

**THE UNIVERSITY OF CALGARY**

**SUPPORTING DOMAINS IN RELATIONAL  
DATABASE SYSTEMS**

**BY**

**Zhao Zhang**

**A THESIS**

**SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF MASTER OF SCIENCE**

**DEPARTMENT OF COMPUTER SCIENCE**

**CALGARY, ALBERTA**

**June, 1992**

**© Zhao Zhang 1992**



National Library  
of Canada

Acquisitions and  
Bibliographic Services Branch

395 Wellington Street  
Ottawa, Ontario  
K1A 0N4

Bibliothèque nationale  
du Canada

Direction des acquisitions et  
des services bibliographiques

395, rue Wellington  
Ottawa (Ontario)  
K1A 0N4

*Your file    Votre référence*

*Our file    Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-79105-5

Canada

Name **ZHAO ZHANG**

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

**Computer Science**

SUBJECT TERM

**0984 U-M-I**

SUBJECT CODE

## Subject Categories

### THE HUMANITIES AND SOCIAL SCIENCES

#### COMMUNICATIONS AND THE ARTS

Architecture .....0729  
Art History .....0377  
Cinema .....0900  
Dance .....0378  
Fine Arts .....0357  
Information Science .....0723  
Journalism .....0391  
Library Science .....0399  
Mass Communications .....0708  
Music .....0413  
Speech Communication .....0459  
Theater .....0465

#### EDUCATION

General .....0515  
Administration .....0514  
Adult and Continuing .....0516  
Agricultural .....0517  
Art .....0273  
Bilingual and Multicultural .....0282  
Business .....0688  
Community College .....0275  
Curriculum and Instruction .....0727  
Early Childhood .....0518  
Elementary .....0524  
Finance .....0277  
Guidance and Counseling .....0519  
Health .....0680  
Higher .....0745  
History of .....0520  
Home Economics .....0278  
Industrial .....0521  
Language and Literature .....0279  
Mathematics .....0280  
Music .....0522  
Philosophy of .....0998  
Physical .....0523

Psychology .....0525  
Reading .....0535  
Religious .....0527  
Sciences .....0714  
Secondary .....0533  
Social Sciences .....0534  
Sociology of .....0340  
Special .....0529  
Teacher Training .....0530  
Technology .....0710  
Tests and Measurements .....0288  
Vocational .....0747

#### LANGUAGE, LITERATURE AND LINGUISTICS

Language  
General .....0679  
Ancient .....0289  
Linguistics .....0290  
Modern .....0291  
Literature  
General .....0401  
Classical .....0294  
Comparative .....0295  
Medieval .....0297  
Modern .....0298  
African .....0316  
American .....0591  
Asian .....0305  
Canadian (English) .....0352  
Canadian (French) .....0355  
English .....0593  
Germanic .....0311  
Latin American .....0312  
Middle Eastern .....0315  
Romance .....0313  
Slavic and East European .....0314

#### PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy .....0422  
Religion  
General .....0318  
Biblical Studies .....0321  
Clergy .....0319  
History of .....0320  
Philosophy of .....0322  
Theology .....0469

#### SOCIAL SCIENCES

American Studies .....0323  
Anthropology  
Archaeology .....0324  
Cultural .....0326  
Physical .....0327  
Business Administration  
General .....0310  
Accounting .....0272  
Banking .....0770  
Management .....0454  
Marketing .....0338  
Canadian Studies .....0385  
Economics  
General .....0501  
Agricultural .....0503  
Commerce-Business .....0505  
Finance .....0508  
History .....0509  
Labor .....0510  
Theory .....0511  
Folklore .....0358  
Geography .....0366  
Gerontology .....0351  
History  
General .....0578

Ancient .....0579  
Medieval .....0581  
Modern .....0582  
Black .....0328  
African .....0331  
Asia, Australia and Oceania .....0332  
Canadian .....0334  
European .....0335  
Latin American .....0336  
Middle Eastern .....0333  
United States .....0337  
History of Science .....0585  
Law .....0398  
Political Science  
General .....0615  
International Law and  
Relations .....0616  
Public Administration .....0617  
Recreation .....0814  
Social Work .....0452  
Sociology  
General .....0626  
Criminology and Penology .....0627  
Demography .....0938  
Ethnic and Racial Studies .....0631  
Individual and Family  
Studies .....0628  
Industrial and Labor  
Relations .....0629  
Public and Social Welfare .....0630  
Social Structure and  
Development .....0700  
Theory and Methods .....0344  
Transportation .....0709  
Urban and Regional Planning .....0999  
Women's Studies .....0453

### THE SCIENCES AND ENGINEERING

#### BIOLOGICAL SCIENCES

Agriculture  
General .....0473  
Agronomy .....0285  
Animal Culture and  
Nutrition .....0475  
Animal Pathology .....0476  
Food Science and  
Technology .....0359  
Forestry and Wildlife .....0478  
Plant Culture .....0479  
Plant Pathology .....0480  
Plant Physiology .....0817  
Range Management .....0777  
Wood Technology .....0746  
Biology  
General .....0306  
Anatomy .....0287  
Biostatistics .....0308  
Botany .....0309  
Cell .....0379  
Ecology .....0329  
Entomology .....0353  
Genetics .....0369  
Limnology .....0793  
Microbiology .....0410  
Molecular .....0307  
Neuroscience .....0317  
Oceanography .....0416  
Physiology .....0433  
Radiation .....0821  
Veterinary Science .....0778  
Zoology .....0472  
Biophysics  
General .....0786  
Medical .....0760

#### EARTH SCIENCES

Biogeochemistry .....0425  
Geochemistry .....0996

Geodesy .....0370  
Geology .....0372  
Geophysics .....0373  
Hydrology .....0388  
Mineralogy .....0411  
Paleobotany .....0345  
Paleoecology .....0426  
Paleontology .....0418  
Paleozoology .....0985  
Palynology .....0427  
Physical Geography .....0368  
Physical Oceanography .....0415

#### HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences .....0768  
Health Sciences  
General .....0566  
Audiology .....0300  
Chemotherapy .....0992  
Dentistry .....0567  
Education .....0350  
Hospital Management .....0769  
Human Development .....0758  
Immunology .....0982  
Medicine and Surgery .....0564  
Mental Health .....0347  
Nursing .....0569  
Nutrition .....0570  
Obstetrics and Gynecology .....0380  
Occupational Health and  
Therapy .....0354  
Ophthalmology .....0381  
Pathology .....0571  
Pharmacology .....0419  
Pharmacy .....0572  
Physical Therapy .....0382  
Public Health .....0573  
Radiology .....0574  
Recreation .....0575

Speech Pathology .....0460  
Toxicology .....0383  
Home Economics .....0386

#### PHYSICAL SCIENCES

##### Pure Sciences

Chemistry  
General .....0485  
Agricultural .....0749  
Analytical .....0486  
Biochemistry .....0487  
Inorganic .....0488  
Nuclear .....0738  
Organic .....0490  
Pharmaceutical .....0491  
Physical .....0494  
Polymer .....0495  
Radiation .....0754  
Mathematics .....0405  
Physics  
General .....0605  
Acoustics .....0986  
Astronomy and  
Astrophysics .....0606  
Atmospheric Science .....0608  
Atomic .....0748  
Electronics and Electricity .....0607  
Elementary Particles and  
High Energy .....0798  
Fluid and Plasma .....0759  
Molecular .....0609  
Nuclear .....0610  
Optics .....0752  
Radiation .....0756  
Solid State .....0611  
Statistics .....0463

##### Applied Sciences

Applied Mechanics .....0346  
Computer Science .....0984

Engineering  
General .....0537  
Aerospace .....0538  
Agricultural .....0539  
Automotive .....0540  
Biomedical .....0541  
Chemical .....0542  
Civil .....0543  
Electronics and Electrical .....0544  
Heat and Thermodynamics .....0348  
Hydraulic .....0545  
Industrial .....0546  
Marine .....0547  
Materials Science .....0794  
Mechanical .....0548  
Metallurgy .....0743  
Mining .....0551  
Nuclear .....0552  
Packaging .....0549  
Petroleum .....0765  
Sanitary and Municipal .....0554  
System Science .....0790  
Geotechnology .....0428  
Operations Research .....0796  
Plastics Technology .....0795  
Textile Technology .....0994

#### PSYCHOLOGY

General .....0621  
Behavioral .....0384  
Clinical .....0622  
Developmental .....0620  
Experimental .....0623  
Industrial .....0624  
Personality .....0625  
Physiological .....0989  
Psychobiology .....0349  
Psychometrics .....0632  
Social .....0451



Nom \_\_\_\_\_

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

SUJET

--	--	--	--

U·M·I

CODE DE SUJET

## Catégories par sujets

### HUMANITÉS ET SCIENCES SOCIALES

#### COMMUNICATIONS ET LES ARTS

Architecture .....	0729
Beaux-arts .....	0357
Bibliothéconomie .....	0399
Cinéma .....	0900
Communication verbale .....	0459
Communications .....	0708
Danse .....	0378
Histoire de l'art .....	0377
Journalisme .....	0391
Musique .....	0413
Sciences de l'information .....	0723
Théâtre .....	0465

#### ÉDUCATION

Généralités .....	515
Administration .....	0514
Art .....	0273
Collèges communautaires .....	0275
Commerce .....	0688
Économie domestique .....	0278
Éducation permanente .....	0516
Éducation préscolaire .....	0518
Éducation sanitaire .....	0680
Enseignement agricole .....	0517
Enseignement bilingue et multiculturel .....	0282
Enseignement industriel .....	0521
Enseignement primaire .....	0524
Enseignement professionnel .....	0747
Enseignement religieux .....	0527
Enseignement secondaire .....	0533
Enseignement spécial .....	0529
Enseignement supérieur .....	0745
Évaluation .....	0288
Finances .....	0277
Formation des enseignants .....	0530
Histoire de l'éducation .....	0520
Langues et littérature .....	0279

Lecture .....	0535
Mathématiques .....	0280
Musique .....	0522
Orientation et consultation .....	0519
Philosophie de l'éducation .....	0998
Physique .....	0523
Programmes d'études et enseignement .....	0727
Psychologie .....	0525
Sciences .....	0714
Sciences sociales .....	0534
Sociologie de l'éducation .....	0340
Technologie .....	0710

#### LANGUE, LITTÉRATURE ET LINGUISTIQUE

Langues	
Généralités .....	0679
Anciennes .....	0289
Linguistique .....	0290
Modernes .....	0291
Littérature	
Généralités .....	0401
Anciennes .....	0294
Comparée .....	0295
Médiévale .....	0297
Moderne .....	0298
Africaine .....	0316
Américaine .....	0591
Anglaise .....	0593
Asiatique .....	0305
Canadienne (Anglaise) .....	0352
Canadienne (Française) .....	0355
Germanique .....	0311
Latino-américaine .....	0312
Moyen-orientale .....	0315
Romane .....	0313
Slave et est-européenne .....	0314

#### PHILOSOPHIE, RELIGION ET

THEOLOGIE	
Philosophie .....	0422
Religion .....	
Généralités .....	0318
Clergé .....	0319
Études bibliques .....	0321
Histoire des religions .....	0320
Philosophie de la religion .....	0322
Théologie .....	0469

#### SCIENCES SOCIALES

Anthropologie	
Archéologie .....	0324
Culturelle .....	0326
Physique .....	0327
Droit .....	0398
Économie	
Généralités .....	0501
Commerce-Affaires .....	0505
Économie agricole .....	0503
Économie du travail .....	0510
Finances .....	0508
Histoire .....	0509
Théorie .....	0511
Études américaines .....	0323
Études canadiennes .....	0385
Études féministes .....	0453
Folklore .....	0358
Géographie .....	0366
Gérontologie .....	0351
Gestion des affaires	
Généralités .....	0310
Administration .....	0454
Banques .....	0770
Comptabilité .....	0272
Marketing .....	0338
Histoire	
Histoire générale .....	0578

Ancienne .....	0579
Médiévale .....	0581
Moderne .....	0582
Histoire des noirs .....	0328
Africaine .....	0331
Canadienne .....	0334
États-Unis .....	0337
Européenne .....	0335
Moyen-orientale .....	0333
Latino-américaine .....	0336
Asie, Australie et Océanie .....	0332
Histoire des sciences .....	0585
Loisirs .....	0814
Planification urbaine et régionale .....	0999
Science politique	
Généralités .....	0615
Administration publique .....	0617
Droit et relations internationales .....	0616
Sociologie	
Généralités .....	0626
Aide et bien-être social .....	0630
Criminologie et établissements pénitentiaires .....	0627
Démographie .....	0938
Études de l'individu et de la famille .....	0628
Études des relations interethniques et des relations raciales .....	0631
Structure et développement social .....	0700
Théorie et méthodes .....	0344
Travail et relations industrielles .....	0629
Transports .....	0709
Travail social .....	0452

### SCIENCES ET INGÉNIERIE

#### SCIENCES BIOLOGIQUES

Agriculture	
Généralités .....	0473
Agronomie .....	0285
Alimentation et technologie alimentaire .....	0359
Culture .....	0479
Élevage et alimentation .....	0475
Exploitation des péturages .....	0777
Pathologie animale .....	0476
Pathologie végétale .....	0480
Physiologie végétale .....	0817
Sylviculture et faune .....	0478
Technologie du bois .....	0746
Biologie	
Généralités .....	0306
Anatomie .....	0287
Biologie (Statistiques) .....	0308
Biologie moléculaire .....	0307
Botanique .....	0309
Cellule .....	0379
Écologie .....	0329
Entomologie .....	0353
Génétiq .....	0369
Limnologie .....	0793
Microbiologie .....	0410
Neurologie .....	0317
Océanographie .....	0416
Physiologie .....	0433
Radiation .....	0821
Science vétérinaire .....	0778
Zoologie .....	0472
Biophysique	
Généralités .....	0786
Medicale .....	0760

Géologie .....	0372
Géophysique .....	0373
Hydrologie .....	0388
Minéralogie .....	0411
Océanographie physique .....	0415
Paléobotanique .....	0345
Paléocologie .....	0426
Paléontologie .....	0418
Paléozoologie .....	0985
Palynologie .....	0427

#### SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT

Économie domestique .....	0386
Sciences de l'environnement .....	0768
Sciences de la santé	
Généralités .....	0566
Administration des hôpitaux .....	0769
Alimentation et nutrition .....	0570
Audiologie .....	0300
Chimiothérapie .....	0992
Dentisterie .....	0567
Développement humain .....	0758
Enseignement .....	0350
Immunologie .....	0982
Loisirs .....	0575
Médecine du travail et thérapie .....	0354
Médecine et chirurgie .....	0564
Obstétrique et gynécologie .....	0380
Ophtalmologie .....	0381
Orthophonie .....	0460
Pathologie .....	0571
Pharmacie .....	0572
Pharmacologie .....	0419
Physiothérapie .....	0382
Radiologie .....	0574
Santé mentale .....	0347
Santé publique .....	0573
Soins infirmiers .....	0569
Toxicologie .....	0383

