of the current transaction.

A *constraint-definition* may also be included as part of a table definition.

Other constraints over columns of a table may be: null restrictions (*Not Null Definition*), a primary key constraint (*Primary Key Definition*), or a referential integrity restriction (detailed in the subsection "Object Restrictions").

Once more, these restrictions can be incorporated using the ADD table-constraint-definition clause and they can be eliminated using

```
DROP CONSTRAINT constraint-name { RESTRICT | CASCADE }
```

The creation of defaults and check constraints for columns cannot be defined for SQL supplied data types and columns of text, image, or timestamp types.

As it can be seen in [Figure 1](#), when the SQL facilities described in this section are insufficient to define a complex domain, assertions, checks, and even triggers can be used.

Not Null Definition

This can be specified as part of a column definition and used to indicate whether a column is allowed to contain nulls. Nulls are different from zeroes or blanks, and they are used to represent missing data items or not applicable ones. When NOT NULL is specified, the system rejects any attempt to insert a null value in the column; otherwise, the system accepts null values. The standard default is NULL.

Uniqueness Constraint Definition

In the relational model, a candidate key is a unique not null identifier involving one or more columns. The standard supports entity integrity with the PRIMARY KEY clause in the CREATE TABLE and ALTER TABLE statements. In this model, a table can have more than one candidate key but one of them must be designated as the primary key while the rest are considered as alternate ones. The uniqueness of alternate keys can be ensured by using the keyword UNIQUE.

On the contrary, in the SQL context the uniqueness restriction is optional and can be specified as part of the CREATE TABLE or the ALTER TABLE definitions, using the PRIMARY KEY or the UNIQUE clauses. All columns included into the uniqueness constraint must also be defined as NOT NULL. A table can have an arbitrary number of uniqueness restrictions but just one definition of a primary key.

A PRIMARY KEY can be specified as a part of the column definition, or separately in the table definition.

The syntax is:

```
[ CONSTRAINT constrain-name ] { PRIMARY | UNIQUE } (
column-list );
```

Data Type Features

The SQL3 standard has incorporated new features concerning user-defined and constructed data types. These types can be created using the CREATE TYPE sentence. A user-defined data type is a schema object whereas a constructed type (atomic or composite) is a data type having values that, in turn, can be composed of zero or more values of a declared data type. In this way, abstractions such as structured types and distinct types may be declared and used to define complex domains. A structured type is simply a user-defined data type comprising a number of attribute values that are encapsulated, i.e., they are not directly accessible to the user. A distinct type is a limited special case of a user-defined data type. Its physical implementation must involve exactly one of the built-in scalar types. Distinct types do not have implicit coercion to any other data type, even with the one on which it is based. On the other hand, row types are sequences of one or more (field name, data type) pairs. A value of a row type consists of one value for each of its fields. Columns of these types can be defined as non-nullable in the same way the predefined data types are.

```
CREATE TYPE <type-name> AS <built-in scalar type name>
FINAL ...
```

Object Restrictions

Regarding the definitions in [Table 1](#), only primary functional dependencies and referential integrity restrictions can be treated in the SQL context. Secondary functional dependencies and non key-based dependencies cannot be specified in SQL, even though the document presents some considerations about known dependencies in specific denormalized tables (SQL99-1, 1999) and the potential utility of the FOREIGN KEY match options.

An object constraint is associated with a specific table (which does not mean that it cannot refer to another table). In the SQL context, it can express a foreign key definition, a primary or candidate key definition, or an arbitrary combination of columns in a table.

Considering [Figure 1](#), primary and candidate keys are related to object restrictions, since they imply a functional dependency. On the other hand, primary and candidate keys are related to domain constraints since they cannot hold NULL values and they must be UNIQUE. For this reason, keys have been considered in the previous section.

Referential-Constraint-Definition

As was previously defined, a relationship can be established through a referential constraint whose syntax is:

```

FOREIGN KEY (column-list)
REFERENCES referenced-table-name [ (primary-key-column-
    list ) ],
[ MATCH { FULL | PARTIAL }
[ ON DELETE { NO ACTION | CASCADE | SET DEFAULT | SET NULL
    | RESTRICT } ]
[ ON UPDATE { NO ACTION | CASCADE | SET DEFAULT | SET NULL
    | RESTRICT } ] ]

```

The referential constraint specified by means of a FOREIGN KEY clause defines a relationship between a table T1 (referencing table) and a table T2 (referenced table). The number of columns involved in the foreign key must match the number of columns of the referenced primary key and their types must be compatible. Table T2 in the clause REFERENCES must identify a base table already defined in the catalog, but not a system table.

The MATCH option provides additional constraints in relation with foreign keys having null values. If the match is simple (default option), for each row of the referencing table either at least one of the values of the referencing columns is a null value or the value of each referencing column is equal to the value of the corresponding referenced column for some row. If the chosen option is MATCH FULL, the foreign key components must all have valid values or must all have null values. If the option is MATCH PARTIAL, the foreign key must be completely null or there must be at least one tuple in the referenced table that could satisfy the constraints if its nulls are properly replaced by valid values. Referential integrity is usually associated with MATCH FULL. Each match type constrains the previous one, i.e., match full is stronger than match partial, which in turn is stronger than match simple.

Within the reference constraint definition, it is possible to indicate the referential actions associated with updates and/or deletions in the referenced table. Referential actions are compensating operations, usually more effective than simply rejecting the operation that would violate a referential constraint. There are five possible actions: NO ACTION, RESTRICT, CASCADE, SET NULL and SET DEFAULT. When RESTRICT or NO ACTION (default action) is specified, no row is updated or deleted in T2 if one or more tuples in T1 reference it. Otherwise, the tuple in T2 may be updated or deleted, respectively. If the specified action is CASCADE, the actualization over T2 is propagated to the dependent tuples in T1. If SET NULL is declared, all foreign key values in the tuples which reference the ones intended to be actualized in T2, are set to null.

Obviously, if one or more of the foreign key columns are constrained by a null restriction, this option is not appropriate. Omitting the declaration of the clauses ON DELETE and/or ON UPDATE sets the default action: NO ACTION. Cycles of deletions must be avoided, with an exception: if all referential actions are CASCADE.

When two or more tables are connected by two or more referential paths starting in the same table T1 and ending in another table T2 (the same for all the paths), some

irregularities may show up if certain referential actions are combined. This problem is known as "the conterminous path problem" and it has been extensively studied (Date, 1989; Markowitz, 1994; Rivero & Doorn, 2000). In those works, sets of rules on the combination of referential actions are presented to avoid unpredictable results when the referenced table is updated.

Once more, as it happens with other restrictions, a foreign key constraint can be specified in the context of a column definition as part of a table definition or it can be added using the ALTER TABLE clause.

Differences Between NO ACTION and RESTRICT

NO ACTION and RESTRICT are referential actions for deletions and updates. They are different in relation to the moment in which the restriction is applied. RESTRICT updates or deletions are applied before other restrictions, including other update rules such as CASCADE or SET NULL. NO ACTION rules for deletions and/or updates are applied after other referential restrictions. The effect of the application of these rules produces different outcomes in only a few cases.

Check Constraints

The CHECK and CONSTRAINT clauses allow the definition of additional constraints. If used as a column constraint, as explained in the previous section, the CHECK clause can reference only the column being specified. If used as a table constraint, it is associated with a specific table and the *search-condition* can involve an arbitrary combination of its columns.

Execution Model For Declarative Integrity Constraints

Together with the declarative support for the integrity constraints given in SQL-99, the semantics of the possible interactions, which can exist among them, are also defined.

Once the set of tuples (rows) affected by the transaction is determined, BEFORE triggers are executed (before the original operation). Note that in this case, it is not possible to carry out database update operations. AFTER triggers are executed after the original operation has been completely executed and all declarative constraints have been verified. The order to which the declarative checkups is applied is:

1. referential integrity restrictions with RESTRICT modality.
2. referential integrity restrictions with CASCADE, SET NULL, or SET DEFAULT modalities
3. not null, unique/primary key, check and referential integrity with NO ACTION modality restrictions.

General Restrictions

Assertions

When a restriction involves an arbitrary complex combination of columns of an arbitrary number of tables, it is preferable to express it by means of the `CREATE ASSERTION` clause rather than duplicating a check on a column in each table definition. Assertions must be expressed following the syntax:

```
CREATE ASSERTION assertion-name CHECK (search-
condition)
```

A column restriction can be specified using `CREATE ASSERTION`, whereas a domain restriction cannot. This happens because in assertions it is impossible to define the domain type.

An assertion can be dropped by using `DROP ASSERTION`.

There are differences between the definitions of a constraint if it is in the `CREATE TABLE` context or if it is defined by a `CREATE ASSERTION` clause. If a column having an associated constraint is dropped from a table definition (with `RESTRICT` option), this operation will succeed. On the contrary, if it is involved in a `CREATE ASSERTION` clause, the dropping will fail. Multi-table assertions need to be evaluated when any table referenced in the condition is modified.

Triggers

An effort to define a standard for SQL triggers has been ongoing since the late 1980s. Even though trigger support was not included as part of the SQL-92 standard (probably because of the inadequacy of the standard document, which was very complex, especially in the section listing measures to avoid mutual triggering), they were supported by some products already in the early to mid-1990s (Zaniolo et al., 1997). The SQL-99 standard has extensive coverage of triggers, and today all major relational DBMS vendors have some support for triggers. Unfortunately, because the standard was influenced by preexisting product support, and many products do not do a good job integrating constraints and triggers, most products support only a subset of the SQL-99 trigger standard, and most do not adhere to some of the more subtle details of the execution model (Cochrane et al., 1996). Furthermore, some trigger implementations rely on proprietary programming languages for specifying parts of their triggers, which makes portability across different DBMSs difficult. There are a number of important details to the specification and execution semantics of triggers, only a few of which are covered here.

In contrast to declarative constraints, triggers are explicitly procedural. A trigger is an SQL compound statement that is automatically executed by the DBMS as a response to an insert, delete, or update on a particular table. Once activated, an optional specified condition is checked and, if the condition is true (or omitted), an action is executed. A trigger is a named SQL block, similar to a routine with declarative, executable, and conditional handling sections (Connolly et al., 1999). The basic format of the CREATE TRIGGER statement is as follows:

```
CREATE TRIGGER trigger-name
{BEFORE | AFTER} firing-event ON table-name
[REFERENCING old-or-new-values]
[FOR EACH {ROW | STATEMENT} ]
[WHEN (trigger condition)]
trigger-body
```

Firing events are the basic table manipulations (insertion, deletion and update). A BEFORE (respectively, AFTER) trigger is fired before (respectively, after) the associated event occurs. The triggered action can be executed in one of two ways: FOR EACH ROW or FOR EACH STATEMENT. In the former case, the action is executed for each row that is affected by the event.

In the second case the triggered action is executed only once for the entire event. This is the default option.

When FOR EACH ROW is stated, the old-or-new-values can refer to an old or new row. In case of an AFTER trigger, it refers to an old or new table.

The body of the trigger is a set of sentences, excepting COMMIT or ROLLBACK, SQL manipulation or definition sentences.

More than one trigger can be activated by the same event and in the same activation time. Hence, several triggers can be simultaneously selected for execution. If several triggers are fired at the same time, their executions are ordered with consideration to their timestamp. In case two or more triggers have the same timestamp, their relative order is determined by the implementation: each system has its own pre-established order, although the standard advises to follow the PostgreSQL/DB2 ordering model. On the other hand, several events can refer to the same trigger.

Even though triggers can be seen as ECA rules, the consideration of their condition and action parts can be neither detached nor deferred.

Stored Procedures and User-Defined Functions

The SQL2 standard, as originally defined, did not include any support for user-defined functions and stored procedures. However, commercial products have been providing

such issues for years (Date, 2000), (Cochrane et al., 1996; Türker & Gertz, 2001). With the incorporation of the Persistent Stored Modules (PSM) into the standard in late 1996, SQL became computationally complete, so object behavior (methods) can be stored and executed. It includes statements such as CALL, RETURN, SET, FOR, WHILE, etc., as well as several related features such as variables and exception handlers. Therefore, there should not be the need to combine SQL with some distinct "host" language in order to develop complete applications.

Current available system-stored procedures support includes the ability to create user-defined functions and procedures, to invoke such functions (for example from a SELECT clause), to invoke such procedures by a CALL (or similar) clause, and the provision of a proprietary programming language for the definition of these components.

PSM utilizes the term "routine" to cover both functions and procedures. Routines can be written in SQL or in another nonSQL language (usually a proprietary one). Key words, such as FUNCTION or PROCEDURE, identify the routine type. PSM routines share the programming language, their definition includes the definition of parameters—corresponding to the arguments provided in the invocation, and they are subject to the same authorization mechanisms, among other similarities (Date, 2000).

Procedures and even pieces of code embedded in the application programs are often employed to express general constraints, even though it is not the most recommended practice.

Although triggers extend the constraint logic with transitional constraints, exception handling and user defined repairing actions, they should not be used in lieu of declarative constraints (Cochrane et al., 1996). Many examples are provided in the section "Integrity Constraints in Current Commercial Products".

A model that integrates the execution of triggers and the evaluation of declarative constraints in SQL database systems is completely described in Cochrane et al. (1996) and Türker & Gertz (2001).

^[2]The following subsections have been developed taking into account the following references: Connolly et al., 1999, Date, 2000, Date & Darwen, 1997, SQL99-1, 1999, SQL99-2, 1999, Ceri et al., 2000, Zaniolo et al., 1997.

^[3]In some cases, it is presented an incomplete definition syntax. Just the clauses sufficient to explain integrity issues are shown.

PART II: REVIEWED PRODUCTS

To exemplify all integrity issues, the following current postrelational or object-relational systems will be considered: DB2®, Informix®, Oracle®, PostgreSQL and Sybase®. These systems are presented in alphabetical order. Details of each of them may be found in the Appendix.^[4]

The references to commercial products correspond to the following versions:

- **DB2 UDB**: DB2 Universal Database V 7.1
- **Informix**: Informix Dynamic Server V. 9.1.
- **Oracle**: Oracle8i Release 8.1.6
- **PostgreSQL**: PostgreSQL V 7.
- **Sybase** Sybase Adaptive Server V 12.0

Whenever necessary, other products or versions are explicitly mentioned.

UDB, as well as Oracle and Informix, are compliant with the SQL-92 Entry level, but include features from the Intermediate and Full levels, in addition to several features of SQL3.

PostgreSQL is compliant with SQL-92 language features, including primary keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input. The newest enhancements in PostgreSQL include important features such as subselects, defaults, constraints, and triggers, and improvements of built-in types (including new wide-range date/time types and additional geometric type supports).

Sybase is compliant with the SQL-92 Entry level. This behavior is set by default for embedded applications but it can be changed using a set of commands in Transact-SQL.

To illustrate the concepts developed in this chapter, according to the integrity issues provided by different commercial products, the following example will be considered.

Motivating Example: A Medicare Organization Database

Consider a Medicare Organization (MO) that provides health services for the employees of certain organization. An MO is supported by the contributions of its associates mainly through enforced deductions from their salaries. MOs have three main actors: the organization itself, the actual health service providers and the user or covered person. The health service provider may be an individual professional or from a health association such as Medicare centers or hospitals or even a district professional collegiate association. The users of the MO are those employees and their family group. Their imposed salary deductions or voluntary contributions are assigned to the MO.

In the MO context, an affiliate is first identified as a *person*, then as a *member* of a family group and finally according to the role he or she plays in the *family group* (dependent affiliates). According to these three perspectives, a person is a *titular affiliate* if he or she contributes to the MO with a percentage of his or her salary or by the payment of a fee in the framework of a pre-paid policy.

Each affiliate has a medical *service history*, i.e., the *services* the affiliate has made use of. It is the chronological record of all services the user has requested.

Within their MO, the set of services to which each family group can access is defined as a *health plan*.

The affiliates are divided into groups, identifying them according to the geographical location of their primary residence. The affiliate may contact the MO to require services or some formalities in the nearest office, the *delegation*.

[Figure 2](#) shows an EER diagram that represents the relevant aspects of the UofD, according to the Information Engineering (IE) methodology.

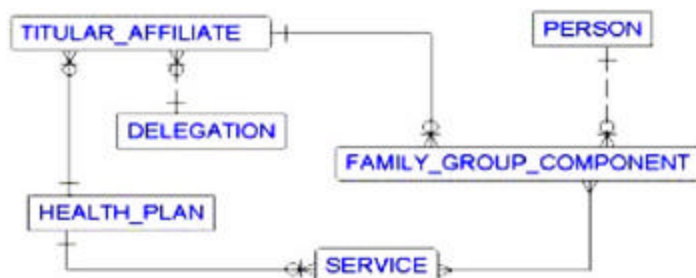


Figure 2: EER of the MO components

[Figure 3](#) shows the physical model of the EER in [Figure 2](#), also following the IE methodology.

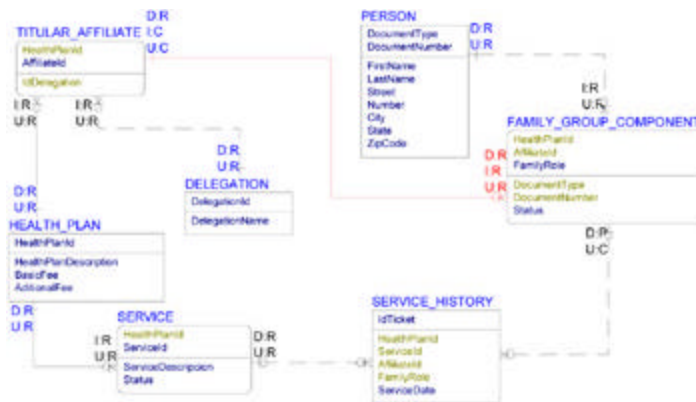


Figure 3: Physical model of the MO components

[4]References of this section are Chamberlin, 1998, DB2 UDB-1,1998, DB2 UDB-2,1998, DB2 UDB-3,1998, DB2 UDB-4, 2000, Informix, 1998, Kim et al, 1994, Oracle, 2000, Postgres, 2001, SQL99-1, 1999, SQL99-2, 1999, Sybase-1, 2001, Sybase-2, 2001, Sybase-3, 1999, Sybase-4, 1999, Sybase-5, 1999, Sybase-6, 1999, Sybase-7, 1999, Sybase-8, 2000, Türker & Gertz, 2001, Zaniolo et.al., 1997.

INTEGRITY CONSTRAINTS IN CURRENT DATABASE MANAGEMENT SYSTEMS

Domain Restrictions

The SQL syntax corresponding to these restrictions has been introduced in the section "[The SQL Standard Facilities](#)".

Informix, Oracle, PostgreSQL and UDB do not support CREATE DOMAIN but they provide facilities to define composite constructed data types instead.

Concerning the UDB design, IBM has included several characteristics in order to capture the data semantics. UDB SQL implementation does not support the sentence CREATE DOMAIN but UDB products provide some other mechanisms to define complex objects and to extend built-in data types with their own functions. User-defined structured types are among the data types in DB2 that allow the user to create a structure containing a sequence of named attributes, each one with their own data type. Structured types can be combined into a hierarchy, and they can be used as the type of a table or a view. Tables or views defined using a structured type are called typed tables and typed views, respectively. Structured types and typed tables enable the user to configure a better model of the business entities and relationships in the real world.

To specify user-defined data types, UDB provides the sentence CREATE DISTINCT TYPE. Example:

```
CREATE DISTINCT TYPE Name AS VARCHAR(30) WITH COMPARISONS

CREATE TABLE DELEGATION (
    DelegationId      INTEGER NOT NULL,
    DelegationName    Name NOT NULL CHECK ((VALUE NOT
    LIKE ' %'))
AND (VALUE <> ''))
);
```

CREATE DOMAIN and CREATE DISTINCT TYPE are different since the latter sentence cannot include default value definitions or constraint definitions. These definitions must be defined at table creation time.

For example, for the creation of a row type:

```
CREATE ROW TYPE Address
(Street      varchar (20),
Number      integer,
City        varchar (20),
State       varchar (20),
Zipcode     integer);
```

To define a domain integrity constraint the Oracle user must include a CONSTRAINT clause in a CREATE TABLE or ALTER TABLE statement.

Informix allows the definition of complex data types such as collections (LIST, SET, MULTISSET); named ROW (the Address definition, for example) and unnamed ROW. In this case, the previous example turns into:

```
CREATE TABLE PERSON (
  DocumentType      VARCHAR2(20) NOT NULL,
  DocumentNumber    NUMBER NOT NULL,
  FirstName         VARCHAR2(30) NOT NULL
  LastName         VARCHAR2(30) NOT NULL
  Address           ROW(Street varchar (20), Num-
                      ber integer, City varchar
                      (20), State varchar (20),
                      Zipcode integer)
);
```

In Oracle, it is possible to create either an object type, a named varying array (VARRAY), a nested table type, or an incomplete object type using CREATE TYPE although this command is available only if the Oracle object option is installed on the database server. For the creation of a type in a schema, the user must have the CREATE TYPE system privilege.

Oracle implicitly defines a constructor method for each user-defined type created. A constructor is a system-supplied procedure which is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the same as the name of the user-defined type.

An incomplete type is created by a forward-type definition. It is called "incomplete" because it has a name but no attributes or methods. However, other types can reference it and it can be used to define types that refer to each other. It is the correlate of the type definition in the SQL standard.

Sybase CREATE DATATYPE can be used as an alternative to CREATE DOMAIN. This practice is not recommended because the SQL3 standard provides the CREATE DOMAIN statement for this purpose. Once a data type is created, the user ID that executed the CREATE DOMAIN statement is the owner of that data type even though any user can use the data type. Domains can be dropped by their owner or by the DBA, using the DROP DOMAIN statement.

Many of the attributes associated with columns, such as allowing NULL values, having a DEFAULT value, and so on, can be built into a domain. Any column that is defined on the data type automatically inherits the NULL setting, CHECK condition, and DEFAULT values.

When needed, the attributes of the data type can be overridden by explicitly provided attributes for the column.

In Sybase, domains are created with a base data type, and optionally a NULL or NOT NULL condition, a default value, and a CHECK condition. Although the standard permits named constraints and named defaults, these issues are not supported.

The CREATE DOMAIN statement can be used only in Adaptive Server Anywhere. The sp_addtype system procedure can be used to add a domain both in Adaptive Server Anywhere and in Adaptive Server Enterprise.

In Oracle and UDB, not null constraints can be attached to each of the structured data components. Notice that this characteristic may promote updating problems. For example, the insertion

```
INSERT INTO Person (DocumentType, DocumentNumber, FirstName,
LastName, P-Address)VALUES ('D', '12345678', 'John Davis',
null);
```

will be rejected (supposed that P-Address is defined on the Address type), whereas

```
INSERT INTO Person (DocumentType, DocumentNumber,
FirstName, LastName, P-Address )VALUES ('D', '12345678',
'John Davis', address (null, null, null, null));
```

will be accepted. To avoid these semantic differences, the latter way should be disallowed. The correct employment of the nullability issues is by defining not null constraints on each structured data type component.

Default Option

All the reviewed products provide the *Default Definition* according to the SQL standard. For instance, the following sentence represents that "Inactive" is the default value of the affiliate status,

```
CREATE TABLE FAMILY_GROUP_COMPONENT
( ..., Status CHAR(8) NOT NULL DEFAULT 'Inactive', ...);
```

On the other hand, Sybase allows the specification of default values as part of the CREATE DOMAIN definition. The previous example, in this case is:

```
CREATE DOMAIN status AS char (8) NOT NULL DEFAULT
'Inactive',
CHECK (status IN ('Inactive', 'Active'));
```

Search Condition

Within all the reviewed products, a constraint definition can be specified as part of the CREATE TABLE sentence or by means of an ALTER TABLE statement. For instance in UDB, if the delegation names cannot begin with spaces and cannot be the empty string:

```
...., DelegationName Name NOT NULL
      CHECK ((VALUE NOT LIKE ' %') AND (VALUE <> ''));
```

Name has been previously defined using CREATE DISTINCT TYPE.

Columns specified in a unique constraint must be defined as NOT NULL. The following example shows the use of domain constraints for the remaining systems.

```
CREATE TABLE HEALTH_PLAN (
.....,
      BasicFee DECIMAL CHECK ((VALUE IS NULL) OR
      (VALUE >= 0)),
      AdditionalFee DECIMAL CHECK (BasicFee +
      AdditionalFee > 500)
);
```

In Sybase, domains are aliases for built-in data types, including precision and scale values, where applicable, and optionally, including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in Adaptive Server Anywhere, but the user can add more of his or her own.

Not Null Definition

In UDB, Informix, PostgreSQL and Oracle, NOT NULL and NULL can be specified in column constraints only in the CREATE TABLE context or via the ALTER TABLE statement. Oracle does not permit NOT NULL definitions in a table constraint.

Sybase allows the specification of NOT NULL or NULL in a CREATE TABLE or ALTER TABLE statement and CHECK IS NOT NULL in CREATE DOMAIN or ALTER DOMAIN statement.

Columns in Adaptive Server Enterprise default to NOT NULL, whereas in Adaptive Server Anywhere the default setting is NULL. This setting can be controlled using the ALLOW_NULLS_BY_DEFAULT database option. The user should explicitly specify NULL or NOT NULL to make data definition statements transferable between both versions.

Informix, Oracle, PostgreSQL and Sybase follow the SQL-99 prescription about primary keys, i.e. a not null definition is made implicit. UDB requires the explicit definition of the nullability clause.

Uniqueness Constraint Definition

A primary key can be defined on a single column or on a set of columns (composite primary keys).

In Oracle a primary key or unique key column cannot be of LONG or LONG RAW data types and a composite primary key can contain a maximum of 16 columns.

When dropping tables in Oracle (`DROP TABLE table_name [CASCADE CONSTRAINT]`) the following operations are automatically performed:

Oracle removes all rows from the table (as if the rows were deleted); it drops all the table indexes, regardless of who created them or whose schema contains them. If the table is a base table for views or if it is referenced in stored procedures, functions, or packages, Oracle invalidates these objects but it does not drop them.

UDB does not allow the user to disable/enable one constraint at a time. However, the user can disable/enable a group of constraints at a time. For example, they can be disabled/enabled all the constraints of a table or all the referential integrity constraints of a table. The command is `SET INTEGRITY`. The following example shows how to disable/enable all the constraint checkings for table T1.

```
SET INTEGRITY FOR T1 OFF;
SET INTEGRITY FOR T1 CHECK IMMEDIATE;
```

For a single transaction, the user can use `SET CONSTRAINT` to set if a deferrable constraint is checked following each DML statement or when the transaction is committed.

[Figure 4](#) shows a complete example which summarizes the previous ones. This example is valid for all systems but since Sybase supports domain definitions, Name definition could be replaced by

```
CREATE DOMAIN Name AS varchar(30) NOT NULL
CHECK ((VALUE NOT LIKE ' %') AND VALUE <> ''));
```

```

CREATE DISTINCT TYPE DESCRIPTION AS VARCHAR(60) WITH
COMPARISONS;
CREATE DISTINCT TYPE NAME AS VARCHAR(30) WITH
COMPARISONS;
CREATE DISTINCT TYPE STATUS AS CHAR(10) WITH
COMPARISONS;
CREATE TABLE FAMILY_GROUP_COMPO (
    HEALTHPLANID    VARCHAR(10) NOT NULL,
    AFFILIATEID      VARCHAR(15) NOT NULL,
    FAMILYROLE       NUMERIC NOT NULL,
    DOCUMENTTYPE     VARCHAR(20) NOT NULL,
    DOCUMENTNUMBER   INTEGER NOT NULL,
    STATUS           CHAR(8) NOT NULL DEFAULT
        'INACTIVE'
    CHECK (STATUS IN ('ACTIVE', 'INACTIVE'))
    PRIMARY KEY (HEALTHPLANID, AFFILIATEID,
        FAMILYROLE)
);

CREATE TABLE DELEGATION (
    DELEGATIONID     INTEGER NOT NULL,
    DELEGATIONNAME    NAME NOT NULL
    CHECK ((VALUE NOT LIKE ' %') AND (VALUE <> ''))
    CONSTRAINT PRIMARY KEY PK_DELEGATION
        (DELEGATIONID)
);

CREATE TABLE PERSON (
    DOCUMENTTYPE     VARCHAR(20) NOT NULL,
    DOCUMENTNUMBER   INTEGER NOT NULL,
    FIRSTNAME        NAME NOT NULL
    CHECK ((VALUE NOT LIKE ' %') AND (VALUE <>
        '')),
    LASTNAME         NAME NOT NULL
    CHECK ((VALUE NOT LIKE ' %') AND (VALUE <>
        '')),
    STREET           VARCHAR(20) NOT NULL,
    NUMBER           INTEGER,
    CITY            VARCHAR(20),
    STATE           VARCHAR(20),
    ZIPCODE         INTEGER
    CHECK (VALUE BETWEEN 1000 AND 9999)
    PRIMARY KEY (DOCUMENTTYPE, DOCUMENTNUMBER)
);

CREATE TABLE HEALTH_PLAN (
    HEALTHPLANID     VARCHAR(10) NOT NULL,
    HEALTHPLANDESCRIPT DESCRIPTION NOT NULL
    CHECK (VALUE BETWEEN 1000 AND 9999),
    BASICFEE         DECIMAL
    CHECK ((VALUE IS NULL) OR (VALUE >= 0)),
    ADDITIONALFEE    DECIMAL,
    CHECK (BASICFEE + ADDITIONALFEE > 500)
);

ALTER TABLE HEALTH_PLAN ADD PRIMARY KEY
(HEALTHPLANID);

```

Figure 4: The medicare organization example

Schema Maintaining

The DROP clause of an ALTER TABLE statement eliminates the constraint. When it is dropped, the system stops enforcing the constraint and removes it from the data dictionary.

Dropping the uniqueness or the primary key condition of an attribute, being part of a referential integrity constraint, requires the dropping of the foreign key and any other restriction mentioning it. In Oracle, the referenced key and the foreign key can be dropped together by specifying the referenced key with the CASCADE option in the DROP clause as established in the SQL-99 standard. When the CASCADE option is omitted (RESTRICT is the default), Oracle does not drop the unique or primary key constraint if any foreign key references it. This is the only reference system bearing this characteristic.

The following statement drops the primary key of the DEPT table:

```
ALTER TABLE dept DROP PRIMARY KEY CASCADE;
```

If the name of the PRIMARY KEY constraint is PK_DEPT, it can also be dropped by the following statement:

```
ALTER TABLE dept DROP CONSTRAINT pk_dept CASCADE;
```

Object Restrictions

Referential Constraint Definition

UDB supports deletions with NO ACTION, RESTRICT, CASCADE and SET NULL referential actions. As regards update operations, only NO ACTION and RESTRICT are permitted.

With respect to Oracle, Informix and Sybase, they implement the NO ACTION semantics for deletions and updates, but they do not allow the keywords ON DELETE (or ON UPDATE) NO ACTION. Additionally Oracle supports CASCADE and SET NULL options for deletions and no other option for updates. In addition, Informix provides the ON DELETE CASCADE clause. Sybase does not provide any support for declarative referential actions (triggers must be used for this purpose) and PostgreSQL permits ON DELETE (and ON UPDATE) with NO ACTION, RESTRICT, CASCADE, SET NULL and SET DEFAULT modalities. NO ACTION is the default option.

UDB has an additional restriction: if a table contains more than one foreign key, all these must coincide in the delete referential action. On the contrary, if the first defined restriction is ON DELETE SET NULL, no other foreign key can be defined for that table. These limitations permit to avoid some of the conterminous path problems (see the ["The SQL Standard Facilities"](#) section).

All reference systems, but PostgreSQL, support only MATCH SIMPLE (default). PostgreSQL permits the MATCH FULL clause and by default, it permits some foreign key columns to be NULL while other parts of the foreign key are not NULL.

In Sybase, a foreign key can reference either a primary key or a column with a unique constraint, but not a unique index, since it may include multiple instances of NULL. If the column-name is specified in a REFERENCES column-constraint, it must be a column in the primary table, it must be subject to a unique constraint or primary key constraint, and that constraint must consist of only one column. If column-name is not specified, the foreign key references the primary key of the primary table. If a foreign key column is not explicitly defined, it is created with the same data type as the corresponding column in the primary table. These automatically created columns cannot be part of the primary

key of the foreign table. Thus, a column used in both the primary and the foreign key of the same table must be explicitly created.

If Sybase foreign key column-names are specified, then primary key column names must also be specified. In these cases, the column names are paired according to their position in the lists. On the other hand, if at least one value in a multi-column foreign key is NULL, there is no restriction on the values which can be held in other columns of the key.

UDB Example:

```
ALTER TABLE SERVICE_HISTORY
ADD FOREIGN KEY (HealthPlanId, AffiliateId, FamilyRole)
REFERENCES FAMILY_GROUP_COMPO
ON DELETE SET NULL;
```

Deferrable or Not Deferrable Integrity Check

Sybase referential integrity has been implemented via the following two extensions of the CREATE TABLE and ALTER TABLE commands.

```
FOREIGN KEY [role-name] [(column-name, ...)]
REFERENCES table-name [(column-name, ...)] [ CHECK ON
COMMIT ]
```

Sybase has the database global option WAIT_FOR_COMMIT. If it is set to ON, or if a foreign key is defined using CHECK ON COMMIT in the CREATE TABLE statement, the database can be updated in such a way that if a referential integrity is violated, these violations are resolved before the changes are committed. In Sybase CHECK ON COMMIT clause overrides the WAIT_FOR_COMMIT database option.

As considered in the SQL2 standard, Oracle table and column constraints can be specified as DEFERRABLE or NOT DEFERRABLE.

General Restrictions

Assertions: Assertions have not been implemented in the commercial products revised, yet.

Triggers

All the reference systems implement triggers following a procedural approach. In most cases, each system provides a proprietary programming language. The relevant

characteristics of this issue in the evaluated system are examined in the following subsections.

UDB Triggers

UDB supports BEFORE ROW, AFTER STATEMENT, and AFTER ROW triggers; BEFORE STATEMENT triggers are not supported. UDB also supports transition variables, OLD and NEW tables, and OLD and NEW column values. BEFORE ROW triggers can only include select, set, and signal statements. AFTER TRIGGERS can include set, signal, inserts, updates, and deletes.

UDB triggers have the following characteristics: calls to DB2 functions and user-defined-functions (UDF) are permitted whereas calls to stored procedures are not; the firing order is based on the creation time; triggers can use the CASE expression; if the body contains more than one SQL statement DB2 allows the use of COMPOUND SQL and the SQL statements are enclosed between BEGIN COMPOUND ATOMIC and END, separated by semicolons.

UDB triggers are stored in the database and compiled at runtime together with the SQL statement associated with the trigger. Multiple triggers can be created for the same event, activation time and subject tables. A triggered action is composed of one or more SQL statements or by an optional condition for the execution of the SQL statements. Adding a trigger to a table having already rows will not cause triggered actions to be activated.

UDB triggers allow the specification of the columns of a table that will cause the trigger to be fired. If split up the trigger based on the affected columns is required, the WHEN clause should be utilized to handle some of the logic flow.

Informix Triggers

Informix triggers allow the inclusion of calls to stored procedures; the coding of the firing order; the use of the WHEN() expression; 'statement' as the default granularity; and AFTER and BEFORE activation times.

A delete_trigger is not allowed on a table containing a foreign key with Cascade referential action.

Informix allows just one trigger per event. In case of updates, multiple triggers are allowed whenever the lists of columns over which they are defined are mutually exclusive. ^[5]

Oracle Triggers

In Oracle, the trigger body consists of an anonymous PL/SQL block. Oracle *triggering statements* are DELETE, INSERT and UPDATE. A trigger must specify at least one of

these commands, allowing all of them. Trigger restriction can be specified within the WHEN clause. The condition must be a SQL condition, rather than a PL/SQL condition.

Oracle triggers can be specified with BEFORE, AFTER, FOR EACH ROW, FOR EACH STATEMENT options within the CREATE TRIGGER command.

An Oracle trigger must be enabled or disabled. If it is *enabled*, whenever a triggering statement is issued and the condition of the trigger restriction is met, the trigger is fired. On the contrary, if a trigger is *disabled*, even through a triggering statement is issued and the condition of the trigger restriction is met, the trigger is not fired.

When a trigger is created, Oracle enables it automatically, but it can be subsequently disabled and enabled with the DISABLE and ENABLE options of the ALTER TRIGGER command or the ALTER TABLE command.

When a trigger is created for more than one operation, *conditional predicates* can be used within the trigger body to execute specific blocks of code, depending on the type of statement that fires the trigger (INSERTING, DELETING, UPDATING and UPDATING (column)).

Whenever multiple triggers of the same type, firing for the same command, are associated with the same table, the order in which Oracle fires these triggers is indeterminate. If the application requires that one trigger is to be fired before another one, both triggers should be combined into a single one whose trigger-action performs the tasks of the original triggers in the appropriate order.

In Oracle allows INSTEAD OF triggers to perform DELETE, UPDATE, or INSERT operations on views, which are not inherently modifiable. Oracle produces the mutating error problem since it implements the concept of versioning.

PostgreSQL Triggers

PostgreSQL allows the invocation of C language functions from the trigger-action. In this version, statement-level triggers are not supported and, as well as in Oracle and Sybase, multiple events per trigger are possible. PostgreSQL accepts AFTER- and BEFORE-triggers and provides the support for cascading triggers, without an explicit limit on the number of levels.

Sybase Triggers

In Sybase, triggers are coded using Transact-SQL and stored in the database. The SQL Server allows nested triggers by default, and multiple events per trigger. Statement-level triggers are the only supported ones. The default granularity is FOR EACH ROW and the activation time is AFTER.

Examples

Since none of the reference systems allows the declarative definition of "ON INSERT CASCADE", the following example illustrates this issue, in relation with the Medicare Organization example. In this case, the desired behavior is that when a TITULAR_AFILIAE is inserted, the corresponding tuple in the FAMILY_GROUP_COMPONENT table must be inserted too (see the relationship between these two entities in [Figure 3](#)).

This behavior is supported by an AFTER trigger since the execution of its body must follow the original sentence. If not, the FAMILY_GROUP_COMPONENT reference should be violated (it is RESTRICT).

Only the HealthPlanId, AffiliateId and FamilyRole values are provided, leaving the system to complete the remaining ones with default values.

UDB Example

```
create trigger INSERT_TIT_AFFIL after INSERT on TITULAR_AFFILIATE
  REFERENCING NEW AS NEW for each row mode db2sql
-   TITULAR_AFFILIATE AND FAMILY_GROUP_COMPO ON PARENT INSERT
    CASCADE
  insert into FAMILY_GROUP_COMPONENT (HealthPlanId,
    AffiliateId, FamilyRole)
    VALUES (inserted.HealthPlanId, inserted.affiliateId,
    0)
end!
```

Informix Example

```
create trigger INSERT_TIT_AFFIL INSERT on TITULAR_AFFILIATE
referencing NEW as inserted
for each row
-   TITULAR_AFFILIATE AND FAMILY_GROUP_COMPO ON PARENT INSERT
    CASCADE
  insert into FAMILY_GROUP_COMPONENT (HealthPlanId,
    AffiliateId, FamilyRole)
    VALUES ( inserted.HealthPlanId, inserted.affiliateId,
    0)
;
```

Oracle Example

```
create trigger INSERT_TIT_AFFIL after INSERT on TITULAR_AFFILIATE
for each row
begin
/*   TITULAR_AFFILIATE AND FAMILY_GROUP_COMPONENT ON PARENT
    INSERT CASCADE */
  insert into FAMILY_GROUP_COMPONENT (HealthPlanId,
    AffiliateId, FamilyRole)
    VALUES ( :new.HealthPlanId, :new.affiliateId, 0)

end;
```

Sybase Example

```
create trigger INSERT_TIT_AFFIL on TITULAR_AFFILIATE
for INSERT as
begin
/*  TITULAR_AFFILIATE AND FAMILY_GROUP_COMPONENT ON PARENT
    INSERT CASCADE */
insert into FAMILY_GROUP_COMPONENT (HealthPlanId,
    AffiliateId, FamilyRole)
    VALUES ( :new.HealthPlanId, :new.affiliateId, 0)

end;
```

PostgreSQL Example

```
create function tg_upd_cascade_ta() returns opaque as '
begin
/*  TITULAR_AFFILIATE AND FAMILY_GROUP_COMPONENT ON PARENT
    INSERT CASCADE */
insert into FAMILY_GROUP_COMPONENT (HealthPlanId,
    AffiliateId, FamilyRole)
    VALUES ( :new.HealthPlanId, :new.affiliateId, 0)
return new;
end;
create trigger INSERT_TIT_AFFIL after INSERT on TITULAR_AFFILIATE
for each row
execute procedure tg_upd_cascade_ta()

end;
```

Stored Procedures and User Defined Functions

Although the declarative support for integrity issues has greatly evolved over the last years, regrettably most products today follow a declarative integrity support approach. While the situation is slowly improving in this regard, some products (specially nonrelational ones) specifically emphasize the opposite approach, i.e., procedural support, using stored or triggered procedures. Stored procedures are precompiled procedures, usually stored at the server site, which can be invoked from application programs (i.e. the client) by a remote procedure call (RPC). One of the advantages of stored procedures is to share the program with multiple clients. From a performance viewpoint, there will be less network overhead because there will be less client-server traffic. Stored procedures provide better security: for instance, a user might be authorized to invoke a stored procedure but not to operate directly on the data accessed by that procedure (Date, 2000).

One disadvantage of stored procedures is that different vendors offer very different facilities in this area, despite the fact that the SQL2 was extended to include some stored procedure support under the name of Persistent Stored Modules (PSM).

UDB stored procedures are written in a 3GL since it does not provide a proprietary 4GL. C, COBOL, Java, FORTRAN, and other languages can be used to code stored procedures using embedded static or dynamic SQL. UDB does not allow a stored procedure to call

another stored procedure and it does not support commit or rollback commands inside stored procedures.

Informix allows a stored procedure to call another stored procedure. It uses the SQL extension procedural language SPL to code them.

In Oracle, stored procedures must be programmed in PL/SQL. Oracle support commit or rollback inside stored procedures.

PostgreSQL follows the SQL proposal as regards the PSM procedural language and Sybase stored procedures are coded using Transact-SQL.

[\[5\]](#) The remaining systems do not have this limitation.

CONCLUDING REMARKS

This chapter provides an overview of semantic integrity characteristics in the SQL-99 standard and some major relational and object relational database management systems: DB2 UDB, Informix, Sybase, Oracle and PostgreSQL.

It can be pointed out that, in relation with integrity features, these systems basically support the Entry level of SQL-92 and some features of SQL-99. Since these systems have implemented triggers prior to the introduction of the SQL-99, there is an inevitable discrepancy with this standard. It is expected that, as time goes by, current DBMSs will incorporate missing characteristics and will be compliant with the standard.

APPENDIX: POSTRELATIONAL DATABASE SYSTEMS

DB2® Universal Database (UDB) Version 7.1

IBM DB2® Universal Database (UDB) is the latest generation of relational database products developed at IBM's laboratories in Toronto, Canada and San Jose, California. UDB shares the DB2 name but uses recent technology based on the Starburst architecture developed at Almaden Research Center (Filkenstein & Widom, 1989). It provides object-relational features, supports different types of applications, in different software and hardware environments. These latter characteristics permit the scaling from a single-user database on a personal workstation to terabyte databases on large multiuser platforms.

UDB servers run on several platforms: Windows NT, OS/2, many Unix-based systems including AIX and Solaris while UDB clients run on Windows 95, 98 and recent releases, and Macintosh systems.

UDB presents some SQL3 compliant features including enhanced and nontraditional data types, procedures and functions, active rules and recursive SQL extensions. It permits the

users to create new data types and functions and to define constraints and triggers to encapsulate their business rules. This product offers a set of "extenders" for the management of images, audio and video datatypes (Chamberlin, 1998; Ceri & Fraternali, 1997).

The DB2 UDB products and components include:

DB2® UDB Satellite Edition is a reduced version of DB2, for a single-user environment. It can be installed in systems supported by 32 bits Windows. It was developed focusing on remote systems that occasionally connect to the database such as portable PC systems. Usually the same server manages several instances of DB2 UDB Satellite Edition in a centralized way.

DB2® UDB Personal Edition is a complete version of DB2 for a single-user environment. It includes an object-relational database engine; the support for Business Intelligence by means of the OLAP Starter Kit; the support for a datawarehouse through the Datawarehouse Center; and multimedia support by means of the DB2 "extenders". It also provides the access to a great variety of IBM data sources through the DB2 DataJoiner facility; data replication support by using the DataPropagator issue; extended tools for managing GUI through the DB2 Control Center; and a client for applications development. This product is available for the OS/2, Windows NT, and Windows 95 operating systems.

DB2® UDB Workgroup Edition is a DB2 version for a small multiuser environment. It has been developed for small organizations. It contains all issues and functions of the Personal Edition version and includes the following extra characteristics: the possibility for remote users to access the data and to perform management tasks in a DB2 workgroup server; Web access through Net.Data; and the application server IBM WebSphere. It enables local clients, remote clients and applications to create, update, control and manage relational databases using Structured Query Language (SQL), ODBC, or CLI and contains all the latest DB2® Client Application Enablers, which enable client workstations to access the DB2 UDB server and all supported DB2 Net.Data products.

DB2® UDB Enterprise Edition has been developed for large databases with an important number of users. It incorporates all Workgroup Edition version characteristics and also includes:

DB2® Connect Enterprise Edition to support the host connectivity. This provides multi-user access to DB2 databases residing on host systems such as MVS/ESA, OS/390, AS/400, VM, and VSE. The DB2 Enterprise Edition supports unlimited LAN database access.

DB2® UDB Enterprise - Extended Edition (formerly known as DB2 Parallel Edition). This is the larger version, thought for large databases. It is the proper product to develop datawarehousing, data mining, and great scale OLTP applications. It includes server cluster support and enables a database to be partitioned across multiple independent

computers of a common platform. SQL operations and utilities can operate in parallel on the individual database partitions.

DB2 Software Developer's Kit (DB2 SDK). This component is a collection of tools that enable database application developers to build character-based, multimedia or object-oriented applications. It includes libraries, header files, documented APIs and sample programs.

It can be used to develop applications that use the following interfaces: Embedded SQL (both static and dynamic), Call Level Interface (CLI) development environment (compatible with ODBC from Microsoft), Java Database Connectivity (JDBC), etc. DB2 SDK also supports several programming languages (including COBOL, FORTRAN, Java, C, and C++) for application development, and provides precompilers for the supported languages. It is available on all DB2 UDB-supported platforms.

Informix® Dynamic Server (IDS)^[6] Version 9.1

The following are some products of the Informix's family:

Informix Dynamic Server (IDS) is Informix latest database server. It is the full product, multi-user version for Intel and UNIX platforms. It does not support user-defined types, user-defined functions or other object relational features without the Universal Data Option. **IDS** supports two types of large object data types: Byte and Text. **IDS** is capable of supporting many concurrent users with high reliability, availability, and scalability. It offers the following features and enhancements: increased performance benefits; enhanced Virtual Table Interface (VTI) (providing the ability to integrate and view legacy data from a variety of dissimilar systems, databases, and formats); fault-tolerant capabilities; and easy migration from previous Informix database products (Informix, 1998).

Developer's Edition This is the single-user version for workstations which is generally used as a development platform.

Personal Edition This product is a subset of Dynamic Server to be properly used in single-user mode, while **Workgroup Server** is also a subset of Informix Dynamic Server which is more suitable for smaller applications running on low-end processors.

Advanced Decision Support Option This product extends the capabilities of Dynamic Server's indexing and optimizer for decision support applications.

Extended Parallel Option (XPS — 8.2) It extends the capabilities of Dynamic Server for inter-parallelism.

Universal Data Option (9.1) It extends the capabilities of Dynamic Server for object relational support.

Related to new technologies to be integrated with the Web, Informix provides the following issues: a character-based user interface that is installed and used on the server for manipulating database objects, running queries and scripts (Dbaccess); a graphical user interface (GUI) for the workstation that is used to manage Informix databases (Dbcockpit); a graphical user interface that runs on the workstation for managing Informix databases and data (Enterprise Command Center); a Web based administration tool (OnWeb); and runtime libraries for INFORMIX-ESQL for C and COBOL and INFORMIX-CLI (Informix-Connect).

With respect to the application development support, some Informix products are: Informix-SQL - a separate product that is used for interactive SQL access; Informix Client Software Developer's Kits, providing single packaging for several application programming interfaces (including C, C++, Java and ESQL); and a rapid development language that compiles into C language (Informix-4GL). Other packages include: an SQL API that permits the embedding of both static and dynamic SQL statements directly into a 3GL program (ESQL product releases for C and COBOL); a graphical development environment (NewEra); and a Call Level Interface that enables application developers to dynamically access Informix database servers (**Informix CLI SQL Extenders (DataBlades)**) are object extensions that expand the capabilities of Informix Dynamic Server to manage complex data types such as video, audio and image, as well as to develop and use functions to manipulate these data types.

With Informix DataBlade technology, the intelligence of the application is extended by adding geospatial and regional information as a natural extension to the data managed by the server.

Informix DataBlade modules are not just options, but actual server extensions that are integrated into the very core of the engine. DataBlade modules integrate traditional alphanumeric data types, without sacrificing the reliability and scalability of the traditional relational DBMS. Informix DataBlade technology lets businesses treat Web sites as applications, letting the Informix Internet Foundation 2000 manage and dynamically deliver all site content. The server can be rapidly modified to accommodate new data types as business requirements evolve.

Informix-Gateway with DRDA provides access to non-Informix databases such as Oracle, Sybase, and DB2. In addition, some products of the Informix family support Web applications:

Internet Foundation 2000; Universal Web Connect (it provides connectivity between Web servers and Informix Dynamic Server); **Data Director for Web** (a suite of graphical user interface tools) designed to enable a developer to build and manage Informix-based Web sites and applications.

An Informix RDBMS consists of a database server, a database, and one or more client applications. Informix Dynamic Server works with relational databases and a

multithreaded relational database server that exploits symmetric multiprocessor (SMP) and uniprocessor architectures.

IDS permits the following types of data: integer, floating-point number, character string, fixed or variable length, date and time, time interval, numeric, decimal and complex data stored in objects System catalog tables track the following objects: tables, constraints, views, triggers, authorized users and privileges that are associated with tables and stored procedures

Informix Dynamic Server supports the following types of databases: ANSI compliant, Distributed, Distributed on multiple vendor servers and Dimensional (data warehouse)

Oracle Server Version 8i. Release 1.6

Oracle is based on the SQL language and includes several features from SQL92 and the preliminary documents of the SQL3 standard for triggers. Oracle Server includes declarative facilities ensuring scalable, reliable enforcement of data integrity while minimizing development, maintenance, and administration costs. It provides PL/SQL, an advanced procedural 4GL language that is tightly integrated with the Oracle Server, offering the power to easily express complex business rules as stored, procedural code.

Application Development SQL implementations are 100-percent ANSI/ISO SQL 92 Entry Level compliant. It includes features to support SQL extensions including UNION, INTERSECT, MINUS, outer join, and tree-structured queries SQL3 inline views (query in the FROM clause of another query). Oracle also provides support for declarative integrity constraints 100-percent ANSI/ISO standard declarative entity and referential integrity constraints.

Some products of the Oracle Corporation are:

Oracle Database Enterprise Edition Options These options for Oracle Database Enterprise Edition extend the power of the Oracle database in secure data management, transaction processing, and datawarehousing.

Application Servers The Oracle Internet Application Server runs a great variety of Internet applications. It enhances the power of the Oracle database to provide all the features needed for a complete, simple platform for the Internet.

Internet Application Server Oracle offers a complete suite of application development and business intelligence tools for building any kind of e-business application using the latest Internet technologies.

Integration Products Oracle Integration Products enable the users to integrate their legacy data and applications into the Oracle environment. Oracle offers solutions for the Data Warehousing requirements, supplying tools to design, to build, to deploy and to manage an Intelligent Webhouse.

Oracle provides Java functionality, XML support, and security features. Breakthrough Internet features, built directly inside the database, help developers build Internet-savvy applications providing global information access across platforms and across the enterprise.

On October 2, 2000, Oracle announced the **Oracle9i** database, the newest generation of the company RDBMS. Oracle9i includes built-in OLAP, Data Mining and ETL functions so that the database can act as a single repository of relational data as well as analytical data. It also includes infrastructure for developers to create hosted applications with common, collaborative software services; and continues to add features and capabilities mainly related to development platform; manageability; Windows2000 integration; Internet content management packaged applications and business intelligence (Oracle, 2000).

PostgreSQL Version 7.1

Implementation of the Postgres DBMS began in 1986 and it has undergone several major releases since then.

The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1 was released in June 1989 (the rule system was redesigned) and Version 2 was released in June 1990. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rewrite rule system. For the most part, releases until Postgres95 focused on portability and reliability. The size of the external user community nearly doubled during 1993. It then became obvious that the maintenance of the prototype code and support was taking up far too much time, which should have been devoted to database research instead. In an effort to stop this burden of support, the project officially ended with Version 4.2. In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter to Postgres. Since Postgres is intended to supercede the Ingres RDBMS, the intention is to integrate object-oriented features into a database system, maintaining its relational database background (Kim, Nelson & Rossiter, 1994). Regarding data and integrity issues, the main design goals of Postgres are to offer better support for complex objects; to provide user extendibility for data types, operators and access methods; and to make available alerters and triggers to implement active characteristics.

Postgres95 was subsequently released to the Web to find its own way in the world as an open-source descendant of the original Postgres Berkeley code. Postgres95 code was completely ANSI C and trimmed in size by 25%. Some of its major enhancements are:

1. The query language Postquel was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL, but they could be imitated in Postgres95 with user-defined SQL functions. Aggregates were re-implemented.

Support for the GROUP BY query clause was also added. The libpq interface remained available for C programs.

2. The instance-level rule system was removed. Rules were still available as rewrite rules.
3. A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code.

Table-level locking has been replaced by multi-version concurrency control, which allows readers to continue reading consistent data during writer activity and enables hot backups from pg_dump while the database remains available for queries.

By 1996, the name "Postgres95" was replaced by PostgreSQL reflecting the relationship between the original Postgres and the more recent versions with SQL capability. The Object-Relational Database Management System now known as PostgreSQL was derived from the Postgres package written at Berkeley.

As a traditional relational database management system (RDBMS), it supports a data model consisting of a collection of named relations, containing attributes of a specific type. It includes floating-point numbers, integers, character strings, money, and date types. Postgres offers the following four additional basic concepts in such a way that users can easily extend the system: classes, inheritance, types and functions. These features put Postgres into the category of databases referred to as object-relational. Other Postgres features are constraints, triggers, rules and transaction integrity.

It is an open-source database offering multi-version concurrency control, supporting almost all SQL constructs (including subselects, transactions and, user-defined types and functions), and having a wide range of language bindings available (including C, C++, Java, perl, tcl, and python) (Postgres, 2000).

Important backend features, including subselects, defaults, constraints, and triggers, have been implemented and additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coercion, type casting and, binary and hexadecimal integer input.

Built-in types have been improved, including new wide-range date/time types and additional geometric type support.

In the relational model the basic data structure is the relation (table), whereas in this system it is the class. Classes are collections of instances of objects used to define complex types and represent abstract data types (ADTs). They permit the definition of types of columns in the relational tables thus, allowing complex data to be stored in a field of a table. In this way, attributes of a table can be user-defined types, operators functions or procedures.

Data manipulation in Postgres is provided via its own query language PostQUEL, which is an extension to the Ingres' QUEL relational calculus that can deal with ADTs, inheritance, and many other features. Additionally, tables in Postgres can be manipulated via C language. Rules can also be incorporated into the data manipulation features (triggers and alerters).

Sybase Adaptive Server Enterprise 12.0

Sybase Adaptive Server Enterprise (ASE) 12.0 is designed to support the demanding requirements of Internet as well as traditional, mission-critical OLTP and DSS applications. The multi-threaded architecture, internal parallelism, and query optimization of Adaptive Server Enterprise deliver high levels of performance and scalability.

Sybase Adaptive Server is compliant with the SQL standard thus, allowing the definition of column-level integrity constraints and table-level integrity constraints. Integrity constraints may be expressed in the CREATE TABLE statement or they can be specified by means of triggers, rules, defaults and indexes. Transact-SQL is the Sybase 4GL language. It provides two methods for maintaining database integrity: a) defining rules, triggers, indexes or defaults and b) defining CREATE TABLE constraints (Sybase-4, 1999).

ASE is designed to support the demanding requirements of Internet and traditional, mission-critical OLTP and DSS applications. It is available on the following platforms: Sun Solaris, IBM RS6000, Digital, UNIX, HP UX, Windows NT. This system introduces row-level locking (RLL) capabilities designed to provide faster performance, fewer deadlock contentions, and greater flexibility in the management of system resources. It presents three types of locking strategies to ensure the widest range of versatility in application environment: Datapage Locking, Datarow Locking, All-Page Locking.

Additionally, this product includes enhancements to the optimizer, query processing improvements such as index statistics and descending keys, database recovery enhancements and improved space management features.

It also provides optimization improvements for SQL queries that contain "OR" clauses and dynamic SQL requests bypassing expensive catalog activities via the use of a feature called "lightweight stored procedures".

Some products of the Sybase line are:

SQL Anywhere Studio is a comprehensive package that provides data management and enterprise synchronization to enable the rapid development and deployment of distributed e-Business solutions.

Adaptive Server IQ is a relational database designed specifically from the ground up to meet the needs of business intelligence and a new generation of scalability requirements for Web-enabled data warehousing.

EAI/Middleware Key Products. Enterprise Application Integration (EAI) solutions from Sybase deliver data integration, data replication, and event handling across the entire enterprise. They are designed to match the needs of enterprise IT customers, from mainframes to the Web and from single database users to global organizations with multiple systems spanning diverse geographies. Sybase EAI/Middleware products used in conjunction with Enterprise Portal provide an array of integration options for data, events and application, without the re-engineering of the legacy systems.

Sybase Enterprise Portal It is the foundation for e-Business. Sybase EP is an extensible portal environment that meets e-Business requirements. Sybase EP is built upon a global-class portal platform for highly secure, personalized and, scalable portal deployments.

Adaptive Server Anywhere It provides relational database technology designed specifically for the needs of mobile and embedded computing. The relational database at the heart of SQL **Anywhere** has been designed from the ground up with this market in mind. **Adaptive Server Anywhere** has been designed to operate efficiently with limited memory, CPU power, and disk space. At the same time, **Adaptive Server Anywhere** contains the features needed to take advantage of workgroup servers, including support for many users, scalability over multiple CPUs and, concurrency features.

Adaptive Server Anywhere runs on Windows (95, 98, NT, and CE), UNIX, Novell NetWare, and Linux. It supports both entity and referential integrity. It supports SQL standards being compatible with SQL/92 Entry level feature.

Applications communicate with the database server using a programming interface (ODBC, JDBC, Sybase Open Client, or Embedded SQL). The programming interface provides a set of function calls for communicating with the database.

The **Personal database** version of **Adaptive Server Anywhere** is generally used for standalone applications. A client application connects through a programming interface to a database server running on the server. With **Adaptive Server Anywhere Network Database Server**, which supports network communications, SQL Anywhere can be used to build an installation with many applications, running on different machines, connected over a network to a single database server running on a separate machine.

In the three-tier computing, application logic is held in an application server, such as **Sybase Enterprise Application Server**, which fits between the database server and the client applications. In many situations, a single application server may access multiple databases in addition to non-relational data stores. In the Internet case, client applications are browser-based, and the application server is generally a Web server extension. Sybase Enterprise Application Server stores application logic in the form of components, and

makes these components available to client applications. The components may be PowerBuilder components, Java beans, or COM components.

Adaptive Server Anywhere personal and network database servers can both mount several databases simultaneously. Databases on multiple database servers, or even on the same server, can be accessed using the **Adaptive Server Anywhere Remote Data Access** features.

[6] IBM (NYSE:IBM) and Informix Corporation (Nasdaq: IFMX) announced on April 24, 2001 that they have entered into a definitive agreement for IBM to acquire the assets of Informix Software—Informix's database business—in a cash transaction valued at \$1 billion.

ENDNOTES

1. Naturally, this issue concerns concepts such as isolation levels, concurrency control and others related to them. They constitute another perspective of the database integrity problem.
2. The following subsections have been developed taking into account the following references: Connolly et al., 1999, Date, 2000, Date & Darwen, 1997, SQL99-1, 1999, SQL99-2, 1999, Ceri et al., 2000, Zaniolo et al., 1997.
3. In some cases, it is presented an incomplete definition syntax. Just the clauses sufficient to explain integrity issues are shown.
4. References of this section are Chamberlin, 1998, DB2 UDB-1, 1998, DB2 UDB-2, 1998, DB2 UDB-3, 1998, DB2 UDB-4, 2000, Informix, 1998, Kim et al, 1994, Oracle, 2000, Postgres, 2001, SQL99-1, 1999, SQL99-2, 1999, Sybase-1, 2001, Sybase-2, 2001, Sybase-3, 1999, Sybase-4, 1999, Sybase-5, 1999, Sybase-6, 1999, Sybase-7, 1999, Sybase-8, 2000, Türker & Gertz, 2001, Zaniolo et.al., 1997.
5. The remaining systems do not have this limitation.
6. IBM (NYSE:IBM) and Informix Corporation (Nasdaq: IFMX) announced on April 24, 2001 that they have entered into a definitive agreement for IBM to acquire the assets of Informix Software—Informix's database business—in a cash transaction valued at \$1 billion.

REFERENCES

- Ceri, S. & Fraternali, P. (1997). *Designing database applications with objects and rules: The IDEA methodology*. Addison Wesley.
- Ceri, S. ; Cochrane, R. J. & Widom, J. (2000). *Practical Applications of Constraints and Triggers: Successes and Lingering Issues. Proceedings of 26th. VLDB Conference*, Cairo Egypt, September 2000.
- Cochrane, R.; Pirahesh, H. & Mattos, N. (1996). *Integrating triggers and declarative constraints in SQL database systems. Proceedings of 22nd. VLDB Conference*, Mumbai (Bombay) India.

- Codd, E. (1990). *The relational model for database management*. Version 2. Addison Wesley Publ. Co.
- Connolly, T., Begg, C. & Strachan, A. (1999). *Database systems: A practical approach to design, implementation and management*. 2nd. Edition. Addison Wesley.
- Chamberlin, D. (1998). *A complete guide to DB2 Universal Database*. Morgan Kauffman Publishers. Co.
- Date, C. & Darwen, H. (1997). *The SQL standard*. 4th. ed. Addison-Wesley.
- Date, C. (1989). *Relational Databases, Selected Writings*. Addison Wesley. Reprinted with corrections.
- Date, C. (2000). *An introduction to database systems*. Addison Wesley.
- DB2 UDB-1. (1998) *Informix to DB2 Migration Comparison White Paper*. Software Migration Project Office. DB2 Migration Team. [On line] Available at: <http://www.ibm.com/solutions/softwaremigration>
- DB2 UDB-2 (1998) *Oracle to DB2 Migration Comparison White Paper*. Software Migration Project Office DB2 Migration Team. [On line] Available at: <http://www.ibm.com/solutions/softwaremigration>
- DB2 UDB-3. (1998) *Sybase to DB2 Migration Comparison White Paper*. Software Migration Project Office DB2 Migration Team. [On line] Available at: <http://www.ibm.com/solutions/softwaremigration>
- DB2 UDB-4. (2000). *DB2 Universal Database Workgroup Edition V. 7.1*. Information Center. DB2 Manuals.
- Etzion, O. (1993). *PARDES - A Data-Driven Oriented Active Database Model*. *SIGMOD Record*, 22(1).
- Informix Software, Inc. (1998) *Informix Guide to SQL: Tutorial*. [On line] Available at: <http://www.informix.com.my/answers/english/docs/visionary/infoshelf/sqlt/>.
- Kim, M.J., Nelson, D.A., Rossiter, B.N. (1994). *Evaluation of the Object-Relational DBMS Postgres .I. Administrative Data*. Newcastle University. October 1994.
- Markowitz, V. (1994), *Safe Referential Integrity and Null Constraint Structures in Relational Databases*. Personal communication.
- Oracle Corporation. (2000). *Oracle8i. Language Reference manual*.
- PostgreSQL Interactive Documentation. (2001) *PostgreSQL 7.1 Documentation*. [On line] Available at: <http://www.postgresql.org/docs/>.
- Rivero L., & Doorn J. (2000). *Static Detection of Sources of Dynamic Anomalies in a Network of Referential Integrity Restrictions*. In *Proceedings of 2000 ACM SAC*. Como, Italy. March 2000.
- Ross, R. G. (1997). *The Business Rule Book. Classifying, Defining and Modeling Rules*. Database Research Group, R. Ross Editor/Publisher.
- SQL99-1. (1999). *Database Language SQL*. Part 1: SQL Framework Document ISO/IEC 9075-1: 1999.
- SQL99-2. (1999). *Database Language SQL*. Part 2: SQL Foundation Document ISO/IEC 9075-2: 1999.
- Sybase-1. Sybase ®Adaptive Server™ (2001) *Introduction to Adaptive Server Enterprise 11.9.2* [On Line] Available at: <http://netimpact.sybase.com/products/databaseservers/ase/ase1192.html>.
- Sybase-2. Sybase ®Adaptive Server™ (2001) *Products*. [On Line] Available at: <http://www.sybase.com/products>.

- Sybase-3. Sybase® Adaptive Server™ Enterprise. (1999). *Transact-SQL User's Guide Adaptive Server Enterprise version 1.2*. Document-Id 32300—01-1200-01.
- Sybase-4. Sybase® Adaptive Server™ Enterprise. (1999). *What's new in Sybase Adaptive Server Enterprise?*. Adaptive Server Enterprise version 1.2. Document-Id 37429—1-1200-01.
- Sybase-5. Sybase® Adaptive Server™ Enterprise. (1999) *Reference Manual Volume 1: Building Blocks Adaptive Server Enterprise Version 12* Document ID: 36271-01-1200-01 (October 1999)
- Sybase-6. Sybase® Adaptive Server™ Enterprise. (1999) *Reference Manual Volume 2: Commands Adaptive Server Enterprise Version 12* Document ID: 36272-01-1200-01 (October 1999)
- Sybase-7. Sybase® Adaptive Server™ Enterprise. (1999) *Reference Manual Volume 3: Procedures Adaptive Server Enterprise Version 12* Document ID: 36271-01-1200-01 (October 1999)
- Sybase-8. Adaptive Sybase Anywhere Reference. (2000) [On Line] Available : <http://download-europe.sybase.com/pdffdocs/awg0702e/dbrfen7.pdf>.
- Thalheim, B. (1996). *An overview on semantical constraints for database models*. In *Proceedings of 6th. International Conference on Intellectual Systems and Computer Science*. Moscow, Russia.
- Türker, C., Gertz, M. (2001). *Semantic integrity support in SQL-99 and commercial (Object-) relational database management systems*. To appear in *VLDB Journal*.
- Van den Berghe, T. (1999). *A methodological framework for active application development*. Ph. D. Thesis. Université Catholique de Louvain. Belgium.
- Zaniolo, C. et. al. (1997). *Advanced Database Systems*. Morgan Kauffman Publishers, Inc.

Chapter III: Preserving Relationship Cardinality Constraints in Relational Schemata

Dolores Cuadra, Carlos Nieto, Paloma Martínez, Elena Castro, Manuel Velasco,
Universidad Carlos III de Madrid,

Spain

INTRODUCTION

Database modelling is a complex task that involves conceiving, understanding, structuring and describing real Universes of Discourse (UD) through the definition of schemata using abstraction processes and data models. To face this problem, methodologies that incorporate intelligent assistance are required. Some current methodologies only provide some recommendations or heuristics while others give well established and formalised processes. Traditionally, three phases are identified in database design: conceptual, logical and physical design. Conceptual modelling phase

represents the most abstract level since it is independent of any database management system (DBMS) and, consequently, is very close to the user and allows him to collect almost completely the semantics of the real world to be modelled.

A conceptual schema, independent of the data formalism used, plays two main roles in the conceptual design phase: a *semantic* role, in which user requirements are gathered together and the entities and relationships in a UD are documented, and a *representational* role that provides a framework that allows a mapping to the logical design of database development. Three topics are involved in the database conceptual modelling process: data modelling formalism, methodological approach and CASE tool support. One of the most extended data modelling formalisms, the Extended Entity Relationship (EER) model has proven to be a precise and comprehensive tool for representing data requirements in information systems development, mainly due to an adequate degree of abstraction of the constructs that it includes. Although the original ER model was proposed by Chen (1976), many extensions and variations as well as different diagrammatic styles have been defined (Hull & King, 1987; McAllister, 1998; Peckhan & Maryanski, 1988).

In database conceptual analysis, one of the most difficult concepts to be modelled are relationships, especially higher order relationships, as well as its associated cardinalities. Some textbooks (Boman et al., 1997; Ullman & Widom, 1997) assume that any conceptual design can be addressed by considering only binary relationships since its aim is to create a computer oriented model. We understand the advantages of this approach although we believe that it may produce certain loss of semantics (some biases are introduced in user requirements) and it forces us to represent information in rather artificial and sometimes unnatural ways.

Concerning the logical design, the transformation process of conceptual schemata into relational schemata should be performed trying to completely preserve the semantics included in the conceptual schema; the final objective is to keep the semantics in the database itself and not in the applications accessing the database. Nevertheless, sometimes a certain loss of semantics is produced; for instance, foreign key and not null options in the relational model are not enough to control ER cardinality constraints.

This chapter is devoted to the study of the transformation of conceptual into logical schemata in a methodological framework focusing on a special ER construct: the relationship and its associated cardinality constraints. The section entitled "EER Model Revised: relationships and cardinality constraint" reviews the relationship and cardinality constraint constructs through different methodological approaches to establish the cardinality constraint definition that will be followed in next sections. The section "Transformation of EER Schemata into Relational Schemata" is related to the transformation of conceptual n -ary relationships ($n \geq 2$) into the relational model following an active rules approach. Finally, several practical implications as well as future research paths are presented.

EER MODEL REVISITED: RELATIONSHIPS AND CARDINALITY CONSTRAINTS

This section reviews entity, relationship and cardinality constraint constructs of different data models in order to highlight some special semantic problems derived from the different methodological approaches given to them. The EER model (Teorey, Yang & Fry, 1986) is considered as the basic framework to study the different meanings of cardinality constraints. The objective is to make a profound study of the different cardinality constraints definitions as well as the implications of their usage.

Basic Concepts: Entities, Relationships and Cardinality Constraints

The central concepts of the ER model are entities and relationships. These constructs were introduced by Chen (1976) and have been incorporated in other conceptual models, although with different names^[1] : class, type, etc., for entities and associations for relationships. Nevertheless, those concepts do not have precise semantics, and consequently, it is necessary to fix their meaning.

In spite of the entity concept being widely used and accepted, there is no agreement on a definition; for instance, Thalheim (2000) collects twelve different entity denotations. Although experts are not able to give a unique definition, the underlying concept is coincident in all of them, and its usage as design element does not suppose great disadvantages. An entity definition is not given here just to highlight, according to Thalheim (2000) that an entity is a *representation* abstraction with modelling purposes. Date (1986) adds that the represented concept is an distinguishable object, but we do not consider this feature as essential because it depends on the designer point of view.

The relationship concept is more confusing; it is defined as an *association* among entities. This definition offers many interpretations; for instance, in several design methods, there are some differences depending on whether relationships can participate in other relationships, as in HERM, (Thalheim, 2000), by means of association entities as in UML, OMG (2000), or by grouping as clusters a set of entities and relationships (Teorey, 1999). These differences are due to the fact that a relationship combines *association* features with *representation* features and therefore might be considered a relationship (if association aspects are highlighted) or an entity (if representation aspects are emphasised). For instance, a marriage can be seen as a relationship (association between two people) or as an entity (representation of a social and legal concept), and both of them are possible. This duality is a source of design problems.

Previous comments are based on several experiments described in Batra & Antony (1994) and Batra & Zanakakis (1994) proving that novice designers do not have any difficulty in representing entities and attributes because they are simple concepts and easily identifiable from specifications. However, the identification of relationships and their properties is more complicated. They argue that the big number of combinations for a

given set of entities is an obstacle in detecting relationships and, consequently, more design errors appear.

Apart from these definitions, the linguistic level applied to these concepts is very important; it is required to distinguish between entity/relationship and occurrences of an entity/relationship. In the first case, there is an algebraic, or abstract data, type perspective that groups a set of possible values that are being represented (or associated) and, in the second case, an occurrence references a specific value of an entity or relationship.

Finally, depending on the number of entities related we distinguish binary relationships if they associate two entities and higher order relationships if they associate three or more entities.

The next section explains how the entity occurrences can be combined in a relationship and how cardinality constraints add more semantics to the relationship definition.

Cardinality Constraint Characterisation

Cardinality constraint is one of the most important restrictions that can be established in a conceptual schema. Its functionality is to limit the number of entity occurrences that are associated in a relationship, i.e., that participate in a relationship. The most commonly employed limits are a lower bound (minimum cardinality) and an upper bound (maximum cardinality), although other cardinality constraints are possible (for example, that occurrence participation follows a predetermined statistical distribution). In spite of a simple concept, the definition of this constraint admits several variants. Without being exhaustive, the following cardinality constraint approaches are presented.

Let E_i and E_j ($i \neq j$) be entities that are linked by the relationship I and E_i the entity on which the cardinality constraint is being defined. If I is a binary relationship it only will associate two entities (E_1 and E_2 , then $i=1$ and $j=2$) and there are the possibilities given below:

- From the I viewpoint:
 - Def. 1: Number of times that an occurrence of E_i appears in the relationship I (it is called *participation*)
- From the E_i viewpoint:

- Def. 2: Number of occurrences (any) of E_2 that can be related to an occurrence of E_1 (*lookup*).
 - Def. 3: Number of times that an occurrence (fixed) of E_2 can be related to an occurrence of E_1 (*individual lookup*).
- From the E_2 viewpoint:
 - Def. 4: Number of occurrences (any) of E_1 that can be related to an occurrence of E_2 (*lookacross or Chen's style*).
 - Def. 5: Number of times that an occurrence (fixed) of E_1 can be related to an occurrence of E_2 (*individual lookacross*).

There are subtle differences and similarities between these definitions that may be clarified by an example. Let us suppose the relationship I of [Figure 1a](#); supposing there are not multivalued attributes^[2] in the relationship [Figure 1b](#) shows all possible occurrences^[3] of the relationship I for E_1 and E_2 entities. Note that according to the definition of the relationship presented above (Definition 1) this table represents the maximum possible extension at any time of the relationship I .

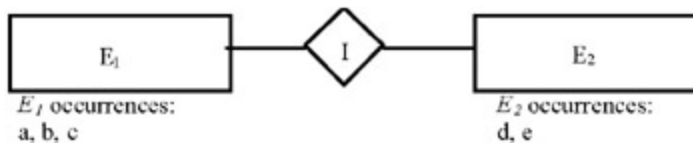


Figure 1a: Binary relationship example

Figure 1b: All possible occurrences of relationship I of [Figure 1a](#)

Any constraint established on the relationship will prevent the appearance of some of the presented occurrences in [Figure 1b](#). Especially, cardinality constraint taking as basis the definition 1 applied to E_1 restricts the number of times that any occurrence of E_1 appears in [Figure 1b](#); that is, the upper resulting value of counting the number of x (from E_1) appearing in each column will not exceed the established maximum cardinality nor will the lower value be inferior to the established minimum cardinality. Note as the principal difference with the other definitions, that the limit over the occurrences of E_1 is established independently of E_2 .

Definition 2 supposes to count for any x (from E_1) the number of y (from E_2) that appear in each column and to keep the maximum or minimum of the resulting values. Definition 3 supposes to count for any x (from E_1) the number of y (from E_2) with each one of the y values that appears in each column. Thus, taking into account the occurrences presented in [Figure 1b](#), cardinality values of E_1 calculated as shown in [Figure 2](#) are two according to Definition 2 and one according to Definition 3.

Def. 1	Cardinality Value
a appears twice in the interrelationship	2
b appears twice in the interrelationship	2
c appears twice in the interrelationship	2
Def. 2	
a can appear once with d and once with e	2 (1+1)
b can appear once with d and once with e	2 (1+1)
c can appear once with d and once with e	2 (1+1)
Def. 3	
a can appear once with d	1
a can appear once with e	1
b can appear once with d	1
b can appear once with e	1
c can appear once with d	1
c can appear once with e	1

Figure 2: Calculus of different cardinality values from [Figure 1b](#)

Definitions 4 and 5 will only exchange the role of E_2 and E_1 . Note that the values obtained with Definition 3 and Definition 5 are the same ones, since the occurrences of both entities are fixed.

For higher order relationships, we will have among others the following possibilities, taking into account that E_i and E_j are defined as before.

- From the I viewpoint:
 - Def. 1: Number of times that an occurrence of E_i appears in the relationship (*participation*)
- From the E_i viewpoint:
 - Def. 2: Number of times that any occurrence of each E_j appears related to an occurrence of E_i .
 - Def. 3: Number of times that a fixed occurrence of each one of the entities E_j appears related to an occurrence of E_i .
 - Def. 4: Number of times that any combination^[4] of occurrences of the entities E_j appears related with an occurrence of E_i (*lookup*).

- Def. 5: Number of times that a specific combination of occurrences of the entities E_j appears related with a occurrence of E_i (*individual lookup*).
- From the E_j viewpoint:
 - Def. 6: Number of any occurrence of E_i that can participate in the relationship with an occurrence of E_j .
 - Def. 7: Number of times that a specific occurrence of E_i can be seen in the relationship with an occurrence of E_j .
 - Def. 8: Number of any occurrence of E_i that can be seen in the relationship with a combination of occurrences of the entities E_j (*lookacross* or *Chen's style*).
 - Def. 9: Number of times that a specific occurrence of E_i can be seen in the relationship with a combination of occurrences of the entities E_j (*individual lookacross*).

Note that binary relationships are particular cases of the higher order relationships. Moreover, for higher order relationships, Definitions 2, 3, 6 and 7 are included as particular cases of Definitions 4, 5, 8 and 9, respectively, when the combination is restricted to only one entity. The next example will clarify the differences among these definitions.

Let us suppose the ternary relationship I presented in the [Figure 3a](#) in which the occurrences of the entities are also indicated. [Figure 3b](#) contains the possible occurrences of the relationship I obtained from the occurrences of the E_1 , E_2 and E_3 entities^[5]. The maximum number of occurrences in the relationship will be considered in absence of any constraint, and their extension will be the Cartesian product of the sets of occurrences of the entities that participate in the relationship.

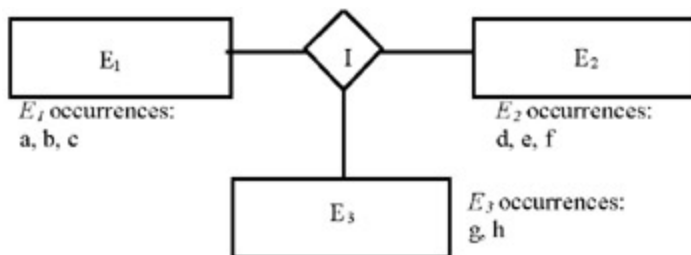


Figure 3a: Ternary relationship example

{a d g,	b d g,	c d g,
a d h,	b d h,	c d h,
a e g,	b e g,	c e g,
a e h,	b e h,	c e h,
a f g,	b f g,	c f g,
a f h,	b f h,	c f h}

Figure 3b: All possible occurrences of relationship *I* of [Figure 3a](#)

The analysis of the first definition is similar to the one carried out for the binary relationships, that is, a constraint on the number of occurrences in each column of [Figure 3b](#). In order to study the rest of the definitions, the cardinality constraint of the entity E_1 will be calculated and, for the sake of simplicity it will be supposed that it is calculated with regard to the combination of entities formed only by entity E_2 . This supposition will force Definition 2 coincident with Definition 4, Definition 3 with Definition 5, Definition 6 with Definition 8 and Definition 7 with Definition 9.

In Definition 2 to Definition 5, cardinality constraint is obtained by fixing an x (from E_1) and then counting the number of occurrences of E_2 that appear in [Figure 3b](#). Therefore, Definition 2 and definition 4 for each x (from E_1) fixed count the number of y (from E_2) that appear in each column and keep the maximum and the minimum. Definition 3 and definition 5 fix an x (from E_1) and an y (from E_2), so that for each y (from E_2) the number of times that it is combined with an x (from E_1) is counted keeping the values maximum and minimum as cardinality values. Finally, notice that the Definition 6 to Definition 9 would only exchange the role of E_2 and E_1 . The values obtained with the Definition 3 and Definition 7 and with the Definition 5 and Definition 9 are the same, since the occurrences of both entities are fixed.

Furthermore, in absence of multivalued attributes in the relationship and establishing that the set of occurrences of each entity is limited, the maximum cardinality constraint is limited by the cardinal of the Cartesian product of the rest of entities (6 in the example), although in the practice the value n is used to avoid specifying a maximum value.

There are other proposals for cardinality constraints. Jones & Song (1998), Elmasri & Navathe (1994) and Ramakrishnan (1997) present an approach that combine the maximum cardinality proposed by Chen (1976) with participation semantics (optional/mandatory) for the minimum cardinality.

In McAllister (1998) and Thalheim (2000), a systematic approach for the cardinality constraint definition is presented. McAllister (1998) shows a tabular representation that allows collecting maximum and minimum cardinalities of any combination of entities with any other combination. Thus, many of the aforementioned definitions can be comprised in an unique formalism; however, the number of cardinality constraints to be defined is very high (e.g., with three entities it would be necessary to define 12 pairs,

with four entities, 50, and so on). To facilitate this task, a set of rules is provided. On the other hand, Thalheim (2000) analyses different variants of cardinality constraints and proposes a general cardinality constraint that subsumes as particular cases all the previously presented definitions. Nevertheless, the implications of each in the design process are hardly explained, although it is important to underline that the HERM data model implements a cardinality constraint with participation semantics.

Main Cardinality Constraint Approaches

In this section, the most extended data models with their corresponding cardinality constraint approaches are studied. Two main approaches are discussed: first, the Chen's style constraint that is one extension of the mapping constraint (a special case of cardinality constraint that considers only the maximum cardinality and that for binary relationships can be 1:1, 1:N or N:M), (Chen, 1976), that different data models and methodologies have adopted or extended and, secondly, the MERISE approach, (Tardieu, 1989), that incorporates the participation semantics. Concerning the aforesaid definitions for higher order relationships they are related to Definition 1 and Definition 8, respectively.

These two approaches meet when cardinality constraints for binary relationships are defined (excepting the natural differences in graphical notations). Both represent the same semantics in binary relationships although the way of expressing them is different.

Binary Relationships

[Figure 4](#) shows an example of cardinality constraint over a binary association using UML notation (OMG, 2000); it is called multiplicity constraint and it represents that an employee works in one department (graphically denoted by a continuous line) and that at least one employee works in each department (graphically denoted by a black circle with the tag +1). Minimum multiplicity of one in both sides forces all objects belonging to the two classes to participate in the association. Maximum multiplicity of n in *Employee* class indicates that a department has n employees and maximum multiplicity of 1 in *Department* means that for each employee there is only one department. Consequently, UML multiplicity follows Chen's style because to achieve the cardinality constraints of one class it is needed to fix an object of the other class and obtain the number of objects related to it.



Figure 4: Cardinality constraints using UML

[Figure 5](#) illustrates the same example but using MERISE methodology (Tardieu, 1989); cardinality constraints represent that an occurrence of the *Employee* entity participates once in the *Works* relationship and an occurrence of the *Department* entity participates at least once in the relationship.



Figure 5: Cardinality constraints using MERISE

Notice that both examples represent the same semantics although expressed in different ways. [Figure 6](#) shows Chen's style notation of the *Works* relationship. Comparing it to MERISE notation, cardinality tags are exchanged (in MERISE notation, cardinality tag is situated near the constrained entity and in Chen's notation, the cardinality tag is located in the opposite ending). This difference reflects the distinct perspective adopted by these methodologies: MERISE methodology constrains the *participation of an entity* in the relationship and Chen methodology limits the *participation of a combination of the other entity(ies)* with an entity in the relationship. Thus sometimes a conceptual schema could be misunderstood if it has been created using another methodology. [Figure 7](#) shows the same constraint expressed in the Teorey notation, (Teorey, Yang & Fry, 1986), the shaded area represents a maximum cardinality of n . With this graphical notation it is only allowed maximum cardinalities of 1 or n and minimum cardinalities of 0 or 1.

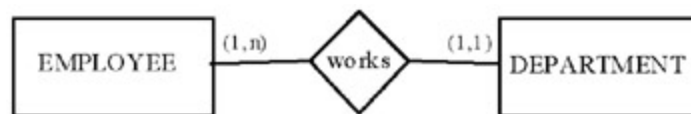


Figure 6: Cardinality constraints using ER model



Figure 7: Cardinality constraints using Teorey model

[Table 1](#) gives a summary of the aforementioned concept for binary relationships.

Table 1: Cardinality constraints summary		
	Minimum Cardinality	Maximum Cardinality
0	<i>Optional</i>	<i>Inapplicable</i> : There are no occurrences in the relationship
1	<i>Mandatory</i> : It is mandatory that all occurrences of entity <i>A</i> participate in the relationship (there is at least one occurrence of entity <i>B</i> related to each occurrence of entity <i>A</i>).	<i>Determination</i> ^[6] or <i>Uniqueness</i> : There is at most an occurrence of entity <i>B</i> related to each occurrence of entity <i>B</i> .
k (>1)	<i>k-Mandatory</i> : It is mandatory that each occurrence of entity <i>A</i> participates at least <i>k</i> times in the relationship (there are at least <i>k</i> occurrences of entity <i>B</i> related to each occurrence of entity <i>A</i>).	<i>k-Limit</i> : There are at most <i>k</i> occurrences of entity <i>B</i> related to each occurrence of entity <i>A</i> .
N	<i>Without limit of minimum participation</i> .	<i>Without limit of maximum participation</i> .

Table 1: Cardinality constraints summary

	Minimum Cardinality	Maximum Cardinality
[6]	This concept is translated from the relational model into a functional dependency that can be used in refining the relational schema.	

Higher Order Relationships

In database conceptual modelling, binary relationships are the most frequently used. In fact, there are several data models that only allow this kind of relationships, see NIAM (Nijssen & Halpin, 1989). That is why most methodologies, see MERISE (Tardieu, 1989), OMT (Rumbaugh, Blaha & Premerlani, 1991), UML (OMG, 2000) and Teorey (1999), do not put the emphasis on n -ary relationships ($n > 2$). Although higher relationships are not so common, sometimes it is not possible to completely represent the UD using binary relationships; for instance, to represent the database requirement: "*it is required to know the programming languages used by the employees in the projects they are working,*" a ternary relationship would reflect this requirement^[7], while three binary relationships would not be able to represent the whole semantics (Figure 8); the combination of binary relationships *Works*, *Uses* and *Requires* neither allow to know the programming languages that a specific employee uses in a specific project nor to know the projects in which a specific employee is working with a specific programming language.

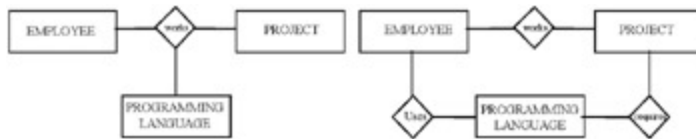


Figure 8: Ternary relationships versus binary relationships (first solution)

The suggested solution provided by the data models that exclusively consider binary relationships is to transform the higher order relationship into one or more entities and to add binary relationships with the remaining entities, (Ullman & Widom, 1997; Boman et al., 1997). Figure 9 shows this solution where an entity *Task* is introduced; three binary relationships connect entity *Task* with *Employee*, *Project* and *Programming Language* entities. The principal advantage of this approach is that is nearer to relational model and thus, closer to implementation. However, this approach implies to include entities that are not explicitly exposed in the UD and to add complex constraints to keep the correct semantics. With these models, designer perspective is conditioned and the conceptual schema obtained could result in an artificial schema.

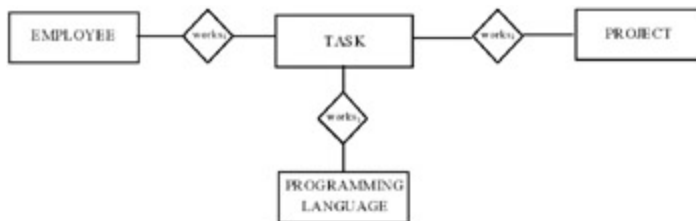


Figure 9: Ternary relationships versus binary relationships (second solution)

Keeping in mind that higher order relationships are necessary in database conceptual modelling, several methodologies have generalised the cardinality constraint definition of binary relationships (in the two approaches previously commented), raising some difficulties that are explained below.

First, there is an inconsistency problem, depending on the adopted approach, because higher order relationships do not represent the same semantics that binary relationships. [Figures 10](#) and [11](#) represent Chen and MERISE cardinality constraints, respectively, for the semantic constraint: "an employee works in several projects and (s)he could use a programming language in each of them".

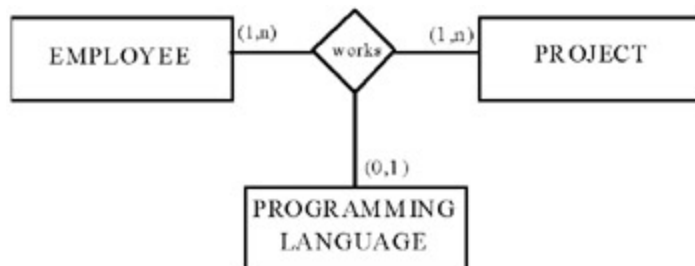


Figure 10: ER cardinality constraints using Chen's style

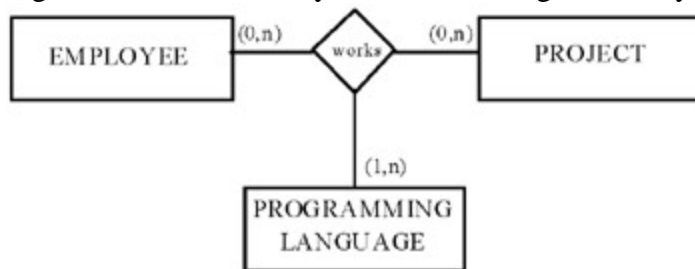


Figure 11: ER cardinality constraints using MERISE approach

In the Chen approach the cardinality constraint of an entity depends on the remaining entities that participate in the relationship, thus, there are several problems in order to scale up his definition from binary to higher order relationships since the remaining entities could be combined in different ways. However, the most frequent generalisation determines that a combination of all remaining entities is used in specifying the cardinality constraints of only one entity. Therefore, using the ER model notation in order to obtain the cardinality constraint of *Programming Language* entity ([Figure 10](#)) it is needed first to fix two occurrences of *Employee* and *Project* entities that are related by *Works* relationship and then to count the number of times (minimum and maximum) that occurrences of *Programming Language* entity could appear related to. Next, the same procedure is applied to *Project* entity (with pairs of *Programming Language* and *Employee* occurrences) and to *Employee* entity (with pairs of *Project* and *Programming Language* occurrences).

[Figure 10](#) illustrates that minimum cardinality of *Programming Language* entity is 0, that is, there are occurrences of *Works* relationship that associate occurrences of *Employee* and *Project* entities but with no occurrence of *Programming Language*^[8]. This circumstance causes problems in identifying the relationship occurrences. The

relationship is capable of representing occurrences with unknown information (the case of *Programming Language* in the example). However, this interesting capability (exclusive of Chen's style) will again allow us to review the meaning of the relationship concept.

In summary opposed to MERISE methodology, Chen's style presents three main troubles: generalisation, difficulty of the treatment of unknown information in relationships and the lack of information about the participation of each occurrence of associated entities.

In contrast, generalisation of cardinality constraint definition in MERISE methodology does not pose any problem because the semantics of cardinality constraints is the same in binary and higher order relationships. [Figure 11](#) illustrates an example of the *Works* ternary relationship; the cardinality constraint of *Programming Language* is obtained by counting the number of appearances of a specific *Programming Language* occurrence in the *Works* relationship. The cardinality constraints of *Employee* and *Project* entities are obtained in the same way.

Additionally, MERISE methodology includes a new construct called Functional Integrity Constraint (CIF^[9]) that represents one of the participating entities is completely determined by a combination of the other entities (an example is shown in [Figure 12](#)). Moreover, these CIF have many implications in decomposing higher order relationships as well as in transforming them into the relational model.

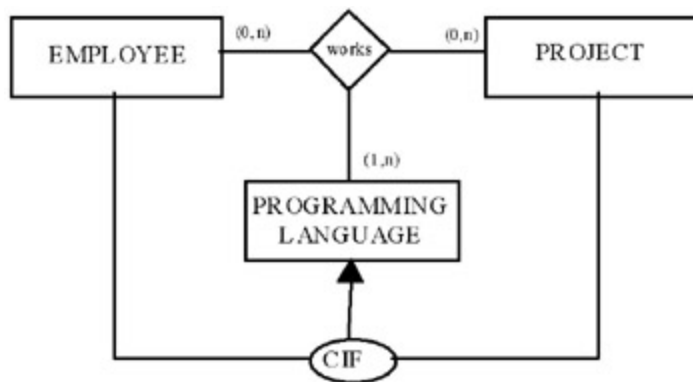


Figure 12: An example of functional integrity constraint in MERISE

Therefore, the MERISE approach has two constructs to represent cardinality constraints while the Chen approach only uses one. On the other hand, CIF constraints do not satisfactorily resolve the treatment of unknown information. Finally, minimum cardinality constraint in MERISE approach represents optional/mandatory participation^[10] and, thus, maximum cardinality constraint typically will be n .

[Table 2](#) (Soutou, 1998), shows the differences between the two approaches for cardinality constraints when higher order relationships are transformed into the relational model.

Table 2: Summary of the differences among cardinality constraints

Cardinality	Models based on the ER model (MER)	Models based on Participation Constraint (MPC)
Min 0	Presence of NULL values	No constraint
Min 1	No constraint	An occurrence of the entity relation cannot exist in the n-ary relationship without being implicated in one occurrence
Min (n)	For each (n-1) record there are at least more than one occurrences for the other single in the n-ary relationship	An occurrence of the entity relation cannot exist in the n-ary relationship without being implicated in many occurrences
Max 1	For each (n-1) record there is a unique occurrence for the other single column in the n-ary relationship	Unique value for the column (No duplicates)
Max (n)	For each (n-1) record there is more than one occurrence for the other single column in the n-ary relationship	No constraint

Adopting a Cardinality Constraint Definition

The decision about adopting a cardinality constraint definition has several theoretical and practical consequences. Let the ternary relationship that represents the requirement *"There are writers that write books that may be concerning different topics"*. [Figure 13](#) shows the conceptual schema solution using the MERISE definition with the next interpretation: there may be occurrences of all entities that do not participate in the occurrences of *Writes* relationship. In this way, less semantics is represented by the cardinality constraints because it is not known how the participation of any of *Author*, *Book* or *Topic* entities in the relationship affects the participation of the remaining entities. Moreover, it is impossible to represent that there can be anonymous books.

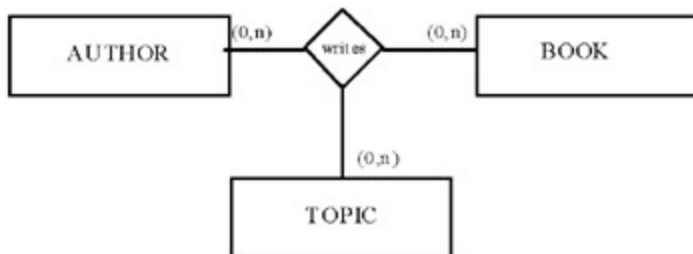


Figure 13: Ternary relationship using MERISE cardinality constraints

From Chen's perspective, the example could be modelled in two semantically equivalent ways ([Figures 14](#) and [15](#)) both of them considering anonymous books. The first solution is more compact although more complex; a unique ternary relationship collects the

association semantics. The second solution is more intuitive because the ternary relationship reflects the books whose authors are known and the binary relationship that represents the anonymous books. The choice of a final solution depends on the designer perspective about the UD.

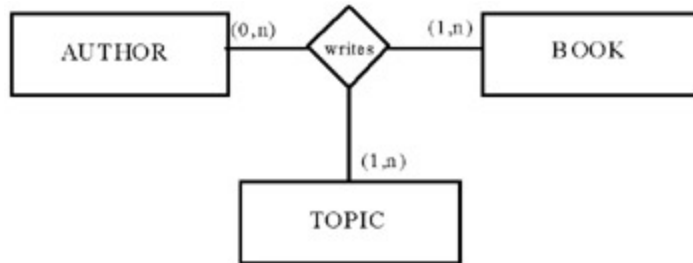


Figure 14: Ternary relationship using Chen's style (first solution)

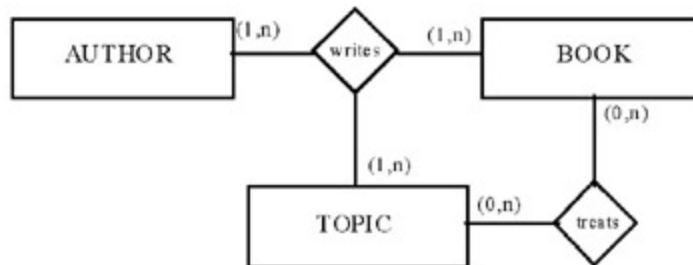


Figure 15: Ternary relationship using Chen's style (second solution)

Both solutions imply that it is necessary to check that a specific book does not simultaneously appear as an anonymous book and a book with a known author. In the first solution, this constraint could be modelled as an exclusivity constraint among the occurrences of the ternary relationship, while in the second solution, it could be modelled as an exclusive-or constraint between the two relationships^[11], that is, the pair book-topic present in the ternary relationship occurrences does not appear in any binary relationship occurrence in the case that the model includes this kind of constraint.

In this chapter, we adopt the Chen's style concerning the cardinality constraints and the ER model as the basic model. Chen notation is able to reflect the functional dependencies as well as incomplete information as we have seen in the previous examples while MERISE cardinality constraints are less restrictive. Before using it a review of the incomplete information concept is given in the next section.

Implications in the Definition of Relationships: Handling Incomplete Information

There is a wide variety of literature about the representation and treatment of incomplete information in the relational model; Van Der Meydem (1998) gives a complete revision from a logical perspective of this subject. This kind of information has not been deeply studied in conceptual modelling. In ER based models, traditionally, optional attributes have been allowed both in entities (except for attributes that compose the entity identifier^[12]) and relationships in order to treat unknown information, although other

proposals Wand, Storey & Weber (1999) argue that the usage of optional attributes contradicts the ontological foundation of entities and relationships.

Sometimes, it is not possible to cover completely the semantics of a UD when designing a schema. So it is necessary to reflect the incomplete information, although it is under discussion if a conceptual model should provide mechanisms to represent this kind of information (implicitly or explicitly as it has been seen in the two approaches for cardinality constraints of previous sections).

A traditional distinction of incomplete information has to do with the source of incompleteness. The *unknown information* source ("it exists a value but it is unknown") and the *undefined information* source ("it does not exist a value" or equivalently "value is inapplicable"). Also a hybrid of these, *no information*, has been considered ("either it exists an unknown value or it is inapplicable"). The perspective adopted in this section only covers the unknown information case. We believe that a conceptual model does not have to contemplate other sources of incompleteness, since the origin of an inapplicable attribute is always an inadequate conceptual design that does not fully reflect the UD requirements. It can be argued that by practical reasons it may be necessary to treat with inapplicable attributes in order to not complicate a conceptual scheme reflecting a exceptional situation, but at the conceptual level, it always exists as an equivalent solution that uses optional attributes or introduces new entities to reflect the exceptional cases.

By observing the two solutions of [Figures 14](#) and [15](#), there are important methodological implications depending on the solution selected, apart from the designer preferences. It is necessary to highlight that the first solution ([Figure 14](#)) allows the existence of relationship occurrences that do not associate occurrences of the three entities but only associating *Book* and *Topic* occurrences; this issue contradicts what many authors consider as a relationship. The cause of this contradiction has its origin in the strong influence of the relational model. Relational model is a logical model (not conceptual) with mathematical foundations. In this model, relations are defined as subsets of the Cartesian product of attribute domains. Many conceptual models define the relationships in a similar way and, consequently, there are no unknown relationship occurrences. So, the next relationship definition has been widely used.

Definition 1: Relationship in complete universes of discourse with information completely known.

Let E be an entity and $Ext^t(E)$ be its *extension at time t* (set of occurrences that compose it in the instant t , supposedly a discrete line of time).

The *extension at time t of a relationship I* among n entities (E_1, E_2, \dots, E_n) , that could not be all different entities, $Ext^t(I)$, is a subset of the Cartesian product of the entity extensions at time t such that its elements verify some predicate that defines this relationship^[13]. So, the next property holds:

Finally, the relationship I as the transfinite union of their extensions at any time is defined.

This definition has a great advantage: the subsequent transformation into the relational model is direct. The relationship is translated, in the most general case, into one relation whose primary key is the combination of the foreign keys referencing the participating entities in the original relationship. So, the entity integrity of the relational model (Codd, 1979), is guaranteed because no null values in the primary key of a relation are allowed. In contrast, the previous definition does not contemplate the probable presence of unknown information in relationships. In this way, this definition of relationship would be inconsistent with the cardinality constraint definition of the Chen approach. Adopting this relationship definition would supposedly not have minimum cardinalities of 0 in higher order relationships.

Incomplete information has been captured in ER based models mostly via optional attributes. The previous discussion shows that merely with optional attributes it is not possible to reflect the presence of unknown information in the relationships. This situation is opposite to what happens in the relational model, where optional attributes alone allow one to design logical schemata that reflect the presence of unknown information in any possible situation. It is important to highlight this fact because in the ER model there are two basic constructs, while in the relational model there is only one.

The next relationship definition is consistent with Chen's cardinality constraint definition and also it allows conceptual models to represent unknown information. To support unknown information, an expansion of extensions of entities, $Ext(E)$, is required; it will be denoted as $bottom^{[14]}(?)$.

Definition 2: Relationship in incomplete Universes of Discourse with unknown information.

Let E be an entity and $Ext'^t(E)$ be its *lifted extension at time t* defined as $Ext'^t(E) = Ext^t(E) \cup ?_E$. Notice that each $?_E$ symbol has to be specific for each entity E although we omit the entity it applies to.

The *lifted extension at time t of a relationship I* among n entities (E_1, E_2, \dots, E_n) , that could not be all different entities is a subset of the Cartesian product of the entity lifted extensions at time t such that its elements verify some predicate that defines this relationship^[15]. The tuple totally composed of unknown information is excluded, although it could be interesting in a higher order model to retain it. So, the next property holds:

$$\forall t, Ext'^t_{\perp}(I) \subseteq Ext'^t_{\perp}(E_1) \times Ext'^t_{\perp}(E_2) \times \dots \times Ext'^t_{\perp}(E_n) \setminus (\perp_{E1}, \perp_{E2}, \dots, \perp_{En})$$

Finally, the *lifted relationship* I as the transfinite union of their lifted extensions at any time is defined.

If an in-depth study is performed and also a partial ordering relation (degree of information definition) among the relationship occurrences is included, the ER model could be seen as a theory of algebraic domains, as it is proposed by Buneman et al. (1991) for the relational model. In the same way, a conceptual modelling process could be considered as a construction of algebraic domains and a conceptual schema could be viewed as the specification of the structural component of an abstract data type, facilitating the integration of programming languages and databases.

Note that this lifted relationship definition could pose some counter-intuitive conclusions; for instance, in the existence of incomplete occurrences in binary relationships; occurrences composed of occurrences of one of the entities^[16] are allowed. Nevertheless, the usual interpretation of Chen's cardinalities for binary relationships excludes this possibility^[17]. Notice also that a N:M:P ternary relationship could have occurrences composed of occurrences of only one entity (at most N), two entities (at most N*M) or three entities (at most N*M*P). Nevertheless, the usual interpretation of Chen's cardinalities express only occurrences of the last two types.

If a conceptual model uses this definition of lifted relationship, the process of transforming conceptual schemata into relational schemata complicates the matter since the presence of unknown information has to be solved. The approach shown in the next section considers this concept of relationship as well as the extension of Chen's cardinality constraint in order to allow a more abstract design process and postponing to the transformation step the set of problems concerning semantic preservation in logical design.

The two definitions of relationship presented here are often indistinct and interchanged, resulting in potential confusion. In these definitions we have differentiated between non-lifted and lifted relationships although in the rest of the chapter we will use the term *relationship* to always mean lifted relationship with the Chen's cardinality constraint.

In equality based models, like the relational, there is no difference between two occurrences with the same values (including the bottom) and this issue will prevent the handling of unknown information. Some values could be utilised to represent two different real world objects, and thus there would be a collision of these two objects in only one representation. Roughly speaking one indefinite occurrence represents many real world occurrences. However, non-equality-based models could differentiate two identical occurrences of the same entity or relationship because they differentiate each occurrence of the bottom symbol by using a partial ordering relation among the tuples or using alternative identification mechanisms such as surrogate keys. These aspects are common to any model that allows us to reflect unknown information, for example, many extensions to tables of relational models have been developed for allowing a better treatment of incomplete information than the simple replacement by nulls of unknown attributes.

In the example shown in the section "[Adopting a cardinality constraint definition](#)", we will adopt the first solution ([Figure 14](#)) as the most adequate choice, while the second solution ([Figure 15](#)) will be considered in the next section as a valid design option as well as an intermediate step for transforming relationships into the relational model by eliminating the unknown information from the original higher order relationship.

Some Conclusions

After reviewing ER constructs, it seems necessary to deepen the definition, foundations, constructs and notation of the ER model to achieve a conceptual tool able to reflect the situations that frequently appear in data modelling scenarios. Hence, the redefinition of the ER model, taking into account previous aspects as well as the development of a wider notation, are tasks to be dealt with.

The detection and specification of abstractions in an UD that lead to correct and complete schemata are critical problems that combine psychological and methodological aspects. There are many other aspects associated with the specification of constraints. Their identification and validation require more formal treatments. Identification can be faced to with a lexical analysis of a problem description. Syntactic and semantic validation of relationships is a critical aspect of internal coherence with the UD. All these topics are not analysed in this chapter.

In this section a set of problems related to cardinality constraints and its influence on the other ER constructs has been analysed and important inconsistencies concerning this kind of constraint have been highlighted. The next section explains some topics involved in how to transform relationships into the relational model while trying to preserve the original ER semantics.

[1] From now on, we will use the original names (entity and relationship).

[2] The presence of a multivalued attribute in a relationship implies that the identification of its occurrences has to consider this attribute.

[3] Possible occurrences of the relationship with incomplete information are not considered.

[4] Multiple combinations of entities are possible; for example, in n -ary relationships ($n \geq 2$), combinations of only one entity up to combinations of $n-1$ entities are possible. In this sense, the definitions that consider combinations do not define a unique possibility, but many (as many as possible combinations).

[5] The possible occurrences of the relationship with incomplete information are not considered.

[7] We suppose that there are not additional semantic constraints between the entities participating in the ternary relationship of type: "an employee works exactly in one department," "an employee uses one programming language," etc.

[8] Maximum cardinality of 1 in Programming Language expresses a functional dependency: employee, project ? programming language.

[9] In French, contrainte d'intégrité fonctionnel.

[10] While Chen's approach is not able to express optional/mandatory participation, it represents functional dependencies.

[11] This exclusive-or constraint is not directly established between the two relationships, because one is a ternary relationship while the other is a binary one. It is established between the binary relationship and an algebraic projection of the ternary relationship.

[12] Entity integrity constraint (defined in the relational model and generalized to the ER model) disallows optional attributes as part of an entity identifier; this constraint has been relaxed in the relational model (Thalheim, 1989; Levene and Loizou, 1998), by replacing the *identification* concept with the *distinguishability* (*disjunctive identification*) concept.

[13] It is not a formal and correct definition if we want to keep the relational model property that the order of attributes is irrelevant. In addition, it does not consider the existence of attributes in the relationship and suppose that all attributes in the entities are primary identifier attributes.

[14] This symbol has been used in a similar way by denotational semantics.

[15] It is not a formal and correct definition if we want to keep the relational model property that the order of attributes is irrelevant. In addition, it does not consider the existence of attributes in the relationship and suppose that all attributes in the entities are primary identifier attributes.

[16] In fact, these occurrences are pairs composed of an occurrence of one entity and the bottom symbol of the other entity.

[17] This observation poses a difficulty in attaining consistency among the model constructs that can interact with each other in unpredictable ways.

TRANSFORMATION OF EER SCHEMATA INTO RELATIONAL SCHEMATA

The major difficulty when transforming an EER schema into a schema in a logical model is information preservation. Generally, to achieve a complete mapping of both elements and their inherent and semantics restrictions from an EER model to a relational model is quite complicated. Usually, restrictions that can not be applied in the relational model must be reflected in the application programs some other way, i.e., outside the DBMS. In this way, there are several extensions to the relational model proposed by Codd (1970), Codd (1979), Date (1995), and Teorey (1999), that provides a more semantic model.

The principal transformation rules are described in most database text-books. In this section, we will show the transformation of relationships into relational model, since this construct collects more semantics than others, as entities for example.

A correct transformation of schemata and constraints expressed in them is necessary in order to preserve their intended meaning. Although initially the standard relational model (Codd, 1970) was insufficient to reflect all the semantics that could be present in a conceptual schema, it has been enhanced with specific elements that are used to preserve the original semantics. In this chapter, transformation of EER schemata into relational schemata is performed using an extended relational model with active capabilities (triggers).

To carry out the transformation of cardinalities of the EER schemata into the relational model without semantic losses the relational model provides mechanisms to express semantic constraints. These mechanisms are: the use of primary key, the use of foreign keys and its delete and update options, the use of alternative keys (UNIQUE), NOT NULL and verification (CHECKS and ASSERTIONS) constraints and triggers.

Once the concept of cardinality constraint has been established, it will be analysed in how to preserve this semantic constraint when a conceptual schema is transformed into a logical schema. This section is divided in two parts depending on the relationship type, since the semantics of the cardinality constraint is different when we deal with binary or n-ary ($n > 2$) relationships.

Binary Relationships

The analysis of binary relationships is structured in three parts: the first one concerns N:M relationships, the second deals with 1:N relationships and the third part is devoted to 1:1 relationships. This classification is due to the fact that a N:M relationship produces a new relation in a relational model while 1:N and 1:1 relationships can be transformed by propagating the primary key from a relation to another relation (Teorey, Yang & Fry, 1986; Fahrner & Vossen, 1995). For each case a solution is provided using relational model constraints and triggers.

The syntax provided for triggers is similar to the SQL3 proposal, (Melton & Simon, 1993), with the exception that procedural calls are allowed in order to interact with the user to capture the required data avoiding semantic losses.

The triggers will have the following structure:

```
CREATE TRIGGER trigger_name
BEFORE/AFTER/INSTEAD OF INSERT/DELETE/UPDATE
ON table_reference [FOR EACH ROW]
BEGIN
    trigger_body
END;
```

For data input the following procedures will be used:

- ASK_PK (table, primary_key)–this procedure obtains the primary key of the "table."
- ASK_REST (table, rest_attributes)–this procedure obtains all attributes of the relation "table," excepting those that compose the primary key.

Transformation of N:M Binary Relationships

A N:M relationship I (Figure 16) becomes a relation I , that has as primary key attributes the set of attributes of the primary keys of the entities that it associates, as it is shown in Figure 17 in which the standard transformation can be seen as it is described in the concerning literature (Date, 1995). If the relationship contains any attribute, it is included as an attribute of the new relation. If there are multivalued attributes it is necessary to study how they will be included in the primary key (Martínez et al., 2001). We study the cardinalities of one of the ends of the relationship, represented in Figure 16 by cardinality (X,n) , since the reasoning for the other end, cardinality $(_,n)$, is similar.

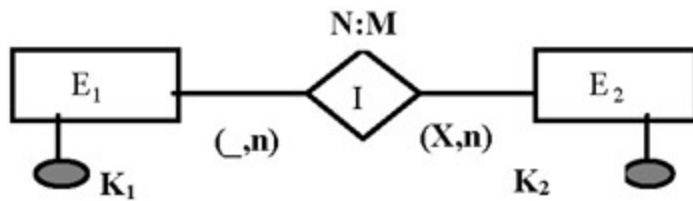


Figure 16: A N:M relationship

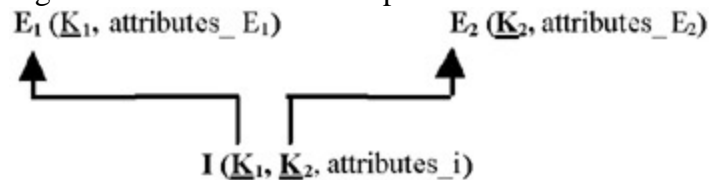


Figure 17: Relational model transformation of Figure 16

Let us study, according to minimum cardinalities, how the connection relation (I) is defined, as well as the delete and update options of the foreign key and, if it is necessary to create some trigger to control these cardinalities when insertions, deletions and updates are performed in each one of the resulting relations.

Cardinality (0,n)

Figure 16 shows the EER schema (where $X=0$) and its standard transformation is displayed in Figure 17.

Table 3 shows the analysis of the possible semantic losses when an updating is made in the relations E_1 , E_2 and I .

Table 3: Semantic loss in cardinality (o, n) updating transactions			
Relations	Updating		Is there any semantic loss?
			Why?
E_1	Insert/Delete/Update	NO	It is possible to insert/delete/update tuples into E_1 without connecting to E_2
E_2	Insert	NO	It is not a case of study
	Delete/Update	NO	It is possible to delete/update tuples from E_2 without

			connecting to E_I
I	Insert	NO	It is possible to insert tuples with no restriction
	Delete/Update	NO	It is possible to delete/update from I and to have some tuples of E_I not connected

Notice that when the minimum cardinality is 0 the participation of E_1 in I is optional, this implies that no restriction has to be added to the standard transformation since it does not exist in any semantic loss. Considering the foreign key K_1 is part of the primary key, the delete and update options of the foreign key K_1 in the relation I cannot be either SET NULL or SET DEFAULT.

Cardinality (1,n)

In this case the minimum cardinality constraint is stronger than in the previous case (see [Figure 16](#) with $X=1$); it is required that all occurrences of E_1 have to be inevitably related to one or more occurrences of E_2 . As in the previous case, in the creation of the relation I , the delete and update options of the foreign key K_2 (in a similar way to K_1), cannot be either SET NULL or SET DEFAULT.

However, foreign key options are not enough to control the minimum cardinality constraint; it is necessary to ensure that all occurrences of E_1 are related to at least one occurrence of E_2 and, therefore, we must take care that the DB is not in an inconsistent state every time that a new tuple is inserted into E_1 or a tuple is deleted from I ([Table 4](#)).

Table 4: Semantic loss in cardinality (1,n) updating transactions			
Relations	Updating		Is there any semantic loss?
			Why?
E_1	Insert	YES	It is not possible to insert tuples into E_1 without connecting to E_2
	Delete/Update	NO	It is checked by the FK delete/update option in I
E_2	Insert/Update	NO	It is not a case of study
	Delete	YES	It is not possible to delete tuples from E_2 and to have some tuples of E_1 not connected
I	Insert	NO	It is possible to insert tuples with no restriction
	Delete/Update	YES	It is not possible to delete/update from I and to have some tuples of E_2 not connected

To preserve cardinality constraints semantics (actions with "YES" in [table 6](#)), four triggers are required.

Table 6: Semantic loss in cardinality (1,n) updating transactions

Relations	Updating		Is there any semantic loss?
			Why?
E_1	Insert	YES	It is not possible to insert tuples of E_1 without connecting to E_2
	Delete/Update	NO	It is checked by the FK delete/update option in E_2
E_2	Insert	NO	It is not a case of study
	Delete/Update	YES	It is not possible to delete/update tuples from E_2 and to have some tuples of E_1 not connected

First, it will be needed a trigger that when inserting into E_1 creates a tuple in I . The two possibilities contemplated in the trigger are:

- The new occurrence of E_1 is related to an occurrence of E_2 that is already in the relation E_2 .
- The new occurrence of E_1 is related to a new occurrence of E_2 .

```

CREATE TRIGGER INSERTION_NM_(1,N)_E1
BEFORE INSERT ON E1
FOR EACH ROW
BEGIN
  ASK_PK (E2, :VK2);
  ASK_REST (I, :VREST_ATTRIBUTES_I);
  IF NOT EXISTS(SELECT * FROM E2 WHERE K2=:VK2) THEN
    BEGIN
      ASK_REST(E2, :VREST_ATTRIBUTES_E2)
      INSERT INTO E2 (K2, REST_ATTRIBUTES_E2)
        VALUES (:VK2, :VREST_ATTRIBUTES_E2)
    END;
  INSERT INTO I (K1, K2, REST_ATTRIBUTES_I)
    VALUES (:NEW.K1, :VK2, :VREST_ATTRIBUTES_I)
END;

```

To control the deletion of tuples from the relations I and E_2 and the update in I the following triggers are required to avoid that an occurrence of E_1 is not related to any occurrence of E_2 :

```

CREATE TRIGGER DELETION_NM_(1,N)_E2
BEFORE DELETE ON E2
FOR EACH ROW

BEGIN

```

```

IF :OLD.K2 IN (SELECT K2 FROM I WHERE K1 IN
              (SELECT K1 FROM I GROUP BY K1 HAVING
               COUNT(*)=1))
THEN ROLLBACK (*WE UNDO THE TRANSACTION*)
END;

```

```

CREATE TRIGGER DELETION_NM_(1,N)_I
BEFORE DELETE ON I
FOR EACH ROW

BEGIN
IF :OLD.K2 IN (SELECT K2 FROM I WHERE K1 IN
              (SELECT K1 FROM I GROUP BY K1 HAVING
               COUNT(*)=1))
THEN ROLLBACK
END;

```

```

CREATE TRIGGER UPDATE_NM_(1,N)_I
BEFORE UPDATE ON I
FOR EACH ROW

BEGIN
IF :OLD.K1<>:NEW.K1 AND :OLD.K2 IN
    (SELECT K2 FROM I WHERE K1 IN
     (SELECT K1 FROM I GROUP BY K1 HAVING
      COUNT(*)=1))
THEN ROLLBACK
END;

```

Transformation of Binary 1:N

For binary 1:N relationships ([Figure 18](#)), there are two solutions when transforming them into the relational model:

- a. Propagating the identifier of the entity that has maximum cardinality 1 to the one that has maximum cardinality N, removing the name of the relationship. (This implies semantic losses, see [Figure 19](#)). If there are attributes in the relationship these will belong to the relation that possesses the foreign key (Date, 1995).
- b. Creating a new relation for the relationship as in the case of the binary N:M relationships (Fahrner & Vossen, 1995).

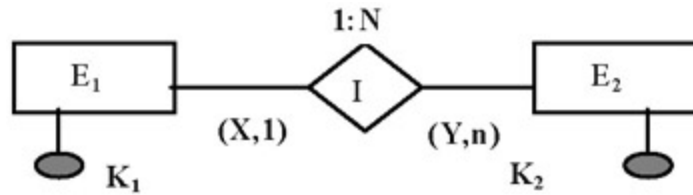


Figure 18: A 1:N relationship
 $E_1(\underline{K_1}, \text{rest_attributes_}E_1)$

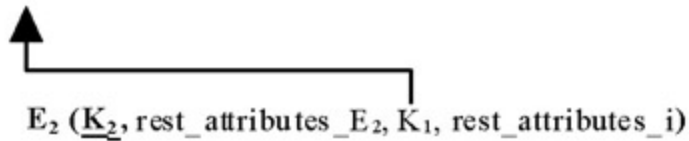


Figure 19: Relational model standard transformation of [Figure 18](#)

We will study the case (a), the most general, distinguishing the different types of minimum cardinality constraints.

Cardinality (0,n)

As the E_1 occurrences can or cannot be related to E_2 occurrences (see [Figure 18](#) with $Y=0$ and $X=0$ or 1). It is not necessary to add anything else to control this cardinality (see [Table 5](#)). The delete and update options should be RESTRICT, CASCADE and SET NULL; depending on the cardinality of the other end of the relationship (case study carried out with $X = 0$ and $X = 1$) and of course on the semantics reflected.

Table 5: Semantic loss in cardinality (0,n) updating transactions			
Relations	Updating		Is there any semantic loss?
			Why?
E_1	Insert	NO	It is possible to insert tuples of E_1 without connecting to E_2
	Delete/Update	NO	It is checked by the FK delete/update option in E_2
E_2	Insert/Delete/Update	NO	It is not a case of study

Cardinality (1,n)

This cardinality indicates that each E_1 occurrence has to be related to at least one E_2 occurrence (see [Figure 18](#) with $Y=1$ and $X=0$ or 1), and thus insertion of new E_1 occurrences should be controlled. Moreover, if an occurrence of E_2 is deleted, it is necessary to control that no element of E_1 remains without being related to an element of E_2 (see [Table 6](#)).

To represent this cardinality the following triggers must be created:

```
CREATE TRIGGER INSERTION_1N_(1,N)_E1
```

```

BEFORE INSERT ON E1
FOR EACH ROW

BEGIN
ASK_PK (E2, :VK2) ;
IF NOT EXISTS (SELECT * FROM E2 WHERE K2=:VK2)
THEN
    (*CREATE A NEW TUPLE IN E2 THAT IS RELATED
    TO THE NEW OCCURRENCE OF E1 *)
    ASK_REST (E2, :VREST_ATTRIBUTES_E2) ;
    INSERT INTO E2 (K2, REST_ATTRIBUTES_E2, K1)
    VALUES (:VK2, :VREST_ATTRIBUTES_E2, :NEW.K1)
ELSE
    UPDATE E2
    SET K1=:NEW.K1
    WHERE K2=:VK2
END ;

```

For the deletions and updates of tuples in the relation E_2 the following triggers have been implemented:

```

CREATE TRIGGER DELETION_1N_(1,N)_E2
BEFORE DELETE ON E2
FOR EACH ROW

BEGIN
IF :OLD.K2 IN (SELECT K2 FROM E2 WHERE K1 IN
    (SELECT K1 FROM E2 GROUP BY K1 HAVING
    COUNT(*)=1))
THEN ROLLBACK
END ;

CREATE TRIGGER UPDATE_1N_(1,N)_E2
BEFORE UPDATE ON E2
FOR EACH ROW
BEGIN
IF :OLD.K1<>:NEW.K1 AND :OLD.K2 IN
    (SELECT K2 FROM E2 WHERE K1 IN
    (SELECT K1 FROM E2 GROUP BY K1 HAVING
    COUNT(*)=1))
THEN ROLLBACK
END ;

```

Both foreign key delete and update options can be RESTRICT or CASCADE, depending on the semantics to reflect.

Cardinality (0,1)

[Figure 18](#) displays this case (with $X=0$ and $Y=0$ or 1). To control that the minimum cardinality is 0 (see [Table 7](#)), the foreign key K_1 (see [Figure 19](#)) has to admit null values. The delete and update options, besides RESTRICT or CASCADE, can be SET NULL; it will depend on the semantics in the UD. The update option will be CASCADE. In this case, it is not necessary the use of triggers.

Table 7: Semantic loss in cardinality (0,1) updating transactions			
Relations	Updating		Is there any semantic loss?
			Why?
E_1	Insert	NO	It is not a case of study
	Delete/Update	NO	It is checked by the FK delete/update option in E_2
E_2	Insert/Delete/Update	NO	It is possible to insert/delete/update tuples into E_2 without connecting to E_1

Cardinality (1,1)

In this case, when forcing all occurrences of E_1 to be related to an occurrence of E_2 (see [Figure 18](#), with $X=1$ and $Y=0$ or 1), the foreign key K_1 (see [Figure 19](#)) can not admit null values and therefore, the delete and update options do not admit the SET NULL.

As noticed in [Table 8](#), it is not necessary to implement any trigger for specifying cardinality constraints in the transformation into relational model.

Table 8: Semantic loss in cardinality (1,1) updating transactions			
Relations	Updating		Is there semantic loss?
			Why?
E_1	Insert	NO	It is not a case study
	Delete/Update	NO	It is checked by the FK delete/update option in E_2
E_2	Insert	NO	It is checked by the NOT NULL option of K_1
	Delete/Update	NO	It is possible to delete/update tuples from E_2

Transformation of Binary 1:1 Relationships

This kind of relationship can be considered a special case of the N:M or 1:N relationships. Therefore, its transformation to the relational model can be performed in different ways. It has been chosen as the transformation that avoids the presence of null values, although other considerations could have been kept in mind; for example, efficiency in updating or querying the database.

Cardinality (0,1), (0,1)

The relationship I (Figure 20) becomes a new relation. None of the two foreign keys admits null values and one of them (in this case K_1 according to Figure 21) will play the role of primary key while the other one is defined as an alternative key (K_2). The delete and update options of the foreign keys can be RESTRICT or CASCADE. Triggers that enforce the database integrity are not needed (see Table 9).

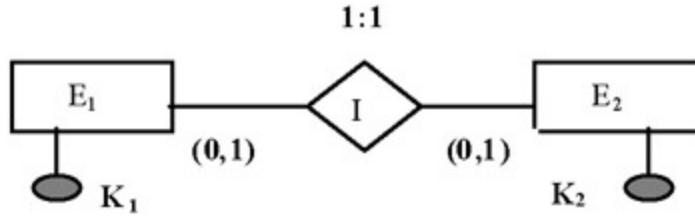


Figure 20: A 1:1 relationship

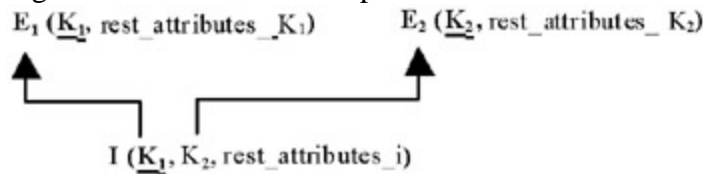


Figure 21: Relational model transformation of Figure 20

Table 9: Semantic loss in cardinality (0,1) (0,1) updating transactions			
Relations	Updating		Is there semantic loss?
			Why?
E_1	Insert/Delete/Update	NO	It is possible to insert/delete/update tuples into E_1 without connecting to E_2
E_2	Insert/Delete/Update	NO	It is possible to insert/delete/update tuples into E_2 without connecting to E_1
I	Insert/Delete/Update	NO	It is possible to insert/delete/update tuples into I

Cardinality (1,1), (0,1)

Figure 22 (with $X=1$ and $Y=0$) shows a relationship of this type. The transformation process propagates the key of the entity with cardinality (1,1) to the resulting relation of the entity with cardinality (0,1), see Figure 19. Null values are not admitted in the foreign key that is also an alternative key (UNIQUE); in this way, it is ensured that all occurrences of E_2 are related to an occurrence of E_1 . The delete and update options of the foreign key can be RESTRICT or CASCADE. Therefore, no triggers are needed. to control this cardinality (see Table 10).

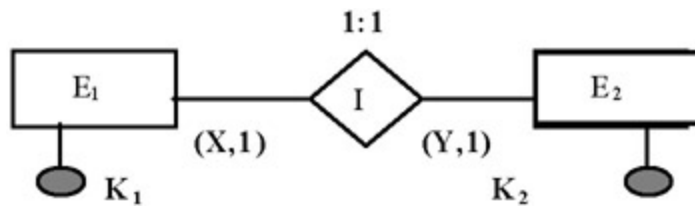


Figure 22: A 1:1 relationship

Table 10: Semantic loss in cardinality (1,1) (0,1) updating transactions			
Relations	Updating		Is there any semantic loss?
			Why?
E_1	Insert	NO	It is possible to insert tuples into E_1 without connecting to E_2
	Delete/Update	NO	It is checked by the FK delete/update option in E_2
E_2	Insert/Delete/Update	NO	It is checked by the FK in E_2

Cardinality (1,1), (1,1)

[Figure 22](#) shows a relationship of this type (with $X=Y=1$). The transformation into the relational model would be performed by propagating the primary key, as it is observed in the [Figure 19](#). Another more symmetrical choice would be the propagation of both keys, which would duplicate the triggers and slow down the database updates. This foreign key is not null (NOT NULL), or unique (UNIQUE), and the deletion option can be RESTRICT or CASCADE (the same values for the update option). With these semantic constraints, it is ensured that all occurrences of E_2 are related to one and only one E_1 . To reflect that all occurrences of E_1 are related to one and only one occurrence of E_2 the following triggers should be created (see [Table 11](#)).

```

CREATE TRIGGER INSERTION_11_(1,1)_E1
BEFORE INSERT ON E1
FOR EACH ROW

BEGIN ASK_PK (E2, :VK2) ;
ASK_REST (E2, :VREST_ATTRIBUTES_E2) ;
(*A NEW TUPLE IN E2 CREATED THAT IS RELATED WITH
THE NEW OCCURRENCE OF E1*)
INSERT INTO E2 (K2, REST_ATTRIBUTES_E2, K1)
VALUES (:VK2, :VREST_ATTRIBUTES_E2, :NEW.K1)
END ;

```

Table 11: Semantic loss in cardinality (1,1) (1,1) updating transactions			
Relations	Updating		Is there any semantic loss?
			Why?
E_1	Insert	YES	It is not possible to insert tuples into E_1 without connecting to E_2
	Delete/Update	NO	It is checked by the FK delete/update option in E_2
E_2	Insert	NO	It is checked by the NOT NULL option of K_1
	Delete	YES	It is not possible to delete tuples from E_2 because there can be tuples not connected
	Update	NO	It is possible to update tuples in E_2

To control the deletion of tuples from the relation E_2 :

```
CREATE TRIGGER DELETION_11_(1,1)_  $E_2$ 
BEFORE DELETE ON  $E_2$ 
FOR EACH ROW

BEGIN
  (*DELETES THE CORRESPONDING TUPLE FROM  $E_1$ *)
  DELETE FROM  $E_1$  WHERE  $K_1 = :OLD.K_1$ 
END;
```

Since the triggering graph obtained for the previous triggers contains execution cycles, it is not a complex issue to control the possibility of non-termination. This would be carried by eliminating the existing cycles from the activation graph if triggers are refined and they control the number of entity occurrences that remains unsettled in accomplishing the relationship cardinality constraints. Currently, an Oracle^[18] prototype that perfectly reproduces this binary relationship behaviour is available. The extension of this approach is to contemplate that several binary relationships naturally imply a bigger interaction among the triggers as more problematic in guaranteeing the termination of the obtained set of triggers.

Higher Order Relationships

In the previous section we presented the transformation of binary relationships to the relational model to guarantee the semantics specified by the cardinalities by means of an active rules based technique (triggers). In this section we will study the transformation of higher order relationships: first, some problems concerning the use of ternary relationships are outlined; next, a semantics preserving transformation is defined and, finally, the generalisation to n-ary relationships ($n > 2$) is explained. In Dey, Storey & Barrow (1999) a similar analysis of transformation is performed considering semantics of participation in cardinality constraints.

Several authors have attempted to reduce the complexity of transforming ternary relationships into the relational model, looking for solutions at the conceptual level and proposing that all ternary relationships become several binary relationships through an intermediate entity type, (Ullman & Widom, 1997). This solution can be seen as a special case of the so-called standard of the transformation (Figure 24). However, we believe that to carry out this transformation at the conceptual level is hasty and may imply a certain semantic loss (Silberschatz, Korth & Sudarshan, 2001).

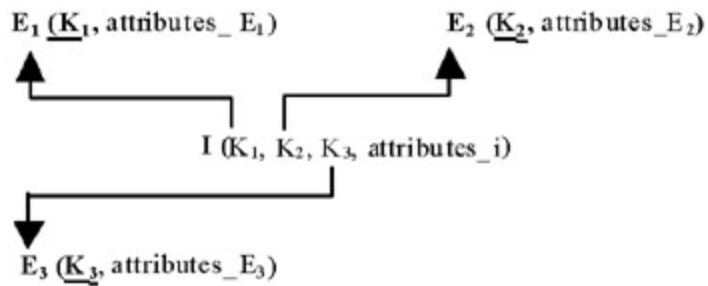


Figure 24: Standard transformation of a ternary relationship (Figure 23)

On the other hand, Elmasri & Navathe (1994), Hansen & Hansen (1995) and others, treat in different ways a ternary relationship in a conceptual model and its transformation into a relational model. They propose as a unique solution the transformation to the general case, although they recognise that combining the ternary relationship with one or more binary relationships could be of interest, even if cardinality constraints are not taken into account. In our opinion, cardinality constraints contribute so much to validate and transform ternary relationships.

The transformation to the general case (Figure 23) translates each entity into a relation and the relationship into a new relation; the foreign key delete and update options would be defined in the cascade mode, as is shown in Figure 24. The general case does not observe the relationship cardinalities. However, as can be seen in the following classification (Table 12) in a ternary relationship there are a total of twenty possible combinations for the minimum and the maximum cardinalities and the previous transformation will not be able to represent the associated semantics in all of them.

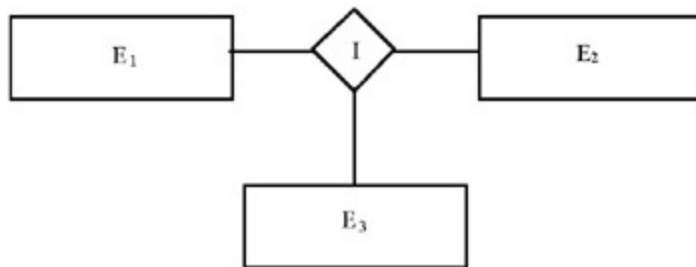


Figure 23: A ternary relationship

Table 12: Ternary relationship classification											
<i>N:M:P</i>			<i>1:N:M</i>			<i>1:1:N</i>			<i>1:1:1</i>		
<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
(0,N)	(0,N)	(0,N)	(0,1)	(1,N)	(1,N)	(0,1)	(0,1)	(0,N)	(0,1)	(0,1)	(0,1)
		(1,N)			(0,N)			(1,N)			(1,1)
	(1,N)	(1,N)		(0,N)	(0,N)	(1,1)	(0,1)	(1,N)		(1,1)	(1,1)
(1,N)	(1,N)	(1,N)	(1,1)	(1,N)	(1,N)			(0,N)	(1,1)	(1,1)	(1,1)
					(0,N)		(1,1)	(1,N)			
				(0,N)	(0,N)		(1,1)	(1,N)			

Contrary to what happens in the binary relationships treatment, where problems arise when the minimum cardinality is 1, in higher order relationships the biggest problems take place when the minimum cardinality is 0. It is not a singular issue because of the different semantics that acquire the cardinalities in each case.

In the case that all the minimum cardinalities are 1, the standard transformation can be applied ([Figure 24](#)), adding as primary key of I the three attributes that come from the propagation of the primary keys of the entities associated by the relationship. Referential integrity in the relationship I would ensure the semantics of the ternary relationship.

If one of the minimum cardinalities is 0 (for example, E_3 in [Figure 25](#)) and comparing it to the relational schema of [Figure 24](#) (standard transformation), standard transformation would not be applicable, since the relationship I would have as a primary key the composition of the primary keys of E_1 , E_2 y E_3 , but the presence of 0 would indicate that there can be occurrences of E_1 related to occurrences of E_2 but not to occurrences of E_3 ; consequently, it means that K_3 as the primary key component could allow nulls, which is contradictory with the standard primary key definition^[19].

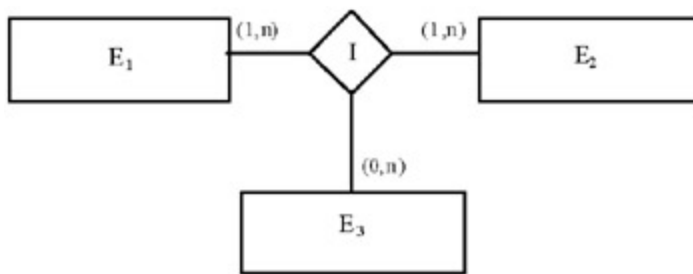


Figure 25: A ternary relationship with minimum cardinality 0

Although the aforementioned authors propose to carry out the transformation in the conceptual schema by means of the use of binary relationships, the same problem remains due to the fact that transformation into the relational model of the connecting entity has as primary key the set of the primary keys of the participant entities E_1 , E_2 and E_3 . By imposing the use of binary relationships, we believe that a limitation of the relational model is moved to the conceptual model, although it is an easier solution nearer to implementation aspects; for example most of the commercial CASE tools only support binary relationships in their models. Since we propose a conceptual model that is completely independent of any logical model, we must concentrate our efforts in the transformation process in order to allow the designer to model at the maximum abstraction level.

To represent the semantics of minimum cardinality 0, the proposed solution, shown in [Figure 26](#), is to transform the original relationship I into two relations, I and I_{E_3} where I contains the occurrences of E_1 related to E_2 and to E_3 , while I_{E_3} only represents those occurrences of E_1 related to E_2 , but not E_3 . As it can be seen, new semantics has been incorporated into the general case transformation.

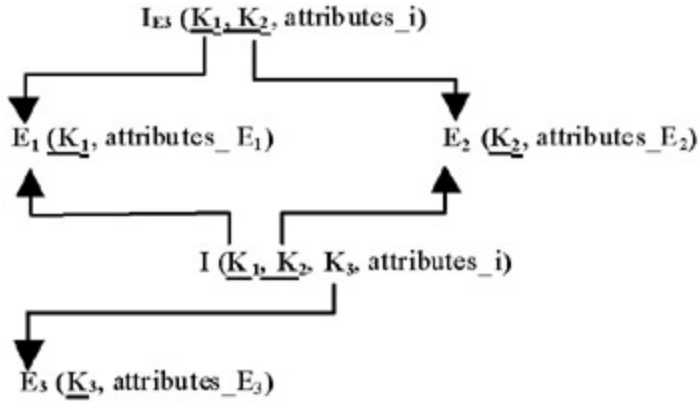


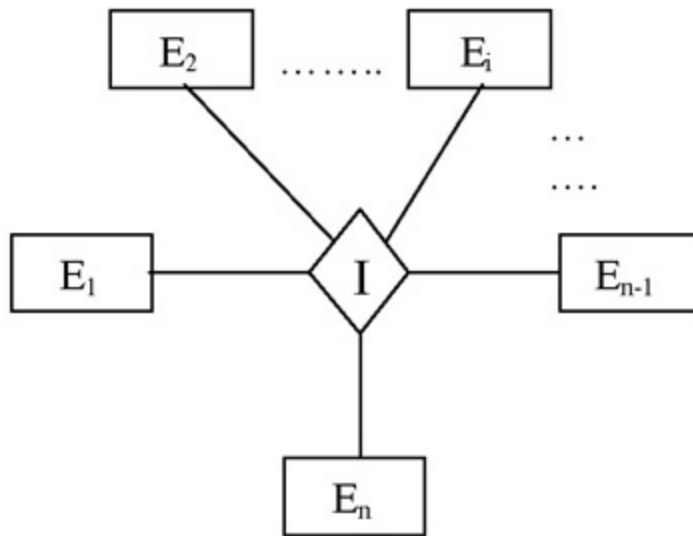
Figure 26: Relational model transformation of [Figure 25](#)

Relations I and I_{E3} are semantically related although their understanding is different from each other and also different from the original relationship I . The semantics of the original relationship I ([Figure 25](#)) is equivalent to the whole semantics of both I and I_{E3} . Moreover, I and I_{E3} should not be overlapped in the sense that the tuples of E_1 and E_2 that participate in I_{E3} cannot participate in I because they are not related to any tuple of E_3 .

Now that the problem for ternary relationships has been presented and the contribution to solving it considering the minimum cardinalities 0 has been explained, let us see how we can generalise this processing for n -ary relationships ($n > 2$).

Transformation of n -ary Relationships ($n > 2$)

Let I be a n -ary relationship with $n > 2$ and no semantic constraint exist among the entities that participate in I . This implies that the relationship cannot be decomposed in lower degree relationships without semantic loss ([Figure 27](#)). The entities E_i have, as a main identifier, an attribute or group of attributes K_i and let us suppose that these entities are different from each other. Besides, to clarify in the notation, the rest of the attributes of the entities will not be considered since they do not affect the transformation process.

Figure 27: N-ary relationship $n > 2$

The standard transformation of the relationship I produces one relation that has at least n attributes and each attribute has associated with it the foreign key constraint corresponding to the entity in which the attribute was main identifier. Therefore, just like the binary relationships case a basic transformation represented in the [Figure 28](#) is used; it includes the primary keys of each E_i and the foreign keys in I . Delete and update options are not represented, although in the transformation of binary relationship (see previous section) these options help us specify the semantics of the cardinalities, but do not contribute to the transformation of higher order relationships. Consequently, this is the standard transformation and additional constraints will be added for taking into account the cardinalities of each E_i in the conceptual model while trying to preserve their semantic.

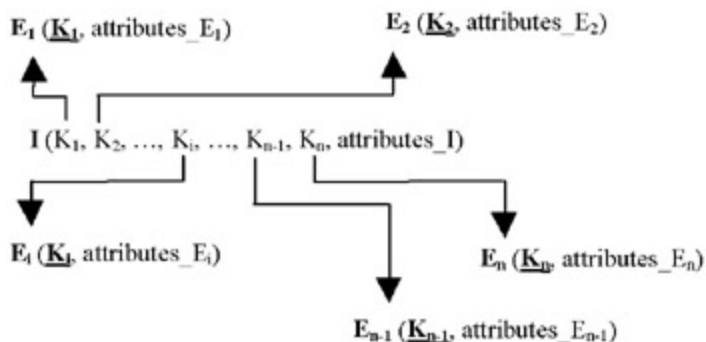


Figure 28: N-ary relationship standard transformation into the relational model

Definitions : Let $C_{(X,Y)}(I)$ be the groups of different cardinality constraints associated to the entities participating in the higher order relationship I ($n > 2$):

in which $C_{(0,n)}(I) \cap C_{(0,1)}(I) \cap C_{(1,n)}(I) \cap C_{(1,1)}(I) = \emptyset$ and $C_{(0,n)}(I) \cup C_{(0,1)}(I) \cup C_{(1,n)}(I) \cup C_{(1,1)}(I) = \{E_1, \dots, E_n\}$ and where $\text{card}(I, E_i)$ indicates the minimum and maximum cardinality of the entity type E_i in the relationship I .

We also define the notation used for the semantic restrictions of relational model:

- $PK(R)$ = attributes that composed the primary key of the relation R .
- $FK_i(R,S)$ = attributes that composed the i -th foreign key of the relation R that references to the relation S .
- $UK_i(R)$ = attributes that composed the i -th alternative key of the relation R .
- $MA(R)$ = mandatory attributes of R .

The transformation algorithm is given below:

```

if  $C_{(0,1)} \neq \emptyset$  then
  1. Let be  $E_i \in C_{(0,1)}$  then add the following constraints:
       $PK(I) = (K_1, \dots, K_{i-1}, K_{i+1}, \dots, K_n)$ 
       $K_i \in MA(I)$ 
  2. For the remainder of  $E_k \in C_{(0,1)}$ , add to each of them the
  following
  constraints:
       $UK_k(I) = (K_1, \dots, K_{k-1}, K_{k+1}, \dots, K_n)$ 
  A create a new relation:
       $I_{EK} (K_1, \dots, K_{k-1}, K_{k+1}, \dots, K_n)$ 
  And the following associated constraints to the new relation:
       $PK(I_{EK}) = (K_1, \dots, K_{k-1}, K_{k+1}, \dots, K_n)$ 
       $FK_t(I_{EK}, E_t) = (K_t), t = 1, \dots, k-1, k+1, \dots, n$ 
  3. For each  $E_r \in C_{(1,1)}$  add the following constraints
       $UK(I)_r = (K_1, \dots, K_{r-1}, K_{r+1}, \dots, K_n)$ 
  4. For each  $E_s \in C_{(0,n)}$  create  $I_{ES} (K_1, \dots, K_{s-1}, K_{s+1}, \dots, K_n)$ 
      With the following constraints:
       $PK(I_{ES}) = (K_1, \dots, K_{s-1}, K_{s+1}, \dots, K_n)$ 
       $FK_t(I_{ES}, E_t) = (K_t), t = 1, \dots, s-1, s+1, \dots, n$ 

else if  $C_{(1,1)} \neq \emptyset$  then
  1. Let be  $E_i \in C_{(1,1)}$  then add the following constraints:
       $PK(I) = (K_1, \dots, K_{i-1}, K_{i+1}, \dots, K_n)$ 
       $K_i \in MA(K)$ 
  2. For the remainder of  $E_k \in C_{(1,1)}$ , add to each one of them the
  following constraints:

```

```

        UKIk = (K1, ..., Kk-1, Kk+1, ..., Kn)
3. For each Es ? C(0,n) create IEs(K1, ..., Ks-1, Ks+1, ..., Kn)
   With the following constraints:
   PK(IEs) = (K1, ..., Ks-1, Ks+1, ..., Kn)
   FKt(IEs, Et) = (Kt), t = 1, ..., s-1, s+1, ..., n

else If C(0,n) ? f then
1. PK(I) = (K1, ..., Ki, ..., Kn)
2. For each Ej ? C(0,n) create IEj(K1, ..., Kj-1,
Kj+1, ..., Kn)
   With the following constraints:
   PK(IEj) = (K1, ..., Kj-1, Kj+1, ..., Kn)
   FKt (IEj, Et) = (Kt), t=1, ..., j-1, j+1, ..., n

else PK(I) = (K1, ..., Ki, ..., Kn)

```

All relationships created, I_{Ei} , will be transparent to the users who access to the database; it means that they are supported on the DBMS. This automatic control of the relations I_{Ei} will make the semantics of the original relationship I ([Figure 27](#)) equivalent to the whole semantics of the relations.

To carry out updates and queries in the appropriate relations when a data request is made for the relation I the use of triggers is required. The problem is that the triggers only are activated when the event is an update, and so has to be built as the result of the union of all relations I_{Ei} and I in order to use triggers in queries. This view will hide the implementation details of the base relations to the users and, consequently, it will avoid that the database could be in an inconsistent state. The update triggers will be associated to this view and depending on the features of the update tuple(s) the transaction will be made in I or in some I_{Ei} . Queries always will be carried out in the view, because this view reflects the semantics of relationship I .

Building a relational structure, such as a view, that is able to represent an abstraction level between the user and the DB with the purpose of maintaining cardinality constraints and in this way not leaving inconsistency in the database, implies several practical implications that will be commented on in the following section.

Finally, an example of the application of the algorithm for a ternary relationship that holds the most restrictive cardinalities ([Figure 29a](#)) from the point of view of Chen's style definition is illustrated.

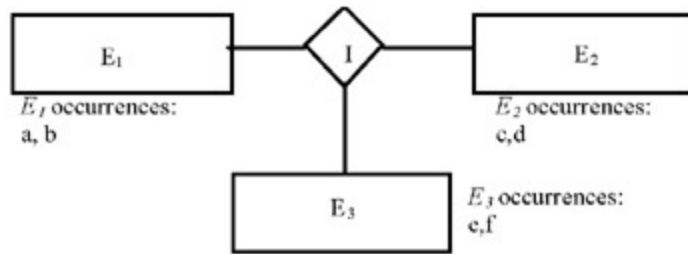


Figure 29a: Ternary relationship example

Let be a and b E_1 occurrences, c and d E_2 occurrences and e and f E_3 occurrences (Figure 29a). And let be, at time t , the following relationship occurrences, Figure 29b.

Figure 29b: I relationship occurrences, at time t

Note that the bottom means the optional cardinalities in E_1 and E_2 . The standard transformation brings about a relation for each entity:

- $E_1(K_1, \dots)$ with PK (E_1)
- $E_2(K_2, \dots)$ with PK (E_2)
- $E_3(K_3, \dots)$ with PK (E_3)

And one relation for the relationship:

- $I(K_1, K_2, K_3)$ with $FK_i(I, E_i) = (K_i)$, $i = 1, 2, 3$.

It cannot be added to the primary key constraint since it might not exist K_1 or K_2 values.

Applying the transformation algorithm, in order to add constraints and new relations to gather the semantic of the cardinality constraint in I :

As it exists one (and only one) entity in $C_{(0,1)}$, the input is the first I_f , so the following constraints must be added:

- $PK(I) = (K_2, K_3)$
- $K_1 \rightarrow MA(I)$

Note that as the E_1 minimum cardinality is 0, K_1 may allow null values so it can not take part of the I primary key. The maximum cardinality 1 is ensured by the primary key K_2, K_3 .

The maximum and minimum cardinality of E_3 is 1, so it exists as a functional dependency $K_1, K_2 \rightarrow K_3$ and is collected through an alternative key:

- $UK(I)_R = (K_1, K_3)$

Finally, to collect the semantic associated to the cardinality $(0,n)$ the relation $I_{E2}(K_1, K_3)$ must be added, with the following constraints:

- $PK(I_{E2}) = (K_1, K_3)$
- $FK_T(I_{E2}, E_T) = (K_T), T = 1,3$

^[18]© Oracle Corporation.

^[19]The standard primary key definition is given in Date (1995), although other authors like Thalheim maintain that the primary key can allow nulls in some of its attributes (1989).

PRACTICAL IMPLICATIONS

The main problem for most database designers is that when using a conceptual model, in spite of its powerful in semantics collection, it is very complicated to not lose part of this semantics in the transformation into a logical model, as in the relational model (Codd,

1979). Designers realise how to transform entities or classes, relationships or associations, but there exist other significant constraints which are not so easy to transform.

This means that conceptual models are used exclusively as a validation tool and as a specification document of the user requirements. The database design will be carried out in a less abstract model and in a model which may be directly implemented in a DBMS.

In many situations, designers lean on CASE tools to carry out the transformation from a conceptual to logical model. However, these tools merge conceptual with logical and even physical aspects and apply simple rules that they know, so they are only useful to save time in design^[20].

In any case, constraints' control or business rules of an information system will be allocated by the different applications that access to the database. In this way, developers will have the responsibility to control the database consistency. This distribution of the semantics has associated a potential lose of control, as well as database inconsistencies.

In this chapter a methodology is presented for achieving the transformation of a significant constraint into conceptual models, as a cardinality or multiplicity constraint, with the objective to keep their semantic in the database.

Due to the fact that the relational model does not support full control for cardinality constraints, the methodology leans on ECA rules that are built into several DBMSs, fitting them of activity. There exist two types of active behaviour, proactive and reactive. Proactive behaviour generates events, in the opposite, reactive behaviour consumes events.

Usually active DBMSs present reactive behaviour; therefore, it will be necessary for some external element to act as a source of events. Once the DBMS detects the appearance of an event, it will analyse its environment and it will autonomously execute an action if necessary.

It is impossible to collect all the semantics into the relational model. Additional elements not belonging to the model are needed. In this way, in the triggers built, elements out of standard proposals, as SQL3 (Melton & Simon, 1993) have been used. Next, the main characteristics needed are shown:

- User interaction.
- Procedural capabilities
- Transactional capabilities.
- External semantic control.

User interaction is needed to solve the information management required in the trigger execution. In the internal constraints case, this interaction may be ruled by the DBMS from the schema information in the database (metadata). In the implemented triggers, user interaction is performed through invocation to procedures. The invocation to procedures as action of ECA rules is considered appropriate (ACT-NET Consortium, 1996) to allow the DBMS to control the information system. For example, this approach could be used for the problem of maintaining updated forms that are visualised by the user or as we have realised for the problem of conduct data entry. Commercial DBMSs give insufficient support for interacting with the user through a call to a procedure for data entrance, although it is possible to maintain several updates in a data structure and only to make them effective if it is proven that they don't violate the cardinality constraint. That test will be carried out with triggers.

In some cases, in the case of user interaction, a potential problem is that the life of a transaction is dependent on user's action. For example, if this user was absent of the work, the resources used by the transaction would be locked. A possible solution would be to establish a timer to force its end.

Many commercial RDBMSs have extended the SQL92 standard with procedural capabilities. SQL3 standard (Melton & Simon, 1993) has a part (SQL/PSM), based on these capabilities. It is required to have, besides user interaction, at least the possibility of minimal control flow capabilities (a conditional statement), comparison and set membership.

The event of the triggers is always an update operation on the DB, thus, they are activated in the scope of a transaction and the trigger execution must be integrated with transaction execution. Interaction with transaction is an active research field of Active DBMS. Moreover, they would solve the synchronisation problems and the non-termination possibility that can take place in the interaction between rules and the usual operation of the DBMSs. These problems would require the establishment of a concrete execution model that is not objective of this study (Paton & Díaz, 1994; Ceri & Faternalli, 1997; Paton & Díaz, 1999). In particular, in the trigger body it is only needed to roll back the whole transaction. It can be useful a nested transaction model (Orfali, Harkey & Edwards, 1999; Gray & Reuter, 1993), such that it would allow reentry in case a procedure begins a transaction.

When using external procedures to control the semantics, the DBMS does not know what actions the procedures perform, so they may violate the integrity. A possible solution to this problem, chosen in this chapter, is to establish a contract or a commitment between the DBMS and the external procedure. In this way, the semantics control is only carried out by the DBMS while the application procedures are limited only to data entry. To ensure the execution of this contract, a concrete system could demand application registration and certification procedures.

To ensure that the semantic is shared by all applications independently, access to the database is necessary to transfer the semantic control to the DBMS. This is especially

significant with the tools that permit users to access the database contents. To maintain the semantics together with the database schemata is an open research and a way to fit to the schemas of the executability property (Hartmann et al., 1994; Sernadas, Gouveia & Sernadas, 1992; Pastor et al., 1997; Ceri & Faternalli, 1997). The case study presented in this chapter tends to adjust to this possibility, because it allows the static semantic in the EER schema may decide the dynamic behaviour in the database, although the system dynamic behaviour is not studied.

[20] Sometimes, even if the tool learning cost is higher enough, it is not even time saving.

FUTURE RESEARCH

In this chapter we have tried to study in depth and clarify the meaning of the features of conceptual models. The disagreements between the main conceptual models, the confusion in the use of some of their constructors, and some open problems in these models, have been shown.

Another important question treated in this chapter is the conceptual schemata transformation process into logical schemas. Some algorithms have been present to preserve the cardinality constraint semantics in both, binary relationships and higher degree relationships for their implementation in a DBMS with active capabilities.

There are two main causes of semantic loss in database design: First, semantics collected in conceptual schemata are not enough to reflect the overall Universe of Discourse due to the limitations of conceptual constructors. This requires adding explicitly to the conceptual schema some informal statements about constraints. A notation extension for reflecting the MERISE participation and Chen cardinality definition in higher order relationships should be proposed.

On the other hand, in any database development methodology there is a process devoted to transform conceptual schemata into logical schemata. In such process, a loss of semantics can exist (logical constructs are not coincident with conceptual constructs; for example, entities and relationships in conceptual schemata become relations in logical design). So, some algorithms with active rules must be applied to achieve that the logical models keep their semantics.

In the last decade, multiple attempts of giving a more systematic focus to the resolution of modelling problems have been developed. One such attempt has been the automation of database design process by using CASE tools that neither have enough intelligent methodological guidance, or provide, usually, adequate support the design tasks. Commercial CASE tools for database developments do not cover database design phase with real EER models; that is, they only provide graphical diagrammatic facilities without refinement and validation tools that are independent of the other development phases. CASE environments usually manage hybrid models (merging aspects from EER and Relational models) sometimes too close to physical aspects and they use a subset of EER graphical notation for representing relational schemata.

A suitable CASE tool is one that incorporates a complete conceptual model, as the ER model, by adding the semantics into the relationships, in the way as we propose in this chapter.

ACKNOWLEDGEMENTS

This work takes part of the project PANDORA (CASE Platform for Database development and learning via Internet) Spanish research CICYT project (TIC99-0215).

ENDNOTES

1. From now on, we will use the original names (entity and relationship).
2. The presence of a multivalued attribute in a relationship implies that the identification of its occurrences has to consider this attribute.
3. Possible occurrences of the relationship with incomplete information are not considered.
4. Multiple combinations of entities are possible; for example, in n -ary relationships ($n \geq 2$), combinations of only one entity up to combinations of $n-1$ entities are possible. In this sense, the definitions that consider combinations do not define a unique possibility, but many (as many as possible combinations).
5. The possible occurrences of the relationship with incomplete information are not considered.
6. This concept is translated from the relational model into a functional dependency that can be used in refining the relational schema.
7. We suppose that there are not additional semantic constraints between the entities participating in the ternary relationship of type: "an employee works exactly in one department," "an employee uses one programming language," etc.
8. Maximum cardinality of 1 in Programming Language expresses a functional dependency: employee, project ? programming language.
9. In French, contrainte d'intégrité fonctionnel.
10. While Chen's approach is not able to express optional/mandatory participation, it represents functional dependencies.
11. This exclusive-or constraint is not directly established between the two relationships, because one is a ternary relationship while the other is a binary one. It is established between the binary relationship and an algebraic projection of the ternary relationship.
12. Entity integrity constraint (defined in the relational model and generalized to the ER model) disallows optional attributes as part of an entity identifier; this constraint has been relaxed in the relational model (Thalheim, 1989; Levene and Loizou, 1998), by replacing the *identification* concept with the *distinguishability* (*disjunctive identification*) concept.
13. It is not a formal and correct definition if we want to keep the relational model property that the order of attributes is irrelevant. In addition, it does not consider

- the existence of attributes in the relationship and suppose that all attributes in the entities are primary identifier attributes.
14. This symbol has been used in a similar way by denotational semantics.
 15. It is not a formal and correct definition if we want to keep the relational model property that the order of attributes is irrelevant. In addition, it does not consider the existence of attributes in the relationship and suppose that all attributes in the entities are primary identifier attributes.
 16. In fact, these occurrences are pairs composed of an occurrence of one entity and the bottom symbol of the other entity.
 17. This observation poses a difficulty in attaining consistency among the model constructs that can interact with each other in unpredictable ways.
 18. © Oracle Corporation.
 19. The standard primary key definition is given in Date (1995), although other authors like Thalheim maintain that the primary key can allow nulls in some of its attributes (1989).
 20. Sometimes, even if the tool learning cost is higher enough, it is not even time saving.