#### SCIENCES DE LA TERRE

Biogéochimie .....	0425
Géochimie .....	0996
Géodésie .....	0370
Géographie physique .....	0368

#### SCIENCES PHYSIQUES

##### Sciences Pures

Chimie	
Généralités .....	0485
Biochimie .....	487
Chimie agricole .....	0749
Chimie analytique .....	0486
Chimie minérale .....	0488
Chimie nucléaire .....	0738
Chimie organique .....	0490
Chimie pharmaceutique .....	0491
Physique .....	0494
Polymères .....	0495
Radiation .....	0754
Mathématiques .....	0405
Physique	
Généralités .....	0605
Acoustique .....	0986
Astronomie et astrophysique .....	0606
Électronique et électricité .....	0607
Fluides et plasma .....	0759
Météorologie .....	0608
Optique .....	0752
Particules (Physique nucléaire) .....	0798
Physique atomique .....	0748
Physique de l'état solide .....	0611
Physique moléculaire .....	0609
Physique nucléaire .....	0610
Radiation .....	0756
Statistiques .....	0463

##### Sciences Appliquées Et Technologie

Informatique .....	0984
Ingénierie	
Généralités .....	0537
Agriculture .....	0539
Automobile .....	0540

Biomédicale .....	0541
Chaleur et thermodynamique .....	0348
Conditionnement (Emballage) .....	0549
Génie aérospatial .....	0538
Génie chimique .....	0542
Génie civil .....	0543
Génie électronique et électrique .....	0544
Génie industriel .....	0546
Génie mécanique .....	0548
Génie nucléaire .....	0552
Ingénierie des systèmes .....	0790
Mécanique navale .....	0547
Métallurgie .....	0743
Science des matériaux .....	0794
Technique du pétrole .....	0765
Technique minière .....	0551
Techniques sanitaires et municipales .....	0554
Technologie hydraulique .....	0545
Mécanique appliquée .....	0346
Géotechnologie .....	0428
Matériaux plastiques (Technologie) .....	0795
Recherche opérationnelle .....	0796
Textiles et tissus (Technologie) .....	0794

#### PSYCHOLOGIE

Généralités .....	0621
Personnalité .....	0625
Psychobiologie .....	0349
Psychologie clinique .....	0622
Psychologie du comportement .....	0384
Psychologie du développement .....	0620
Psychologie expérimentale .....	0623
Psychologie industrielle .....	0624
Psychologie physiologique .....	0989
Psychologie sociale .....	0451
Psychométrie .....	0632

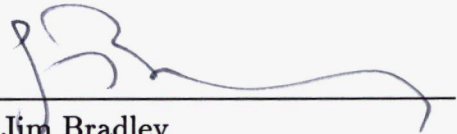


**THE UNIVERSITY OF CALGARY**  
**FACULTY OF GRADUATE STUDIES**

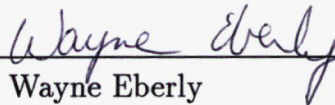
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "*Supporting Domains in Relational Database Systems*" submitted by Zhao Zhang in partial fulfillment of the requirements for the degree of Master of Science.



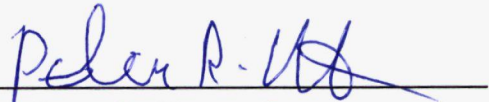
Supervisor, Dr. Anton W. Colijn  
Department of Computer Science



Dr. Jim Bradley  
Department of Computer Science



Dr. Wayne Eberly  
Department of Computer Science



Dr. Peter R. Newsted  
Faculty of Management

Date June 26, 1992

# Abstract

The concept of domains is one of the most fundamental concepts of the relational data model; without support for domains database systems would not be fully relational. Existing relational systems do not fully support domains, and hence inevitably suffer from many domain-related problems. Although some of the problems (but only a few) have been studied before, the real causes of the problems have never been properly pinpointed, and hence the problems have not been solved thoroughly.

In this thesis, we first examine the major domain-related problems in existing relational database systems: data comparability; domain-oriented data manipulation; domain update anomalies; domain integrity and “referential integrity”, etc. We strongly recommend supporting domains in relational database systems to solve these problems. The main features of domain support are described and SQL/D, an extension to the SQL language with domain features, is proposed.

In addition to discovering domain-related problems and proposing solutions, this thesis also clarifies several common misconceptions about domains and the relational data model. For example, the so-called “referential integrity” problem is identified as a special case of the domain integrity problem on “derived domains”.

Our current implementation of the Interactive SQL/D Interface on the SYBASE database manager is briefly described. Although only part of the proposed SQL/D is implemented, the running results tend to show that with domain support, database systems will become more reliable, flexible and natural. Some long-term perplexing domain-related problems of traditional systems could eventually be solved and the full potential of the relational data model could be fulfilled.

## Acknowledgements

I would like to express great thanks to my supervisor, Dr. Anton W. Colijn, for his guidance over the course of this research. His interest in and excitement over the topic of this dissertation was often inspirational. His editorial suggestions made a substantial improvement to the literary quality of this dissertation.

I would also like to extend special thanks to Dr. James Bradley for providing valuable advice and suggestions, especially to my thesis proposal.

I would also like to thank Dr. Wayne Eberly, whose careful comments make this thesis more accurate.

I would like to thank Andrew Ginter, Mengchi Liu and Chengfu Yao for their helpful discussions and comprehensive comments.

I am deeply indebted to my family, especially to my mother who passed away during this research period, for their love and continuous encouragement.

Finally, I would like to thank the office staff at the Department of Computer Science for their cooperation. Without their help, this research would not have been possible.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>List of Figures</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Domains in Relational Databases . . . . .	1
1.2 Related Research . . . . .	5
1.3 Domains in Current RDBMS . . . . .	9
1.4 Thesis Outline . . . . .	12
<b>2 The Problems Involving Domains</b>	<b>15</b>
2.1 What Is a Domain? . . . . .	15
2.2 Comparability . . . . .	21
2.3 Domain-Oriented Data Manipulation . . . . .	24
2.4 Domain Update Anomalies . . . . .	27
2.5 Referential Integrity as Domain Integrity . . . . .	31
2.5.1 The Current Conception of Referential Integrity . . . . .	32
2.5.2 Derived Domains . . . . .	35
2.5.3 Integrity of Derived Domains . . . . .	39
<b>3 Supporting Domains in Relational Databases</b>	<b>43</b>
3.1 General Domain Support . . . . .	43



3.2	Data Definition Facilities . . . . .	46
3.2.1	Types of Domains . . . . .	48
3.2.2	Defining a Database . . . . .	55
3.2.3	Other Features . . . . .	56
3.3	Data Manipulation Facilities . . . . .	57
3.3.1	Data Dictionary Inquiries . . . . .	57
3.3.2	Domain-Oriented Operations . . . . .	58
3.3.3	Impact of Relation-oriented Operations . . . . .	62
3.4	Adopting Internal Representation Techniques . . . . .	65
<b>4</b>	<b>SQL/D — Domain Extension to SQL</b>	<b>67</b>
4.1	A Brief Review of the SQL Language . . . . .	68
4.2	The Syntax of SQL/D . . . . .	69
4.2.1	Syntax Conventions . . . . .	70
4.2.2	Syntax of the Proposed SQL/D Extension . . . . .	71
4.3	Data Definition in SQL/D . . . . .	74
4.3.1	Defining Domains in SQL/D . . . . .	75
4.3.2	Defining Relations in SQL/D . . . . .	84
4.3.3	A Comprehensive Example . . . . .	86
4.4	Data Manipulation in SQL/D . . . . .	88
4.4.1	SELECT . . . . .	88
4.4.2	INSERT . . . . .	89
4.4.3	DELETE . . . . .	90
4.4.4	UPDATE . . . . .	91

4.4.5	GRANT and REVOKE . . . . .	92
<b>5</b>	<b>The Implementation of SQL/D</b>	<b>94</b>
5.1	Overview of SYBASE . . . . .	94
5.2	Overview of ISQLD . . . . .	96
5.3	Data Definition in ISQLD . . . . .	97
5.3.1	Types of Domains Implemented in ISQLD . . . . .	98
5.3.2	Define Domains and Relations . . . . .	100
5.3.3	Drop Domains and Relations . . . . .	101
5.4	The ISQLD Data Dictionary . . . . .	101
5.5	Data Manipulation in ISQLD . . . . .	104
5.5.1	Restricted Data Comparison . . . . .	104
5.5.2	Domain-oriented Data Manipulation . . . . .	105
5.5.3	Other Features . . . . .	107
5.6	Remarks . . . . .	108
<b>6</b>	<b>Conclusions</b>	<b>112</b>
6.1	Summary and Conclusion . . . . .	112
6.2	Future Research . . . . .	115
	<b>Appendix</b>	<b>118</b>
	<b>Bibliography</b>	<b>137</b>

## List of Figures

1.1	Overall Data Structure Comparison . . . . .	4
1.2	The Example Data Base . . . . .	14

# Chapter 1

---

## Introduction

---

### 1.1 Domains in Relational Databases

Since E. F. Codd proposed the relational model for database systems in his milestone paper [Codd70], various relational database management systems (RDBMS) have been developed in the past two decades. Although the number of aspects of the relational data model implemented in these systems varies, according to Codd [Codd82] and Date [Date91], none of the existing systems support all aspects of the relational model, or in other words, none of the existing systems are fully relational, so they all fail to realize the full potential of the data model. Among the few common aspects which are not supported in the existing RDBMS, the notion of **domain** is a very important and essential aspect which could have great impact on the overall structure of a relational database system.

The notion of domain is the most primitive concept of the relational model. It is the domains along with relations (and nothing else) that constitute the structural part of the data model. Informally, a domain is simply a pool of legal data values of the same type<sup>1</sup> [Date91, Ullm88, Mair83]. While a domain denotes a set of data values from which one or more attribute(s) may assume values, an attribute is merely the name for a column in the tabular representation of a relation. A relation in turn

---

<sup>1</sup>We argue that data values in a domain should be of the same semantic type, not necessarily be of same syntactic type.

is just a subset of the Cartesian product of a collection of domains. Hence we always say that a relational database is based on a certain collection of domains or we say that domains are the basis of relational databases.

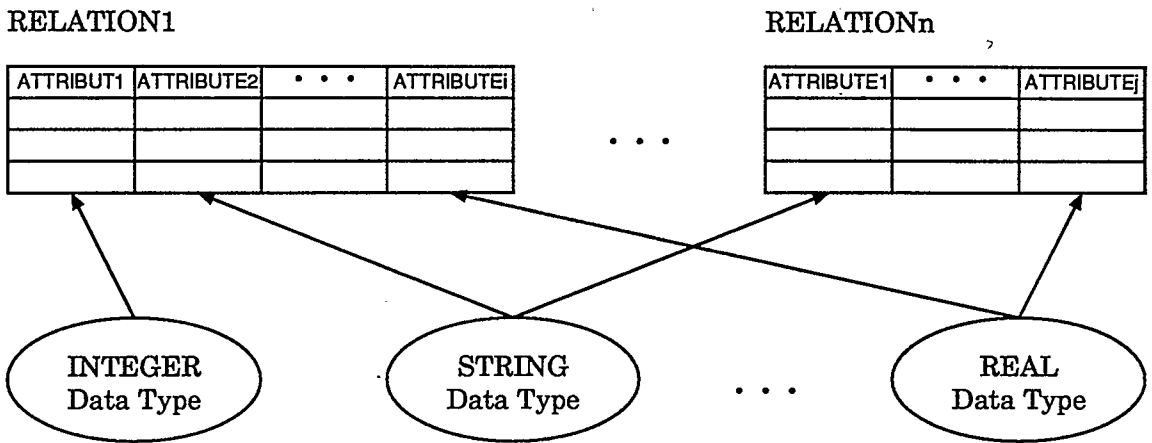
During the evolution of the relational data model, some obvious misunderstandings about domains made the relational database developers mistakenly neglect to implement domains in their products. For example, the consensus that domains are primarily conceptual in nature is the main reason for most of the current systems not physically storing any domains. By observing the misunderstandings about domains, we strongly recommend supporting domains in relational database systems, preferably in the following manner. First, domains, especially some special ones, should be explicitly stored in databases independently of ordinary relations. For most types of domains, the system can generate a unary relation for each domain and store all the distinct data values of the domain into the corresponding domain relation. Second, comprehensive domain definition and manipulation facilities should be added into the database language, enabling users to precisely define domains and to properly handle domains. Third, with domains supported, relations and attributes of a database must be defined on the underlying domains of the database. That means when designing a relational database, prior to declaring relations and their attributes, database administrators and/or database designers must define domains first. Finally, some new facilities should be added into domain-supported systems. For example, data comparison should be strictly confined to attributes that are based on the same domain.

By explicitly supporting domains in relational database systems, the overall data

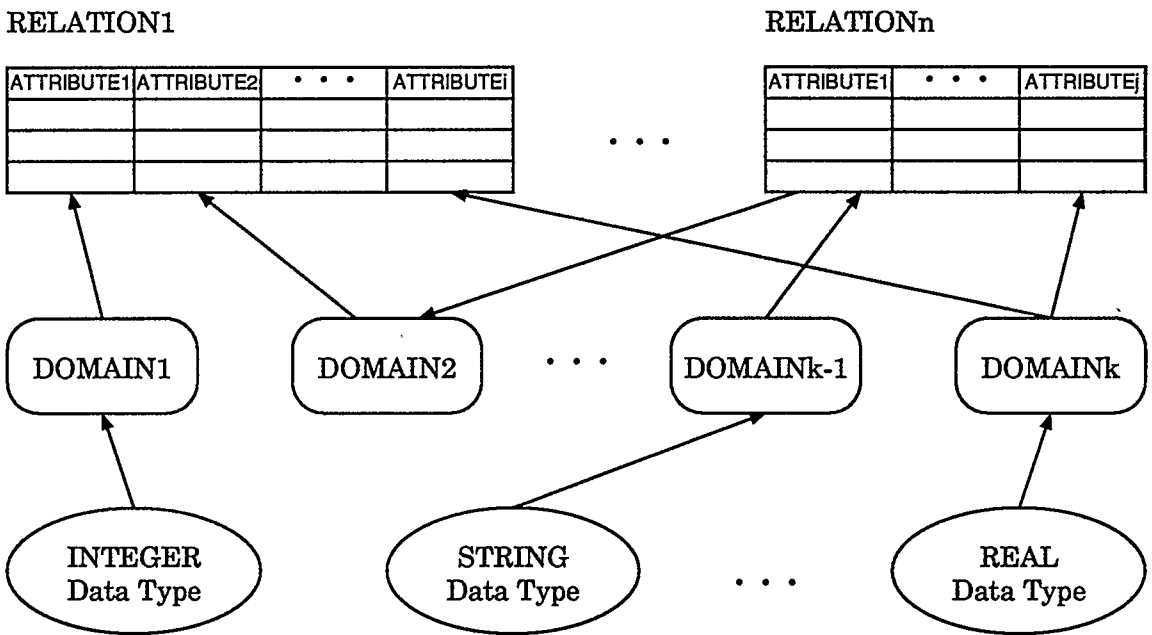
structure of relational databases will be changed. Figure 1.1 illustrates the different overall data structures of relational databases with and without domain support.