REFERENCES

- ACT-NET Consortium (1996). *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*. *ACM Sigmod Record* 25 (3), 40–49, 1996.
- Batra, D. & Antony, S. R. (1994). *Novice Errors in Conceptual Database Design*. *European Journal of Information Systems*, 3(1), 57–69.
- Batra, D. & Zanakis, H. (1994). *A Conceptual Database Design Approach Based on Rules and Heuristics*. *European Journal of Information Systems*, 3(3), 228–239.
- Boman, M. et al. (1997). *Conceptual Modelling*. Prentice Hall Series in Computer Science.
- Buneman, P. et. al. (1991). *Using power domains to generalize relational databases*. *Theoretical Computer Science* 91, 23–55.
- Ceri, S. & Fraternali, P. (1997). *Designing database applications with objects and rules : the IDEA Methodology*. Addison-Wesley.
- Chen, P. P. (1976). *The Entity-Relationship Model: Toward a Unified View of Data*. *ACM Transactions on Database Systems*. 1, 1, 9–36.
- Codd, E. F. (1970). *A Relational Model of Data for Large Shared Data Banks*. *CACM* 13 (6), June 1970.
- Codd, E. F. (1979). *Extending the Database Relational Model to Capture More Meaning*. *ACM Transactions on Database Systems*, 4 (4), 397–434, December 1979.
- Date, C. J. (1986). *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass.
- Date, C. J. (1990). *An Introduction to Database Systems*. 5th ed. Addison-Wesley, Reading, Mass.
- Date, C. J. (1995). *An Introduction to Database Systems*. 6th ed. Addison-Wesley, Reading, Mass.
- Dey, D., Storey, V. C. & Barron, T. M. (1999). *Improving Database Design Through the Analysis of Relationships*. *TODS* 24(4), 453–486.

- Elmasri, R. & Navathe, S. (1994). *Fundamentals of Database Systems*. 2nd ed. Benjamin-Cummings, 1994.
- Fahrner, C. & Vossen, G (1995). *A survey of database design transformations based on the Entity-Relationship model*. *Data & Knowledge Engineering* 15, 213–250, 1995.
- Gray, J. & Reuter, A. (1993). *Transactions processing, concepts and techniques*. San Mateo: Morgan Kaufmann, cop.
- Hansen, G. & Hansen, J. (1995). *Database Management and Design*. Prentice-Hall, 1995.
- Hartmann, T. et. al. (1994). *Revised Version of the Conceptual Modelling and Design Language TROLL*. *Proceedings ISCORE Workshop*, Amsterdam, 89–103.
- Hull, R. & King, R. (1987). *Semantic Database Modelling: Survey, Application, and Research Issues*. *ACM Computing Surveys* 19 (3), 201–260, 1987.
- Jones, T. H. & Song, IL-Y. (1998). *And Analysis of the Structural Validity of Ternary Relationships in Entity-Relationship Modelling*. *Proceedings of the 7th International Conference on Information and Knowledge Management*, CIKM'98, 331–339, November, 1998.
- Levene, M. & Loizou, G. (1998). *A Generalisation of Entity and Referential Integrity in Relational Databases*. *Theoretical Computer Science*, 206(1–2), 6 October 1998, 283–300.
- Martínez, P. et. al. (2001). *Multivalued attributes in conceptual modelling: design implications*. To appear in *Journal of Object Oriented Programming*.
- McAllister, A. (1998). *Complete Rules for n-ary Relationship Cardinality Constraints*. *Data & Knowledge Engineering* 27, 255–288.
- Melton, J. & Simon, A.R (1993). *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, Inc., San Mateo, CA.
- Nijssen, G. M. & Halpin, T. A. (1989). *Conceptual Schema and Relational Database Design—A Fact Oriented Approach*. Prentice-Hall, New York.
- OMG (2000). *Unified Modelling Language Specification, Version 1-3*. *Object Management Group*. *ACM Computing Surveys* 31(1): 63–103
- Orfali, R., Harkey, D. & Edwards, J. (1999). *Essential Client/Server Survival Guide*. John Wiley & Sons, Inc. 3^a Edition.
- Paton, N. W. & Díaz, O. (1999). *Active Database Systems*. *ACM Computing Surveys*, 31 (1), 63–103.
- Paton, N. W. (1999). *Active Rules in Database Systems*. Springer-Verlag, New York.
- Pastor, O. et. al. (1997). *OO-METHOD: An OO Software Production Environment for Combining Conventional and Formal Methods*. *Proceedings CAISE'97*, Olivé A., Pastor J.A. (Eds.), LNCS 1250, Barcelona, June 1997, 145–158.
- Peckham, J. & Maryanski, F. (1988). *Semantic Data Models*. *ACM Computing Surveys* 20 (3): 153–189, 1988.
- Ramakrishnan, R. (1997). *Database Management Systems*. MacGraw-Hill International Editions, 1997.
- Rumbaugh, J., Blaha, M. & Premerlani, W. J. (1991). *Object Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- Sernadas, C., Gouveia, P. & Sernadas, A. (1992). *OBLOG: Object-Oriented, Logic-Based, Conceptual Modelling*. Research report, Instituto Superior Técnico.
- Silberschatz, A., Korth, F., & Sudarshan, S. (2001). *Database Design Concepts*. 4th ed. McGraw-Hill, July 2001.

- Soutou, C. (1998). *Relational Database Reverse Engineering: Algorithms to extract Cardinality Constraints*. *Data & Knowledge Engineering* 28, 161–207.
- Tardieu, H., Rochfeld, A. & Coletti, R. (1983). *La Méthode MERISE. Tome 1: Principes et Outils Les Editions d'Organisation*, Paris.
- Teorey, T. J., Yang, D. Fry, J. P. (1986). *A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model*. *ACM Computing Survey*. 18(2).
- Teorey, T. J. (1999). *Database Modeling and Design: The Entity-Relationship Approach*. 3rd ed. Morgan Kaufmann, San Mateo, 1990.
- Thalheim, B. (1989). *On Semantic Issues connected with Keys in Relational Databases permitting Null Values*. *Journal of Information Processing Cybernetics*, 25:11–20.
- Thalheim, B. (2000). *Entity-Relationship Modelling. Foundations of Database Technology*. Springer-Verlag.
- Ullman, J. D. & Widom, J. (1997). *A First Course In Database Systems*. Prentice-Hall International.
- Van der Meydem, R. (1998). *Logical Approaches to Incomplete Information: A Survey*. In J. Chomicky and G. Saake editors, *Logic for Databases and Information Systems*. Kluwer Academic Publishers. Chapter 10, 307–356
- Wand, Y., Storey, V. C. & Weber, R. (1999). *An Ontological Analysis of the Relationship Construct in Conceptual Modelling*. *ACM Transactions on Database Systems*, 24(4), 494–528.

Chapter IV: Integrity Constraints in an Active Database Environment

Juan M. Ale, Universidad de Buenos Aires,

Argentina

Mauricio Minuto Espil, Pontifica Universidad Católica Argentina,

Argentina

This chapter surveys the interaction between active rules and integrity constraints. First, we analyze the static case following the SQL-1999 Standard Committee point of view which, up to date, represents the state of the art. Then, we consider the case of dynamic constraints for which we use a temporal logic formalism. Finally, we discuss the applicability, limitations and partial solutions found when attempting to ensure the satisfaction of dynamic constraints.

INTRODUCTION

Databases are essentially large repositories of data. From the mid-1980s to the mid-1990s, a considerable effort has been made to incorporate reactive behavior to the data management facilities available (Dayal et al., 1988), (Chakravarthy, 1989) and (Stonebraker, 1986). Reactive behavior is seen as an interesting and practical way for checking satisfaction of integrity constraints. Nevertheless, constraint maintenance is not the only area of application of data repositories with reactive behavior. Other interesting applications areas are materialized view maintenance (especially useful in the warehousing area), replication of data for audit purpose, data sampling, workflow processing, implementation of business rules, scheduling, and many others. In fact, practically all products offered today in the marketplace support complex reactive behavior on the client side. Nevertheless, the reactive behavior supported by these products on the server side is, in fact, quite poor. Recently, the topic has regained attention, however, because of the inherent reactive nature demanded in Web applications, and the necessity of migrating much of their functionality from browsers to active Web servers.

Supporting reactive behavior implies that a database management system has to be viewed from a production rule system perspective. Production rule definitions must be supported, therefore, by an active database system. These production rules are well known nowadays, in database terminology, as active rules or simply triggers.

Undesired behavioral characteristics have been observed related to production rule systems, though. For example, termination is not always guaranteed, non-determinism can be expected in the results, and confluence with respect to an desired goal cannot be achieved (Aiken, Hellerstein, Widom, 1995; Baralis & Widom, 2000b). Since triggers and declarative integrity constraints definitions may appear intermingled in a concrete application, an integrating model is needed to soften, to some extent, the effects of this undesirable behavior, ensuring that, no matter what the nature of the rules involved, integrity is always preserved.

Active rules and integrity constraints are related topics (Ceri, Cochrane, Widom, 2000). Systems do not support both completely, but partially, in their kernels. When a constraint must be enforced on data, if such constraint cannot be declared, it may be implemented by means of triggers. Studying the relationships between constraints and triggers from this point of view is therefore mandatory. In simple words, we need methods to check and enforce constraints by means of triggers.

From a user point of view, reactivity is a concept related to object state evolution over time. Dynamic constraints, constraints making assertions on the evolution of object states, may be needed to control changes in the state of data objects (Sistla & Wolfson, 1995a). Dynamic constraints are mandatory in the correct design of applications, particularly for workflow processing and for the Web. Actual products support some kind of functionality in this area, allowing triggers to refer to transitions, the relations existent between states, when an atomic modification operation is executed. Supporting such type of constraints by means of handcrafted triggers written by a novice, without any method in mind, may result in potentially dangerous effects from the perspective of correctness.

Formal methods guaranteeing correctness are thus needed for a good deployment of such triggers.

The main goal of this chapter is to analyze the concepts related to integrity constraints in an active database environment. Hence we focus our discussion on the interaction between active rules and declarative constraints from both static and dynamic perspectives.

BACKGROUND

Usually, a database system performs its actions in response to requests from the users in a passive way. In some cases it is highly desirable that actions could be taken with no human intervention, that is, automatically responding to certain events.

Traditionally, the latter has been obtained by embedding that behavior into the applications, that is, the application software recognizes some happenings and performs some actions in response.

On the necessity of such reactive behavior, obviously, it would be desirable that such functionality would be provided by the database system. A database with the capability to react to stimulus, be these external or internal, is called an *active database*. Among the applications we can find are inventory control systems, on-line reservation systems, portfolio management systems, just to name a few (Paton & Diaz, 1999).

An *active database system* can be thought of as coupling databases and rule-based programming. The active database rules enable many desired database features such as integrity constraint checking and enforcement, derived data maintenance, alerts, authorization checking and versioning.

Knowledge Model

A central issue in the knowledge model of active databases is the concept of *active rule*.

An active rule can be defined throughout three dimensions: *event*, *condition* and *action*. In this case, the rule is termed an ECA or eventcondition-action rule, which specifies an action to be executed upon the happening that is to be monitored, provided a condition holds.

An event is defined as something that happens at a point in time. The source of the event determines how the event can be detected and how it can be described. We have several alternative sources, such as *transactions*, where the event is originated by transaction commands abort, commit and begin-transaction. Other sources are *operations on structure*, where the event is raised by an operation such as insert, delete or update, on some components of the data structure; *clock* or temporal, where the event raises at a given point in time; *external*, in the case that the event is raised by something happening

outside the database. An example of the latter is the level of water reaching some specified height.

An event can be either *primitive*, in which case it is raised by a single occurrence in one source, or *composite*, in which case it is raised by a combination of events, whether primitive or composite.

A condition, i.e., a situation with respect to circumstances, is the second component of an active rule. We must consider the context in which the condition is evaluated. In general, we can associate three different database states with the condition in the processing of a rule, i.e., the database at

- the start of the current transaction,
- the time when the event took place,
- the time the condition is evaluated.

Moreover, since the state before and after the occurrence of an event may be different, the condition can require the access to a previous or a new value.

An action consists of a sequence of operations. There are several options or possible actions, such as updating the structure of the database, make an external call, abort a transaction, or inform the user about some situation. The action has a similar context to that of the condition. The context, in this case, determines which data is available to the action.

In general, an event can be explicit or implicit. In the first case, the event must always be given in a rule. It is said that the system supports ECA-rules. If the event is not specified, the rules are called condition-action.

In ECA rules the condition can be optional. Then, if no condition is given, we have event-action rules.

Execution Model

The execution model determines how the rules are managed at execution time. This model is strongly dependent on the particular implementation, however, it is possible describe it in general by using a set of common activities or phases:

- *Signaling* begins when some source causes an event occurrence.

- *Triggering* analyzes the event signaled and triggers the rules associated with that event. This is called rule instantiation.
- *Evaluation* evaluates the condition part of instantiated rules. In this phase the *rule conflict set* is built containing every rule with satisfied conditions.
- *Scheduling* determines how the conflictive rules will be processed.
- *Execution* runs the corresponding actions from the instantiated rules with satisfying the conditions.

How these phases are synchronized depend on the so-called coupling modes of ECA rules. There are two coupling modes: Event-Condition (E-C) and Condition-Action (C-A). We describe them as follows:

1. E-C coupling mode: Determines when the condition is evaluated, considering the triggering event produced by some source. In this case, we have three different options available:
 - Immediate coupling, where the condition is evaluated as soon as the event has happened.
 - Delayed coupling, where the evaluation of the condition part of the rule is not performed immediately after the event triggering but is delayed until something happens before the commit of the transaction.
 - Detached coupling, where the condition is evaluated in a different transaction from the one triggering the event.
2. C-A coupling mode: determines when the action is executed, considering the condition evaluation. The same options as for E-C are applicable in the C-A mode.

Activation time is a concept that fixes the position of the signaling phase with respect to the event occurrence. It can be expressed by using a temporal modal operator such as *before*, *after*, *while*, and so on.

Transition granularity is a concept used in analyzing the relationship between event occurrences and rule instantiations. This relationship can be one-to-one when the transition granularity is *elementary*. In this case, one event occurrence triggers one rule. The relationship can also be many-to-one, when the transition granularity is *complex*. In this case, several event occurrences trigger one rule.

Net Effect Policy is a feature that indicates whether it should be considered the net effect of several event occurrences or each individual occurrence. The prototype database system Starburst, as an example, computes the net effect of event occurrences as follow:

- If an instance is created and possibly updated, and then deleted, the net effect is null.
- If an instance is created and then updated several times, the net effect is the creation of the final version of the instance.
- If an instance is updated and then deleted, the net effect is the deletion of the instance.

Cycle Policy is related to what happens when an event is signaled as a consequence of a condition evaluation or an action evaluation in a rule. We consider two options: iterative and recursive. In the former case, the events signaled by a condition or an action evaluation are combined with those generated from the event sources and, consequently, are then consumed by the rules. In the latter case, the events signaled by a condition or an action evaluation cause the condition or action to be suspended in such a way that the rules monitoring the events can be processed immediately.

In the Scheduling phase, the order of rule execution is to be determined when multiple rules are triggered simultaneously. Hence, the scheduler must consider the choice for the next rule to be fired, applying some conflict resolutions policies, and the number of rules to be fired. The latter presents several options such as: a) to fire all rules sequentially; b) to fire all rules in parallel; c) to fire all instantiations of a rule, before considering any other rule.

Termination and Confluence

Even though active database systems are very powerful, the development of applications can be difficult, mainly because of the unstructured and unpredictable nature of rule processing. This is represented, basically, by rule interaction. Two important properties related to this problem are *termination* and *confluence*. It is said that a rule set is guaranteed to terminate if, for any database state, and initial modification, rule processing cannot continue for ever. A rule set is confluent if, for any database state, and initial modification, the final database state after rule processing is independent of the order in which the activated rules are executed.

In the last few years, many researchers have developed techniques that allow knowing in advance if a rule set has the properties of termination and confluence. These techniques statically analyze a rule set before setting the rules in the database. In particular, Baralis & Widom (2000a) analyze some techniques for performing static analysis of Event-condition-Action and Condition-Action rules. These techniques allow us to determine

when the condition of one rule is affected by the action of other rules, and to determine if two rule actions commute.

In the commercial systems side the approach consists of imposing syntactic limitations, in order to guarantee termination or confluence at runtime, although in other cases counters are used to prevent infinite execution.

INTEGRATING ACTIVE RULES AND DECLARATIVE CONSTRAINTS

Let's get started by describing how kernels of present commercial DBMS support active rules and declarative constraints together.

Today, almost every commercial relational DBMS to some degree adheres to the proposal of the SQL-1999 standard. This standard establishes, in a more or less accurate way, how active rule mechanisms and declarative constraints should be defined and integrated.

Declarative Constraints

We assume the reader is already familiar with SQL constraints, so we simply start with a brief introduction here to ease further comprehension. In a SQL-1999 compliant system, four classes of declarative constraints are supported: *check predicate constraints*, *referential constraints*, *assertions*, and *view check options*. Check predicate constraints aim at validating conditions against the actual state of *one* table in the database and include *primary key* and *unique* definitions, *not null* column definition, and *explicit check* clauses that validate general predicates on the values of some of the columns of the table. Referential constraints aim at guaranteeing that a many-to-one relationship holds on the actual state of two tables: the *referencing* or *child* table, and the *referenced* or *parent* table. A many-to-one relationship ensures that the column values of a foreign key (a list of columns of the referencing table) match the column values of a candidate key (a list of columns of the referenced table). Assertions aim at validating general predicates on rows in *different* tables. View check options deal with the problem of admitting modification operations through cascade defined views, yet retaining the natural meaning of the operations.

A declarative constraint can be declared as having a deferrable or a nondeferrable activation time. However, we limit our analysis in the chapter to non-deferrable constraints only. The reader interested in a more thorough vision of constraint activation time may refer to the standard documents. For example, suppose we have defined the following table schemas:

- `invoice(invoice_number, customer, date, item_total);`

- detail (invoice_number, item_id, quantity);
- goods (item_id, price, quantity);

Declarative constraints for these tables could be:

- c1: PRIMARY KEY (invoice_number), on table invoice;
- c2: PRIMARY KEY (invoice_number, item_id), on table detail;
- c3: PRIMARY KEY (item_id), on table goods;
- c4: invoice_number REFERENCES INVOICE(invoice_number), on table detail CASCADE;
- c5 item_id references GOODS (item_id), on table detail RESTRICT.

Most of the declarative constraints included in the standard are currently supported by almost every SQL-1999 compliant DBMS in the marketplace. Exceptions arise, however. Complex conditions on rows in a table, like nested predicates and predicates involving aggregation, although allowed to appear in explicit check clauses by the standard, are rarely supported nowadays in commercial systems. Assertions are scarcely seen in commercial systems, either. We put off discussing these exceptions for the moment; we will proceed with them in a later section.

Triggers

In SQL-1999 an active rule defined by the user is called a *trigger*, which is a schema object in a database. The trigger structure is defined as follows:

- CREATE TRIGGER <trigger_name> [BEFORE | AFTER] [<event > | <events >]
- ON <table >
- REFERENCING NEW AS <new_value> OLD AS <old_value > NEW TABLE AS <new_table> OLD TABLE AS <old_table >
- FOR EACH [ROW | STATEMENT] WHEN <condition > <action >

Events can be statements INSERT, DELETE, or UPDATE <list >; <table > must be the name of a defined base table or view name, <list > a list of column names of table <table >. When understood from the context, the list of columns in UPDATE statements is omitted. As we have pointed out before, we do not treat triggers on views here. From now on, a trigger event is therefore a modification operation on a base table. The

activation time is specified by keywords BEFORE or AFTER, thus yielding before and after triggers. Before triggers fire immediately before the operation specified as the trigger event has been issued. After triggers fire immediately upon operation completion. The referencing clause admits defining correlation variables for transition values and transition tables, which allows the trigger to access column values in the affected rows before and after the execution of the modification operation. The transition granularity is specified by clause FOR EACH, and can be either set to ROW or STATEMENT. Hence, row level and statement level triggers can be defined. Row level triggers fire one instantiation for each row affected by the modification operation. Provided no row is affected, a row level trigger is never instantiated. Statement triggers fire only once per statement invocation and are evaluated even in the case the event does not happen to affect any row. For example, triggers for the tables defined above are:

```

TRIGGER t1: AFTER DELETE ON invoice
    REFERENCING OLD AS old_inv_t FOR EACH
    STATEMENT
    WHEN exists ( select * from old_inv_t where
        old_inv_t.date > actual_date )

        raise error and undo the delete operation;

```

Trigger t1 prevents the user from removing future pendant invoices.

```

TRIGGER t2: AFTER DELETE ON detail
    REFERENCING OLD AS dtl FOR EACH ROW
    update goods set quantity = goods.quantity - dtl.quantity
    where goods.item_id=dtl.item_id.;

```

Trigger t2 updates the stock of an item whenever the item is removed from the detail of an invoice.

Viewing Constraints as Rules

Integrating triggers with declarative constraints has proved to be a nonsimple task, due to subtleties present in actual implementations. Signaling, triggering and scheduling models for active rules turn out to be non-uniform among database vendors, thus compromising the clear understanding of the meaning of active rules in general.

Moreover, a SQL constraint, although specified in a declarative manner, cannot be regarded simply as a passive component. A declarative constraint includes, explicitly or implicitly, the specification of repairing actions. Hence, declaring a SQL constraint may be thought of as entailing the activation of internal active rules that enforce repairing actions whenever the constraint is violated. Bulk data import and load operations are different matters, of course, but these operations are normally supported by special utility

packages and not by the kernel itself. Concede us putting away import-export operations, therefore.

In summary:

1. Once a *check* constraint is declared, two after rules for events INSERT and UPDATE candidate key, respectively, become active on the (target) table where the constraint is defined, with statement level granularity, and a condition defined so as to be satisfied whenever the associated predicate is violated. The action part for both rules consists in the execution of a controlled rollback undoing the effects of the application of the modification operation. For instance, constraint c1 implies that a constraint rule with statement level granularity become active, having INSERT as the rule event, invoice as the target table, and predicate:
 2. **exists** (**select** * **from** invoice, ?(invoice,insert)^{new}
 3. **where** invoice.invoice_number =
 - ?(invoice,insert)^{new}.invoice_number
 4. **and not** (invoice.ROWID = ?(invoice,insert)^{new}.ROWID))

as the rule condition. In the predicate above, ? (invoice,insert)^{new} stands for the new transition table, and ROWID stands for a dummy column containing the identifier of each row in the table.

2. Whenever a referential integrity constraint is declared, the activation of the following internal active rules are generated:
 - a. Two after rules for events INSERT and UPDATE foreign key, respectively, on the referencing table, with statement level granularity, and a condition stating that there exists at least one row in the new transition table whose foreign key value does not match the candidate key value of any row in the referenced table. As it is the case with check constraints, the action prescribed by these rule specifies a controlled rollback. For instance, constraint c4 entails the activation of a rule for event UPDATE invoice_number on table detail, with predicate:
 - b. **exists** (**select** * **from** ?(detail,update)^{new}
 - c. **where** ?(detail,update)^{new}.invoice_number **not in**
 - (**select**
 - d. invoice_number **from** invoice)),

as the rule condition. $?(detail,update)^{new}$ above stands for the new transition table.

- e. Providing that the repairing action for constraint violation is neither RESTRICT nor NO ACTION, two after rules for events UPDATE candidate key and DELETE, respectively, on the referenced table, with row level granularity, and a condition stating that there exists at least one (dangling) row in the referencing table whose foreign key value matches the old value of the row instantiating the rule. For instance, constraint c4 entails the activation of a rule for event DELETE on table invoice, with row granularity, and predicate:

```
exists (select * from detail where
?(invoice,delete)old.invoice_number =
        detail.invoice_number)
```

as the rule condition. $?(invoice,delete)^{old}$ above stands for the old value of each row.

The firing of any of these rules would carry out the execution of an UPDATE operation that sets the foreign key value of each dangling row in the referencing table to null or a default value (options SET NULL and SET DEFAULT, respectively), or the execution of a DELETE operation, removing all dangling rows from the referencing table (option CASCADE). For instance, the constraint rule for event DELETE on table invoice associated with constraint c4 has the SQL command:

```
delete from detail where detail.invoice_number =
        ?(invoice,delete)old.invoice_number
```

as the rule action. Again, $?(invoice,delete)^{old}$ stands for the old value of the row being deleted.

6. Providing the repairing action for constraint violation is RESTRICT or NO ACTION, two after triggers on the referenced table, for events UPDATE candidate key and DELETE, respectively, with statement level granularity, and a condition stating that there exists at least one row in the referencing table whose foreign key value matches the candidate key of a

row in the old transition table. For instance, constraint c5 implies the activation of a rule for event DELETE on table goods, with predicate:

7. **exists** (select * from detail,
d_k(goods,delete)^{old}
8. **where** d_k(goods,delete)^{old}.item_id = detail.item_id)

as the rule condition. d_k(detail,delete)^{old} stands here for the old transition table (the notation will be clarified later). The firing of any of these rules would carry out the failure of the modification operation and a controlled rollback undoing all changes.

Up to this point, the reader may wonder if declarative constraints and triggers are all the same thing. Despite their similarities, declarative constraints and constraint rules must be distinguished.

First, declarative constraints should be processed only after all changes entailed by an SQL modification statement are effectively applied. This is not an arbitrary policy if we accept that a modification statement in SQL may affect many rows at once and some declarative constraints as primary and foreign key definitions involve the analysis of many rows, too.

Second, it is unlikely to suppose that a rule designer is aware, when writing a trigger, of all possible inconsistent states the database could reach. Hence, admitting a lack of consistency when firing a trigger would introduce unpredictable behavior in user applications. The query optimizer could also outperform due to lack of consistency, when a query is prepared for execution in the body of a rule, because the optimizer usually makes use of constraints to simplify execution plans. A declarative constraint, on the contrary, is meant to deal with inconsistent database states. Consequently, a trigger should not be exposed to an inconsistent database state when the evaluation phase of a rule instantiation begins, while a constraint could.

Moreover, a particular user would expect that the success or failure of the execution of a particular statement could be predicted, particularly in the presence of triggers. In a sense, she requires the process to be confluent. Unfortunately, this is not a simple goal to achieve. The outcome of a modification statement may be affected in many ways; by the order in which the rows involved in the modification are processed; by the particular ordering chosen when applying cascade repairing actions to enforce multiple integrity constraints; by the firing of triggers; and so on.

The considerations above imposed the obligation of producing a precise specification on how declarative integrity constraints and constraint rules should be integrated. The actual accepted specification produced by the SQL-1999 standardization committee is based on

a proposal submitted by a research group at the IBM Almaden Research Center (Cochrane, Pirahesh, Mattos, 1996) We proceed now to review the set of recommendations the standard draft establishes on how to proceed when triggers and declarative constraints are specified together.

Binding Variables to Transitions

The execution of an operation e affecting one row of a table t in the database (an elementary modification operation) can be abstracted by the existence of a *transition*, a pair consisting of the state $?(t,e)^{old}$ of the row immediately before the execution starts, and the state $?(t,e)^{new}$ reached when the execution is complete (considering meaningless values as old values in insert operations and new values in delete operations as nulls). Since we have already established that old and new transition variables are defined along with a trigger, and since they can be referred by the condition and action parts of the trigger, transition values have to be saved in memory. A binding has to be provided therefore for each affected row, that links the trigger transition variables to the area in memory that stores the values.

SQL modification operations are essentially bulk operations, so they can be abstracted as sets of elementary transitions. Since the sets of transitions describing the SQL operation must become available for use whenever firing a trigger or a constraint rule, transition tables $?(t,e)^{old}$ and $?(t,e)^{new}$ has to be created so as to store these sets of transitions. A separate working area organized as forming a stack of storage space slots must be provided to hold transition tables.

Evaluating Constraints and Triggers

Suppose an SQL modification operation e , as an INSERT, UPDATE, or DELETE statement, has been issued on a table t . The following steps are followed:

1. The transition old and new values implied by operation e are computed and stored in tables $?(t,e)^{old}$ and $?(t,e)^{new}$ placed in the working space slot on top of the stack.
For example, suppose we have the statement:
delete invoice where invoice_number=15932; and the database instance:
2. invoice = { ..., ?₁ (15932, 'A&R Lmt'd', 10-2-2001, 2), ... }
3. detail = { ..., ?₂ (15932, 'AZ532', 15), ?₃ (15932, ' BD225', 3), ... }
4. goods = { ..., ?₄ ('AZ532', U\$S45, 15751), ?₅ ('BD225', U\$S18, 2769), ... }

?_i standing for row numbers:

The transition tables computed for this operation are:

$?(invoice,delete)^{old} = \{ ?_1 (15932, 'A\&R\ Lmt d', 10-2-2001, 2) \}$
 and
 $?(invoice,delete)^{new} = \{ ?_1 (- , - , - , -) \}$

5. A variable k , denoting the round number, is set to 0. Old and new transition tables $?(t,e)^{old}$ and $?(t,e)^{new}$, currently at the top of the stack, are given aliases $d_0(t,e)^{old}$ and $d_0(t,e)^{new}$, respectively, for round 0. Note that when $k=0$, it turns out to be one pair of tables $d_0(t,e)^{old}$ and $d_0(t,e)^{new}$ only, that corresponds to the transitions computed for SQL statement e .
6. For each pair of tables $d_k(t,e)^{old}$ and $d_k(t,e)^{new}$, before triggers with e in $\langle events \rangle$ and t as $\langle table \rangle$ are considering for application, on a one by one basis, according to a global ordering criteria, and enter their signaling phase. Statement triggers are considered first whether currently selected table $d_k(t,e)^{old}$ is empty, providing that they have not been fired yet. Row level triggers are fired next, once for each row in table $d_k(t,e)^{old}$, on a row by row basis. Each row in table $d_k(t,e)^{old}$ generates an instantiation of the trigger, and is attached to the trigger old transition variable. If the event is INSERT or UPDATE, and the trigger action updates its new transition variable, the corresponding row in table $d_k(t,e)^{new}$ is updated. If an error occurs or is raised when executing the action part of the trigger, or an attempt to modify the database is made, the entire process breaks down, all changes to the database are undone and an error is reported. If $d_k(t,e)^{old}$ is empty, no before trigger is instantiated.
7. The database is modified according the contents in table $d_k(t,e)^{new}$. Inserts are carried out first, updates are performed next, and deletes are postponed to the end. In our example, row $?_1$ is removed from table invoice. Note that any row in $d_k(t,e)^{new}$, modified in the previous step, due to the execution of a before trigger action that modifies its new transition variable, implies that a modification to the corresponding database table applies here.
8. The constraints must be checked, so that the database state remains consistent. Recall that constraints are viewed as rules. The first rules to be selected for constraint satisfaction checking are the rules corresponding to referential constraints on Table t that match the event e , and have RESTRICT specified as its repairing action. The reason for this preemption criterion is that referential constraints with RESTRICT semantics are meant to be checked before any cascade action has taken place. Because RESTRICT semantics prescribe undoing all work performed in association with a modification operation that brings out a dangling foreign key, no harm is done if the constraints are chosen on an arbitrary order basis. If the condition in any constraint rule is satisfied (the constraint is violated), the process ends in an error.
9. Rules generated by referential constraints on Table t having cascade repairing actions as SET DEFAULT, SET NULL or CASCADE are considered now. Because repairing actions e' (an update statement for SET DEFAULT, a delete statement for CASCADE) refer to the parent table, let's call it t' , new intermediate transition tables $d_{k+1}(t',e')^{old}$ and $d_{k+1}(t',e')^{new}$ are generated in the working storage, before any change is effectively made. Many new intermediate transition tables

may appear as a result. In our example, when $k=0$, constraint $c4$ activates an instantiation of a rule that entails the execution of the SQL command:

```

10.      delete from detail where detail.invoice_number =
11.      ?(invoice,delete)old.invoice_number

```

with row $?_1$ in $d_0(\text{invoice}, \text{delete})^{\text{old}}$ replacing $?(invoice, delete)^{\text{old}}$, so yielding the transient tables $d_1(\text{detail}, \text{delete})^{\text{old}}$ as consisting of rows $?_2$ (15932, 'AZ532', 15) and $?_3$ (15932, 'BD225', 3), and $d_1(\text{detail}, \text{delete})^{\text{new}}$ as consisting of rows $?_2$ (- , - , -) and $?_3$ (- , - , -).

12. The contents of all tables $d_k(t, e)^{\text{old}}$ and $d_k(t, e)^{\text{new}}$ are appended to the contents of tables $? (t, e)^{\text{old}}$ and $? (t, e)^{\text{new}}$, as long as $k > 0$. Tables $? (t, e)^{\text{old}}$ and $? (t, e)^{\text{new}}$ are created and allocated in the current working area (at the top of the stack), on the condition that they do not already exist. If no transient table with subscript $k+1$ exists, then there are no pending cascade actions to be applied, so the resultant state of the database must be checked over for constraints not entailing cascade actions or restricting semantics. This checking process is described in the next step. If at least one transient table with subscript $k+1$ exists, variable k is updated to $k+1$, and the process is resumed at step 3.

In our example, when $k=1$:

- step 3 performs no action, because no before triggers are associated with table detail;
 - rows $?_2$ and $?_3$ are removed from table detail in step 4;
 - no constraint with restrict semantics is violated, so step 5 performs no action;
 - no referential constraint with cascade action is defined on table detail, so no transient table is generated with subscript 2.
13. Check constraints and referential constraints with NO ACTION semantics are considered. The associated rules are fired then, as long as the rule target table matches the table argument t of any transient table $? (t, e)^{\text{new}}$, and the event argument e is the firing event in the rule. If the condition of any of these rules holds (the constraint is violated), then the entire process fails, all modifications to the database are undone, and an error is returned. If none of the conditions are satisfied, then the database is consistent, and after triggers would apply safely. In our example, when $k=0$, no rules for primary key constraints $c1$, $c2$ and $c3$ are fired, because, whereas constraint $c1$ matches the table argument in transient table $?(invoice, delete)^{\text{new}}$ and constraint $c2$ matches the table argument in transient table $? (detail, delete)^{\text{new}}$, their event argument is neither UPDATE nor INSERT.

14. After triggers call for attention now. For each pair of existing tables $? (t,e)^{old}$ and $? (t,e)^{new}$, after triggers with e in $\langle events \rangle$ and t as $\langle table \rangle$ are considered for application, on a one by one basis again, according to the global ordering. Row level triggers are considered first in this case, once for each row in current table $? (t,e)^{old}$, on a row by row basis. As it was the case with before triggers, each row in table $? (t,e)^{old}$ generates an instantiation of the trigger, and a binding to the trigger's old transition variable is established for each row. If table $? (t,e)^{new}$ is empty, no row level trigger is instantiated and subsequently fired. Statement triggers on table t for event e are fired providing that they have not been fired before, table $? (t,e)^{new}$ exists, and all row level after triggers have already been instantiated and fired. The new transition variable is useless in the case of after triggers; issuing an update of such a variable makes no sense. Whereas failure is treated identically to how it was treated in the case with before triggers, attempts to execute SQL modification operations against the database must receive a different treatment; they are allowed to occur in the action part of after triggers. Hence if we recall that an SQL modification operation e' on a table t , occurring in the action part of the trigger entails the presence of transitions, and that these transitions should be saved, tables $d(t,e)^{old}$ and $d(t,e)^{new}$ will be created to contain the new transitions.

In our example, trigger $t2$ is instantiated twice, because table $? (detail,delete)$ has two rows ($?_2$ and $?_3$). The instance of $t2$ corresponding to row $?_2$ entails the execution of the update statement:

```
t2(?2): update goods set quantity = goods.quantity - 15
      where goods.item_id = 'AZ532';
```

The instance of $t2$ corresponding to row $?_3$ entails the execution of the update statement:

```
t2(?3): update goods set quantity = goods.quantity - 3
      where goods.item_id = 'BD225';
```

Update statements $t2(?_2)$ and $t2(?_3)$ produce the tables

- $d(goods,update)^{old} = \{ ?_4 ('AZ532', U\$45, 15751), ?_5 ('BD225', U\$18, 2769) \}$
- $d(goods,update)^{new} = \{ ?_4 ('AZ532', U\$45, 15736), ?_5 ('BD225', U\$18, 2766) \}$

On the contrary, trigger $t1$ enter its signaling phase, table variable old_inv_t is bound to table $? (invoice,delete)^{old}$, and no action is executed, for condition:


```
exists ( select * from ?(invoice,delete)old where
?(invoice,delete)old.date
> actual_date ) does not hold.
```

15. Finally, if no pair of tables $d(t,e)^{old}$ and $d(t,e)^{new}$ exists or, if there both tables are empty, the process ends successfully. Otherwise, the top of the stack is advanced, each pair of nonempty tables $d(t,e)^{old}$ and $d(t,e)^{new}$ become the new $?(t,e)^{old}$ and $?(t,e)^{new}$ at the top of the stack, and the process is resumed at step 2.

In our example, tables $d(goods,update)^{old}$ and $d(goods,update)^{new}$ are non empty, so they become the sole tables $?(goods,update)^{old}$ and $?(goods,update)^{new}$ at the top of the stack. A new pass is accomplished, starting at step 2. During the second pass, updates to rows $?_4$ and $?_5$ are applied to the database (step 4), and because no constraint is now violated, step 10 ends successfully.

GENERAL STATIC CONSTRAINTS AS TRIGGERS

As was pointed out in the previous section, highly expressive declarative static constraints, as general check conditions and assertions, are rarely supported in commercial systems. Hence, a question is imposed: How can we enforce such constraints, since they are not supported by vendors in the kernel of their products? Fortunately, we have seen that a declarative static constraint, in general, can be viewed as a generator of active rules. We simply need to code appropriate triggers therefore, in order to enforce general static constraints. The transcription of constraints into triggers has received considerable attention in the last decade, and a body of work has dealt with the problem, focusing particularly on SQL (Ceri & Widom, 1990, Ceri, Fraternali, Paraboschi & Tanca, 1995; Baralis & Widom, 2000b).

We will present the main underlying ideas herein. First, we proceed to negate the assertion required to hold, and embed the resultant formula as the condition of a trigger template. If the database product does not allow complicate formulas to appear in the condition part of a trigger, a conditional statement on the result of the evaluation of the formula can be introduced instead in the body of the rule, as a guard for the action to be fired. A trigger template is thus generated for each constraint with this technique in mind.

There is a problem, however, if we follow such an approach. It is necessary to determine which events are candidates to fire the trigger and which tables are the target for these events. An extremely cautious and conservative approach would see any modification operation as potentially able to produce values violating the constraint, and would lead to the generation of many triggers as there are modification operations, checking for event occurrences on every table that appear in the formula. This approach can be improved considerably if we think that there exists a close relationship between modification

operations and query predicates, thus indicating that certain modification operations might not affect the result of certain queries. If these relationships could be analyzed in a systematic manner, the number of triggers to generate in order to emulate the behavior of an assertion could be considerably reduced.

A good method for studying relationships between modification operations and queries is to analyze *propagation*. Propagation consists essentially in treating modification as queries. The fact must not surprise the reader, as long as she realizes that transition tables may be computed by executing a select query on the instance of the table been modified, with the arguments of the modification statement interspersed along the from and where clauses. For example suppose we have a table USER with column names NAME and PASSWORD in its schema. Now suppose we have the following modification statement Upd:

```
Upd:  update USER
      set  PASSWORD=" "
      where NAME=:input_name
```

The transition table for this statement can be computed by executing the select statement:

```
?Upd:  select NAME as NAMEold, PASSWORD as PASSWORDold,
           NAME as NAMEnew, " " as PASSWORDnew
      from USER
      where not NAME=:input_name )
```

The new state of table USER after the modification can be then computed as the result of:

```
( select NAME, PASSWORD
  from USER
 where not NAME=:input_name )
union
( select NAMEnew as NAME, PASSWORDnew as PASS-
  WORD
  from ?Upd )
```

If we have a constraint involving a complex SQL **where** condition q on table USER, we can replace references to table USER by the last SQL expression, to form the propagation of the modification Upd in the query q . We can then study if: a) the query may contain more data after the modification, an *insert* propagation case; b) the query may contain less data after the modification, a *delete* propagation case; c) the query may contain updated data after the modification, an *update* propagation case; d) the query remains unchanged, a *null-effect* propagation case. Cases a-, and c-, lead to the generation of a

trigger for the event UPDATE PASSWORD on the table USER. Cases b- and d- do not lead to generate any trigger.

Several propagation analysis techniques have been devised and developed: algebraic, syntactical, and rule-based. Many of them have been incorporated in products for automatic generation of constraint maintenance, especially for SQL database products. Because these techniques are applied at design time, a certain degree of conservatism is imposed, in the sense of considering the generation of more triggers than what is strictly necessary. Study and improvements consider toward reducing conservatism.

DYNAMIC CONSTRAINTS ENFORCEMENT AND ACTIVE RULES

A different situation arises when a constraint is to be imposed on the evolution of database states, not on single states. Demand of support for dynamic constraint enforcement thus arises. Again, as was the case of static constraints, several attempts have been made to describe and formalize dynamic constraints, all aiming for capturing accurately the meaning of such constraints, thus implementing mechanisms for checking and enforcing the constraints properly. When the approach chosen is to support dynamic constraints directly in the kernel of the database system, a certain modal temporal formalism is needed to describe the meaning of the constraints. If the formalism implies the existence of a user language, the developer would be able to express the constraints in a declarative manner, so simplifying a good deployment of a complex system.

Declarative dynamic constraints are not supported in SQL-1999 at present, so conformant products do not provide any help in the matter. However, a simple family of dynamic constraints, *transition* or *two-state constraints*, can be easily emulated by means of triggers, with almost no cost (Widom & Finkelstein, 1990). Let see how.

Transition Constraints

A transition constraint can be expressed in a declarative manner, associated to a CREATE TABLE statement, by a construct of the form:

```
referencing old as Told
           new as Tnew
check C ( Told, Tnew ) on [modified rows | table ]
```

where T^{old} denotes the state of the table, on which the constraint should be checked, immediately before the *transition event* occurrence; T^{new} denotes the state of the table, immediately after the event has occurred; and C is a SQL **where** condition on tuples from T^{old} and T^{new} . *<action>* stands for an optional line of action to be followed in order to enforce consistence when a violation has taken place. The **on** clause in the constraint specification, the *granularity*, which shows to what extent the table must be checked through (modified rows or the entire table). As it was the case with static check constraints, the repairing action to be followed when the constraint is violated consists

simply in undoing all changes introduced by the transition event. For example, suppose we have a table `SPEED_LIMIT` with a single column `VALUE`. We can assert a transition constraint saying that the speed limit value must remain unchanged. The constraint for table `SPEED_LIMIT` in this case would be:

```
referencing old as old_spl
               new as new_spl
check old_spl.VALUE = new_spl.VALUE on modified rows
```

It is necessary to analyze first which kind of operations may potentially produce a violation of the constraint, in order to check this constraint by means of triggers, as was the case with static constraints. We do not treat the subject here. We simply assume that all modification operations on the table `SPEED_LIMIT` excepting deletions, are potentially dangerous for constraint preservation. Note that this is an extremely conservative position. In our case, insertions do not affect the result of the constraint evaluation; only updates may imply a potential violation.

It is easy to see that the constraint check and enforcement process can be emulated by the trigger:

```
after update on SPEED_LIMIT
referencing old_table as old_spl
new_table as new_spl
for each statement
when exists
    (select *
     from new_spl, old_spl
     where old_spl.ROWID = new_spl:ROWID and
           not old_spl.VALUE = new_spl.VALUE )
undo all changes
```

The reader may wonder why the granularity specified in the trigger is statement and not row. The reason is that a row level trigger fires independently for each row present in the transition tables; while a row change may violate the constraint, another row may satisfy it, thus making it impossible to determine the precise point when the repairing action specified in the constraint should start up.

Note that the trigger can be easily built upon a trigger template if the problem of deciding which events should fire the trigger is solved.

A table granularity can be specified in the constraint, indicating that the constraint must be checked against the entire table instead of the affected rows. A more involved translation is needed in this case. The trigger would be:

```
after update on SPEED_LIMIT
```

```

referencing old_table as old_spl
for each statement
when exists
( select *
from SPEED_LIMIT new_spl, old_SPEED_LIMIT old_spl
where old_spl.ROWID = new_spl:ROWID and
      not old_spl.VALUE = new_spl.VALUE )
undo all changes

```

with old_SPEED_LIMIT standing for:

```

( select * from old_spl ) union
( select * from SPEED_LIMIT where SPEED_LIMIT.
ROWID not in
( select ROWID from old_spl ) )

```

The reason for this apparently complicate query is that because of the granularity specified in the constraint, we need to check the constraint on the entire table, not only on the rows affected by the update. The rows not affected by the update, that is, not satisfying the predicate in the update statement, remain unchanged. Thus, we must test the constraint against them (recall that the constraints refer to transitions, not to states, and some condition may violate them). Note that, in the example before, the constraint is trivially satisfied on non-changing rows, but this is not the general case (note that if the check condition would have been `old_spl.VALUE > new_spl.VALUE`, the constraint would not have been satisfied in the entire table). Nevertheless, an optimization is only possible if we are able to prove that the situation arising when no change is made on the table always entails the constraint satisfaction, as it is the case in the example. The condition in the trigger becomes the same condition as was the case for a row level granularity constraint. On the other hand, if a proof exists that the no change situation always entails that the constraint is not satisfied, we can simplify the trigger by eliminating the condition part. In this almost rare case, the trigger will always be fired after an update. Unfortunately, the implication problem is undecidable in general, so the technique of simplification shown above can be attempted only when the class of the conditions involved guarantees decidability.

A More General Approach: Active Rules as Temporal Logic Expressions

We have hitherto presented constructs expressing declarative transition constraints in a rather intuitive manner. No formal meaning has been produced yet. Now, we must consider the topic in a more formal manner. The semantics of declarative transition constraints should be built upon some temporal formalism. Actually, a class of dynamic constraints broader than transition constraints may be needed and methods for emulation should be developed. We must warn the reader, though, that, because some of the concepts involved are not quite well understood, and performance payoffs for these emulating methods seem to be huge up to now, all these efforts have not entirely succeeded in introducing products and solutions into the marketplace. Nevertheless, we

choose to present an attempt in such a direction to serve as a good example of the problem and the proposed solutions.

A good starting point is to think that every time a modification statement is executed, the state produced by the statement execution is *remembered*. A history h is thus maintained, defined as the sequence of pairs (E_i, S_i) , $i=0$ (the t transitions in h), with E_i an *event* (the name of a modification operation, for instance), and S_i the state of the database immediately before the event occurrence has taken place. Then, we can use a language to express conditions on h itself, rather than on states. Languages expressing conditions on histories (linear discrete structures) have been extensively studied, as in Manna & Pnueli (1992, 1995; Sistla, 1983; Wolper, 1983). We follow the language style of PTL (Sistla & Wolfson, 1995a, 1995b) in the chapter, because it has been conceived with an active database in mind and serves well the purpose to introduce the reader in the topic. Other similar approaches can be found in Chomicki (1992) and Lipeck & Saake (1987). We have augmented its constructs to support the specific events needed to modify data in a SQL-like database, and the concept of transition tables.

In PTL, the syntax of a first order language expressing conditions (SQL **where** clauses, for instance) is augmented with two *modal past temporal* constructs: $?_1$ **since** $?_2$, and **last time** $?_1$, and with an *assignment* construct $?_1$ **provided q as X**. $?_1, ?_2$ stand for well formed formulae in the language, X stands for a query variable, and q stands for a query, a function on states in the history. A special query name is provided, **modified rows**, such that, when evaluated on a state in position i of the history h , returns the identification of all rows affected by the occurrence of E_i . A set of 0-ary predicates, *event* or *location predicates* such as **inserting**, **updating**, and **deleting**, optionally qualified with a table name, is also supported. Variables appearing in a PTL formula are considered *bound*, provided they appear in the leftmost part of an assignment subformula or are table name aliases in a first order subformula (table names in from clauses can be regarded as the identity query). Otherwise, a variable is considered to be *free*. A constraint in PTL is a formula with no free variables.

The semantics for PTL formulas is defined with respect to a history and an evaluation function r which maps variables appearing free in the formula into domain values of the appropriate sort. Satisfaction of a PTL formula in a history h with respect to an evaluation r is defined inductively as follows:

- if $?_1$ is an event formula, then $?_1$ is satisfied by h with respect to r if and only if the event of the last transition in history h , agrees with the event formula.
- if $?_1$ is a non-event atomic formula, then $?_1$ is satisfied by h with respect to r if and only if $?_1$ holds, in a first order sense, in the database state of the last transition in history h .
- If a formula is built upon the usual first order connectives as **not**, **and**, **or**, and so on, a first order criteria for satisfaction is applied.

- if ϕ is a formula of the form ϕ_1 **provided** q **as** X then h satisfies ϕ with respect to r if and only if h satisfies ϕ_1 with respect to an evaluation γ_1 such that $\gamma_1(X) = q(S_i)$, the query predicate evaluated over the state of the transition, and $\gamma_1(Y) = \gamma(Y)$ for each variable $Y \neq X$ appearing free in ϕ_1 .
- if ϕ is a formula of the form **last time** ϕ_1 then h satisfies ϕ with respect to γ if and only if ϕ_1 satisfies h' with respect to γ , with h' resulting from deleting the last transition from history h .
- if ϕ is a formula of the form ϕ_1 **since** ϕ_2 then h satisfies ϕ with respect to γ if and only if there exists a $j = i$, such that history h up to position j satisfies ϕ_2 with respect to γ , and for all positions $k > j$ up to the last position, history h up to k satisfies ϕ_1 with respect to γ .

Other useful modal constructs can be obtained, from the basic ones. For example, the modal operator **first** can be obtained as a synonymous of **not last time true**. Hence, the constraint studied in the previous section, can be expressed in PTL as follows:

```

first or
( last time not exists
  ( select *
    from SPEED_LIMIT old_spl, new_spl
  where old_spl.ROWID = new_spl.ROWID
    and not old_spl.VALUE = new_spl.VALUE ) )
provided modified rows as new_spl

```

It is not easy to appreciate the real expressive power of this language. It is sufficient to say that it can express, not only transition predicates, but more powerful conditions on the whole previous history of the database changes, as complex events and deadlines (conditions on time). The problem here is not merely to determine the real power of a language like the one presented here, but how to detect efficiently eventual violations, and how to repair these violations accurately. The issue is not completely solved nowadays, and is an interesting open area of research. We will postpone the treatment of these problems to an upcoming section, and we will concentrate in presenting the sketch of a correct algorithm that checks for constraint violations.

Checking for Satisfaction of Temporal Logic Constraints

An algorithm has been devised to enforce constraints expressed in PTL. The key idea is to fire a trigger after each modification statement execution, so as to produce and maintain sufficient auxiliary information to detect a constraint violation and react in consequence.

The first step in the devising process of the method is to proceed to negate the temporal formula, to obtain a monitoring formula. For example, the formula in the example above is now presented by its opposite, the monitoring Formula f:

```

not first and
(last time exists
(select *
  from SPEED_LIMIT old_spl, new_spl
 where old_spl.ROWID = new_spl.ROWID
   and not old_spl.VALUE = new_spl.VALUE) )
provided modified rows as new_spl

```

When a modification statement is executed, a trigger is fired and proceeds to compute, for each subformula g in the main formula f , a Boolean expression $F_{g,i}$ (a SQL where clause), the *firing formula*, where i is the position in the history of the modification transition. If the firing formula $F_{g,i}$ evaluates to false then the constraint is satisfied by state S_i . If the state reached after the execution of the i -th modification. $F_{g,i}$ evaluates to true then the constraint is violated, then a correcting action is fired. Note that the trivial correcting action: "undo the last modification" actually works well as a correcting action. It preserves the satisfaction of the constraint.

When the formula contains temporal past operators, as **last time** and **since**, $F_{g,i}$ must be evaluated inductively, referencing previous transitions in the history. To reduce the potential huge amount of space needed to "remember" the whole previous history of the evolution of the database to a minimum, a technique is needed: An auxiliary historic table q^h is maintained for each different query q appearing in the formula. The historic table schema is formed by adding a timestamp column (a position number) and an event name column to the query schema. In what follows, we denote by q^h_i the table containing the result of query q at past state s_i , for any query q appearing in the formula. Tables q^h_i could be easily reconstructed by examining the content of historic tables q^h .

The idea in the computation is to proceed with the elimination of temporal operators, by means of a careful rewriting process of $F_{g,i}$. $F_{g,i}$ is then evaluated inductively as follows:

- If g is an event formula and $i > 0$, $F_{g,i} = \text{true}$ if the event formula agrees the event name firing the trigger, otherwise it evaluates to false.
- If g is a non-event atomic formula (a SQL where clause), two cases must be distinguished. If g has at least one free variable, $F_{g,i} = g'$ where g' is a Boolean expression involving free and bound variables. Otherwise, if all variables are bound in g , $F_{g,i}$ results the formula evaluated in the database state s_i (true or false).
- If $g = g_1$ **and** g_2 , g_1 **or** g_2 , or **not** g_1 , $F_{g,i}$ must evaluate to $F_{g_1,i} ? F_{g_2,i}, F_{g_1,i} ? F_{g_2,i}$, or $\neg F_{g_1,i}$.

- If $g = \text{last time } g_1$ two cases arise: if $i = 0$ then $F_{g, i}$ always evaluates to false; otherwise, if $i > 0$, $F_{g, i} = F_{g, i} [q_1/q_1^{h_{i-1}}] \dots [q_k/q_k^{h_{i-1}}]$, for all queries q_j , $1 = j = k$, in g . $e [X / Y]$ stands for the substitution of all free occurrences of variable X in e by Y .
- If $g = g_1 \text{ since } g_2$, can be reduced to the case of **last time**. $F_{g, i} = F_{g_2, i}$ or $(F_{g_1, i} \text{ and last time } F_{g_2, i})$.
- If $g = g_1 \text{ provided } q \text{ as } X$ then $F_{g, i}$ turn out to be simply $g_1 [X / q]$.

If f is a temporal formula, when $i > 0$ and before any $F_{g, i}$ has been computed, rows appearing in the old transition table T^{old}_i are appended to the auxiliary historic tables, timestamped with $i - 1$. Once any of the $F_{g, i}$ has been computed, all rows associated with queries on states up to the first state mentioned in the newly computed $F_{f, i}$ are deleted from the historic tables. Note that this case arises provided that no **since** operator appears in the formula f .

We will continue with the constraint formula in the example, to see how the algorithm works:

Suppose we have a sequence of two update statements on the table SPEED_LIMIT. The first of these update statements actually happens to produce no change in the column VALUE. The second update, however, changes the column value. The sequence is shown graphically, as follows (Figure 1).

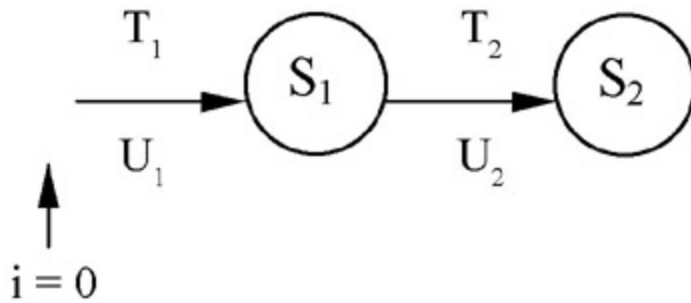


Figure 1: Sequence of two update statements on the table SPEED_LIMIT

U_1 and U_2 in the figure represent the updates. T_1 and T_2 represent the transition tables, with old and new values of modified rows. The current position in the history is indicated by i (In this initial case, we are placed in the situation before any update has been issued, so $i = 0$). The syntactic structure of the formula f is: $g_1 \text{ and } g_2$; g_1 is: **not** g_2 ; g_2 is: **first**; g_3 is: $g_4 \text{ provided modified rows as new_spl}$; g_4 is: **last time** g_5 ;

g_5 is: **exists** (select * from SPEED_LIMIT old_spl, new_spl
 where old_spl.ROWID = new_spl.ROWID
 and not old_spl.VALUE = new_spl.VALUE).

Variables appearing bound in f are old_spl and new_spl . Because both refer to the same query, the identity function applied on table $SPEED_LIMIT$, only one historic table is needed. We will name this table $SPEED_LIMIT^h$. In what follows, we will denote the state of rows in the query $SPEED_LIMIT$, relevant to state s_i , as $SPEED_LIMIT^h_i$. As it was pointed out before, this state can be reconstructed from table $SPEED_LIMIT^h$.

Now, we advance the actual position to 1 (after the execution of update U_1). Real firing of the monitoring trigger is produced. We have the situation ([Figure 2](#)).

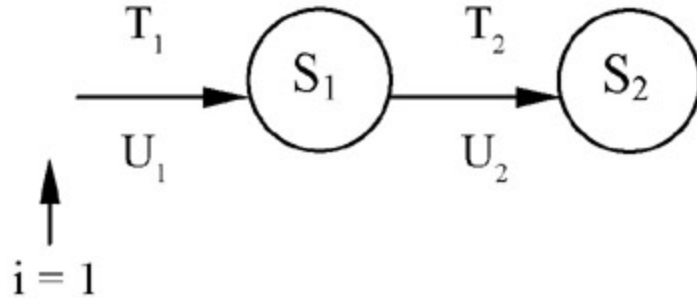


Figure 2: Situation after a real firing of the monitoring trigger is produced

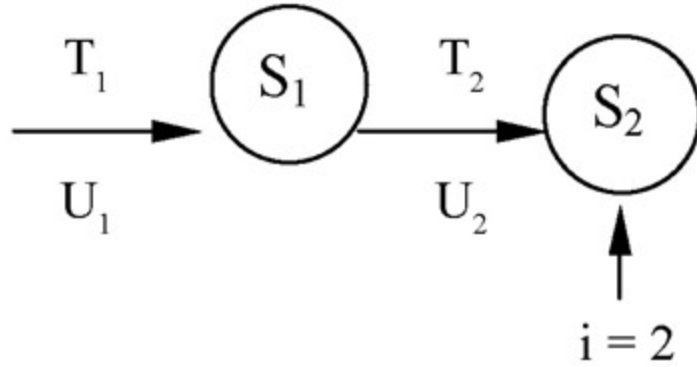


Figure 3: Position 2

All rows in $?T^{old}_1$ are appended to the table $SPEED_LIMIT^h$, with value 0 in the timestamp column, and value update in the event name column.

- $F_{f,1} = F_{g1,1}(S_1) ? F_{g3,1}(S_1) = false ? false = false$
- $F_{g1,1} = \neg F_{g2,1} = false;$
- $F_{g2,1} = true;$
- $F_{g3,1} = F_{g4,1}[new_spl / ?T^{new}_1];$
- $F_{g4,1} = F_{g5,1}[SPEED_LIMIT / SPEED_LIMIT^h_0];$
- and $F_{g5,1} = g5;$

$F_{g2,1} = true$ because i points to the first position in the sequence. $F_{g3,1}$ evaluates to false in state S_1 because the rows modified by U_1 agree in the column $VALUE$ with the

contents of $\text{SPEED_LIMIT}^h_0 = ? T^{old}_1$. Recall that update U_1 does not produce any change in the column VALUE of the table SPEED_LIMIT at the initial state. The formula $F_{f,1}$ evaluates to false in S_1 , then the constraint is not violated in state S_1 . The rows in the table SPEED_LIMIT^h corresponding to position 0 can be deleted safely. This is the first position number mentioned in a query, present in $F_{g3,1}$ (SPEED_LIMIT^h_0).

Let's go on now with position 2 as the current position:

All rows in $? T^{old}_2$ are appended to the table SPEED_LIMIT^h , with value 1 in the timestamp column, and value **update** in the event name column.

- $F_{f,2} = F_{g1,2}(S_2) \wedge F_{g3,2}(S_2) = \text{true} \wedge \text{true} = \text{true}$
- $F_{g1,2} = \neg F_{g2,2} = \text{true};$
- $F_{g2,2} = \text{true},$
- $S_1 \wedge S_2 \wedge F_{g3,2} = F_{g4,2}[\text{new_spl} / ? T^{old}_2]; F_{g4,2} = F_{g5,2}[\text{SPEED_LIMIT} / \text{SPEED_LIMIT}^h_1];$
- and $F_{g5,2} = g_5;$

$F_{g2,1} = \text{false}$ because i does not point to the first position in the sequence $F_{g3,2}$ evaluates to false in state S_2 because the rows modified by U_2 does not agree in the column VALUE with the contents of SPEED_LIMIT^h_1 . Recall that update U_2 changes the column VALUE of at least one row of the table SPEED_LIMIT at the previous state. The formula $F_{f,2}$ evaluates to true in S_2 , then the constraint is violated in state S_2 and a correcting action is needed. Note that, in the case we undo the changes introduced by the last update, the state S_1 is reached again, and the constraint is satisfied at position 1.

Note also that, neither in the composition of $F_{g3,2}$ nor in $F_{g1,2}$, a reference to position 0 is present. The rows in the table SPEED_LIMIT^h , corresponding to position 1 can be deleted. This is the first position number mentioned in a query present in $F_{g3,2}$ (SPEED_LIMIT^h_1).

It is clear that this deletion can be done because no since operator appears in the formula, otherwise a reference to the deleted position would be present.

Applicability, Drawbacks and Partial Remedies

Several drawbacks in the above techniques appear when the main issue is applicability.

1. The processing effort paid off in each instantiation of a rule may compromise seriously the throughput and response time of the system. Recall that the

- monitoring for constraint satisfaction is accomplished every time a modification operation is executed.
2. The amount of space required to store auxiliary historic tables. Formulas with constructs equivalent to since, are good examples of this.
 3. The relationship between constraint satisfaction and transaction processing is not quite clearly established. For example, if several transaction can run concurrently, and some of them does not prevent the others from seeing uncommitted data, repeatable reads isolation level is not granted.

The question is not entirely solved. Partial remedies to the first problem have been outlined: Monitoring of constraint satisfaction may be postponed up to specific event occurrences, and only updating of historic tables is issued in the meantime. Deferring the checking process to commit time or supporting extra-system time events is a possibility. New event types are needed, such as **attempting to commit, at time t**, or pseudo queries returning time values. Logical or net-effect modification operations may also be supported, in opposition to physical operations. An automata based approach has been devised to recognize such logical events from patterns of physical events. This approach could be observed in the ODE object database system from AT&T Bell Laboratories (Gehani & Jagadish, 1991) and in the prototype database rule oriented system Ariel (Hanson, 1992), where so called Ariel-TREAT discrimination networks serve the purpose of testing complex historic patterns. More recent works focused on the specification and detection of complex events (Chakravarthy, Krishnaprasad, Anwar, Kim, 1994; Chakravarthy & Mishra, 1994; Chakravarthy, 1997; Yang & Chakravarthy, 1999), but constraints has not received special attention therein. The second problem has received less attention than the first, but again attempts to solve the space problem has been addressed. For example, in Ariel, some query results could be maintained intentionally, especially when the selection factor is low. The efforts paid in that direction in the early nineties have not been reflected in the database marketplace, but increasing interest is regained presently, specifically in the area of Web servers, and particularly when e-commerce is the issue. The concept of elapsed time and expiration time serves, in this area, to prevent histories from growing indefinitely.

The third problem is the most problematic. (Sistla & Wolfson, 1995) have been defined the concepts of *off-line* and *on-line* satisfaction of a constraint with respect to transactions. A constraint is said to be *off-line satisfied* if it is satisfied at the commit point of all transactions, considering, up to these points, all modification operations of committed transactions. A constraint is said to be *on-line satisfied* if it is satisfied at the commit point of all transactions, considering only modification operations of committed transactions with commit point reached up to these points. These two notions of satisfaction differ with respect to which modifications a transaction could see. *Off-line* implies that a transaction sees all committed work, independently of the commit point. This notion is closer to the notion of a transaction manager guaranteeing cursor stability. *On-line* satisfaction implies that a transaction only sees all previously committed work.

This last notion of satisfaction is closer to the notion of a system guaranteeing a repeatable reads isolation level.

CONCLUDING REMARKS

In this chapter, we have presented a brief survey of the interaction between active rules and integrity constraints. We have discussed the current proposed techniques to deal with situations when both declarative static constraints and triggers are defined. We have shown that the main problem consists in ensuring that the constraints are preserved in the presence of cascade firing of before and after triggers. The perspective of our treatment follows the SQL-1999 Standard Committee point of view, which constitutes the state of the art in that matter. We have given a brief sketch on how to generate triggers for integrity constraint maintenance, manually or automatically, for the static case when such a constraint definition is not supported by database kernels. Then, we have addressed the problem of ensuring satisfaction of dynamic constraints, and we review a formal approach based on temporal logic formalism. We have shown that if the dynamic constraints are simply two-state or transition constraints, the satisfaction problem can be easily implemented by means of triggers. We have also seen that the approach, although formal, can be implemented as well for the general case on actual systems. Some issues concerning applicability related with the last techniques remain open to researchers and practitioners, and improvements in these techniques are expected in the future.

REFERENCES

- Aiken, A., Hellerstein, J., Widom, J. (1995). *Static Analysis Techniques for Predicting the Behavior of Active Database Rules*. *ACM Transactions on Database Systems*, 20:1, 3–41.
- Baralis, E., Ceri, S., Paraboschi, S. (1996). *Modularization Techniques for Active Rules Design*. *ACM Transactions on Database Systems* 21:1, 1–29.
- Baralis, E., Ceri, S., Widom, J. (1993). *Better Termination Analysis for Active Databases*. *Proc. Int'l Workshop on Rules in Database Systems*. 163–179.
- Baralis, E., Widom, J. (2000a). *Better Static Rule Analysis for Active Database Systems*. Stanford Univ. Research Report 02/06/2000.
- Baralis, E., Widom, J. (2000b). *An Algebraic Approach to Static Analysis of Active Database Rules*. *ACM Transactions on Database Systems*, 25:3, 269–332.
- Ceri, S., Cochrane, R., Widom, J. (2000). *Practical Applications of Triggers and Constraints: Successes and Lingering Issues*. *Proc. Int'l Conf. on Very Large Databases*.
- Ceri, S., Widom, J. (1990). *Deriving Production Rules for Constraint Maintenance*. *Proc. Int'l Conf. On Very Large Databases*, 566–577.
- Ceri, S., Fraternali, P., Paraboschi, S., Tanca, L. (1995). *Automatic Generation of Production Rules for Integrity Maintenance*. *ACM Transactions on Database Systems*, 19:3, 367–422.
- Chakravarthy, S. (1989). *Rule Management and Evaluation: An Active DBMS Perspective*. *SIGMOD Record* 18:3, 20–28.

- Chakravarthy, S. (1997). *SENTINEL: An Object-Oriented DBMS with Event-Based Rules*. *Proc. ACM Int'l Conf. SIGMOD*. 572–575.
- Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S. (1994) *Composite Events for Active Databases: Semantic Contexts and Detection*. *Proc. Int'l Conference on Very Large Data Bases*. 606–617.
- Chakravarthy, S., Mishra, D. (1994) *Snoop: an Expressive Event Specification Language for Active Databases*. *Data and Knowledge Engineering*, 14:1, 1–26.
- Chomicki, J. (1992) *History-less Checking of Dynamic Constraints*. *Proc. Int'l Conf. on Data Engineering*. IEEE Computer Society Press.
- Gehani, N., Jagadish, H. (1991). *ODE as an Active Database: Constraints and Triggers*. *Proc. Int'l Conference on Very Large Databases*. 327–336.
- Dayal, U. et al. (1988). *The HiPAC Project: Combining Active Databases and Timing Constraints*. *SIGMOD Record* 17:1, 51–70.
- Hanson, P. (1992). *Rule Condition Testing and Action Execution in Ariel*. *Proc. ACM Int'l Conf. SIGMOD*. 49–58.
- Lipeck, U., Saake, G. (1987). *Monitoring Dynamic Integrity Constraints Based on Temporal Logic*. *Information Systems*, 12:3, 255–266.
- Manna, Z., Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.
- Manna, Z., Pnueli, A. (1995). *The Temporal Logic of Reactive Systems: Safety*. Springer-Verlag.
- Paton, N., Diaz, O. (1999). *Active Database Systems*. *ACM Computing Surveys*, 31:1, 63–103.
- Sistla, A.P. (1983). *Theoretical Issues in the Design of Distributed and Concurrent Systems*. Ph.D. Thesis, Harvard Univ., Cambridge, MA.
- Sistla, A. P., Wolfson, O. (1995a). *Temporal Triggers in Active Databases*. *IEEE Trans. On Knowledge and Data Engineering*. 7:471–486.
- Sistla, A. P., Wolfson, O. (1995b). *Temporal Conditions and Integrity Constraints in Active Databases*. *Proc. ACM Int'l Conf. SIGMOD*. 269–280.
- Stonebraker, M. (1986). *"Triggers and Inference in Database Systems"*, in *On Knowledge Base Management Systems*. Brodie, M. and Mylopoulos, J. Editors. Springer-Verlag.
- Widom, J., Ceri, S. (1996). *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers.
- Widom, J., Finkelstein, S. (1990). *Set-oriented Production Rules in Database Management Systems*. *Proc. ACM Int'l Conf. SIGMOD*, 259–270.
- Wolper, P. (1983). *Temporal Logic Can Be More Expressive*. *Info. and Cont.*, 56:72–99
- Yang, S., Chakravarthy, S. (1999). *Formal Semantics of Composite Events for Distributed Environment*. *Proc. Int'l Conf. on Data Engineering*, 400–407. IEEE Computer Society Press.

Chapter V: Integrity Constraints in Spatial Databases

Karla A. V. Borges, Federal University of Minas Gerais and Prodabel—Empresa de Informática e Informação do Município de Belo Horizonte,

Brazil

Clodoveu A. Davis, Jr., Prodabel—Empresa de Informática e Informação do Município de Belo Horizonte,

Brazil

Alberto H. F. Laender, Federal University of Minas Gerais,

Brazil

INTRODUCTION

A number of integrity constraints must be observed when updating a database, to preserve the semantics and the quality of stored data (Elmasri & Navathe, 2000). Achieving and preserving the integrity of data is an established field in the database area. However, within the scope of geographic applications, special problems come up due to the locational aspects of data (Plumber & Groger, 1997). Most geographical information systems (GIS) use data that depend on topological relationships, and sometimes these data must be explicitly represented in the database, requiring special attention for the maintenance of the semantic integrity. Enforcing the integrity constraints must be considered one of the main implementation goals (Borges et al., 1999). Thus, it is convenient to explicitly specify on the geographic application schema the situations where the constraints cannot be disregarded. Many mistakes in the data entry process could be avoided if digitizing processes based on these constraints were implemented.

Even though there is a very active research area interested in the design of robust and efficient spatial databases, the inability of current GIS regarding the implementation and management of spatial integrity constraints is still evident (Plumber & Groger, 1997; Worboys, 1994). A modification in a spatial database may cause simultaneous updates in a large number of records in multiple files, making it hard to manage all the environment. A very sophisticated control is required to avoid redundancy and loss of integrity.

In the traditional database systems approach, there is a relationship between conceptual, logical, and physical design, in which, through mapping operations, constraints that are identified in the conceptual schema are inherited and transformed into implicit constraints expressed by the data definition language (DDL) or into explicit constraints coded in the application programs (Elmasri & Navathe, 2000). This relationship must also exist in spatial information systems, so that spatial constraints can be likewise identified and implemented.

Improvement of quality is one of the key objectives of establishing integrity constraints in spatial databases (Cockroft, 1997). It is possible to improve data quality by enforcing constraints upon data entered into a database. These constraints must be identified and recorded at the database design level. However, it can be perceived that modeling geographic data requires models that are more specific and capable of capturing the semantics of geographic data, offering higher abstraction mechanisms and implementation independence (Borges, 1997; Câmara, 1995). There are particular characteristics of geographic data that make modeling more complex than in the case of conventional applications. Within the geographic context, topologic relations and other spatial relationships are fundamentally important to the definition of spatial integrity rules. In geographic applications, topological and other spatial relationships are translated into topological integrity constraints among database entities, taking a relevant role in the data entry/updating process. "The imposition of such constraints on data entry/update is considered to have potential for the reduction of errors in data input and hence improvement in data quality" (Cockroft, 1997, p. 341).

This chapter addresses the relationship that exists between the nature of spatial information, spatial relationships, and spatial integrity constraints, and proposes the use of OMT-G (Borges et al., 1999; Borges et al., 2001), an object-oriented data model for geographic applications, at an early stage in the specification of integrity constraints in spatial databases. OMT-G provides appropriate primitives for representing spatial data, supports spatial relationships and allows the specification of spatial integrity rules (topological, semantic and user integrity rules) through its spatial primitives and spatial relationship constructs. Being an object-oriented data model, it also allows some spatial constraints to be encapsulated as methods associated to specific georeferenced classes. Once constraints are explicitly documented in the conceptual modeling phase, and methods to enforce the spatial integrity constraints are defined, the spatial database management system and the application must implement such constraints.

This chapter does not cover integrity constraints associated to the representation of simple objects, such as constraints implicit to the geometric description of a polygon. Geometric constraints are related to the implementation, and are covered here in a higher level view, considering only the shape of geographic objects. Consistency rules associated with the representation of spatial objects are discussed in Laurini and Thompson (1992).

CLASSIFICATION OF SPATIAL INTEGRITY CONSTRAINTS

One important activity in the design of a schema for a particular database application consists of identifying the integrity constraints that must hold on the database. The main types of integrity constraints that occur frequently in database modeling are *domain constraints*, *key and relationship structural constraints*, and *general semantic integrity constraints* (Elmasri & Navathe, 2000). Cockroft (1997) extends that classification in

order to encompass the peculiarities of spatial data. This classification is based on the distinction between topological, semantic, and user rules, as follows.

Topological integrity constraints Topology is the study of geometrical properties and spatial relations. There has been some theoretical research into the principles of formally defining spatial relationships (Egenhofer & Franzosa, 1991). These principles can be applied to application-specific entities and relationships to provide a basis for integrity control. Area subdivision is an example of this constraint. One city's administrative regions must be contained within the city limits, and there must be no spot in the municipal territory that belongs to more than one administrative region or to none.

Semantic integrity constraints These constraints are concerned with the meaning of geographic features. Semantic integrity constraints apply to database states that are valid by virtue of the properties of the objects that need to be stored. An example of this constraint is the rule that does not allow a building to be intercepted by a street segment.

User defined integrity constraints User defined integrity constraints allow database consistency to be maintained as defined by the equivalent of "business rules" in non-spatial DBMS. This type of constraint acts, for instance, on the location of a gas station, which, for legal reasons, must lie farther than 200 meters from any existing school. The municipal permitting process must consider this limitation in its analysis. In another example, engineering limitations regarding the minimum allowable slope must be observed while installing sewer pipes. User-defined constraints may be stored and enforced by an active repository.

According to Elmasri and Navathe (2000), every data model has a set of built-in constraints associated with its constructs. The OMT-G model allows several spatial integrity rules to be derived from its primitives. These rules constitute a set of constraints that must be observed in the operations that update a geographic database.

The GIS can include features that enforce the fulfillment of some spatial integrity rules, but most require the definition of integrity control operations to be associated with the classes. In most cases, such operations must be implemented by the application's developer. Controlling the integrity constraints must be considered one of the main implementation activities. It is convenient to have the geographic application schema to reinforce at least the situations where this control cannot be disregarded. Many mistakes in the data entry process could be avoided if digitizing procedures based on these constraints are implemented. However, the approach usually employed by commercial GIS products is rather different, since rarely the integrity constraints are enforced by the interactive data entry procedures. In general, inconsistent information is allowed to enter the database, through import functions; later, a range of correction functions is used to "clean up" the data, verifying its consistency.

Both in the case of integrity constraints and consistency detection, there is the need for some mechanism that will allow the relaxing of the constraints in special situations. For instance, a semantic constraint could naturally establish that streets cannot cross buildings.

However, there are some situations, such as blocks of buildings connected by overpasses, in which this rule would need to be relaxed (Laurini, 2001).

Topological integrity constraints are achieved through spatial dependence, spatial association, connectivity, and geo-fields rules. Likewise, semantic integrity constraints are achieved through spatial association and disjunction rules. User-defined integrity constraints are, in turn, obtained from methods that can be associated to the classes. The implementation of any of these rules is dependent on the underlying GIS. Some of them are available as internal functions, while others must be implemented by the developer of the application, using the programming language provided with the GIS.

To adequately explain such integrity constraints, we must first present OMT-G in more detail. Later, we will describe formally each integrity constraint that can be derived from the OMT-G primitives.

THE OMT-G MODEL AND SPATIAL INTEGRITY CONSTRAINTS

Model Overview

Starting from the primitives of the UML class diagram, geographic primitives were introduced with the objective of increasing its semantic capabilities, thereby reducing the distance between the mental model of the space to be modeled and the usual representation model. Therefore, OMT-G provides primitives to model the geometry and topology of geographic data, providing support for "whole-part" topologic structures, network structures, multiple views of objects, and spatial relationships. Besides, the model allows for the specification of alphanumeric attributes and associated methods for each class. The main strong points of the model are its graphic expressiveness and its representation capabilities, since textual annotations are replaced by the drawing of explicit relationships, representing the dynamics of the interaction between the various spatial or non-spatial objects.

The OMT-G model is based on three main concepts: *classes*, *relationships*, and *spatial integrity constraints*. Classes and relationships are the basic primitives that are used to create application schemas with OMT-G. For that purpose, OMT-G proposes the use of three different diagrams in the process of designing a geographic application (Borges et al., 2001; Davis, 2000). The first and more usual one is the *class diagram*, in which all classes are specified, along with their representations and relationships. From this schema, it is possible to derive a set of spatial integrity constraints that must be observed in the implementation. When the class diagram indicates the need for multiple representations of any class, or when the application involves the derivation of some class from others, a *transformation diagram* must be built. In it, all transformation processes can be specified, allowing for the identification of any required methods for the implementation. Finally, a *presentation diagram* must be built to provide guidelines for the visual aspect of objects

in the implementation. There can be several visual aspects for any given class, which allows the definition of a view or set of views for each application or group of users.

The next sections cover the primitives used to build class diagrams, from which spatial integrity constraints can be obtained. Transformation and presentation diagrams are not covered here. For more information on the use of these tools for the specification of geographic applications, including multiple representation and multiple presentation aspects, see Borges et al. (2001).

Classes

The classes defined by the OMT-G model represent the three main groups of data (continuous, discrete, and non-spatial) that can be found in geographic applications, thereby allowing for an integrated view of the modeled space. The classes can be *georeferenced* or *conventional*.

The distinction between georeferenced and conventional classes allows different applications to share non-spatial data, therefore making it easier to develop integrated applications and to reuse data (Oliveira et al., 1997). A *georeferenced class* describes a set of objects that have spatial representation and are associated to features on Earth (Câmara, 1995), assuming the fields and objects view as proposed in Frank & Goodchild (1990 and Goodchild (1992). A *conventional class* describes a set of objects with similar properties, behavior, relationships, and semantics, and which can have some sort of relationship with spatial objects, but which do not have geometric or geographic properties.

Georeferenced classes are specialized into *geo-field* and *geo-object* classes. Geo-field classes represent objects and phenomena that are continuously distributed over the space, corresponding to variables such as soil type, relief, and mineral contents (Câmara, 1995). Geo-object classes represent individual, particular geographic objects, which can usually be traced back to real world elements, such as buildings, rivers, and trees. A georeferenced class is symbolized by a rectangle, subdivided in three sections ([Figure 1a](#)). The top section carries a pictogram in its left side to indicate the geometry of the representation. Adding pictograms to the primitive element used to portray geographic classes (instead of using relationships to describe the geometry of the object) significantly simplifies the final schema. The notation used for conventional classes corresponds to the notation used in the UML (Rational, 1997). Objects may or may not have non-spatial attributes, listed in the middle section of the complete representation. Associated methods or operations are specified in the lower section. A simplified symbolization can be used both for georeferenced and conventional classes, leaving out the bottom section and listing only the main attributes in the middle section ([Figure 1b](#)).

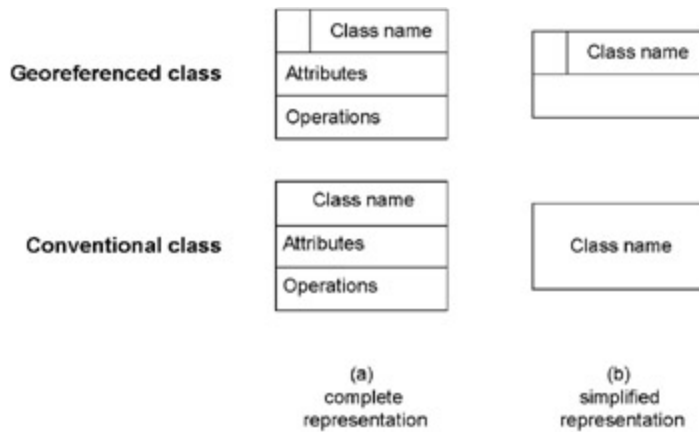


Figure 1: Graphic notation for the basic classes

OMT-G presents a fixed set of geometric types, using a symbolic representation that distinguishes geo-object and geo-field classes within a georeferenced class ([Figures 2 and 3](#)). The next subsections present details on geo-field and geo-object classes.

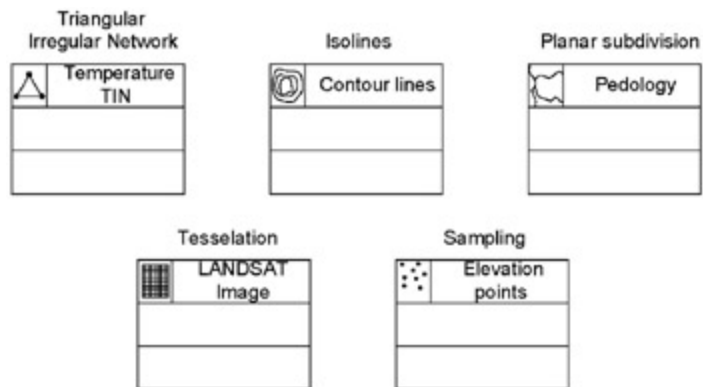


Figure 2: Geo-field classes

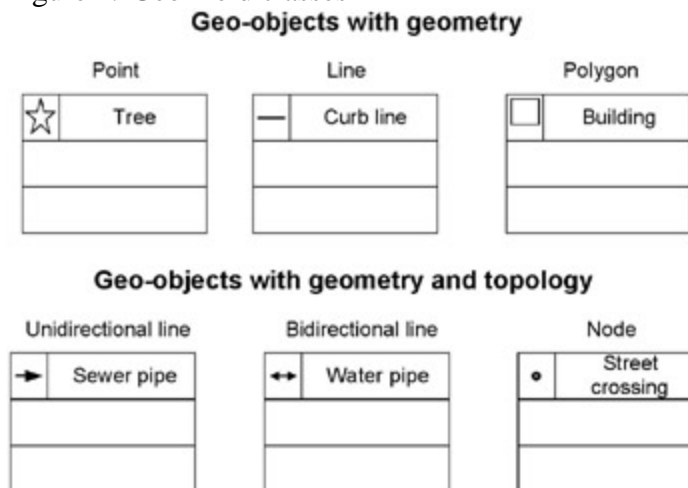


Figure 3: Geo-object classes

Geo-fields

OMT-G has five geo-field descendant classes: isoline, planar subdivision, tessellation, sampling, and triangular irregular network (Figure 2). From the semantics involved in the concept of geo-fields, and from the specific definition of these classes, some spatial integrity rules can be deduced. These rules constitute a set of constraints that must be observed in the operations that update the geographic database. In the case of the geo-field primitives, the spatial integrity rules listed in Table 1 can be derived. These rules are mostly derived from the semantics of the geo-field descendant classes.

Table 1: Geo-field integrity rules

Planar Enforcement Rule	<p>1. Let F be a geo-field and let P be a point such that $P \in F$. Then a value $V(P) = f(P, F)$, i.e., the value of F at P, can be univocally determined.</p>
Isoline	<p>2. Let F be a geo-field. Let v_0, v_1, \dots, v_n be $n+1$ points in the plane.</p> <p>Let a_0, a_1, \dots, a_n be n segments, connecting the points. These segments form an <i>isoline</i> L if, and only if, (1) the intersection of adjacent segments in L is only the extreme point shared by the segments (i.e., $a_i \cap a_{i+1} = v_{i+1}$), (2) non-adjacent segments do not intercept (that is, $a_i \cap a_j = \emptyset$ for all i, j such that $j \neq i+1$), and (3) the value of F at every point $P \in L$ such that $P \in a_i$, $0 \leq i \leq n-1$, is constant.</p>
Tessellation	<p>3. Let F be a geo-field. Let $C = \{c_0, c_1, c_2, \dots, c_n\}$ be a set of regularly-shaped cells covering F. C is a <i>tessellation</i> of F if and only if for any point $P \in F$, there is exactly one corresponding cell $c_i \in C$ and, for each cell c_i, the value of F is given.</p>
Planar Subdivision	<p>4. Let F be a geo-field. Let $A = \{A_0, A_1, A_2, \dots, A_n\}$ be a set of polygons such that $A_i \in F$ for all i such that $0 \leq i \leq n-1$. A forms a <i>planar subdivision</i> representing F if and only if for any point $P \in F$, there is exactly one corresponding polygon $A_i \in A$, for which a value of F is given (that is, the polygons are non-</p>

	overlapping and cover F entirely).
Triangular Irregular Network	<p>5. Let F be a geo-field. Let $T = \{T_0, T_1, T_2, \dots, T_n\}$ be a set of triangles such that $T_i \cap F$ for all i such that $0 \leq i \leq n - 1$. T forms an <i>triangular irregular network</i> representing F if and only if for any point $P \in F$, there is exactly one corresponding triangle $T_i \cap T$, and the value of F is known at all of vertices of T_i.</p>

Geo-objects

OMT-G has two geo-object descendant classes: *geo-object with geometry* and *geo-object with geometry and topology*.

A *geo-object with geometry* class represents objects that have only geometric properties (points, lines, and polygons), and is specialized precisely in classes named *Point*, *Line*, and *Polygon*. Examples include, respectively, bus stop, curb line, and municipal limits.

A *geo-object with geometry and topology* represents objects that have, in addition to geometric properties, topological connectivity properties, and are specifically suited to the representation of spatial network structures, such as water supply systems, electrical distribution systems, or road networks. These properties are present in objects that are either nodes or arcs in a graph-theoretic approach. Unidirectional lines indicate that the network has a definite flow direction, such as in sewage systems. Bidirectional lines indicate that there is a flow and a connection. The direction of the flow, in this case, is deemed irrelevant, since it can occur in any direction, as in water or electrical networks. The focus here is not on the implementation of the relationship, but rather on the semantics of the connection among network elements, which is a relevant element for spatial integrity assurance procedures. The implementation will depend on specific characteristics of the underlying GIS. This class specializes into subclasses *Node*, *Unidirectional Line*, and *Bidirectional Line* (Figure 3). Geo-objects with geometry and topology are not subject to a set of integrity constraints by themselves, but their use is conditioned to the existence of network relationships, which are specified in "Simple Association, Spatial Relations and Network Relations" (see Table 4 for the corresponding integrity constraints).

Table 4: Connectivity rules	
Arc-node structure	Let $G = \{N, A\}$ be a network structure, composed of a set of nodes $N = \{n_0, n_1, \dots, n_p\}$ and a set of arcs $A = \{a_0, a_1, \dots, a_q\}$. Members of N and members of A are related according to the following constraints:

	<ol style="list-style-type: none"> 1. For every node $n_i \in N$ there must be at least one arc $a_k \in A$. 2. For every arc $a_k \in A$ there must be exactly two nodes $n_i, n_j \in N$.
Arc-arc structure	<p>Let $G = \{A\}$ be a network structure, composed of a set of arcs $A = \{a_0, a_1, \dots, a_q\}$. Then the following constraint applies:</p> <ol style="list-style-type: none"> 1. Every arc $a_k \in A$ must be related to at least one other arc $a_i \in A$, where $k \neq i$.

The geometric concepts used in the definition of points, lines (including lines with a topological role), and polygons lead to some integrity constraints. These constraints should be intrinsically enforced by the GIS, but since this is not always the case, this matter will be discussed here.

In computational geometry, a polygonal line or a polygon are defined as simple whenever there are no crossings between non-adjacent segments, and complex in the opposite case. The formal conditions for a line to be considered simple correspond to the first two conditions in the isoline constraint ([Table 1](#)). As a matter of fact, GIS software usually does not forbid the creation of complex lines; however, this type of line seldom occurs in nature. Furthermore, such lines raise difficulties for topological analysis and in operations such as the creation of buffers. Actually, several GIS include data entry cleaning modules, which are capable of finding and eliminating such situations, by displacing vertices and/or dividing the lines into two or more simple parts.

The geometric definitions adopted in the OMT-G model admit the existence of geo-objects that are formed by several polygons, establishing one of them as the "basic" polygon and considering the others as *islands* or *holes*. These polygons which are composed of multiple parts (or polygonal regions (Laurini & Thompson, 1992)) are important, since there is no guarantee that the results of traditional operations, such as buffer creation, union, intersection, and difference between simple polygons, is always formed with simple polygons. In this case, an important requirement is that the basic polygon and the islands have their vertices stored in counterclockwise order, while the holes are stored in clockwise order (Margalit & Knott, 1989). Constraints regarding lines and polygons are presented in [Table 2](#).

Table 2: Geo-object constraints

Line	<p>1. Let v_0, v_1, \dots, v_n be $n+1$ points in the plane. Let s_i be n segments, connecting the points. These segments form a <i>polygonal line</i> L if, and only if, (1) the intersection of adjacent segments in L is only the extreme point shared by the segments (i.e., $s_i \cap s_{i+1} = v_{i+1}$), (2) non-adjacent segments do not intercept (i.e., $s_i \cap s_j = \emptyset$ for all i, j such that $j \neq i+1$), and (3) $v_0 \neq v_{n-1}$ that is, the polygonal line is not closed.</p>
Simple Polygon	<p>2. Let v_0, v_1, \dots, v_{n-1} be n points in the plane, with $n \geq 3$. Let s_i be a sequence of $n-1$ segments, connecting the points. These segments form a <i>simple polygon</i> P if, and only if, (1) the intersection of adjacent segments in P is only the extreme point shared by the segments (i.e., $s_i \cap s_{i+1} = v_{i+1}$), (2) non-adjacent segments do not intercept (that is, $s_i \cap s_j = \emptyset$ for all i, j such that $j \neq i+1$), and (3) $v_0 = v_{n-1}$. that is, the polygon is closed.</p>
Polygonal Region	<p>3. Let $R = \{P_0, P_1, \dots, P_{n-1}\}$ be a set formed by n simple polygons in the plane, with $n \geq 1$. Considering P_0 to be a basic polygon, R forms a <i>polygonal region</i> if, and only if, (1) $P_i \cap P_j = \emptyset$, for all $i \neq j$, (2) polygon P_0 has its vertices coded in a counterclockwise fashion, (3) P_i disjoint P_j (see Table 3) for all $P_i \cap P_0$ in which the vertices are coded counterclockwisely, and (4) P_0 contains P_i (see Table 3) for all $P_i \cap P_0$ in which the vertices are coded clockwise.</p>

Relationships

Considering the importance of spatial and non-spatial relations in the understanding of the modeled space, OMT-G represents the three types of relationship that can occur between its classes: *simple associations*, *spatial relations*, and *topological network relations*. The discrimination of such relations has the objective of defining explicitly the type of interaction that occurs between classes. There are some applications that do not

make use of spatial relations, but nevertheless there are applications on which spatial relations have a very relevant meaning, and therefore should be explicitly included in the application schema. Likewise, topological network relations are of fundamental importance for any applications that intend to employ geographic features in the management of spatially-distributed facilities or in the management of flows, such as those in the fields of transportation, energy, telecommunications, and sanitation.

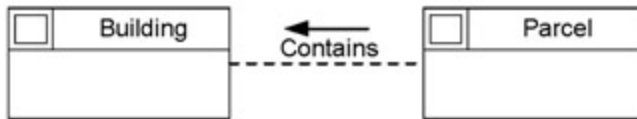
Simple Associations, Spatial Relations, and Network Relations

Simple associations represent structural relationships between objects of different classes, conventional as well as georeferenced. *Spatial relations* represent the topologic, metric, ordinal, and fuzzy relationships. Some relations can be derived automatically from the geometry of each object, during the execution of data entry or spatial analysis operations. Geometric relations, such as *contain* and *disjoint*, are an example of this. Others need to be specified by the user, in order to allow the system to store and maintain that information. The latter are called *explicit relations* (Peuquet, 1984).

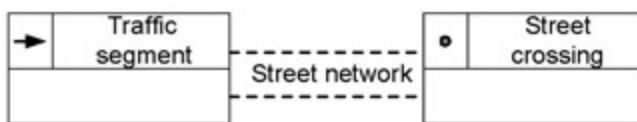
In OMT-G, simple associations are indicated by continuous lines, whereas spatial relations are indicated by dashed lines ([Figure 4](#)). Therefore, it is simple to distinguish between simple associations (alphanumeric relationships) and spatial relations.



(a) Simple association



(b) Spatial relationship



(c) Arc-node network relationship



(d) Arc-arc network relationship

Figure 4: Relationships

Based on previous works (Câmara, 1995; Clementini et al., 1993; Egenhofer & Franzosa, 1991; Egenhofer & Herring, 1990; Papadias & Theodoridis, 1997), OMT-G considers a set of nine different spatial relations between georeferenced classes. Clementini et al. (1993) identify a minimum set of spatial relation operators, comprising only five spatial relations, from which all others can be specified: *touch*, *in*, *cross*, *overlap*, and *disjoint*. However, we consider that sometimes a larger set is required, due to cultural or semantic concepts that are familiar to the users. These include relations such as *adjacent to*, *coincide*, *contain*, and *near*, which are in fact special cases of one of the five basic relations, but deserve special treatment because of their common use in practice. Spatial integrity constraints for these relations are listed in [Table 3](#), but additional constraints can be formulated in case some additional relation is required by the application. These include any kind of directional or relative spatial relations, such as *north of*, *left of*, *in front of*, or *above*.

Table 3: Spatial relationship integrity rules

Basic relations

Table 3: Spatial relationship integrity rules

Basic relations	
Touch	1. Let A, B be two geo-objects, where neither A nor B are members of the Point class. Then $(A \text{ touch } B) = \text{TRUE} ? (A^\circ \cap B^\circ = \emptyset) ? (A \cap B ? \emptyset)$.
In	2. Let A, B be two geo-objects. Then $(A \text{ in } B) = \text{TRUE} ? (A \cap B ? A) ? (A^\circ \cap B^\circ = \emptyset)$.
Cross	3. Let A be a geo-object of the Line class, and let B be a geo-object of either the Line or the Polygon class. Then $(A \text{ cross } B) = \text{TRUE} ?$ $\dim(A) = ((\max(\dim(A^\circ), \dim(B^\circ)) - 1)$ $? (A \cap B ? A) ? (A \cap B ? B)$
Overlap	4. Let A, B be two geo-objects, both members of the Line or of the Polygon class. Then $(A \text{ overlap } B) = \text{TRUE} ?$ $\dim(A^\circ) = \dim(B^\circ) = \dim(A^\circ \cap B^\circ)$ $? (A \cap B ? A) ? (A \cap B ? B)$.
Disjoint	5. Let A, B be two geo-objects. Then $(A \text{ disjoint } B) = \text{TRUE} ? A \cap B = \emptyset$
Special cases	
Adjacent to	6. Let A be a geo-object of the Polygon class and let B be a geo-object of either the Line or the Polygon class. Then $(A \text{ adjacent to } B) = \text{TRUE} ? (A \text{ touch } B) ? \dim(A \cap B) = 1$.
Coincide	7. Let A, B be two geo-objects. Then $(A \text{ coincide } B) = \text{TRUE} ? A \cap B = A = B$.
Contain	8. Let A, B be two geo-objects, where A is a member of the Polygon class. Then $(A \text{ contain } B) = \text{TRUE} ? ((B \text{ in } A) = \text{TRUE}) ? ((A \text{ coincide } B) = \text{FALSE})$
Near(dist)	9. Let A, B be two geo-objects. Let C be a buffer, created at a distance $dist$ around A . Then $(A \text{ near}(dist) B) = \text{TRUE} ? (B \text{ disjoint } C) = \text{FALSE}$

Some relationships are only allowed between specific classes, because they depend on the geometric representation. For instance, the existence of a *contain* relationship assumes that one of the classes involved is a polygon. In this aspect, traditional applications differ from geographic ones, where associations between conventional classes can be freely built, being independent from factors such as geometric behavior. The set of concepts the user has about each real world object strongly suggests a particular representation, because there is an interdependence between the representation, the type of interpretation, and the usage given to each object class. In OMT-G this is considered in order to allow the placement of relations involving georeferenced classes.

Considering the previously listed spatial relationship types, some spatial integrity rules can be established ([Table 3](#)). These rules are formulated using a notation commonly found in computational geometry, in which objects are indicated by upper-case italic letters (e.g. A, B), their boundaries are denoted as ∂A , and their interiors as A° (note that $A^\circ = A - \partial A$). The boundary of a point object is considered to be always empty (therefore the point is equivalent to its interior), and the boundary of a line is comprised of its two endpoints. A function, called *dim*, is used to return the dimension of an object, and returns 0 if the object is a point, 1 if it is a line, or 2 if it is a polygon.

The *disjoint* rule is very important to maintain the integrity of the data stored in the database, and it must be used in order to check input data. For instance, if the classes Street Segment and Building are disjoint, it means that there can never be a street segment overlapping a building. If it becomes necessary to draw a street segment over a building, the building must first be deleted. The street segment and building creation routines can enforce this rule.

The *near* rule is the only one described in [Table 3](#) that requires a parameter. Since the notion of proximity varies according to the situation, a precise distance must be supplied to allow for the correct evaluation of the relationship. As an example, consider the classes Address and Bus Stop. To establish the relationship between instances of these classes, the maximum distance at which the bus stop is still considered to be near some address must be defined, for instance 500 meters.

In OMT-G, *network relations* are relationships among objects that are connected with each other. As previously mentioned, a network relationship only shows the need for a logical connection, not a requirement for the implementation of a particular structure. Network relations are indicated by two parallel dashed lines, linking a node class to an arc class. Network structures can be built without nodes, requiring a recursive relationship on the class which represents graph segments. The name given to the network is annotated between the two dashed lines ([Figure 4c](#)). The connectivity rules, which apply to network relationship primitives, are listed in [Table 4](#).

As an example of the usage of these rules, consider a sewage network which is an arc-node logical structure. Nodes are used to represent network elements such as manholes, sewage treatment stations, and discharges, and arcs are used to represent piping segments. The system is required to ensure the connection between all types of nodes and segments. Network relations can be maintained by the GIS using special data structures, and are represented by connecting arcs and nodes. Connectivity rules are usually enforced by the GIS itself.

Cardinality

Relationships are characterized by their cardinality. The notation for cardinality adopted by OMT-G ([Figure 5](#)) is the same as that used by UML (Rational, 1997). Of course, the cardinality of the relationships constitutes a form of integrity constraint, usually called

structural constraints (Elmasri & Navathe, 2000), which exist regardless of the spatial characteristics of the data.

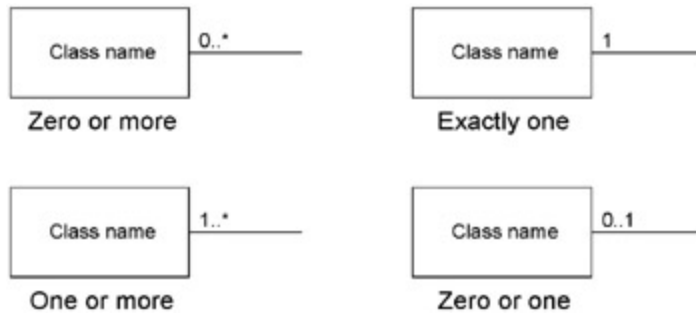
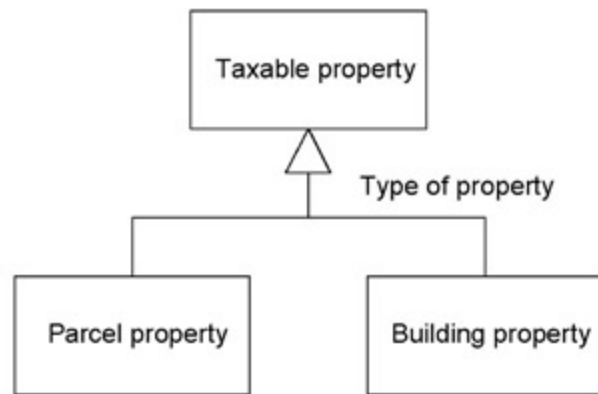


Figure 5: Cardinality

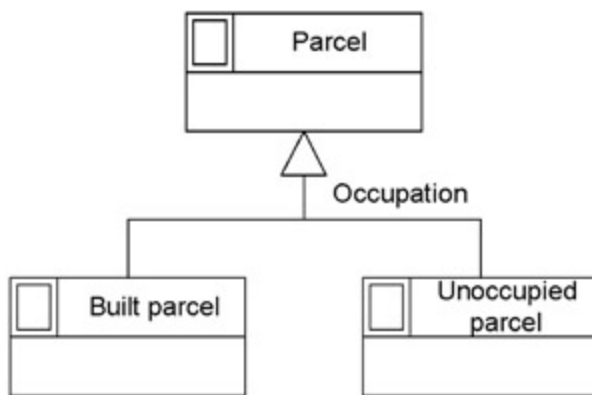
Generalization and Specialization

Generalization is the process of defining classes that are more general (superclasses) than classes with similar characteristics (subclasses) (Elmasri & Navathe, 2000; Laender & Flynn, 1994). *Specialization* is the inverse process in which more specific classes are detailed from generic ones, adding new properties in the form of attributes. Each subclass inherits attributes, operations, and associations from the superclass.

In the OMT-G model, the generalization and specialization abstractions apply to both georeferenced classes and conventional classes, following the definitions and notation proposed for UML, where a triangle connects a superclass to its subclasses ([Figure 6a, b](#)). Each generalization can have an associated *discriminator*, indicating which property is being abstracted by the generalization relationship.



(a) UML notation



(b) Spatial generalization

Figure 6: Generalization

Generalizations (spatial or not) can be specified as *total* or *partial* (Laender & Flynn, 1994; Rational, 1997). A generalization is total when the union of all instances of the subclasses is equivalent to the complete set of instances of the superclass. UML represents the totality constraint by using the predefined constraint elements *complete* and *incomplete*, but in OMT-G we have adopted the notation presented by Laender and Flynn (1994), in which a dot is placed in the upper vertex of the triangle that denotes generalization (Figure 7). Additionally, OMT-G also adopts the original OMT notation (Rumbaugh et al., 1991) for the UML predefined constraint elements *disjoint* and *overlapping*, that is, in a disjoint relation the triangle is left blank, and in a overlapping relation the triangle is filled. Therefore, the combination of the disjunction and totality aspects of generalization generates four types of constraints that apply to generalization/specialization. Figure 7 shows examples of each combination. Notice that completeness and disjointness are also specifications that force the implementation of corresponding integrity constraints, regardless of the spatial characteristics of the data.

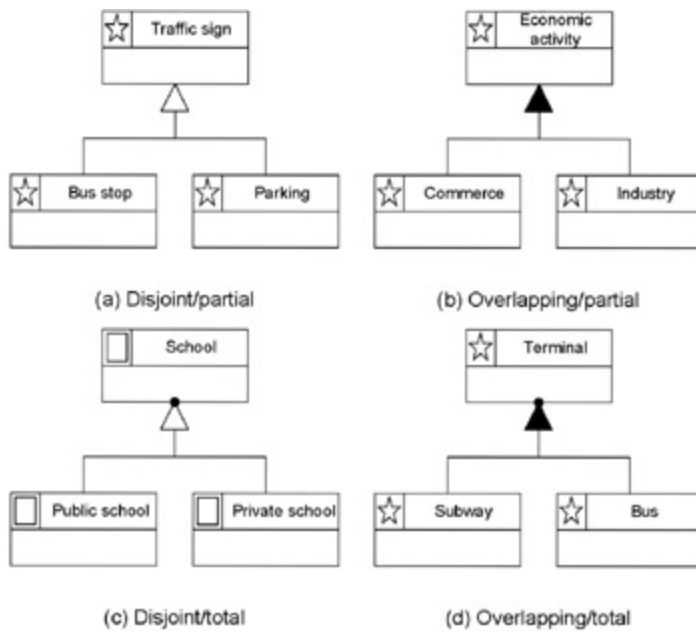


Figure 7: Spatial generalization examples

Aggregation

Aggregation is a special form of association between objects, where one is considered to be assembled from others. The graphic notation used in OMT-G follows the one used by UML (Figure 8). An aggregation can occur between conventional classes, between georeferenced classes, and between georeferenced and conventional classes (Figure 9). When the aggregation is between georeferenced classes, *spatial aggregation* must be used.

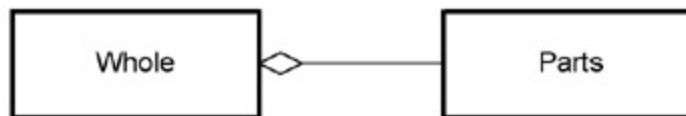


Figure 8: UML aggregation



Figure 9: Aggregation between conventional and georeferenced classes

Spatial aggregation is a special case of aggregation in which topological "whole-part" relationships are made explicit (Abrantes & Carapuça, 1994; Kösters et al., 1997). The usage of this kind of aggregation imposes spatial integrity constraints regarding the existence of the aggregated object and the corresponding sub-objects. Beyond providing more clarity and expressiveness to the model, the observation of these rules contributes to the maintenance of the semantic integrity of the geographic database. In spatial aggregation, also called topological "whole-part", the geometry of each part is entirely contained within the geometry of the whole. Also, no overlapping among the parts is allowed and the geometry of the whole is fully covered by the geometry of the parts. The

notation for this structure is presented in [Figure 10](#), where it is specified that blocks are composed of parcels, that is, blocks are geometrically equivalent to the union of adjacent parcels. This implies that (1) no area belonging to the block can exist outside of a parcel, (2) no overlapping can occur among parcels that belong to a block, and (3) no area belonging to a parcel can exist outside of a block. These three principles are stated in [Table 5](#) and correspond to the spatial integrity constraints associated with the spatial aggregation primitive.

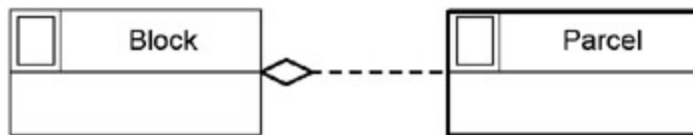


Figure 10: Spatial aggregation ("whole part")

Table 5: Spatial aggregation integrity rules	
Spatial aggregation	<p>Let $P = \{P_0, P_1, \dots, P_n\}$ be a set of geo-objects. Then P forms another object W by spatial aggregation if and only if</p> <ol style="list-style-type: none"> 1. $P_i \cap W = P_i$ for all i such that $0 \leq i \leq n$, and 2. $P_i \cap P_j = \emptyset$, and 3. $((P_i \text{ touch } P_j) \rightarrow (P_i \text{ disjoint } P_j)) = \text{TRUE}$ for all i, j such that $i \neq j$.

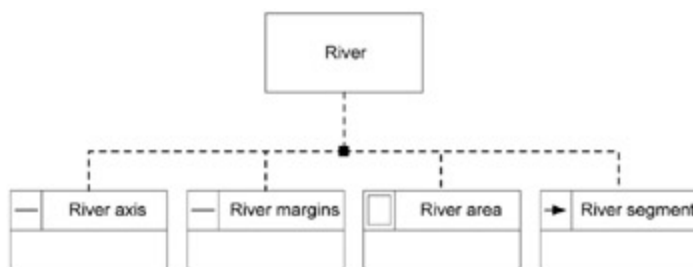
Notice that the class diagram does not specify whether the whole can be assembled from individual parts in an automatic fashion, nor does it specify whether the parts can be obtained automatically from the whole. If such automatic generation of instances can be specified, then it is done in the transformation diagram (Davis, 2000) by specifying exactly which transformation operation should be used. This transformation must ensure the application of the integrity constraints for spatial aggregation, as stated in [Table 5](#).

Conceptual Generalization

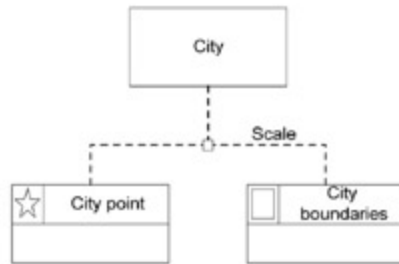
The spatial primitive *conceptual generalization* is used to record the need for different representations for a given object (Davis & Laender, 1999). In this type of relationship, the superclass does not need to have a specific representation. However, its subclasses are represented by distinct geometric shapes, being allowed to inherit the superclass' alphanumeric attributes and to include specific attributes of their own. The objective of the use of this primitive is to allow relationships involving each representation style to be made explicit. As previously shown, the way a class is represented influences the spatial

relationship types that can occur. The same representation alternative is allowed in more than one subclass, because in each one the level of detail or resolution can vary.

Conceptual generalization can occur in two representation variations: *according to geometric shape* and *according to scale*. The variation according to geometric shape is used to record the simultaneous existence of multiple scale-independent representations for a class. For instance, a river can be represented by its axis, as a single line, as the space between its margins, as a polygon covered by water, or as a set of flows (directed arcs) within river sections, forming a hydrographic network (Figure 11a). Variation according to scale is used in the representation of different geometric aspects of a given class, each corresponding to a range of scales. A city can be represented by its political borders (a polygon) in a larger scale, and by a symbol (a point) in a smaller scale (Figure 11b).



(a) Variation according to shape (overlapping)



(b) Variation according to scale (disjoint)

Figure 11: Conceptual generalization

The notation used for cartographic generalization uses a square to connect the superclass to its subclasses. The subclass is connected to the square by a dashed line. As a discriminator, the word *Scale* is used to mean variation according to scale, and the word *Shape* is used to determine variation according to geometric shape. The square is blank when subclasses are disjoint and filled if subclass overlapping is allowed (Figure 11). As in the case of generalization and specialization, the disjointness defines an integrity constraint, in which an instance of the superclass can only belong to one of the subclasses, and therefore multiple representations for a single superclass instance are not allowed.

The variation according to geometric shape can also be used in the representation of classes which simultaneously have georeferenced and conventional instances. For instance, a traffic sign can exist in the database as a non-georeferenced object, such as a

warehouse item, but it becomes georeferenced when installed at a particular location (Figure 12). Notice that the conceptual generalization in Figure 12 is also disjoint, and therefore a given traffic sign can either be in stock, or installed at a definite geographic position — it cannot be both at the same time.

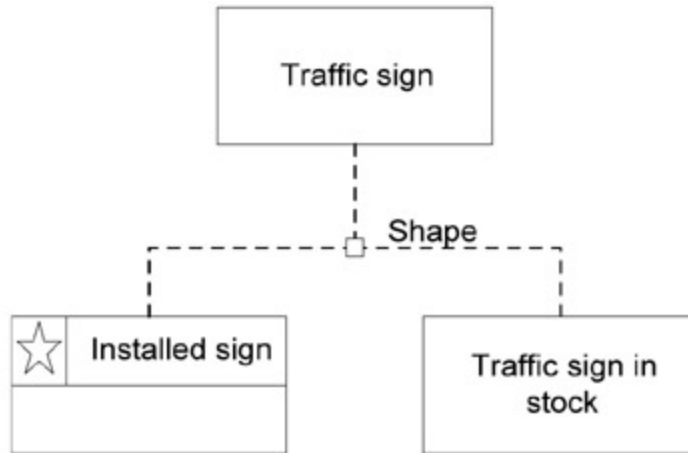


Figure 12: Conceptual generalization with a conventional class

Except for the inheritance of superclass characteristics by subclasses, the conceptual generalization primitive does not define any additional spatial integrity constraints. Notice, however, that some generalization operations (particularly cartographic generalization) can inadvertently cause modifications in spatial relationships. The application must ensure that when a more general (less detailed) class is generated from a more detailed one, the same topological relationships must hold (Egenhofer et al., 1994; Paiva, 1998).

APPLICATION EXAMPLE

To illustrate the spatial integrity constraints derived from the primitives and spatial relationships included in OMT-G, a sample model is presented in this section, corresponding to part of a family health application.

Figure 13 shows the class diagram for the example application. The geographic space for the application corresponds to a municipality. A set of digital orthophotos covers the entire municipal territory, to be used as background information for the application. The municipal space is subdivided into health districts, which are responsible for decentralized health services. Each district employs health agents, who care for families who live within the district's area. The districts contain blocks, which are in turn subdivided into parcels. Blocks and parcels are represented by their polygonal boundary. Parcels can be unoccupied or built, depending on whether one or more buildings have been erected on them. Building addresses are formed by concatenating the thoroughfare code to the street number. Each address is defined as a symbol, and is to be located inside the parcel's boundary. Only built parcels can have addresses (a user defined integrity constraint). A thoroughfare is represented by its segments, which take on the role of arcs

in a street network. The nodes are thoroughfare intersections, at the crossings. Each health agent is responsible for regularly visiting a number of families, applying preventive medicine actions, such as newborn follow-up, pregnancy control, vaccination, sanitary conditions inspection, and others. Family members are registered in the system, along with their individual data and medical history. The agents have routes to follow, going from home to home.

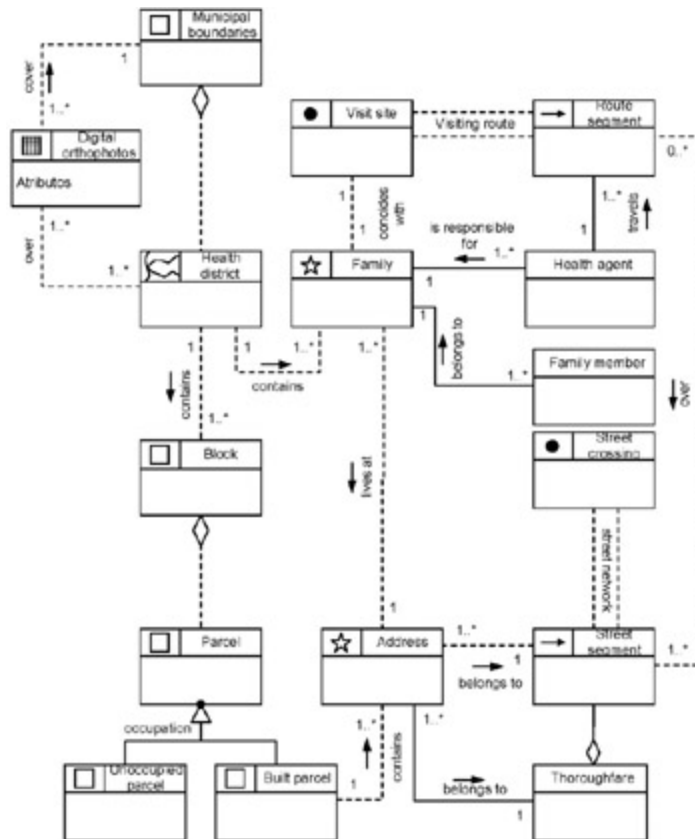


Figure 13: Application example

From the class diagram, following the definitions of the spatial constraints provided earlier, it can be observed that the integrity constraints listed in [Table 6](#) apply.

Table 6: Spatial integrity constraints from the example		
Class	Description	Constraint
Topologic	Municipal boundaries are a spatial aggregation of Health District	Spatial aggregation (Table 5)
	Block is a spatial aggregation of Parcel	
	Health district contains Family	Contain (Table 3)
	Health district contains Block	

Table 6: Spatial integrity constraints from the example

Class	Description	Constraint
	Built parcel contains Address	
	The street network is composed of Street segments (arcs) and Street crossings (nodes)	Connectivity (Table 4)
	Visiting routes are composed of Route segments (arcs) and Visit sites (nodes)	
	Address and Family coincide	Coincide (Table 3)
	Family and Visit site coincide	
	Health districts must fill the modeled space (municipal boundaries)	Planar enforcement (Table 1)
	Digital orthophotos must fill the modeled space (municipal boundaries)	
	Health districts form a planar subdivision	Planar subdivision (Table 1)
	Digital orthophotos are tessellations	Tessellation (Table 1)
	Street segment and Route segment are unidirectional lines	Line (Table 2)
	Block, Parcel, Unoccupied parcel, Built parcel, Municipal boundaries are polygons	Simple polygon (Table 2)
Semantic	Street segments cannot cross Blocks	—
	Addresses must be contained in Blocks	—
User defined	An Unoccupied parcel must not contain any Address	—

Besides the spatial integrity constraints listed in [Table 6](#), integrity constraints corresponding to the simple associations included in the diagram must be specified. There is also the need to specify the integrity constraint on the specialization relationship between Parcel, Unoccupied parcel and Built parcel. The cardinality of all simple associations and spatial relationships must also be specified as structural constraints.

FUTURE DEVELOPMENTS

Current GIS products are the descendants of a long line of verticalized software in which all the relevant functions were implemented by the GIS developers and incorporated, usually in a proprietary fashion, to the software. It is very common to find GIS implementations which incorporate primitive spatial database management functions,

while providing some sort of interface to standard relational database management products. Only recently, after the release of the Open GIS Consortium's Simple Features Specification, traditional Database Management Systems (DBMS) have begun to incorporate more adequate support for spatial data, using object-relational tables and spatial indexing. GIS developers are slowly realizing that the possibility of using a spatial DBMS underlying their products is a good alternative, and are therefore delivering interfaces to them.

However, current spatial DBMSs do not implement spatial integrity constraints in the same way that they support relational integrity constraints. Rather, they assume that all the spatial integrity checking will be made by the GIS, during the data entry process, and prior to the storage in the database. We feel that, if clear rules can be formulated, spatial integrity constraints can be translated from the conceptual schema to the implementation schema, and could therefore be incorporated as a part of the spatial database's design. A careful examination of the spatial integrity rules presented in this chapter shows that every one of them can be implemented with the help of well-known computational geometric algorithms, such as point-in-polygon (Preparata & Shamos, 1988), line intersection (Cormen et al., 1990; Preparata & Shamos, 1988), polygon intersection/union/difference (Margalit & Knott, 1989), or by locally building and using well-known topological structures, such as winged-edge (Baumgart, 1975). Also, the user must be allowed to formulate specific spatial integrity constraints, as required by the application. This can be done by either providing a verification function, or by using a combination of existing ones, algorithms that have been a part of commercial GIS for a long time, such as slivers and gaps detection, edge-matching, line simplification, and others.

While these constraint verifications can be incorporated to the GIS, one of the strongest arguments for installing a spatially-enabled DBMS in a corporate environment is to enable the use of a wide array of client products, each specializing in a specific aspect of the use of spatial data in the organization: database maintenance, spatial analysis, map production, integration with legacy systems, and so on. The only way to be sure that every modification of the spatial data results in an integral database is to implement the spatial integrity constraints as a function of the DBMS, adapting the client applications to reflect (and to benefit from) that functionality.

When the integration of spatial integrity constraints to spatially-enabled DBMSs is implemented, GIS developers and users will be allowed to invest on other aspects of the applications, such as multiple representations (Davis, 2000) and the use of ontologies to bring the application closer to the user's mental model (Fonseca, 2001).

REFERENCES

Abrantes, G. & Carapuça, R. (1994) *Explicit representation of data that depend on topological relationships and control over data consistency*. In *Proc. Fifth European Conference and Exhibition on Geographical Information Systems — EGIS/MARI'94*, 1:869–877. (<http://www.sgi.ursus.maine.edu/gisweb/egis/eg94100.html>).

- Baumgart, B. (1975) *A polyhedron representation for computer vision*. In *Proceedings of the AFIPS Conference*, 589–596, Anaheim, California.
- Borges, K. A. V., Laender, A. H. F. & Davis Jr., C. A. (1999) *Spatial data integrity constraints in object oriented geographic data modeling*. In *Proceedings of the 7th International Symposium on Advances in Geographic Information Systems (ACM GIS'99)*, 1–6, Kansas City, Missouri.
- Borges, K. A. V. (1997) *Geographic data modeling — an extension of the OMT model for geographic applications*. Master's thesis, João Pinheiro Foundation, Minas Gerais Government School. In Portuguese.
- Borges, K. A. V., Davis Jr., C. A. & Laender, A. H. F. (2001) *OMT-G: An Object-Oriented Data Model for Geographic Applications*. *GeoInformatica* 5(3):221–260.
- Câmara, G. (1995) *Models, languages, and architectures for geographic databases*. Ph.D. Thesis, inpe. In Portuguese.
- Clementini, E., DiFelice, P. & Oosterom, P. (1993) *A small set of formal topological relationships suitable for end-user interaction*. In *Proceedings of the 3rd Symposium on Spatial Database Systems*, 277–295, Singapore.
- Cockcroft, S. (1997) *A taxonomy of spatial data integrity constraints*. *GeoInformatica* 1(4): 327–343.
- Cormen, T. H., Leiserson, C. E. & Rivest, R. L. (1990) *Introduction to Algorithms*. McGraw-Hill and MIT Press, Cambridge, Massachusetts.
- Davis Jr., C. A. (2000) *Multiple representations in geographic information systems*. Ph.D. Thesis, Universidade Federal de Minas Gerais, Belo Horizonte. In Portuguese.
- Davis Jr., C. A. & Laender, A. H. F. (1999) *Multiple representations in GIS: materialization through geometric, map generalization, and spatial analysis operations*. In *Proceedings of the 7th International Symposium on Advances in Geographic Information Systems (ACM GIS'99)*, 60–65, Kansas City, Missouri.
- Egenhofer, M. J., Clementini, E. & Di Felice, P. (1994) *Evaluating Inconsistencies among Multiple Representations*. In *Proceedings of the Sixth International Symposium on Spatial Data Handling*, 2:901–920, Edinburgh.
- Egenhofer, M. J. & Franzosa, R. D. (1991) *Point-set topological spatial relations*. *International Journal of Geographical Information Systems*, 5(2):161–174.
- Egenhofer, M. J. & Herring, J. (1990) *A mathematical framework for the definition of topological relationships*. In *Proc. 4th International Symposium on Spatial Data Handling*, 803–813, Zurich.
- Elmasri, R. & Navathe, S. (2000) *Fundamentals of database systems*. 3rd Edition. Addison-Wesley, Reading, Massachusetts.
- Fonseca, F. T. (2001) *Ontology-driven Geographic Information Systems*. Ph.D. Thesis, University of Maine.
- Frank, A. U. & Goodchild, M. F. (1990) *Two perspectives on geographical data modeling*. National Center for Geographic Information and Analysis (NCGIA). Technical Report 90-11.
- Goodchild, M. F. (1992) *Geographical data modeling*. *Computers & Geosciences*, 18(4):401–408.
- Kösters, G., Pagel, B. & Six, H. (1997) *GIS-application development with GeoOOA*. *International Journal of Geographical Information Science*, 11(4):307–335.

- Laender, A. H. F. & Flynn, D. J. (1994) *A semantic comparison of modelling capabilities of the ER and NIAM models*. In: R. Elmasri, V. Kouramajian, and B. Thalheim (eds.) *Entity-Relationship approach — ER'93*, 242–256, Springer-Verlag.
- Laurini, R. (2001) *Information Systems for Urban Planning: a Hypermedia Co-operative Approach*. Taylor & Francis.
- Laurini, R., Thompson, D. (1992) *Fundamentals of spatial information systems*. Academic Press, London.
- Margalit, A., Knott, G. D. (1989) *An Algorithm for Computing the Union, Intersection or Difference of Two Polygons*. *Computers & Graphics* 13(2): 167–183.
- Oliveira, J. L., Pires, F. & Medeiros, C. M. B. (1997) *An environment for modeling and design of geographic applications*. *GeoInformatica* 1(1):29–58.
- Papadias, D. & Theodoridis, Y. (1997) *Spatial relations, minimum bounding rectangles, and spatial data structures*. *International Journal of Geographical Information Science*, 11(2):111–138.
- Paiva, J. A. C. (1998) *Topological consistency in geographic databases with multiple representations*. Ph.D. Thesis, University of Maine.
- Peuquet, D. J. (1984) *A conceptual framework and comparison of spatial data models*. *Cartographica*, 21:666–113.
- Plumber, L. & Groger, G. (1997) *Achieving integrity in geographic information systems: maps and nested maps*, *GeoInformatica* 1(4): 346–367.
- Preparata, F. P. & Shamos, M. I. (1988) *Computational Geometry: an Introduction*, Springer-Verlag, New York.
- Rational Software Corporation (1997) *The Unified Language: notation guide*, version 1.1 July 1997. (<http://www.rational.com>).
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen W. (1991) *Object-Oriented Modeling and Design*. Prentice-Hall.
- Worboys, M. F. (1994) *A unified model for spatial and temporal information*. *The Computer Journal*, 37(1):26–34.

Chapter VI: Consistent Queries Over Databases With Integrity Constraints^[1]

Sergio Greco, Ester Zumpano, Università della Calabria,

Italy

INTRODUCTION

Integrity constraints represent an important source of information about the real world. They are usually used to define constraints on data (functional dependencies, inclusion dependencies, etc.). Nowadays integrity constraints have a wide applicability in several contexts such as semantic query optimization, cooperative query answering, database integration and view update.

Often databases may be inconsistent with respect to integrity constraints, that is, one or more integrity constraints are not satisfied. This may happen, for instance, when the database is obtained from the integration of different information sources. The integration of knowledge from multiple sources is an important aspect in several areas such as data warehousing, database integration, automated reasoning systems and active reactive databases.

Since the satisfaction of integrity constraints cannot generally be guaranteed, in the evaluation of queries, we must compute answers which are consistent with the integrity constraints. Example 1 shows a case of inconsistency.

Example 1 Consider the following database schema consisting of the single binary relation *Teaches* (*Course*, *Professor*) where the attribute *Course* is a key for the relation. Assume there are two different instances for the relations *Teaches* as reported in [Figure 1](#).

Course	Professor
c1	p1
c2	p2

Course	Professor
c1	p1
c2	p3

Figure 1

The two instances satisfy the constraint that *Course* is a key but, from their union we derive a relation which does not satisfy the constraint since there are two distinct tuples with the same value for the attribute *Course*.

In the integration of two conflicting databases simple solutions could be based on the definition of preference criteria such as a partial order on the source information or a majority criteria (Lin and Mendelzon, 1996). However, these solutions are not generally satisfactory and more useful solutions are those based on 1) the computation of repairs for the database, 2) the computation of consistent answers (Arenas et al., 1999).

The computation of repairs is based on the definition of minimal sets of insertion and deletion operations so that the resulting database satisfies all constraints. The computation of consistent answers is based on the identification of tuples satisfying integrity constraints and on the selection of tuples matching the goal.

For instance, for the integrated database of *Example 1*, we have two alternative repairs consisting of the deletion of one of the tuples $(c2, p2)$ and $(c2, p3)$. The consistent answer to a query over the relation *Teaches* contains the unique tuple $(c1, p1)$ so that we don't know which professor teaches course *c2*.

Therefore, it is very important, in the presence of inconsistent data, not only to compute the set of consistent answers, but also to know which facts are unknown and if there are possible repairs for the database. In our approach it is possible to compute the tuples that are consistent with the integrity constraints and answer queries by considering as true facts—those contained in every repaired database, false facts—those that are not contained in any repaired database and unknown—the remaining facts.

Example 2 Consider the integrated relation of Example 1 containing the tuples $(c1, p1)$, $(c2, p2)$ and $(c2, p3)$. The database is inconsistent and there are two possible repairs which make it consistent: $R1 = (\emptyset, \{Teaches(c2, p2)\})$ and $R2 = (\emptyset, \{Teaches(c2, p3)\})$ which delete, respectively, the tuples $(c2, p2)$ and $(c2, p3)$, from the relation *Teaches*. The set of consistent tuples in the relation *Teaches* consists of the singleton $(c1, p1)$.

This chapter illustrates recent techniques for computing consistent answers and repairs for possibly inconsistent databases.

[1] Work partially supported by MURST grants under the projects "Data-X" and D2I. The first author is also supported by ISI-CNR.

ORGANIZATION OF THE CHAPTER

We first present some preliminaries on relational databases, disjunctive deductive databases and integrity constraints and then we introduce the formal definition of repair, consistent answer and the different techniques for querying and repairing inconsistent databases. In particular we present: 1) an extension of relational algebra, called flexible relational algebra, for inconsistent relations; 2) the integrated relational calculus which extends relations and algebra for querying inconsistent data; 3) a technique for merging relations based on majority criteria; 4) a technique for querying and repairing inconsistent data based on the concept of residual; 5) a technique for querying inconsistent databases based on the definition of a logic program for defining possible repairs and 6) a technique based on the rewriting of integrity constraints into disjunctive Datalog rules.

Relational Databases

We assume there are finite sets of relation names \mathbf{R} , attribute names \mathbf{A} and attribute values (also called *database domain*) \mathbf{V} . A relation schema of a relation $R \in \mathbf{R}$ is of the form (A_1, \dots, A_n) where $A_1, \dots, A_n \in \mathbf{A}$. A relational database schema is a set of relation schemas. Each attribute A has associated a domain denoted by $\text{DOM}(A)$. The null value \perp is not contained in $\text{DOM}(A)$ and $\text{DOM}_{\perp}(A) = \text{DOM}(A) \cup \{\perp\}$.

A tuple for a relation R is a mapping assigning to each attribute A of R an element in $\text{DOM}_{\perp}(A)$, i.e. a list of values (v_1, \dots, v_n) where v_i is the value of the attribute A_i , for each i in $[1..n]$. A relation (instance) is a set of tuples. In the following, a tuple (v_1, \dots, v_n) of a relation R , will also be denoted by $R(v_1, \dots, v_n)$.

The set of keys of a relation R will be denoted by $\text{keys}(R)$ and the primary key is denoted by $\text{pkey}(R)$. We assume that the value of the attributes in the primary key is not null.

Disjunctive Deductive Databases

A (disjunctive Datalog) rule r is a clause of the form

$$A_1 \vee \dots \vee A_k \leftarrow B_1, \dots, B_m, \text{not } B_{m+1}, \dots, B_n, \quad k+m+n > 0$$

where $A_1, \dots, A_k, B_1, \dots, B_n$ are atoms of the form $p(t_1, \dots, t_h)$, p is a predicate symbol of arity h and the terms t_1, \dots, t_h are constants or variables (Eiter et al., 1998). The disjunction $A_1 \vee \dots \vee A_k$ is the *head* of r , while the conjunction $B_1, \dots, B_m, \text{not } B_{m+1}, \dots, \text{not } B_n$ is the *body* of r . We also assume the existence of the binary built-in predicate symbols (comparison operators) which can only be used in the body of rules.

The *Herbrand Universe* U_P of a program P is the set of all constants appearing in P , and its *Herbrand Base* B_P is the set of all ground atoms constructed from the predicates appearing in P and the constants from U_P . A term, (resp. an atom, a literal, a rule or a program) is *ground* if no variables occur in it. A rule r' is a *ground instance* of a rule r , if r' is obtained from r by replacing every variable in r with some constant in U_P . We denote by $\text{ground}(P)$ the set of all ground instances of the rules in P .

An interpretation of P is any subset of B_P . The value of a ground atom L w.r.t. an interpretation I , $\text{value}_I(L)$, is true if $L \in I$ and *false* otherwise. The value of a ground negated literal $\text{not } L$ is $\text{not } \text{value}_I(L)$. The truth value of a conjunction of ground literals $C = L_1 \wedge \dots \wedge L_n$ is the minimum over the values of the L_i , i.e. $\text{value}_I(C) = \min(\{\text{value}_I(L_i) \mid 1 \leq i \leq n\})$, while the value $\text{value}_I(D)$ of a disjunction $D = L_1 \vee \dots \vee L_n$ is their maximum, i.e., $\text{value}_I(D) = \max(\{\text{value}_I(L_i) \mid 1 \leq i \leq n\})$; if $n=0$, then $\text{value}_I(C) = \text{true}$ and $\text{value}_I(D) = \text{false}$.

A ground rule r is *satisfied* by I if $\text{value}_I(\text{Head}(r)) = \text{value}_I(\text{Body}(r))$. Thus, a rule r with empty body is satisfied by I if $\text{value}_I(\text{Head}(r)) = \text{true}$. In the following we also assume the existence of rules with empty head which define denials (under total semantics), i.e., rules which are satisfied only if the body is false ($\text{value}_I(\text{Body}(r)) = \text{false}$). An interpretation M for P is a model of P if M satisfies each rule in $\text{ground}(P)$. The (model-theoretic) semantics for a positive program, say P , assigns to P the set of its *minimal models* $MM(P)$, where a model M for P is minimal, if no proper subset of M is a model for P (Minker, 1982). The more general *disjunctive stable model semantics* also applies to programs with (unstratified) negation (Gelfond and Lifschitz, 1991). For any interpretation I , denote with P^I the ground positive program derived from $\text{ground}(P)$ 1) by removing all rules that contain a negative literal $\text{not } a$ in the body and $a \in I$, and 2) by removing all negative literals from the remaining rules. An interpretation M is a (disjunctive) stable model of P if and only if $M \in MM(P^M)$.

For general P , the stable model semantics assigns to P the set $SM(P)$ of its *stable models*. It is well known that stable models are minimal models (i.e. $SM(P) \subseteq MM(P)$) and that for negation free programs minimal and stable model semantics coincide (i.e., $SM(P) = MM(P)$). Observe that stable models are minimal models which are "supported", i.e., their atoms can be derived from the program. An alternative semantics which overcomes some problems of stable model semantics has been recently proposed in Greco (1999).

Extended Disjunctive Databases

An extended atom is either an atom, say A or its negation $\neg A$. An extended Datalog program is a set of rules of the form

where $A_1, \dots, A_k, B_1, \dots, B_n$ are extended atoms.

A (2-valued) interpretation I for an extended program P is a pair $\langle T, F \rangle$ where T and F define a partition of B_P ? $\neg B_P$ and $\neg B_P = \{ \neg A \mid A \in B_P \}$. The semantics of an extended program P is defined by considering each negated predicate symbol, say, $\neg p$, as a new symbol syntactically different from p and by adding to the program, for each predicate symbol p with arity n the constraint ? $p(X_1, \dots, X_n), \neg p(X_1, \dots, X_n)$ (Gelfond and Lifschitz, 1991, Greco and Sacca, 1990; Kowalski and Sadri, 1991). The existence of a (2-valued) model for an extended program is not guaranteed, also in the case of negation (as-failure) free programs. In the following, for the sake of simplicity, we shall also use rules whose bodies may contain disjunctions. Such rules, called generalized disjunctive rules, are used as shorthand for multiple standard disjunctive rules.

Disjunctive Queries

Predicate symbols are partitioned into two distinct sets: *base predicates* (also called EDB predicates) and *derived predicates* (also called IDB predicates). Base predicates correspond to database relations defined over a given domain and they do not appear in the head of any rule whereas derived predicates are defined by means of rules.

Given a database D , a predicate symbol r and a program P , $D(r)$ denotes the set of r -tuples in D whereas P_D denotes the program derived from the union of P with the tuples in D , i.e. $P_D = P \cup \{ r(t) \mid t \in D(r) \}$. In the following a tuple t of a relation r will also be denoted as a fact $r(t)$. The semantics of P_D is given by the set of its stable models by considering either their union (*possible semantics* or *brave reasoning*) or their intersection (*certain semantics* or *cautious reasoning*). A *query* Q is a pair (g, P) where g is a predicate symbol, called the *query goal*, and P is a program. The answer to a query $Q = (g, P)$ over a database D , under the possible (resp. certain) semantics is given by $D'(g)$ where $D' = \bigcup_{M \models SM(P_D)} M$ (resp. $D' = \bigcap_{M \models SM(P_D)} M$).

INTEGRITY CONSTRAINTS

Integrity constraints express information that is not directly derivable from the database data. They are introduced to provide information on the relationships among data and to restrict the state a database can take, i.e., to prevent the insertion or deletion of data which could produce incorrect states. A database D has associated a schema $DS = (R_S, IC)$ which defines the intentional properties of D : R_S denotes the set of relation schemas whereas IC contains the set of integrity constraints.

Integrity constraints express semantic information over data, i.e., relationships that must hold among data in the theory. Generally, integrity constraints represent the interaction among data and define properties which are supposed to be explicitly satisfied by all instances over a given database schema. Therefore, they are mainly used to validate database transactions.

Definition 1

An integrity constraint (or embedded dependency) is a formula of the first order predicate calculus of the form:

$$(\forall x_1 \dots \forall x_n) [\Phi(x_1, \dots, x_n) \supset (\exists z_1 \dots \exists z_k) \Psi(y_1, \dots, y_m)]$$

where $\Phi(x_1, \dots, x_n)$ and $\Psi(y_1, \dots, y_m)$ are two conjunctions of literals such that x_1, \dots, x_n and y_1, \dots, y_m are the distinct variables appearing in Φ and Ψ respectively and $\{z_1, \dots, z_k\} = \{y_1, \dots, y_m\} - \{x_1, \dots, x_n\}$ is the set of variables existentially quantified.

In the definition above, conjunction Φ is called the *body* and conjunction Ψ the *head* of the integrity constraint. Moreover, an integrity constraint is said to be *positive* if no negated literals occur in it (classical definitions of integrity constraints only consider positive nondisjunctive constraints, called *embedded dependencies* (Kanellakis, 1991)).

Six common restrictions on embedded dependencies that give us six classes of dependencies have been defined in the literature (Kanellakis, 1991):

- The *full* (or *universal*) are those not containing existential quantified variables.
- The *unirelational* are those with one relation symbol only; dependencies with more than one relation symbols are called *multirelational*.
- The *single-head* are those with a single atom in the head; dependencies with more than one atom in the head are called *multi-head*.
- The *tuple-generating* are those without the equality symbol.
- The *equality-generating* are full, single-head, with an equality atom in the head.
- The *typed* are those whose variables are assigned to fixed positions of base atoms and every equality atom involves a pair of variables assigned to the same position of the same base atom; dependencies which are not typed will be called *untyped*.

Most of the dependencies developed in database theory are restricted cases of some of the above classes. For instance, functional dependencies are positive, full, single-head, unirelational, equality-generating constraints.

In the rest of this section we concentrate on *full* (or *universal*) disjunctive constraints, where Y is a possibly empty disjunction of literals and a literal can be either a base literal

or a conjunction of built-in literals (i.e., literals using as predicate symbols comparison operators).

Therefore, an integrity constraint is a formula of the form:

$$(\forall X) [B_1 \wedge \dots \wedge B_n \wedge \phi \supset A_1 \vee \dots \vee A_m \vee \psi_1 \vee \dots \vee \psi_k]$$

where $A_1, \dots, A_m, B_1, \dots, B_n$ are base positive literals, $\phi, \psi_1, \dots, \psi_k$ are built-in literals, X denotes the list of all variables appearing in B_1, \dots, B_n and it is supposed that variables appearing in $A_1, \dots, A_m, \phi, \psi_1, \dots, \psi_k$ also appear in B_1, \dots, B_n .

Often we shall write our constraints in a different format by moving literals from the head to the body and vice versa. For instance, the above constraint could be rewritten as

where $\phi' = \phi \wedge \text{not } \psi_1 \wedge \dots \wedge \text{not } \psi_k$ is a conjunction of built-in atoms or in the form of rule with empty head, called *denial*:

which is satisfied only if the body is false.

Example 3 The integrity constraint

called *inclusion dependency* states that the relation p must be contained in the union of the relations q and r . It could be rewritten as

We now introduce some basic notions including what we understand as a consistent database, a consistent set of integrity constraints, a database repair and a consistent answer.

Definition 2

Given a database schema $DS = (Rs, IC)$ we say that IC is consistent if there exists a database instance D over DS such that $D \models IC$. Moreover, we say that a database instance D over DS is consistent if $D \models IC$, i.e. if all integrity constraints in IC are satisfied by D , otherwise it is inconsistent.

Example 4 The set of integrity constraint

is not consistent since there is no instance of relation p satisfying both constraints.

Intuitively, a *repair* for a (possibly inconsistent) database D is a minimal consistent set of insert and delete operations which makes D consistent, whereas a consistent answer for a query consists of two sets containing, respectively, the maximal set of true and undefined atoms which match the query goal; atoms which are neither true nor undefined can be assumed to be false.

QUERYING AND REPAIRING RELATIONS

Databases contain, other than data, intentional knowledge expressed by means of integrity constraints. Database schemata contain the knowledge on the structure of data, i.e., they put constraints on the form the data must have.

The relationships among data are usually defined by constraints such as functional dependencies, inclusion dependencies, referential constraints, etc. Integrity constraints and relation schemata are introduced to prevent the insertion or deletion of data which could produce incorrect states. Generally, databases contain explicit representation of intentional knowledge.

Definition 3

Given a database schema $DS = \langle Rs, IC \rangle$ and a database D over DS a repair for D is a pair of sets of atoms (R^+, R^-) such that 1) $R^+ \cap R^- = \emptyset$, 2) $D \cup R^+ - R^- \models IC$ and 3) there is no pair $(S^+, S^-) \neq (R^+, R^-)$ such that $R^+ \subseteq S^+$, $R^- \subseteq S^-$ and $D \cup S^+ - S^- \models IC$. The database $D \cup R^+ - R^-$ will be called the repaired database.

Thus, repaired databases are consistent databases which are derived from the source database by means of a minimal (under total semantics) set of insertion and deletion of tuples. Given a repair R for D , R^+ denotes the set of tuples which will be added to the database whereas R^- denotes the set of tuples of D which will be canceled. In the following, for a given repair R and a database D , $R(D) = D \cup R^+ - R^-$ denotes the application of R to D .

Example 5 Assume we have a database $D = \{p(a), p(b), q(a), q(c)\}$ with the *inclusion dependency* $(\exists X) [p(X) \rightarrow q(X)]$. The database D is inconsistent since the constraint $p(X) \rightarrow q(X)$ is not satisfied. The repairs for D are $R1 = (\{q(b)\}, \emptyset)$ and $R2 = (\emptyset, \{p(b)\})$

producing, respectively, the repaired databases $R1(D)=\{p(a), p(b), q(a), q(c), q(b)\}$ and $R2(D) = \{ p(a), q(a), q(c) \}$.

A (relational) query over a database defines a function from the database to a relation. It can be expressed by means of alternative equivalent languages such as relational algebra, ‘safe’ relational calculus or ‘safe’ non recursive Datalog (Abiteboul et al., 1995, Ullman, 1988). In the following, we shall use Datalog. Thus, a query is a pair (g, P) where P is a safe non-recursive Datalog program and g is a predicate symbol specifying the output (derived) relation. Observe that relational queries define a restricted case of disjunctive queries. The reason for considering relational and disjunctive queries is that, as we shall show in the next section, relational queries over databases with constraints can be rewritten into extended disjunctive queries over databases without constraints.

Definition 4

The set of repairs for a database D with respect to IC will be denoted by $Repair(D, IC)$. A tuple t over DS is consistent with respect to D if t belongs to all repaired databases, i.e. $t \in \bigcap_{D' \in Repair(D, IC)} D'$

Definition 5

Given a database schema $DS = (Rs, IC)$ and a database D over DS , an atom A is true (resp. false) with respect to (D, IC) if A belongs to all repaired databases (resp. there is no repaired database containing A). The set of atoms which are neither true nor false are undefined.

Thus, true atoms appear in all repaired databases whereas undefined atoms appear in a proper subset of repaired databases. Given a database D and a set of integrity constraints IC , the application of IC to D , denoted by $IC(D)$, defines three distinct sets of atoms: the set of true atoms $IC(D)^+$, the set of undefined atoms $IC(D)^u$ and the set of false atoms $IC(D)^-$.

Definition 6

Given a database schema $DS = \langle Rs, IC \rangle$, a database D over DS , and a query $Q = (g, P)$, the consistent answer to the query Q on the database D , denoted as $Q(D, IC)$, gives three sets, denoted as $Q(D, IC)^+$, $Q(D, IC)^-$ and $Q(D, IC)^u$. These contain, respectively, the sets of g -tuples which are true (i.e. belonging to $Q(D')$ for all repaired databases D'), false (i.e. not belonging to $Q(D')$ for all repaired databases D') and undefined (i.e. the set of tuples which are neither true nor false).

TECHNIQUES FOR QUERYING AND REPAIRING DATABASES

Recently, there have been several proposals considering the integration of databases as well as the computation of queries over inconsistent databases (Agarwal, 1992; Agarwal et al., 1995; Arenas et al., 1999; Bry, 1997; Dung, 1996; Greco and Zumpano, 2000; Greco and Zumpano, 2000b; Greco et al., 2001; Lin, 1996; Lin, 1996b; Lin and Mendelzon, 1996; Lin and Mendelzon, 1999). Techniques for the integration of knowledge bases, expressed by means of first order formulas, have been proposed as well (Baral et al., 1991; Baral et al., 1991b; Subrahmanian, 1994; Grant and Subrahmanian, 1995). Most of the techniques for computing queries over inconsistent databases work for restricted cases and only recently have there been proposals to consider more general constraints. In this chapter we give an informal description of the main techniques proposed in the literature.

Flexible Algebra (Agarwal-Keller-Wiederhold-Saraswat)

The flexible algebra extends relational algebra through the introduction of *flexible relations*, i.e. non 1NF relations that contain sets of non-key attributes, to provide semantics for database operations in the presence of potentially inconsistent data (Agarwal et al., 1995).

A flexible relation is obtained by applying the flexify (\sim) operator to a relation R with schema (K, Z) , where K denotes the set of attributes in the primary key and Z is the set of remaining attributes. The schema of $\sim(R)$ is $(K, Z, Cons, Sel, Src)$, where *Cons* is the *consistent status* attribute, *Sel* is the *selection status* attribute and *Src* is the *source attribute*. Thus, a flexible relation is derived from a classical relation by extending its schema with the *ancilliary* attributes and assigning values for these attributes for each of the tuples. Obviously a classical relation is consistent by definition. Inconsistencies may arise if the integration of a set of consistent and autonomous databases is performed. In order to represent inconsistent data in a flexible relation the method introduces the notion of *ctuple*.

A *ctuple* is defined as a cluster of tuples having the same values for the key attributes. A flexible relation is a set of *ctuples*. Two tuples t_1 and t_2 in the same *ctuple* are conflicting if there is some non key attribute A such that $? \ t_1[A] \ ? \ t_2[A] \ ? \ ?$, where the interpretation given to the null value consists in *no information* (Zaniolo, 1984). A *ctuple* is consistent if it contains non conflicting pairs of tuples. Note that a *ctuple* containing exactly a tuple is consistent by definition.

Example 6 Consider the following three relations R_1 , R_2 and R_3 coming, respectively, from the sources s_1 , s_2 and s_3 (Figure 2).

R1			
Key	Z1	Z2	Z3
10	x	⊥	z
20	y	⊥	z

R2			
Key	Z1	Z2	Z3
10	x	y	z
20	y	⊥	⊥

R3			
Key	Z1	Z2	Z3
10	x	w	z

Figure 2

The integrated relation R consists of two *ctuples* (c_1 and c_2) (Figure 3) where the *ctuple* c_2 is consistent whereas the *ctuple* c_1 is not consistent.

	R			
	Key	Z1	Z2	Z3
c1	10	x	⊥	z
	10	x	y	z
	10	x	w	z
c2	20	y	⊥	z
	20	y	⊥	⊥

Figure 3

As previously stated, in addition to the original attributes, the flexible operator extends the schema of the flexible relation with three ancillary attributes: *Cons*, *Sel* and *Src*. These attributes are instantiated by the application of the flexify operator. Each tuple of a flexible relation has a value for each ancillary attribute and the managing of these attributes is performed by the system.

- The *Cons* attribute defines the consistency status of the ctuple; its domain is $\{true, false\}$ and all tuples in the same ctuple have the same value.
- The *Sel* attribute denotes the selection status of the ctuples. It contains information about possible restrictions on the selection of tuples in ctuples and its domain is $\{true, false, maybe\}$; all tuples in the same ctuple have the same value. For flexible relations derived from source relations through the application of the *flexify* operator, its value is true whereas for relations derived from other flexible relations its value can also be *false* or *maybe*.
- The *Src* attribute refers to the source relation from which a particular tuple has been derived. Thus if we define a primary key for each ctuple it would be (Key, Src) .

Example 7 The flexible relation derived from the relation of Example 6 is shown in [Figure 4](#).

	$\sim R$						
	Key	Z1	Z2	Z3	Cons	Sel	Src
c1	10	x	\perp	z	false	true	s1
	10	x	y	z	false	true	s2
	10	x	w	z	false	true	s3
c2	20	y	\perp	z	true	true	s1
	20	y	\perp	\perp	true	true	s2

Figure 4

- In the above relation $\sim R$ the value of the attribute *Sel* equal to *true* means that if the selection operator is applied to the tuples of the same ctuple, the resulting set is ‘correct’.

Take, for instance, the relation R with attributes (A,B,C) with key attribute A and three tuples $t1=(a1,b,10)$, $t2=(a1,c,10)$ and $t3=(a2,b,20)$ where $t1$ and $t2$ are conflicting (they belong to the same ctuple with key "a1"). The selection $s_{B='b'}(R)$ gives a (consistent) relation consisting of the tuples $t1$ and $t3$. Moreover this result is not correct since the tuple $t1$ is conflicting with $t2$ in the source relation whereas in the resulting relation it is not conflicting with any tuple. This means that the attribute *Sel* for the ctuple with key value $a1$ must be false (these tuples cannot be selected).

Flexible Relational Algebra

The Flexible Algebra defines a set of operation on the Flexible Relations. These operations are defined in order to perform meaningful operation in the presence of conflicting data. The full algebra for flexible relation is defined in (Agarwal et al., 1992). In this section we briefly describe some of the operation in this algebra. The set of ctuple operation includes *merging*, *equivalence*, *selection*, *union*, *cartesian product* and *projection*.

The merge operator merges the tuples in a ctuple in order to obtain a single nested tuple referred to as *merged ctuple*. An attribute, say A , of the *merged ctuple* will be *null* if and only if this is the unique value the attribute A assumes in the ctuple.

Example 8 The merged relation derived from the relation of Example 7 is shown in [Figure 5](#).

	Key	Z1	Z2	Z3	Cons	Sel	Src
c1	10	x	{y,w}	z	false	true	{s1,s2,s3}
c2	20	y	\perp	z	true	true	{s1,s2}

Figure 5

Two merged ctuples $X(c1)$ and $X(c2)$ associated with the schema $(K,Z,Cons,Sel,Src)$ are equivalent ($X(c1) ? X(c2)$) if they do not conflict in any attribute but the *Src* attribute. More formally, $X(c1) ? X(c2)$ if $X(c1)[K] = X(c2)[K]$ and for each A in Z is: i)

$X(c1)[Cons] = X(c2)[Cons]$; ii) $X(c1)[Sel] = X(c2)[Sel]$; and iii) either $X(c1)[A] = X(c2)[A]$ or $? \in \{X(c1)[A], X(c2)[A]\}$.

Two ctuples $c1$ and $c2$ are considered equivalent if the corresponding merged ctuples are equivalent.

Selection Operator

The *Sel* attribute will be modified after the application of selection operations. In particular, for a given tuple c , and a given selection condition q , the attribute *Sel* will be: i) true if q is satisfied by all tuples in c ; ii) *false* if there is no tuple in c satisfying q and iii) *maybe* otherwise.

In classical relational algebra the select operator determines the selection status of a tuple for a given selection condition, thus the selection status over a tuple can be either *true* or *false*. In order to apply the selection predicate to a tuple c , the selection predicate is applied to a nested tuple $X(c)$. The semantics of the selection operator, in the flexible relational algebra, has to be extended to operate over non-1NF tuples; in fact, the attributes of a tuple may be associated with more than one value due to data conflicts.

Given a flexible relational schema $(K, Z, Cons, Sel, Src)$, a *simple partial predicate* is of the form $(A \text{ op } ?)$ or $(A \text{ op } B)$, where $A, B \in K \cup Z$, $op \in \{=, \neq, >, <, \leq, \geq\}$ and $?$ is a single value, i.e. $? \in \text{DOM}(A) \cup \{null\}$.

The predicate $(A \text{ op } B)$ evaluates to true, false or maybe as follows:

- true, if $\forall a_i \in A, \forall b_j \in B \mid (a_i \text{ op } b_j)$ is true.
- false, if $\exists a_i \in A, \exists b_j \in B \mid (a_i \text{ op } b_j)$ is false.
- maybe, otherwise.

The predicate $(A \text{ op } ?)$ is equivalent to $(A \text{ op } \{?\})$.

Obviously, since the semantics given to null is that of no information, any comparisons with null values evaluates to false. Hence predicate $(a \text{ op } ?)$ evaluates to false if a or $?$ is null, and predicate $(A1 \text{ op } A2)$ evaluates to false if $A1$ or $A2$ is null.

Union Operator

The union operator combines the tuples of two source ctuples in order to obtain a new tuple. Note that this operation is meaningful if and only if the two ctuples represent data of the same concept, and so their schema coincide and the value of the selection attribute

is *true*. The union operation has to be applied before any selection operation, because the selection operation can lead to a loss of information.

A union of two ctuples $c1$ and $c2$ associated with schema $(K, Z, Cons, Sel, Src)$ where $c1[K] = c2[K]$, denoted by $c = c1 \cup c2$, is such that for each tuple $t \in c$ either $t \in c1$ or $t \in c2$.

Integrated Relational Calculus (Dung)

An extension of flexible algebra for other key functional dependencies, called *Integrated Relational Calculus*, was proposed by Dung (1996). The integrated relational calculus is based on the definition of *maximal consistent subsets* for a possible inconsistent database. Dung proposed extending relations by also considering null values denoting the absence of information with the restriction that tuples cannot have null values for the key attributes. The integrated relational calculus overcomes some drawbacks of the flexible relational algebra:

- flexible relational algebra is not able to integrate possibly inconsistent relations if the associated relation schema has more than one key;
- flexible relational model provides a rather weak query language. The following two examples show two cases where the flexible algebra fails.

Example 9 Consider the database containing the single binary relation R whose schema is $(employee, wife)$ with two keys $\{ employee \}$ (primary key) and $\{ wife \}$ (secondary key). Assume there are two different instances for R , $R1 = \{(Terry, Lisa)\}$ and $R2 = \{(Peter, Lisa)\}$. Integrating $R1$ and $R2$ using the flexible model we obtain the relation $D = \{(Terry, Lisa), (Peter, Lisa)\}$. Now asking "Whose wife is Lisa?" the flexible algebra will return the incorrect answer $\{Terry, Peter\}$. In this example it is evident that flexible algebra fails in detecting the inconsistency in the data in $R1$ and $R2$, due to the fact that *wife* is a key. A correct answer would have been that it is undetermined who is the husband of Lisa.

Example 10 Consider the database schema consisting of the single binary relation R with two attributes $\{ employee, department \}$ and $\{ employee \}$ being the primary key. Assume there are two different instances of R , $R1 = \{(Terry, CS)\}$ and $R2 = \{(Terry, Math)\}$. By integrating $R1$ and $R2$ using the flexible model we obtain the relation $D = \{(Terry, \{CS, Math\})\}$. Now asking the question "who is employed in CS or Math?" the expected answer is $\{Terry\}$, but flexible model will give \emptyset , that is, it does not know who is working in *CS* or *Math*. Thus the flexible relational algebra is not able to express the selection formula $(department = CS \vee department = Math)$; moreover there is not even a way to ask a query like "who is possibly employed in Math?"

The model proposed by Dung generalizes the model of flexible relational algebra. He argues that the semantics of integrating possibly inconsistent data is naturally captured by the maximal consistent subsets of the set of all information contained in the collection data.

The assumption in the Integrated Relational Calculus is that the values of the attributes in the primary key are correct.

Let R be a relation with schema (K, Z) , where K is the set of attributes in the primary key and Z the set of remaining attributes. Given two tuples t and t' over R we say

- t and t' are *related* if $t[K] = t'[K]$, i.e. they agree on the key attributes;
- t and t' are *conflicting* if there exists a key K' of R such that i) for each $B \in K'$, $t[B] \neq t'[B]$ and ii) there is an attribute $A \in K \cap Z$ such that $t[A] \neq t'[A]$.
- t is *less informative* than t' , denoted by $t \preceq t'$ if and only if for each attribute $A \in K \cap Z$ is $t[A] = t'[A]$ or $t[A] = \text{null}$ and $t'[A] \neq \text{null}$.

A set of tuples T over R is said to be *joinable* if there exists a tuple t' such that for each $t \in T$, t is less informative than t' .

The notion of less informative can be extended to relations. Given two relation instances R_1 and R_2 over R , we say that R_2 is less informative than R_1 (and write $R_2 \preceq R_1$) if for each tuple $t_2 \in R_2$ there exists a related tuple $t_1 \in R_1$ ($t_1[K] = t_2[K]$) which is more informative than t_2 ($t_2 \preceq t_1$).

The Integrated Relational Model

The *Integrated Relational Model* integrates data contained in autonomous information sources by a collecting step consisting in the union of the relations.

Let R_1, R_2 be two relations over a relation R with schema (K, Z) . If the information collected from R_1 and R_2 , represented by $R = R_1 \cup R_2$, is consistent, R represents the integration of information in R_1 and R_2 . Moreover, if $R = R_1 \cup R_2$ is inconsistent, a maximal consistent subset of the information contained in R would be one possible admissible collection of information a user could extract from the integration.

Given a relation instance R , and two tuples t_1, t_2 in R with $t_1[K] = t_2[K]$, the extension of t_1 w.r.t. t_2 , denoted by $\text{ext}(t_1, t_2)$ is the tuple derived from t_1 by replacing every null value $t_1[a]$ with $t_2[a]$.

The extension of a relation R , denoted $Ext(R)$, is the relation derived from R by first adding to R all possible extensions of tuples in R made with other tuples of R and next deleting tuples which are subsumed by other tuples. More formally,

$$Ext(R) = R' - \{ t \text{ in } R' \mid \exists t1 \in R' \text{ s.t. } t \supseteq t1 \text{ and } t \neq t1 \}$$

where:

$$R' = R \cup \{ ext(t1, t2) \mid \exists t1, t2 \in R \}$$

Example 11 Consider the inconsistent relation R below. The relation R' is obtained from R by adding a tuple obtained by extending the tuple containing a null value. The relation $Ext(R)$ is obtained from R' by deleting the tuple with a null value ([Figure 6](#)).

R		
emp	tel	salary
Terry	5709	35
Terry	⊥	20

R'		
emp	tel	salary
Terry	5709	35
Terry	⊥	20
Terry	5709	20

Ext(R)		
emp	tel	salary
Terry	5709	35
Terry	5709	20

Figure 6

Let R_1, \dots, R_n be n relation instances over the same relational schema (K, Z) .