In the figure, it seems that domain-supported relational database systems have become more complicated. But this is not really true. In fact, with domains as real objects in databases we can (and sometimes we have to) manipulate data via domains instead of via relations. This will simplify data processing rather than complicate it. Also with (and only with) domains supported, relational databases will be built on their original theoretical basis. Then some long-term perplexing problems of traditional database systems could be solved in natural ways.

This research is mainly motivated by the authors' database system development experiences. Although relational database systems have improved very much during the twenty years of development, we can always find some flaws from existing systems when using them and some of the problems seem to be too difficult, or too cumbersome to be solved under the current structure of the relational systems. By analysing the problems, we found that many of the problems involve the fundamental concept of relational databases: domains. This research was not intended to introduce a new model of database systems; instead it was an attempt to extend existing relational database systems — focusing in particular on those using the SQL language — to support domains. We will see later that the goal can be achieved without too much effort and that, with the proposed domain support, many of the flaws of relational database systems will be eliminated or reduced.



(a) Database without domain supported



(b) Database with domain supported

Figure 1.1: Overall Data Structure Comparison

## 1.2 Related Research

Although in the past twenty years, many papers have been published on the subject of database systems, only a few of them ever discuss the domain concept and the role of domains in relational databases. The reason for the neglect of domains by database researchers is rather difficult to understand. Perhaps people just thought that domains were too simple to be studied. Date [Date86] certainly realised that domains were more complex than it might appear at first sight, and proposed extensive support for domains in relational database systems. However, his proposals require an undersirable amount of user-supplied code. Furthermore, they failed to solve at least some of the domain related problems discussed in this thesis.

As early as 1976, McLeod [McLe76] published his research result on a high level domain definition language. His paper might be one of the most valuable contribution to the topic of domains so far. In the paper, he outlines the main features of a high level domain definition language and discusses the data integrity and data comparability problems in domain-supported systems. The paper suggests that four components should be included in a domain definition; they are:

1. Domain name.
2. Domain description, which specifies the set of data values constituting the domain.
3. The ordering of data values in the domain.
4. The violation-action which is to occur if data integrity of the domain is violated.



In addition to the above, some implementation considerations are briefly discussed in the paper. But this pioneering paper did not attract much attention from other researchers because the importance of supporting domains in relational databases is not recognized in the paper.

Although all the relational database books have some coverage on the subject of domains, only the discussion in Date's famous textbook [Date91, Date86] is detailed enough. Date almost reaches the conclusion that relational database systems should support the domain concept. He suggests that at least domains should be specified as part of the database definition, and attributes of the database should be defined on the underlying domains. A database definition example on a pseudo SQL data definition language with domain support is given to show how domains and relations should be defined. The book also describes some other domain related topics like simple and composite domains, the operational significance of domains, etc. Date argues strongly for (what he feels to be) full support for domains in relational databases. However, he also seems to feel that any such support is necessarily very complex, and suggests this as a reason why current systems fail to provide this support.

Osbron and Heaven [OsHe86] extend relational databases to accept user-defined abstract data types for domains and user-coded operations on the data types<sup>2</sup>. In the paper they describe their experimental prototype which implements part of their idea. Now their idea has been adopted in some newly-developed commercial systems

---

<sup>2</sup>Note: the proposed system can only accept abstract data types but can not provide such data types; and the main purpose of DBMS is providing the users with more functions and reducing the burden of coding on the users. In their system, the users must code their own procedures to process the domains.

because of the high demand of such facilities from CAD and engineering database applications. Unlike the above papers, this paper does not discuss the basic problems related to so-called plain domains, which are dealt with by current RDBMS, so it is not directly relevant to our research.

It is very interesting that Kocharekar [Koch89], while revising the “maybe” operations on null values in relational databases, introduces the concept of dynamically defined domains. The problem for which the new concept is proposed is that a query which involves “maybe” matching of a foreign key value, no matter if the value exists in the referenced relation or not, will always get the same result provided there are some tuples with “null” or unknown values on the foreign key attribute of the referencing relation. Although the paper suggests that if there were “dynamically defined domains” in a relational database, the problem discovered would be solved by answering “unknown” to an existing value and “False” to a non-existing value respectively, the paper does not discuss how this special kind of domain could be dynamically defined, nor if there should be any other domains in the databases.

After we finished the implementation of our proposed SQL/D and during the time of writing this thesis, we found Date’s newest paper [Date90A] on the topic of domains. According to the author’s own description of the paper in the paper itself, the paper is a “systematic and comprehensive tutorial on the relational domain concept”. The paper argues strongly that a domain is basically nothing more or less than a data type, either built-in or (more generally, though few systems provide any such support today) user-defined. It identifies the following “aspects of domain support” (i.e. features that DBMSs need to provide in order to be able to claim full

support for domains):

- An operator for defining new domains (specifying at least a name and a representation for the domain)
- The ability to specify the relevant domain for each attribute
- Operators to drop and alter domains
- Domain-level integrity checking
- Support for appropriate literals
- Appropriate data type conversion or coercion rules, including in particular certain domains
- The ability to specify the operators that apply to each domain or combination of domains
- Appropriate catalog support for all of the foregoing

Many of the ideas in this paper are also summarized in the domain section of the latest edition of the author's database textbook [Date91]. Although the paper is the newest publication specifically on the concept of domains, and it intends to serve as a convenient single-source reference in which to find a comprehensive answer to the question "What is a domain?", it does not appear to be as comprehensive as it claims, because it does not cover most of the domain-related problems discussed in this thesis, for example, the domain-oriented data manipulation.

The relative scarcity of related studies on the topic of supporting domains made this research more interesting. The original question, whether relational database systems should support domains can now be answered in the affirmative after this research.

### 1.3 Domains in Current RDBMS

We mentioned before that none of the existing relational database management systems, no matter whether they are commercial systems or research prototype systems, fully support the concept of domains. Not everyone may agree with this claim until a consensus is reached on the meaning of “full support of domains”. In order to avoid disagreement, at this point we will not examine the existing database systems in terms of “full support of domains”. Instead we will evaluate them in terms of “minimum support of domains”.

In our point of view, a database system with minimum support of domains should have at least the following domain features.

- The system should explicitly distinguish domains of the database from attributes of relations and from the built-in data types. If needed, some domains should be physically stored independently of relations.
- Databases along with relations and attributes must be defined on underlying domains.
- Data comparisons between attributes based on different domains should be restricted or controlled to prevent certain nonsensical query conditions, like

“S.STATUS > P.WEIGHT”, from happening.

The existing mainframe relational database management systems could be classified into two broad groups, i.e. SQL-based systems which range from System R, SQL/DS, DB2 to ORACLE and SYBASE, and non-SQL systems which include INGRES<sup>3</sup>, SABRINA, SUPRA, UNIFY, ADABAS, etc. The systems in the former group share the same standard relational data language, the SQL language, as the main vehicle for expressing data requests. The systems in the latter group invent their own main data languages even if some of the systems provide interfaces to SQL. Because in the proposed ANSI/ISO SQL standard [Date89, YaCh88], which is the sole relational language standard today, there is no notation about domains, there is no doubt that all the current SQL-based systems, whose data languages are merely a subset of the SQL standard, do not support the domain concept at all. On the other hand, the majority of non-SQL systems, like those listed above, neither distinguish domains from attributes nor restrict inter-domain data comparisons [VaGa89]; therefore they all fail to support domains, too. Kruglinski [Krug86] evaluates more than ten MS-DOS based PC database systems, including the most popular systems like dBASE, KnowledgeMan, Informix, etc. None of these systems seems to have any support for domains.

According to some authors [Date91], there are quite a few systems that do have some sort of support for domains. For instance, Query-By-Example (QBE for short) is claimed by Date [Date91] as an explicitly domain-supporting relational database system. In QBE, domains are distinguished from columns of tables, or attributes of

---

<sup>3</sup>INGRES, though supporting SQL now, is not a SQL-based system

relations. Each column of a table is defined on some underlying domain and each domain in turn is assigned a certain data type. Column names may be the same as or different from the corresponding domain name. Several columns can share the same domain and most importantly, data comparisons are restricted to columns based on the same underlying domains, especially when tables are linked together on common columns. Although possessing more domain features than most other relational database systems, QBE still fails to be classified as a minimum supporter of domains, since no domain is ever physically stored in QBE.

Another system with similar domain features is R:BASE from MicroRim, Inc. R:BASE is the PC-based descendant of a mainframe DBMS product called Microrim, which was developed for NASA in support of the space shuttle program. In the data definition part, R:BASE comes closer than any other PC-based systems to supporting domains. To create a database in R:BASE, one must define all the “attributes” for the entire database first, though the definition of an attribute consists of a data type declaration only. Then the relations are formed by assigning some of the attributes to each of the relations. Since the same attribute can appear in several relations, we can say in a sense that “attribute” in R:BASE means both attribute for tables and domain for the entire database. So domains are somehow separated from attributes. R:BASE employs relational algebra in data manipulation but has no restrictions on inter-domain data comparisons. In fact, it is peculiar that R:BASE really needs inter-domain data comparison when joining several relations together, since the “JOIN” command of R:BASE only applies to attributes whose names are different from each other.

Now it is clear that even in terms of minimum support of domains, none of the existing relational database management systems have explicit support for the domain concept. The main reasons for that are twofold. First the lack of systematic studies on the roles of domains in relational database systems gave the database vendors such an incorrect impression that domain support is not necessary in their products. Second, as Date [Date86] pointed out, supporting domains is considerably more complex than it might appear at first sight.

## 1.4 Thesis Outline

In the next chapter, we will discuss the domain related problems in current relational databases. The main topics are: inter-domain data comparisons, data integrity, domain update anomalies, and data manipulation via domains. We will also discuss some misunderstandings about domains and the relational data model. It is those misunderstandings which have led to the development of relational database systems without supporting domains. During the discussion, we will also introduce some new ideas about domains and relational databases.

In chapter 3, in addition to outlining the main features of a domain-supported system, we will explain how a domain-supported system can solve the problems listed in Chapter 2. The benefits of supporting domains in relational databases will be described in this chapter, too.

In chapter 4, SQL/D, an extension to the SQL language with domain features will be proposed. First we describe the syntax of the SQL/D language. Then data definition and data manipulation facilities of SQL/D will be explained in detail.

In chapter 5, our current implementation of SQL/D on the SYBASE database management system will be presented. Although the prototype is only an interactive SQL/D interface to the SYBASE RDBMS and it only implements a subset of the proposed language, it does demonstrate that the domain concept could be supported in relational databases without too much effort and that relational database systems could be significantly improved in many respects.

In some examples of the thesis, we will use the SQL language to illustrate data operations on the example database. For the details of the language please refer to reference [Date89].

Throughout this paper, a “Suppliers-and-Parts” example database similar to that in [Date91] is used for explanation purposes. The only change to the the database is that the attribute COLOR is replaced with a PRICE attribute in relation P. The final structure of the database is:

S(SNUMB, SNAME, STATUS, CITY)

P(PNUMB, PNAME, PRICE, WEIGHT, CITY)

SP(SNUMB, PNUMB, QTY)

The sample data are listed in Figure 1.2.



S	S#	SNAME	STATUS	CITY
	S1	Smith	20	London
	S2	Jones	10	Paris
	S3	Blake	10	Paris
	S4	Clark	20	London
	S5	Adams	30	Athens

P	P#	PNAME	PRICE	WEIGHT	CITY
	P1	Nut	9.99	12	London
	P2	Bolt	14.99	17	Paris
	P3	Screw	14.99	17	Rome
	P4	Screw	9.99	14	London
	P5	Cam	4.99	12	Paris
	P6	Cog	19.99	19	London

SP	S#	P#	QTY
	S1	P1	300
	S1	P2	200
	S1	P3	400
	S1	P4	200
	S1	P5	100
	S1	P6	100
	S2	P1	300
	S2	P2	400
	S3	P2	200
	S4	P2	200
	S4	P4	300
	S4	P5	400

Figure 1.2: The Example Data Base

## Chapter 2

---

# The Problems Involving Domains

---

In current relational database systems, there are many problems directly or indirectly involving domains. Also in database publications, there is a certain amount of confusion about domains. In this chapter, we will mainly examine the problems, disclose and clarify the confusions and then in the next chapter we will present our solutions to the problems: supporting domains in relational databases.

### 2.1 What Is a Domain?

Before discussing any other confusion about domains, it is necessary to clarify the fundamental concept of this thesis: What is indeed a domain.

Below are six typical definitions of domains from authoritative sources.

1. E. F. Codd [Codd70]: *“Each of these domains is, in effect, a pool of values, some or all of which may be represented in the data bank.”*
2. C. J. Date [Date91]: *“We define a domain to be a named set of scalar values, all of the same type ... Thus domains are pools of values, from which the actual values appearing in attributes are drawn.”*
3. J. D. Ullman [Ullm88]: *“Formally, a domain is simply a set of values, not unlike a data type. For example, the set of integers is a domain. So are the set of character strings, the set of character strings of length 20, and the set  $\{0,1\}$ , for additional examples.”*

4. D. J. McLeod [McLe76]: *“A domain is a set of atomic data values (objects). In particular, a domain is a subset of one of the two ‘natural’ domains: real number and character string.”*
5. D. Maier [Mair83]: *“Corresponding to each attribute name is a set of permissible values for the associated column. This set is called the domain of the attribute name.”*
6. C. Yang [Yang86]: *“The domain or value-set of an attribute  $A_j$  is a finite set of the values of  $A_j$ , which must be of the same data type.”*
7. J. Bradley [Brad82]: *“A domain is the set of values from which the set of attribute values of a relation may be taken; that is, from which a column of a table may be formed.”*

Now let us analyze the above definitions of domains. First, we can easily draw a consensus from the above six definitions, that is: A domain is a set of values. We agree with them on this fundamental aspect of definition of domains. As simple examples, the domain of supplier numbers is the set of all valid supplier numbers and the domain of shipment quantities might be the set of integers between 0 and 1,000 (say).