- A possible integration of R_1, \dots, R_n is defined as the relational representation of a maximal consistent subset of $Ext(R_1 \bowtie \dots \bowtie R_n)$.
- The collection of all possible integrations of R_1, \dots, R_n is defined as the semantics of integrating R_1, \dots, R_n denoted by $Integ(R_1, \dots, R_n)$ (i.e. the set of maximal subsets of $Ext(R_1 \bowtie \dots \bowtie R_n)$).

Example 12 The maximal consistent subsets of relation $Ext(R)$ in the above examples are shown in [Figure 7](#).

Ext(R)		
emp	tel	salary
Terry	5709	35

Ext(R)		
emp	tel	salary
Terry	5709	20

Figure 7

Querying Integrated Relations

Queries over integrated data are formulated by means of a language derived by relational calculus, called *integrated relational calculus*, through the insertion of quantifiers which refer to the possible integrations.

Example 13 Consider the inconsistent relation $D = \{(Frank, Ann), (Carl, Ann)\}$ over the schema $(employee, wife)$ with the two alternative keys $\{employee\}$ and $\{wife\}$. $Integ(D)$ consists of two possible integrations: $\{(Frank, Ann)\}$ and $\{(Carl, Ann)\}$. The query "Whose wife is Ann ?" can be formulated in the Integrated Relational Calculus by

- $Q1 = ? \text{ employee}.R(\text{employee}, \text{wife}) ? \text{ wife} = Ann$ which can be stated as "Whose wife is Ann in a possible scenario?".
- $Q2 = K (? \text{ employee}.R(\text{employee}, \text{wife}) ? \text{ wife} = Ann)$ which can be stated as "Whose wife is Ann in every scenario?" (here the modal quantifier K refers to all possible integrations).

In the first case the answer to the query $Q1$ is given by taking the union of the tuples matching the goal in all possible scenarios (brave reasoning), that is $Ans(Q1) = \{Frank, Carl\}$. The answer to the Query $Q2$ is obtained by considering the intersection of the tuples matching the goal in each possible scenario (cautious reasoning), thus $Ans(Q2) = \emptyset$.

Knowledge Base Merging by Majority (Lin-Mendelzon)

In the integration of different databases, an alternative approach, taking the disjunction of the maximal consistent subsets of the union of the databases, has been proposed in Baral et al. (1991). A refinement of this technique has been presented in Lin and Mendelzon (1996), which proposed taking into account the majority view of the knowledge bases in order to obtain a new relation which is consistent with the integrity constraint. The technique proposes a formal semantics to merge first-order theories under a set of constraints.

Semantics of Theory Merging

The basic idea is that given a set of theories to merge T_1, \dots, T_n and a set of constraints IC the models of the resulting theory, $Merge(\{T_1, \dots, T_n\}, IC)$, have to be those worlds 'closest' to the original theories, that is the worlds that have a minimal distance from $\{T_1, \dots, T_n\}$. The distance between two worlds w and w' , denoted by $dist(w, w')$ is the cardinality of the symmetric difference of w and w' , that is

$$dist(w, w') = |w \Delta w'| = (w - w') \cup (w' - w).$$

Then the distance between a possible world w and $\{T_1, \dots, T_n\}$ is:

$\text{Merge}(\{T_1, \dots, T_n\}, IC) = \{w \mid w \text{ is a model of } IC \text{ and } \text{dist}(w, \{T_1, \dots, T_n\}) \text{ is minimum}\}$

Example 14 Consider the three relation instances which collect information regarding author, title and year of publication of papers ([Figure 8](#)).

Bib1		
Author	Title	Year
John	T1	1980
Mary	T2	1990

Bib2		
Author	Title	Year
John	T1	1981
Mary	T2	1990

Bib3		
Author	Title	Year
John	T1	1980
Frank	T3	1990

Figure 8

From the integration of the three databases Bib1, Bib2 and Bib3 we obtain the database Bib ([Figure 9](#)).

Bib		
Author	Title	Year
John	T1	1980
Mary	T2	1990
Frank	T3	1980

Figure 9

The value of $\text{Merge}(\text{Bib}, \{\text{Bib1}, \text{Bib2}, \text{Bib3}\})$ is equal to $\text{Merge}(\text{Bib}, \text{Bib1}) + \text{Merge}(\text{Bib}, \text{Bib2}) = \text{Merge}(\text{Bib}, \text{Bib3}) = 1 + 3 + 1 = 5$ which is the minimum distance (among the relations satisfying IC) from the relations *Bib1, Bib2, Bib3*.

Thus, the technique proposed by Lin and Mendelson, removes the conflict about the year of publication of the paper T1 written by the author John observing that two of the three source databases, that have to be integrated, store the value 1980; thus the information that is maintained is the one which is present in the majority of the knowledge bases.

However, the ‘merging by majority’ technique does not resolve conflicts in all cases since information is not always present in the majority of the databases and, therefore, it is not always possible to choose between alternative values. In this case the integrated database contains disjunctive information. This is obtained by considering generalized tuples, i.e. tuples where each attribute value can be either a simple value or a set.

Example 15 Suppose now that in relation R3 the first tuple (*John, T1, 1980*) is replaced by the tuple (*John, T1, 1982*). The merged database contains now disjunctive information since it is not possible to decide the year of the book written by John ([Figure 10](#)).

Author	Title	Year
John	T1	{1980,1981,1982}
Mary	T2	1990
Frank	T3	1980

Figure 10

Here the first tuple states that the year of publication of the book written by John with title T1 can be one of the values belonging to the set {1980, 1981, 1982}.

In the absence of integrity constraints the merge operation reduces to the union of the databases, i.e., $Merge(\{T_1, \dots, T_n\}, \{\}) = T_1 \cup \dots \cup T_n$, whereas if IC is a set of functional dependencies $Merge(\{T_1, \dots, T_n\}, IC) = T_1 \cup \dots \cup T_n \cup IC$.

Computing Repairs (Arenas-Bertossi-Chomicki)

An interesting technique has recently been proposed in (Arenas et al.,1999). The technique introduces a logical characterization of the notion of consistent answer in a possibly inconsistent database. Queries are assumed to be given in prefix disjunctive normal form.

A query $Q(X)$ is a prenex disjunctive first order formula of the form:

where K is a sequence of quantifiers, ϕ_i contains only built-in predicates and X denotes the list of variables in the formula.

Given a query $Q(X)$ and a set of integrity constraints IC a tuple t is a *consistent answer* to the query $Q(X)$ over a database instance D , written $(Q,D) \models_{IC} t$, if t is a substitution for the variables in X such that for each repair D' of D , $(Q,D') \models t$.

Example 16 Consider the relation *Student* with schema $(Code, Name, Faculty)$ with the attribute *Code* as key. The functional dependencies $Code \twoheadrightarrow Name$ and $Code \twoheadrightarrow Address$ can be expressed by the following two constraints:

Assume there is an inconsistent instance of *Student* as reported in [Figure 11](#).

The inconsistent database has two repairs Repair1 and Repair2 ([Figure 12](#)).

Student		
Code	Name	Faculty
s1	Mary	Engineering
s2	John	Science
s2	Frank	Engineering

Figure 11

Repair1		
Code	Name	Faculty
s1	Mary	Engineering
s2	John	Science

Repair2		
Code	Name	Faculty
s1	Mary	Engineering
s2	Frank	Engineering

Figure 12

The consistent answers to the query $\exists z \text{ Student}(s1, y, z)$ is "Engineering", while there is no consistent answer to the query $\exists z (\text{Student}(s2, y, z))$.

General Approach

The technique is based on the computation of an equivalent query $T_{\exists}(Q)$ derived from the source query Q . The definition of $T_{\exists}(Q)$ is based on the notion of residue developed in the context of semantic query optimization.

More specifically, for each literal B appearing in some integrity constraint, a residue $\text{Res}(B)$ is computed. Intuitively, $\text{Res}(B)$ is a universal quantified first order formula that must be true, because of the constraints, if B is true. Universal constraints can be rewritten as denials, i.e. logic rules with empty heads of the form $\exists B_1 \exists \dots \exists B_n$.

Let A be a literal, r a denial of the form $\exists B_1 \exists \dots \exists B_n, B_i$ (for some $1 \leq i \leq n$) a literal unifying with A and θ the most general unifier for A and B_i such that variables in A are used to substitute variables in B_i but they are not substituted by other variables. Then, the residue of A with respect to r and B_i is

$$\begin{aligned} \text{Res}(A, r, B_i) &= \text{not}((B_1 \wedge \dots \wedge B_{i-1} \wedge B_{i+1} \wedge \dots \wedge B_n) \theta) \\ &= \text{not } B_1 \theta \vee \dots \vee \text{not } B_{i-1} \theta \vee \text{not } B_{i+1} \theta \vee \dots \vee \text{not } B_n \theta. \end{aligned}$$

The residue of A with respect to r is $\text{Res}(A, r) = \exists B_i \mid A = B_i \exists \text{Res}(A, r, B_i)$ consisting of the conjunction of all the possible residues of A in r whereas the residue of A with respect to a set of integrity constraints IC is $\text{Res}(A) = \exists r \mid IC \text{Res}(A, r)$.

Thus, the residue of a literal A is a first order formula which must be true if A is true.

The operator $T_{\exists}(Q)$ is defined as follows:

- $T_0(Q) = Q$;
- $T_i(Q) = T_{i-1}(Q) \text{ ? } R$ where R is a residue of some literal in T_{i-1} .

The operator $T_?$ represents the fixpoint of T .

It has been shown that the operator T has a fixpoint for universal quantified queries and universal binary integrity constraints, i.e. constraints, which when written in disjunctive format, are of the form: $\text{? } X (B_1 \text{ ? } B_2 \text{ ? } \text{?})$ where B_1, B_2 are literals and ? is a conjunctive formula with built-in operators. Moreover, it has also been shown that the technique is complete for universal binary integrity constraints and universal quantified queries.

Example 17 Consider a database D consisting of the two relations ([Figure 13](#)) with the integrity constraint, defined by the following first order formula

Supplier	Department	Item
c1	d1	i1
c2	d2	i2

Item	Type
i1	t
i2	t

Figure 13

stating that only supplier $c1$ can supply items of type t .

The database $D = \{ \text{Supply}(c1, d1, i1), \text{Supply}(c2, d2, i2), \text{Class}(i1, t), \text{Class}(i2, t) \}$ is inconsistent because the integrity constraint is not satisfied (an item of type t is also supplied by supplier $c2$).

This constraint can be rewritten as $\text{? } \text{Supply}(X,Y,Z) \text{ ? } \text{Class}(Z,t) \text{ ? } X \text{ ? } c1$, where all variables are (implicitly) universally quantified. The residue of the literals appearing in the constraint are

The iteration of the operator T to the query goal $\text{Class}(Z,t)$ gives

- $T_0(\text{Class}(Z,t)) = \text{Class}(Z,t)$,
- $T_1(\text{Class}(Z,t)) = \text{Class}(Z,t) \text{ ? } (\text{not Supply}(X,Y,Z) \text{ ? } X = c1)$,
- $T_2(\text{Class}(Z,t)) = \text{Class}(Z,t) \text{ ? } (\text{not Supply}(X,Y,Z) \text{ ? } X = c1)$.

At Step 2 a fixpoint is reached since the literal $Class(Z,t)$ has been ‘expanded’ and the literal $not Supply(X,Y,Z)$ does not have a residue associated to it. Thus, to answer the query $Q = Class(Z,t)$ with the above integrity constraint, the query $T_?(Q) = Class(Z,t) ? (not Supply(X,Y,Z) ? X = cI)$ is evaluated. The computation of $T_?(Q)$ over the above database gives the result $Z=iI$.

The following example shows a case where the technique proposed is not complete.

Example 18 Consider the integrity constraint $(X,Y,Z) [p(X,Y) ? p(X,Z) ? Y=Z]$, the database $D = \{ p(a,b), p(a,c) \}$ and the query $Q = ? U p(a,U)$ (we are using the formalism used in (Arenas et al., 1999)). The technique proposed generates the new query $T_?(Q) = ? U [p(a,U) ? Z(\neg p(a,Z) ? Z=U)]$ which is not satisfied contradicting the expected answer which is true.

This technique is complete for universal binary integrity constraints and universal quantified queries. Moreover the detection of fixpoint conditions is, generally, not easy.

Querying Database using Logic Programs with Exceptions (Arenas-Bertossi-Chomicki)

The new approach proposed by Arenas-Bertossi-Chomicki in Arenas et al. (2000) consists in the use of a Logic Program with Exceptions (LPe) for obtaining consistent query answers. An LPe is a program with the syntax of an extended logic program (ELP), that is, in it we may find both logical (or strong) negation (\neg) and procedural negation (not). In this program, rules with a positive literal in the head represent a sort of general default, whereas rules with a logically negated head represent exceptions. The semantic of an LPe is obtained from the semantics for ELPs, by adding extra conditions that assign higher priority to exceptions. The method, given a set of integrity constraints ICs and an inconsistent database instance, consists in the direct specification of database repairs in a logic programming formalism. The resulting program will have both negative and positive exceptions, strong and procedural negations, and disjunctions of literals in the head of some of the clauses; that is it will be a disjunctive extended logic program with exceptions. As in Arenas et al. (1999) the method considers a set of integrity constraints, IC, written in the standard format $? \bigwedge_{i=1}^n P_i(x_i) ? ? \bigwedge_{i=1}^m (\neg Q_i(y_i)) ? ?$ where $? ?$ is a formula containing only built-in predicates, and there is an implicit universal quantification in front. This method specifies the repairs of the database, D, that violate IC, by means of a logical program with exceptions $?^D$. In $?^D$ for each predicate P a new predicate P' is introduced and each occurrence of P is replaced by P' . More specifically, $?^D$ is obtained by introducing:

1. **Persistence Defaults.** For each base predicate P , the method introduces the persistence defaults:

The predicate P' is the repaired version of the predicate P , so it contains the tuples corresponding to P in a repair of the original database.

2. **Stabilizing Exceptions.** From each IC and for each negative literal $not\ Q_{i0}$ in IC , the negative exception clause is introduced:

where \neg is a formula that is logically equivalent to the logical negation of φ . Similarly, for each positive literal P_{i1} in the constraint the positive exception clause

is generated. The meaning of the Stabilizing Exceptions is to make the ICs be satisfied by the new predicates. These exceptions are necessary but not sufficient to ensure that the changes the original subject should be subject to, in order to restore consistency, are propagated to the new predicates.

3. **Triggering Exceptions.** From the IC in standard form the disjunctive exception clause

$$\bigvee_{i=1..n} P'_i(x_i) \vee \bigvee_{i=1..m} Q'_i(y_i) \leftarrow \bigwedge_{i=1..n} not\ P_i(x_i), \bigwedge_{i=1..m} Q_i(y_i), \varphi$$

is produced.

The program φ^D constructed as shown above is a 'disjunctive extended repair logic program with exceptions for the database instance D '. In φ^D positive defaults are blocked by negative conclusions, and negative defaults, by positive conclusions.

Example 19 Consider the database $D = \{p(a), q(b)\}$ with the inclusion dependency ID :

In order to specify the database repairs the new predicates p' and q' are introduced. The resulting repair program has four default rules expressing that p' and q' contain exactly what p and q contain, resp.:

- $p'(x) ? p(x);$
- $q'(x) ? q(x);$
- $\neg p'(x) ? \text{ not } p(x) \text{ and}$
- $\neg q'(x) ? \text{ not } q(x);$

two stabilizing exceptions:

- $q'(x) ? p'(x);$
- $\neg p'(x) ? \neg q'(x);$

and the triggering exception:

- $\neg p'(x) ? q'(x) ? p(x), \text{ not } q(x).$

The e-answer sets are $\{ p(a), q(b), p'(a), q'(b), \neg p'(a) \}$ and $\{ p(a), q(b), p'(a), q'(b), q'(b) \}$ that correspond to the two expected database repairs.

The method can be applied to a set of domain independent binary integrity constraints IC , that is the constraint can be checked w.r.t. satisfaction by looking to the active domain, and in each IC appear at most two literals.

Rewriting into Disjunctive queries (Greco-Zumpano)

In (Greco and Zumpano, 2000) a general framework for computing repairs and consistent answers over inconsistent databases with universally quantified variables was proposed. The technique is based on the rewriting of constraints into extended disjunctive rules with two different forms of negation (negation as failure and classical negation). The disjunctive program can be used for two different purposes: compute ‘repairs’ for the database, and produce consistent answers, i.e. a maximal set of atoms which do not violate the constraints. The technique is sound and complete (each stable model defines a repair and each repair is derived from a stable model) and more general than techniques previously proposed.

More specifically, the technique is based on the generation of an extended disjunctive program LP derived from the set of integrity constraints. The repairs for the database can be generated from the stable models of LP whereas the computation of the consistent answers of a query (g, P) can be derived by considering the stable models of the program $P \cup LP$ over the database D .

Let c be a universally quantified constraint of the form

$$\forall X [B_1 \wedge \dots \wedge B_k \wedge \text{not } B_{k+1} \wedge \dots \wedge \text{not } B_n \wedge \phi \supset B_\theta]$$

then, $dj(c)$ denotes the extended disjunctive rule

$$\begin{aligned} \neg B'_1 \vee \dots \vee \neg B'_k \vee B'_{k+1} \vee \dots \vee B'_n \vee B'_\theta \vee (B'_1 \vee B'_\theta), \dots, (B'_k \vee B'_\theta), \\ (not\ B'_{k+1} \vee \neg B'_{k+1}), \dots, (not\ B'_n \vee \neg B'_n), \phi, \\ (not\ B'_\theta \vee \neg B'_\theta), \end{aligned}$$

where B'_i denotes the atom derived from B_i , by replacing the predicate symbol p with the new symbol p_d if B_i is a base atom otherwise is equal to false. Let IC be a set of universally quantified integrity constraints, then $DP(IC) = \{ dj(c) \mid c \in IC \}$ whereas $LP(IC)$ is the set of standard disjunctive rules derived from $DP(IC)$ by rewriting the body disjunctions.

Clearly, given a database D and a set of constraints IC , $LP(IC)_D$ denotes the program derived from the union of the rules $LP(IC)$ with the facts in D whereas $SM(LP(IC)_D)$ denotes the set of stable models of $LP(IC)_D$ and every stable model is consistent since it cannot contain two atoms of the form A and $\neg A$. The following example shows how constraints are rewritten.

Example 20 Consider the following integrity constraints:

- $\forall X [p(X) \vee \text{not } s(X) \vee q(X)]$
- $\forall X [q(X) \vee r(X)]$

and the database D containing the facts $p(a)$, $p(b)$, $s(a)$ and $q(a)$.

The derived generalized extended disjunctive program is defined as follows:

$$\begin{aligned}
\neg p_d(X) \vee s_d(X) \vee q_d(X) &\leftarrow (p(X) \vee p_d(X)) \wedge (\text{not } s(X) \vee \neg s_d(X)) \wedge (\text{not } q(X) \\
&\quad \vee \neg q_d(X)). \\
\neg q_d(X) \vee r_d(X) &\leftarrow (q(X) \vee q_d(X)) \wedge (\text{not } r(X) \vee \neg r_d(X)).
\end{aligned}$$

The above rules can now be rewritten in standard form. Let P be the corresponding extended disjunctive Datalog program. The computation of the program P_D gives the following stable models:

$$\begin{aligned}
M_1 &= D \cup \{ \neg p_d(b), \neg q_d(a) \}, & M_2 &= D \cup \{ \neg p_d(b), r_d(a) \}, \\
M_3 &= D \cup \{ \neg q_d(a), s_d(b) \}, & M_4 &= D \cup \{ r_d(a), s_d(b) \}, \\
M_5 &= D \cup \{ q_d(b), \neg q_d(a), r_d(b) \} \text{ and } M_6 &= D \cup \{ q_d(b), r_d(a), r_d(b) \}.
\end{aligned}$$

A (generalized) extended disjunctive Datalog program can be simplified by eliminating from the body rules all literals whose predicate symbols are derived and do not appear in the head of any rule (these literals cannot be true). For instance, the generalized rules of the above example can be rewritten as

$$\begin{aligned}
\neg p_d(X) \vee s_d(X) \vee q_d(X) &\leftarrow p(X), \text{ not } s(X), (\text{not } q(X) \vee \neg q_d(X)) \\
\neg q_d(X) \vee r_d(X) &\leftarrow (q(X) \vee q_d(X)), \text{ not } r(X)
\end{aligned}$$

because the predicate symbols p , $\neg s_d$ and $\neg r_d$ do not appear in the head of any rule. As mentioned before, the rewriting of constraints into disjunctive rules is useful for both i) making the database consistent through the insertion and deletion of tuples, and ii) computing consistent answers leaving the database inconsistent.

Computing Database Repairs

Every stable model can be used to define a possible repair for the database by interpreting new derived atoms (denoted by the subscript "d") as insertions and deletions of tuples. Thus, if a stable model M contains two atoms $\neg p_d(t)$ (derived atom) and $p(t)$ (base atom) we deduce that the atom $p(t)$ violates some constraints and, therefore, it must be deleted. Analogously, if M contains the derived atoms $p_d(t)$ and does not contain $p(t)$ (i.e. $p(t)$ is not in the database) we deduce that the atom $p(t)$ should be inserted in the database. We now formalize the definition of repaired database.

Given a database schema $DS = (Rs, IC)$ and a database D over DS . Let M be a stable model of $LP(IC)_D$, then, a repair for D is a pair

$$R(M) = (\{ p(t) \mid p_d(t) \in M \wedge p(t) \notin D \}, \{ p(t) \mid \neg p_d(t) \in M \wedge p_d(t) \in D \}).$$

Given a database schema $DS = (R_s, IC)$ and a database D over DS a repair for D is a pair of sets of atoms (R^+, R^-) such that 1) $R^+ \cap R^- = \emptyset$, 2) $D \cup R^+ - R^- \models IC$ and 3) there is no pair $(S^+, S^-) \neq (R^+, R^-)$ such that $S^+ \cup S^- \models IC$. The database $D \cup R^+ - R^-$ will be called the repaired database.

Thus, repaired databases are consistent databases which are derived from the source database by means of a minimal set of insertion and deletion of tuples. Given a repair R for D , R^+ denotes the set of tuples which will be added to the database whereas R^- denotes the set of tuples of D which will be canceled. In the following, for a given repair R and a database D , $R(D) = D \cup R^+ - R^-$ denotes the application of R to D .

Example 21 Assume we are given a database $D = \{p(a), p(b), q(a), q(c)\}$ with the inclusion dependency $(\exists X) [p(X) \rightarrow q(X)]$. D is inconsistent since $p(X) \rightarrow q(X)$ is not satisfied. The repairs for D are $R_1 = (\{q(b)\}, \emptyset)$ and $R_2 = (\emptyset, \{p(b)\})$ producing, respectively, the repaired databases $R_1(D) = \{p(a), p(b), q(a), q(c), q(b)\}$ and $R_2(D) = \{p(a), q(a), q(c)\}$.

Example 22 Consider the integrity constraint $IC = \{(\exists X, Y, Z) [Teaches(X, Y), Teaches(X, Y) \rightarrow Y=Z]\}$ over the database D of Example 1. The associated disjunctive program $DP(IC)$ is

which can be simplified as follows

since the predicate symbol $Teaches_d$ does not appear in any positive head atom.

The program $LP(IC)_D$ has two stable models $M_1 = \{\neg Teaches_d(c2, p2)\} \cup D$ and $M_2 = \{\neg Teaches_d(c2, p3)\} \cup D$. The associated repairs are $R(M_1) = (\{\}, \{Teaches_d(c2, p2)\})$ and are $R(M_2) = (\{\}, \{Teaches_d(c2, p3)\})$ denoting, respectively, the deletion of tuples $Teaches_d(c2, p2)$ and $Teaches_d(c2, p3)$.

The technique is sound and complete:

- (Soundness) for every stable model M of $LP(IC)_D$, $R(M)$ is a repair for D ;
- (Completeness) for every database repair S for D there exists a stable model M for $LP(IC)_D$ such that $S = R(M)$.

Example 23 Consider the database of Example 5. The rewriting of the integrity constraint $(\neg \exists X) [p(X) \vee q(X)]$, produces the disjunctive rule

which can be rewritten into the simpler rule r'

The program P_D , where P is the program consisting of the disjunctive rule r' , has two stable models $M_1 = D \vee \{\neg p_d(b)\}$ and $M_2 = D \vee \{q_d(b)\}$. The derived repairs are $R(M_1) = (\{\}, \{p(b)\})$ and $R(M_2) = (\{q(b)\}, \{\})$ corresponding, respectively, to the deletion of $p(b)$ and the insertion of $q(b)$.

Computing Consistent Answer

We now consider the problem of computing a consistent answer without modifying the (possibly inconsistent) database. We assume the truth value of tuples, contained in the database or implied by the constraints, may be either *true* or *false* or *undefined*.

Given a database schema $DS = (Rs, IC)$ and a database D over DS , an atom A is true (resp. false) with respect to (D, IC) if A belongs to all repaired databases (resp. there is no repaired database containing A). The set of atoms which are neither true nor false are undefined.

Thus, true atoms appear in all repaired databases whereas undefined atoms appear in a proper subset of repaired databases. Given a database D and a set of integrity constraints IC , the application of IC to D , denoted by $IC(D)$, defines the three distinct sets of atoms: $IC(D)^+$ (true atoms), $IC(D)^u$ (undefined atoms) and $IC(D)^-$ (false atoms).

- $IC(D)^+ = \{p(t) \mid p(t) \in D \text{ and } M \models SM(LP(IC)_D) \text{ is } \neg p_d(t) \in M \} \vee \{p(t) \mid p(t) \in D \text{ and } M \models SM(LP(IC)_D) \text{ is } p_d(t) \in M \}$
- $IC(D)^- = \{p(t) \mid p(t) \in D \text{ and } M \models SM(LP(IC)_D) \text{ is } \neg p_d(t) \in M \} \vee \{p(t) \mid p(t) \in D \text{ and } M \models SM(LP(IC)_D) \text{ is } p_d(t) \in M \}$
- $IC(D)^u = \{p(t) \mid p(t) \in D \text{ and } \nexists M_1, M_2 \models SM(LP(IC)_D) \text{ s.t. } \neg p_d(t) \in M_1 \text{ and } \neg p_d(t) \in M_2 \} \vee \{p(t) \mid p(t) \in D \text{ and } \nexists M_1, M_2 \models SM(LP(IC)_D) \text{ s.t. } p_d(t) \in M_1 \text{ and } p_d(t) \in M_2 \}$

The *consistent answer* of a query Q on the database D , denoted as $Q(D, IC)$, gives three sets, denoted as $Q(D, IC)^+$, $Q(D, IC)^-$ and $Q(D, IC)^\mu$. These contain, respectively, the sets of g-tuples which are *true* (i.e. belonging to $Q(D')$ for all repaired databases D'), *false* (i.e. not belonging to $Q(D')$ for all repaired databases D') and *undefined* (i.e. set of tuples which are neither true nor false) and are defined as follows:

- $Q(D, IC)^+ = \{g(t) \mid g(t) ? D \text{ and } M ? SM((P ? LP(IC))_D) \text{ is } \neg g_d(t) ? M\} ? \{g(t) \mid g(t) ? D \text{ and } M ? SM((P ? LP(IC))_D) \text{ is } g_d(t) ? M\}$
- $Q(D, IC)^- = \{g(t) \mid g(t) ? D \text{ and } M ? SM((P ? LP(IC))_D) \text{ is } \neg g_d(t) ? M\} ? \{g(t) \mid g(t) ? D \text{ and } M ? SM((P ? LP(IC))_D) \text{ is } g_d(t) ? M\}$
- $Q(D, IC)^\mu = \{g(t) \mid g(t) ? D \text{ and } ? M_1, M_2 ? SM((P ? LP(IC))_D) \text{ s.t. } \neg g_d(t) ? M_1 \text{ and } \neg g_d(t) ? M_2\} ? \{g(t) \mid g(t) ? D \text{ and } ? M_1, M_2 ? SM((P ? LP(IC))_D) \text{ s.t. } g_d(t) ? M_1 \text{ and } g_d(t) ? M_2\}$

For instance, in Example 21 the set of true tuples are those belonging to the intersection of the two models, that is $p(a)$, $q(a)$ and $q(c)$, whereas the set of undefined tuples are those belonging to the union of the two models and not belonging to their intersection.

Example 24 Consider the database of Example 17. To answer a query it is necessary to define, first, the atoms which are true, undefined and false:

- $IC(D)^+ = \{Supply(c1, d1, i1), Class(i1, t)\}$, the set of true atoms
- $IC(D)^\mu = \{Supply(c2, d2, i2), Class(i2, t)\}$, the set of undefined atoms.

The atoms not belonging to $IC(D)^+$ and $IC(D)^\mu$ are false.

The answer to the query $(Class, \{ \})$ gives the tuple $(i1, t)$.

Observe that for every database D over a given schema $DS = (Rs, IC)$, for every query $Q = (g, P)$ and for every repaired database D'

- each atom $A ? Q(D, IC)^+$ belongs to the stable model of $P_{D'}$ (soundness)
- each atom $A ? Q(D, IC)^-$ does not belong to any stable model of $P_{D'}$ (completeness).

Example 25 Consider the integrated database $D = \{Teaches(c1,p1), Teaches(c2,p2), Teaches(c2,p3)\}$ of Example 1 and the functional dependency defined by the key of relation *Teaches* which can be defined as

The disjunctive program LP_D has two stable models: $M_1 = D \cup \{\neg Teaches_d(c2, p2)\}$ and $M_2 = D \cup \{\neg Teaches_d(c2, p3)\}$. Therefore, the set of facts which can be assumed to be true contains the single element $Teaches(c1,p1)$.

We conclude by mentioning that the technique above proposed has been further extended by considering constraints and priorities on the alternative repairs (Greco and Zumpano, 2000b; Greco et al., 2001).

CONCLUSION

The technique proposed in (Agarwal et al., 1995) only considers constraints defining functional dependencies and it is sound only for the class of databases having dependencies determined by the primary key consisting of a single attribute. The technique proposed by Dung considers a larger class of functional dependencies where the left parts of the functional dependencies are keys. Both techniques consider restricted cases but the computation of answers can be done efficiently (in polynomial time).

The technique proposed in (Lin and Mendelzon, 1996), generally, stores disjunctive information. This makes the computation of answers more complex, although the computation becomes efficient if the ‘merging by majority’ technique can be applied. However, the use of the majority criteria involves discarding inconsistent data, and hence the loss of potentially useful information.

Regarding to the technique proposed in (Arenas et al., 1999), it has been shown to be complete for universal binary integrity constraints and universal quantified queries. This technique is more general than the previous ones. However, the rewriting of queries is complex since the termination conditions are not easy to detect and the computation of answers generally is not guaranteed to be polynomial. The technique proposed by Greco and Zumpano is the most general but the computation of answers is also more complex.

ENDNOTE

1. Work partially supported by MURST grants under the projects "Data-X" and D2I. The first author is also supported by ISI-CNR.

REFERENCES

- Abiteboul, S., Hull, R., Vianu, V. (1995). *Foundations of Databases* Addison-Wesley.
- Arenas, M., Bertossi, L., Chomicki, J. (1999). *Consistent Query Answers in Inconsistent Databases*. *Proc. Int. Conf. on Principles of Database Systems*, 68–79.
- Arenas, M., Bertossi, L., Chomicki, J. (2000). *Specifying and querying Database repairs using logic Programs with Exceptions*. *Proc. Int. Conf. On Flexible Query Answering*, 27–41.
- Argarwal, S. S. (1992). *Flexible Relation: A model for Data in Distributed, Autonomous and Heterogeneous Databases*, PhD Thesis, Department of Electrical Engineering, Stanford University, June.
- Argarwal, S., Keller, A.M., Wiederhold, G. and K. Saraswat. (1995). *Flexible Relation: an Approach for Integrating Data from Multiple, Possibly Inconsistent Databases*. *Proc. of the IEEE Int. Conf. on Data Engineering*, 495–504.
- Baral, C., Kraus, S., Minker, J. (1991). *Combining Multiple Knowledge Bases*. *IEEE-Trans. on Knowledge and Data Engineering*, 3(2), 208–220, 1991.
- Baral, C., Kraus, S., Minker, J., Subrahmanian, V. S., *Combining Knowledge Bases Consisting of First Order Theories*. *Proc. Int. Symp. on Methodologies for Intelligent Systems*, pp. 92–101, 1991.
- Bry, F. (1997). *Query Answering in Information System with Integrity Constraints*, In *IFIP WG 11.5 Working Conf. on Integrity and Control in Inform. System*.
- Dung, P. M. (1996). *Integrating Data from Possibly Inconsistent Databases*. *Proc. Int. Conf. on Cooperative Information Systems*, 58–65.
- Eiter, T., Gottlob, G. and Mannila, H. (1997). *Disjunctive Datalog*, *ACM Transactions on Database Systems*, 22(3), 364–418.
- Gelfond, M., Lifschitz, V. (1991). *Classical Negation in Logic Programs and Disjunctive Databases*. *New Generation Computing*, 3(4), 365–386.
- Grant, J., Subrahmanian, V. S. (1995). *Reasoning in Inconsistent Knowledge Bases*. *IEEE-Transaction on Knowledge and Data Engineering*, 7(1), 177–189.
- Greco, S., Saccà, D. (1990). *Negative Logic Programs*. In *Proc. North American Conference on Logic Programming*, 480–497.
- Greco, S. (1999). *Minimal founded semantics for disjunctive logic programming*, *Int. Conf. on Logic Programming and Nonmonotonic Reasoning*, 221–235.
- Greco, S., Zumpano, E. (2000). *Querying Inconsistent Databases*, *Proc. Int. Conf. on Logic Programming and Automated Reasoning*, 308–325.
- Greco, S., Zumpano, E. (2000). *Computing Repairs for Inconsistent Data-bases*, *Proc. Int. Symp. on Cooperative Database System for Advanced Applications*, 33–40.
- Greco, G., Greco, S., Zumpano, E. (2001). *A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases*, *Proc. Int. Conf. on Logic Programming*.
- Kanellakis, P. C. (1991). *Elements of Relational Database Theory*. *Handbook of Theoretical Computer Science*, Vol. 2, J. van Leewen (ed.), North-Holland.
- Kowalski, R. A., Sadri, F. (1991). *Logic Programs with Exceptions*. *New Generation Computing*, 9(3/4), 387–400.
- Lin, J. (1996). *Integration of Weighted Knowledge Bases*. *Artificial Intelligence*, 83(2), 363–378.

- Lin, J. (1996). *A Semantics for Reasoning Consistently in the Presence of Inconsistency*. *Artificial Intelligence* 86(1), 75–95.
- Lin, J., Mendelzon, A. O. (1996). *Merging Databases Under Constraints*. *Int. Journal of Cooperative Information Systems*, 7(1), 55–76.
- Lin, J., Mendelzon, A. O., (1999). *Knowledge Base Merging by Majority*, in R. Pareschi and B. Fronhoefer (eds.), *Dynamic Worlds: From the Frame Problem to Knowledge Management*, Kluwer.
- Minker, J. (1982). *On Indefinite Data Bases and the Closed World Assumption*, *Proc. 6-th Conf. on Automated Deduction*, 292–308.
- Subrahmanian, V.S. (1994). *Amalgamating Knowledge Bases*. *ACM Transaction on Database Systems*, 19(2), 291–331.
- Ullman, J.K. (1988). *Principles of Database and Knowledge-Base Systems*, Vol. 1, Computer Science Press, Rockville, Md.
- Zaniolo, C. (1984). *Database Relations with Null Values*. *Journal of Computer and System Sciences*, 142–166.

Chapter VII: Translating Advanced Integrity Checking Technology to SQL

Hendrik Decker, Instituto Tecnológico de Informática,

Spain

INTRODUCTION

The main goal of this chapter is to arrive at a coherent technology for deriving efficient SQL triggers from declarative specifications of arbitrary integrity constraints. The user may specify integrity constraints declaratively as closed queries in predicate calculus syntax (i.e., sentences in the language of first-order logic, abbr. FOL), as *datalog* denials, as query conditions in SQL `WHERE` clauses, or in some other, possibly more user-friendly manner (e.g., via a dialog-driven graphical or natural language interface which internally translates to equivalent `WHERE` clause conditions). As we are going to see, the triggers derived from such specifications behave such that whenever some update event would violate any of the integrity constraints, one or several of the triggers derived from that constraint are activated in order to enforce the constraint. That is, the violation is either prevented by rolling back the update or repaired instantly by subsequent further updates.

In this chapter, we describe how to implement advanced *datalog* technology for integrity checking in the framework of SQL. That is, we show how to represent and evaluate arbitrarily complex constraints in SQL without incurring major disadvantages usually associated to integrity checking in knowledge-rich applications. Error-prone procedural specification and laborious maintenance of integrity constraints is avoided by the declarativity of *datalog*. The cost of evaluation is considerably reduced by an automated translation of declarative specifications to SQL triggers. That way, the advantages of

declarativity of specification and efficiency of execution can be combined, while the performance disadvantage of `CHECK` clauses in `ASSERTION` statements, as well as the disadvantage of procedural specifications by the user, is avoided.

As already indicated, three different, though mutually related declarative languages for specifying integrity constraints are addressed in this chapter: FOL, *datalog* (i.e., Horn clauses with negation-as-failure) and SQL. *Datalog* is primarily of historical interest, but it fits well into our presentation since most of the techniques discussed in this chapter have been developed in the *datalog* framework of deductive database systems (cf. (Ramakrishnan & Ullman, 1995) for a survey). For representing arbitrarily complex integrity constraints in *datalog*, an extended first-order syntax is needed in the body of *datalog* queries. Thus, FOL is a natural choice for expressing integrity constraints.

In the first section, we survey the history and the state of the art of integrity constraint checking. In '[Principles of Simplified Integrity Checking](#),' we recapitulate common principles of simplifying integrity checking. In '[An SQL Syntax for Integrity Constraints](#),' we first define a syntax for integrity constraints as a subset of standard SQL. In '[Translating Principles of Simplified Integrity Checking to SQL](#),' we discuss the applicability of the principles in '[Principles of Simplified Integrity Checking](#)' in an SQL framework. In '[First-Order Logic Representation of Integrity Constraints](#),' we discuss a FOL syntax for integrity constraints which is sufficiently expressive and lends itself well toward a straightforward translation into SQL. In '[Translating Integrity Constraints to SQL Conditions](#),' we describe a translation of constraints in this syntax to `WHERE` clause conditions. In '[Identifying and Specializing Relevant Integrity Constraints](#),' we describe how constraints represented as such conditions can be simplified for the purpose of improving the efficiency of integrity checking. In '[Translating Integrity Constraints to Optimized SQL Triggers](#),' we describe a translation of simplified SQL conditions into equivalent triggers. They closely correspond to what is called "update constraints" in (Decker, 1987). The syntactic transformations, rewritings and simplifications described in '[First-Order Logic Representation of Integrity Constraints](#)' to '[Translating Integrity Constraints to Optimized SQL Triggers](#)' are easily automated. In the conclusion, we summarize the chapter, address related work and point out directions for future work. For simplicity, we assume that updates are single tuple insertions or deletions in base tables. An extension to more general transactions does not pose essential new problems, but dealing with SQL transaction semantics (which are not yet standardized) would become too sumptuous. However, we do address implicit updates of views caused by explicit updates of underlying tables, as well as imposing integrity constraints on views.

In this chapter, we only deal with static integrity, i.e., with database properties that are invariant across all states. In other words, we are not dealing with dynamic integrity, i.e., with properties applying to particular states or particular state transitions. Typically, dynamic integrity constraints are inherently nondeclarative. In general, a purely declarative treatment of dynamic integrity would be possible only if database states are included as a proper domain into the query language, such that they would become ordinary attributes, rather than meta data. To some extent, that might be achievable in SQL implementations which offer `BEFORE` and `AFTER` constructs, for specifying database

states before and after a given update. However, such constructs are usually allowed to be used only within non-declarative triggers. In general, a declarative amalgamation of meta data and application data is out of scope of standard SQL.

We assume the reader is acquainted with SQL as well as with basic notions of *datalog* and predicate calculus, i.e., FOL. For an introduction to SQL, we suggest to consult Date & Darwen (1997), Melton & Simon (1993) or some appropriate links in Ocelot (2001). For *datalog*, we recommend to see any of Ullman (1988), Abiteboul, Hull & Vianu (1995) or Date (1995). Also, several texts on deductive database systems provide thorough introductions to *datalog*, in the form of function-free Horn clause syntax as used in logic programming, e.g., Gallaire, Minker & Nicolas, 1984; Das, 1992; Ceri, Gottlob & Tanca, 1989. All of these references for *datalog* also contain a good deal of background material for predicate logic, as appropriate for the topic of this chapter. Further basic material is provided in Gallaire & Minker (1978) and Kowalski (1979).

HISTORY AND STATE OF THE ART

In this initial section, we are going to outline how declarative database theory and practice evolved together, with regard to the issue of integrity constraint specification and evaluation.

Early and Prime Time History

Arguably, the data description and query facility SQL has become the most successful declarative language worldwide. (For more detailed accounts, cf. Melton & Simon, 1993, McJones, 1997.) Yet, no constructs (and certainly no declarative ones) for expressing database integrity appeared in any of the early (pre-1990) implementations of SQL. This is remarkable since integrity has always been regarded as an important conceptual issue for database management systems, as witnessed by many related publications in the field (early ones are, e.g., Fraser, 1969; Wilkes, 1972; Eswaran & Chamberlin, 1975; Hammer & McLeod, 1975; Nicolas, 1978, 1982; Hammer & Sarin, 1978; Codd, 1979; Bernstein, Blaustein & Clarke, 1980; Bernstein & Blaustein, 1982. Later ones are too numerous to mention). The need to express part of the semantics of databases as invariants, i.e., properties persisting across updates, had been pointed out early on in Minsky (1974). Apparently, the first to propose that integrity constraints should be expressed in first-order predicate calculus (indeed, the most declarative language there is), was Florentin (1974). Perhaps, Stonebraker was the first to come up with the idea to formulate and check integrity constraints declaratively as SQL-like database queries (Stonebraker, 1975).

Referential integrity (a special case of Armstrong's functional dependencies (Armstrong, 1974)) was first included in the 1989 SQL ANSI and ISO standards (cf. (McJones, 1997)). The SQL2 standard of 1992 introduced the `ASSERTION` construct and the `CHECK` option as the most general means to express boolean integrity constraint conditions (cf. Melton & Simon, 1993; Date & Darwen, 1997). In the 1990s, uniqueness constraints, foreign keys, subqueries as well as the `EXISTS` and the `NOT` construct (sometimes also constructs `SOME`

and `ANY`) became fairly common features in most commercial DBMS. Finally, with the addition of recursive query traversal of hierarchically nested view definitions around the turn of the century, the expressiveness of SQL approached Turing-completeness. In other words, arbitrarily general query conditions, and thus arbitrarily general integrity constraints, can now be formulated and evaluated in most relational DBMSs.

State of the Art in Commercial DBMSs

In the previous section, we have seen that, conceptually speaking, SQL has the capacity of unrestricted declarative expressiveness in general, and in particular for avoiding procedural specifications of arbitrarily general integrity constraints. However, until this day, the manuals of most SQL engines still recommend to implement user-defined conditions for semantic data integrity non-declaratively, namely by triggers or stored procedures, for reason of efficiency. In fact, integrity constraints which go beyond the limitations of standard SQL entry level expressiveness typically involve several tables, potentially huge joins, full table scans, nested subqueries, nested negation and the like. Thus, their evaluation easily becomes prohibitively expensive, in terms of computation time, storage and CPU resources. In particular, standard OLTP applications as well as time-critical data warehousing processes for extracting, cleansing, transforming, homogenizing and uploading business data typically cannot afford voluminous expenditures for integrity checking. And indeed, even those DBMS, which do have an `ASSERTION` statement in their repertoire (e.g., Ocelot, 2001), do not really encourage its use, because evaluating such constraints after each update (or at least after each `COMMIT` of a transaction) would fatally hamper their performance. As a consequence, most of the DBMS in practical use are contented with supporting only the following kinds of declarative integrity constraints:

- Domain constraints (i.e., user-defined data types defined by restrictions on the range of standard scalar SQL data types, including options for permitting default and null values),
- Uniqueness constraints (such as enforced by primary keys or unique indices),
- Foreign key constraints.

In other words, commercial implementations of SQL typically offer declarative constructs to express constraints on permitted attribute values (i.e., domain and uniqueness constraints), and to express a simple (though most frequent) kind of functional dependencies (viz., referential integrity, as expressed by foreign keys). The expressive power of these three constructs is quite limited. For example, it is not possible to express, with any combination of the three constructs, that, for each row R_1 in some table T_1 with a column C_1 , there must be a row R_2 in some table T_2 with a column C_2 such that the value of R_1 at C_1 is the same as the value of R_2 at C_2 , as long as no primary key constraint is imposed on R_2 . In general, domain constraints are tied to single columns

only, while uniqueness constraints may apply to a combination of columns, but only within a single table. Foreign key constraints relate columns in at most two tables, where referenced columns additionally have to satisfy a uniqueness condition. That is, foreign key constraints capture total $1:n$ relationships, but neither partial $1:n$ nor $n:m$ relationships, as in the example above.

The failure of expressing $n:m$ relationships between tables declaratively in SQL is just the tip of an iceberg. For semantic integrity constraints that are less common or just slightly more general than the standard ones above, no declarative formalism is available in industrial-strength DBMS. For instance, it is commonplace in datawarehousing and other applications with knowledge-rich data models that users do not want certain constellations of data to occur. Such kind of negative information, which typically encompasses several columns, possibly in different tables, can be expressed very conveniently by so-called "denials" in *datalog*. (A denial is a Horn clause the head of which is either empty or signals inconsistency or violation of integrity in case the body of the clause is satisfied.) Already a straightforward declarative expression of simple denials, e.g., that a person must not be married to more than one other person and cannot be married to him/herself, is out of scope of SQL. In *datalog*, this knowledge is described by the two denials

In SQL-based DBMS products, there are essentially three choices of expression for constraints which are more general than the standard ones mentioned above: either as SQL triggers which take action ("fire") upon predefined updates of particular tables, or as stored procedures which usually are activated by predefined transaction events, or directly embedded in the code of applications which interoperate with the DBMS.

Each of the three options severely compromises the ideal of database declarativity, by dynamically tying the specification of constraints to procedural events. Procedurality of constraint specification entails the known hazards of aggravated maintenance of the database schema and the application programs. On the other hand, SQL manuals usually point out that triggers, stored procedures and dedicated encodings of constraint enforcement within application programs tend to be much less resource-consumptive and much faster than general `CHECK` clauses or `ASSERTION` statements. However, each procedural implementation of integrity conditions has the additional disadvantage of thwarting a possibly large potential of simplification which would speed up their evaluation. In this chapter, we are going to have a closer look at such simplifications and the way they can be enabled and made useful.

History, Continued

Thorough methodologies for simplifying the evaluation of arbitrary integrity constraints in relational databases had been devised in Nicolas (1982), Bernstein & Blaustein (1982) and others (cf. Decker, 1998) for more references). Surprisingly, they never have been taken up by implementors of marketable relational DBMSs. An implementation of the method in Nicolas (1982) for a relational database prototype is reported in Homeier (1981).

In Decker (1985, 1987), this author developed a generalization of the approach in Nicolas (1982) to the deductive case. The resulting research prototype was called *soundcheck*. Similar methods were proposed in Lloyd, Sonenberg & Topor (1987), Sadri & Kowalski (1988) and by many other authors (Celma, Garcia, Mota & Decker, 1994 contains a comparison). Orthogonal approaches to simplify and optimize integrity checking in terms of conjunctive query optimization have been discussed in Elkan (1990); Levy & Sagiv (1993); Gupta, Sagiv, Ullman & Widom (1994); and Ross, Srivastava & Sudarshan (1996) and others. Common to all of them is the declarativity of integrity constraint specification.

Early implementations of *soundcheck* and variants thereof were operational in several versions of prototype knowledge base systems at ECRC (cf., e.g., Bocca, Decker, Nicolas, Vieille & Wallace, 1986; Bocca, 1986; Vieille, Bayer, Küchenhoff & Lefebvre, 1999; Bocca, Dahmen & Freeston, 1992). However, none of these systems ever went commercial. After all, the language of choice on the database market has not been relational algebra nor FOL nor *datalog*, but SQL. In fact, many visions, concepts and achievements of the theoretical databases community have found their way into SQL database systems. As a striking example, materialized views in data warehouses as back-ends of systems for business information management, decision support, enterprise resource planning and customer relationship management come to mind (cf. Ullman's foreword in Gupta & Mumick (1999)). And, with the introduction of recursive queries in the SQL3 standard proposal, nothing much seems to be left which would still distinguish deductive from relational DBMS, from a practical point of view. However, beyond commonplace kinds of constraints, more advanced declarative integrity checking has remained a rather theoretical issue which seems to have never really found its way into practical DBMSs. Rather than trying to explain why that is so, this chapter sets out to show how deductive database technology for integrity checking can be translated to practice just as well as materialized views or complex queries.

State of the Art, Continued

While no major vendor's DBMS product sports advanced declarative integrity checking features, most of them offer triggers, stored procedures and other procedural extensions of SQL with which it is possible to implement constraints (or, more generally, business rules) and their enforcement. In the literature, various combinations of commonplace declarative and procedural SQL constructs for implementing business rule applications have been proposed, e.g., Cochrane, Pirahesh & Mattos, 1996; Martin & Perrin, 1997; Liu & Ong, 1999), but none of them goes beyond the rudimentary declarativeness of standard SQL implementations.

The author of Date (2000) makes an emphatic case for a declarative understanding of business rules and their deployment in the framework of relational database technology. However, he gives little guidance for how it could actually be done. A number of enterprises offer proprietary tools for supporting SQL-based business rule applications (e.g., USoft, 2001; Knowledge Partners Inc., 2001; Omnibuilder, 2001; Ross & Lam, 2001). Similar to active database systems, most of them have to cope with the potential unpredictability of mutually dependent triggers designed by the user (cf., e.g., Widom & Ceri, 1996; Ceri, Cochrane & Widom, 2000). Some of these business rule tools support the use of declarative SQL `WHERE` clause conditions for generating from them event-driven business rules. But the logic of the rule generation process remains opaque, i.e., there is no way to assure by a proof of correctness that the outcome is going to behave exactly as intended by the declarative specification. Likewise, any systematics for making time and storage consumption of generated triggers more efficient than evaluating the original `WHERE` clauses remains hidden and inscrutable in such business rule systems.

In (Decker, 2001), we described a detailed method for generating provably correct triggers from declarative integrity constraints specified as first-order predicate calculus sentences. The method essentially consists of a translation of the results of the *soundcheck* approach (Decker, 1987) to SQL. In this chapter, we elaborate on some aspects which, due to space limitation, have received only scant treatment in Decker (2001). We put some more emphasis on existing implementations of SQL and their provision of means to express and support user-defined integrity constraints. In a sense, this chapter can be taken as a technical addendum to Date (2000) that has been sorely missed by several reviewers of the latter (cf. the customers reviews for Date, 2000 at Amazon, 2001).

Related Translations

A large share of this chapter deals with translations from one form of representation of integrity constraints into another. The ultimate goal is to automatically translate integrity constraints into simplified SQL conditions, as characterized in more detail in '[Principles of Simplified Integrity Checking](#).' Thus, the question arises if we could take advantage of any already existing translations in the literature. In this subsection, we shortly discuss this question.

In Ullman (1988), translations of specifications from relational algebra to "logical rules" (i.e., *datalog*) and vice-versa are sketched. Ullman is mainly interested in demonstrating that both representation formalisms are equivalent in terms of expressive power. However, he is not concerned about the efficiency of evaluating the results of the sketched translations, while efficiency is an essential concern of the translations described in this chapter. We are going to describe a translation from FOL (which is neither relational algebra nor *datalog*) to SQL. As far as the author of this chapter is aware, there is no text of comparable generality in the literature which would deal with a translation of FOL to SQL, let alone an efficiency-conscious one. Our translation exploits the structural properties of a specific FOL normal form syntax (called "range form" in

[‘Definition of the Range Form Syntax’](#)), which requires a level of attention to detail as we have tried to achieve in [‘First-Order Logic Representation of Integrity Constraints’](#) and [‘Translating Integrity Constraints to SQL Conditions.’](#)

In Van Gelder & Topor (1991), a translation of queries and integrity constraints in relational calculus syntax to a specific normal form in relational algebra is described. In fact, it would be possible to translate FOL to relational calculus, then apply the translation in Van Gelder & Topor (1991), and then translate from relational algebra to SQL, along the lines described in Ullman (1988). However, rather than taking a detour via relational calculus and algebra, we prefer to specify a direct translation from FOL to SQL. As we are going to see in more detail later on, the FOL syntax used in this chapter is an extension of the usual *datalog* syntax of conjunctive queries, in order to achieve the generality of expressive power needed for arbitrary integrity constraints. In SQL, this extension essentially corresponds to using the constructs `EXISTS` and `NOT EXISTS` in nested `WHERE` clauses.

PRINCIPLES OF SIMPLIFIED INTEGRITY CHECKING

Typically, integrity constraints involve universal quantifications, i.e., generalizations over large extents of one or several tables, such that their evaluation can become critically costly. SQL engines are optimized for checking simple domain constraints, uniqueness constraints and referential constraints that are readily expressible in standard SQL, but in order to make more general constraints behave efficiently, the designer is usually asked to resort to triggers and stored procedures. However, as already mentioned in [‘History and State of the Art,’](#) it is known since a long time to the logic & databases community that the declarativity of integrity constraint specification does not need to be sacrificed in order to obtain an efficient evaluation. One approach developed to that end was *soundcheck* (cf. [‘History, continued’](#)).

In this section, we are going to outline the *soundcheck* approach for simplifying integrity constraints. The purpose of simplifying the general form of constraints to more simple ones is to improve the efficiency of evaluating them. We present the approach as a succession of six phases. Except phase I, this approach has originally been used in (Nicolas, 1982), and all or part of it is effectively used in one way or another (possibly with different sequencing or interleaving of phases) in most known methods for integrity checking. In later sections, we show how it can also be made available to SQL databases. The six phases are listed below. The example discussed in the following section illustrates what the headings I–VI mean. In [‘Principles for Simplified Integrity Checking, continued,’](#) we discuss the six phases in general and present criteria for their effective application.

I. *Generate the difference between the old and the new state*

- II. *Skip idle updates*
- III. *Focus on relevant integrity constraints*
- IV. *Specialize relevant constraints*
- V. *Optimize specialized constraints*
- VI. *Evaluate optimized constraints*

An Example of Simplified Integrity Checking

For illustrating phases I – VI above, let us consider an update of an SQL database with relations for workers and managers, defined as follows:

```
CREATE TABLE(worker(CHAR[ ] name, CHAR[ ] department, DATE start))
CREATE TABLE(manager (CHAR[ ] name)).
```

The start attribute is supposed to contain the date when the worker was employed. The other attributes are self-explaining. Now, suppose there is an integrity constraint requiring that no worker is a manager. That can be expressed by the SQL condition

```
NOT EXISTS (SELECT * FROM worker, manager WHERE worker.name =
manager.name).
```

If the number of workers and managers is large (e.g., in the database of a large company with possibly hundreds of thousands of workers and a huge management hierarchy), then checking whether this constraint is violated or not can be very costly. The number of facts to be retrieved and compared from the two relations is in the order of the product of their respective sizes (i.e., the cardinality of their Cartesian product), whenever the constraint is checked. However, we are going to see that the frequency and the amount of accessing stored facts can be significantly reduced by taking steps I – VI. But before we go through them, let us briefly deal with a possible objection at this stage.

SQL programmers might feel compelled to point out that the constraint above is probably much easier checked by a trigger such as

```
CREATE TRIGGER ON worker FOR INSERT :
  IF EXISTS
    (SELECT * FROM inserted, manager WHERE inserted.name =
    manager.name)
  ROLLBACK
```

which needs to be evaluated only for each attempt to insert a row into *worker*. Evaluation only needs to access the stored *manager* relation and a singleton (or, in general, small-sized) built-in cached relation *inserted* of rows to be inserted to *worker*, but not the

stored part of the `worker` relation. And indeed, the automatic translation of the SQL condition above into equivalent triggers as described later on produces a trigger which is essentially the same as the one above. However, things are less straightforward than this easy example might suggest. Integrity constraints can be much more complicated. It is well known that triggers may bring about unforeseen effects that are hard to control. In general, we argue that using correct mechanisms for translating declarative specifications into more efficient procedural code is preferable to a more error-prone hand-coding of triggers. This point of view is of course in line with the general philosophy of declarative languages. For instance, it is easy to overlook that the integrity constraint above is "symmetric" for `worker` and `manager`, since it also requires implicitly that somebody who is promoted to a manager is not a worker, which thus necessitates a second trigger for insertions into `manager`. If only a single trigger for `worker` or `manager` is present, then updates of the other relation which violate the constraint will go unnoticed. However, the translation of *soundcheck* to SQL also produces the second trigger, as we are going to see later on.

Now, back to the six phases. Let `INSERT worker(Fred, sales, 01/01/2001)` be an update. Then, going from I through VI means the following:

I. *Generate the difference between the old and the new state*

In case there are database views the definition of which involves `worker`, the explicit update `INSERT worker(Fred, sales, 01/12/2001)` may have implicit update consequences on such views. Thus, all implicit updates which are consequences of the explicit update must be generated, and for each of them, all steps in phases II – VI need to be considered. For example, suppose there is a view `pension` which contains all workers who are entitled to obtain a pension (e.g., if their start date is at least five years ago), and a constraint on that view (e.g., expressing an exceptional condition under which pension is not granted). Then, that constraint needs to be evaluated only if `Fred` is entitled for pension; otherwise, no additional constraint needs to be checked.

II. *Skip idle updates*

If `Fred` already has been a worker (possibly in some other department) before the `INSERT` statement was launched, then it clearly is not necessary to check again the constraint that he must not be a manager, because it has already been known to the database that `Fred` is not a manager (since he has been a worker and because the constraint has been required to be satisfied in each database state).

III. *Focus on relevant integrity constraints*

Unless II applies, the constraint that no worker must be a manager is clearly relevant and must be checked. However, any integrity constraint which does not involve the relation `worker` needs not be checked. More precisely, each constraint which is not relevant for the *insertion* of rows into the `worker` table needs not be checked. For instance, a constraint which requires that in each department, there must be some least number of workers, is not relevant for insertions but only for deletions in the `worker` table. A general rule for identifying relevant constraints according to (Nicolas, 1982) is discussed in [‘Principles for Simplified Integrity Checking, continued.’](#)

IV. *Specialize relevant constraints*

For the given `INSERT` statement, the `WHERE` clause of the SQL condition

```
EXISTS (SELECT * FROM worker, manager WHERE worker.name =
                                             manager.name)
```

can be specialized to a much less expensive form:

```
EXISTS (SELECT * FROM worker, manager WHERE
worker.name = 'Fred' AND worker.name = manager.name)
```

Specializing constraints in general is discussed in [‘Principles for Simplified Integrity Checking, continued.’](#)

V. *Optimize specialized constraints*

Clearly, the specialized condition in IV can be optimized to the statement

```
EXISTS (SELECT * FROM manager WHERE name = 'Fred')
```

VI. *Evaluate optimized constraints*

After having gone through I to V, evaluation of the resulting query whether Fred is a manager is easy. Looking up a single fact in a stored relation is, of course, much less costly than having to evaluate the original integrity constraint in its full generality (not to mention other constraints that might be unnecessarily checked if phase III has been ignored).

The example above is an extremely simple one. (Even the checking of referential constraints is more involved; cf. example 4 in [‘Translating Integrity Constraints to Optimized SQL Triggers’](#)). However, we are going to see that the same proportions of

simplification and reduction of necessary work can be obtained systematically for arbitrarily complex integrity constraints. Moreover, for integrity constraints involving nested `EXISTS` and `NOT` constructs in their representation as an SQL `WHERE` clause condition, analyzing and translating such constraints into equivalent triggers is more intricate than in the example above.

Principles for Simplified Integrity Checking, Continued

In this section, we walk again through the six phases of simplified integrity checking, generalizing the lessons learned from the example in the previous section into principles that apply to arbitrary cases.

Let \mathcal{D} be a relational database. Suppose that views in \mathcal{D} , if any, are defined as in *datalog*, i.e., as conjunctions of literals where each variable in a negative literal occurs in at least one non-negated literal. Further, let \mathcal{IC} be a first-order predicate calculus sentence representing some integrity constraint in \mathcal{D} , and `UPDATE fact` be an update request, where `UPDATE` stands for either `INSERT` or `DELETE`, and `fact` be a ground base fact with a predicate, say, p , i.e., `fact` is a row of values to be inserted to or deleted from a table named p where p is not a view but a basic table. Then, going from I through VI means the following:

I. *Generate the difference between the old and the new state*

For each view v in \mathcal{D} and each occurrence A of an atom in the definition of v which matches `fact` (i.e., `fact` and A have the same predicate and their column values can be unified), inserting or deleting `fact` may implicitly update v . More precisely: The insertion of `fact` may cause an implicit insertion in v if A is not negated, and an implicit deletion in v if A is negated. The deletion of `fact` may cause an implicit deletion in v if A is not negated, and an implicit insertion in v if A is negated. In general, the existence of an implicit update of v depends not only on the explicitly updated `fact` but also on other conditions in the view's definition. That possibly makes the generation of all consequences of the explicit update quite intricate. This problem has been studied more in-depth in, e.g., Küchenhoff, 1991; Celma & Decker, 1994; Decker & Celma, 1994; for the very closely related problem of materializing views upon updates (cf. Ross, Srivastava & Sudarshan, 1996; Gupta & Mumick, 1999). Anyway, each implicitly updated `fact` which is a consequence of the original update has to be run through phases II – VI.

II. *Skip idle updates*

As usual in *datalog* implementations, there must never be two identical tuples in a base relation. However, there may be a redundancy of facts defined by views. So, we say an insertion is idle if the `fact` to be inserted is already present in the table or derivable from a view definition, and a deletion is idle if a second (explicit or

view-defined) copy of that fact will persist after the update. For each such idle update, all of phases III – VI can be skipped, and no further potential consequences according to phase I need to be considered. However, depending on the average degree of redundancy in a given database and the complexity of view definitions, the checking for idleness of updates may bring along a considerable overhead. That overhead may even outweigh the gains obtained by avoiding unnecessary integrity checks for idle updates. Therefore, depending on the application, it might be advisable to "skip the skip" of integrity checking for potentially idle updates.

III. *Focus on relevant integrity constraints*

For an explicit update of a table or an implicit update of a view, the general rule for focusing the checking of integrity on those constraints that are potentially relevant at all, is as follows. According to Nicolas (1982), \mathcal{IC} is potentially relevant for the insertion (resp., deletion) of fact, and thus needs to be checked, only if there is an atom A with negative (resp., positive) polarity in \mathcal{IC} which unifies with fact. Otherwise, \mathcal{IC} is not relevant and thus needs not be checked.

Polarity can be defined as follows: A has positive polarity in A ; for formulae B, C , if an occurrence of A has positive (or, resp., negative) polarity in B , then that occurrence has positive (resp., negative) polarity in $A \rightarrow B$, $A \rightarrow B$, $B \rightarrow A$, and also in the universal and the existential closure of B ; A has negative (resp., positive) polarity in $\neg A$ and in $A \rightarrow B$. In this context, it is interesting to note that neither the number of occurrences of an atom nor their polarity is always the same for logically equivalent formulae. For instance, $\text{fact} \rightarrow \neg \text{fact}$ is equivalent to true , i.e., there may be both negative and positive occurrences in one formula and none in an equivalent one. Thus, for minimizing the amount of necessary integrity checking, it should pay off to care for representations with a minimum number of occurrences of atoms. This is reinforced by the circumstance that each relevant constraint has to be checked anew for each matching occurrence of an atom, as we are going to see in point IV. (The observation about the number of occurrences of atoms has been made by D.S. Warren, during the conference presentation of Decker (2001).)

IV. *Specialize relevant constraints*

The focus obtained in phase III can be further narrowed, by specializing variables in constraints that have been identified as relevant in to ground values occurring in facts to be updated. According to Nicolas (1982), the general rule for specializing an integrity constraint which is relevant with regard to the insertion or deletion of a fact is as follows. For \mathcal{IC} and fact as in III, suppose that A is an occurrence of an atom in \mathcal{IC} with negative or, resp., positive polarity which unifies with fact. Let f be an *mgu*, i.e., a most general unifying substitution of fact and A . Then, \mathcal{IC} can be specialized to $\mathcal{IC}?$, where the substitution $?$ is obtained from f by restricting the latter to those variables in A that are universally

quantified in \mathcal{IC} without being dominated by an exists-quantifier \exists (i.e., no \exists occurs on the left of $\forall x$ in \mathcal{IC}). Thus, \exists grounds each such variable x to the matching constant value in fact.

Note that, by definition, the quantification of a variable in a formula can be obtained either by moving all negations innermost or by moving all quantifiers outermost, such that logical equivalence is preserved. That is, negations are moved immediately in front of predicate symbols and eliminating double negation, or by moving all quantifiers outermost, while respecting the equivalences

$\neg \forall x (F) \equiv \exists x (\neg F)$ and $\neg \exists x (F) \equiv \forall x (\neg F)$ and de Morgan's law. Also note that, in most cases, moving a universal quantifier which is dominated by an \exists in front of \forall usually does not preserve the semantics of the formula. However, should it be possible to do so (e.g., in $\forall x \exists y (p(x) \wedge \neg q(y) \wedge q(x))$), then that is beneficial for the efficiency of evaluating the resulting formula, because the more grounded a formula is, the easier it is evaluated.

V. *Optimize specialized constraints*

For optimizing specialized constraints, it is useful to distinguish equivalence-preserving syntactic rewrite optimization, such as described in Nicolas (1982) and Demolombe & Illarramendi (1989), from operational optimization by revising access plans at compile time or even at run time, such as it may be built into a DBMS. The latter is addressed under point VI. Here, we only deal with syntactic rewrites. In fact, they may already be applied beneficially at earlier stages of the simplification process, as indicated in points III and IV, above. However, some particular issues should be observed for generalizing the optimization which is exemplified in point V of '[An Example of Simplified Integrity Checking](#).' If *fact* is to be inserted and if all variables in the occurrence of the atom *A* which unifies with *fact* are \forall -quantified and not dominated by \exists in \mathcal{IC} , then that occurrence can be replaced by *true*. Then, the usual rewrite optimizations can be applied. For example, $\neg \text{true}$ is replaced by *false*, each of $\text{true} \wedge B$, $\text{true} \vee B$, $\text{false} \wedge B$ is replaced by *B*, $\text{true} \vee B$ is replaced by *true*; $\text{false} \wedge B$ is replaced by *false*, etc. More care needs to be taken for the symmetric case. That is, if *fact* is to be deleted and all variables in *A* are quantified as above, then *A* can be replaced by *false* only if phase II has been applied and has confirmed that the deletion of *fact* is not idle, i.e., that *fact* is really *false* after the update.

VI. *Evaluate optimized constraints*

It goes without saying that, for evaluating the queries resulting from having gone through I–V, available built-in query optimization facilities should of course be exploited. It should also be advisable to upgrade them, if possible, with semantic query optimization techniques developed for deductive databases (cf., e.g., Chakravarthy, Grant & Minker, 1990; Godfrey, Gryz & Minker, 1996; Wetzel &

Toni, 1998). In particular, it should pay off to use semantic query optimization techniques which specifically apply to querying integrity constraints, as described, e.g., in Godfrey, Gryz & Zuzarte (2001).

AN SQL SYNTAX FOR INTEGRITY CONSTRAINTS

In Nicolas (1982), most of the principles outlined in '[Principles for Simplified Integrity Checking, continued](#)' are formalized in the language of first-order predicate calculus. In Decker (1987), we showed how those principles can be translated to so-called "update constraints" in *datalog*. In Decker (2001), we showed how update constraints can be translated to specific SQL `WHERE` clause conditions. In this section, we discuss how integrity constraints that are directly expressed in SQL can be simplified according to the phased approach discussed in '[Principles of Simplified Integrity Checking](#).' In this section, we define a sufficiently general SQL syntax of integrity constraints in BNF.

To say that integrity constraints can be expressed as `WHERE` clause conditions is not yet precise enough. In the language of first-order logic, integrity constraints are closed well-formed predicate calculus formulae. Since they express properties which express properties to be satisfied by each database state, they are quantified over certain attributes in specific relations. In terms of relational databases, they require or forbid the existence of certain rows, i.e., of certain column values, in specific tables and/or views, depending on conditions which depend on the current database state. Thus, it is convenient to express them as `EXISTS` or `NOT EXISTS` conditions, as they may occur in ordinary `WHERE` clauses of SQL `SELECT` statements. In fact, it can be shown that each first-order logic sentence, with predicates corresponding to tables and views, can be expressed by (possibly nested) `EXISTS` or `NOT EXISTS` conditions, as defined below. For reason of simplicity, the syntax of such conditions is somewhat less involved than in the original SQL2 standard definitions of `SELECT` statements and `WHERE` clauses. However, its expressive power is essentially as high as a full-fledged version. A careful extension to the full standard is possible, but the elaboration of all details would require disproportionate length with which we do not want to burden our presentation.

We have oriented the BNF rules below at the SQL ANSI/ISO standard of 1992 (cf. (Melton & Simon, 1993)). Instead of the usual list of column names or the `ALL` symbol in `SELECT` clauses, we use the `ANY` construct. Intentionally, it provides a chance for an intelligent query optimizer to search efficiently for just a single tuple which satisfies the subsequent `WHERE` clause condition. If there is one, the condition is satisfied, and it is not if there is none. Since no more general form of `SELECT` statements is needed, we have baptized this particular form "boolean select". We have not detailed the syntax of table and view names, which is understood. Also, we have not spent any effort in distinguishing between different types of expressions nor of operators with which numeric or character string values are connected. As usual in SQL, column names are supposed to be unambiguous references to specific attribute positions. In general, table names will have to be prefixed in front of column name identifiers in order to avoid name clashes. After all, our integrity constraint conditions may involve several tables in the

schema and also several instances of the same table. Aggregate functions comprise AVG, MAX, MIN, SUM, COUNT, as usual.

```

<integrity constraint> ::= <condition>
                           | <integrity constraint>
OR <condition>

<condition> ::= <quantified condition>
                | <condition> AND
<quantified condition>

<quantified condition> ::= [ NOT ] EXISTS <boolean select>
                           | (<integrity constraint>)

<boolean select> ::= SELECT ANY FROM <list> WHERE
                           <subcondition>

<list> ::= <relation name> { , <relation name> } *

<relation name> ::= <table name> | <view name>

<subcondition> ::= <boolean term>
                  | <subcondition> OR
<boolean term>

<boolean term> ::= <boolean factor>
                  | <boolean term> AND
<boolean factor>

<boolean factor> ::= <comparison> | (<subcondition>)
                  | [ NOT ] EXISTS <boolean
select>

<comparison> ::= <expression> { = | ? | < | = | > | = }
                           <expression>

<expression> ::= <expression> { + | - | * | / }
<expression>
                  | <prime expression>

<prime expression> ::= <constant> | <column name>
                  | <aggregate function
term> | (<expression>)

```

TRANSLATING PRINCIPLES OF SIMPLIFIED INTEGRITY CHECKING TO SQL

In this section, we walk through the six phases of simplified integrity checking yet another time. In ‘[Principles of Simplified Integrity Checking](#),’ the framework of representation was *datalog* and first-order logic, but now it is SQL, according to the syntax defined in ‘[An SQL Syntax for Integrity Constraints](#).’ The basic SELECT-FROM-WHERE structure of SQL tends to appeal to an operational understanding of searching joins

of tables, rather than to suggest a purely declarative reading. Several difficulties arising from the idiosyncrasies of SQL's syntax are going to be addressed in the discussion of points I–VI below.

Again, let \mathcal{D} be a relational database, IC be an integrity constraint in \mathcal{D} , expressed in the syntax of '[An SQL Syntax for Integrity Constraints](#),' and UPDATE fact be an update request, as in '[Principles for Simplified Integrity Checking, continued](#).' Then, going from I through VI means the following:

I. *Generate the difference between the old and the new state*

We suppose that views in \mathcal{D} , if any, are defined using join operations on tables and views, expressed using `SELECT-FROM-WHERE` syntax. In most of the contemporary commercial DBMSs, integrity constraints involving views are not expressible. However, future versions of products by several major vendors envisage the incorporation of constraints on views. Implicit updates of views caused by explicit updates of base tables essentially obey the same rules as outlined in point I of '[Principles for Simplified Integrity Checking, continued](#).' However, the identification of the polarity of occurrences of table names seems to be less transparent in SQL than in FOL. Rather than dealing at length with the intricacies of SQL syntax, we refer to related work in Gupta & Mumick (1999) on the problem of incremental view maintenance in SQL.

Another interesting thing with views is that they can be used to express denial integrity constraints. Then, integrity checking means to check if each denial representing a constraint is an empty view, in which case integrity is satisfied. That way, update propagation through views, view maintenance and integrity checking coincide and thus can be uniformly handled. However, the problem of correctly expressing a complex constraint as a denial view which recurs hierarchically or even recursively on other views and base relations must not be underestimated. A possible solution to this problem would be to express the constraint directly as a quantified `EXISTS` or `NOT EXISTS` condition, typically involving nested subconditions, deny that by prefixing or, resp., dropping `NOT`, and use the resulting query as the definition of a view. However, a simplified evaluation or materialization of such views according to phases II – VI, below, may then turn out to be quite difficult. Another solution would be, e.g., to use a natural language interface for specifying complex constraints, have these specs translated into the syntax introduced in '[First-Order Logic Representation of Integrity Constraints](#)' and then use the translations described in '[Translating Integrity Constraints to SQL Conditions](#)' and '[Identifying and Specializing Relevant Integrity Constraints](#).'

II. *Skip idle updates*

In commercial DBMSs, redundancy, i.e., multiple copies of the same row in a table, is usually permitted (unless prohibited by appropriate uniqueness constraints). Also, as in *datalog*, some facts may be defined redundantly by views, i.e., there may be several derivation paths for the same fact. In any case, an insertion is idle if the fact to be inserted is already present in the table or derivable from a view definition, and a deletion is idle if a second copy or view definition of that fact will persist after the update. (We here leave aside further complications arising from different semantics or different behavior of insert and delete statements in different RDBMSs.) For each such idle update, skipping of phases III – VI can be done, or renounced, as described in [‘Principles for Simplified Integrity Checking, continued.’](#)

A nice non-standard feature in the data manipulation part of SQL in several commercial RDBMSs is to offer constructs such as `BEFORE` and `AFTER`, with which both the old state (before the update) and the new state (after) can be queried in conjunction with an update request. Since integrity is supposed to be satisfied in the old state and an update may violate it, integrity checking needs to be done by evaluating integrity conditions on the new state (which may be simulated on the old state by taking built-in `inserted` and `deleted` relations into account, such as they exist, e.g., in the Microsoft SQL Server). However, for some update requests and constraints, an evaluation of conditions on the old state may be sufficient. In general, sufficient (but not always necessary) conditions for checking integrity have been studied, e.g., in Kobayashi (1984) and Gupta, Sagiv, Ullman & Widom (1994). In any case, care should be taken when using `BEFORE` for checking static integrity constraints, since that tends to introduce a non-declarative element. And indeed, these constructs are used best for checking *dynamic* integrity constraints.

III. *Focus on relevant integrity constraints*

For a given base table or view update in SQL, the general rule for focusing on relevant constraints is essentially the same as in [‘Principles for Simplified Integrity Checking, continued.’](#) Again, representations in *datalog* tend to be more transparent than in SQL. In any case, however, a solution for correctly identifying the polarity of occurrences of table names, which already was desirable for point I, would basically solve as well the problem of focusing on relevant constraints. Clearly, a sufficient condition for leaving a constraint unchecked is that it contains no occurrence of the name of an updated table or view at all. Otherwise, in case of doubt, it is advisable to consider a constraint as relevant and check it anyway, rather than to speculate that the polarity of an occurrence is such that it could be ignored.

IV. *Specialize relevant constraints*

Specializing relevant constraints represented in SQL, i.e. figuring out its quantification structure, may be as fishy as getting clear about polarities in SQL.

However, using an original first-order logic or *datalog* representation for applying points I–V as described in ‘[Principles of Simplified Integrity Checking](#),’ and only then translate the result to SQL for the final evaluation (point VI), avoids any hassle of having to figure out the logic of SQL. In ‘[Translating Principles of Simplified Integrity Checking to SQL](#)’ and the rest of the chapter, we give a detailed description of such a translation, which can be fully automated. Here, we just mention that cached built-in SQL tables *inserted* and *deleted* (already mentioned in ‘[An Example of Simplified Integrity Checking](#)’ and under point II, above) can be used conveniently to support phases III and IV. This is going to be described in detail in ‘[Identifying and Specializing Relevant Integrity Constraints](#).’

V. *Optimize specialized constraints*

Syntactic rewrite query optimizers remain to be a desideratum rather than a reality in commercial RDBMSs. System manuals sometimes give useful hints to write efficient queries, but in the end, they leave that to the user (and thus tacitly blame him/her for writing declarative queries which turn out to behave sub-optimally). So, rather than to prefigure the specialized forms of SQL constraints for certain update patterns and then figure out syntactic optimizations, it seems more advisable to use predicate logic for III – V, and only then translate the results to SQL, as described in ‘[Translating Static Conditions to Dynamic Triggers](#).’

VI. *Evaluate optimized constraints*

At query specification time, the built-in query optimizers of commercial DBMSs can be activated to figure out clever access plans for searching answers. Access plan optimizations typically decides about questions such as "is it preferable to search the cross-product of tables p , q by taking advantage of a user-defined index, or use to use a hashed join instead?", and "which extent of several candidate table spaces should preferably be cached so that an overhead of swapping is avoided?" However, no matter if the original specification of constraints is in predicate logic, *datalog* or SQL, access plans for constraints in their original form may not be very helpful because their specialized and rewritten form is likely to suggest completely different evaluation strategies. But, fortunately, all of phases III–V can be done already at specification time, such that also access plans for improving the efficiency of phase VI can in principle be determined already ahead of evaluation time. The same applies to semantic query optimizations, as mentioned in ‘[Principles for Simplified Integrity Checking, continued](#),’ point VI. And, for integrity constraints represented by intentionally empty views (i.e., denials, which often necessitates the use of additional auxiliary views), it should also be useful to look into Afrati, Li & Ullman (2001) for further possibilities of optimizing access plans.

FIRST-ORDER LOGIC REPRESENTATION OF INTEGRITY CONSTRAINTS

In Nicolas (1982), integrity constraints are expressed as range-restricted first-order predicate calculus sentences, i.e., closed well-formed formulae which obey the range-restricted property. This property is decidable and ensures the essential but undecidable property of domain-independence (di Paola, 1969), and thus the evaluability of queries, as shown in Nicolas (1982) and Demolombe (1992). It has been generalized, preserving evaluability, in Decker (1987, 1989, 2001) and Van Gelder & Topor (1991), in order to also cover built-in predicates corresponding to comparisons $=$, \neq , $<$, $=$, $>$, $=$. In terms of relational databases, "range-restricted" guarantees that, for each variable in a query for which answers are sought, there always exists a table column in which these values will be found. In fact, that is nothing special for SQL, because the lists of columns and tables specified in `SELECT-FROM` clauses precisely determine where to look for possible values as answers. Thus, for enabling a straightforward translation of integrity constraints into SQL conditions, the "range form" syntax defined in [‘Definition of the Range Form Syntax’](#) provides a specific column range for each variable. These ranges can then be translated easily into corresponding table and column names in SQL. Also, a general mapping of arbitrary (but range-restricted) integrity constraints to logically equivalent representations in range form can be easily automated. Such mappings have been specified in Decker (1987) and Van Gelder & Topor (1991).