Second, two of the definitions (2, 4) explicitly indicate that the data values of a domain must be atomic or scalar—that is, data values of a domain are non-decomposable. In other words, domains are not composite at all and values of a domain are of the smallest unit as far as the relational model is concerned. We endorse this with some reservation since it seems that in some situations compound domains are unavoidable. For an extreme example, a domain of birth date, of which values are the combination of year, month and day values, may need to be (and of course could be) decomposed into its three components in some situations. However, because domains are invariably assumed to be simple in most database literature,

we will follow this convention throughout this thesis unless explicitly stated to be otherwise.

Third, two of the definitions (2, 6) confine values of a domain to be of one data type. That means no domains can take values of different data types. If the domain of shipment quantities is of the type integer number, then it would not accept any character string or any real number as a valid value. We could not agree that this assertion is reasonable. As we stated in the beginning of this thesis, we feel that values of a domain must be of same SEMANTIC type, but need not to be of same SYNTACTIC data type. For example, the domain of month should probably allow two different data types or two different formats: integer and character string so that the integer 12 and the character string 'December' could both represent the last month of a year.

More interesting, definition 3 (and perhaps 4) implies that a domain is simply a data type or a subset of a data type. We could not say that this is totally incorrect as far as computerized databases are concerned but in the real world a domain is certainly not just a data type and some domains can even accept values of different types.

Finally, three of the definitions (2, 5, 6) describe that one or more attributes can draw values from a domain. That means domains provide attributes with all the legal values to appear in the attribute(s). This reflects the main role that domains play in a database system. For example, the domain of supplier numbers provides all the permissible values for attribute *S#* of relation *S*, while the domain of city contains all the possible city values to appear in the attribute *CITY* of relation *S* and the

attribute CITY of relation P. Here attribute CITY of relation S and attribute CITY of relation P share the same domain CITY.

Summarizing the above domain definitions, a domain is described as: a named set (or pool) of atomic (or scalar) values, all being of the same data type. The actual values appearing in attributes (or in a database) are drawn from underlying domains. A domain is not unlike a data type or it is a subset of the two “natural” domains: real numbers and character strings.

In contrast to the above definition, our definition of domain is as follows.

Definition 2.1: A *Domain D*, like a relation, is a named independent object in a database. It consists of a finite or potentially infinite set of values<sup>1</sup>. In the case of a finite set of values, the set may be denoted by  $\{ d1, d2, \dots, dn \}$ , where  $n$  is the cardinality of the domain. Each value of  $D$ ,  $di$  ( $i = 1, 2, \dots, n$ ) is generally an atomic value and the values of a domain, while not necessarily being of the same syntactic data type, must be of the same semantic type. One or more attributes may draw actual values from a domain and all the values in the domain are legal for the attribute(s). It then follows that data comparisons among several attributes make sense if and only if these attributes are drawing values from the same domain,

Our definition disagrees with the above definitions in at least two major respects. First, the other definitions do not specify that a domain is an independent object. In fact most of authors describe domains as being primarily conceptual in nature. This leads to the misconception that it is not necessary to physically store domains as objects in databases and it is not necessary to implement (or support) domains in database systems. Second, it is not mentioned as a major role of domains in the

---

<sup>1</sup>As long as a digital computer is involved even floating point numbers are really a finite set.

other domain definitions that only intra-domain data comparisons, which compare like with like, make sense. This rule of data comparability should be explicitly enforced in any system which supports domains.

We will discuss some more disagreements between our definition of domains in more detail later in this chapter.

To conclude this section, we adopt Date's definition of relation [Date91] as our formal definition of relation.

Definition 2.2: A *relation*  $R$  on a collection of domains  $D = \{D1, D2, \dots, Dn\}$  ( $D1, D2, \dots, Dn$  are not necessarily distinct) consists of two parts, a *heading* and a *body*.

- The heading consists of a fixed set of *attributes*, or more precisely attribute-domain pairs,  $\{ (A1:D1), (A2:D2), \dots, (An:Dn) \}$  such that each attribute  $Aj$  corresponds to exactly one of the underlying domains  $Dj$  ( $j = 1, 2, \dots, n$ ). The  $Aj$ 's must all be distinct.
- The body consists of a time-varying set of *tuples*, where each tuple consists of a set of attribute-value pairs  $\{ (A1:vi1), (A2:vi2), \dots, (An:vin) \}$  ( $i = 1, 2, \dots, m$ , where  $m$  is the number of tuples in the set). In each such tuple, there is one such attribute-value pair  $(Aj:vij)$  for each attribute  $Aj$  in the heading. For any given attribute-value pair  $(Aj:vij)$ ,  $vij$  is a value from the domain  $Dj$  that is associated with attribute  $Aj$ .
- The values  $m$  and  $n$  are called the *cardinality* and the *degree*, respectively, of relation  $R$ . The cardinality changes with time, whereas the degree does not except that the relation is otherwise reorganized. A relation of degree one is

called *unary*, a relation of degree two *binary*, a relation of degree three *ternary*, ..., and a relation of degree  $n$  *n-ary*.

Notation: In the following discussion, we will use the conventional format:

`relation_name.attribute_name`

to refer an attribute of a relation. For example `S.S#` refers to the attribute `S#` of relation `S`.

With the above formal definitions about domains and relations, we will use the formal relational terms almost exclusively in our following discussion; however some less formal terms may be used occasionally, especially when quoting other authors and in the keywords of the proposed domain extension to the SQL language. That is instead of using the “standard” SQL term “table”, we will use “relation” and so on. The following table summarizes the terms we will use and the corresponding terms of SQL:

Formal term	SQL term
domain	pool of legal values
cardinality of domain	number of legal values
relation	table
tuple	row or record
attribute	column or field
cardinality of relation	number of rows
degree of relation	number of columns

## 2.2 Comparability

The term comparability here refers to the general problem of determining whether two or more data values may be compared or otherwise manipulated. In existing relational database systems where domains are not supported, data comparability is a very confusing problem. In these systems, the only mechanism for determining the validity of a data comparison is data type checking. If a data comparison involves data of the same data type, say two integers, then it will be accepted and performed, otherwise it will be rejected. This simple mechanism causes the following problems.

First, on the one hand, database operations with syntactically valid data comparison may yield nonsensical results in existing systems. For example, the following SQL retrieval

```
SELECT *  
FROM P, SP  
WHERE P.WEIGHT < SP.QTY
```

may produce some resulting tuples, but nothing in the result is valuable. The data comparison in the retrieval involves two attributes which are presumably of same data type, say integer, so the comparison is syntactically valid and performable and hence so is the retrieval itself. Then in the current SQL systems, the operation will certainly be executed and result only in some (72 from the sample database) nonsensical tuples.

Semantically, the above retrieval would be interpreted as: retrieve all the detailed information for each pair of a part and a shipment where the weight of the part is less than the quantity of parts shipped in the shipment. The semantics of the retrieval



implies that no resulting tuple from such a retrieval would make much sense because of the nonsensical data comparison. This data comparison involves two different domains, domain of part weight for attribute P.WEIGHT and domain of shipment quantities for attribute SP.QTY. Although the data in the two domains are probably of the same data type, say integer, semantically they are different. So a comparison of data from the two domains is not comparing “like with like”; hence the result of the comparison is meaningless, and so is the retrieval result itself.

Generally, comparisons of data from different domains<sup>2</sup> are just mistakes. According to the definition of domains, a domain is the set of values of same type, or in other words all values of the same type are contained in the same domain. So for any meaningful data comparison which is comparing “like with like” or is comparing data of the same type, the data compared must come from the same domain. Otherwise if a data comparison involves data from different domains, then it is to compare data of different types or to compare “like with unlike”. Such comparisons would not make sense and any possible result from the comparisons is just garbage, hence the comparisons should be rejected.

The foregoing discussion shows that it is the underlying domains which determine the comparability of data in relational databases. Now the question is how to find out if two attributes are defined on the same domain. In a system which does not support domains, it is absolutely impossible to find out whether an inter-attribute comparison involves two different domains, so it is almost inevitable for the system to perform some silly inter-domain data comparisons. Conversely, in a system with

---

<sup>2</sup>Except from compatible domains, like a DERIVED domain and its parent domain; see next chapter for more details.

domain support, every attribute is defined on an underlying domain, so the system is aware of which attributes are comparable with one another and which are not. Then nonsensical inter-domain data comparisons are detectable and could be rejected by the system.

While data comparisons in a domain-supported system are principally limited to values from the same domain, a “forced comparison” option, which may take the format of any ordinary comparators prefixed with a ‘@’ sign, should be provided in case of some inter-domain comparison insisted on by a user. For example, if one insists on doing the above retrieval, the following “forced” version of SQL query could be used:

```
SELECT *  
FROM P, SP  
WHERE P.WEIGHT @< SP.QTY
```

In contrast to the above inter-domain data comparison problems, there are some intra-domain data comparison problems in existing relational database systems. That is, sometimes a reasonable and semantically correct data comparison may be rejected by the system for reasons of syntactic errors and sometimes a data query with comparisons of similar data may not get the desired result. Suppose values of a domain are represented in different formats, in different units (WEIGHT in pounds and in kilograms) or just in different data types. Then comparisons of the values of the domain may confuse the systems. For example, if values of domain MONTH are represented in multiple formats. That is the value for the last month of a year is represented as character strings “December” in some attribute, and as “Dec.” in some other attribute, and even as the integer 12 in another attribute of

a database. Then data comparison between those “different” values would either produce a “NOT EQUAL” (between “December” and “Dec”) result or be rejected by the system for a data type mismatching error (between “December” and 12). But in fact those are just different representations of the same value.

Another kind of intra-domain data comparison problem is calculation of the difference between data values. For the domain MONTH, sometimes we may need to find out the difference between value “October” and value “December”, that is the number of months from October to December. This kind of calculation, though reasonable in practice, seems impossible or illegal in most existing systems because it is impossible to calculate the difference between two character strings.

To make database systems capable of handling the above intra-domain data comparison requests we will need a special kind of domain, namely MULTITYPED domain, of which data values could be represented in multiple formats or multiple types. This kind of domain will be discussed in the next chapter.

## 2.3 Domain-Oriented Data Manipulation

In existing relational databases, all data operations are expressed and carried out in terms of relations<sup>3</sup>, no matter if the operations are really relation-oriented or not. The data manipulation facilities available in the systems are all relation-oriented data operations; that is, you can retrieve data from relations, you can insert new data into a relation, or you can delete unwanted data from a relation and so on. But

---

<sup>3</sup>In SQL, one can express data operation in terms of views, which are virtual relations derived from base relations. Even in this case, the actual operation is carried out on the base relations on which the views are defined.

you cannot manipulate data on domains, although domains should be independent objects in databases. No system today provides any domain-oriented data manipulation facilities, such as insertion of a new value into a domain or update of existing values in a domain, etc. Even in some so-called domain-oriented data languages, like Domain Relational Language [LaPi77] which is a relational calculus language, domains are not the direct objects of data operation. Instead, domains only provide the value ranges for calculus variables. So it seems that in relational databases there might be no need of direct data manipulations on domains.

But this is not true in real world situations. In fact, although most data requests in database systems are relation-oriented, there are some other data requests which are intrinsically domain-oriented. That is to say, in addition to manipulating data via relations, sometimes we also need to manipulate data via domains. For example, when a part is to be allocated in a city, we may wish to get a list of all the permissible CITY values in the Suppliers-and-Part database. Such a data request is apparently aimed at the domain CITY, not at the relations which contain attributes based on the domain CITY. In existing relational systems, because the domain CITY has no physical existence, it is totally impossible to retrieve all the legal values of domain CITY. What we can retrieve is the values existing in the attribute CITY of relation S and of relation P. But usually that is only part of the values of domain CITY. Generally speaking, without domain support, a retrieval of all values of a domain is impossible in relational database systems.

Besides domain-oriented retrieval requests, we may have domain-oriented update requests. For example, we may want to add a new value, say "Stockholm", into

the domain CITY without inserting the value into any relation. This request is apparently impossible in the systems without domain support. We refer to this sort of problems as domain update anomalies and discuss it later in a separate section. Another typical domain-oriented update example is an increase of all the values of the domain PRICE by ten percent. Such a request will affect not only the values in the domain itself (if the domain is physically stored) but also the values spread over all the attributes that are defined on the domain. Despite the inability to update the domain itself in existing systems, updating all the values in the related attributes is not an easy task. First we must know which attributes are based on the domain, then we need to issue a separate update request for each of the relations that contains the attributes. If any of the attributes is forgotten during the updating then the database will become inconsistent.

The last kind of domain-oriented data request is an inquiry on the relationships between domains and attributes of databases. For example, we might want to list the attributes which are defined on the domain CITY or to find the underlying domain of attribute CITY of relation S, etc. In a system that supports the domain concept, the requests could be translated into a simple interrogation against the system data dictionary. But in existing systems, it is obviously not possible to interrogate the system dictionary regarding domains because there is nothing about domains ever recorded.

Now we can say that in relational database systems, in addition to relation-oriented data requests, there are also domain-oriented data requests. Because of the lack of domain-oriented data operations and, more importantly, the lack of physical

existence of domains in existing systems, to realize a domain-oriented data request, the only thing we can do is to translate the request into (a series of) relation-oriented operations and then perform them by relation-oriented data operations provided by the system. This solution is obviously not a natural way to solve the problems. Even worse, in many cases such a translation is impossible. The only natural solution is explicitly to implement domains. That means the system must:

1. Physically store domains, especially some special domains.
2. Provide facilities to manipulate domains directly.

The detailed features of domain support will be discussed in the next chapter.

## 2.4 Domain Update Anomalies

When Codd first introduced the relational data model [Codd70], one of the most significant contributions he made was the introduction of the principle of database normalization. The intent of normalizing a database is not simply to make relations in some particular normal form; instead it is to eliminate data redundancy in the database, and hence to solve so-called “update anomalies” problems. This problem, which is solved by normalization of relations, concerns data from at least two attributes, or in other words, concerns relationships between at least two attributes. For instance, suppose all suppliers in the same city have a fixed status. Then the relationship between the two attributes, CITY and STATUS, is totally independent of other attributes. But with the current structure of relation S, a pair of CITY and STATUS values could not exist in the database unless there is at least one S# (the

primary key of relation S) value associated with the pair. So the anomaly is that the existence of the independent relationship between attributes CITY and STATUS is dependent on some other attribute, say S#, in the relation. The cause of the anomaly is improper relation structures and the anomalies could be avoided by “normalization” of relations. Since this kind of update anomaly is related to relations, we call it a *relation update anomaly*.

In relational database systems, there exists another kind of update anomaly, namely a *domain update anomaly*. In the previous section, we have already seen a simple example showing the problem. However the entire domain update anomaly problem is much more complex than that in the example. In existing relational database systems, since domains are not explicitly stored, the set of values for a domain never exists on its own in a database. The only existing values of a domain in the database are the values occurring in the attributes based on the domain. The set of values for a domain changes whenever relations are updated. So the physical existence of a domain totally depends on the existence of relevant relations and the survival of an individual domain value in turn depends on its occurrence in relations with attributes based on the domain. Then we will encounter the following domain update anomalies:

- *Insertion anomaly*: It is impossible to insert a value (directly) into a domain without inserting the value into a relation with an attribute based on the domain. Consider, for example, the attempt to directly insert value “Stockholm” into domain CITY.
- *Deletion anomaly*: If a domain value has only one occurrence in a database and

the sole occurrence is deleted from the relation containing it, then the value will disappear from the domain and hence from the database. For example, the valid value “ATHENS” of domain CITY will disappear if the following SQL expression is executed,

```
DELETE FROM S WHERE S#='S5'
```

though it only intended to delete the tuple containing the CITY value of 'Athens'.

- *Modification anomaly:* If a domain value has only one occurrence in a database and the sole occurrence is replaced by another value of the same domain, then the original value may disappear from the database. For example, the valid value “ATHENS” of domain CITY will disappear if the following SQL expression is executed,

```
UPDATE S SET CITY='LONDON' WHERE CITY='ATHENS'
```

Generally speaking domain update anomaly problems are side-effects of certain relation-oriented data operations. That is, when manipulating data via relations, not only are values in the relations affected, but the corresponding values in the “would-be” domains will also be affected. The reason for such anomalies is quite simple: domain-oriented updates are not distinguished from relation-oriented updates so that only relation-oriented updates are provided in existing systems.

Whether an update is domain-oriented or relation-oriented depends on the semantics of the update. Simply speaking, if an update is made with the intention of altering the value(s) of a domain then the update is domain-oriented and if it



aims only at occurrence(s) of value(s) of a domain in a relation then it is relation-oriented. For example, “delete value ‘S1’ from the domain of supplier numbers” is domain-oriented and “delete value ‘S1’ from relation S” is relation-oriented. The first deletion will not only delete value ‘S1’ from the corresponding domain but will also automatically delete all the occurrences of the value from the entire database. The second deletion will only delete the occurrences of the value in relation S (and probably the occurrences of ‘S1’ in relation SP for reasons of referential integrity). They are apparently different deletions and should be treated differently. But unfortunately in a system which does not provide domain-oriented data manipulation facilities, the two deletions may be expressed as the same. For example in SQL it would be:

DELETE FROM S WHERE S# = ‘S1’

It seems to the system that this is to delete all occurrences of value ‘S1’ in relation S, but in fact it might actually intend to delete the value from the entire database or from the corresponding domain<sup>4</sup>. We argue that relational database systems should provide domain-oriented update facilities in addition to the relation-oriented update facilities so that users can always use the correct update facilities without risking domain update anomalies.

These domain update anomalies problems have never been mentioned in the database literature before. This seems to be one of the major misconceptions about domains. The direct cause of the problem is that there is no physical existence of domains independently of relations in databases. Although it is widely acknowledged

---

<sup>4</sup>To delete values from the entire database we need to delete all its occurrences from all relations which contain occurrences of the value.

that domains have an abstract existence apart from the database relations, in existing relational database systems the only existence of domains is the occurrences of domain values in the relations of the database. The lack of physical existence of domains leads to the lack of domain manipulation facilities and then to the domain update anomalies.

Domain update anomalies differ from relation update anomalies in at least two respects. First, domain update anomalies usually involve only one domain. For example, in the above update only domain CITY is involved; but relation update anomalies always involve two or more attributes. For example, you cannot insert a value of attribute CITY into relation S unless it is associated with a value of attribute S#. Second, while relation update anomalies could be avoided by normalization of the database or properly designed database structures, domain update anomalies could not be avoided by any efforts from database designers. They will only be eliminated by explicitly implementing domains in relational database systems, i.e. through physical storage of domains plus domain-oriented data manipulation facilities.

## 2.5 Referential Integrity as Domain Integrity

Another data integrity problem involving domains is referential integrity. Referential integrity is one of the most important and difficult problems in relational database systems. In this section, we will analyze the problem and point out that referential integrity is just a special case of domain integrity, assuming the general notion of domain defined in this thesis.

### 2.5.1 The Current Conception of Referential Integrity

The intent of referential integrity constraints is simply that if a data value  $v_2$  refers to some other data value  $v_1$ , then  $v_1$  must exist in the database. For example, the attribute  $S\#$  of relation  $SP$ , as a referencing attribute, must represent an existing supplier. Then its values must have a counterpart in the referenced attribute  $S\#$  of relation  $S$ , since relation  $S$  is the list of all existing suppliers. In general, if a value in the referencing relation does not have a counterpart in the referenced relation, then the referential constraint between the two relations is violated and the invalid referencing value should be rejected by the system.

The referential integrity concepts are of paramount importance in the field of database technology. Although in the past twenty years a great amount of research has been done on the subject, Date[Date90B] still described the situation on the conception of the referential integrity in 1990 as:

Yet they (the referential integrity concepts) are surrounded by an extraordinary degree of confusion— confusion in the open literature, confusion in the database community at large and (especially) confusion in the database marketplace. There are certain numerous conflicting definitions in various books, papers, trade journals, and elsewhere. Clarification is urgently needed.

And hence he offers his clarified definition of the concepts in the same paper as:

- *Primary key*: Loosely, a primary key is just a unique identifier. A little more precisely: The primary key for a table  $T_1$  is a column  $PK$  of  $T_1$  such that, at any given time, no two rows of  $T_1$  have the same value for  $PK$ .

- *Foreign key:* Loosely, a foreign key in one table is a column whose values are values of the primary key of some other table (or possibly the same table). A little more precisely: A foreign key is a column FK of some table T2 such that at any given time, every nonnull value of the FK in T2 is required to be equal to the value of the primary key in some row of some table T1. Table T2 here is a referencing table, table T1 the referenced or target table. The two tables are not necessarily distinct.
- *Referential integrity:* Since a given foreign key value obviously represents a reference to the row containing the matching primary key value (the referenced row or target row), the problem of ensuring that the database does not contain any invalid foreign key values is known as the referential integrity problem. The constraint that values of a given foreign key must match values of the corresponding primary key is known as a referential constraint.

According to the above definition, referential integrity always happens between the primary key of a relation and a foreign key of a relation. An integrity constraint implies that if any value appears in a foreign key attribute then it must have a counterpart occurrence at the corresponding primary key attribute. In our example database, the primary key of relation S is attribute S# and the foreign key of relation SP that references to the primary key is attribute S#. The integrity constraint between the two relations is that every value of attribute S# in relation SP must match some value of attribute S# in relation S. By the definition, we will always need such a pair of primary key and foreign key to establish a referential integrity constraint.

Referential integrity constraints are quite difficult to satisfy. In some current relational database systems, in order to maintain referential integrity, a great deal of code that checks all the possible constraint violations must be included in whatever programs update the database. In some other systems, one can make the system enforce constraint checking by explicitly declaring primary keys and referencing foreign keys (perhaps plus some other utilities). For example in DB2 [DaWh88, HeHe89] and some other SQL-based systems [VaGa89], to enforce a referential integrity constraint, say that between the two relations  $S$  and  $SP$ , the user has to declare  $S\#$  of  $S$  the primary key in  $S$  and  $S\#$  of  $SP$  a foreign key in  $SP$  referencing to relation  $S$ . In addition to defining keys, a unique index on attribute  $S\#$  of relation  $S$  must be created and a similar index on attribute  $S\#$  of relation  $SP$  is strongly recommended. It is really difficult to see the point that, in order to enforce data integrity, creation of indices on relations is needed. Such a solution is certainly not concise nor natural. Even worse, in Date's six-step recipe [Date86] to enforce referential integrity on relational databases, not only are coding and extra data structures needed but also some unacceptable restrictions are to be imposed. For example, in some systems it is necessary simply to prohibit all on-line operations that may violate the referential constraints because there is no way to force all the on-line users to check all the possible constraint violations.

The above definition of referential integrity and related concepts is the latest that could be found. It reflects the most recent understanding of the concept by the database community. But even this is not all correct. We will discuss our conceptions about referential integrity in the next section.

### 2.5.2 Derived Domains

During the period of this thesis research, we found some problems in the current interpretation of the concept of referential integrity. First of all, the common understanding that the referencing and the referenced attributes involved in a referential integrity relationship share a common domain (or the attributes draw values from the same domain) is true but not sufficient. According to the definition of domains, a domain is the set of all valid values from which an attribute may draw its actual value. If all that is needed is that a referencing attribute and its referenced attribute share a common domain then any value in the domain should be usable to both attributes without any constraint. But this is obviously not true for the referencing attribute.

Observe the foregoing example again. It is well accepted that attributes  $S.S\#$  and  $SP.S\#$  share a common domain, say domain  $S\#$  which contains all the possible valid supplier numbers of the database. Then according to the above formulation of the *domain integrity rule*, all values of domain  $S\#$  are allowed to appear in both the attributes without any restriction. But the database semantics, which is described by the *referential integrity constraint* between relation  $S$  and relation  $SP$ , clearly states that values in the referencing attribute  $SP.S\#$  must be values in the referenced attribute  $S.S\#$ . That is to say, a value of domain  $S\#$  could not appear in the referencing attribute  $SP.S\#$  unless the value exists in the corresponding referenced attribute  $S.S\#$ . So the referencing attribute  $SP.S\#$  can not freely assume any value from the alleged underlying domain  $S\#$ . Therefore the “common” domain is not the underlying domain of the referencing attribute  $SP.S\#$  but only the “private”

underlying domain of the referenced attribute  $S.S\#$ . The underlying domain of the referencing attribute  $SP.S\#$  is the set of supplier numbers of the existing suppliers; that is the set of values in the referenced attribute  $S.S\#$ .

Now we are ready to introduce a new concept or a special type of domain: **derived domain**.

**Definition 2.3 Derived domain.** A derived domain is composed of all values from specified attributes of a relation<sup>5</sup>.

In general, a derived domain could be a composite domain which is composed of values from several attributes of a relation. But to make our discussion simple, we will only consider derived domains that only involve one attribute of one relation. This assumption coincides with Date's opinion [Date90A] that relations should have a single-attribute primary key.

In our example database, there should be at least two derived domains. One is composed of values of attribute  $S\#$  of relation  $S$ , which could be used as the underlying domain of attribute  $S\#$  of relation  $SP$ . Currently this domain contains five distinct values: 'S1', 'S2', 'S3', 'S4', 'S5'. Another similar derived domain is the set of  $P\#$  values in relation  $P$ . Each value in this derived domain represents a part which could be shipped by any supplier, hence the value is usable for the attribute  $P\#$  of relation  $SP$ . For reasons of simplicity we will refer to the underlying domains of the deriving attributes as parent domains of the derived domain. So the domain  $S\#$  is the parent domain of the derived domain from attribute  $S.S\#$ .

A derived domain differs from other domains. Other domains are "static" because

---

<sup>5</sup>It is possible for a domain to be composed of values from several relations. In this case the domain could be formed by uniting several derived domains, each deriving values from a single relation.

their values seldom change. A derived domain is dynamic since the domain takes values from a relation and the relation is time-varying. The existence of a derived domain depends on the relation from which the domain takes values. A derived domain is somewhat similar to the concept of *view* in the SQL language. While a view is a virtual relation which acquires values from actual relation(s), a derived domain could also be seen as a unary relation which acquires values from an attribute of a relation.

The fact that domains can be derived from relations clarifies that it is not always true that domains have abstract existence apart from relations. Because the derived domains dynamically acquire values from relations, the existence of the domains depends on the existence of the relations. Hence we say that at least some of the domains (like derived domains) are really dependent on relations.

Another concern involving referential integrity is that we do not think a referential integrity constraint must always involve a primary key and a foreign key; that is, the referencing attribute must be a foreign key in a relation referencing a primary key attribute of a (usually different) relation. The following case represents a counterexample.

Suppose we have a university administration database with five relations:

- LECTURERS(L#, LNAME, ADDRESS), describes all lecturers of the university. Primary key: L#.
- COURSES(C#, CNAME, DESCRIPTION), describes all courses ever available in the university. Primary key: C#.
- STUDENTS(S#, SNAME) describes all students of the university. Primary



key: S#.

- OFFERS(L#, C#), each record describes that a lecturer identified by L# is currently teaching a course identified by C#. Note that not all the courses in relation COURSES are being taught. Primary key: (L#, C#).
- TAKES(S#, C#, MARK) each record describes that a student identified by S# is currently taking a course identified by C#. Note that it is not of interest to the database users which lecturers are teaching which students. Primary key: (S#, C#).

Let us consider the referential integrity problem concerning attribute C# of relation TAKES only. The values of the attribute C# in the relation TAKES represent courses that are being taken by students. But which courses can students take? All the courses ever available in the university, or the courses currently taught by some lecturer? In database terms, the question is: What is the underlying domain of the attribute C# of relation TAKES? Is it a derived domain from attribute C# of the relation COURSES, or a derived domain from the attribute C# of the relation OFFERS? Obviously, the answer is that the courses must be courses taught by some lecturers and the domain must be the domain derived from the attribute C# of relation OFFERS, because a student can only take a course which is currently offered. No “idle” course which is in the relation COURSES but not in the relation OFFERS could appear in the attribute TAKES.C#.

The referential constraint between the two relations is that any value of attribute C# of relation TAKES must have a counterpart value in attribute C# of relation OFFERS. The most important point here is that the referenced attribute C# of

relation OFFERS is not the primary key of the relation (the primary key is the combination of attributes C# and L#) and it is to this non-key attribute that the relation TAKE must refer. This illustrates that a referential constraint is not always based on the primary key and a foreign key of the relevant relations; instead sometimes a reference may happen between non-key attributes. Then it follows that the concept of foreign key, which was initially invented to describe referential integrity, is no longer sufficient. We will see in the next section that referential integrity is nothing but a special case of domain integrity which confines an attribute to a derived domain.

There is another major drawback of the primary key/foreign key approach to the referential integrity problem: The approach does not apply to a relation with more than one attribute to be referenced by different referencing attributes, because only one primary key is allowed in a single relation. In practice this situation will not even be very rare. For example, in a personal information relation, person's name and social insurance number could be both referenced by two different attributes of some other relations.