Actually, there are some seemingly superficial differences in the range form definitions of Decker (1987) and [‘Definition of the Range Form Syntax’](#) below. In fact, the variant in Decker (1987) had been developed for supporting an efficient evaluation of arbitrary integrity constraints by Prolog interpreters. In turn, the range form in [‘Definition of the Range Form Syntax’](#) lends itself particularly well toward evaluation with an SQL engine. However, from a conceptual point of view, the differences are just on the "syntactic sugar" level. The syntax defined in the following section is discussed in [‘Discussion of the Range Form’](#). There, we also sketch a proof that each range-restricted integrity constraint can be expressed in this syntax.

Definition of the Range Form Syntax

The only connectors used in the range form defined below are conjunction \wedge , disjunction \vee and negation \neg , the only quantifier is \exists . That precisely corresponds to the connectors `AND`, `OR`, `NOT` and the quantifier `EXISTS` in our SQL syntax. In the BNF rules for a formula `RF` in range form, below, we distinguish built-in *system predicates* such as comparisons $=$, \neq , $<$, $=$, $>$, etc from *user-defined predicates*, which correspond in SQL to relations declared by `CREATE TABLE` statements. For compatibility with SQL, we implicitly assume that each term which occurs in the position of some argument of a predicate has an appropriate type. We denote the identity of formulae by $=$.

`RF ::= RF ?` where additional brackets can be used to establish or override
`RF | RF ? RF` precedences of connectors.

RF ::= PRF NRF	where PRF and NRF stand for <i>positive</i> and <i>negative range form</i> , respectively. An expansion of RF by PRF or NRF is called a <i>top-level range form</i> .
NRF ::= ¬PRF	
PRF ::= ? x (Range(x) ? SF)	where x is a vector of m distinct variables ($m \geq 1$); the <i>range expression</i> Range(x) is a conjunction of n positive literals ($n \geq 1$) called <i>range literals</i> , with user-defined predicates p_1, \dots, p_n . Each p_i ($1 \leq i \leq n$) is of arity = 1, the sum of their arities is m , each argument of p_i is a variable and each variable in x occurs in Range(x). For convenience, each variable in x which does not occur in the subformula SF may be represented by an anonymous symbol. For a variable x in x , each occurrence of x in SF is said to be <i>covered by</i> Range(x).
PRF ::= ? x (Range(x))	where, for x and Range(x), the same as in the preceding rule applies.
SF ::= SF ? SF SF ? SF	where additional brackets can be used to establish or override precedences of connectors.
SF ::= PRF NRF LS	where LS is a literal with a system predicate, and each variable in LS must be covered by some range expression, according to the first rule for PRF.

Discussion of the Range Form

It can be shown that the range form syntax defined above shares all essential advantages of the range form in Decker (1987) and Van Gelder & Topor (1991). In fact, its evaluation is even more efficient than the latter, since the range form above permits minimization of the scope of quantifiers, thus avoiding multiple occurrences of literals due to the distribution of ? over ? (as required in Decker (1987) and Van Gelder & Topor (1991)), or of ? over ? as required in Nicolas (1982).

Note that no negative literals with user-defined predicate occur in the range form syntax of '[Definition of the Range Form Syntax](#).' However, there is no loss of generality, since a negative literal of form $\neg p(t_1, \dots, t_k)$ can be expressed equivalently by the negative range form

where x_1, \dots, x_k are fresh variable symbols and each variable in t_1, \dots, t_k is supposed to be covered by some range expression. In general, equalities must be used, according to the syntax of '[Definition of the Range Form Syntax](#),' for expressing the unification of variables among each other or with ground (constant) terms. For example, the range form of

is

$$\neg \exists x_1, x_2 (p(x_1, x_2) \wedge x_1 = x_2 \wedge (\neg \exists y_1, y_2 (q(y_1, y_2) \wedge y_1 = x_1 \wedge y_2 = c) \vee \neg \exists z (r(z, _) \wedge z = x_1)))$$

Also note that, in each formula in range form, at least one literal with user-defined predicate must occur in its range expression. Moreover, the syntax in ‘[Definition of the Range Form Syntax](#)’ does not allow for user-defined 0-place predicates. Analogously, SQL only admits constraints on user-defined tables, not on system tables. Also, the range form does not admit formulae consisting of nothing but ground literals with system predicates and/or 0-place predicates, e.g., $\neg((\max(2 \times 3, 4) = 5) ? \text{flag})$. Clearly, such ground expressions have a constant value which does not depend on the database state. Thus, it would make no sense to consider them as conditions for expressing database integrity. Analogously, SQL does not permit the creation of tables without columns.

It is easy to verify that, by its context-sensitive requirements, the definition in ‘[Definition of the Range Form Syntax](#)’ effectively imposes a closure condition on each formula in range form, i.e., each variable in a formula in range form occurs in the scope of an \exists quantifier such that it is covered by the adjacent range expression. Related to that, it can be shown that each well-formed formula which complies with the syntax above is *allowed* (Van Gelder & Topor, 1991). And, except trivial ground boolean sub-formulae as those mentioned above, each closed formula which is range-restricted (Decker, 1987) can be equivalently represented in the range form syntax defined above. A sketch of a proof of this statement follows.

It has been shown in Decker (1987, 1989) and Van Gelder & Topor (1991) that each range-restricted formula (and thus each integrity constraint) can be equivalently represented in the range form syntax defined in Decker (1987). Moreover, each formula in that syntax can be equivalently represented in what may be called *existential range form*: The latter is obtained from the former by exhaustively replacing each quantifier \exists by $\neg \forall$ (plus additional brackets as needed to avoid ambiguities of scope) and moving the right hand side negation in $\neg \forall$ innermost (i.e., each such \neg is distributed over quantifiers and connectors to its right, and adjacent $\neg \forall$ are dropped). Next, we argue that each formula in existential range form can be represented in what may be called *normalized existential range form*. The latter is obtained from the former by replacing the argument terms of each range literal by distinguished fresh variable symbols and equality conjuncts, as sketched above. Moreover, each negative literal needs to be replaced by an equivalent negative range form, as described above. Then, it can be easily shown that the set of all such formulae in normalized existential range form is a subset of the set of formulae in range form. Thus, except trivial ground boolean sub-formulae, each range-restricted formula can be represented by a logically equivalent formula in range form, i.e., the syntax in ‘[Definition of the Range Form Syntax](#)’ practically incurs no loss of generality. Along the lines of Decker (1987, 1989) and Van Gelder & Topor (1991), it

can be argued that, without a sophisticated syntactic device such as the range form, the evaluation of many integrity constraints in *datalog* would be much more complicated or even impossible. The following example illustrates this case.

Example

The integrity constraint

expresses that there must be an individual x who is superior of all employees in the sales department. IC is not "safe" in the usual sense, but it is range-restricted and hence domain-independent. While a query which would ask for all values of x that satisfy IC is not domain-independent, IC always returns a boolean truth value (rather than a possibly infinite relation) and thus can always be evaluated safely. A representation of IC in the range form of (Decker, 1987) is

$$\exists x (\text{sup}(x, _) \wedge \forall y (\text{empl}(y, \text{sales}) \rightarrow \text{sup}(x, y))) \vee \neg \exists y (\text{empl}(y, \text{sales})).$$

A representation of IC in the range form of [‘Definition of the Range Form Syntax’](#) is

$$\begin{aligned} &\exists x (\text{sup}(x, _) \wedge \neg \exists y_1, y_2 (\text{empl}(y_1, y_2) \wedge y_2 = \text{sales} \wedge \neg \exists x_1, x_2 (\text{sup}(x_1, x_2) \wedge x_1 \\ &= x \wedge x_2 = y_1)) \vee \neg \exists y (\text{empl}(_, y) \wedge y = \text{sales})). \end{aligned}$$

Maybe, this example is a bit contrived, because normally, database designers might prefer to simply state an SQL equivalent of the first of the two disjuncts above, i.e.,

for defining the intended constraint. However, leaving away the second disjunct

would mean to ignore the border case that both the extents of sup and empl are empty. In that case, IC would still be satisfied, while IC' would be violated.

TRANSLATING INTEGRITY CONSTRAINTS TO SQL CONDITIONS

In this section, we specify an easily automated translation of constraints in range form into equivalent SQL conditions. To begin with, we are going to show the result of the

translation of the integrity constraint IC in the previous example, as an example which conveys a taste of things to come. We assume that relations $empl$ and sup have been defined by appropriate `CREATE TABLE` statements. For convenience, let the argument in the i -th column of a relation rel be denoted by $rel:i$, from now on. It is easy to see that the SQL version of IC , below, reflects very closely the structure of IC in range form, above.

```

EXISTS (SELECT ANY FROM sup s1 WHERE NOT EXISTS
        (SELECT ANY FROM empl WHERE empl:2 = sales AND NOT
         EXISTS
           (SELECT ANY FROM sup s2 WHERE s2:1 = s1:1 AND
            s2:2 = empl:1)))
OR NOT EXISTS (SELECT ANY FROM empl WHERE empl:2 = sales)

```

In general, multiple occurrences of relation names in SQL statements need to be kept apart by postfixed alias names, as usual in SQL. In the example above, two occurrences of sup are distinguished by aliases $s1$ and $s2$. For convenience, we may loosely speak, from now on, of a "relation" or "predicate", say, p in some SQL statement when we really mean to identify with p the alias name of a particular occurrence of the relation corresponding to p .

The BNF rules for translating a formula in range form into an equivalent SQL condition below recur on the grammar defined in '[Definition of the Range Form Syntax](#).' For convenience, multiple occurrences of grammar variables RF and SF in the same rule are identified by different subscripts, to keep them apart.

BNF rule for $sql(F)$ is the result of the translation of F to SQL formula F

$RF ::= RF_1 ? \quad sql(RF) = sql(RF_1) \text{ AND } sql(RF_2)$
 RF_2

$RF ::= RF_1 ? \quad sql(RF) = sql(RF_1) \text{ OR } sql(RF_2)$
 RF_2

$RF ::= PRF \mid \quad sql(RF) = sql(PRF); \quad sql(RF) = sql(NRF)$
 NRF

$NRF ::= \neg PRF \quad sql(NRF) = \text{NOT } sql(PRF)$

$PRF ::= ? \quad x \quad sql(PRF) =$
 $(\text{Range}(x) ? \quad \text{EXISTS } (SELECT ANY FROM p_1, \dots, p_n \text{ WHERE } sql(SF))$
 $SF) \quad \text{where } p_1, \dots, p_n \text{ are the relation names corresponding to the predicates of}$
 $\text{the } n \text{ literals in } \text{Range}(x). \text{ Multiple occurrences of } p_i \text{ (} 1 \leq i \leq n \text{) in range}$
 $\text{expressions of } PRF \text{ (including nested ones in } SF \text{) need to be consistently}$
 $\text{postfixed with distinguished alias names in } sql(RF) \text{ (which, for}$
 $\text{simplicity, is not denoted explicitly here).}$

$PRF ::= ? \quad x \quad sql(PRF) = \text{EXISTS } (SELECT ANY FROM p_1, \dots, p_n)$
 $(\text{Range}(x)) \quad \text{where, for } x \text{ and } \text{Range}(x), \text{ the same as in the pre-ceding rule applies.}$

$SF ::= SF_1 ? \quad sql(SF) = sql(SF_1) \text{ AND } sql(SF_2)$

BNF rule for $sql(\mathcal{F})$ is the result of the translation of \mathcal{F} to SQL formula \mathcal{F}

SF_2

$SF ::= SF_1 \text{ ? } sql(SF) = sql(SF_1) \text{ OR } sql(SF_2)$

SF_2

$SF ::= PRF \mid sql(SF) = sql(PRF); sql(SF) = sql(NRF)$

NRF

$SF ::= LS \quad sql(SF) = sql(LS)$

where $sql(LS)$ is defined as follows: each variable x in LS is covered and thus uniquely occurs in the, say, i -th position of a literal with user-defined predicate, say, p in the range expression which covers x . So, $sql(LS)$ is obtained by replacing each occurrence of x in LS with $p:i$.

IDENTIFYING AND SPECIALIZING RELEVANT INTEGRITY CONSTRAINTS

Up to this point, it should be fairly obvious that the evaluation of an integrity constraint IC as a first-order logic query precisely corresponds to evaluating the SQL query $sql(IC)$. However, any potential for improving the efficiency of evaluation according to phases I - VI discussed in '[Principles of Simplified Integrity Checking](#)' and '[Translating Principles of Simplified Integrity Checking to SQL](#)' still remains to be exploited.

Precisely that is the purpose of this section, in which we describe how to translate an SQL condition obtained as specified in '[Translating Integrity Constraints to SQL Conditions](#)' into a set of equivalent SQL triggers, the firing of which is more efficient than the evaluation of the original condition. Equivalence here is meant in the following sense. Let us assume an update request of a database state which satisfies a given constraint.

- a. If the update does not violate the constraint, then the execution of the update is not prohibited by the triggers obtained from the constraint.
- b. If the update would violate the constraint, then at least one of the triggers will be fired upon an attempt to execute the update, will detect violation and will take appropriate action (e.g., a roll-back).

Efficiency of firing triggers and evaluating SQL conditions can reasonably be measured in terms of the number of facts retrieved from stored relations, or as well of the number of times that stored relations have to be accessed. We assume that all of the current state is stored and that the facts to be `inserted` or `deleted` are accessible in cached system tables with relation names `inserted` and `deleted`, respectively, as usual in commercial DBMSs.

In '[Identifying Relevant Constraints](#),' we outline how to represent constraints identified according to phase III in a form which eventually will enable SQL to focus on relevant constraints. In '[Specializing Relevant Constraints](#),' we describe how to specialize the formulae obtained in '[Identifying Relevant Constraints](#),' according to phase IV. In order to have sufficient syntactic flexibility, we allow any first-order logic representation of integrity constraints at this stage, i.e., we do not require a representation in range form, at this point (although we do not rule it out either). Phases V and VI are going to be catered for in '[Translating Integrity Constraints to Optimized SQL Triggers](#).' For limiting the length of this chapter, we do not work out the issue of how also phase II can be incorporated into SQL (e.g., with `BEFORE-AFTER` constructs, as mentioned in point II of that section, or with a translation of related techniques described in Bry, Decker & Manthey (1988) to SQL).

Identifying Relevant Constraints

In this subsection, we elaborate on phase III (cf. '[Principles of Simplified Integrity Checking](#)' and '[Translating Principles of Simplified Integrity Checking to SQL](#)'). We show, for a given update, how to partition the set of constraints imposed on a database into a (typically large) subset of constraints which can be ignored, and a (typically small) set of constraints which could be violated by the update and thus have to be checked. The latter kind of constraints are called *relevant*, with regard to the update. Since the identification of relevant constraints is based on purely syntactic criteria (cf. '[Principles of Simplified Integrity Checking](#)'), it can be accomplished already at constraint specification time, for update patterns corresponding to predicate symbols in the underlying language, i.e., to table names in the corresponding DBMS.

As already indicated in '[Principles for Simplified Integrity Checking, continued](#),' an integrity constraint IC is defined to be relevant for the insertion (resp., deletion) of fact if there is an atom A with negative (resp., positive) polarity in IC which unifies with fact . Otherwise, IC is not relevant and can be ignored. We recall that there need to be as many checks of specialized forms of IC as there are occurrences of facts matching the update argument in IC with the respective polarity. For example, the constraint $\text{IC} = ? \ x \sim p(x, b) \ ? \ \sim p(a, x)$ is relevant for insertions of facts about p , but not for any deletions.

Similar to a formalism called "update constraint" in *soundcheck* (Decker, 1987), we are going to incorporate the principle of relevance into the constraints to be checked upon a given update pattern. Similar to what is common in the standard SQL extensions of several DBMS vendors, we assume the existence of two distinguished predicates *inserted* and *deleted* (cf. '[An Example of Simplified Integrity Checking](#)' and section '[Translating Principles of Simplified Integrity Checking to SQL](#),' point IV). They are cached at update time until commit time and are not accessible to the user. For an update which requests the insertion or the deletion of some fact $p(c_1, \dots, c_k)$ where c_1, \dots, c_k are constants, querying *inserted* or, resp., *deleted* returns the answer c_1, \dots, c_k . (As usual in SQL DBMSs which feature *inserted* and *deleted* as built-in tables, the arity of these relations adapts to the arity of the update relation proper.) With that, it is possible to incorporate the identification of relevancy into constraints, as follows.

Let IC be an integrity constraint, p a user-defined (i.e., updatable) predicate with arity $k=1$ and $p(t_1, \dots, t_k)$ an occurrence of an atom in IC , where each term t_i ($1 \leq i \leq k$) is either a constant or a variable. For convenience, let *updated* stand for *inserted* if the polarity of p is negative, and for *deleted* if it is positive. Further, for some h , $0 < h \leq k$, let x_1, \dots, x_h be the variables among the t_i . Then, one of the two formulae (*) below identifies the relevance of IC with regard to requests for inserting or, resp., deleting facts which unify with $p(t_1, \dots, t_k)$.

If $h = 0$ (i.e., there are no variables among t_1, \dots, t_k), then

else

Notice the negation of IC in (*). Negating constraints corresponds to the good practice of representing integrity constraints in denial form, as introduced in Sadri & Kowalski (1988). Denial form is convenient for declaring what should *not* be the case, i.e., for stating conditions that should not hold. If such a condition becomes true in a database, it means that integrity is violated.

An integrity constraint IC is logically equivalent to its denial form $\neg \text{IC}$. Thus, the formula $\neg \text{IC}$ in (*) is the condition which, when satisfied, signals integrity violation. So, if (*) returns *true* (or, as in many systems, *yes*) upon updating a ground instance of $p(t_1, \dots, t_k)$, then integrity is violated. Conversely, an evaluation of (*) in case IC is not relevant for a given update would immediately return false (resp., no) because of the cached conjunct on the left-hand side of (*), without evaluating IC .

For example, the following two instances of the second formula (*) are obtained for the integrity constraint $\text{IC} = ? x \sim p(x, b) ? \sim p(a, x)$.

This example also illustrates that update constraints of form (*) act as a relevance filter, not just on the level of relation names, but also on the level of attribute values. For instance, no integrity checking for IC is needed for insertions of facts about p that neither match $p(x, b)$ nor $p(a, x)$, e.g., $p(c, c)$. Examples for update constraints for deletions are given in [‘Translating Integrity Constraints to Optimized SQL Triggers.’](#)

Specializing Relevant Constraints

In this section, we show how formulae of form (*) above can be specialized, in the sense of phase IV (cf. sections '[Principles of Simplified Integrity Checking](#),' '[Translating Principles of Simplified Integrity Checking to SQL](#)').

Again, let IC be an integrity constraint, p an updatable predicate with arity $k = 1$ and F a fact about p to be inserted or deleted such that IC is relevant for this update request. That is, F matches with some occurrence of an atom $p(t_1, \dots, t_k)$ in IC with negative or, resp., positive polarity. Again, let *updated* stand for *inserted* if the polarity is negative, and for *deleted* if it is positive. Further, let f be an mgu of F and $p(t_1, \dots, t_k)$. According to Nicolas (1982), IC can then be specialized to $\text{IC}?$, where the substitution $?$ is obtained from f by restricting the latter to those variables in $p(t_1, \dots, t_k)$ that are $?$ -quantified in the negation-innermost form of IC without being dominated by an $?$. Thus, $?$ grounds each such variable to the corresponding constant value in F .

Now, we are prepared to translate this principle of specialization to the conditions of form (*) in '[Identifying Relevant Constraints](#).' For convenience, let us designate a variable in IC as $\underline{?}$ -quantified when it is $?$ -quantified and not dominated by an $?$ in its negation-innermost form.

Again, for some h , $0 < h = k$, let x_1, \dots, x_h be the variables among the t_1, \dots, t_k . Without loss of generality, let, for some g , $0 = g = h$, x_1, \dots, x_g be the $\underline{?}$ -quantified variables, and x_{g+1}, \dots, x_h be the remaining variables in $p(t_1, \dots, t_k)$, if any. Further, let IC denote the formula obtained by dropping the quantifiers of each x_j ($1 = j = g$) in IC . (Recall that, in general, a variable x which is $\underline{?}$ -quantified in IC may be quantified by either one of $?$ and $?$ in IC , since the latter is not required to be in negation-innermost form.) Then, instead of costly conditions of form (*), it suffices to evaluate one of the following specialized conditions).

If $h = 0$ (i.e., there are no variables among t_1, \dots, t_k), then

$$(**) \text{ updated}(t_1, \dots, t_k) \wedge \neg \text{IC}$$

else

Clearly, the first case is the same as (*), since there are no variables which could be specialized. But if $g > 0$, i.e., if there are $\underline{?}$ -quantified variables, then an instantiation of the variables in *updated* with ground values of a fact to be inserted or, resp., deleted, also grounds each $\underline{?}$ -quantified variable x_1, \dots, x_g in IC .

For convenience, let us call formulae of form (**) "update constraints." More precisely, let IC be an integrity constraint, p a user-defined predicate and A an atom in IC with

predicate p . Then, for each occurrence, say, $p(t_1, \dots, t_k)$ of A in IC , precisely one of the formulae (***) is obtained as described above, and that formula is called *the update constraint of IC for $p(t_1, \dots, t_k)$* .

For the sample constraint $IC = ? x \sim p(x, b) ? \sim p(a, x)$ in ‘[Identifying Relevant Constraints](#),’ the following two specialized update constraints are obtained.

In general, further optimizations of simplified update constraint formulae are possible. But we leave that to the following section, where phase V (cf. ‘[Principles of Simplified Integrity Checking](#),’ ‘[Translating Principles of Simplified Integrity Checking to SQL](#)’) is going to be addressed again.

TRANSLATING INTEGRITY CONSTRAINTS TO OPTIMIZED SQL TRIGGERS

In this section, we describe how the specialized relevant constraints of form (***) in ‘[Specializing Relevant Constraints](#)’ are translated into SQL triggers. In ‘[Translating Static Conditions to Dynamic Triggers](#),’ we specify how to translate static SQL conditions (as obtained in ‘[Translating Integrity Constraints to SQL Conditions](#)’ from integrity constraints represented as first-order logic formulae) into dynamic SQL triggers. In ‘[Examples](#),’ we illustrate the results obtained so far by some examples, and indicate some possibilities of optimization.

Translating Static Conditions to Dynamic Triggers

For convenience, let us assume that SQL triggers are of the following form (which essentially is a common denominator of the usual appearance of triggers in commercial SQL database systems):

```
CREATE TRIGGER ON <relation> FOR {INSERT | DELETE}:
  IF <condition> <action>
```

where $\langle relation \rangle$ names the table which is updated by an insertion or deletion, resp.; $\langle condition \rangle$ is an SQL condition which, when its evaluation in the updated state returns true, signifies violation of integrity; $\langle action \rangle$ is a statement which, in practice, usually is a ROLLBACK command for reinstalling the database state as it has been before the update attempt, and the output of a warning text or a reject message. In principle, it may also involve an explanation for the update failure, or even a repair action. However, we are not concerned in this chapter about what a DBMS does when an update would lead to

integrity violation. In particular, we would make little sense to take into account any potential access to stored relations caused by $\langle action \rangle$, for measuring the efficiency of the trigger, because the very same action would be taken if integrity violation is detected by evaluating the original SQL condition, instead of associated triggers.

According to ‘[Specializing Relevant Constraints](#),’ it suffices to evaluate update constraints of form (**), for integrity checking upon the insertion or deletion of some fact which matches the occurrence of some atom in some integrity constraint. Thus, it suffices to have a suitable SQL representation of update constraints as conditions of SQL triggers, which are fired upon such updates. In ‘[Translating Integrity Constraints to SQL Conditions](#),’ we have described how to translate arbitrary (but range-restricted) conditions, represented as first-order predicate calculus sentences in range form, into equivalent SQL conditions. However, since we have not required range form syntax in ‘[Identifying and Specializing Relevant Integrity Constraints](#),’ the update constraints of form (**) must first be transformed into range form.

So, let us suppose that rf is a mapping which transforms a first-order predicate calculus sentence into a representation in the range form syntax of ‘[Definition of the Range Form Syntax](#).’ Further, for an integrity constraint IC and the occurrence A of an atom with user-defined predicate, say, p in IC , let $up(IC, A)$ denote the update constraint (**) obtained as described in ‘[Identifying and Specializing Relevant Integrity Constraints](#).’ Then, according to what we have seen in ‘[Specializing Relevant Constraints](#),’ triggers of the following form (one for each occurrence A of an updatable atom in IC) are sufficient for a sound integrity check of IC .

```
(***) CREATE TRIGGER ON p FOR {INSERT | DELETE}:
      IF sql( rf( up( IC, A ))) ROLLBACK
```

For convenience, we call $sql(rf(up(IC, A)))$ the *body* of update triggers of form (***). In terms of integrity checking, such triggers are equivalent to the original integrity constraint, but more efficient than the evaluation of the latter. Equivalence here is meant in the sense of the definition at the beginning of ‘[Translating Integrity Constraints to SQL Conditions](#).’ A formal proof of equivalence would essentially rely on the equivalence preservation of translations rf and sql .

From what we have seen already in ‘[Principles of Simplified Integrity Checking](#),’ it should be obvious that the firing of these triggers, which is controlled by update attempts of facts which match A , is more efficient than evaluating each constraint for each update, or even only each relevant constraint in its original, non-specialized form.

At this point, it should be interesting to recall that, in ‘[Identifying Relevant Constraints and Specializing Relevant Constraints](#),’ we have not required that integrity constraints be

represented in range form or in SQL. In fact, the main reason for that was to be able to apply specialization as described in ‘[Specializing Relevant Constraints](#)’ to update constraints of form (*) (‘Identifying Relevant Constraints and Specializing Relevant Constraints’) and a transformation into range form *before* the translation to SQL (‘[Translating Integrity Constraints to SQL Conditions](#)’) is done, because experience has shown that it is much easier (or, at least, less messy) to apply syntactic transformations and optimizations on formulae in a pure logic syntax than to do so on SQL expressions. In principle, operations *sql*, *rf*, *up* in the body of update triggers could be applied in any sequence, but the one proposed in (***) has turned out to be the most convenient one.

In this context, another interesting issue is that syntactic transformations *sql* and *rf* do not just preserve semantic equivalence, but also the validity of the method itself. In fact, that may not be self-evident, since, e.g., the number of occurrences of atoms in an integrity constraint formula may vary under some transformations, and their polarity is not invariant under arbitrary equivalence transformations. However, it can be shown that our transformations can do no harm to the validity of any of the six phases addressed throughout the chapter. Also, syntactic rewrites used in further syntactic optimizations for implementing phase V, as discussed in Nicolas (1982), Van Gelder & Topor (1991) and Demolombe & Illarramendi (1989), can be shown to preserve the required properties.

Examples

In this section, we feature some examples of triggers obtained by applying the method outlined in this chapter. For triggers of form (***) above, a suitable SQL query optimizer may recognize plenty of possibilities for optimizing them, such that their evaluation becomes even more efficient, according to phase V of simplified integrity checking (cf. ‘[Principles of Simplified Integrity Checking](#),’ ‘[Translating Principles of Simplified Integrity Checking to SQL](#)’). In general, optimization can take place already at an early stage, e.g., already when update constraints of form (**) (‘[Specializing Relevant Constraints](#)’) are obtained, or in fact at any point between the initial constraint specification time and anywhere in phases I through V (recall that all of I - V can be done ahead of update time). However, at any time before phase V, an optimizer for first-order predicate calculus sentences, such as the one described in Demolombe & Illarramendi (1989), should be used, rather than an SQL optimizer.

Examples 1 – 4 are deliberately chosen because of the multiple occurrences of the same relation and because of their mutual similarities. The subtle syntactic differences contrast with considerably large semantic differences, which become more apparent in the triggers. Because of the semantic intricacies, hand-crafting these triggers, even by experts, has continuously proven to be pretty cumbersome and exceedingly error-prone. This experience, of course, confirms the advantage of having a method for deriving triggers automatically from the original integrity constraints.

Example 1

The integrity constraint

translates into the following two triggers.

```
CREATE TRIGGER ON p FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE inserted:2 = b AND
              EXISTS (SELECT ANY FROM p WHERE p:1 = a))
ROLLBACK

CREATE TRIGGER ON p FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE inserted:1 = a AND
              EXISTS (SELECT ANY FROM p WHERE p:2 = b))
ROLLBACK.
```

Example 2

The integrity constraint

has already been used as an example in ‘[Identifying Relevant Constraints](#)’ and ‘[Specializing Relevant Constraints](#).’ It translates into the two triggers

```
CREATE TRIGGER ON p FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE inserted:2 = b AND
              EXISTS (SELECT ANY FROM p WHERE p:1 = a AND p:2 =
                      inserted:1))
ROLLBACK

CREATE TRIGGER ON p FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE inserted:1 = a AND
              EXISTS (SELECT ANY FROM p WHERE p:1 = inserted:2
                      AND p:2 = b))
ROLLBACK.
```

By analogy to a similar example in (Nicolas, 1982), an optimization of the bodies of the two triggers above is possible. For a request to insert the fact $p(a, b)$, only one of the two update constraints needs to be checked, i.e., only one of the two triggers needs to be fired, the other can remain passive. Detecting this or similar possibilities, however, is in general only reasonable at update time, and might be hardly less costly than doing an unnecessary check. In general, there is a trade-off between the effort invested in optimization at update time and the returned gains in efficiency.

Example 3

The translation of the constraint

results in two triggers which turn out to be equivalent variants of each other. Thus, we obtain the following single optimized trigger.

```
CREATE TRIGGER ON p FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE
    EXISTS (SELECT ANY FROM p WHERE p:1 = inserted:2
      AND p:2 = inserted:1))
  ROLLBACK.
```

Example 4

The integrity constraint

expresses a referential relationship between the first and the second columns of *p*. However, notice that this constraint is not expressible with standard SQL key constructs. Indeed, a FOREIGN KEY constraint on *p*:2 which would reference *p*:1 would require that an additional uniqueness constraint be imposed on the referenced column. The translation of this constraint requires alias names *p*1 and *p*2 for the two occurrences of *p*. It yields the following two triggers.

```
CREATE TRIGGER ON p FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE
    NOT EXISTS (SELECT ANY FROM p WHERE p:1 = inserted:2 ))
  ROLLBACK

CREATE TRIGGER ON p FOR DELETE:
  IF EXISTS (SELECT ANY FROM deleted WHERE
    EXISTS (SELECT ANY FROM p p1 WHERE p1:2 = deleted:1 AND
      NOT EXISTS (SELECT ANY FROM p p2 WHERE p2:1 =
        deleted:1)))
  ROLLBACK.
```

So far, we have dealt only with insertions and deletions of facts; an update is treated as a transaction consisting of a deletion, followed by an insertion. Moreover, the identification of relevant constraints, as described in [‘Principles of Simplified Integrity Checking,’](#) [‘Translating Principles of Simplified Integrity Checking to SQL’](#) and [‘Identifying Relevant Constraints,’](#) focuses on the occurrence of relation names, but is not as fine-grained as to take a selective updating of particular columns into account. However, there is a natural way to automatically incorporate such a fine-tuning into triggers. For instance, let *D* be a database which satisfies *IC* above, containing, say, the facts *p*(*a*, *b*), *p*(*b*, *a*), *p*(*c*, *c*). Further, consider an update for modifying *p*(*a*, *b*) to *p*(*d*, *b*). Processing this

update request as a transaction $\{delete p(a, b), insert p(d, b)\}$ according to the approach outlined so far will identify IC as relevant with regard to $insert p(d, b)$, while IC is not relevant and thus ignored with regard to $delete p(a, b)$. Consequently, the simplified constraint $IC' = ? z (p(b, z))$ then has to be evaluated. It will succeed and thus indicate that integrity remains satisfied. However, it is easy to see that no constraint needs to be checked at all as long as the second attribute of p is not updated and no existing value of the first attribute of p is deleted. *Datalog* does not seem flexible enough to easily accommodate such a fine degree of relevance, but an appropriate extension of SQL such as the following seems to be natural. In general, triggers of the following form are conceivable in SQL.

```
CREATE TRIGGER ON <relation> FOR UPDATE{:<position>}+:
IF <condition> <action>
```

where $\{:<position>\}^+$ is a list of natural numbers, separated by :, which indicate the positions of the attribute in $<relation>$ to be updated. Thus, the specification of the position of updated attributes acts as an additional fine-grained relevance filter. Analogous to (***) in ‘[Specializing Relevant Constraints](#),’ such triggers could then serve to accommodate automatically generated triggers of the form

```
CREATE TRIGGER ON p FOR UPDATE{:<position>}+:
IF sql( rf( up(IC, A))) ROLLBACK
```

where, as in (***), $up(IC, A)$ stands for *inserted* or, resp., *deleted*, for negative or, resp., positive polarity of A in IC . In our example, the following additional update trigger is then generated:

```
CREATE TRIGGER ON p FOR UPDATE:2 :
IF EXISTS (SELECT ANY FROM inserted WHERE NOT EXISTS
            (SELECT ANY FROM q WHERE inserted:2 = q:1))
ROLLBACK.
```

Example 5

The integrity constraint

has already been discussed in the example in ‘[First-Order Logic Representation of Integrity Constraints](#).’ According to ‘[Identifying Relevant Constraints](#),’ it translates into two triggers, one for *sup* and one for *empl*. The optimized trigger for *sup* is

```
CREATE TRIGGER ON sup FOR DELETE:
IF NOT C AND EXISTS (SELECT ANY FROM empl WHERE
                    empl:2 = sales)
ROLLBACK
```

where c stands for the condition

```
EXISTS (SELECT ANY FROM sup s1 WHERE NOT EXISTS
  (SELECT ANY FROM empl WHERE empl:2 = sales
  AND NOT EXISTS
    (SELECT ANY FROM sup s2 WHERE s2:1 = s1:1 AND s2:2 = empl:1)))
```

Notice that c corresponds to the first of the two disjuncts of the representation of IC in range form (cf. the example in ‘[First-Order Logic Representation of Integrity Constraints](#)’). Because of the denial of IC as effected by the transformation (*) (cf. ‘[Identifying Relevant Constraints](#)’), the disjuncts turn into conjuncts, the second of which corresponds to the expression `EXISTS (SELECT ANY FROM empl WHERE empl:2 = sales)` above. Also notice that the body of the `DELETE` trigger above would have to be prefixed with the conjunct `EXISTS SELECT ANY FROM deleted`, according to (**) and (***). However, because of the quantification structure of IC (with a dominant $?$), that conjunct would contribute nothing in terms of specializing the rest of the body. Thus, it can be dropped, similar to the example for phase V in ‘[An Example of Simplified Integrity Checking](#).’

The optimized trigger for `empl` is

```
CREATE TRIGGER ON empl FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE inserted:2 = sales
  AND NOT C)
  ROLLBACK
```

where c is again the condition above.

As opposed to the optimization of the `DELETE` trigger mentioned above, the analogous conjunct `SELECT ANY FROM inserted WHERE inserted:2 = sales` in the body of the `INSERT` trigger is not dropped, since the `WHERE` clause acts as a relevance filter which prevents the rest of the body to be evaluated in case the second attribute of the inserted `empl` tuple is not sales.

Notice that, according to (***), the full form of the `INSERT` trigger above is

```
CREATE TRIGGER ON empl FOR INSERT:
  IF EXISTS (SELECT ANY FROM inserted WHERE inserted:2 = sales
  AND NOT C AND EXISTS (SELECT ANY FROM empl WHERE
  empl:2 = sales))
  ROLLBACK.
```


As in the `DELETE` trigger above, the additional conjunct corresponds to the second disjunct in the range form of `IC`. It is dropped in the optimized version above because the only kind of update for which this trigger fires is insert requests for `empl` where `empl:2 = sales`. So, that conjunct is known in advance to always be satisfied, and hence it is dispensable.

Towards the end of ‘[Translating Static Conditions to Dynamic Triggers](#),’ we mentioned that operations `up` and `rf` should be applied before applying `sql`, i.e., before mapping predicate logic syntax into SQL. Example 5 illustrates nicely that it is also advisable to determine all expressions `up(IC, A)`, one for each occurrence `A` of some atom with user-defined predicate in `IC`, *before* application of `rf`. That is because the range form tends to increase the number of occurrences of atoms. To see an example of this, suppose `rf` was applied before `up`, for `IC` above. Then, we would obtain a total of four update constraints and hence four triggers, two for `empl` and two for `sup`, because each of the two relation names occurs twice in `rf(IC)`, as we have seen in the example in ‘[First-Order Logic Representation of Integrity Constraints](#).’ Not surprisingly, each of those triggers for `empl` turns out to be equivalent to the one obtained above. But trying to find out about the equivalence of the triggers in phase V would be less than straightforward, for this example, and undecidable in general. So, it is better to apply `up` to a representation with a presumably minimal length, as indicated already in point III of ‘[Principles for Simplified Integrity Checking, continued](#),’ rather than to apply `rf` before `up`.

CONCLUSION

We have described how to translate the *soundcheck* approach for integrity checking in deductive databases to SQL-based databases. Our results provide the grounds for an implementation of this methodology in commercial DBMSs. In particular, we have specified a declarative syntax for expressing arbitrarily quantified integrity constraints, which lends itself well toward an efficient evaluation by SQL engines. Also, we have translated to SQL the approach originally described in Nicolas (1982) to simplify integrity checking. Essentially, it focuses attention on constraints that are relevant for a given update, and specializes them to update values. To our knowledge, no such translations have yet been implemented in SQL databases. Rather, most DBMSs on the marketplace do not support declarative specifications of arbitrary constraints, but require hand-crafting of procedural triggers or stored procedures. Proposals such as Cochrane, Pirahesh & Mattos (1996) which by now have found their way into standard relational databases, amount, in one way or another, to a combination of disparate declarative and procedural mechanisms. As opposed to that, our approach is uniformly declarative. It does not sacrifice the advantages of efficiency that otherwise may only be obtained less systematically and less reliably, by compromising declarativity at an early stage. Evaluation of optimized triggers according to our approach is much less expensive than evaluating unsimplified SQL conditions, in terms of the costs of accessing stored facts.

In continuation of the ideas presented in this chapter, we intend to transpose the SLIC approach (Decker & Celma, 1994) for efficient integrity checking of constraints on views to SQL. An integrity constraint then would itself be represented as a denial view, which

materializes to the empty relation if and only if the constraint is not violated. Representing an arbitrary first-order predicate logic sentence IC as a denial view is achieved by applying the transformation in Lloyd & Topor (1984) to the generalized clause $? \text{rf}(\neg \text{IC})$, where $\text{rf}(\neg \text{IC})$ is the negation of IC represented in range form (cf. Decker, Teniente & Urpí, 1996). In general, that transformation results in several additional view definitions on which the denial view recurs. (Applying that transformation directly to $? \neg \text{IC}$ typically yields floundering queries (cf. Decker, 1989). We also envisage a transposition of the approach described in Bry, Decker & Manthey (1988) for checking the mutual consistency of integrity constraints to SQL. Moreover, we intend to translate techniques for integrity-preserving view updating in logic databases (e.g., Decker, 1990, 1996) to SQL, along the lines of this chapter and their continuation in terms of SLIC.

In general, much more of the theoretical state of the art than what we have just mentioned can be brought to bear on practical implementations of relational databases. In fact, many other sources can be tapped. For instance, further optimizations of simplified triggers in terms of conjunctive query optimization and semantic query optimization should be investigated. Potential synergies of data mining, rule discovery, semantic query optimization and `CHECK` constraints are outlined in Gryz, Schiefer, Zheng & Zuzarte (2001). Also, translating partial evaluation techniques for specializing integrity constraints in Leuschel & De Schreye (1998) to SQL would be a tempting challenge.

ACKNOWLEDGMENTS

This work originated in a project at the Database Competence Center at Siemens Corporate Research (Decker, 1997). Large portions of its contents were elaborated at the Institute of Computer Science of the University of Munich (Germany), where the author has been hosted as a guest scientist at the teaching and research unit "Programming and Modelling Languages". Earlier versions of this chapter have been circulating since 1999. I should like to acknowledge the many bits and pieces of feedback received along the way to its present form. In particular, I'd like to thank Irina Kogan from York University, Toronto, for her very careful proof reading and useful suggestions.

REFERENCES

- Abiteboul, S., R. Hull, V. Vianu (1995). *Foundations of Databases*. Addison-Wesley.
- Afrati, F., C. Li, J. Ullman (2001). *Generating Efficient Plans for Queries Using Views*. *Proc. SIGMOD '01*. On-line, available at: <http://www.acm.org/sigs/sigmod/sigmod01/e-proceedings/>.
- Amazon: Books Search site, 2001. On-line, available at: <http://www.amazon.com>.
- Armstrong, W. (1974). *Dependency Structures of Data Base Relationships*. *Proc. IFIP '74*, 580–583. North-Holland.
- Bernstein, P., B. Blaustein (1982). *Fast Methods for Testing Quantified Relational Calculus Assertions*. *Proc. SIGMOD '82*, 39–50. ACM Press,.

- Bernstein, P., B. Blaustein, & E. Clarke (1980). *Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data*. *Proc. 6thVLDB*, 126–136. IEEE-CS.
- Bocca, J. (1986). *EDUCE: A Marriage of Convenience: Prolog and a Relational DBMS*. *Proc. Symp. Logic Programming*, 36–45. IEEE-CS.
- Bocca, J., M., Dahmen & M. Freeston (1992). *MegaLog: A Platform for Developing Knowledge Base Management Systems*. *Proc. 3rdLPAR*, 457–459. Springer LNCS, Vol. 624.
- Bocca, J., H. Decker, J.-M. Nicolas, L. Vieille, M. Wallace (1986). *Some Steps towards a DBMS-based KBMS*. *Proc. 10thIFIP*, 1061–1067. North-Holland, 36–45.
- Bry, F., H. Decker, & R. Manthey (1988). *A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases*. *Proc. EDBT '88*, 488–505. Springer LNCS, Vol. 303.
- Celma, M. & H. Decker (1994). *Integrity Checking in Deductive Databases - The Ultimate Method?* *Proc. 5th Australasian Database Conference*, 136–146. World Scientific.
- Celma, M., C. Garcia, L. Mota, H. Decker (1994). *Comparing and Synthesizing Integrity Checking Methods for Deductive Databases*. *Proc. ICDE '94*, 214–222, IEEE CS.
- Ceri, S., R. Cochrane, & J. Widom (2000). *Practical Applications of Triggers and Constraints: Successes and Lingering Issues*. *Proc. 26thVLDB*, 254–262. Morgan Kaufmann. On-line, available at: <http://www.acm.org/sigmod/vldb/conf/2000/P254.pdf>.
- Ceri, S., G. Gottlob, & L. Tanca (1989). *Logic Programming and Databases*. Springer.
- Chakravarthy, U., J. Grant, & J. Minker (1990). *Logic-based Approach to Semantic Query Optimization*, *ACM Transactions on Database Systems* 15(2): 162–207.
- Cochrane, R., H. Pirahesh, & N. Mattos (1996). *Integrating Triggers and Declarative Constraints in SQL Database Systems*. *Proc. 22ndVLDB*, 567–578. Morgan Kaufmann. On-line, available: <http://www.acm.org/sigmod/vldb/conf/1996/P567.pdf>
- Codd, E. *Extending the Data Base Relational Model to Capture More Meaning*. *ACM Transactions on Database Systems* 4(4): 397–434, 1979.
- Das (1992). *Deductive Databases and Logic Programming*. Addison-Wesley.
- Date, C. (1999). *An Introduction to Database Systems*, 7th edition. Addison-Wesley.
- Date, C. (2000). *What, Not How: The Business Rules Approach to Application Development*. Addison-Wesley.
- Date, C. & H. Darwen (1997). *A Guide to the SQL Standard*. Addison-Wesley.
- Decker, H. (1985). *Expression and Enforcement of Integrity Constraints in Prolog KB, Version 0*. ECRC Technical Report KB-3.
- Decker, H. (1987). *Integrity Enforcement on Deductive Databases*. In L. Kerschberg (ed): *Experts Database Systems*. Benjamin Cummings.
- Decker, H. (1989). *The Range Form of Databases and Queries, or: How to Avoid Floundering*. *Proc. 5. ÖGAI*, 114–123. Springer Informatik-Fachberichte 208.
- Decker, H. (1990). *Drawing Updates from Derivations*. ECRC Technical Report KB-65, 1989. Short version in *Proc. 3rdICDT*, 437–451. Springer LNCS, Vol. 470.
- Decker, H. (1996). *An Extension of SLD by Abduction and Integrity Maintenance for View Updating in Deductive Databases*. *Proc. JICSLP '96*, 157–169. MIT Press.
- Decker, H. (1997). *Möglichkeiten der Konsistenzsicherung im MS SQL Server, internal report*, Database Competence Center, Siemens ZT SE 2.

- Decker, H. (1998). *Some Notes on Knowledge Assimilation in Deductive Databases*, in B. Freitag et al (eds), *Transactions and Change in Logic Databases*, 249–286. Springer LNCS, Vol. 1472.
- Decker, H. (2001). *Soundcheck for SQL*. *Proc. PADL '01*, 214–228. Springer LNCS, Vol. 1990.
- Decker, H., M. Celma: (1994). *A Slick Procedure for Integrity Checking in Deductive Databases*. *Proc. ICLP '94*, 456–469. MIT Press, 1994.
- Decker, H., E. Teniente, & T. Urpí (1996). *How to Tackle Schema Validation by View Updating*. *Proc. 5th EDBT*, 535–549. Springer LNCS, Vol. 1057.
- Demolombe, R. (1992). *Syntactical Characterization of a Subset of Domain-Independent Formulas*. *Journal of the ACM* 39(1): 71–94.
- Demolombe, R. (1989). *A. Illarramendi: Heuristics for Syntactical Optimization of Relational Queries*. *Information Processing Letters* 39(6):313–316.
- di Paola, R. (1969). *The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas*. *Journal of the ACM* 16(2): 324–327,.
- Elkan, C. (1990). *Independence of Logic Database Queries and Updates*. *Proc. 9th PoDS*, 154–160. ACM Press.
- Eswaran, K. & D. Chamberlin (1975). *Functional Specifications of a Subsystem for Database Integrity*. *Proc. 1st VLDB*, 48–68. ACM Press.
- Florentin, J. (1974). *Consistency Auditing of Databases*. *The Computer Journal* 17(1), 52–58.
- Fraser, A. (1969). *Integrity of a Mass Storage Filing System*. *The Computer Journal* 12(1): 1–5.
- Gallaire, H. & J. Minker (eds) (1978). *Logic and Data Bases*. Plenum Press,.
- Gallaire, H. & J. Minker, J.-M. Nicolas (1984). *Logic and Databases: A Deductive Approach*. *ACM Computing Surveys* 16(2): 153–185.
- Godfrey, P., J. Gryz, & J. Minker (1996). *Semantic Query Optimization for Bottom-Up Evaluation*, *Proc. ISMIS '96*, 561–571. Springer LNCS Vol. 1079.
- Godfrey, P., J. Gryz & C. Zuzarte (2001). *Exploiting Constraint-like Data Characterizations in Query Optimization*, *Proc. SIGMOD '01*. On-line, available at <http://www.acm.org/sigs/sigmod/sigmod01/e proceedings/>.
- Gryz, J., B. Schiefer, J. Zheng, & C. Zuzarte (2001). *Discovery and Application of Check Constraints in DB2*. *Proc. 17th ICDE*, 551–556. IEEE CS,.
- Gupta, A., I. Mumick, (eds) (1999). *Materialized Views: Techniques, Implementations, and Applications*. MIT Press.
- Gupta, A., Y. Sagiv, J. Ullman, & J. Widom (1994). *Constraint Checking with Partial Information*. *Proc. 13th PoDS*, 45–55. ACM Press.
- Hammer, M. & D. McLeod (1975). *Semantic Integrity in Relational Data Base Systems*. *Proc. 1st VLDB*, 25–47. ACM Press.
- Hammer, M. & S. Sarin (1978). *Efficient Monitoring of Database Assertions (Abstract)*. *Proc. SIGMOD '78*, 159. ACM Press, 1978.
- Homeier, P. V. (1981). *Simplifying Integrity Constraints in a Relational Data Base: An Implementation*. *Master Thesis*, Comp. Sci. Dep't, UCLA.
- Knowledge Partners Inc.: Homepage (2001). On-line, available at: http://www.kpiusa.com/Service_Offerings/Business_Rules.htm.
- Kobayashi, I. (1984). *Validating Database Updates*. *Information Systems* 9(1): 1–17.

- Kowalski, R. (1979). *Logic for Problem Solving*. North-Holland.
- Küchenhoff, V. (1991). *On the Efficient Computation of the Difference between Consecutive Database States*. *Proc. DOOD '91*, 478–502. Springer LNCS, Vol. 566.
- Leuschel, M. & D. De Schreye (1998). *Creating Specialised Integrity Checks through Partial Evaluation of Meta-Interpreters*. *J. Logic Programming* 36(2): 149–193.
- Levy, A. & Y. Sagiv, (1993). *Queries Independent of Updates*. *Proc. 19th VLDB*, 171–181. Morgan Kaufmann.
- Liu, K. & T. Ong (1999). *A Modelling Approach for Handling Business Rules and Exceptions*. *The Computer Journal* 42(3):221–231.
- Lloyd, J., L. Sonenberg & R. Topor (1987). *Integrity Constraint Checking in Stratified Databases*. *J. Logic Programming* 4(4): 331–343.
- Lloyd, J. & R. Topor (1984). *Making Prolog More Expressive*. *J. Logic Programming* 1(3): 225–240.
- Martin, O. & J. F. Perrin (1997). *A Generic Business Rule Validation System for ORACLE Applications*. Presented at *European Oracle User Group Conf.*, 1997. On-line, available at: <http://www.fors.com/eoug97/papers/0230.htm>.
- McJones, P. (ed) (1997). *The 1995 SQL Reunion: People, Projects, and Politics* (2nd edition). On-line, available at: http://www.mcjones.org/System_R/SQL_Reunion_95/.
- Melton, J. & A. R. Simon (1993). *Understanding The New SQL: A Complete Guide*. Morgan Kaufmann.
- Minsky, N. (1974). *On Interaction with Data Bases*. *Proc. SIGMoD '74*, Vol. 1, 51–62. ACM Press.
- Nicolas, J.-M. (1978). *First Order Logic Formalization for Functional, Multivalued and Mutual Dependencies*. *Proc. SIGMoD '78*, 40–46, ACM Press.
- Nicolas, J.-M. (1982). *Logic for Improving Integrity Checking in Relational Databases*. *Acta Informatica* 18: 227–253.
- Ocelot: OCELOTSQL Homepage (2001). On-line, available at: <http://ourworld.compuserve.com/homepages/OCELOTSQL/>.
- Omnibuilder: Business Rules (Overview) (2001). On-line, available at: http://www.omnibuilder.com/overview/bus_rule.htm.
- Ramakrishnan, R. & J. Ullman (1995). *A Survey of Deductive Database Systems*. *J. Logic Programming* 23(2): 125–149.
- Ross, K., D. Srivastava & S. Sudarshan (1996). *Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time*. *Proc. SIGMoD '96*, 447–458. ACM Press.
- Ross, K. & G. Lam (2001). *Capturing Business Rules. Business Analysis and Requirements Workshop*. Online description available at <http://www.dci.com/events/rules/>.
- Sadri, F. & R. Kowalski (1988). *A Theorem-Proving Approach to Database Integrity*, in J. Minker, (ed), *Foundations of Deductive Databases and Logic Programming*, 313–36. Morgan Kaufmann.
- Stonebraker, M. (1975). *Implementation of Integrity Constraints and Views by Query Modification*. *Proc. SIGMoD '75*, 65–78. ACM Press.
- Ullman, J. (1988). *Principles of Database and Knowledge-Base Systems*, Volume Computer Science Press.
- USoft: Homepage (2001). On-line, available at: <http://www.ness-europe.com>.

- Van Gelder, A. & R. Topor (1991). *Safety and Translation of Relational Calculus Queries*. *ACM Transactions on Database Systems* 16(2): 235–278.
- Vieille, L., P. Bayer, V. Küchenhoff & A. Lefebvre, (1999). *Integrity Checking and Materializing Views by Update Propagation in the EKS System*. In (Gupta & Mumick), 421–440.
- Wetzel, F. & F. Toni (1998). *Semantic Query Optimization through Abduction and Constraint Handling*. *Proc. FQAS '98*, 366–381. Springer LNAI, Vol. 1495.
- Widom, J. & S.Ceri, (eds) (1996). *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.
- Wilkes, M. L. (1972). *On Preserving the Integrity of Data Bases*. *The Computer Journal* 15(3): 191–194, 1972.

Chapter VIII: Functional Dependencies for Value Based Identification in Object-Oriented Databases

Jochen Rasch, SAP AG,

Germany

Hans-Joachim Klein, Universität Kiel,

Germany

INTRODUCTION

The Entity-Relationship (ER) model (Chen, 1976) is frequently used for the specification of conceptual database schemas. Entity types and relationship types are the building blocks provided by this data model. A major objective of conceptual database design (see e.g. Batini, Ceri, & Navathe, 1992) is to define entity types and relationship types in such a way that they represent meaningful units of information with respect to the semantics of the modeled application domain. Object types in object-oriented data models are comparable to entity types as far as structural aspects are concerned. However, the ER model explicitly supports relationship types, whereas different approaches are found in object-oriented data models: On the one hand, relationship types may be represented by reference attributes, which means that they are part of the object type itself. This approach, for example, is often found in programming languages. On the other hand, they may be represented explicitly by means of a separate concept as in the ER model. In the latter case there is a close resemblance between entity types and object types. Object types represent sets of objects which are of relevance in the application domain, while relationship types represent relationships between objects.

One of the fundamental concepts of the object-oriented approach is object identity (Koshafian & Copeland, 1986). Objects are distinguishable even if they coincide in all their externally visible properties. In conceptual data models, this abstract concept is usually realized by abstract object identifiers or surrogates (see e.g. Hall, Owlett, & Todd, 1976; Codd, 1979; Abiteboul & Kanellakis, 1989), i.e. by internal identifiers to which the query language of a data model does not provide direct access. The only operation which usually is available, is the test whether two given objects are identical or not. Thus, a realization of identity does not necessarily support *identification*, i.e. *external* access to a single object. Hence, in general, objects have to be addressed by their properties.

In a database state it should be possible to address every object of an object type by specifying some of its properties at the level of values and relationships according to the semantic units which are specified by the conceptual schema. Therefore the access to an object by starting with a value or some combination of values is of importance. Identifying values either give direct access to a single object, or they determine objects as starting points for the retrieval of an object by navigating along relationships in the given state. This navigational view, i.e. entering a database state via values and following references among objects, is related to the functionality provided typically by visual interfaces for browsing object databases.

Application domains often suggest natural value based identification criteria (VBICs). From the user's point of view these criteria are preferable to any artificial identifier attributes since they carry semantics of the modeled domain. The significance of value based identification mechanisms for objects has been emphasized by several authors (e.g. Abiteboul & Van den Bussche, 1995; Beerli, 1993; Gogolla, 1995; Kim, 1993; Schewe & Thalheim, 1993; a comparison of different mechanisms for distinguishing objects can be found in Beerli & Thalheim, 1993). However, less work has been done in characterizing and investigating reasonable forms of VBICs together with their interaction.

In the relational data model (Codd, 1970) with the *relation* as single data-modeling concept, value based identification is connected with the notion of *key*. A key of a relation type is a set π of attributes such that in every meaningful relation of this type no two entries exist having identical values on π . Keys are a special form of *functional dependencies*. This kind of *static integrity constraint* allows to formulate conditions for relations of a given type, requiring that identical values on one specified set of attributes imply identical values on another specified set of attributes for every pair of entries. The theory of constraints in general and of functional dependencies and keys in particular is well-established for the relational data model (Maier, 1983). In data models with more than one data-modeling concept, the formal foundations of constraints are less developed and there are a number of open problems especially in connection with reasonable forms of functional dependencies and VBICs.

Some examples of VBICs to be covered by a more general theory of constraints on both object and value level are given in [Figure 1](#) and [Figure 2](#): Hotels offer rooms in different categories, single and double rooms ([Figure 1\(a\)](#)). The values of *Accommodation* attributes cannot be used as entry values to access a single *Accommodation* object

because different hotels may offer rooms in the same category. A *Hotel* object together with a value for *Room_category*, however, uniquely identifies an *Accommodation* object. If *Name* is a key for *Hotel*, i.e. *Hotel* objects are identified by their *Name* value, *Room_category* and *Name* can serve as a VBIC for *Accommodation*. Consider the schema in [Figure 1\(b\)](#), representing information about different branches of a bank. A customer may have accounts at different branches. At each branch, one clerk is assigned to her or him as investment consultant. Therefore *Branch* and *Customer* objects together determine *Clerk* objects. If *Street* and *C_no* are keys for *Branch* and *Customer*, respectively, a VBIC for *Clerk* is given. This provides an additional identification criterion besides the obvious identification of *Clerk* by *Emp_no*, which may be useful for some applications. Neither *Street* nor *C_no* alone is sufficient to identify *Clerk* objects.

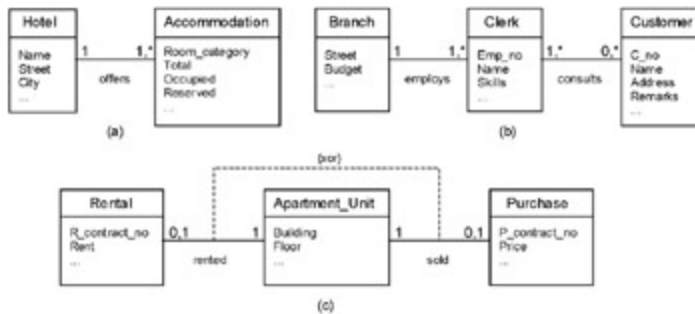


Figure 1: Some examples of VBICs

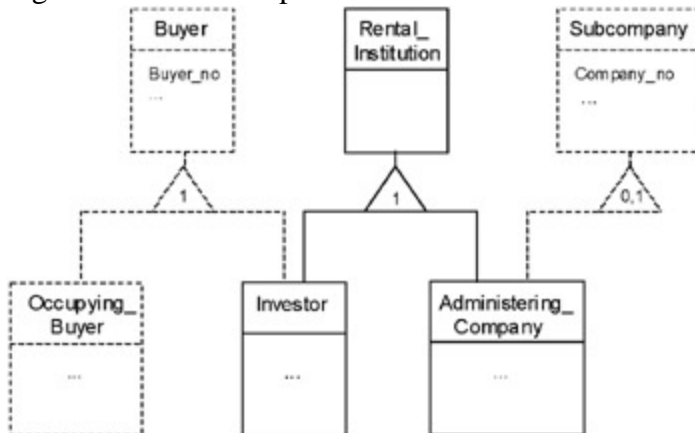


Figure 2: VBIC by inheritance

[Figure 1\(c\)](#) shows an example of a different form of identification: Apartments in a building are either rented or condominium apartments. Therefore the number of the lease or the sales contract can be used to distinguish *Apartment_Unit* objects by value, leading to a ‘disjunctive identification’ of apartments either by *Rental* or by *Purchase*. The exclusive-or constraint between the relationships *rented* and *sold* is expressed by the dotted line with constraint type specified as {xor} (cf. Booch, Rumbaugh, & Jacobson, 1998).

Assume that a building contractor runs, via subcompanies, some of the apartments he builds. He offers this service also to private investors who bought an apartment but do not occupy it themselves. So a *Rental_Institution* is specialized either to *Investor* or to

Administering_Company, as shown by the mandatory inheritance hierarchy in [Figure 2](#). *Investor* is a specialization of type *Buyer* and *Administering_Company* an optional specialization of type *Subcompany* (not every subcompany is an owner of apartments). *Buyer* and *Subcompany* objects are distinguishable by their *Buyer_no* and *Company_no* value, respectively. The attributes are inherited to the specialized types therefore also providing VBICs for *Investor* and *Administering_Company*, respectively. In this scenario, value based identification of *Rental_Institution* objects becomes possible since every *Rental_Institution* object is specialized. This leads to ‘identification by generalization/specialization’, similar to the disjunctive criteria presented in [Figure 1\(c\)](#).

One simple approach to provide value based identification of objects (especially in the frequent case of object-oriented database design and relational implementation) is to introduce an artificial identifier attribute (Hull & Yoshikawa, 1991; Rumbaugh, Blaha, Premerlani et al., 1991), i.e. to make the abstract identifier visible. This approach should not be regarded as value based identification in the original sense because it does not refer to the values and relationships of the objects themselves. The other extreme is to use the complete object value for identification, including references to other objects recursively (Abiteboul & Kanellakis, 1989; Abiteboul & Van den Bussche, 1995; Denninghoff & Vianu, 1993), leading to the notion of *deep equality* of objects. This is a very general way to identify objects by values. However, deep equality takes into account only references starting from an object and therefore does not consider the complete ‘relationship environment’ of an object, since references directed towards the object are ignored. From a practical point of view, the use of the complete object value as a VBIC, including all relationships to other objects, is inappropriate. Thus, the question is raised how to determine VBICs similar to the key concept of the relational data model. A reasonable solution should lie between these two extremes by exploiting all features of the object-oriented data model. In Chen (1976) a relationship with cardinality restricted to 1, i.e. representing a function between entity sets, may contribute to obtain a key for an entity type (so-called *weak entity type*, similar to the example in [Figure 1\(a\)](#)) by using the key of another entity type related to it. The idea of using relationships to find VBICs was applied in Zaniolo (1979) to determine keys for records in a network schema. There not only record types of a single set type were taken into account but also record types reachable via a sequence of set types. In Schewe and Thalheim (1993) this proceeding was applied to object-oriented schemas using a sequence of relationships between classes, including inheritance. More general approaches to determine keys are presented by *observation formulas* and *object terms* in Abiteboul and Van den Bussche (1995) and Gogolla (1995), respectively. However, there is no further consideration how to determine the proposed terms and how to distinguish different kinds of identification terms. This chapter presents an approach to this matter by generalizing the well-known functional dependencies to object schemas. Like functional dependencies provide a foundation for the specification and derivation of key constraints, these generalized constraints—denoted as object functional dependencies—provide a framework for the specification of VBICs.

We first introduce some basic notions of the object model we use, including a formalization of the terms object schema and schema graph, as well as some concepts of

the relational data model. Then we describe an approach to generalize functional dependencies to object functional dependencies. These graph based constraints are spanning trees of subgraphs of a given schema graph, labeled appropriately with types of the schema. Different semantics for object functional dependencies are presented based on a relational representation of relevant parts of states. The proposal is compared to related approaches and some interesting challenges for future research are pointed out in the conclusions.

SYNTAX OF OBJECT FUNCTIONAL DEPENDENCIES

Basics and Notation

The object-oriented data model used in the following is similar to the static part of the Object Modeling Technique and the Unified Modeling Language (Rumbaugh, Blaha, Premerlani et al., 1991; Booch, Rumbaugh & Jacobson, 1998) and covers their basic structural modeling constructs: object types supporting object identity and tuple-structured object values, binary relationships and inheritance hierarchies between object types; for relationships and inheritance hierarchies *cardinality constraints* can be specified.

An **object schema** S consists of a finite number of **object types** and binary **relationship types (relationships)** for short) between them, including inheritance. An object type O has a finite set $attr_S(O)$ of **attributes** with a domain assigned to each attribute. Relationships may have **cardinalities** as additional constraints. Object types, attributes, and relationships are assumed to be unique within a schema. Inheritance hierarchies are considered as a set of binary relationships with additional constraints for optional and mandatory hierarchies.

Object identity is covered by introducing a countably infinite set I of **object identifiers**. An object o of type O is a pair $o = (i, v)$ with $i \in I$ and v being a tuple with attribute set $attr_S(O)$, called the **object value**.

The separation of identifiers and values reflects the requirement that object identity has to be independent of object values. An **extension** $ext(O)$ of an **object type** O is a finite set of objects of type O such that the identifiers are unique within $ext(O)$. Uniqueness is necessary in order to guarantee distinguishability of objects independent of their values.

An **extension** $ext(r)$ of a **relationship** r between object types O, O' with extensions $ext(O), ext(O')$ is a finite set of **links** (i, j) with $i \in ext(O), j \in ext(O')$. If a cardinality constraint is specified for r then each extension of r has to comply with the constraint. As an example, consider the relationship *offers* in [Figure 1\(a\)](#). In an extension of *offers*, every *Accommodation* object must participate in exactly one link whereas *Hotel* objects must be present in one or more links.

A **state** $s(S)$ of an object schema S consists of an extension for every object type and every relationship of S , with the sets of object identifiers occurring in the extensions being disjoint for each pair of different objects types O, O' of S .

For the handling of object schemata a graph representation is appropriate. The **schema graph** G_S of an object schema S is an edge-labeled graph $(V_S, E_S, ?_S)$. The set of nodes V_S corresponds to the object types of S and the set of edges E_S represents the relationships of S . The edge-labeling function $?_S$ is given by the names of the relationships.

In the following, some notions from the relational data model are compiled which are subsequently used:

Let $\beta = \{B_1, \dots, B_m\}$ be a finite set of attributes with domain function $dom : \beta \rightarrow \{D_1, \dots, D_k\}$, where each D_i is a non-empty domain of atomic values. A **tuple** over β is a function $t : \beta \rightarrow \prod_{i=1}^k D_i$ with $t(B) \in dom(B)$ for each $B \in \beta$. For $\beta' \subseteq \beta$, $t|_{\beta'}$ denotes the **restriction** of t to attribute set β' . A **partial tuple** t over β is a tuple over β with $t(B) \in dom(B)$ or $t(B)$ undefined. An **undefined value** (or **null value**) is represented by special symbol '–'. A tuple t over β is **total** on β' iff $t(C) \neq \text{–}$ for each $C \in \beta'$. t is **undefined** on β' iff $t(C) = \text{–}$ for each attribute $C \in \beta'$. A **(partial) relation** over β is a finite set of (partial) tuples over β . For a relation R , a_R denotes the **set of attributes** of R . If R is a relation and $\beta \subseteq a_R$, then $R[\beta] := \{t|_{\beta} : t \in R\}$ denotes the **projection** of R onto β .

In examples, tuples and relations will be written in the usual tuple and table notation assuming an arbitrary but fixed order on β to be given.

For an attribute set $\beta \subseteq a_R$, the **strong null filter** $SNF(R, \beta)$ denotes the set of all tuples of R which are total on β . The **weak null filter** $WNF(R, \beta)$ denotes the set of all tuples of R which are not undefined on β . The **total projection** $R[\beta]_{tot}$ of relation R onto β is the set $SNF(R[\beta], \beta)$ of all tuples from $R[\beta]$ which are total on β .

Consider the following relations:

R_1	R_2	R_3	R_4
A B C	A B C	A B C	A B
1 2 3	1 2 3	1 2 3	1 2
1 3 –	1 3 –	1 3 –	1 3
– 3 4		– 3 4	
4 – 4		4 – 4	
– – 5			

R_2 is the result of applying the strong null filter on attribute set $\beta = \{A, B\}$ to R_1 , whereas R_3 is obtained by applying the weak null filter on β to R_1 . R_4 is the result of the total projection $R_1[\beta]_{tot}$ of R_1 onto β .

Let t and t' be partial tuples over β . t' **subsumes** t on β (in symbols: $t' =_{\beta} t$, $t =_{\beta} t'$) iff for each $C \in \beta$, $t'(C)$ or $t(C) = \text{--}$ holds. To give an example, consider tuples $t_1 = (1, 2, 3)$, $t_2 = (1, 2, -)$, $t_3 = (1, -, 3)$, and $t_4 = (-, 2, -)$ over an attribute set $\beta = \{A, B, C\}$. Then t_1 subsumes t_2 , t_3 , and t_4 on β ; t_2 subsumes t_4 on β ; t_3 subsumes t_4 on β ; but neither does t_2 subsume t_3 nor does t_3 subsume t_2 on β .

For an extension $ext(O)$ of an object type O , the **relational representation** $rel(ext(O))$ is the relation R with $a_R = \{Id_O\}$ and R being the set of tuples t such that for each t an object $(i, ?) \in ext(O)$ with $i = t(Id_O)$ and $? = t|_{attr_S(O)}$. Id_O is called **identifier attribute**. For every relationship r between object types O and O' , the set of links $ext(r)$ can be considered as a relation over attribute set $\{Id_O, Id_{O'}\}$. This **relational representation** of r is denoted by $rel(ext(r))$.

Relationships in hierarchies can be treated analogously. Here, the relational representation of a relationship consists of pairs (i, i') of identifiers being elements of the extensions of both object types involved in the relationship.

Functional Dependencies Generalized

A straightforward approach to the specification of integrity constraints for object types is the generalization of functional dependencies (FDs) known from the relational model. This proceeding allows to take advantage of the well-founded theory of FDs. Furthermore, such ‘extended FDs’ provide a more general framework for the specification of integrity constraints for object schemas, with VBICs being a special form of these constraints.

Consider a relation R with attribute set a_R and an FD $f: \beta \rightarrow \beta'$ with $\beta, \beta' \subseteq a_R$. f refers only to attributes of a_R , and whether f is satisfied in a given relational database solely depends on R and not on any further relations in the database. A straightforward application of this concept to object schemas leads to a constraint on the level of object types, i.e. left-hand and right-hand side of an FD f may not only refer to attributes of O but also to the object type itself, by regarding the object identifier as the value of a special attribute: $f: \beta \rightarrow \beta'$ with $\beta, \beta' \subseteq attr_S(O) \cup \{O\}$, for some object type O of a schema S . This allows to express constraints stating that objects of O are identifiable by their value or a part thereof in the same way as tuples in a relation can be distinguished by looking at their values in key attributes. f is restricted to local identification of O solely by its own attributes, like it is known from common FDs as intra-relational integrity constraints on a single relation type. However, if the attribute values of an object are not sufficient to identify it in the extension of O in a state $s(S)$, relationships to other objects and the values of these objects can be taken into account. The simplest example for this kind of identification is the weak entity concept of the ER model (Chen, 1976). Generalizing this approach leads to FDs of the form $f: \beta \rightarrow \beta'$ with sets β and β' where several object types O of S may contribute to β or β' , i.e., $\beta \subseteq attr_S(O)$ or $\beta = \{O\}$ for each $O \in \beta$ and some $O \in O_S$. Object types, between which a dependency of this kind exists, do not have to be directly connected in the schema graph G_S , i.e., they do not have to be

connected by a single relationship as it is the case for weak entity types. A path between them is sufficient.

Between any two object types contributing to γ , or between a type contributing to γ and a type contributing to G more than one path may exist in G_S . In schema (a) from [Figure 3](#), for example, there are two different paths of relationships (via p, q and via r, s, t) for an FD $\{O_1\} \rightarrow \{O_3\}$, connecting O_1 and O_3 . For schema (b) and FD $\{O_1, D\} \rightarrow \{L\}$, two paths connecting O_3 with O_6 and two paths between O_1 and O_3 can be found. Usually, different paths correspond to different semantics and a database modeler has one of these paths in mind when specifying a dependency (cf. Lien, 1982). This ambiguity is also known from the universal relation approach (cf. Maier, Ullman & Vardi, 1984). Moreover, it is possible for an FD to be satisfied with respect to one path and not with respect to another one. A state for the schema in [Figure 3\(b\)](#) illustrating this can easily be constructed.

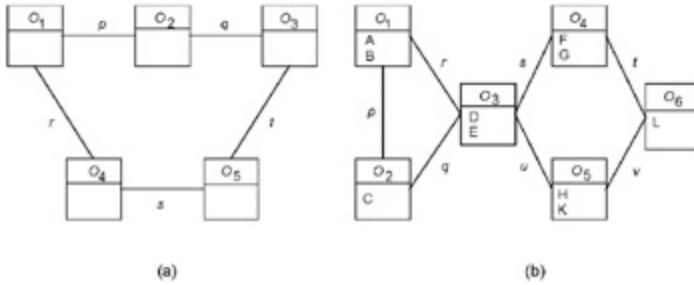


Figure 3: Ambiguity of an FDI at object schema level

For a generalization dealing with the mentioned aspects, a graph-based approach seems to be appropriate. We concentrate on dependencies with tree structure, i.e. subgraphs of the schema graph that are trees. Every leaf node of such a tree corresponds to an object type that provides an ‘entry’ to ‘targets’ and/or whose objects or attribute values thereof are determined by some ‘entry values’. Non-leaf nodes can be entries and/or targets, too, but they may also play none of these roles, representing merely a connecting condition. The next two definitions present the notion of generalized FDs, called *object functional dependencies*.

Definition 1 Let S be an object schema. For an object type $O \in O_S$, $sets_S(O)$ denotes the **label set** consisting of the set $\{O\}$ and all attribute subsets of O . L_S denotes the union of all label sets of object types belonging to S .

Definition 2 Let S be an object schema with schema graph $G_S = (V_S, E_S, \gamma_S)$. An **object functional dependency (OFD)** of S is an edge- and node-labeled graph $f = (G_f, \gamma_f)$ with the following properties:

- i. The **OFD graph** $G_f = (V_f, E_f, \gamma_f)$ is an edge-labeled spanning tree of node set $V_f \subseteq V_S$ in G_S with $V_f \neq \emptyset$. γ_f is the restriction of γ_S to E_f .
- ii. $\gamma_f: V_f \rightarrow L_S \times L_S$ is a partial **node-labeling function** such that for each $O \in V_f$ with γ_f defined and $\gamma_f(O) = (d, \gamma)$ holds:

- $d, ?? \text{ sets}_S(O)$ and $d? ?? \emptyset$
- $?_f$ has to be defined at least for every leaf node.

A **source (object) type** is a node O with $?_f(O)=(d, ?)$ and $d? \emptyset$. A **sink (object) type** is a node O with $?_f(O)=(d, ?)$ and $?? \emptyset$. A node O for which $?_f$ is undefined is called **connecting (object) type**.

Figure 4 shows the OFDs corresponding to the examples from Figure 1(a) and (b). The graph in (a) represents the constraint that a *Hotel* object together with a value for *Room_category* in one of the *Accommodation* objects related by *offers* to the *Hotel* object uniquely identifies one of these *Accommodation* objects. The graph in (b) represents the constraint that the values in attributes *Street* and *C_no* of *Branch* and *Customer* objects, respectively, uniquely determine an object among the *Clerk* objects which are connected to them via *employs* and *consults*.



Figure 4: Examples of OFDs

The specification of OFDs by spanning trees of nodes of the schema graph guarantees that there are no ambiguities with respect to the connections between source and sink types. Since an object type can be a source as well as a sink type (e.g. object type *Accommodation* in Figure 4(a)), the node labels are chosen to be pairs. For an object type O , attributes are not mixed together with the type in a node label from $\text{sets}_S(O)$ since the object identifier uniquely determines an object and therefore its value, too.

A set oriented FD-like notation is often sufficient and more convenient to denote OFDs. It can be derived from the node labels of an OFD as follows:

Definition 3 Let S be a schema with schema graph G_S and $f = (G_f, ?_f)$ be an OFD of S with node set V_f . The **set notation** $? - G_f ?$ $\text{Gof } ?_f$ is obtained by collecting the first (?) and the second (G) non-empty components of node labels separately. ? is called the **left-hand side**, G the **right-hand side** of f . $d? ?$ is called an **entry** and $?? G$ is called a **target** (of f). A combination of values for d (or object, if d is an object type) is denoted as **entry unit**. Analogously, a combination of values for ? is called **target unit**. For a given state, the terms **source object** and **sink object** will be used to denote objects belonging to the extension of an object type involved in the left-hand and right-hand side of an OFD, respectively.

An object type O is **referred** to by the OFD or **involved** in the OFD iff O itself or any subset of its attribute set occurs in the left-hand or right-hand side of f . O is involved in f

at **type-level** iff $\{O\} \rightarrow G$. f is called **canonical** iff only one object type is involved in G .

$f = G_f$? Or simply G , if G_f is uniquely determined by f , G and the underlying schema graph, will be used as a shorthand notation for an OFD

$f = (G_f, v_f)$ with OFD graph G_f and set notation v_f . We occasionally omit set braces, especially for singletons, in order to simplify notation.

$f_a: \{Hotel, Room_category\} \rightarrow \{Accommodation\}$ and

$f_b: \{Street, C_no\} \rightarrow \{Clerk\}$

are the set notations for the OFDs from [Figure 4](#).

The following example demonstrates that, in general, the OFD graph is necessary to represent the subgraph of G_f referenced by an OFD f , i.e., the set notation alone is not sufficient.

Example 1 [Figure 5](#) shows two OFDs f_1, f_2 of the schema from [Figure 3\(b\)](#), with $V_{f1} = \{O_1, O_3, O_4, O_5\}$ and $V_{f2} = \{O_1, O_2, O_3, O_4, O_5\}$. Both OFDs have the set notation $\{A, O_5, F\} \rightarrow \{B, O_4\}$.

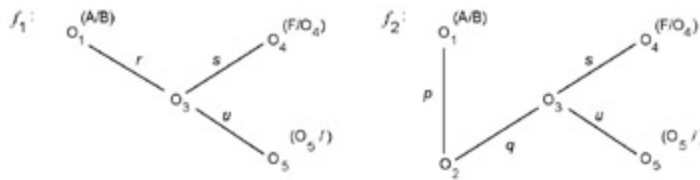


Figure 5: Different OFDs with identical set notation

For the identification of objects by attribute values, the key concept of the relational data model can be generalized for object schemas using OFDs as follows:

Definition 4 A value-based identification criterion (VBIC) for an object type O is an OFD $f: \{O\} \rightarrow \{O\}$ with each $d \in V_f$ containing only attributes.

For example, OFD f_b from [Figure 4](#) is a VBIC, whereas OFD f_a is no VBIC since it contains object type *Hotel* in its left-hand side.

Remark With a more restrictive meaning the notion of an ‘object functional dependency’ was used by Lee (1995) to denote functional dependencies specified with respect to a single class. The object identifier is treated as a special attribute and may be used in dependencies.

SEMANTICS OF OBJECT FUNCTIONAL DEPENDENCIES

To capture its intuitive meaning and to understand the restriction imposed by an OFD on states of a schema, a formal semantics is needed. Since OFDs are a generalization of FDs, it is an obvious choice to define semantics for OFDs in the style of FD semantics. For this, an equivalent of a relation, i.e., the part of a state that is of relevance for checking the satisfaction of an OFD, is needed.

For a state $s(S)$ and an OFD $f: ? - G_f ?$ of schema S we can specify which objects and which links between them are of relevance with respect to f . Such ‘units’ of relevant objects connected by links are denoted as **linkages**. A linkage is a connected maximal subtree of the state graph which structurally corresponds to the OFD graph G_f . It consists of objects of types occurring in the node set of G_f and of links connecting them. The relevant objects are source or sink objects.