### 2.5.3 Integrity of Derived Domains

With the concept of derived domains, referential integrity would become simpler to specify and to maintain. The original definition of referential integrity constraints just states that a value in the referencing attribute must match a value in the corresponding referenced attribute. That is, the referencing attribute can only assume values existing in the referenced attribute. Therefore in a relational database system which supports the concept of derived domains, the referenced attribute could be

regarded as the source of a derived domain and the derived domain would become the underlying domain of the referencing attribute. Since the referencing attribute can only draw values from its underlying domain — the derived domain containing values of the referenced attribute in this case — the referential constraint that the referencing attribute can only assume values of the referenced attribute is just an domain integrity constraint that any attribute can only assume values from its underlying domain. In the foregoing example, the referential integrity constraint that each supplier number in the shipment relation SP must have a counterpart supplier number in the supplier relation S could become a simple domain integrity constraint that attribute S# of relation can only assume values from a derived domain which is composed of values in the attribute S# of relation S.

If relational database systems did support the concept of derived domains it would be very easy to achieve referential integrity as a special case of domain integrity. First, one could simply define a derived domain on the referenced attribute. Next, declare the referencing attribute based the derived domain. That is all to specify a referential integrity constraint. We no longer need the concept of foreign keys or of indexes on primary key and foreign key. Perhaps the concept of primary key can also be abandoned as long as we can use the “NOT NULL” declaration to protect so-called entity integrity [Date91], which just prohibits nullified primary key values. For the domain-supported system no special facility is needed to maintain the referential constraint, since the system will certainly guarantee that all the values in any attribute, including the referencing attributes, are drawn from the underlying domains of the attributes regardless of whether the domain is an ordinary domain or

a derived domain. With the concept of derived domains, referential integrity would be achieved naturally during the database design stage. There is no need to code any constraint-checking procedure and the prohibition of on-line updates on the related attributes can be lifted.

It is possible that a referencing attribute may reference multiple attributes from more than one relation. That is referred to as the “multiple-target” reference problem. For the current primary key/foreign key solution to referential integrity problem, it is impossible to have a foreign key referencing several primary keys. But with the concept of derived domains, the problem could be solved by simply defining a derived domain as collecting values from several relations. For example, suppose that in a database there are two relations, say MANAGER and EMPLOYEE, containing ID numbers (attribute EMP#) of existing employees (Note: managers are employees too):

```
MANAGER(EMP#, . . . )
```

```
EMPLOYEE(EMP#, . . . )
```

Suppose a third relation, say ASSIGN, needs to use the ID numbers of existing employees from the two relations. Then the attribute EMP# in this relation could be defined on a derived domain like the view XEMP# created by the following SQL statement:

```
CREATE VIEW XEMP# AS
  SELECT EMP# FROM MANAGER
  UNION
  SELECT EMP# FROM EMPLOYEE
```

In contrast to the primary key/foreign key approach, the derived domain in-

egrity approach can also be used in the case that multiple attribute of a relation are referenced individually by multiple referencing attributes. We can create a derived domain on each of the referenced attributes and define each referencing attribute on the corresponding domain. But in the primary key/foreign key approach, only one attribute could be referenced since only one primary key is allowed in a relation.

In this chapter, some domain-related problems, including data comparability, domain-oriented data manipulation, domain update anomalies and domain integrity in existing relational database systems have been discussed. The significance of this chapter is that the majority of the problems discussed in the chapter do not appear to have been discussed before in the database literature. Further, some serious misunderstandings about domains have been discovered, for the first time again, in this chapter. And it is these misunderstandings that have led to the unfortunate situation that the majority of existing relational database systems do not support domains and hence have domain-related problems. Although people may argue that some (only a few in fact) of the problems discussed in this section have some resolutions in existing relational database systems, we must point out that most of the resolutions are unnatural and tedious. There is no doubt that in order to solve the problems in natural and proper ways, we must support domains in relational databases. In the next chapter, we will discuss the basic features of domain support in relational database systems.

## Chapter 3

---

# Supporting Domains in Relational Databases

---

When examining the domain-related problems in existing relational database systems, we suggested that the natural solution to the problems is supporting domains in relational database systems. In this chapter we will describe the main features of domain support in relational database systems and explain how systems which support domains can solve the problems.

In the first section, we will present the general features of domain support. Then in the following two sections, detailed description of data definition and data manipulation facilities in domain-supported systems will be given. Finally, it will be shown that internal representation techniques can be adopted more easily in systems that support domains than in systems that do not provide this support.

### 3.1 General Domain Support

Relational database systems with domain support will have many new features that cannot be found in systems without domain support. First of all, domains, as independent objects in relational databases, should have physical existence in databases. That means domains, at least some special ones, should be explicitly stored in

databases independently of relations. Second, domain definition and manipulation facilities should be incorporated into the database language to enable the database users properly to define domains, to specify domain integrity constraints and to conduct domain-oriented data operations. With domain support, relational database systems will be free of the domain-related problems.

One of the basic aspects of domain support in relational database systems is that some domains should be physically stored, in order to avoid the domain-related problems — especially the domain update anomalies problems — described in Chapter 2. As mentioned before, none of the existing relational database systems have ever physically stored any domain<sup>1</sup> independently of relations. Such systems do suffer from those domain-related problems.

When all the values of a domain are physically stored, independently of relations, say in a unary system relation of which each tuple stores a distinct value of the domain, a relation-oriented deletion or update on relations which contain values of the domain would only affect the occurrences of the values of the domain but would not affect the values stored in the system domain relation. So the deletion and update anomalies problems would be avoided. Also with the system domain relations, an insertion of a new domain value directly into a domain could be carried out on the corresponding system domain relation and there is no need to insert the same value into any relations which contain values of the domain, so the domain-insertion anomalies problem can also be avoided.

With domains being physically stored, not only can the domain update anomalies

---

<sup>1</sup>In fact in these systems a domain is just a “would-be”, because there is no such a concept in these systems at all.

problems be avoided, but data manipulations on domains will also become possible. We will see domain-oriented data manipulations in Section 3.2.

Do all the domains of a database need to be stored in their entirety? The answer is no. Suppose that domain QTY consists of all integers between 0 and 1000 and the system is aware that any integer greater than 1000 or smaller than 0 is illegal for the domain. Then the 1001 different legal integer values of the domain obviously need not be explicitly stored separately from the relations of the database. For such a domain the system is capable of maintaining the integrity of the domain by storing the lower bound and the upper bound of the domain.

In contrast, a domain whose values are enumerated needs to be physically stored separately from relations. For example, suppose the domain of cities contains four values 'London', 'Paris', 'Rome', 'Athens'. Then only these four values can be used in any attribute defined on the domain. If the four values were not stored then the system would not be able to validate any value to be used in any attributes based on the domain, and would thus be unable to maintain the integrity of the domain. Also, if there is no place to store values of the domain, then an attempt to insert a new value into the domain would fail unless the value was inserted into a relation at the same time.

In practice, which domains need to be stored and which domains do not need to be stored is a question that should be primarily decided by the database administrators or the database users, and not by the system. The major decision-making considerations are:

- Once a value is inserted into the database, should it be retained in the database



forever? If so, then the domain needs to be stored.

- Is it necessary to be able to insert a new value into the domain without insertion of the value into any relation? If so, then the domain needs to be stored.

However, there are some domains that ought to be stored automatically. For example if the values of a domain are enumerated at the database design stage then all the values should definitely be stored. Otherwise there would be no way for the system to validate any newly inserted data value against the enumerated values. We will discuss such kinds of domains in more detail in the next section.

Supporting domains in relational databases is not just a matter of storing domains; there are more features of domain support, which are the topics of the following sections.

## 3.2 Data Definition Facilities

In existing relational database systems which claim to support domains, the only domain definition facility seems to be the ability to specify the data types for domains. For example, in QBE, domain QTY could be specified as being of INTEGER data type and domain S# of character string type. The data types available in relational systems vary among systems, from the fundamental types INTEGER, REAL and STRING to some more complicated types such as DATE, MONEY, etc.

In the literature, data types are regarded as primitive [Date83] domains and natural [McLe76] domains. These “data type” domains are so primitive that in reality no attributes of relations can take all the valid values of the data types as

their actual domains. In fact, just about every “data type” domain contains some values which are useless to attributes based on the data type. For example, if the domain QTY is of INTEGER data type then its valid value -1 will not be used in some attribute, say QTY of relation SP, based on the domain. Therefore the “useless” values, though theoretically legal in the domain, are in practice invalid for attributes.

In existing systems, once a data type is assigned to a domain then all the values of the data type are considered valid values for every attribute based on the domain, and there is no way for the systems to validate each individual domain value for each individual attribute. Thus specifying a data type for a domain could hardly prohibit useless (and sometimes illegal) values from being used in the attributes based on the domain. Then the fundamental purpose of domain support in database systems, that is to guarantee the domain integrity, is never achieved in the systems.

Actually, the only benefit of the above simple domain definition mechanism is the so-called “global column definition”[Date90A]. When several attributes share the same data type, then instead of specifying the same data type to each of the attributes, the data type could be assigned to a domain and all the attributes could be defined on the domain. Such a mechanism not only saves keystrokes but also prevents multiple attributes that ought to have the same definition from inadvertently being given different definitions.

From our point of view, the main purpose of domain definition in relational databases is to inform the system of the exact contents of each domain, so that the systems will be able to maintain the integrity of the domains on its own initiative.

Therefore the definition of a domain should describe, as precisely as possible, the actual set of values of the domain. And in order to achieve such a goal, sometimes the legal values of the domain may have to be enumerated in the definition of the domain. However there are also other ways to specify the set of values of the domain. For example, suppose the domain QTY consists of integers from 0 to 1000. Instead of enumerating all the 1001 integers in its definition, we can describe the domain as a set of integers which ranges between 0 and 1000 or as the set of integers which are greater than -1 and less than 1001. According to the different ways of specifying the contents of domains, we can classify domains into different **domain types**. For example, the domain QTY is in fact of “ranged” domain type whose values fall into a range between the lower bound value 0 and the upper bound value 1000.

There are several other domain types, and each domain type has its own integrity constraint which is implied by the domain type. For example, the integrity constraint of domain QTY is implied by the “ranged” domain type: the values of the domain are equal to or greater than the lower bound value of the domain (0 in this case), and are less than or equal to the upper bound value of the domain (1000 in this case).

### 3.2.1 Types of Domains

Now we describe the most common domain types. Note: the domain types are not mutually exclusive, or in other words, a domain can possibly belong to more than one of the domain types if it bears all the properties of the types.

1. **Enumerated domain** — A domain is enumerated if the values of the domain could be explicitly listed. For an enumerated domain, the data values are

usually enumerated during the domain definition stage while in some cases the values of the domain could also be inserted into the domain via explicit domain-oriented insertion operations. But it is not possible to insert any new value into the domain through any relation-oriented insertion operation. In other words, any attempt to insert a value into an attribute based on an enumerated domain will invoke a check on the existence of the value in the enumerated domain. Only if the value has already been enumerated or inserted into the domain, will the operation be accepted and will the value be inserted into the attribute; otherwise the insertion will be rejected. The enumerated domain is useful in practice to define domains whose values are all known at the database design stage.

In the example database, suppose there are certain valid CITY values, say the set {'London', 'Paris', 'Rome', 'Athens'}, for the attributes S.CITY and P.CITY, then we can define a domain CITY enumerated with the values in the set. If the attributes P.CITY and S.CITY are defined on the domain, then only the values in the above set would be allowed to appear in any relation based on the domain.

In standard SQL there is a mechanism to enumerate values for an attribute of a relation: the CHECK clause in CREATE TABLE command. But if more than one attribute assumes values from the same set, or from the same domain, then the same declaration must be repeated for each of the attributes. Another problem for the SQL approach is that after the table has been created, there is no way to add any new legal value into the set. In this case, it will be necessary

to redefine the database for adding only a single new value to the enumerated set.

2. **Pictured domain** — A domain is pictured if the values of the domain share a similar pattern and only values matching the pattern will be allowed in the domain. For example, the domain of attribute S.S# could be pictured as character strings starting with a capital letter 'S' and followed by one or two digits. Then values like 'S1', 'S99' will be accepted by the attribute defined on the domain while values 's1', 'SS1' and 'S100' will not be accepted. Pictured domains are very common in applications. Especially when entities need to be assigned some unique identifiers, it is usual to give the identifiers a similar pattern, as in the example of S# as identifier of suppliers.
3. **Calculated domain** — A domain is calculated if its values are “calculated” from values of some other domain(s). For example, with domains YEAR, MONTH and DAY, a fourth domain DATE whose values are the combinations of values from the three domains can be composed. The other possible ways to generate new domains from existing domains is applying the traditional set operations, such as *union*, *intersection*, *difference* and *Cartesian Product*<sup>2</sup>, to domains. It seems that the most common calculated domains are domains formed from existing domains by the set operation UNION so in the next chapter we will only propose “united” domains instead of generic calculated domains.
4. **Ranged domain** — A domain is ranged if its values are all in some specified data interval. In practice, most numerical domains are indeed ranged domains

---

<sup>2</sup>These operations are basic relational algebra operations.

since at least in a computerized database no domain could be infinite. In our example database, if the weight of a part can only range from 1 to 100 then the domain of attribute P.WEIGHT should be defined as a set of integer numbers ranging between 1 to 100. Therefore any value to appear in the attribute must be within the interval between 1 and 100. It is possible that the values of a ranged domain span several separate intervals; in that case the domain should be a “united” domain of several ranged domains of which each range is an interval.

5. **Derived domain** — A domain is derived if its values are derived from an attribute of a relation. This kind of domain is a little like the concept of active domain [Mair83, Yang86]<sup>3</sup>. But the purpose of derived domains is entirely different from that of active domains. As described in the previous chapter, a derived domain can be used to specify so-called referential integrity constraints, which we treat as a special cases of domain integrity constraints. Since our previous discussion on derived domains is detailed enough, we will only briefly describe the data comparison issues of derived domains here.

A problem concerning derived domains is data comparison. In the previous chapter, we suggested that in domain-supported systems, data comparison should be restricted to attributes based on the same domain or on compatible domains. Here we must point out that derived domains are not independent domains; rather they are dependent on some other domains. All the values of a derived domain come from the underlying domain of the attribute, from

---

<sup>3</sup>An active domain is composed of the current occurrences of domain values in the database.

which the derived domain acquires values. That means the derived domain is just a dynamic subset of the underlying domain. Thus data comparison and, more importantly, joining of relations on a derived domain and its underlying domain should be allowed in domain-supported systems.

6. **Exclusive domain** — A domain is exclusive if each value of the domain can only be assumed once in the database. That means each value of the domain can have only one occurrence in the database. For example, in an enterprise, each employee and each manager is assigned an employee number and in the database of the enterprise two relations are used to store employee information and manager information separately. The domain of the employee numbers should be exclusive within the two relations, since an employee number could not be shared by two different persons, no matter whether they are managers or ordinary employees. The scope of an exclusive domain is not always the entire database; instead it may be only one relation or several relations of the database like the two relations for the domain of employee numbers in the above example.
7. **Multityped domain** — A domain is multityped if the values of the domain can be represented using different data types. This domain type is used to solve the intra-domain data comparison problem when data of the same domain are represented in different data types. Each multityped domain should have a main data type and one or more alternative data types and often the main type should be a numerical data type so that ordering of the values of the domain would be possible. Since the data types available in many database

systems are limited, we doubt that most multityped domains will allow two basic data types: integer and character string with the former one as the main type.

For each value of a multityped domain, the number of different representations is not necessarily limited, even when only two data types are allowed in the domain. For example in the multityped domain MONTH, the value of the last month in a year could be represented as the integer 12 and as the character strings 'December', 'Dec.', etc.

The main problem with multityped domains is that there must be a mechanism to recognize equality between different representations of the same value and to distinguish between some value and an alternative representation of some other value. We will discuss this problem in the next section. However we must point out that all the different representations of a multityped domain must be stored in the database separately from relations, perhaps in a binary system relation with one attribute as the value in the main data type and another attribute for the alternative representation in any other data type.

8. **Multiunit domain** — A domain is multiunit if its values could be represented in several units of measurement. The multiunit domain may be seen as a special case of multityped domain. But there are some differences between them. First, the values of a multiunit domain are all in one data type, say real numbers, instead of in multiple data types for a multityped domain. Second, the actual values of a multiunit domain stored in the database are all in one unit, the default unit of the domain, so there is no need for a system relation



to store values of different units.

We suggest that the best way to handle the multiunit domains is to assign a default unit and several alternative units to the domain. Then at any time one of the units is the “current” unit of the domain. So every piece of data input into and output from the database is represented in the current unit. Once the current unit is changed to another unit, the data will appear in the new current unit. We certainly cannot use multiple units simultaneously unless data are explicitly accompanied by their unit, like 10 pounds, 1 Dollar, etc. It is not convenient for both the users and the systems to attach unit information to the data values.

The main purpose of the multiunit domain, similar to that of the multityped domain, is further to improve data comparability and make the database more flexible.

9. **Stored domain** — A domain is stored if the values of the domain are physically stored in the database separately from all relations. As pointed out before, only if domains are physically stored separately from relations can domain update anomalies be avoided. If a domain needs to be free of update anomalies, then it should be stored. The system will only automatically store enumerated domains and multityped domains. For other domains the database administrator may decide whether they should be stored or not. As stored domains have been discussed extensively in the previous section, we will not repeat the discussion here.

In the above discussion domains have been classified into several domain types, not according to the data types of the domain values, but according to the different types of domain integrity constraints of the domains. Domains of the same type share similar integrity constraints, so with these different domain types, it becomes unnecessary to describe the integrity constraints for each individual domain. Because the integrity constraints are implied by the domain type, defining domain integrity constraints becomes a problem of choosing the appropriate domain type for a domain. Note again: Sometimes a domain may hold several integrity constraints, so a domain can be a combination of several types. For example, domain QTY could be both ranged and multiunit.

### 3.2.2 Defining a Database

Defining a database in a domain-supported database system is quite different from defining a database in an ordinary database system. It takes two steps rather than one step as in ordinary systems.

First of all, before defining a relation, all the underlying domains of the relation must have been defined, because relations in such a system are built on domains. In order to precisely define a domain, we need to specify the domain type(s) for the domain which in turn sets up the corresponding domain integrity constraints on the domain/attribute. Also we need to specify the data type of the domain<sup>4</sup>.

Secondly, with domains being properly defined, relations can be built on the pre-defined domains. Building relations on domains is simpler than building relations

---

<sup>4</sup>If the domain is of multityped type, what needs to be specified is the main data type of the domain.

on “data type domains”. For each attribute of a relation, we just need to specify an appropriate domain as the underlying domain of the attribute. We do not need to specify the data type of the attribute because that is already defined as the data type of the underlying domain. For a relation we only need to define all the attributes of the relation. Some other aspects of relation definition in traditional systems, like primary keys and foreign keys, no longer need to be specified because they were originally used to specify so-called “referential integrity” constraints which become simple domain integrity constraints on “derived” domains in domain-supported systems.

### 3.2.3 Other Features

In a domain-supported system, it is desirable to be able to specify whether NULL or unknown values are allowed in a domain in addition to the ordinary NULL specification on relations. If NULL is not allowed in a domain then NULL will not be allowed in any attribute based on the domain. But if NULL is allowed in a domain, then some attributes which are defined on the domain may still not accept NULL. When NULL is allowed, it is possible to have a default value for NULL in a domain.

In addition to all the above, there are more features which can be incorporated into data definition facilities. For example, one might include the ability to create a *view* or a subdomain of an existing domain, and the facility to drop a domain definition and so on.

A RDBMS with perfect domain definition and relation definition facilities will relieve database programmers and database administrators of a significant burden of maintaining data integrity. For instance, if the domain WEIGHT is declared as *ranged* from 0 to 100, a negative value or a value greater than 100 would be

automatically rejected by any attribute defined on the domain. Programmers thus no longer need to specify any code to validate the values to be used in any of the attributes based on the domain.

### **3.3 Data Manipulation Facilities**

In domain-supported relational database systems, some domain manipulation facilities should be implemented. The facilities differ from the domain calculus [LaPi77] which is used for expressing data manipulations on relations; instead, they are the facilities that manipulate domains. In addition to domain manipulation facilities, ordinary relation-oriented operations will certainly have some side-effects on domains. This section discusses these facilities and side-effects.

#### **3.3.1 Data Dictionary Inquiries**

In a domain-supported system, the database administrators (DBA) and/or database programmers need some facilities to obtain structural information about domains from the data dictionary of the system. When designing or maintaining a database, a DBA will typically refer to the data dictionary for the characteristics of domains. When coding applications, a database programmer may need to check associations between domains and attributes. Typical system catalogue inquiries on domains are:

1. List all the underlying domains of a database or a relation;
2. List the domain on which a particular attribute is defined;
3. List all the attributes that are defined on a particular domain;

4. List all the relations of which one or more attributes are defined on a particular domain;
5. List the name of the derived domain(s) that is based on a particular attribute or relation;
6. List the attribute or the relation from which a derived domain acquires values.

To handle the above inquiries in a domain-supported system, we do not need any special facilities; instead the above inquiries could be expressed in terms of ordinary database query languages such as SQL, provided that the system catalogue is organized to facilitate them. For example, a system relation which stores names and other properties of all domains is suitable for answering the first query, and a relation which lists all the attributes and their underlying domains could be used for answering the second query, and so on.

In addition to the above retrieval inquiries on the data dictionary, it might be desirable to have some domain-related data dictionary update operations, such as changing a domain's name, changing the underlying domain of an attribute, etc. These can also be achieved by the ordinary data manipulation language provided the data dictionary is properly organized. Some other operations, like changing the definition of a domain, should be controlled carefully to prevent certain problems, like data type conversion, from happening.

### **3.3.2 Domain-Oriented Operations**

As indicated in the previous chapter, some database operations are intrinsically domain-oriented instead of relation-oriented. A typical example of such operations

is increasing all values of domain PRICE by ten percent. In existing relational database systems, due to the lack of domain implementation, such domain-oriented inquiries are forced to be expressed as relation-oriented data queries and to be carried out on the relations. But in a domain-supported system, domains are considered to be independent objects, so it should be possible to carry out domain-oriented data manipulations on the domains themselves, provided that direct domain manipulation facilities are available.

The typical domain-oriented data operations should be:

- Retrieve a single value, several values or even all values of a domain, regardless of whether the value(s) have corresponding occurrences in relations or not. The purpose of such retrievals might be to check the existence of particular domain values or to list some or all legal values for attributes defined on the domain.
- Insert a single new value or several new values directly into a domain. The newly inserted value(s) will not have occurrences in relations until the value(s) is otherwise explicitly inserted into relations.
- Update a single value, several values or all values of a domain. All the corresponding occurrences of the updated domain value(s) in relations should be updated too. Such updates are intended to update domain value(s) and the corresponding occurrences in relations regardless of whether the value(s) has occurrences in relations or not.
- Delete a single value, several values or even all values from a domain. The corresponding occurrences of the deleted values should also be deleted from

relations. Like domain-oriented updates, such deletions are aimed at both the value(s) and the occurrence of the value(s). Note: Deletion of values directly from domains should be strictly controlled, because it may endanger the database.

- Other operations, like changing the current unit of a multiunit domain and insertion of alternative representatives for a value of a multityped domain.

As indicated in section 3.1, in domain-supported systems, each stored domain can be implemented as a unary system relation which explicitly stores all the valid values of that domain. Therefore the stored domains can be manipulated just as ordinary relations, and thus domain-oriented operations can be expressed as operations on the system domain relations. For example, when we want to insert a new data value into a domain, especially into an enumerated domain, we just insert the new value as a new tuple into the corresponding system domain relation. Also, with all values of a domain stored in a system domain relation, to retrieve all the values of the domain we just retrieve all tuples in the system domain relation.

For a domain whose values are not stored separately from relations, we can always use some system facility to form a virtual system domain relation which contains all the distinct occurrences of domain values in relations<sup>5</sup>. Then we can treat the virtual relation as if it were a real system domain relation and domain-oriented operations thus can be also expressed as ordinary relation-oriented retrieval operations on the virtual system domain relation, though the operations themselves will be carried out on the relations involved.

---

<sup>5</sup>In SQL, we can use "CREATE VIEW" to form such a virtual relation.

At this point, it seems that to accomplish most domain-oriented operations no special domain-oriented facilities are needed, provided that the system creates and maintains some system relations that are either system domain relations storing all values of domains, or virtual system domain relations which contain values of domains that have occurrences in relations. In other words, we do not need special operations to directly manipulate domains; instead we can use the ordinary relation-oriented operations to manipulate domains. The only change is on the object of the operation: from ordinary relations to system relations that represent domains. Usually the system relations are just implied by the names of domains and if the names of domains are distinguished from the name of relations, then the syntax of data manipulation languages will remain unmodified to identify the different objects of data operations. For example in SQL, retrieval of all values of domain CITY can be expressed as:

```
SELECT * FROM CITY
```

Since there is no relation named CITY, the system therefore selects all the tuples from the system domain relation corresponding to domain CITY.

There are some exceptions which need special domain-oriented operation and thus cannot be supported without changing the syntax of the original data manipulation language. For example, in a multityped domain we do need some special facilities to handle values and their alternative representatives. In order to insert a value into a multityped domain we need to specify if the value is a brand new value that has no other representative in the domain or if the value is a new representative of an existing value in the domain. In an SQL-based system, we may modify the syntax



of the INSERT statement to:

```
INSERT INTO [ DOMAIN ] domain-name
VALUES ( constant [, constant ] ... )
[ WHERE VALUE = constant ]
```

An insertion without the WHERE clause adds the constant(s) of VALUES clause to the specified domain as brand new values. An insertion with WHERE clause inserts the values in the VALUES clause into the domain as alternative representatives for the value indicated by the constant in the WHERE clause. Compared with the original INSERT statement of SQL, the only minor modification here is the new key-word “VALUE” which indicates that this is a true domain-oriented insertion on multityped domains. In the next chapter we will see how much the SQL language has been modified in our proposed SQL/D, which is a domain extension to the SQL language.

In a domain-supported system, the database programmer’s awareness of the domain-oriented operations is more important than the existence of the operations themselves. The programmers must be able to distinguish domain-oriented inquiries from relation-oriented inquiries so that the data can be properly manipulated and the integrity and accuracy of the database can be guaranteed.

### **3.3.3 Impact of Relation-oriented Operations**

In this section, we examine the impact of relation-oriented updating operations on domains. Generally speaking, ordinary relation-oriented updating operations, such as deleting a tuple from a relation, inserting a new tuple into a relation and replacing an existing tuple with a new one, may have some side-effects on the underlying

domains. The effect varies according to the types of domains and the types of the operations. For domains which are not physically stored separately from relations, the side-effects of relation-oriented updates are straightforward. If a new value is added into an attribute of a relation, then the value is automatically inserted into the underlying domain. If a tuple is deleted from a relation, the component values in the tuple will be removed from their corresponding domains if the values have no other occurrences in the database. But for some special types of domains, the case is more complicated. The following are some important ones.

For an enumerated domain, the natural way to insert, delete or update data values is via some domain-oriented operations. An attempt to update data on an attribute which is defined on an enumerated domain should have no effect on the domain. That is if a value is deleted from a relation, the value should remain in the domain and if a new value which is not enumerated in the domain is to be inserted via an attribute defined on the domain, the insertion should be rejected. Actually in order to make the system more flexible, we may suggest an option allowing users to insert new values into an enumerated domain via successful insertions of the values on some special attribute defined on the domain. These special attribute(s) should be carefully chosen by the DBA.

For a stored domain the situation is similar to an enumerated domain with only one exception: a new value can be inserted into a stored domain via an insertion of the value into an attribute based on the domain. But when the last occurrence of a value is deleted from relations, the value should remain in the domain because it is “stored” in the domain.

The case of multityped domains is different from the above domain types because in this type of domain there are principal values and their alternative representatives. First, the principal values should be treated as ordinary values of a plain domain. Then the alternative representatives should be treated similarly to the values of enumerated domains since each alternative representative must be explicitly inserted into the multityped domain via a domain-oriented insertion before it can be used in relations.

The case of derived domains is more complicated. The scope of an update on the attribute from which a derived domain derives values is the whole database including the derived domain itself and all the attributes defined on it. But an update on any attributes which are defined on a derived domain should not affect the referential attribute and its underlying domain. For example, an update on attribute  $S\#$  of relation  $SP$  will not affect the attribute  $S\#$  in relation  $S$  and its underlying domain. Ironically while domain-oriented data manipulation facilities are introduced for most of the domain types, no domain-oriented update operations can apply to a derived domain because the values of a derived domain are drawn from a relation and do not have their own existence apart from the referential relation. However, an update on an attribute which is defined on a derived domain is allowed provided the update does not violate the integrity of the derived domain. The scope of such an update is limited only to the relation which contains the attribute.

For a calculated domain, the situation is similar to that of derived domains because the values of a calculated domain are derived from another domain while the values of a derived domain are derived from an attribute or are derived indirectly

from another domain.