Depending on the extensions and cardinalities of the relationships participating in G_f , sink objects may exist which are connected to objects of some but not all of the source types of f . In such a case the linkage corresponds to a connected, proper subgraph of G_f and is called **partial linkage**. If a linkage contains an object for each of the source types of G_f , it is called **total linkage**.

Furthermore, sink objects can exist which are not connected to any source object. They are represented by **insufficient linkages** that contain no source object. These have to be considered, too, because the existence of such objects in a state might indicate that f is invalid.

Linkages provide information about entry combinations by which objects of sink types can be accessed. A total linkage l corresponds to a total entry combination by which the sink objects or a combination of sink objects occurring in l can be reached. Analogously, a partial linkage of a sink object corresponds to a partial entry combination. Sink objects that occur only in insufficient linkages of a state cannot be accessed from the entries specified by an OFD via any total or partial entry combination, i.e. they are not reachable from any entry provided by the OFD.

Base Relation and Validation Relations

Intuitively speaking, the basis for checking whether an OFD $f = (G_f, v_f)$ does hold or does not hold in a state $s(S)$ is the set of all linkages of f . This set can be represented by a *base relation* $base_{f,s(S)}$. Then the validity of an OFD can be defined similar to the validity of an FD with respect to a relation. If f is a local OFD, i.e., referring to exactly one object type O , the base relation is determined uniquely by $rel(ext(O))$, the relation representing $ext(O)$. If f is non-local, such a relation is built by joining the relations that represent the extensions of all object types and relationships occurring in G_f . Linkages are represented by tuples in the resulting relation.

For a non-local OFD with an object type O involved in the right-hand side, it has to be ensured that every object of $ext(O)$ appears in the base relation in order to be able to check whether every object of a sink type can be accessed by an entry combination, i.e. a combination of source objects or attribute values thereof. The construction of the base relation $base_{f,s(S)}$ has to take into account the possible existence of partial or insufficient linkages. Thus, $base_{f,s(S)}$ will in general be a partial relation, although the extensions of a state are total, and the natural join (see e.g. Codd, 1970) cannot be used to combine relations due to the well-known effect of discarding ‘dangling tuples’. To appropriately represent the part of $s(S)$ referenced by f , the *full outer join* (Lacroix & Pirotte, 1976) has to be used, modified to operate on partial relations. Because the symbol ‘—’ represents non-existing links, ‘—’ is a null value in the sense of ‘value does not exist’ and partial relations represent complete information about the underlying extensions. Moreover, this null value does not affect the evaluation of f on $base_{f,s(S)}$: As in the case of an FD, checking f is done by checking equality of attribute values in $base_{f,s(S)}$. In this context, a comparison $c = \text{‘—’}$ can be evaluated to *false*, and $\text{‘—’} = \text{‘—’}$ to *true* for every domain value or object identifier c since two objects, one with, the other without link of the same relationship, can obviously be distinguished. For this reason, the issues which in general have to be taken into account if FDs are extended to partial relations can be ignored in this case.

The full outer join operation, modified to handle objects without links, i.e., dangling tuples which are undefined on the intersection attributes, is presented in the next definition.

Definition 5 Let R, S be partial relations over attribute sets a_R, a_S , respectively, with $|a_R \cap a_S| = 1$. The **full outer join (FOJ)** $R \Join_{fo} S$ of R and S is defined as

$$\begin{aligned}
 R \Join_{fo} S := & \\
 & \{t \mid t \text{ tuple over } \alpha_R \cup \alpha_S \wedge \\
 & ((t \text{ total on } \alpha_R \cap \alpha_S \wedge t|_{\alpha_R} \in R \wedge t|_{\alpha_S} \in S) \\
 & \vee (t|_{\alpha_R} \in R \wedge t \text{ total on } \alpha_R \cap \alpha_S \wedge \neg(\exists t' \in S)(t|_{\alpha_R \cap \alpha_S} = t'|_{\alpha_R \cap \alpha_S})) \\
 & \wedge t \text{ undefined on } \alpha_S \setminus \alpha_R) \\
 & \vee (t|_{\alpha_S} \in S \wedge t \text{ total on } \alpha_R \cap \alpha_S \wedge \neg(\exists t' \in R)(t|_{\alpha_R \cap \alpha_S} = t'|_{\alpha_R \cap \alpha_S})) \\
 & \wedge t \text{ undefined on } \alpha_R \setminus \alpha_S) \\
 & \vee (t|_{\alpha_R} \in R \wedge t \text{ undefined on } \alpha_S) \\
 & \vee (t|_{\alpha_S} \in S \wedge t \text{ undefined on } \alpha_R))\}
 \end{aligned}$$

By the first three join conditions of the disjunction, the outer join is built for tuples total on the intersection $a_R \cap a_S$. They correspond to the definition of the full outer natural join for total relations, preserving dangling tuples from both operands. Because it is sufficient for our purposes, the intersection is restricted to a single join attribute. Thus, tuples are either total or undefined on $a_R \cap a_S$. By the last two conditions, tuples undefined on the join attribute are added to the result. They guarantee that partial tuples in intermediate relations are not lost during further join operations.

$rel(ext(O_1))$	$rel(ext(r_1))$	$rel(ext(O_2))$	$rel(ext(r_2))$	$rel(ext(O_3))$																																														
<table> <tr> <th>Id_{O_1}</th> <th>A</th> </tr> <tr> <td>i_1</td> <td>1</td> </tr> <tr> <td>i_2</td> <td>1</td> </tr> <tr> <td>i_3</td> <td>2</td> </tr> </table>	Id_{O_1}	A	i_1	1	i_2	1	i_3	2	<table> <tr> <th>Id_{O_1}</th> <th>Id_{O_2}</th> </tr> <tr> <td>i_1</td> <td>i_4</td> </tr> <tr> <td>i_1</td> <td>i_5</td> </tr> <tr> <td>i_2</td> <td>i_6</td> </tr> </table>	Id_{O_1}	Id_{O_2}	i_1	i_4	i_1	i_5	i_2	i_6	<table> <tr> <th>Id_{O_2}</th> <th>B</th> </tr> <tr> <td>i_4</td> <td>2</td> </tr> <tr> <td>i_5</td> <td>3</td> </tr> <tr> <td>i_6</td> <td>4</td> </tr> <tr> <td>i_7</td> <td>4</td> </tr> </table>	Id_{O_2}	B	i_4	2	i_5	3	i_6	4	i_7	4	<table> <tr> <th>Id_{O_2}</th> <th>Id_{O_3}</th> </tr> <tr> <td>i_5</td> <td>i_9</td> </tr> <tr> <td>i_6</td> <td>i_{10}</td> </tr> <tr> <td>i_6</td> <td>i_{11}</td> </tr> <tr> <td>i_7</td> <td>i_{11}</td> </tr> </table>	Id_{O_2}	Id_{O_3}	i_5	i_9	i_6	i_{10}	i_6	i_{11}	i_7	i_{11}	<table> <tr> <th>Id_{O_3}</th> <th>C</th> </tr> <tr> <td>i_8</td> <td>3</td> </tr> <tr> <td>i_9</td> <td>4</td> </tr> <tr> <td>i_{10}</td> <td>5</td> </tr> <tr> <td>i_{11}</td> <td>6</td> </tr> </table>	Id_{O_3}	C	i_8	3	i_9	4	i_{10}	5	i_{11}	6
Id_{O_1}	A																																																	
i_1	1																																																	
i_2	1																																																	
i_3	2																																																	
Id_{O_1}	Id_{O_2}																																																	
i_1	i_4																																																	
i_1	i_5																																																	
i_2	i_6																																																	
Id_{O_2}	B																																																	
i_4	2																																																	
i_5	3																																																	
i_6	4																																																	
i_7	4																																																	
Id_{O_2}	Id_{O_3}																																																	
i_5	i_9																																																	
i_6	i_{10}																																																	
i_6	i_{11}																																																	
i_7	i_{11}																																																	
Id_{O_3}	C																																																	
i_8	3																																																	
i_9	4																																																	
i_{10}	5																																																	
i_{11}	6																																																	

Example 2 Consider the schema S and the state $s(S)$ in [Figure 6](#). The relational representations of the extensions from $s(S)$ are listed below the state graph. Applying the FOJ to $rel(ext(O_1))$ and $rel(ext(r_1))$ yields relation R_1 , and joining R_1 with $rel(ext(O_2))$ results in R_2 :

$$R_1 := rel(ext(O_1)) \bowtie_{j_0} rel(ext(r_1)) \quad R_2 := R_1 \bowtie_{j_0} rel(ext(O_2))$$

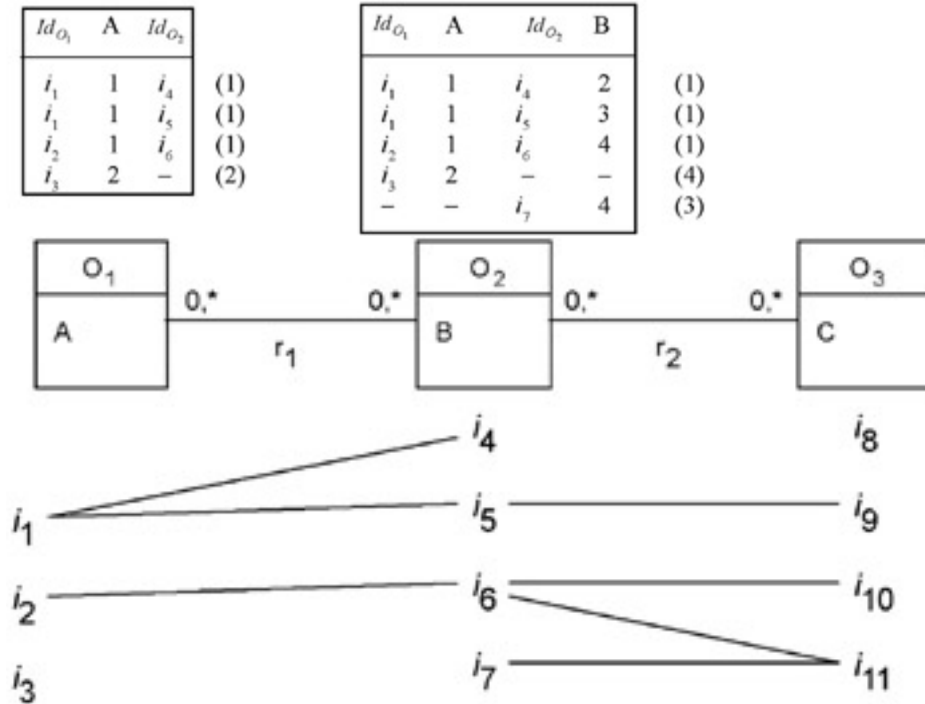


Figure 6: Simple schema with state and relational representations (object values, edge and node labels are omitted from the state graph)

Here each join attribute is an identifier attribute. The number listed for each tuple of a relation refers to one of the five join conditions. For example, the fourth tuple in R_1 is a dangling tuple from the left operand while the fifth tuple in R_2 is a dangling tuple from the right operand. Condition (4) guarantees that the fourth tuple from R_1 is represented in R_2 , too. All other tuples in R_1 and R_2 stem from the natural join condition of the FOJ. Analogously, relations R_3 and R_4 are constructed:

Here, the fifth tuple of R_3 , being undefined on attribute Id_{O_2} , is represented in the final relation R_4 due to the fifth join condition.

$$R_3 := \text{rel}(\text{ext}(r_2)) \bowtie_{f_2} \text{rel}(\text{ext}(O_3))$$

$$R_4 := R_2 \bowtie_{f_0} R_3$$

Id_{O_2}	Id_{O_3}	C	
i_5	i_9	4	(1)
i_6	i_{10}	5	(1)
i_6	i_{11}	6	(1)
i_7	i_{11}	6	(1)
—	i_8	3	(3)

Id_{O_1}	A	Id_{O_2}	B	Id_{O_3}	C	
i_1	1	i_4	2	—	—	(2)
i_1	1	i_5	3	i_9	4	(1)
i_2	1	i_6	4	i_{10}	5	(1)
i_2	1	i_6	4	i_{11}	6	(1)
i_3	2	—	—	—	—	(4)
—	—	i_7	4	i_{11}	6	(1)
—	—	—	—	i_8	3	(5)

Using the FOJ, a relation for checking an OFD f can be constructed. It contains a tuple for each linkage of f and is built by joining successively the relational representations of the extensions of all object types and relationships occurring in the OFD graph. The sequence of FOJ operations pays regard to the structure of the OFD graph.

Definition 6 Let S be an object schema, $s(S)$ be a state of S and let $f: \{d_1, \dots, d_k\} - G_f? \{?, \dots, ?_m\}$ be an OFD with graph $G_f = (V_f, E_f, ?_f)$ and node-labeling function v_f . Let

and

The **base relation** $base_{f,s(S)}$ of f under $s(S)$ is defined as follows:

- i. If f is a local OFD referring to object type O , $base_{f,s(S)}$ is the relational representation $\text{rel}(\text{ext}((O))$ of $\text{ext}(O)$.
- ii. If f is non-local OFD, $base_{f,s(S)}$ is obtained as follows:

Let $\{O_1, \dots, O_m\} \subseteq V_f$ be the set of all sink object types and $? = \{Id_{O_1}, \dots, Id_{O_m}\}$ the set of their identifier attributes. For a node $O \in V_f$ let the set f_O of attributes contributed by O to $base_{f,s(S)}$ be defined as

$$\phi_O := \begin{cases} \delta \cup \gamma \cup \{Id_o\} & \text{if node label } v_f(O) = (\delta, \gamma) \\ \{Id_o\} & \text{if node label } v_f(O) \text{ is undefined} \end{cases}$$

(1) select a start node $O \in V_f$; $\mathcal{B} := rel(ext(O))[\phi_O]$

for all edges $e \in E_f$ incident on O with $\lambda_f(e) = r$:

$\mathcal{B} := \mathcal{B} \bowtie_{f_o} rel(ext(r))$; remove e from E_f

remove O from V_f

while $V_f \neq \emptyset$

select a node $O' \in V_f$ with $Id_{O'} \in \alpha_B$

$\mathcal{B} := \mathcal{B} \bowtie_{f_o} rel(ext(O'))[\phi_{O'}]$

for all edges $e' \in E_f$ incident on O' with $\lambda_f(e') = r'$:

$\mathcal{B} := \mathcal{B} \bowtie_{f_o} rel(ext(r'))$; remove e' from E_f

remove O' from V_f

(2) **right-hand side normalization (rs-normalization):**

If $t \neq \emptyset$, remove all tuples undefined on t :

$base_{f,s(S)} := WNF(\mathcal{B}, t)$

?'(G) is called the set of **source (sink) attributes**. WNF denotes, as previously introduced, the weak null filter.

The base relation $base_{f,s(S)}$ is unique and independent of the choice of nodes in step (ii)(1) of Definition 6, because no tuples are lost during a join operation and in each step a single join attribute exists, provided by an identifier attribute. Furthermore, each relational representation is used exactly once during the construction of $base_{f,s(S)}$. Thus, the join operations in (ii)(1) are always applicable.

The rs-normalization step removes tuples which are undefined on all identifier attributes that belong to a sink type. Such tuples represent no linkages and by discarding them no information about sink objects is lost.

Example 3 Consider OFD $f: \{A, O_3\} - G_f? \{O_2\}$ of schema S and the state from [Figure 6](#). The base relation $base_{f,s(S)}$ can be computed by the FOJ sequence implied by G_f when starting at node O_3 :

$$(((rel(ext(O_3))[Id_{O_3}] \bowtie_{f_o} rel(ext(r_2))) \bowtie_{f_o} rel(ext(O_2))[Id_{O_2}]) \bowtie_{f_o} rel(ext(r_1))) \bowtie_{f_o} rel(ext(O_1))[Id_{O_1}, A]$$

The result of the FOJ sequence is relation R which corresponds to relation R_4 from Example 2, projected onto attribute set $\{Id_{O_1}, A, Id_{O_2}, Id_{O_3}\}$. O_2 is the sole sink type of f . Thus, rs-normalization deletes the fifth and seventh tuple from R since both are undefined on the identifier attribute Id_{O_2} .

$$R = R_0[Id_{O_1}, A, Id_{O_2}, Id_{O_3}]$$

Id_{O_1}	A	Id_{O_2}	Id_{O_3}
i_1	1	i_4	—
i_1	1	i_5	i_9
i_2	1	i_6	i_{10}
i_2	1	i_6	i_{11}
i_3	2	—	—
—	—	i_7	i_{11}
—	—	—	i_8

$$base_{f,s(S)}$$

Id_{O_1}	A	Id_{O_2}	Id_{O_3}
i_1	1	i_4	—
i_1	1	i_5	i_9
i_2	1	i_6	i_{10}
i_2	1	i_6	i_{11}
—	—	i_7	i_{11}

$$base_{f',s(S)}$$

Id_{O_1}	A	Id_{O_2}	B	Id_{O_3}
i_1	1	i_4	2	—
i_1	1	i_5	3	i_9
i_2	1	i_6	4	i_{10}
i_2	1	i_6	4	i_{11}
—	—	i_7	4	i_{11}

$base_{f',s(S)}$ is the base relation obtained for f' : $\{A, O_3\}$ - $G_{f'}$? $\{B\}$ under the same state.

Although the base relation $base_{f,s(S)}$ corresponds to the set of all linkages of $s(S)$ with respect to f , $base_{f,s(S)}$ may contain ‘redundant’ linkages or linkages that possibly interfere with the evaluation of f on $base_{f,s(S)}$. If partial linkages exist in a state, different linkages for the same sink object may exist, where the combination of source objects in one linkage is less specific than in another linkage. In this case, one linkage—and thus the respective tuples in the base relation—subsumes the other. Such linkages represent redundant information and they would interfere with the evaluation of an OFD if additional partial linkages are to be taken into account for OFD semantics (cf. Rasch, 1998). They are removed from the base relation by a second normalization step, called *subsumption normalization*. Only objects (or object values) of the source and sink types are taken into account for this step. Objects of connecting types are not considered.

Definition 7 Let t and t' be partial tuples over attribute set β . t' **properly subsumes** t on β (in symbols: $t <_{\beta} t'$) iff $t =_{\beta} t'$ and $t|_{\beta} \neq t'|_{\beta}$. Let R be a partial relation over attribute set a_R and $\beta \subseteq a_R$. The **subsumption normalization (sub-normalization)** $snorm(R, \beta)$ of R on β is defined as

$$snorm(R, \beta) := \{t \mid t \in R \text{ and no } t' \in R \text{ exists with } t <_{\beta} t'\}$$

For a relation R and an attribute set β the result of sub-normalization is uniquely determined. Applied to a base relation, sub-normalization discards all linkages of a sink object whose source object combinations are properly subsumed by another source object combination for the same sink object. This results in a *validation relation* that can be used to check the validity of an OFD.

Two sets can be used for sub-normalization, corresponding to the difference between objects and values in object-oriented data models: On the one hand, sink types can be taken into account like they are involved in the right-hand side of an OFD, i.e., either on object level or on value level. On the other hand, the sink types may be used always on object level during normalization. In the first case, information about different sink objects having identical values may be discarded. This loss of possibly relevant information from the base relation is avoided in the second case (cf. Rasch, 1998).

Definition 8 Let S be an object schema, $s(S)$ be a state of S , and $?-G_f?$ Gbe an OFD with $?'$ being the set of identifier attributes of the sink types of f . $val_{f,s(S)} := snorm(base_{f,s(S)}, ?')$ $? G$ is the **validation relation** of f under $s(S)$. $val_{f,s(S)}^{str} := snorm(base_{f,s(S)}, ?' ? ?)$ is the **strict validation relation** of f under $s(S)$.

The validation relation is obtained by applying the normalization step to the source and sink attributes of $base_{f,s(S)}$. Thus, if a target of f is an attribute set of a type O , the normalization step towards the validation relation uses only value attributes of the sink type. Information about different O -objects with identical values may be lost, because the identifier attribute is ignored in the normalization step. However, with respect to the constraint stated by f no information is lost since at least one O -object with this common 'sink value' occurs in a linkage of the validation relation. The strict validation relation avoids this by including the identifier attributes of all sink types of f in the normalization step. Following from the definition, the validation relation $val_{f,s(S)}$ of an OFD f is always a subset of the strict validation relation $val_{f,s(S)}^{str}$ of f .

Example 4 Consider the base relations $base_{f,s(S)}$ and $base_{f',s(S)}$ for OFDs f and f'' from Example 3. To obtain $val_{f,s(S)}$, sub-normalization has to be applied to $base_{f,s(S)}$ on attribute set $\{A, Id_{O_2}, Id_{O_3}\}$. No subsuming tuples exist and thus, $val_{f,s(S)} = base_{f,s(S)}$. Since sink type O_2 is involved in f at type-level, $val_{f,s(S)}$ and $val_{f,s(S)}^{str}$ coincide. For f' , however, this is not the case. To determine $val_{f',s(S)}$, sub-normalization on $\{A, B, Id_{O_3}\}$ has to be applied to $base_{f',s(S)}$. By this the last tuple is discarded from $base_{f',s(S)}$:

$base_{f,s(S)} = val_{f,s(S)} = val_{f,s(S)}^{str}$					$val_{f',s(S)}$					$val_{f',s(S)}^{str} = base_{f',s(S)}$				
Id_{O_1}	A	Id_{O_2}	Id_{O_3}		Id_{O_1}	A	Id_{O_2}	B	Id_{O_3}	Id_{O_1}	A	Id_{O_2}	B	Id_{O_3}
i_1	1	i_4	—		i_1	1	i_4	2	—	i_1	1	i_4	2	—
i_1	1	i_5	i_9		i_1	1	i_5	3	i_9	i_1	1	i_5	3	i_9
i_2	1	i_6	i_{10}		i_2	1	i_6	4	i_{10}	i_2	1	i_6	4	i_{10}
i_2	1	i_6	i_{11}		i_2	1	i_6	4	i_{11}	i_2	1	i_6	4	i_{11}
—	—	i_7	i_{11}							—	—	i_7	4	i_{11}

The strict validation relation for f'' , however, is obtained by subnormalization on $\{A, Id_{O_2}, Id_{O_3}\}$. Therefore, the last tuple is not deleted and $val_{f',s(S)}^{str}$ coincides with $base_{f',s(S)}$. In general, the base relation and the strict validation relation do not coincide.

Transferring Semantics of Functional Dependencies

Using the concept of a validation relation, a semantics for OFDs can be defined. The most obvious way to do this is to simply adopt the meaning of an FD $g:\beta \rightarrow ?$ for OFDs: Each combination of values in the attributes of β in a given relation determines at most one $?$ -combination, i.e., g is a function mapping β -combinations to $?$ -combinations, with tuples being total. In the context of an OFD $f: ?-G_f?$ G this means looking only at $?'$ and

G , the source and sink attributes of $val_{f,s(S)}$ or $val_{f,s(S)}^{str}$, i.e., total linkages are taken into account and every target unit has to be reachable by using them. For each entry of τ , exactly one entry unit is specified. Only source- and sink-objects are considered, but not the connecting objects in linkages and the links between them although they are represented by the validation relations, too. This view corresponds to the following notions:

Definition 9: Let S be an object schema with state $s(S)$ and let $\tau - G_f \tau$ be an OFD of S . Let $G = \{\tau_1, \dots, \tau_m\}$ and $\{Id_{O_1}, \dots, Id_{O_m}\}$ be the set of identifier attributes of the sink object types of f . $\tau'(G)$ denotes the set of source (sink) attributes. f is **strongly O-satisfied** by $s(S)$ iff the following conditions hold:

f is **strongly satisfied** by $s(S)$ iff the following conditions hold:

with SNF denoting the strong null filter.

The *uniqueness requirement* (i) of strong O-satisfaction states that f induces a function, mapping each *total* combination of τ' -values of $val_{f,s(S)}^{str}$ to exactly one G -combination. The *surjectivity requirement* (ii) guarantees *reachability* or, regarding f as a function, *surjectivity* of each sink object solely by such total linkages: For each sink object at least one total combination of entry units has to exist via which it can be accessed.

For strong satisfaction, similar conditions have to hold. Both uniqueness and surjectivity requirement are defined with respect to the validation relation. In contrast to strong O-satisfaction the focus is on the target and not on the object type providing the target. Hence, condition (iv) does not necessarily guarantee reachability of each sink *object* in the given state but reachability of each target unit: If the target is an attribute set $\tau' attr_S(O)$ of a sink type O , not every O -object may be reachable by a total entry combination. However, for every target unit, i.e., for every combination of τ' -values occurring in the extension of O , at least one O -object with these τ' -values is reachable; and surjectivity is given with respect to the set of all τ' -values occurring in the values of O -objects. A surjectivity requirement in the sense of condition (ii), i.e., checking

reachability at object-level, would not be suited for strong satisfaction since information about some objects of a sink type O may have been discarded during the construction of the validation relation.

For FDs and local OFDs, surjectivity as introduced previously is given implicitly. If relationships are involved in an OFD, i.e., if the OFD is nonlocal, this is usually not the case.

Example 5 The state graph in [Figure 7](#) shows a state $s_1(S)$ of schema S , similar to state $s(S)$ in [Figure 6](#). The object values are listed next to the nodes. Consider OFD $f : \{A, O_3\} ? \{O_2\}$ from Example 3 and OFDs $g_1 : \{O_3\} ? \{A\}$ and $g_2 : \{O_2\} ? \{A\}$ of S . For f and g_2 the following validation relations are obtained from $s_1(S)$. They coincide with the base relations of f and g_2 , respectively:

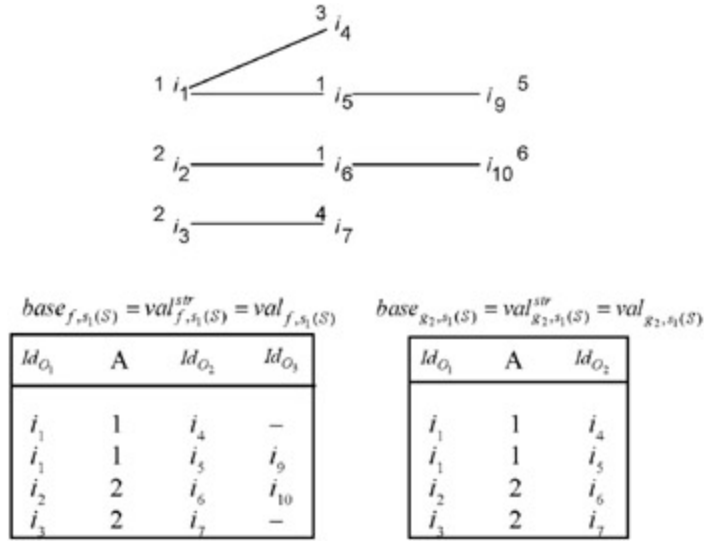


Figure 7: Example of a state

For f to be strongly O-satisfied, $t|_{\{A, Id_{O3}\}} = t'|_{\{A, Id_{O3}\}} ? \quad t|_{\{A, Id_{O2}\}} = t'|_{\{A, Id_{O3}\}}$ has to hold for every $t, t' ? SNF(val_{f, s_1(S)}^{str}, \{A, Id_{O3}\})$. The two tuples of $val_{f, s_1(S)}^{str}$ which are total on $\{A, Id_{O3}\}$ satisfy this condition. However, surjectivity requirement (ii) is violated since two tuples are discarded from $val_{f, s_1(S)}^{str}$ by the strong null filter on $\{A, Id_{O3}\}$. These tuples are the only ones in which sink objects i_4 and i_7 occur. Thus, f is not strongly O-satisfied by $s_1(S)$. Analogously, f is not strongly satisfied by $s_1(S)$: Surjectivity requirement (iv) is violated. OFD g_2 gives an example of an OFD that is both strongly O-satisfied and strongly satisfied: Requirements (i) and (iii) are obviously satisfied. No tuples are deleted by the strong null filter on attribute Id_{O2} and thus, (ii) and (iv) hold as well. OFD $g'_2 : \{B\} ? \{A\}$, obtained by a left-hand side change of g_2 to attribute-level, is neither strongly O-satisfied nor strongly satisfied since O_2 -objects i_5 and i_6 with identical B -values are connected to O_1 -objects with different A -values. Finally, dependency g_1 is strongly satisfied but not strongly O-satisfied. The following validation relations are obtained for g_1 under $s_1(S)$. The base relation $base_{g_1, s_1(S)}$ coincides with the base relation $base_{f, s_1(S)}$ for f .

$val_{g_1, s_1}^{str}(S)$			
Id_{O_1}	A	Id_{O_2}	Id_{O_3}
i_1	1	i_5	i_9
i_2	2	i_6	i_{10}
i_3	2	i_7	—

$val_{g_1, s_1}(S)$			
Id_{O_1}	A	Id_{O_2}	Id_{O_3}
i_1	1	i_5	i_9
i_2	2	i_6	i_{10}

The Id_{O_3} -values in $val_{g_1, s_1}(S)$ determine the A-values and both tuples are total on Id_{O_3} . Thus, no tuples are removed by the strong null filter and g_1 is strongly satisfied. $val_{g_1, s_1}^{str}(S)$, however, contains a tuple which is undefined on Id_{O_3} and therefore is deleted by the strong null filter. Because of this, g_1 is not strongly O-satisfied.

In general, the validation relation and the strict validation relation of an OFD which is neither strongly satisfied nor strongly O-satisfied or which is both strongly satisfied and strongly O-satisfied do not coincide. It can be shown that strong O-satisfaction of an OFD f implies strong satisfaction of f . Strong satisfaction, however, does not imply strong O-satisfaction as demonstrated in the example above.

Beyond Semantics of Functional Dependencies

The notions of strong satisfaction require unique reachability of each sink object or value thereof, respectively, by total entry combinations. Less restrictive semantics for OFDs are of interest, too. For example, uniqueness of entry combinations with respect to the reachable sink objects may be sufficient, ignoring sink objects which cannot be reached from any total entry combination.

Definition 10 Let S be an object schema with state $s(S)$ and $f: ? - G_f ?$ Gbe an OFD of S .

f is **weakly O-satisfied** by $s(S)$ iff condition (i) from Definition 9 holds. f is **weakly satisfied** by $s(S)$ iff condition (iii) from Definition 9 holds. Because surjectivity requirements are discarded, these semantics impose a restriction only on the reachable sink objects.

Strong (O-)satisfaction implies weak (O-)satisfaction and weak O-satisfaction implies weak satisfaction. The converse implications do not hold in general. [Figure 8](#) summarizes the relationships between the OFD semantics that regard total linkages, with strong O-satisfaction as the most restrictive notion of satisfaction.

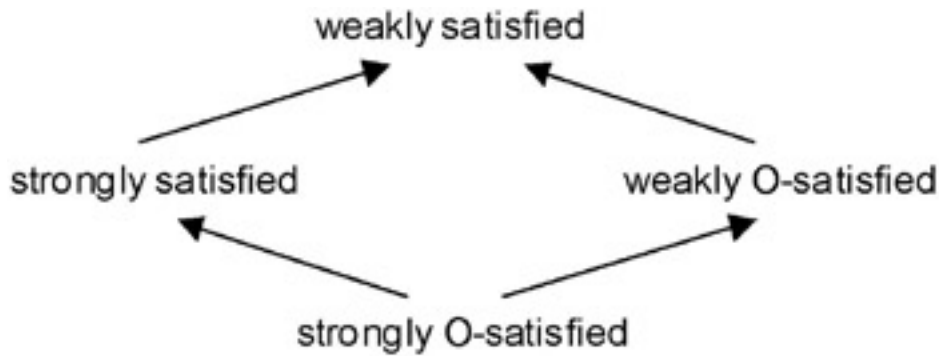


Figure 8: Implications between OFD semantics

Partial linkages can be taken into account for an OFD as well. In this case, total as well as partial entry combinations are considered for the access to sink objects. Sink objects which are not reachable from any total combination of entry values may be uniquely reachable from a partial entry combination. From this observation, four more semantics for OFDs result which are counterparts of the semantics presented previously (Rasch, 1998; Klein & Rasch, 1997).

As in the case of functional dependencies in the relational data model, inference rules are of interest for OFDs in order to derive dependencies implied by a given set of OFDs. In addition to generalizations of the well-known rules for FDs, more sophisticated rules are needed for OFDs. Such rules have to take into account the possible change between object level and value level, modifications of OFD graphs, and information given by the schema itself, like inheritance hierarchies or cardinality constraints, restricting numerically the participation of objects in relationships. Obviously, surjectivity may be too restrictive a requirement to maintain if an OFD graph is extended by applying an inference rule or if two OFD graphs are merged. To enforce the uniqueness requirement, restrictions on the OFD graphs or a weakening of the OFD semantics may be necessary, especially if partial linkages or OFDs with more than one sink type are considered.

RELATED APPROACHES

Key and Uniqueness Constraints in Semantic Data Models

Almost every semantic data model allows the specification of key constraints for object types. In a number of approaches they are restricted to local attributes. But even if non-local keys are allowed, a one-to-one assignment between objects and value combinations of keys is frequently assumed. Concerning value based identification of objects this is an unnecessary restriction. Often it arises from the intention to replace internal object identifiers by value combinations of keys, for example, as a preparation for an implementation of an object schema in a relational database system. In the following, some of these approaches are discussed.

Obviously any identifier key or object key, i.e. primary keys for object types, can be expressed by a local OFD. The simplest kind of non-local identification constraints found

in semantical data models like the ER model are *weak entity types* (Chen, 1976). A weak entity type W is connected by a relationship type R to a *strong entity type* S , i.e., for S a local key consisting of attributes of S exists. In every state each W -entity is connected to exactly one S -entity. Hence the combinations of key values of an S -entity can be used to identify the associated W -entity. If more than one W -entity is related to a single entity of S , the key provided by S has to be extended with one or more attributes of W , whose values are unique with respect to the W -entities associated to the same S -entity. If W is related to another weak entity type, the weak entity constraint may consist of a sequence of relationship types until an entity type is reached for which a local key is given. Batini, Ceri, and Navathe (1992) generalize this kind of non-local identification of entities by linear constraints to *identifiers*, non-local constraints which correspond to trees of depth one, if the type for which a key is desired, is regarded as root: An identifier for an entity type E is a minimal, non-empty set $\{A_1, \dots, A_n, E_1, \dots, E_m\}$ with $n, m = 0$ and $n+m = 1$. A_1, \dots, A_n are attributes of E with domains of atomic values. No null values may occur in an entity of E with respect to these attributes. E_1, \dots, E_m are entity types which are connected to E by binary relationships R_1, \dots, R_m . The cardinality constraint for E in each R_i is restricted to (1,1), i.e. for each E -entity in a state exactly one combination of values and entities from $\{A_1, \dots, A_n, E_1, \dots, E_m\}$ exists. This one-to-one assignment between identifying combinations of values on the one hand and entities on the other hand arises naturally in the context of local keys. For non-local identification constraints this is not required. OFDs do not limit the number of entry combinations which lead to a single target unit.

An identifier of an entity type is called *internal* if $m = 0$; it is *external* if $n = 0$ holds and it is *mixed* if $m > 0$ and $n > 0$ hold. Using these notions, a weak entity type as discussed previously is a type for which only external or mixed identifiers are given. An identifier for an entity type corresponds to a strongly satisfied, canonical OFD f , where E is the sink type, involved in f at type-level. If $n > 0$ holds, E is also a source type. Otherwise, only types E_1, \dots, E_m occur as sources of f . In this case, f is a purely object-based OFD and provides an identification criterion for E . If the restriction of cardinalities is not waived, OFDs like the one shown in [Figure 4\(a\)](#) cannot be expressed by means of identifiers. Moreover, dependencies that involve partial entry combinations are not covered by identifiers. Batini et al. (1992) note that circularity in the definition of identifiers has to be avoided. They argue that for an entity type E which has only external identifiers, identifiers can be obtained by replacing types in the external identifiers with internal identifiers, if such exist.

A similar approach is made in the context of the functional data model FDM (Shipman, 1981) by P/FDM (Paton & Gray, 1988), a Prolog-based implementation of the functional data model. The building blocks of P/FDM are *classes* and *functions*. The latter are used to represent attributes as well as relationships between classes. A key for a class C consists of a non-empty set of single-valued functions of the class, i.e. the concept of a key in P/FDM corresponds to the concept of an identifier in the ER model: A one-to-one assignment of combinations of key values to objects is required, and the key of C consists of attributes of C or classes which are connected to C by relationships, or both. Again, classes in a key definition have to be replaced to obtain a purely attribute-based key, once

more leading to keys of tree structure. Hence the remarks on OFDs and identifiers in the ER model apply for keys in P/FDM as well.

Nienhuys-Cheng (1990) introduces *key constraints* for *nolots* in *binary semantical networks*, which are conceptual database schemes specified by means of the *Nijssen Information Analysis Method* (NIAM, see e.g., Verheijen & van Bekkum, 1982). Nolots (non-lexical object types) correspond to object types and are connected by binary relationships called *relations*. A key constraint f for a nolot A is given by $n = 1$ paths that start with A and lead to nolots or *lots* (lexical object types) B_1, \dots, B_n . f is a tree with root A and leafs B_1, \dots, B_n . Every instance of A has to be reachable from a (B_1, \dots, B_n) -combination. Moreover at most one combination may exist for a single instance, i.e. exactly one key combination is assigned to each instance. Since only binary relationships are considered, a binary semantical network can be regarded as an object schema and a key constraint can be expressed by an OFD with entries B_1, \dots, B_n and target A . If $\{B_1, \dots, B_n\}$ contains a nolot, f corresponds to an identification criterion, otherwise it is a VBIC for A . As in the case of identifiers, OFDs in which relationships with cardinalities different from (1,1) occur, cannot be expressed by these key constraints. Only total key combinations are taken into account.

Van Bommel, ter Hofstede, and van der Weide (1991) introduce the *Predicator Model* as a formalization for *object-role models*. The authors point out the ER model, NIAM, and functional data models like FDM as examples for data models of this family. The building blocks of the Predicator Model are *predicators*, which are pairs consisting of an object type and a role. Relationships are modeled as sets of predicators and may have an arity greater than two. Moreover, complex objects, i.e., nested types, are supported. For the Predicator Model, the notion of a *uniqueness constraint* $\text{unique}(s)$ is introduced as a set s of predicators in a Predicator Model schema. Let s' be the remaining predicators of the relationships from which s is taken. Then, intuitively speaking, $\text{unique}(s)$ is satisfied by a state, if each combination of objects belonging to the types addressed by s functionally determines the combination of objects belonging to the types addressed by s' . The predicators in s have to satisfy conditions concerning connectivity in order to be valid constraints. The semantics of such a uniqueness constraint is given by a relation obtained by joining relations that represent the object types participating in s (van der Weide, ter Hofstede, and van Bommel, 1992). However, in general only those objects are taken into account which participate in a relationship, whereas a base relation of an OFD f considers all objects of the sink types of f . The semantics of $\text{unique}(s)$ is related to the notion of weak identification with only total entry combinations and total combinations of target units being considered. This corresponds to a 'link-centered' view of a state where only reachable objects are considered. Thus, if we ignore relationships of higher arity, only weakly satisfied OFDs can be specified by uniqueness constraints. An example for such a constraint is the dependency from [Figure 4\(b\)](#) with respect to weak satisfaction: Assume the constraint represents information about the actual daily assignment of consultants to customers. For a customer of a branch, his assigned consultant may change temporarily if his primary consultant is unavailable. Thus, *Clerk* objects may exist in a state which are not reachable by any total or partial entry combination, and weak satisfaction as OFD semantics would be a suitable choice.

Employees which are not assigned to any customer are not of interest in this context. Partial entry combinations are not considered by van Bommel et al. (1991). The OFD approach can be extended to cover relationships of higher arity as well.

Ter Hofstede and van der Weide (1993) introduce the *Predicator Set Model* to extend NIAM and the Predicator Model with set types. Identification constraints analogously to those of Nienhuys-Cheng (1990) and van Bommel et al. (1991) are considered, where set-valued attributes are allowed. Object types with cyclic, i.e. recursive, type structures are allowed but they are not identifiable by a fixed set of properties.

Mok and Embley (1996) and Embley (1998) use *co-occurrence constraints* in the context of Object-oriented Systems Analysis (OSA, see e.g., Embley, Kurtz & Woodfield, 1992) and the Object-oriented Systems Model (OSM, cf. Embley, 1998). OSA, too, belongs to the family of object-role models and is the predecessor of OSM. The building blocks of OSM schemas are *object sets* and *relationship sets* which connect object types and may be binary or of higher arity. Object sets are either *non-lexical* sets representing objects, or *lexical* sets representing atomic values. Co-occurrence constraints are related to uniqueness constraints of the Predicator Model. In contrast to these they refer only to a single relationship set. A co-occurrence constraint with respect to a relationship set R is a

dependency with $n, m = 1, k_1 > 0, \{A_1, \dots, B_m\} = \emptyset$,
 where each A_i, B_j denotes an object set participating in R . It indicates that in each state a single $\{A_1, \dots, A_n\}$ -combination can occur in at least k_1 and most k_2 instances of R with different $\{B_1, \dots, B_m\}$ -combinations. For $(k_1, k_2) = (1, 1)$ the constraint corresponds to a generalized FD and the previous remarks on OFDs as well as the remarks on uniqueness constraints in object-role models and the link-centered view of a state apply.

The idea of using relationships to determine keys for record types in a network schema was applied by Zaniolo (1979). For a record type T , a set of *synonyms* is determined by taking set types into account. Each synonym of T may consist solely of *data items*, i.e., attributes, of T itself. If the local data items provide no unique identification of T -records, a non-cyclic sequence of set types, i.e., relationships starting with T and leading to a record type T' may be used. T' as well as any other record type of the sequence may contribute data items for the identification of T -records. Optional set types may participate in the sequence and thus missing links are taken into account, too. In this case, the sequence provides a *pseudo synonym* for T . A synonym of T corresponds to a canonical, strongly satisfied OFD f where the OFD graph is a path. Since only a single path emerging from T is considered, OFDs like the dependency f from Example 3 cannot be expressed by a synonym. In G_f the source types are not nodes of the same path starting at the sink type.

Schewe and Thalheim (1993) apply the use of non-local dependencies with path structure to an object-oriented data model in order to obtain a VBIC for a class. Similar to the functional data model, a relationship from a class C_1 to a class C_2 is represented by an attribute of C_1 with type C_2 . It is denoted as a *reference* from C_1 to C_2 . A class C is *value identifiable*, if a set of attributes with basic types is given for C , by which C -objects can be identified, i.e., a local VBIC for C exists. Non-local identification is considered, too. A

class C is *weakly value identifiable* if a value identifiable class C' and a sequence of classes C_0, \dots, C_n , $n > 0$, exists such that $C' = C_0$, $C = C_n$ holds, and either a reference from C_{i-1} to C_i exists, or C_i is a subclass of C_{i-1} . Moreover, each of the references has to comply with a *surjectivity requirement*: In every state, each object of class C_i has to be referenced by an object of C_{i-1} , i.e., all C_i -objects have to be reachable from C_{i-1} -objects. Under these prerequisites, every C -object is reachable from at least one C' -object, and the combinations of key values of C' -objects can be used to uniquely access C -objects. The local VBIC of C' provides a non-local VBIC for C . Moreover, a single C -object o may be reachable from several C' -objects and thus, more than just one combination of key values may exist for o . The VBIC for C may not be composed, and solely consists of the VBIC of C' . In contrast to the previously discussed synonyms in a network schema, neither other types involved in the sequence nor C itself may contribute attributes to it. Hence, dependencies like the OFD from [Figure 4\(a\)](#) or the OFDs from [Figure 5](#) are not covered. Partial entry combinations are not addressed because only a single entry is considered.

Demanding the surjectivity requirement to hold for *every* reference occurring in the sequence imposes an additional restriction on identification. For C to be weakly value identifiable it is sufficient that all C -objects are reachable from C' . The existence of additional objects of the classes C_1, \dots, C_{n-1} , which are not reachable from any C' -object, does not interfere with the identification of C -objects. For this reason, the notion of strong satisfaction of OFDs requires surjectivity only with respect to the sinks of an OFD but not with respect to all connecting types. As far as identification is concerned, permitting only surjective references imposes an unnecessary restriction. If the differences in the data models and the surjectivity requirement for connecting types are ignored, a class C which is weakly value identifiable by a class C' is comparable to a strongly satisfied OFD f with a single sink type C and a single entry d that constitutes the entry for the local VBIC of C' . The graph of f is a path connecting nodes C and C' .

Generalizations of Functional Dependencies

Occasionally, generalizations of FDs are introduced which allow not only the specification of keys but also of other dependencies between object types and attributes, or, as a direct application of relational FDs, between attributes of a single type. In the following, some of these approaches which often aim at the development of normal forms for object types or schemas, are compared with OFDs. In principle, all these key specifications and dependencies are expressible in the framework of OFDs. A proposal deviating from this common framework is discussed at the end of this section.

The normalization of entity types and relationship types of an ER schema has been addressed, for example, by Chung, Nakamura, and Chen (1981), Ling (1985) and Ling and Teo (1994). Rauh and Stickel (1996) define normal forms for entity types and relationship types by simply transforming them into canonical relation types, which are induced by the attributes of the entity types or, in the case of a relationship type R , by the local attributes of R and the key attributes of entity types participating in R . The normal forms for entity types and relationship types are obtained by requiring the relational normal forms for the canonical relation types. By this, FDs are implicitly generalized to

entity types and relationship types. This corresponds to strongly satisfied local OFDs. For binary relationships this is similar to simple non-local OFDs. However, only entities participating in the relationship are taken into account, i.e. the link-centered view of a state has to be employed. Hence, strongly satisfied OFDs like dependencies g_1, g_2 from Example 5 cannot be expressed. The sketched approach illustrates that the normalization of ER schemas or of types thereof often is guided primarily by the principles of the relational normalization. Ling (1985), for example, defines normal forms for entity types and relationship types in a way similar to Rauh and Stickel (1996), but takes nested attributes and multi-valued dependencies into account, too.

Lee (1995) introduces a restricted generalization of FDs to *object classes*, denoted as *object functional dependencies*. Note that for the following the abbreviation ‘OFD’ will refer to the notion of dependency introduced previously, whereas ‘object functional dependency’ will refer to the notion introduced by Lee. An object class is specified by a class name and a set of attributes. An attribute may be of basic type, of collection type (e.g. a set type or array type), or it may be of class type, i.e. it represents a reference to an object of another object class. An object functional dependency $A_1 \bullet A_2$ is defined with respect to a single object class C by two attributes A_1 and A_2 of the attribute set of C . The dependency is satisfied by a state if for each object o of C the A_1 -value of o determines the A_2 -value of o as it is known from FDs. The object identifier may be one of the attributes occurring in an object functional dependency. Except for the involvement of attributes of collection type and the representation of relationships by reference attributes, the dependency introduced by Lee corresponds to a strongly satisfied local OFD or, if A_2 is an attribute of class type C' , is comparable to a simple non-local OFD consisting of two nodes for the object classes, with C' as sink type, and an edge representing the reference between them. In this case, the OFD would have to be weakly satisfied since not every C' -object may be referenced by an object of C . More general weakly satisfied dependencies, as discussed above for the OFD from [Figure 4\(b\)](#), are not covered by this approach. Lee uses object functional dependencies to introduce an *object normal form* for object classes. The examples for violations of the object normal form which Lee states, correspond to violations of normal forms as known from the relational model. As mentioned, Lee takes the internal identifier attribute, i.e. the object class itself, into account, analogous to the use of object types in OFDs. The normalization approaches for the ER model use externally visible keys. Of course, both approaches provide uniqueness. However, the use of the object type in dependencies emphasizes the difference between objects and values.

Wijzen, Vendenbulcke, and Olivie (1994) investigate FDs for an object oriented temporal data model and introduce *snapshot functional dependencies* for this purpose. A snapshot functional dependency (SFD) $C(X \rightarrow Y)$ is defined with respect to a single class C and is comparable to an object functional dependency as used by Lee (1995): Attributes of class type are allowed. Both X and Y are subsets of the attribute set of C and the identifier attribute may occur in the dependency. X and Y , however, are not restricted to singletons. The semantics of an SFD is analogous to that of an FD; a single state is considered and the equality of two C -objects on X implies the equality on Y . Hence, as in the case of object functional dependencies, an SFD corresponds to a local OFD or a simple non-local

OFD with two types being involved. Furthermore, *dynamic* and *temporal functional dependencies* are considered. The satisfaction of these constraints is defined regarding two or more subsequent database states.

An approach to normalization completely different from those discussed previously, which intend to avoid anomalies, is made by Tari, Stokes, and Spaccapietra (1997). They introduce *path dependencies*, *local dependencies*, and *global dependencies* for an object-oriented data model which includes classes, object identity, inheritance, relationships by means of bidirectional, class-typed attributes, and complex objects, i.e. classes with attributes of set type or tuple type in arbitrary nestings. The model combines the features of an extended ER model and nested relations as known from the NF₂ data model (non first normal form, see e.g. Scholl & Schek, 1986). As in the previous two approaches and in the OFD approach, the identifiers of objects are taken into account by the dependencies. The three kinds of dependencies are all defined with respect to paths or walks of a schema, i.e. cycles are allowed. Local and global dependencies are constraints on single classes and consider paths within a class C , having a complex nested type. The first one is used to specify FD-like restrictions on the nested value of each single object of C in a state: For two attribute sets X and Y of C , connected by a given path within the complex type of C , exactly one combination of Y -values is reachable from each combination of X -values within the object value of an object o of C . Global dependencies extend local dependencies to hold not only for each *single* instance, but also with respect to *all* instances of C : The local dependency between X and Y has to hold and additionally for any two objects of C which coincide in their X -values, the Y -values have to coincide. To some extent, OFDs are related to path dependencies which state constraints between classes C_1 and C_2 of a schema, connected by a path $?$. Unlike OFDs or FDs, a path dependency specifies a constraint on every single object and not on pairs of objects: For each C_1 -object the C_2 -objects (or attribute values thereof, if specified so by the dependency) which are reachable via $?$ have to coincide. Because only reachable C_2 -objects are considered, this is similar to the notion of weak satisfaction of OFDs.

In the context of an object-oriented data model that supports object identity, tuple-valued objects, and class-typed attributes for directed references between objects, Weddell (1990, 1992) introduces *path functional dependencies* as a generalization of functional dependencies and key constraints. Path functional dependencies, like OFDs, allow the specification of non-local constraints. Since a sound and complete set of inference rules is given for these dependencies, we will discuss this approach in more detail. A path functional dependency is defined with respect to a *class scheme* of a schema S . S is given by a finite set of class schemes, where each class scheme is of the form $C\{P_1:C_1, \dots, P_n:C_n\}$. Each P_i is a *property*, i.e. attribute, of C and each C_i is the name of another class scheme of S , the type of P_i . Cyclic references among class schemes are allowed. Basic types are considered to be ‘trivial’ class schemes with $n = 0$, e.g. $Integer\{\}$. A *state* s for S is a directed, edge and node-labeled graph, where the nodes, labeled by their class name, correspond to objects, i.e. identifiers thereof, or basic values. Edges are labeled with property names and connect an identifier with the components of its object value. Schema S , too, can be read as a directed graph $G(S)$. This is similar to the notions of state graph and schema graph used above. Under this view, a set of *path*

functions is associated with S . It consists of all finite directed walks, i.e. sequences of properties, in $G(S)$. Every state s has to comply with the following restrictions:

- i. *property value integrity*: If (u,v) is an edge in s , labeled with property P , then u is labeled with a class name C such that P is a property of C , and v is labeled with the type of P in C . This guarantees that s corresponds to the structure of S .
- ii. *property functionality*: If (u,v) and (u,w) are edges in s , both labeled with property P , then $v = w$ holds, i.e., properties are single-valued.
- iii. *property value completeness*: If u is a node of s , with C being the label of u , then an edge (u,v) labeled with P exists in s for each property P of C , i.e., no missing values are allowed.

Especially the last two requirements enforce a tight relationship between paths in $G(S)$ and paths in s : If pf is a path function connecting class schemes C_1 and C_2 , and if u is an object node in s representing an object of C_1 , then exactly one path exists in s which connects u to an object node belonging to C_2 . Otherwise, one of the requirements would be violated. As Weddell notes, the data model corresponds to a restricted nested relational model.

A path functional dependency (PFD) over schema S , denoted by,

refers to a single class scheme C and path functions of it. Each $pf_i, i \in \{1, \dots, n\}$, is a path function starting at C . We will denote C as the *center class* of the PFD. A *key PFD* is denoted by $C(pf_1 \dots pf_m \mid Id)$, where Id is the path function of length zero, referring to C itself. A PFD is *satisfied* by a state s of S , if for any two nodes u, v in s labeled with C , i.e. u and v are both objects of C , the following condition holds: if $u.pf_i = v.pf_i$ for each $i \in \{1, \dots, m\}$ then $u.pf_j = v.pf_j$ for each $j \in \{m+1, \dots, n\}$. Here $u.pf_i$, for example, denotes the object or value which is reachable from u by following the path pf_i in s . Analogously, $u.Id$ refers to u itself. Note that due to the restrictions exactly one object or value node is reachable for each C -node and each path function.

Weddell (1990, 1992) does not consider different notions of identification. If we ignore for a moment the characteristics of the data model and the restrictions on states, the question is raised how PFDs and OFDs are related. To check the satisfaction of a PFD with respect to a state s , all objects of the center class C are inspected. This can be regarded as a surjectivity requirement with respect to C . Hence a key PFD $C(X \mid Id)$, with X being a set of path functions, is comparable to a strongly satisfied OFD $X \mid \{C\}$ which states an identification criterion for C . The path functions in the left-hand side of a key

PFD may lead to properties which are not of basic type. Because of this, a key PFD is no VBIC for its center class in general. Analogously, a PFD $C(X? Y)$ where Y is a subset of the properties of C , is comparable to an OFD $f: X? Y$. Since every object of C is considered by the PFD, f has to be strongly O-satisfied. In the same way, any strongly satisfied, canonical OFD can be simulated by a PFD. A PFD $C(X? Y)$ where Y contains ‘non-local path functions’, i.e., path functions which do not represent properties of C , cannot be expressed by an OFD. For this a ‘mixed semantics’ is necessary that takes every C -object into account, i.e. enforces surjectivity regarding C but requires uniqueness only with respect to the reachable Y -combinations. Note, however, that under the restrictions imposed on s , reachability of C -objects from X -combinations is always given. Roughly speaking, a PFD demands that each total X -combination in s leads via C -objects to at most one Y -combination. A (strongly satisfied) OFD $X? Y$ demands uniqueness, too, but additionally requires reachability of all Y -combinations in s . None of the connecting types is emphasized by it. In the same way, certain kinds of OFDs cannot be simulated by PFDs. Consider, for example, a non-canonical OFD $? ? G$. Since G -combinations are to be determined, G has to become the right-hand side of a corresponding PFD: $C(? ? G)$. This raises the question how to choose the center class C for the PFD. Because surjectivity is requested with respect to it and because an OFD imposes no restriction on connecting types, it has to be a sink type of f . This, however, results in an imbalanced treatment of sink types. For C , reachability of every object is given independent of an attribute-level involvement of C in G , whereas for any other sink type only the reachable target units are considered. Due to the prerequisites for states, it is evident that PFDs cannot simulate any OFD under any notion of identification which considers partial linkages. Analogously, no partial sink combinations are allowed. However, the examples from [Figure 1\(c\)](#) and [Figure 2](#) illustrate that in the context of inheritance hierarchies or exclusive-or constraints partial linkages have to be taken into account for the identification of objects.

There are substantial differences in the ‘PFD approach’ and the ‘OFD approach’. They result from the difference in the structure of dependencies (PFDs being grouped around a center class), and from the tight coupling of schema paths and state paths in the case of PFDs. Both have far-reaching implications on inference rules. The specification of a sound and complete rule set for PFDs relies on these requirements. They are notable restrictions which allow to obtain inference rules that do not hold for OFDs in general. If similar restrictions are required to hold for an admissible state of an object schema, the use of partial entry combinations for identification would be excluded, although they arise, for example, naturally in the context of inheritance hierarchies. Thalheim (2000) outlines an extension of PFDs to the ER model, where paths in a state are considered that fit to the schema paths.

Identification and Distinguishability

The focus of value based object identification is on the unique access to single objects. A different view is employed by value based *distinguishability* of objects. Here the focus is to decide whether two given objects are the same or not. This usually is done by inspecting the value of an object, its links, and the values of objects reachable along links.

In the following, a few approaches are discussed which are primarily concerned with property-based distinguishability of objects.

Abiteboul and Van den Bussche (1995) investigate the distinguishability of objects for an object-oriented data model with object values being basic values or object identifiers, or tuples of these. Thus, an object value may contain references to other objects of a state. Object identifiers as employed by Abiteboul and Van den Bussche correspond to abstract identifiers, i.e., they are not visible. Hence, two given objects can be distinguished only by the basic values occurring in their object value and by recursively dereferencing object identifiers in the object values and looking at the basic values of objects which are reachable by this. For an object o , this unfolded complex object value can be regarded as a tree $tree(o)$ which may be infinite if cyclic references occur between objects of a state. In any case, a tree is obtained where the leafs correspond to basic values. Two objects o_1 and o_2 are indistinguishable, or *deep equal*, if $tree(o_1) = tree(o_2)$ holds. Abiteboul and Van den Bussche show this notion of distinguishability to be equivalent to the *coarsest value based equivalence relation* on object identifiers and to distinguishability by *observation formulas* which are queries of a value based calculus language, called *observation calculus*. Under the first notion, a given equivalence relation on identifiers is extended to object values inductively in the following way: Each basic value is equivalent to itself and tuple values of the same arity are equivalent if they are equivalent with respect to each component. Using this notion, two identifiers are equivalent if and only if their object values are equivalent. This corresponds to the notion of *similarity* of objects, used by Denninghoff and Vianu (1993). The second notion of distinguishability relies on the observation calculus, a value based query language where equality comparisons on identifiers are not permitted. Variables in observation formulas range over basic values and identifiers. The only comparison operator is the equality of basic values. Queries are built inductively from these simple comparisons by conjunctive combination and negation. Moreover, existential quantification is allowed which also takes the dereferencing of identifiers into account. Details are given in Abiteboul and Van den Bussche (1995). All three notions are defined with respect to the complete unfolded value of an object and hence do not consider a fixed entry set for the value based access to object. The latter is one of the main motives for introducing OFDs. Due to this, these notions in general are neither suited nor intended for the support of identifying access to objects.

A related approach is presented by Kosky (1995, 1996). He addresses distinguishability in an object-oriented data model which supports object identifiers and reference attributes, including possibly cyclic references between objects of a state. Kosky focuses on the distinguishability of two different database *states*, not on the distinguishability of *objects* within a single state. For this, isomorphisms between states and an equivalence relation between states, called *bisimulation correspondence*, which corresponds to the value based equivalence relation employed by Abiteboul and Van den Bussche (1995), are investigated. Two states s_1 and s_2 are *isomorphic* if they differ only in object identifiers, i.e., s_1 can be obtained from s_2 by renaming the object identifiers of s_2 and vice versa. Bisimulation of states is defined analogously to the value based equivalence among object identifiers used by Abiteboul and Van den Bussche (1995). Two states s_1 and s_2

are *bisimilar* if for each class C and each object identifier which belongs to the extension of C in s_1 , an equivalent object identifier exists in the extension of C in s_2 and vice versa. Distinguishability of states can also be viewed in terms of query languages. Kosky shows that isomorphism of states is equivalent to the indistinguishability of states by means of a query language which allows equality tests on object identifiers, i.e. intuitively speaking two states are isomorphic if and only if every query of the considered languages yields the same result for both states. Bisimilarity corresponds to indistinguishability of states by the same query language without allowing equality tests on identifiers.

Isomorphism and bisimulation are the finest and coarsest, respectively, levels of distinguishability of objects. In addition to this, Kosky considers local and non-local keys for classes. Using these, two objects of a class are taken to be equivalent if and only if they coincide in their key values. In this context, acyclic keys, i.e., VBICs as introduced by OFDs, are considered since in general these provide an efficient means to compare nested object values without examining object identifiers directly or unfolding the complete object value.

Beeri and Thalheim (1999) introduce several notions of distinguishability for objects by regarding a state as a graph given by the objects, the object values and directed references between them. In this framework, notions of identification based upon homomorphisms (H-identifiability), and automorphisms (A-identifiability) of graphs, bisimulations (B-identifiability), values (V-identifiability), equations (E-identifiability), and queries (Q-identifiability) are investigated. It is shown that some of the notions coincide (e.g. E-identifiability and V-identifiability) or imply other ones (e.g. Q-identifiability implies V-identifiability). The various notions exploit differences in the graph structures of objects or the existence of different basic values in object values. This does not necessarily provide identifying *access* to objects, especially not access via the same set of entries for each state as it is proposed by OFDs and similar approaches. The attributes providing those different values or ‘structural aspects’ of the graph of o which allow a distinction, may vary from state to state. Beeri and Thalheim, like Abiteboul and Van den Bussche (1995), and Kosky (1995, 1996) do not consider inheritance hierarchies.

CONCLUSIONS AND FUTURE RESEARCH

In the relational data model, real world objects and their relationships are modeled by data values using a single concept, the relation. The set-orientation of this model allows to identify facts stored in a database by keys which are explicitly given or which can be derived from a set of functional dependencies by applying inference rules. In data models offering a richer set of modeling constructs such as object-oriented data models, identification is a much more involved problem which is not solved at the conceptual level by the concept of object identifiers (cf. Beeri, 1990). From a practical point of view, value based identification of objects should be possible independent of their identifiers. To tackle this problem, a framework for the specification of so-called object functional dependencies between attributes and object types was presented in the context of a simple object-oriented data model. Different identification mechanisms can be defined using these constraints. We concentrated on the unique access by means of total entry

combinations and reachability of all target units. By modification of this ‘fundamental view’ on the identification of objects, several notions of identification are obtained, arising from the use of partial entry combinations and from ignoring objects which are not reachable by entry combinations. These notions are of particular relevance if inheritance hierarchies or exclusive-or constraints are taken into account as demonstrated by examples in the introduction.

For the notion of strong satisfaction, inference rules have been given in Rasch (1998). However, the rule set presented there is not complete and it seems to be a non-trivial task to achieve completeness. What has to be taken into account is that the structure of a given schema, especially inheritance hierarchies and cardinality constraints for relationships, may have influence on the set of dependencies derivable from a given set of OFDs. These interactions are far more complicated than what is known for functional dependencies in the relational data model. Furthermore, both kinds of satisfaction, strong and weak, must be considered even if only dependencies with respect to strong satisfaction shall be derived: It may be possible to derive a strongly satisfied OFD $f_3: ?? \rightarrow F$ from a weakly satisfied OFD $f_1: ?? \rightarrow G$ and a strongly satisfied OFD $f_2: G \rightarrow F$ because for f_3 reachability is only required with respect to F but not with respect to G .

In the definition of OFDs, cycles have been excluded in order to keep things simple. Nevertheless, cyclic dependencies are of interest in practice and should be covered in the framework. They also have to be considered for a complete set of inference rules since cycles may result from the application of inference rules. As an example, consider two OFDs $f_1: ?? \rightarrow G, f_2: G \rightarrow F$ for the same schema with f_1 (f_2) having more than one sink (source) object type. Then a new cyclic dependency could be derived from f_1 and f_2 by transitivity.

From a practical point of view, the topics of efficiently checking and enforcing OFDs in object-oriented (and object-relational) databases become a major issue. Integrity constraints arising from the structure of an object schema, e.g., from relationships between types, or constraints implied by an inheritance hierarchy are maintained automatically by the type system of an object-oriented database system since the schema can be directly represented by features of the system. Any further constraints, e.g., additional cardinality constraints, key constraints, or functional dependencies, are often specified by means of a declarative language or in a semi-procedural style, and are coded directly by methods of the respective object types. Another possibility is given by rewriting those methods of an object type, which realize insert or update operations. In most cases, only local or simple non-local constraints, similar to the weak entity concept, are considered. Hence the question is raised, how to check more general non-local dependencies like OFDs, and which kind of OFDs can be efficiently maintained in available object-oriented database systems. Kemper and Moerkotte (1992) propose *access support relations* for the support of query processing in object-oriented databases. An access support relation materializes chains of links between objects in order to support a path in a schema which is frequently addressed in queries. This is related to the approach of Bernstein, Blaustein, and Clarke (1980), where in a relational database

redundant data is materialized to support the efficient checking of integrity constraints, and hence may be helpful for the maintenance of OFDs, too.

A frequent case is the object-oriented design of a database schema combined with the use of a relational database system for implementation. An interesting question is how OFDs and value based identification criteria derived from them can be taken into account for the transformation of an object schema into a relational schema. It is a challenge to develop methods for generating relational implementations which reflect the given constraints in such a way that their maintenance is well supported.

REFERENCES

- Abiteboul, S. & Kanellakis, P. C. (1989). *Object identity as a query language primitive*. In J. Clifford, B. G. Lindsay & D. Maier (Eds.), *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, ACM SIGMOD Record*, 18(2), 159–173.
- Abiteboul, S. & Van den Bussche, J. (1995). *Deep equality revisited*. In *Proceedings Deductive and Object-Oriented Databases, Fourth International Conference, DOOD'95*, Singapore, Lecture Notes in Computer Science: Vol. 1013 (pp. 213–228). Springer-Verlag.
- Batini, C., Ceri, S. & Navathe, S. B. (1992). *Conceptual database design: An entity-relationship approach*. Redwood City, USA: Benjamin/Cummings Publishing Company.
- Beeri, C. (1990). *A formal approach to object-oriented databases*. *Data & Knowledge Engineering*, 5, 353–382.
- Beeri, C. (1993). *Some thoughts on the future evolution of object-oriented database concepts*. In W. Stucky & A. Oberweis (Eds.), *Datenbanksysteme in Buero, Technik und Wissenschaft*, 5. GI-Fachtagung, Braunschweig, Germany (pp. 18–32). Springer-Verlag.
- Beeri, C. & Thalheim, B. (1999). *Identification as a primitive of database models*. In T. Polle, T. Ripke & K.-D. Schewe (Eds.), *Proceedings Fundamentals of Information Systems — FoMLaDO '98*, Timmel, Germany (pp. 19–36). Kluwer.
- Bernstein, P. A., Blaustein, B. & Clarke, E. M. (1980). *Fast maintenance of semantic integrity assertions using redundant aggregate data*. In *Proceedings Sixth International Conference on Very Large Databases*, Montreal, Canada (pp. 126–136). IEEE Computer Society Press.
- Booch, G., Rumbaugh, J. & Jacobson, I. (1998). *The unified modeling language user guide*. Reading, USA: Addison-Wesley Longman.
- Chen, P. P.-S. (1976). *The entity-relationship model—toward a unified view of data*. *ACM Transactions on Database Systems*, 1(1), 9–36.
- Chung, I., Nakamura, F. & Chen, P. P.-S. (1981). *A decomposition of relations using the entity-relationship approach*. In P. P.-S. Chen (Ed.), *Proceedings of the Second International Conference on the Entity Relationship Approach (ER'81)* (pp. 149–172). North-Holland.
- Codd, E. F. (1970). *A relational model of data for large shared data banks*. *Communications of the ACM*, 13(6), 377–387.
- Codd, E. F. (1979). *Extending the database relational model to capture more meaning*. *ACM Transactions on Database Systems*, 4(4), 397–434.

- Denninghoff, K. & Vianu, V. (1993). *Database method schemas and object creation*. In *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems (PoDS)*, Washington, D.C., USA (pp. 265–275).
- Embley, D. W. (1998). *Object database development: Concepts and principles*. Reading, USA: Addison-Wesley.
- Embley, D. W., Kurtz, B. D. & Woodfield, S. N. (1992). *Object-oriented systems analysis: A model-driven approach*. Englewood Cliffs, USA: Prentice-Hall
- Gogolla, M. (1995). *A declarative query approach to object identification*. In M. P. Papazoglou (Ed.), *Proceedings OOER'95: Object-Oriented and Entity-Relationship Modelling*, Fourteenth International Conference, Gold Coast, Queensland, Australia, Lecture Notes in Computer Science: Vol. 1021 (pp. 65–76). Springer-Verlag.
- Hall, P., Owlett, J. & Todd, S. (1976). *Relations and entities*. In G. M. Nijssen (Ed.), *Modelling in Database Systems* (pp. 201–220). North Holland Publishing Company.
- Hull, R. & Yoshikawa, M. (1991). *On the equivalence of database restructurings involving object identifiers*. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems (PoDS)*, Denver, Colorado, USA (pp. 328–340).
- Kemper, A. & Moerkotte, G. (1990). *Access support relations: An indexing method for object bases*. *Information Systems*, 17(2), 117–145.
- Khoshafian, S. N. & Copeland, G. P. (1986). *Object identity*. In N. K. Meyrowitz (Ed.), *Proceedings OOPSLA 1986, Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA. ACM SIGPLAN Notices, 21(11), 406–416.
- Kim, W. (1993). *Object-oriented database systems: Promises, reality, and future*. In R. Agrawal, S. Baker & D. A. Bell (Eds.), *Proceedings Nineteenth International Conference on Very Large Databases*, Dublin, Ireland (pp. 676–687). San Francisco, USA: Morgan Kaufmann Publishers.
- Klein, H.-J. & Rasch, J. (1997). *Value based identification and functional dependencies for object databases*. In *Data Management Systems—Proceedings of the Third International Basque Workshop on Information Technology (BIWIT 97)*, Biarritz, France (pp. 22–32). IEEE Computer Society Press.
- Kosky, A. S. (1995). *Observational distinguishability of databases with object identity*. In P. Atzeni and V. Tannen (Eds.), *Proceedings of the Fifth International Workshop on Database Programming Languages (DBPL-5)*, Gubbio, Italy, Electronic Workshops in Computing. Springer Verlag
- Kosky, A. S. (1996). *Transforming Databases with recursive data structures*. *Doctoral dissertation, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, USA*.
- Lacroix, M. & Pirotte, A. (1976). *Generalized joins*. *ACM SIGMOD Record*, 8(3), 15–16.
- Lee, B. S. (1995). *Normalization in OODB design*. *ACM SIGMOD Record*, 24(3), 23–27.
- Lien, Y. E. (1982). *On the equivalence of database models*. *Journal of the ACM*, 29(2), 333–362.
- Ling, T. W. (1985). *A normal form for entity-relationship diagrams*. In P. P.- S. Chen (Ed.), *Entity-Relationship Approach: The Use of ER Concept in Knowledge Representation*, *Proceedings of the Fourth International Conference on Entity-Relationship Approach*, Chicago, Illinois, USA (pp. 149–172). North-Holland.

- Ling, T. W. & Teo, P. K. (1994). *A normal form object-oriented entity relationship diagram*. In P. Loucopoulos (Ed.), *Entity-Relationship Approach—ER'94, Business Modelling and Re-Engineering, Proceedings Thirteenth International Conference on the Entity-Relationship Approach*, Manchester, U. K., Lecture Notes in Computer Science: Vol. 881 (pp. 241–258). Springer-Verlag.
- Maier, D. (1983). *The Theory of Relational Databases*. Rockville, USA: Computer Science Press.
- Maier, D., Ullman, J. D. & Vardi, M. Y. (1984). *On the foundations of the universal relation model*. *ACM Transactions on Database Systems*, 9(2), 283–308.
- Mok, W. Y. & Embley, D. W. (1996). *Transforming conceptual models to object-oriented database designs: Practicalities, properties, and peculiarities*. In B. Thalheim (Ed.), *Conceptual Modeling—ER'96, Fifteenth International Conference on Conceptual Modeling*, Cottbus, Germany, Lecture Notes in Computer Science: Vol. 1157 (pp. 309–324). Springer-Verlag.
- Nienhuys-Cheng, S.-H. (1990). *Classification and syntax of constraints in binary semantical networks*. *Information Systems*, 15(5), 497–513.
- Paton, N. W. & Gray, P. M. D. (1988). *Identification of database objects by key*. In K. R. Dittrich (Ed.), *Advances in Object-Oriented Database Systems, Proceedings Second International Workshop on Object-Oriented Database Systems*, Bad Muenster, Germany, Lecture Notes in Computer Science: Vol. 334. (pp. 280–285). Springer-Verlag.
- Rasch, J. (1998). *On value-based identification in object-oriented data models*. Doctoral dissertation, Institut für Informatik und Praktische Mathematik, Universität Kiel, Germany. Report 9815.
- Rauh, O. & Stickel, E. (1996). *Standard transformations for the normalization of ER schemata*. *Information Systems*, 21(2), 187–208.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-oriented modeling and design*. Prentice-Hall International (UK).
- Schewe, K.-D. & Thalheim, B. (1993). *Fundamental concepts of object oriented databases*. *Acta Cybernetica*, 11(1–2), 49–83.
- Scholl, M. H. & Schek, H.-J. (1986). *The relational model with relation-valued attributes*. *Information Systems*, 11(2), 137–147.
- Shipman, D. W. (1981). *The functional data model and the data language DAPLEX*. *ACM Transactions on Database Systems*, 6(1), 140–173.
- Tari, Z., Stokes, J. & Spaccapietra, S. (1997). *Object normal forms and dependency constraints for object-oriented schemata*. *ACM Transactions on Database Systems*, 22(4), 513–569.
- ter Hofstede, A. H. M. & van der Weide, T. P. (1993). *Expressiveness in conceptual data modelling*. *Data & Knowledge Engineering*, 10, 65–100.
- Thalheim, B. (2000). *Entity-Relationship Modeling*. Berlin, Germany: Springer-Verlag.
- van Bommel, P., ter Hofstede, A. H. M. & van der Weide, T. (1991). *Semantics and verification of object-role models*. *Information Systems*, 16(5), 471–495.
- van der Weide, T. P., ter Hofstede, A. H. M. & van Bommel, P. (1992). *Uniquet: Determining the semantics of complex uniqueness constraints*. *The Computer Journal* 35(2), 148–156.

- Verheijen, G. M. A. & van Bekkum, J. (1982). *NIAM: an information analysis method*. In T. W. Olle, H. G. Sol & A. A. Verrijn-Stuart (Eds.), *Information Systems Design Methodologies: A Comparative Review* (pp. 537–590). North-Holland.
- Weddell, G. E. (1990). *A theory of functional dependencies for object-oriented data models*. In W. Kim, J.-M. Nicolas & S. Nishio (Eds.), *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD89)*, Kyoto, Japan, 1989 (pp. 165–184). Elsevier Science Publisher B.V..
- Weddell, G. E. (1992). *Reasoning about functional dependencies generalized for semantic data models*. *ACM Transactions on Database Systems*, 17(1), 32–64.
- Wijssen, J., Vendenbulcke, J. & Olivie, H. (1994). *Functional dependencies generalized for temporal databases that include object-identity*. In R. Elmasri, V. Kouramajian & B. Thalheim (Eds.), *Entity-Relationship Approach- ER'93, Twelfth International Conference on the Entity Relationship Approach*, Arlington, Texas, USA, 1993, Lecture Notes in Computer Science: Vol. 823 (pp. 99–109). Springer-Verlag.
- Zaniolo, C. (1979). *Design of relational views over network schemas*. In P. A. Bernstein (Ed.), *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, USA (pp. 179–190).

Chapter IX: Integrity Issues in the Web—Beyond Distributed Databases

José F. Aldana Montes, Mariemma I. Yagüe del Valle, Antonio C. Gómez Lora,
Universidad de Málaga,

Spain

INTRODUCTION

Issues related to integrity in databases and distributed databases have been introduced in previous chapters. Therefore, the integrity problem in databases and how it can be managed in several data models (relational, active, temporal, geographical, and object-relational databases) are well known to the reader. The focus of this chapter is on introducing a new paradigm: The Web as the database, and its implications regarding integrity, i.e., the progressive adaptation of database techniques to Web usage. We consider that this will be done in a quite similar way to the evolution from integrated file management systems to database management systems.

In any case, this will be a much more difficult goal and quite a lot of work is still to be done. The special features of the Web make things which are necessary on a database system just optional in this environment. On the other hand, some other things which are usually considered as essential parts of any database, are now disassembled into its building blocks and used as needed (Silberschatz & Zdonik, 1996; Bernstein et al., 1998).

At first glance, the Web is a huge repository of information without any structure whatsoever. Nowadays, this is changing quickly. The consolidation of the Extensible Markup Language (XML, 1998) as a new standard adopted by the World Wide Web Consortium (W3C) has made the publication of electronic data easier. With a simple syntax for data, XML is, at the same time, human and machine understandable. XML has important advantages over HTML (HyperText Markup Language). While HTML is a data visualization language for the Web (this was not its initial intended purpose), with XML, data structure and rendering are orthogonal. We can represent meta-information about data through user-defined tags. No rendering information is included in an XML document.

It could be considered that the main feature of XML is that of being a data exchange format, but we will show that it is much more than this in this chapter.

Thinking about the Web as a huge, highly distributed database, we may consider different dimensions to conceptually describe it. Özsu and Valduriez (1999) defines a classification of database systems with respect to: 1) their distribution; 2) the autonomy of local systems; and 3) the heterogeneity of database systems. The autonomy concept is considered as the distribution of control, not of data. This indicates the degree to which individual DBMSs can operate independently. Whereas autonomy refers to the distribution of control, the distribution dimension deals with the physical distribution of data over multiple sites. With respect to heterogeneity, this can range from hardware heterogeneity, differences in networking protocols, variations in DBMSs, etc., to the data model or the policy for managing integrity on the database.

Obviously, the Web is on the distribution plane, and, as shown in [figure 1](#), we think that "it falls out" of the cube because it presents the highest degree of distribution, heterogeneity, and autonomy, and therefore, traditional distributed database techniques must be further extended to deal with this new environment. It is within this context that we are going to study the different issues related to integrity and its maintenance on the Web. We are also going to introduce the reader to other related and open issues, such as the query problem and query optimisation on the Web, since the special features of the Web environment make techniques for querying or maintaining the Web, different to those of traditional databases.

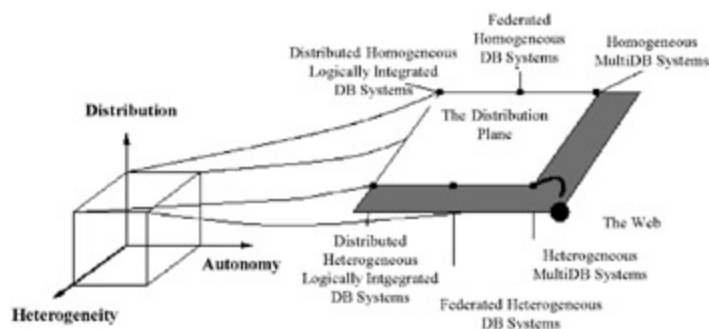


Figure 1: Extending the cube

SEMISTRUCTURED DATA AND XML: AN OVERVIEW

With respect to the information available on the Web, we can distinguish between data which is completely unstructured, such as images, sounds, and raw text, and highly structured data, such as data from a traditional database (relational, object-oriented or object-relational).

However, we can also find many documents on the Web that fall in between these two extremes. Such kinds of data have become relevant during the last few years and have been denominated semistructured data. A good introduction to this topic is found in (Buneman, 1997). In semistructured data, the information normally associated with a schema is contained within the data (self-describing data). In some cases there is no separate schema, whereas in others it exists but only places loose constraints on the data.

Therefore, semistructured data is characterized by the lack of any fixed and rigid schema, although typically the data has some implicit structure. The most immediate example of data that cannot be constrained by a schema is the Web itself.

One approach to providing database-like querying for semistructured WWW sources is to build wrappers for such sources. Ashish and Knoblock (1997) present an approach for semi-automatically generating wrappers through a wrapper-generation toolkit. The key idea is to exploit the formatting information in pages from the source to hypothesize the underlying structure of a page. From this structure the system generates a wrapper that facilitates querying a source and possibly integrating it with other sources.

Semistructured data may be irregular and incomplete and does not necessarily conform to a fixed schema. As with structured data, it is often desirable to maintain a history of changes to data, and to run queries over both the data and the changes. Representing and querying changes in semistructured data is more difficult than in structured data due to the irregularity and lack of schema. In Chawathe, Abiteboul & Widom (1998) a model for representing changes in semistructured data and a language for querying these changes is presented.

Several languages, such as Lorel (Abiteboul, Quass, McHugh, Widom & Wiener, 1997) and UnQL (Fernandez, 1996), support querying semistructured data. Others, such as WebSQL (Mihaila, 1996) and WebLog (Lakshmanan, Sadri & Subramanian, 1996), query Web sites. All these languages model data as labelled graphs and use regular path expressions to express queries that traverse arbitrary paths in graphs. As the data model is an edge-labelled directed graph, a path expression is a sequence of edge labels $l_1 l_2 \dots l_n$. Abiteboul, Buneman, and Suciu (2000) consider a path expression as a simple query whose result, for a given data graph, is a set of nodes. In general, the result of the path expression $l_1 l_2 \dots l_n$ on a data graph is the set of nodes v_n such that there exist edges $(r, l_1, v_1), (v_1, l_2, v_2), \dots, (v_{n-1}, l_n, v_n)$, where r is the root. Thus, path expressions result in set of nodes and not in pieces of semistructured data. In (Fernandez & Suciu, 1998) two

optimisation techniques for queries with regular path expressions are described, both of them relying on graph schemas which specify partial knowledge of a graph's structure.

All semistructured data models have converged around a graph-based data representation that combines data and structure into one simple data format. Some works (Nestorov, Abiteboul & Motwani, 1998) on typing semistructured data have been proposed based on labelled, directed graphs as a general form of semistructured data. XML (XML,1998) has this kind of representation based on labelled and directed graphs, although some minor differences exist between them, since the semistructured data model is based on unordered collections, whereas XML is ordered. The close similarity of both models (figures 2 and 3) made systems like LORE (Goldman, McHugh & Widom, 1999), initially built for a semistructured model, migrate from semistructured data to XML data.

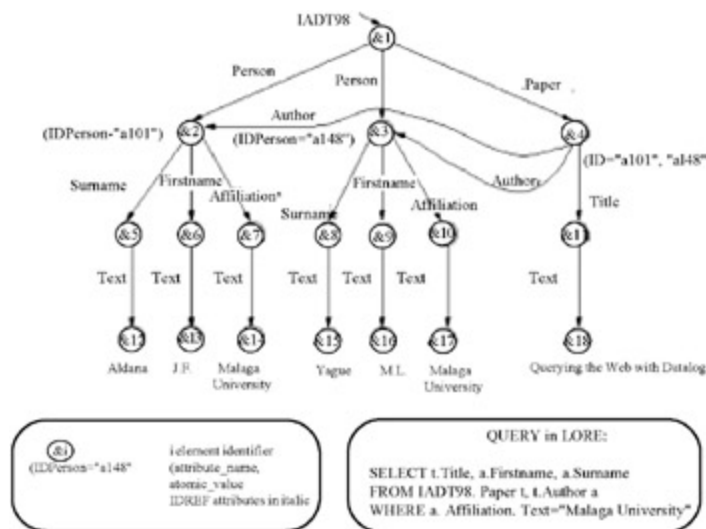


Figure 2: Data model in LORE

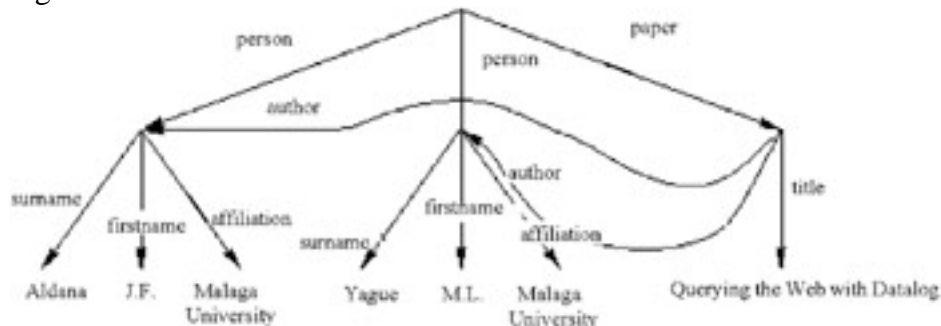


Figure 3: Data model in XML

XML (eXtensible Markup Language) was designed specifically to describe content, rather than presentation. XML is a textual representation of data that allows users to define new tags to indicate structure. In Figure 4, we can see that the textual structure enclosed by <Publication>...</Publication> is used to describe a publication tuple (the prefix of the tag names is relative to the namespace where the elements are going to be defined). An XML document does not provide any instructions on how it is to be displayed, and you can include such kind of information in a separate stylesheet. With the

use of XSL stylesheets (XSL, 1999) you can translate XML data to HTML for visualization by standard browsers.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="http://www.lcc.uma.es/XSL-Stylesheets/Publication.xsl"?>
<pri:Publication xmlns:pri="http://www.lcc.uma.es" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <pri>Title Uri="http://128.253.154.5/ProyectoXML_Cocos/Publicaciones/ iact08.xml">Querying the Web with
  Datalog</pri>Title>
  <pri:Author xsi:type="pri:StaffMember_Type">
    <pri:Name>José Francisco Aldana Montes</pri:Name>
    <pri:Identifier>JFAM1</pri:Identifier>
  </pri:Author>
  <pri:Author xsi:type="pri:StaffMember_Type">
    <pri:Name>Maria Inmaculada Yagüe del Valle</pri:Name>
    <pri:Identifier>MIYV1</pri:Identifier>
  </pri:Author>
  <pri:PublicationYear>1998</pri:Year>
  <pri:Scope>Internacional</pri:Scope>
  <pri:Congress>
    <pri:CongressName>
      International Workshop on Issues and Applications of Database Technology
    </pri:CongressName>
    <pri:StartPage>499</pri:StartPage>
    <pri:EndPage>504</pri:EndPage>
    <pri:Issn>1090-9380</pri:Issn>
    <pri:PublicationPlace>Berlin, Germany</pri:PublicationPlace>
  </pri:Congress>
</pri:Publication>
```

Figure 4: Example of XML document representing a publication page

XML Related Technology

A Document Type Definition (DTD) is a context-free grammar which defines the structure for an XML document type. DTDs are part of the XML language. A DTD can also serve as the "schema" for the data represented by an XML document. This is not as close as we would like to a database schema language, because it lacks semantics. Other limitations we find are that a DTD imposes order and lacks the notion of atomic types. That is, we cannot express that a 'weight' element has to be a non-negative integer, and moreover, we cannot express a range for constraining the weight between 0 and a maximum value. These and other limitations make DTDs inadequate from a database viewpoint. Therefore, new XML-based standards for representing structural and semantic information about the data have arisen. One of these proposals is the Resource Description Framework (RDF, 1999) and the RDF Schema (RDF Schema, 2000). However, and above all, we are going to place emphasis on the XML Schema (XML Schema, 2001), a new technological standard which enables us to represent data semantics like a database does.

RDF is a foundation for processing metadata, providing a simple data model and a standardized syntax for metadata. Basically, it provides the language for writing down factual statements. Its intended applications are mainly: 1) providing better search engine capabilities for resource discovery; 2) cataloging for describing the content and content relationships available at a particular Web site; and 3) allowing intelligent software agents to share and exchange knowledge (Abiteboul et al., 2000). RDF consists of a data

model (an edge-labeled graph with nodes called resources and edge labels called properties) and a syntax for representing this model in XML.

On top of RDF, the simple schema language RDFS, Resource Description Framework Schema (RDF Schema, 2000) has been defined to offer a specific vocabulary to model class and property hierarchies and other basic schema primitives that can be referred to from RDF models. The RDF Schema provides a means to define vocabulary, structure, and integrity constraints for expressing metadata about Web resources. Once again, the main problem with RDF is its lack of a standard semantics and, therefore, this semantics must be defined in each of its instances. An RDF Schema instance allows for the standardization of the metadata defined over Web resources, and the specification of predicates and integrity constraints on these resources. In knowledge engineering terminology, the RDF Schema defines a simple ontology that particular RDF documents may be checked against to determine consistency. In addition, the RDF Schema is a type system for RDF since it provides a mechanism to define classes of resources and property types which restrict the domain and range of a property.

The RDF and RDF Schema specifications use XML as an interchange format to exchange RDF and RDF Schema triples (Bowers & Delcambre, 2000).

As mentioned, some schema languages for describing XML data structures and constraints have been proposed.

XML DTD is the de facto standard XML schema language but has limited capabilities compared to other schema languages, such as its successor XML Schema. Its main building block consists of an element and an attribute and it uses hierarchical element structures. Other schema languages have been proposed, such as Schematron, DSD, SOX, XDR, among others. A comparative analysis of the more representative XML schema languages is found in Lee & Chu (2000). In this chapter, the focus will be on XML Schema because the XML Schema is an ongoing effort by the W3C for replacing DTD with a more expressive language.

The XML Schema is written in XML enabling the use of XML tools. It presents a rich set of data types, capabilities for declaring user-defined datatypes, mechanisms such as inheritance, and so on. In fact, XML Schemas are object-oriented.

Although the XML Schema identifies many commonly recurring schema constraints and incorporates them into the language specification, it will be interesting to see how constraint support will evolve in XML Schema in the future.

THE INTEGRITY PROBLEM ON THE WEB

A part of the semantics of a database is expressed as integrity constraints. Constraints are properties that the data of a database are required to satisfy and they are expected to be satisfied after each transaction performed on the database. The verification of constraints in a database is often quite expensive in terms of time as well as being complex. Some

important factors related to this issue include the structure of the underlying database upon which the constraints are imposed, the nature of the imposed constraints, and the method adopted for their evaluation.

Using the notation of previous chapters we are going to consider Domain Restrictions and Relationships Restrictions and their representation in a Web data model.

Integrity Constraints and XML Standard Technology. Declaration and Enforcement

XML is a data model which can not only represent semantic information through descriptive tags, but thanks to other related technologies, also through certain types of database-like schemes.

In this section, we are going to focus on the XML Schema as the most appropriate related technology for representing such kinds of information as integrity constraints.

XML Schemas are selected because:

1. they provide enhanced data types (more than 41), user-defined data types, and the extension or restriction of a type (derivation of new type definitions on the basis of old ones).
2. it is possible to define the lexical representation. For example, "This element can contain strings of this form: ddd-dddd, where 'd' is a digit".
3. being written in XML, they enable the use of XML tools.
4. they are object-oriented.
5. they can express sets (the child elements may occur in any order).
6. they can specify the element content as being unique (keys on content) and uniqueness within a region.
7. they can define multiple elements with the same name but different content.
8. they can define elements with null content.
9. they can create equivalent elements. For example, the "publication" element is equivalent to the "paper" element.

This last feature means that the XML Schema enables the use of different vocabularies while remaining understandable by every XML application which validates that XML Schema.

```

<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
targetNamespace="http://www.lcc.uma.es"
xmlns:pri="http://www.lcc.uma.es"
xmlns:typ="http://www.w3.org/2000/10/XMLSchema-datatypes">
<include schemaLocation = "Staff.xsd" />
.
.
.
<xsd:complexType name="Publication_Type">
  <xsd:element ref="pri:Title"/>
  <xsd:element ref="pri:Author" minOccurs="1"
maxOccurs="unbounded"/>
  <xsd:element ref="pri:PublicationYear"/>
  <xsd:element ref="pri:Scope"/>
  <!-- choice allows to include one of the elements referenced -->
  <xsd:choice>
    <element ref="pri:BookChapter"/>
    <element ref="pri:Book"/>
    <element ref="pri:JournalArticle"/>
    <element ref="pri:Journal"/>
    <element ref="pri:Thesis"/>
    <element ref="pri:Conference"/>
    <element ref="pri:ResearchReport"/>
    <element ref="pri:Congress"/>
  </xsd:choice>
  <!-- Element declared in Staff.xsd -->
  <xsd:element ref="pri:Research" minOccurs="0"
maxOccurs="unbounded"/>
  <xsd:element ref="pri:Key Word" minOccurs="0" maxOccurs="15"/>
  <xsd:element ref="pri:Abstract" minOccurs="0" maxOccurs="1"/>
  <xsd:element ref="pri:Format" minOccurs="0" maxOccurs="1"/>
</xsd:complexType>
.
.
</schema>

```

Figure 5: Example of XML schema for a publication page

Domain Restrictions

A domain restriction defines the set of values that an attribute may have.

For example: the age of a person is a non-negative integer number.

```

<simpleType name= "age" base="integer">
  <minInclusive value="0"/>
  <maxInclusive value="150"/>
</simpleType>

```

This XML-Schema code defines an element named ‘age’, of type ‘integer’, and whose values are restricted to the range [0..150].

The following code in XML-Schema declares a new data type called ‘PhoneNumber_Type’ which represents values for valid Spanish phone numbers. Elements of this type must have string values, its length has to be exactly 9 characters, and values have to match the following pattern: the first digit can be 6 or 9, followed by

another 8 digits (0..9). The pattern is represented with a regular expression where ‘d’ represents a digit.

```
<xsd:simpleType name="PhoneNumber_Type">
  <xsd:restriction base="typ:string">
    <xsd:length value="9"/>
    <xsd:pattern value="[6|9]\d{8}"/>
  </xsd:restriction>
</xsd:simpleType>
```

The following declaration in XML-Schema defines a type supporting correct e-mail addresses:

<!-- E-mail_Type contains the e-mail address of a member of the department. This type is expressed with a regular expression indicating the template to follow for a correct e-mail address. -->

```
<xsd:simpleType name="E-mail_Type">
  <xsd:restriction base="typ:string">
    <xsd:pattern value="([A-Z]|[a-z]|\.|\\d|_)*@([A-Z]|[a-z]|\.|\\d|_)*"/>
  </xsd:restriction>
</xsd:simpleType>
```

As mentioned, XML-Schemas provide enhanced data types (more than 41 different basic data types), and user-defined data types. They include predefined types such as integer, float, double, uriReference, etc. Furthermore, we can create new data types from base data types specifying values for one or more facets for the base data type.

For example, for the primitive data type string we have new optional facets (some of them used in the examples above):

- pattern
- enumeration
- length
- maxlength
- maxInclusive
- maxExclusive
- minlength
- minInclusive
- minExclusive

For more information about primitive types and their facets for the derivation of new type definitions, the reader can visit the W3C Consortium web page regarding this standard (XML Schema, 2001).

Moreover, on XML-Schema we can define subclasses and superclasses of types as we can see in the following.

Generalization as Restriction: we can restrict some of the elements of a more general type making them only accept a more restricted range of values or a minor number of instances.

If we have defined the following data type:

```
<complexType name="Publication">
  <sequence>
    <element name="Title" type="string" minOccurs="1"
      maxOccurs="1"/>
    <element name="Author" type="string" minOccurs="1"
      maxOccurs="unbounded"/>
    <element name="PublicationYear" type="year"
      minOccurs="1" maxOccurs="1"/>
  </sequence>
</complexType>
```

then we can derive for extension a type for a Publication, such as:

```
<complexType name="Proceedings" base="Publication"
  derivedBy="extension">
  <sequence>
    <element name="ISBN" type="string" minOccurs="1"
      maxOccurs="1"/>
    <element name="Publisher" type="string"
      minOccurs="1" maxOccurs="1"/>
    <element name="PlaceMeeting" type="string"
      minOccurs="1" maxOccurs="1"/>
    <element name="DateMeeting" type="date"
      minOccurs="1" maxOccurs="1"/>
  </sequence>
</complexType>
```

or we can derive for restriction a type for a Publication:

```
<complexType name="SingleAuthor" base="Publication"
  derivedBy="restriction">
  <sequence>
    <element name="Title" type="string"
      minOccurs="1" maxOccurs="unbounded"/>
    <element name="Author" type="string"
      minOccurs="1" maxOccurs="1"/>
    <element name="PublicationYear" type="year"
      minOccurs="1" maxOccurs="1"/>
  </sequence>
```

```
</complexType>
```

Sometimes we want to create a data type and disable any kind of derivation from it.

For example, we can specify that Publication cannot be extended or restricted:

```
<complexType name="Publication" final="#all" ...>
```

Or we can disable the restriction of the type Publication:

```
<complexType name="Publication" final="restriction" ...>
```

Similarly, to disable the extension:

```
<complexType name="Publication" final="extension" ...>
```

Relationships

With respect to structural restrictions which express semantic properties implicit in a model, such as unique keys on the relational model or one-to-many associations on the network model, in this schema language we can represent:

1. Uniqueness for attribute: XML Schemas support this feature using `<Unique>`, where the scope and target object of the uniqueness are specified by `<Selector>` and `<Field>` constructs, respectively. Moreover, XML Schemas specify uniqueness not only for attributes but also for arbitrary elements or even composite objects (attribute + element) in a portion of the document or the whole document using the same construct `<Unique>`.

For instance, the following schema ensures there exists a unique `PhoneNumber` element under `office` sub-elements of the `teacher` element.

```
<unique><selector>teacher/office</selector>
<field>PhoneNumber</field></unique>
```

2. Key for attribute: In databases, being a key requires being unique as well as not being null. A similar concept is defined in XML Schemas.
3. Foreign key for attribute: Foreign key states: a) who is a referencing key; and b) who is being referenced by the referencing key.

The XML Schema uses `<Keyref>` for this purpose. In addition to this, XML Schemas support a method to specify whom the foreign key actually points to using constructs `<Refer>` and `<PointsTo>`, respectively.

Using syntax almost identical to <Unique>, a construct <Key> can specify an attribute as a key in XML Schemas.

```
<key name="dNumKey">
    <selector>departments/department</selector>
    <field>@number</field>
</key>
<keyref refer="dNumKey">
    <selector>subject/department</selector>
    <field>@number</field>
</keyref>
```

Similar to specifying uniqueness for non-attributes, XML Schemas can specify foreign keys for arbitrary elements or composite objects using the same <Keyref> construct.

In the XML data model, entity and referential constraint types have significant differences with respect to the relational and object-oriented data model, generating great research activity and discussion (Fan & Siméon 2000; Buneman, Davidson, Fan, Hara & Tan, 2001; Buneman, Fan & Siméon, 2000). Nowadays, XML is mainly being used in data interchange between relational databases and applications over the Web and it seems that this situation will not change in the near future (Fan, Kuper & Siméon, 2001; Lee & Chu, 2000). This means that XML integrity semantics must be at least as expressive as relational integrity semantics, including referential aspects.

Entity and referential integrity are commonly used close terms. There are two possible and non-contradictory implementations: logical or physical pointers. The relational model uses logical pointers and object-oriented models typically use physical ones. As the reader knows, in relational models primary and alternative keys implement entity integrity, and foreign keys implement referential integrity. In object-oriented systems a built-in physical pointer, usually called ID, that is present on every object, is in general used. In these systems, the reference and inverse reference clauses specify the existence of referential integrity. On the other hand, in earlier XML Schema drafts, the XML entity and referential mechanisms were close to the object-oriented solution, introducing a mechanism that is called *id/idref*. Later studies showed semantic anomalies between this and the relational one. The *id/idref* mechanism is defined as the way of referencing explicit data into an XML document; this means, it is closer to pointers in a programming language than to keys in a relational or object-oriented model. Another interesting aspect of references in XML is that they are generic pointers, with no associated type. Thus, we have no information, and we cannot maintain any control, on what the reference is pointing to.

The latest versions (XML Schema, 2001) have increased the capability of expressing integrity constraints. The new *key/keyref* mechanism complements the *id/idref* one and solves its associated problems. It also allows us to define complex types as keys, generating new interesting practical problems. This is just one example of how quickly this new technology is evolving.

Referential integrity implements relationships in the data model. The majority of complex queries involve one or more relationship computation, that is, keys are commonly processed in query evaluation. Keys, both primary and alternative or foreign ones, generally support any type of implicit index optimisation technique in relational database management systems.

Behavior Restrictions

These can express associations between objects, such as the inclusion dependency in the relational model, or can describe object properties and structures. At the moment, they cannot be expressed in XML Schemas but we hope this will change in the future. However, we can construct an XML-validating application which can enforce this kind of additional restriction over the data.

Integrity Control For Distributed XML-Validating Applications and the Web

Nowadays, with the use of XML as a logical model - and not just a standardized document markup language and a data exchange format (a physical data model) -, modern database technology has become practical for the storage and retrieval of data on the Web. In fact, we can automatically process the information represented in the XML Schema (it is important to note that all this technology is built over XML). This can be done by using an API, such as the Document Object Model (DOM, 2000). DOM enables us to compile an XML document and construct a tree representation for it. It offers an object-oriented view of the XML document, that is, each document component defines an interface specifying its behavior. The data (or state) can be accessed only via this interface. Thus, an application can completely restructure the document via this interface. It can manage the different elements in it in an "intelligent way", because of the semantic information which is present in the document.

An XML parser is code which reads a document and analyses its structure. We can find validating parsers vs. non-validating ones, DOM-compliant parsers, SAX-compliant (SAX, Simple API for XML) parsers, parsers written in Java, C++, Perl, etc.

On the one hand, we can represent integrity constraints for data which could be distributed all over the Web by means of the XML Schema. On the other hand, we can develop XML applications ([figure 6](#)) using DOM and standard XML Schema-validating parsers for checking integrity at each updating operation.

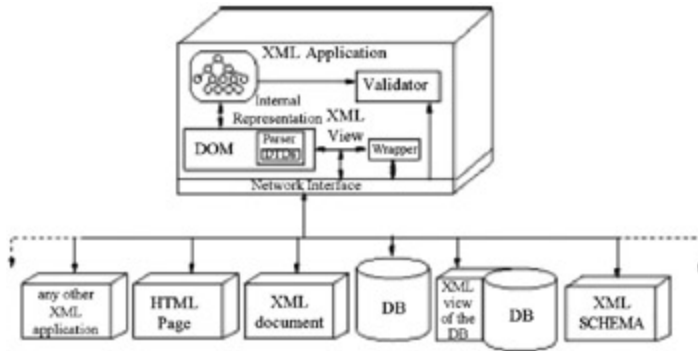


Figure 6: XML application

For example, if we have an XML Schema for a "publications" document type at a University Department we can easily think of some integrity constraints. For example: "a publication has at least one author". Now, this can be represented in XML Schemas. An XML Schema-validating parser could be responsible for accepting or rejecting an XML document enforcing a non-null attribute integrity constraint. In addition, it might be interesting to add another constraint to this: "at least one of the authors of a publication has to be a member of the department". This kind of restriction is much more difficult to validate, as you have to check for the different XML documents representing the Department staff looking for a publication's author. If this search is successful (one of the authors is a member of the Department), then the integrity constraint is not violated and the updating operation that adds a new publication page can be carried out, and in this way the database (the part of the Web which we are dealing with) attains a consistent state. In case of the search not being successful, the publication could not be added, if we want to enforce the integrity. The cost of this checking is not too high, as every XML document is going to be locally checked by means of a standard XML Schema-validating parser.

Query Optimisation in the Web

Although in XML column integrity would be called element or attribute constraints we will preserve the traditional name. Column and domain constraints do not present additional problems like entity and referential integrity does. Nevertheless, the processing of XML documents is computationally more expensive than the processing of relations. More generally, the computational management of semistructured data is more complex and expensive than the management of structured data. This means that semistructured data management requires more and better optimisations than relational database management systems. XML optimisation techniques are still quasi-unexplored, due to the absence of a definitive and stable query language and algebra specification. However, there is much work in specific areas and many optimisation techniques developed under different paradigms that could be adapted to XML.

Optimisation on regular path queries (Grahne & Thomo, 2000; Grahne & Thomo 2001) and indexing techniques over semistructured information (McHugh & Widom, 1999) have already been studied. However, other relevant aspects, such as composition reordering and restriction propagations, have still not been analyzed under the XML data

model, although they can be performed in this new model with relatively less effort. These techniques are well known and used in relational data models. More complex and sophisticated techniques, such as magic rewriting (Bancilhon, Maier, Sagiv & Ullman, 1986), which have demonstrated goods results, have yet to be tested in the XML context.

Some algorithms require less effort; others will require more complex and refined translation. Thus, semantic query optimisation via residue computation (Chakravarthy, Grant & Minker, 1990) could be complex in XML, because it would require a complete redesign of the original technique. However, if we introduce domain restrictions in the query graph, predicate move around (Levy, Mumick & Sagiv, 1994), which could be easily adapted to XML, would yield a similar optimisation level.

One of the most important studies on XML and semistructured data optimisation (McHugh & Widom, 1999) has been developed for the LOREL system (Abiteboul, Quass, McHugh, Widom & Wiener, 1997), which defines several index structures over XML data and schema, yielding efficient data management. In the physical query plan, LORE not only supports the traditional value index, but also label, edge, and path indexes.

Domain and Column Constraint Propagation techniques have the same basis as selection propagation techniques; they are based on query algebra laws.

Traditional selection propagation methods are based on the axioms and properties of the query algebra, especially on those defining the commutation of selection with the rest of the operators. They use these algebra equivalences as rewriting rules, but each algorithm determine how these ones must be applied (when, where and how many times). Some good examples for the Relational Algebra can be found in traditional database literature (Ullman, 1989; Abiteboul et al., 1995). Predicate Move Around (Levy et al., 1994) is an extension of the selection propagation algorithm that yields better results and, furthermore, those constraints that could be expressed as selections in the query could be used in the optimisation process.

Constraint propagation must follow the laws defined in XML algebra. There are three basic groups of rules ([figures 7a](#), [7b](#) and [7c](#)) than can be used for query optimisation, including constraint propagation.

for v in e1 do e2	=	e2{v := e1}
for v in e do v	=	E
for v2 in (for v1 in e1 do e2) do e3	=	for v1 in e1 do (for v2 in e2 do e3)

Figure 7a: Monad laws

e/a	→	For v1 in e do
		for v2 in nodes(v1) do
		match v2
		case v3 : a[AnyComplexType] do v3
		else ()

Figure 7b: Equivalence between projection and iteration

for v in () do e	→	()
for v in (e1, e2) do e3	→	(for v in e1 do e3), (for v in e2 do e3)
for v in e1 do e2	→	E2{e1/v}, if e1 : u
for v in e do v	→	E
e1 : {t1}, e2 : {t2}, v1 free in e2	→	for v2 in e2 do for v1 in e1 do e3
for v1 in e1 do for v2 in e2 do e3		
E[if e1 then e2 else e3]	→	if e1 then E[e2] else E[e3]
E[let v = e1 do e2]	→	let v = e1 do E[e2]
E[for v in e1 do e2]	→	for v in e1 do E[e2]
E[match e1	→	match e1
case v : t do e2		case v : t do E[e2]
...		...
case v : t do en-1		case v : t do E[en-1]
else en]		else E[en]

Figure 7c: Optimisation laws

As mentioned, constraint propagation and selection propagation are similar in the relational algebra context. In XML algebra, selection is not defined explicitly, but we can use the WHEN algebra element to define it, which is semantically defined as a particular case of the IF-THEN-ELSE structure. This construction is more generic than selection; it implements both filters and complex compositions. WHEN or IF-THEN-ELSE structures act like filters and can implement any derived condition from known constraints, at least in the simplest algorithms.

Example 1 Let's see a single example of constraint propagation in XML. Let's assume the following Schema:

```
<xsd:simpleType name="distance" type="xs:integer"
minIncluded="0"/>
<xsd:simpleType name="autonomy" type="xs:integer"
minIncluded="0" maxIncluded="900"/>

<xsd:element name="car">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="model" type="xs:string"
        use="required"/>
      <xsd:element name="kilometers" type="xs:
        autonomy" use="required"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="road"
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="sourcecity" type="xs:
        string" use="required"/>
      <xsd:element name="targetcity" type="xs:
        string" use="required"/>
      <xsd:element name="kilometers" type="xs:
        distance" use="required"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```



```

        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

```

In the following, the W3C query algebra (XML Formal Semantics, 2001) is used. This algebra is based on the for iteration operator, as SQL is based on the select statement.

And now, for the query

```

for r in base/road do
  where r/sourcecity/data() = "Madrid" do
    for c in base/car do
      where c/model/data = "mondeo" do
        where c/kilometres/data() <= r/kilometres/data() do
          possibleroad[ r/targetcity, r/kilometres ]

```

with the derived type

```

<xsd:element name="possibleroad"
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="targetcity" type="xs:string"
        use="required"/>
      <xsd:element name="kilometers" type="xs:
        distance" use="required"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

An adapted version of the *predicate move around* technique (Levy et al., 1994) propagates the constraint "*car/kilometres/data() < 900*". We indicate this constraint propagation using C comments (*/*...*/*).

```

for r in base/road do
  where r/sourcecity/data() = "Madrid" do
    for c in base/car /* base/car/kilometres/data() < 900 */ do
      where c/model/data = "mondeo" do
        where and c/kilometres/data() >= r/kilometres/data() do
          possibleroad[ r/targetcity, r/kilometres ]

?
for r in base/road do
  where r/sourcecity/data() = "Madrid" do
    for c /* c/kilometres/data() < 900 */ in base/car do
      where c/model/data = "mondeo" do
        where and c/kilometres/data() >= r/kilometres/data() do
          possibleroad[ r/targetcity, r/kilometres ]
?
for r in base/road do
  where r/sourcecity/data() = "Madrid" do
    for c in base/car do
      where c/model/data = ">mondeo" /* c/kilometres/data() <
        900 */ do

```

```

        where and c/kilometres/data() >= r/kilometres/data()do
            possibleroad[ r/targetcity, r/kilometres ]
    ?
for r in base/road do
    where r/sourcecity/data() = "Madrid" do
        for c in base/car do
            where c/model/data = "mondeo" do
                where and c/kilometres/data() /* c/kilometres/
                    data() < 900 */ >= r/kilometres/data() do
                    possibleroad[ r/targetcity, r/kilometres ]
            ?
for r in base/road do
    where r/sourcecity/data() = "Madrid" do
        for c in base/car do
            where c/model/data = "mondeo" do
                where and c/kilometres/data() >= r/kilometres/
                    data() /* r/kilometres/data() < 900 */ do
                    possibleroad[ r/targetcity, r/kilometres ]

```

Here, the restriction can take two ways: upwards (towards the inner do) and downwards (towards the outer for).

```

for r in base/road do
    where r/sourcecity/data() = "Madrid" /* r/kilometres/
        data() < 900 */ do
        for c in base/car do
            where c/model/data = "mondeo" do
                where and c/kilometres/data() >= r/kilometres/data()
                    do
                    possibleroad[ r/targetcity, r/kilometres ] /* r/
                        kilometres/data() < 900 */

```

Finally no more propagation can be done.

```

for r in base/road do
    where r/sourcecity/data() = "Madrid" and r/kilometres/
        data() < 900 do
        for c in base/car do
            where c/model/data = "mondeo" do
                where and c/kilometres/data() >= r/kilometres/data()
                    do
                    possibleroad[ r/targetcity, r/kilometres ]

```

We can observe that not only the query is more efficient, but the query derived type is now:

```

<xsd:element name="possibleroad"
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="targetcity" type="xs:string" use=
        "required"/>
      <xsd:element name="kilometers" type="xs:distance"
        use="required" maxExcluded="900"/>
    </xsd:sequence>

```

```

</xsd:complexType>
</xsd:element>

```

The propagation can be viewed graphically in [Figure 8](#).

From the XML perspective, constraint propagation cannot be used exclusively for query optimisation, but it can and must be used during new type derivation or inference (Fan et al., 2001). This is generically called subtyping. One of the main characteristics of XML is that it allows new type extensions and restrictions through inheritance. When a new type is defined or derived from an existing one, it not only inherits its members, but also a new set of associated constraints. Some of these new restrictions will be explicitly defined, but others will be inherited from the base type. The computation of this new set of constraints is fundamental in object-oriented systems, and very important in preventing integrity constraints violations in our schema. There are two basic ways to implement and validate a particular object of a known type: on the one hand, verifying each of the defined constraints for the object type and all its ancestors; and, on the other hand, deriving and making explicit all the constraints that must be validated during type definition, allowing us to verify each object exclusively using the set of constraints previously computed. From an optimisation point of view, the second choice is better, because it can detect inconsistencies at type definition time, and not at query time.

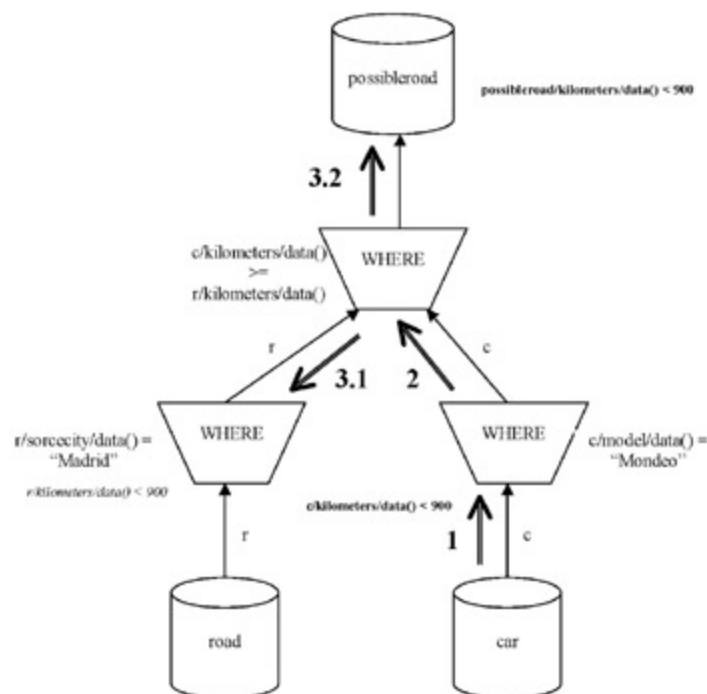


Figure 8: Constraint propagation

Constraint derivation in subtyping is useful with domain and column constraints, but it is also useful with entity and referential constraints (Fan et al., 2001). The existence of physical index associated with keys and references in order to accelerate relationship composition is common.

PRACTICAL IMPLICATIONS

Before starting this section, we have to note that when this chapter was being written, the World Wide Web Consortium (W3C) was still developing the XML Schema, XML Algebra, and XML Query recommendations. These are not definitive, but just drafts (the latest version was published on February 16th, 2001). The number and relevancy of the still open issues has made future significant changes possible. On the other hand, in the last six months, two new draft versions have been published, which introduce new and important changes with respect to previous releases.

The paradigm of distribution in databases is a well-known problem in both the research and industry communities. Nowadays, many commercial databases support certain degrees of parallelism and distribution in both data and control. Nevertheless, almost all of them are based on the relational model. The relational model manages structured information in a formal data model using a simple and well-defined algebra. However, even in this well-known and extended model, the distribution paradigm has no trivial solution.

In summary, the representation of meta-information along with data opens up a new way to automatically process Web information because of the use of explicit semantic information. The goal of this chapter is to show how the amalgamation of Web and Database technology appears to be very promising. The focus has been on semantic integrity issues. A semantic integrity subsystem has two main components: a language for expressing and manipulating integrity assertions and an enforcement mechanism that performs specific actions to enforce database integrity between state changes. We have proposed XML Schemas as the basis for solving the semantic integrity problem in the Web. We have showed how integrity constraints can be represented by means of the XML Schema and how this integrity can be maintained using XML applications and standard XML Schema-validating parsers. We have also mentioned how XML Schema may "understand" different vocabularies, which has important implications with respect to schema integration.

The Web environment is not equivalent to a traditional distributed environment for a relational database, the problem of managing semistructured information in such an environment being significantly more complex. It is generally accepted that XML and its associated recommendations will be the future model for Web query processing. However, XML introduces new additional problems to the paradigm of semistructured information management in the Web environment.

In any case, a lot of work has still to be carried out in the database community to resolve all the issues related to such a kind of distributed and heterogeneous database, which is what the Web actually is.

Since Codd formally defined the relational model in the early 1970s, it has proved its expressiveness and efficiency, but also has presented limitations. This has motivated the definition of many extensions. Among these, two have shown an unusually high degree

of success. Deductive and object-oriented database paradigms are attempts to introduce recursion and object-orientation in databases. XML is a standard based on a semistructured model that allows structural recursion. Its future algebra has functional language characteristics that support both static and dynamic type inference. This means that XML includes and extends the problems of distribution, recursion, and object-orientation in relational models.

Although XML is in its early stages, its general acceptance is focusing much research and development in both the industry and research communities. Even though not completely mature, it is being successfully used in the e-commerce field, B2B, data interchange and integration, etc.

Obviously, XML has inherited not only the advantages from its ancestors, but also many still open problems at both theoretical and practical levels, which affect many aspects, including constraints management.

It has been shown that there exist several ways to specify integrity constraints in XML, using DTDs or XML Schema, among others. To avoid multiple fetching of constraints expressed in different formats during data management, it would be desirable to choose a unique format of constraints specification. The XML Schema seems to be the best candidate due to its expressiveness, as is shown in recent studies (Lee & Chu, 2000). Nevertheless, other standards, like RDF and RDF Schemas, are complementary and can be used together in a higher abstraction level, as proposed in the Semantic Web.

In a keynote session at XML 2000, the Director of the World Wide Web Consortium, Tim Berners-Lee, outlined his vision for the Semantic Web: "in the context of the Semantic Web, the word semantic means machine processable. For data, the semantics convey what a machine can do with that data." He described the "semantic test," which is passed if, when you give data to a machine, it will do the right thing with it. He also underlined that the Semantic Web is, like XML, a declarative environment, where you say what you mean by some data, and not what you want to do with it.

Having outlined its scope, Berners-Lee explained each of the elements in the Semantic Web architecture. He explained the importance of RDF/RDF Schema as a language for the description of "things" (resources) and their types. Above this, he described the ontology layer. An ontology is capable of describing relationships between types of things, such as "this is a transitive property", but does not convey any information about how to use those relationships computationally. On top of the ontology layer sits the logic layer. This is the point at which assertions from around the Web can be used to derive new knowledge. The problem here is that deduction systems are not terribly interoperable. Rather than design one overarching reasoning system, Berners-Lee suggests a universal language for representing proofs. Systems can then digitally sign and export these proofs for other systems to use and possibly incorporate into the Semantic Web.

Most of the new work today is happening regarding ontologies. Practical solutions include the use of XSLT to derive RDF from XML sources, work on topic maps and

RDF convergence, the emergence of general-purpose RDF databases and engines, and general and specific GUIs for RDF data.

Almost all of the aspects related to maintenance and query optimisation via integrity constraints are open in XML, because of its recent development. The effort and collaboration of many and different disciplines is necessary for the development of future XML database management systems with the same features as current commercial relational systems, such as Oracle, Informix, DB2, etc.

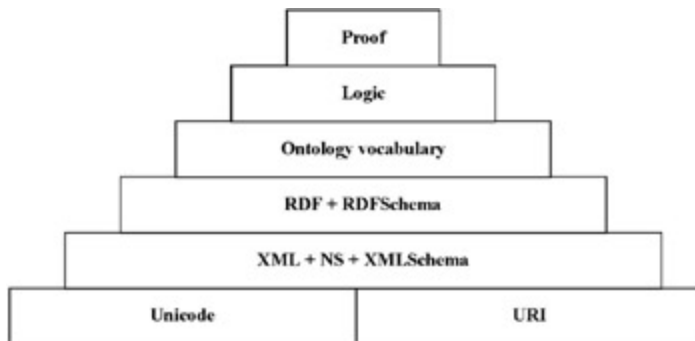


Figure 9: Semantic Web

Many frontiers are open to research and development. Moreover, we still cannot ensure that the W3C XML query language and algebra recommendations, in its current status, would be valid as a practical query language for data intensive processing. Alternative proposals exist, see the comparative analysis of Bonifati and Ceri (2000), although many of them conform to the W3C's one. A good example is the proposal (Beech, Malhotra & Rys, 1999) developed by three important W3C members: Oracle, IBM, and Microsoft. Together, they have developed a query language and algebra very close to SQL and relational algebra whose results are compatible with XQuery's although these are especially oriented to data intensive processing.

An update semantic model is still undefined revealing the amount of work yet to do. For a complete definition of the data manipulation language, it will be necessary to define new recommendations including the given update commands. Having completed the process of complete formalization of the language, the development of a transactional system for XML would be necessary. It would maintain data integrity under multiple concurrent accesses.

This work is mainly related to the logical data schema. Improvements in the physical model will begin later on.

The current commercial interest and pressure for XML technology development will make almost all computer science disciplines converge. An interesting example of this can be observed in the XML query algebra draft. This algebra is defined on a mathematical modal concept, which is usual in functional languages. This concept has been exhaustively studied for languages like Haskell, and has been applied to generate optimisation mechanisms based on binding propagation. Magic rewriting techniques have

proved their good results in query optimisation on both deductive and relational databases. They are also based on binding strategies, called Sideway Information Passing, or *sip* strategies. The most sophisticated versions of these algorithms use integrity constraints to determine the binding strategy. Thus, optimisation based on binding strategies could be approached in XML by both Datalog and Haskell developers.

As a result, we may note how many aspects of XML query processing, including integrity constraints management, would have to receive serious attention from the database community. New fields of research are open and in-depth research on all aspects related to this new data model on the Web are of vital interest regarding its application to industry.

REFERENCES

- Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.
- Abiteboul, S., Quass, D., McHugh, J., Widom, J., & Wiener, J. (1997). *The Lorel query language for Semistructured Data*. *Journal of Digital Libraries.*, 1(1), 68–88.
- Abiteboul, S., Buneman, P., & Suciu, D. (2000). *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers.
- Abiteboul, S., Quass, D., McHugh, J., Widom, J., & Wiener, J. (1997). *The Lorel query language for semistructured data*. *International Journal on Digital Libraries*, 1(1), 68–88.
- Ashish, N., & Knoblock, C. A. (1997). *Wrapper Generation for Semi-structured Internet Sources*. *SIGMOD RECORD*, 26(4), 8–15.
- Bancilhon, F., Maier, D., Sagiv, Y., & Ullman, J.D. (1986). *Magic Sets and other strange ways to implement logic programs*. *ACM Symposium on Principles of Database Systems (PODS'86)*, 1–15.
- Beech, D., Malhotra, A., & Rys, M. (1999). *A formal data model and algebra for XML. Communication to the W3C*. [On-Line]. Available: <http://www-db.stanford.edu/dbseminar/Archive/FallY99/malhotra-slides/malhotra.pdf>
- Bernstein, P. A., Brodie, M. L., Ceri, S., DeWitt, D. J., Franklin, M. J., Garcia-Molina, H., Gray, J., Held, G., Hellerstein, J. M., Jagadish, H. V., Lesk, M., Maier, D., Naughton, J. F., Pirahesh, H., Stonebraker, M., & Ullman, J. D. (1998). *The Asilomar Report on Database Research*. *SIGMOD Record* 27(4), 74–80.
- Bonifati, A., & Ceri, S. (2000). *Comparative Analysis of Five XML Query Languages*. *SIGMOD Record* 29(1): 68–79.
- Bowers, S., & Delcambre, L. (2000). *Representing and Transforming Model Based Information*. ECDL 2000, Workshop on the Semantic Web.
- Buneman, P. (1997). *Semistructured Data*. *Proceedings of the 16th ACM Symposium on Principles of Database Systems (PODS'97)*, 117–121.
- Buneman, P., Davidson, S., Fan, W., Hara, C., & Tan, W. (2001). *Keys for XML*. *The 10th International World Wide Web Conference (WWW'10)*.
- Buneman, P., Fan, W., & Siméon, J. (2001). *Constraint for semistructured data and XML*. *SIGMOD Record*, 30(1).
- Chakravarthy, U. S., Grant, J., & Minker, J. (1990). *Logic-based approach to semantic query optimisation*. *ACM Transactions on Database Systems* 15, 2, 162–207.

- Chawathe, S., Abiteboul, S., & Widom, J. (1998). *Representing and Querying Changes in Semistructured Data*. *International Conference on Data Engineering*, 4–13.
- Document Object Model (DOM) Level 1 Specification* (Second Edition), 2000. (2000). [On-Line]. Available: <http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/>
- Fan, W., & Siméon, S. (2000). *Integrity constraints for XML*. *Nineteenth ACM Symposium on Principles of Database Systems (PODS 2000)*, 23–34.
- Fan, W., Kuper, G. M., & Siméon, J. (2001). *A unified constraint model for XML*. The 10th International World Wide Web Conference (WWW'10).
- Fernández M. (1996). *UnQL Project*. [On-Line]. Available: <http://www.research.att.com/projects/unql/>
- Fernandez, M., & Suciu, D. (1998). *Optimising Regular Path Expressions Using Graph Schemas*. *14th International Conference on Data Engineering*, 14–23.
- Goldman, R., McHugh, J., & Widom, J. (1999). *From Semistructured Data to XML: Migrating the Lore Data Model and Query Language*. *Workshop on the Web and Databases (WebDB '99)*, 25–30.
- Grahne, G., & Thomo, A. (2000). *An optimisation technique for answering regular path queries*. *International Workshop on the Web and Databases (WebDB 2000 Informal Proceedings)*, 99–104.
- Grahne, G., & Thomo, A. (2001). *Algebraic rewritings for optimising regular path queries*. *International Conference on Database Theory (ICDT 2001)*, 301–315.
- Lakshmanan, L.V.S., Sadri, F., & Subramanian, I.N. (1996). *A declarative language for querying and restructuring the WEB*. *Sixth International Workshop on Research Issues in Data Engineering (RIDE'96)*, 12–21.
- Lee, D., & Chu W. W. (2000). *Comparative Analysis of Six XML Schema Languages*. *SIGMOD Record*, 29(3), 76–87.
- Lee, D., & Chu, W. W. (2000). *Constraints-preserving transformation from XML document type definition to relational schema*. *International Conference on Conceptual Modeling (ER 2000)*, 323–338.
- Levy, A. Y., Mumick, I. S., & Sagiv, Y. (1994). *Query optimisation by predicate move around*. *Proceedings of the 20th Very Large Data Base Conference (VLDB 1994)*, 96–107.
- McHugh, J., & Widom, J. (1999). *Query optimisation for XML*. *Proceedings of the 25th Very Large Data Bases Conference (VLDB 1999)*, 315–326.
- Mihaila, G. (1996). *WebSQL: an SQL-like query language for the WWW*. MSc. Thesis, University of Toronto, 1996. [On-Line]. Available: <http://www.cs.toronto.edu/~websql/>
- Nestorov, S., Abiteboul, S., & Motwani, R. (1998). *Extracting Schema from Semistructured Data*. *ACM SIGMOD International Conference on Management of Data (COMAD'98)*, 295–306.
- Özsu, M. T. & Valduriez, P. (1999). *Principles of Distributed Database Systems*. Second Edition. Prentice Hall.
- Resource Description Framework (RDF) model and syntax specification*. (1999). [On-Line]. Available: <http://www.w3.org/TR/REC-rdf-syntax>.
- Resource Description Framework (RDF) Schema Specification 1.0*. W3C Candidate Recommendation, 27 March 2000. (2000). [On-Line]. Available: <http://www.w3.org/TR/2000/CR-rdf-schema-20000327>

- Silberschatz, A., & Zdonik, S. B. (1996). *Strategic Directions in Database Systems - Breaking Out of the Box*. *Computing Surveys* 28(4): 764–778.
- Ullman, J. (1989). *Principles of Database and Knowledge-Base Systems. Volumes 1 and 2*. Computer Science Press, New York.
- XML 1.0 W3C Recommendation February 1998. (1998). [On-Line]. Available : <http://www.w3.org/XML/>.
- XML Formal Semantics W3C Working Draft 7 June 2001. (2001). [On-Line]. Available : <http://www.w3.org/TR/query-semantics/>
- XML Schema W3C Proposed Recommendation 30 March 2001. (2001). [On-Line]. Available : <http://www.w3.org/XML/Schema#dev>
- XSL. The eXtensible Stylesheet Language. W3C Recommendation 16 November 1999. (1999). [On-Line]. Available: <http://www.w3.org/Style/XSL>

Chapter X: Integrity Maintenance in Extensible Databases

Ulrich Schiel, Universidade Federal da Paraiba,

Brazil

INTRODUCTION

The use of databases for advanced applications is a rapidly growing and changing field, due to the continuous incorporation of new technologies and media in current systems. Whereas in the near past Database Management Systems (DBMS) mainly use to store and manage tabular data, now they need to model complex structured objects, multimedia data, semi-structured and unstructured documents. Each of these improvements has its own semantics and complexity.

In order to allow an adequate description of database applications, data models are used to describe the conceptual schema of the database. If new categories of applications need to be incorporated or created, and the data model does not fit well with these applications, the model itself must be expanded. The semantics of the new constructs must be defined and the integrity of objects in the new constructs must be guaranteed.

Since a DBMS is in general not expandable, except for future versions of the same product, there are two alternatives: (i) to move the whole application to another system that is capable to adequately process the new structures, or (ii) to develop specific routines, probably with its own storage systems, in order to incorporate the new application. Clearly both solutions are unsatisfactory.

The first solution is only applicable if there exists a DBMS that considers the new structures. Even if it exists, moving to the new environment means reimplementing of the application, and this is very traumatic and demands a lot of time and money. The

other solution, to expand existing applications by special modules is more straightforward, but creates an unbalanced heterogeneous system, combining a databases with a file system. This generates problems of integration and does not allow a unified view of the data of the application.

Despite actual existing DBMS consider many modern database concepts, such as object-orientation and triggers, there are a lot of applications needing more. Concepts of temporal databases, geographic databases, hypertext (Web-) databases are not well attended.

In this chapter we introduce an approach of defining the semantics of a complex data model by means of general (schema-) integrity constraints integrated to the system as rules. This approach allows an easy way to define the semantics of complex data models. The rules systems can, at any time, be expanded in order to incorporate concepts of new applications and can also be used to add application specific integrity constraints. Therefore, data model specific constraints and application specific constraints are treated in a unified manner.

INTEGRITY IN DATABASES

Integrity maintenance in a database is achieved with two kinds of integrity constraints: *implicit integrity constraints* and *explicit integrity constraints* (Elmasri & Navathe, 1999).

Implicit integrity constraints, also denoted as schema constraints, are constraints defined in the conceptual database schema using the language of a data model, including attribute types, keys, null values, relationship cardinalities, generalizations, and aggregations. If we use a complex semantic data model to describe the conceptual schema, several implicit constraints are built in the schema, which reflects the expressiveness of the underlying data model.

Schema constraint satisfaction can be achieved mainly by three distinct approaches: (1) the DBMS supports the data model completely, and therefore its semantics is embedded in the software of the DBMS; (2) with the mapping of the conceptual schema into an internal schema, supported by the DBMS, the implicit constraints are implemented in the structure of the internal schema and, for features not foreseen in the model of the internal schema, create some controlling procedures; (3) the semantics of the data model is described in form of rules which are able to guarantee full semantic integrity and may be achieved in the DBMS. Since the rules are model dependent, and not application dependent, they are mapped only once to the internal schema. If solution (1) can be applied, it is the most straightforward and efficient one. If we do not have the adequate DBMS, solutions (2) or (3) may be applied. Solution (3) is better than (1) and (2) with regard to generality and flexibility. It is generic since it applies to all data models to be considered, and flexible because we can, at any time, add new rules in order to capture changes in the data model itself.

Instead of the richness of a semantic data model, it remains constraints, which cannot be expressed in the structure of the conceptual schema. These are the explicit integrity constraints. A constraint language must be used for this case. Actually, the importance of this topic has been recognized, and special databases, containing constraints as ordinal data, so called Constraint Databases are under investigation (Ramakrishnan & Stuckey, 1997) (Kupfer, Libkin & Paradaens, 2000). For systems using the relational language SQL, the typical language elements for expressing explicit constraints are the ASSERTION and TRIGGER commands. Also expressions of the relational algebra can be used to create predicates as integrity constraints (Ullman & Widom, 1997). For instance, if E1 and E2 are two relational expressions, we can state a constraint as $E1 = \emptyset$ or $E1 \neq E2$.

The most complete language for expressing constraints in object-oriented databases is OCL - Object Constraint Language (UML, 1997). OCL is part of the UML - Unified Modeling Language specification. It is a very rich formal language able to specify implicit constraints of applications modeled with the UML language. OCL allows the statement of invariants on classes and types, describe pre- and post-conditions on operations, guards, and so on. An OCL expression can refer to types, classes, properties, and datatypes. For instance, the constraint that "Married people are of age ≥ 18 " is attached to the class *Person* as

```
Person
self.wife ->. notEmpty implies self.wife.age >= 18 and
self.husband ->. notEmpty implies self.husband.age >= 18
```

OCL is a pure expression or declarative language. This means that if an OCL expression is evaluated it always returns a value and has no direct side-effect in the system. The user must program his side-effects based on the value returned. Another restriction is that it only acts at the application level. It is not possible to state generic constraints about classes, generalizations, associations, or other constructs of a data model. Even at the application level, there is no explicit reference to UML stereotypes such as generalization, aggregation and composition.

This chapter presents a formalism, which enables the specification and enforcement of both implicit and explicit integrity constraints. Considering the intension/extension dimension of the ANSI/SPARC DBMS architecture proposal (Burns, 1986), which divides a database description into four levels (Metadata Model, Data Model, Application Model and Application Data), the formalism is intended to allow the specification of the semantics at the Data Model level as well as at the Application Model level.

Implicit constraints are located at the Data Model level and explicit constraints are located at the Conceptual schema/Application Model level. Using this approach it is also easy to extend the data model itself, in order to incorporate new technologies and concepts into the model. This update is done adding new rules at the Data Model level.

The new concepts are defined at the Metadata Model level and their semantics established by the creation of new implicit constraints at the Data Model level.

The formalism described in this chapter uses a rules model, based on the well-known ECA-Rules (event-condition-action) (Dayal, 1988). Two kinds of rules are distinguished: *Dynamic Axioms* and *Side Effects*. Dynamic Axioms (DAs) inhibit operations that violate the semantic integrity and Side Effects (SEs) react on potential integrity violations and trigger actions to recompose the integrity. A side effect of an implicit integrity constraint is called a System Side-Effect (SSE), whereas application dependent side-effects are called User Side-Effects (USE).

Despite the formalism can be used or adapted for any data model we will use a relatively complex object-oriented data model, in order to prove its generality.

The data model we consider is based on the notions of class as a collection of objects. Relationships are defined between classes with two directions and for each direction minimum and maximum cardinalities are associated. The well-known abstractions of generalization/specialization, aggregation and grouping can hold between classes. Generalization and aggregation has been defined first in (Smith & Smith, 1977) as an extension of the relational data model. It has been considered in the most semantic data models, such as SDM (Hammer & McLeod, 1981), THM (Schiel, 1983) and ACM/PCM (Brodie & Silva, 1984). Also object-oriented models, such as ODMG (Catell, 2000) and UML (Jacobson et al., 2000) use generalization and aggregation. The abstraction we call grouping is also contained in SDM, THM and ACM/PCM (here called association), and is discussed in (Odell, 1998) as power type. In the UML world it is considered as composition by some authors (Page-Jones, 1999). The main difference between aggregation and grouping is that the first one is heteromeous, with a fixed number of components, and grouping is homeomeous with a variable number of components (Page-Jones, 1999). Details of these concepts will become clear during the text.

The system presented in this chapter is based on the data model TOM (Schiel, 1991), which is richer than ODMG and UML, whereas UML has some concepts not included in TOM. Therefore, its use for ODMG based applications, the exceeding DAs and SSEs must be excluded and for UML an adaptation is necessary.

ELEMENTARY OPERATIONS AND STRUCTURAL PREDICATES

We define some primitive operations and predicates over objects, relationships and classes. The notation for operations follows the notion of message in object-oriented systems, and has the form: <receptor> <message (parameters)>.

Operations

- *C create (e)*: creates object *e* in class *C*.

- $e_delete(C)$: deletes object e from class C .
- $e_establish(r, e_1, e_2)$: establishes relationship r between objects e_1 and e_2 .
- $e_remove(r, e_1, e_2)$: removes relationship r between objects e_1 and e_2 .

There are two special operations for the insertion and removal of elements of groups. The effect is similar to the creation and deletion of objects, only that they act on group objects and not on classes:

- $g_gr-insert(o)$: inserts o as a new element of the group g .
- $g_gr-delete(o)$: eliminates o of the group g .

In order to verify facts in the database, several predicates are defined. We divide them into basic predicates and hierarchies.

Basic Predicates

- $in(e, C)$: object e is an instance of class C .
- $is-rel(e_1, r, e_2)$: objects e_1 and e_2 are related by r .
- min_r, max_r

Each relationship r has associated with it a cardinality expressed by a pair (min_r, max_r) , which means that each instance of the starting class is associated at least to min_r and at most to max_r instances of the target class.

Hierarchies

Generalization/Specialization

Means the creation of more specific subclasses of a given generic class. We can apply several specializations, characterized by several *roles*. Each role is given by a predicate $p(e)$ that establishes to which subclass a specific object can be associated. For instance, we can have $sex(PERSON) = MALE, FEMALE$ and $age(PERSON) = YOUNG, MIDDLE, OLD$. For a given person p the predicate $age_{YOUNG}(p)$ is true if p is a young person.

- $is-a(C_1, C_2, p)$: class C_1 is a subclass of C_2 by role p .
- $p_G(e)$

The specialization role is a disjunctive predicate $p_G(e) = p_{c1}(e) \vee \dots \vee p_{cn}(e)$ where e is an object of a generic class G and $p_{ci}(e)$ is true iff e is an instance of the subclass C_i of G .

- *disjunctive* (D, C_1, \dots, C_n)

D is a disjunctive generalization of C_1, \dots, C_n , i.e., each instance of D can be contained in only one subclass C_i of D .

- *covering* (D, C_1, \dots, C_n)

G is a covering generalization of C_1, \dots, C_n , i.e., each instance of D must occur in at least one subclass C_i of D .

Aggregation

Means the creation of new object as combination of several distinct parts.

- *is-part*(e_1, e_2): object e_1 is one component of the aggregate e_2 .
- *aggregation*(A, C_1, \dots, C_n): class A is an aggregation of classes C_1, \dots, C_n .
- *aggregation-r*(A, C_1, C_2, r)

Class A is an aggregation of classes C_1 e C_2 by relationship r iff the related objects are just the elements of the aggregated class A .

- *agg-inherit*(A, C_1, \dots, C_n, r)

A is an aggregation of C_1, \dots, C_n and the component classes inherit the relationship r .

- *agg-comp-inherit*(A, C_1, \dots, C_n, r, s)

A is an aggregation of C_1, \dots, C_n and the inheritance is determined by a function s .

Grouping

Several objects of a class are grouped together to form higher order objects.

- $is\text{-}elem(e_1, e_2)$: e_1 is an element of the group object e_2 .
- $is\text{-}elem(C, G, p)$ G is a class of groups of elements of C and group containment is governed by predicate p , i.e. $p(e, g)$ holds iff e is an element of g .
- $disjoint\text{-}gr(C, D)$

D is a grouping of C and each instance of C occurs in at most one group of D .

- $covering\text{-}gr(C, D)$

D is a grouping of C and each instance of C occurs in at least one group of D .

DYNAMIC AXIOMS AND SIDE-EFFECTS

The semantic integrity of an application is maintained by two kinds of rules: Dynamic Axioms and Side Effects. The dynamic axioms are integrity restrictions that avoid that the user realizes updates that are not consistent with the conceptual schema, whereas the side effects execute auxiliary operations in order to recompose the integrity of the database. A set of *System-Side-Effects*, which guarantee the structural components of the data model, are presented. The application designer can add application dependent rules, called *User-Side-Effects*. All rules are of the form ON <event> IF <condition> DO <action>.

Dynamic Axioms

The action *abort* in a rule characterizes an integrity rule and avoids the execution of the operation on the event part.

DA1 ? An *establish* may not violate the maximal cardinality

ON x *establish*(r, y) IF $\#\{ \langle x, z \rangle / is\text{-}rel(x, r, z) \} = max, DO abort$

DA2 ? A *remove* may not violate the minimal cardinality

ON x *remove* (r, y) IF $\#\{ \langle x, z \rangle / is\text{-}rel(x, r, z) \} = min, DO abort$

DA3 ? The insertion of a new instance in a subclass must respect the role

ON C *create* (x) IF $is\text{-}a(C, D, p) \wedge not p_e(x) DO Abort$

DA4 ? In a *covering* generalization an object cannot remain only in the generic class

ON x delete (C)	IF $is-a(C, D, p) \wedge$	DO Abort
	covering (D, C_p, \dots, C_n) \wedge	
	$(C_i \triangleleft C \Rightarrow not\ x\ in\ C_i)$	

DA5 ? Group elements must satisfy the grouping predicate

DA6 ? For a *covering* grouping, a new instance of the element class must be in a group

System Side Effects

For system side effects the action part of the rule, given in the DO clause, contains a primitive operation necessary to maintain the integrity of the database.

Generalization

SSE1 ? An instance of a subclass must be also in the superclass

ON C create (x) IF $is-a(C, D) \wedge not\ x\ in\ D$ DO D create (x)

SSE2 ? A *create* may not violate the disjoint generalization

SSE3 ? A *delete* may not violate the *covering* generalization

SSE4 ? A new instance of a generic class must be inserted in the compatible subclasses

SSE5 ? An object deleted from a generic class must be deleted from all its subclasses

SSE6 ? If, as consequence of the change in a relationship, an object cannot remain in its subclass, it must be removed to the compatible new subclass

$$\begin{array}{lll}
 \text{ON } x \text{ establish } (r, y) & \text{IF } x \text{ in } C \wedge & \text{DO } x \text{ delete } (C) \\
 \vee x \text{ remove}(r, y) & \text{is-}\alpha(C, D, p) \wedge \text{is-}\alpha(C', D, p) \wedge & \\
 & p_c(x) \wedge o(\text{not } p_c(x)) & \\
 \text{ON } x \text{ remove } (r, y) & \text{IF } x \text{ in } C & \text{DO } C' \text{ create } (x) \\
 \vee x \text{ establish } (<r, y>) & \text{is-}\alpha(C, D, p) \wedge \text{is-}\alpha(C', D, p) & \\
 & \text{not } p_c(x) \wedge o(p_c(x)) &
 \end{array}$$

Aggregation

SSE7 ? An aggregate cannot lose one of its components

$$\begin{array}{lll}
 \text{ON } x \text{ delete } (C) & \text{IF is-part } (C, D) \wedge & \text{DO } y \text{ delete } (D) \\
 & \text{aggregate}(y, x_1, \dots, x_n) \wedge & \\
 & y \text{ in } D \wedge x \in \{x_1, \dots, x_n\} &
 \end{array}$$

SSE8 ? For an aggregation by relationship, the removal of a relationship eliminates the corresponding aggregate

$$\begin{array}{lll}
 \text{ON } x \text{ remove}(r, y) & \text{IF aggregate-}r(D, C_1, C_2, r) & \text{DO } <x, y> \text{ delete}(D) \\
 & \wedge x \text{ in } C_1 \wedge y \text{ in } C_2 &
 \end{array}$$

SSE9 ? For an aggregation by relationship, the establishing of a relationship creates the corresponding aggregate

$$\begin{array}{lll}
 \text{ON } r \text{ establish } (<x, y>) & \text{IF aggregate-}r(D, C_1, C_2, r) & \text{DO } D \text{ create } (<x, y>) \\
 & \wedge x \text{ in } C_1 \wedge y \text{ in } C_2 &
 \end{array}$$

SSE10 ? The creation of an aggregate creates also its parts

$$\begin{array}{lll}
 \text{ON } D \text{ create } (y) & \text{IF aggregate } (D, C_1, \dots, C_n) & \text{DO } C_i \text{ create } (x_i) \\
 & \text{is-part } (x_i, y) &
 \end{array}$$

SSE11 ? The creation of an aggregate defined by a relationship, also the relationship must be established

$$\begin{array}{lll}
 \text{ON } D \text{ create } (y) & \text{IF aggregate-}r(D, C_1, C_2, r) & \text{DO } C_1 \text{ create } (x_1) \wedge \\
 & \text{is-rel } (C_1, r, C_2) & C_2 \text{ create } (x_2) \wedge \\
 & \text{is-part } (x_1, y) \wedge \text{is-part } (x_2, y) & x_1 \text{ establish } (<r, x_2>)
 \end{array}$$

Grouping

SSE12 ? If $p(x, g)$ holds and x is inserted in the element class, it must also be added to g

ON $C \text{ create}(x)$ IF $\text{is-elem}(C, G)$ DO $g \text{ gr-insert}(x)$
 $g \text{ in } G \wedge p(x, g)$

SSE13 ? In a *covering* grouping, if an object is eliminated from a group, it must also be eliminated from the elements class

ON $x \text{ gr-delete}(g)$ IF $\text{gr-covering}(C, G) \wedge x \text{ in } C$ DO $x \text{ delete}(C)$

SSE14 ? If an object is deleted from the elements class, it must also be removed from the groups

ON $x \text{ delete}(C)$ IF $\text{is-elem}(C, G, p) \wedge x \text{ in } g$ DO $x \text{ gr-delete}(g)$

SSE15 ? The elements of a new group must be on the element class

ON $G \text{ create}(g)$ IF $\text{is-elem}(C, G) \wedge$ DO $C \text{ create}(x)$
 $p(x, g) \wedge \text{not } x \text{ in } C$

SSE16 ? In a *covering* grouping, the removal of a group removes also its elements from the element class

ON $g \text{ delete}(G)$ IF $\text{is-elem}(C, G)$ DO $x \text{ delete}(C)$
 $\text{gr-covering}(C, G) \wedge x \text{ in } g$
 $\exists h (x \text{ in } h \wedge h \text{ in } G \wedge h \neq g)$

SSE17 ? In a *disjoint* grouping, a new element of a group must be eliminated from other groups

ON $g \text{ gr-insert}(x)$ IF $\text{disjoint}(C, G)$ DO $x \text{ gr-delete}(h)$
 $\exists h (x \text{ in } h \wedge h \neq g)$

Relationships

SSE18 ? The *delete* of an object must remove all its relationships

ON $x \text{ delete}(C)$ IF $\text{is-rel}(x, r, y)$ DO $x \text{ remove}(r, y)$

SSE19 ? Each relationships has an inverse

ON $r \text{ establish}(\langle x, y \rangle)$ IF $\text{not is-rel}(y, \text{inv}(r), x)$ DO $y \text{ establish}(\text{inv}(r), x)$

EXTENSION

In this section we show how the system can be expanded in order to include new functionalities.

Suppose we want to expand the existing data model with two new concepts. First, we would be able to model applications who needs valid-time temporal objects (Tansel et al., 1993). In order to consider Temporal Databases, the following additions are needed.

Classes and relationships may be defined as temporal, by the predicates:

- $t\text{-Class}(C', C) ? \text{aggregation}(C', C, I)$

A temporal class C' is obtained aggregating a ordinal class C with the class of time intervals I .

- $t\text{-Rel}(r', C_1, C_2) ? \text{aggregation}(r, C_1, C_2, I)$

A temporal relationship r' between two classes C_1 e C_2 aggregates these two classes with I

Now we define a dynamic axiom that avoids the creation of objects with valid time intersecting the valid time of this object in the database.

DA7 ? In a temporal class it is not allowed to create instances with valid-time intersecting with its existing time

ON C' create $\langle x, (t_1, t_2) \rangle$ IF $t\text{-Class}(C', C, I) \wedge$ DO Abort
 $\langle x, (t_1', t_2') \rangle \text{ in } C' \wedge$
 $\text{overlap}((t_1, t_2), (t_1', t_2'))$

As side effects we consider the following. The first one allows the user to define an object without considering if it is temporal or not. He just creates the object and the side effect attaches the default valid time.

SSE20 ? If a temporal object is created, its creation time must be established

ON $C \text{ create}(x)$ IF $t\text{-Class}(C', C) \wedge t = \text{now}$ DO $C' \text{ create}(\langle x, (t, \text{now}) \rangle)$

SSE21 ? In a temporal class, a deleted object is moved to the past

ON $x \text{ delete}(C)$ IF $t\text{-Class}(C', C) \wedge t = \text{now}$ DO $\langle x, (t, \text{now}) \rangle \text{ delete}(C')$
 $\langle x, (t_o, \text{now}) \rangle \text{ in } C' \quad C' \text{ create}(\langle x, (t_o, t) \rangle)$

SSE22 ? On the creation of a temporal relationship, the time must also be created

ON $x \text{ establish}(\langle r, y \rangle)$ IF $t\text{-Rel}(r, C_1, C_2) \wedge t = \text{now}$ DO $r \text{ create}(\langle x, y, (t, \text{now}) \rangle)$

SSE23 ? The removal of a temporal relationship must be moved to the past

ON $\langle x, r, y \rangle \text{ remove}$ IF $t\text{-Rel}(r, C_1, C_2) \wedge t = \text{now}$ DO $\langle x, y, (t, \text{now}) \rangle \text{ delete}$
 $\langle x, y, (t_o, \text{now}) \rangle \text{ in } R' \quad R' \text{ create}(\langle x, y, (t_o, t) \rangle)$

Suppose now that the user has an application with classes EMPLOYEE, SALARY and STATUS and with the relationships EMPLOYEEhas-salarySALARY and EMPLOYEEhas-statusSTATUS. We want to state a User Side Effect (USE) which guarantees that, ever when the salary of an employee becomes greater that \$100,000 his status must be fixed to be "4". Therefore we have:

USE1: ON $e \text{ establish}(\text{has-salary}, s)$ IF $s > 100,000 \wedge$ DO $e \text{ remove}(\text{has-status}, s_o) \wedge$
 $e.\text{has-status} = s_o \quad e \text{ establish}(\text{has-status}, '4')$
 $\wedge s_o \neq '4'$

CONCLUSION

In this chapter a system for the enforcement of implicit and explicit integrity constraints for a complex object oriented data model has been presented. This system is flexible enough to be used for extensible data models, also known as open data models, which allows the incorporation of new capabilities in the data model, in order to facilitate the modelling of new application categories.

The idea of system side effects has first been developed together with the Temporal Hierarchic Model - THM (Schiel, 1983), later improved to the Temporal Object Model - TOM (Schiel, 1991). A running prototype of the dynamic axioms and side effects for the TOM model has been implemented together with a mapping module, which transforms an Object Schema to the relational DBMS Oracle (daSilva, 2000). According to the relational schema generated by this mapping, the rules are converted to adequate triggers of the DBMS and are executed on the database.

REFERENCES

- Burns, T. (1986) *Reference Model for DBMS Standardization*, *ACM SIGMOD RECORD*, 15(1), 19–58
- Brodie, M. and Silva, E. (1982). *Active and Passive Component Modelling: ACM/PCM*, in *Information Systems Design Methodologies: A Comparative Review*, W.Olle, H.Sol and A.Verrijn-Stuart (eds), North Holland.
- Catell et al. (2000). *Object Database Standard: Odmg 3.0*. Morgan Kaufman.
- Dayal, U., Buchanan, A. P. and McCarthy, D. R. (1988). *Rules are Objects Too: A Knowledge Model for Active Object Oriented Database Systems*, in *Proceedings of the 2nd International Workshop on Object Oriented DataBase Systems*, LNCS 334, Springer Verlag, 129–143.
- Elmasri, R. and Navathe, S. (1999). *Fundamentals of Database Systems*. 3rd ed., Benjamin Cummings.
- Hammer, M. and McLeod, D. (1981). *Database Descriptions with SDM: A Semantic Database Model* *ACM TODS*, 6(3).
- Jacobson, I., Booch, G. and Rumbaugh, J. (1998). *The Unified Modeling Language - User Guide*, Addison Wesley.
- Kupfer, G, Libkin, L. and Paradaens, J. (2000). *Constraint Databases*, Springer Verlag.
- Odell, J. (1998). *Advanced Object-Oriented Analysis & Design using UML*, Cambridge University Press.
- Page-Jones, M. (1999). *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley.
- Ramakrishnan, R. and Stuckey, P. (1997). *Constraints and Databases*, Kluwer Academic.
- Schiel, U. (1983). *An Abstract Introduction to the Temporal-Hierarchic Data Model (THM)*. In *Proceedings of the 9th International Conference on Very Large Data Bases*, Florence, p. 322–330.
- Schiel, U. (1991). *An Open Environment for Objects with Time and Versioning*. In *Proceedings East Europe*, Bratislava, p. 116–125.
- Smith, J. M. and Smith, D. (1977). *Database abstractions: Aggregation and Generalization*. *ACM TODS* Vol. 2, no. 2.
- da Silva, M. N. (2000). *Um sistema de Controle de Integridade para um Modelo de Dados Aberto*, Master thesis, COPIN/UFPB, August.
- Tansel, A., Clifford, J., Gadia S., Jajodia S. e Segev A., and Snodgrass, R. (1993) *Temporal Databases*, Benjamin/Cummings Publishing Company, Inc.
- UML (1997). *Object Constraint Language Specification - version 1.1*. On line, available at: <http://www-4.ibm.com/software/ad/library/standards/ocl.html>.
- Ullman, J. and Widom, J. (1997). *A First Course in Database Systems*, Prentice Hall.

List of Figures

Chapter II: Database Integrity—Fundamentals and Current Implementations

[Figure 1:](#) Restriction Mapping

[Figure 2:](#) EER of the MO components

[Figure 3:](#) Physical model of the MO components

[Figure 4:](#) The medicare organization example

Chapter III: Preserving Relationship Cardinality Constraints in Relational Schemata

[Figure 1a:](#) Binary relationship example

[Figure 1b:](#) All possible occurrences of relationship *I* of Figure 1a

[Figure 2:](#) Calculus of different cardinality values from Figure 1b

[Figure 3a:](#) Ternary relationship example

[Figure 3b:](#) All possible occurrences of relationship *I* of Figure 3a

[Figure 4:](#) Cardinality constraints using UML

[Figure 5:](#) Cardinality constraints using MERISE

[Figure 6:](#) Cardinality constraints using ER model

[Figure 7:](#) Cardinality constraints using Teorey model

[Figure 8:](#) Ternary relationships versus binary relationships (first solution)

[Figure 9:](#) Ternary relationships versus binary relationships (second solution)

[Figure 10:](#) ER cardinality constraints using Chen's style

[Figure 11:](#) ER cardinality constraints using MERISE approach

[Figure 12:](#) An example of functional integrity constraint in MERISE

[Figure 13:](#) Ternary relationship using MERISE cardinality constraints

[Figure 14:](#) Ternary relationship using Chen's style (first solution)

[Figure 15:](#) Ternary relationship using Chen's style (second solution)

[Figure 16:](#) A N:M relationship

[Figure 17:](#) Relational model transformation of Figure 16

[Figure 18:](#) A 1:N relationship

[Figure 19:](#) Relational model standard transformation of Figure 18

[Figure 20:](#) A 1:1 relationship

[Figure 21:](#) Relational model transformation of Figure 20

[Figure 22:](#) A 1:1 relationship

[Figure 24:](#) Standard transformation of an ternary relationship (Figure 23)

[Figure 23:](#) A ternary relationship

[Figure 25:](#) A ternary relationship with minimum cardinality 0

[Figure 26:](#) Relational model transformation of Figure 25

[Figure 27:](#) N-ary relationship $n > 2$

[Figure 28:](#) N-ary relationship standard transformation into the relational model

[Figure 29a:](#) Ternary relationship example

[Figure 29b:](#) I relationship occurrences, at time *t*

Chapter IV: Integrity Constraints in an Active Database Environment

[Figure 1:](#) Sequence of two update statements on the table SPEED_LIMIT

[Figure 2:](#) Situation after a real firing of the monitoring trigger is produced

[Figure 3](#): Position 2

Chapter V: Integrity Constraints in Spatial Databases

[Figure 1](#): Graphic notation for the basic classes

[Figure 2](#): Geo-field classes

[Figure 3](#): Geo-object classes

[Figure 4](#): Relationships

[Figure 5](#): Cardinality

[Figure 6](#): Generalization

[Figure 7](#): Spatial generalization examples

[Figure 8](#): UML aggregation

[Figure 9](#): Aggregation between conventional and georeferenced classes

[Figure 10](#): Spatial aggregation ("whole part")

[Figure 11](#): Conceptual generalization

[Figure 12](#): Conceptual generalization with a conventional class

[Figure 13](#): Application example

Chapter VI: Consistent Queries Over Databases With Integrity Constraints

[Figure 1](#)

[Figure 2](#)

[Figure 3](#)

[Figure 4](#)

[Figure 5](#)

[Figure 6](#)

[Figure 7](#)

[Figure 8](#)

[Figure 9](#)

[Figure 10](#)

[Figure 11](#)

[Figure 12](#)

[Figure 13](#)

Chapter VIII: Functional Dependencies for Value Based Identification in Object-Oriented Databases

[Figure 1](#): Some examples of VBICs

[Figure 2](#): VBIC by inheritance

[Figure 3](#): Ambiguity of an FDI at object schema level

[Figure 4](#): Examples of OFDs

[Figure 5](#): Different OFDs with identical set notation

[Figure 6](#): Simple schema with state and relational representations (object values, edge and node labels are omitted from the state graph)

[Figure 7](#): Example of a state

[Figure 8](#): Implications between OFD semantics

[Chapter IX](#): Integrity Issues in the Web—Beyond Distributed Databases

[Figure 1](#): Extending the cube

[Figure 2](#): Data model in LORE

[Figure 3](#): Data model in XML

[Figure 4](#): Example of XML document representing a publication page

[Figure 5](#): Example of XML schema for a publication page

[Figure 6](#): XML application

[Figure 7a](#): Monad laws

[Figure 7b](#): Equivalence between projection and iteration

[Figure 7c](#): Optimisation laws

[Figure 8](#): Constraint propagation

[Figure 9](#): Semantic Web

List of Tables

[Chapter II](#): Database Integrity—Fundamentals and Current Implementations

[Table 1](#): Real world restrictions and their correlates in the database world

[Chapter III](#): Preserving Relationship Cardinality Constraints in Relational Schemata

[Table 1](#): Cardinality constraints summary

[Table 2](#): Summary of the differences among cardinality constraints

[Table 3](#): Semantic loss in cardinality (o, n) updating transactions

[Table 4](#): Semantic loss in cardinality (1,n) updating transactions

[Table 6](#): Semantic loss in cardinality (1,n) updating transactions

[Table 5](#): Semantic loss in cardinality (o,n) updating transactions

[Table 7](#): Semantic loss in cardinality (0,1) updating transactions

[Table 8](#): Semantic loss in cardinality (1,1) updating transactions

[Table 9](#): Semantic loss in cardinality (0,1) (0,1) updating transactions

[Table 10](#): Semantic loss in cardinality (1,1) (0,1) updating transactions

[Table 11](#): Semantic loss in cardinality (1,1) (1,1) updating transactions

[Table 12](#): Ternary relationship classification

Chapter V: Integrity Constraints in Spatial Databases

[Table 1:](#) Geo-field integrity rules

[Table 4:](#) Connectivity rules

[Table 2:](#) Geo-object constraints

[Table 3:](#) Spatial relationship integrity rules

[Table 5:](#) Spatial aggregation integrity rules

[Table 6:](#) Spatial integrity constraints from the example