When implementing domains in relational database systems, in addition to incorporating the domain-oriented data manipulation facilities into the data manipulation language, the relation-oriented data manipulation facilities must be modified to incorporate the above-discussed side-effects on domains. We will see such impact in our implementation discussed in chapter 5. Also the database programmers must be aware of the side-effects so that databases can be properly manipulated.

### 3.4 Adopting Internal Representation Techniques

In most existing database systems, data are stored in files on their original format. An integer is stored as a binary integer; a 10-character string is stored as a 10-byte-long field with each byte containing the binary code of a character, and so on. Such a simple data storage scheme is easy to implement but has a major disadvantage: more storage space.

It has been suggested for years to use some kind of internal representation to replace the original data in database files. But for some reasons, the techniques are not widely adopted. One of the considerations is that the original data must be kept somewhere in the database. That will need some extra storage space.

Now in a domain-supported system, since domains (at least some of domains) are physically stored in system relations, it becomes possible and easy to adopt internal representation techniques. The simple way is using binary relations to store domains. The first attribute of the relation is used to store the original data and the second attribute is used to store the corresponding internal representations. Then in

database files, which represent relations, the internal representations<sup>6</sup> will be used.

In addition to saving storage space, adopting internal representation techniques may also benefit database systems in some other respects: security and portability. Security is gained because the actual data values are hidden by their internal representations in the files. Portability is improved by the fact that data of different formats or data in different human languages, say in English and in Chinese, would be exchangeable when the data are represented in the same standard internal format.

Internal representation is an old technique. In relational databases that do not support domains, it is hard to implement it, but in the systems that support domains the scheme is easier to implement especially for the “stored” domains. Then to efficiently implement such techniques, some careful studies on the techniques are still needed.

In this chapter we have described the main features of domain support in relational databases. In the next chapter we will see how the features could be added into a relational system without rebuilding the system from the bottom.

---

<sup>6</sup>At least for the attributes based on the stored domains, the internal representatives can be used.

## Chapter 4

---

# SQL/D — Domain Extension to SQL

---

In the previous chapters, we have already shown that it is worthwhile to support domains in relational database systems. Now a new question arises: how much effort does it take to achieve that goal? Is it necessary to rebuild relational database systems in their entirety to get the extra domain features? In this chapter and the following chapter, we will answer this question by introducing our proposed SQL/D, a domain extension to the SQL language and our current (partial) implementation of SQL/D on the SYBASE database system.

We choose SQL as a vehicle for illustrating domain support in relational databases for the following reasons. First, “for better or worse, SQL is intergalactic dataspeak” [CADF90]. During the evolution of the relational data model various systems and languages have been developed and proposed. But after twenty years of competition, SQL is apparently the overwhelmingly dominant language in the relational database market. Some non-SQL systems, like INGRES, had to adopt an SQL interface before being commercialized. So it seems that SQL must be the most common language that both we and our readers are fluent in. The second reason for choosing SQL is that current SQL does not support the domain concept at all. It is obviously

more convincing to show how domain features could be incorporated into a totally domain-free environment.

## 4.1 A Brief Review of the SQL Language

The SQL language was first designed as the sole data language to be used in System R, which in turn was IBM's first prototype relational database system. Because of its advanced features and its simplicity, SQL rapidly became the internationally recognized standard relational data language. Although SQL is an abbreviation for "Structured Query Language", the language itself is much more than a "query" language. In fact, SQL is a unique relational data language which integrates data definition, data manipulation and data control facilities all into a single language.

The principal SQL data definition statements are:

CREATE/ALTER/DROP TABLE — Create, alter or drop a base table.

CREATE/DROP VIEW — Create or drop a "virtual" table.

CREATE/DROP INDEX — Create or drop an index on a base table.

The four powerful SQL data manipulation statements are:

SELECT — Retrieve data from relation(s).

INSERT — Enter data into a relation.

UPDATE — Update data in a relation.

DELETE — Remove data from a relation.

Two typical SQL data control statements are:

GRANT/REVOKE — Authorize data manipulation privilege to user(s).

COMMIT/ROLLBACK — Commit or rollback a transaction.

In addition to the above statements, SQL also provides some aggregation functions, for example, SUM(), COUNT(), etc., which are mainly embedded into SELECT statements to tailor sophisticated data requests. SQL is also characterized as an English-like language. It is easy to understand and easy to use, especially for its famous SELECT-FROM-WHERE “Query Block” structure.

As early as 1986, SQL had been adopted as an official standard language for relational database systems by the American National Standards Institute (ANSI), but even now none of the actual SQL systems support exactly the pure ANSI version of the language; every system has its own particular dialect of SQL. Besides, ANSI seems to modify its standard draft every year. This makes it a little difficult to choose an appropriate SQL version to illustrate our domain extension. However, we finally selected the SYBASE dialect because SYBASE is a relatively new SQL product, so its SQL is closer to the standard SQL. Also, SYBASE, the only SQL resource available at this department, is the system on which we are implementing our proposal.

## 4.2 The Syntax of SQL/D

Before presenting the syntax of the proposed SQL/D, it is necessary to point out that SQL/D is not a completely new version of the SQL language. Instead it is a proposed extension to SYBASE SQL with domain definition and domain manipulation features. Therefore, we will only include the extended statements in the following syntax presentation. Those trivial SQL components, like CREATE INDEX, COMMIT, etc., which are not relevant to the domain concept and require no change to



their syntax, will not be covered.

In designing the proposal, one of our main criteria is to stay as close as possible to the standard SQL language and to introduce as few new keywords as possible. We will see the effect later.

#### 4.2.1 Syntax Conventions

We will not use the formal Backus-Naur Form (BNF) to describe our SQL/D proposal; instead the more conventional “upper case/lower case and brackets” method will be used. The reason is simply that this method, which nowadays is not only prevalent throughout the data processing industry but also widely adopted in the literature, is the dominant method used to describe SQL and any proposed extensions to SQL in related books and papers.

The elementary syntactic notations which we will use throughout this chapter are explained below.

- Square brackets ([ ]) denote optional element(s).
- Curly brackets ({ }) denote an obligatory group of elements from which one and only one element must be chosen.
- The vertical bar separates (|) alternatives from a multiple choice list of elements.
- Ellipses (...) indicate elements that may be repeated one or more times. If a comma appears prior to the ellipses (,...), a comma must then be used as a separator between any two consecutive elements. For example,

{literal}... means *literal1 literal2 literal3* and so on, while

{literal},... means *literal1,literal2,literal3* and so on.

- Words in upper case and lower case letters are KEYWORDS and elements respectively. Elements are written in underscore-hyphenated words in the general format for a more precise presentation. For example, the word ‘domain name’ is written as ‘domain\_name’ in the format. The format of user-provided elements, like domain\_name and so on, is not significant to the domain concept and is not specified.

#### 4.2.2 Syntax of the Proposed SQL/D Extension

In this section we present the syntax of the proposed SQL/D extension to SQL. Then in the following section we discuss the details of the proposal. Note that, except for the reserved keywords, the standard SQL terms “table” and “column” are replaced by SQL/D terms “relation” and “attribute” respectively. This is because SQL/D is more relational than the original SQL.

```
CREATE DOMAIN domain_name
{ [DERIVED] AS selection_subquery
  | UNION ( domain_name {,domain_name}... )
  | ( {domain_name [[NOT] NULL]},... )
  | data_type_description
  | [[NOT] NULL]
  | [{UNIQUE | EXCLUSIVE} [IN {( relation_name[.attribute_name]
    {,relation_name[.attribute_name]}... )},...]]
```

```

[MULTIUNIT [DEFAULT=]default_unit_name
  {,alternative_unit_name=constant}...]
[RANGE FROM lower_bound TO upper_bound]
[ENUMERATED [( {constant},... )] [INSERTION [ALLOWED]
  {relation_name[.attribute_name]},...]]
[PICTURED picture_description]
[MULTITYPED {data_type_description},...]
[CARDINALITY integer_cardinality [OCCURRENCE integer_occurrence]]
[STORED] }

```

The data\_type\_description in the SYBASE environment could be as below.

```

{ BIT | BOOLEAN
  | TEXT
  | {CHAR | VARCHAR}[(length)]
  | {FLOAT | REAL}
  | {BINARY | VARBINARY}[(length)]
  | {INT | SMALLINT | TINYINT}
  | {MONEY | SMALLMONEY}
  | {DATETIME | SMALLDATETIME}
  | {IMAGE | TIMESTAMP | SYSNAME }
}

```

```

DROP {DOMAIN domain_name | TABLE relation_name | VIEW view_name}

```

```
CREATE TABLE relation_name  
( {attribute_name [ON] domain_name [[NOT] NULL [UNIQUE]]},...  
[UNIQUE {( {attribute_name },... )},... ] )
```

```
ALTER TABLE relation_name ADD  
( {attribute_name [ON] domain_name [UNIQUE]},... )
```

```
SELECT  
{ [ALL | DISTINCT] {*} | {value_expression},...}  
FROM {relation_name [alias_name]},...  
[WHERE search_condition]  
[GROUP BY {[relation_name | alias_name].attribute_name},...]  
[HAVING search_condition]  
| VALUE FROM domain_name  
}
```

```
INSERT INTO  
{ relation_name [( {attribute_name},... )]  
{ VALUES ( {constant},... ) | select_subquery}  
| [DOMAIN] domain_name VALUES ( {constant},... )  
[WHERE VALUE=constant]  
}
```

```
DELETE [CASCADE]
FROM {relation_name | domain_name}
[WHERE search_condition]

UPDATE
{ relation_name [CASCADE] SET {attribute_name = expression},...
  [WHERE search_condition]
  | domain_name SET VALUE=expression [WHERE search_condition]
  | UNIT SET CURRENT=unit_name WHERE DOMAIN=domain_name
}

{GRANT | REVOKE}
{ALL | {SELECT | INSERT | DELETE | UPDATE},...}
ON {domain_name | relation_name[.attribute_name]},...
TO {PUBLIC | {user_name},...}
[WITH GRANT OPTION]
```

### 4.3 Data Definition in SQL/D

Data definition in SQL/D is quite different from that of ordinary SQL. First, SQL/D domain definition facilities make it possible to precisely define various types of domain described in Section 3.2.1. Then with domains created, relations are no longer defined on the system-provided “data type” domains but are established on top of pre-defined domains. Data base definition in SQL/D consists of two parts: domain

definition and relation definition.

For simplicity, from now on our discussion of SQL/D, except where indicated, will be limited to atomic domains only. In other words, we will assume that all domains in SQL/D are atomic.

#### 4.3.1 Defining Domains in SQL/D

The only domain definition statement in SQL/D is CREATE DOMAIN. It provides all the domain definition features discussed before. This statement is totally new to SQL. The features of CREATE DOMAIN include:

- **domain\_name**

Domain name is the unique identifier for the domain being defined. It is desired that domain\_names be distinct from relation names so that the syntax of some SQL/D data manipulation statements can be simplified. For more detail see later discussion.

- **[DERIVED] AS selection\_subquery**

This option is used to specify that the named domain is a derived domain. The selection\_subquery works in a similar manner as the selection subquery in CREATE VIEW of SQL. It determines the set of valid values for the named domain. For example, the derived domain S\_S#, which consists of all S# values in relation S, is specified as:

```
CREATE DOMAIN S_S# DERIVED AS SELECT S# FROM S
```

The data type<sup>1</sup> of a derived domain is always inherited as that of the parent domain of the named derived domain. In the above example, the data type of domain `S.S#` is the same data type of domain of attribute `S.S#`. The optional keyword `DERIVED` has no special effect, it is just for precision.

- **UNION ( domain\_name {,domain\_name}... )**

This option is used to form a united domain from at least two existing domains, which are listed between the parentheses. The united domain consists of all distinct values of the participating domains. The participating domains are required to be compatible, that is, they must be of same data type. Therefore the data type of a united domain is the same as that of the participating domains.

Note: The key word `UNION` could be `INTERSECT` or `DIFFERENCE` to form a domain by the corresponding set operations on domains. In order to simplify the SQL/D syntax, we did not include them here.

- **( {domain\_name [[NOT] NULL]},... )**

This is used to show how a composite domain, which is the Cartesian product of several existing atomic domains, could be formed if SQL/D did support this type of domain. For reasons mentioned before, we would not discuss this type of domain here.

The above three exclusive options define domains which derive values from other existing domains. The next option is used to create new domains inde-

---

<sup>1</sup>In fact, not only the data type but all the properties of the parent domain should be inherited by derived domain.

pendently of other existing domains.

- **data\_type\_description**

This specifies the data type for the named domain. If the domain is a multi-typed domain, then the data type specified here is the main data type for the domain (see MULTITYPED option for more detail).

We suggested before that a domain support system must provide rich data types, or in our terms, rich built-in domains. The data types supported in SYBASE are listed below the CREATE DOMAIN syntax. We can see that SYBASE does provide relatively richer data types than earlier SQL systems. However, as the selection of data types is not a main topic of this proposal, we prefer to leave it as a further research subject to interested readers.

Note that the following options are not exclusive. we suspect that in practice most domains will have only a few of the properties.

- **[[NOT] NULL]**

This specifies whether null is globally allowed in all the attributes which are defined on the named domain. “Globally” here means that the scope of the domain NULL specification is the entire database. The option NOT NULL prohibits null in all attributes defined on the named domain, since null is not allowed in the named domain. The option NULL allows null in any attributes defined on the domain unless it is otherwise prohibited in relation definition (see CREATE TABLE for more detail). If this option is not chosen, the default is NULL.



- `[{UNIQUE | EXCLUSIVE} [IN { ( relation_name[.attribute_name]`  
`{,relation_name[.attribute_name]}... )},...]]`

The keywords `UNIQUE` and `EXCLUSIVE` have the exactly same meaning. While the former is a well known SQL preserved word, the latter is more meaningful in this context. This specifies that the values of the named domain are assumed exclusively. If the `IN` clause is omitted then each value of the domain can only be assumed once in the entire database, while the `IN` clause otherwise specifies alternative scopes for the values of the domain. The alternative scope must consist of at least two attributes, since the `UNIQUE` clause in `CREATE TABLE` could prohibit redundant values in a single relation. The attributes could be identified by names of their containing relations, or be explicitly identified by the unique identifiers of the attributes. The following example specifies that each value of domain `EMP#` can only appear once either in the `EMPLOYEE` relation or in the `MANAGER` relation:

```
CREATE DOMAIN EMP# char(4) EXCLUSIVE IN (EMPLOYEE, MAN-
AGER)
```

- `[MULTIUNIT [DEFAULT=]default_unit_name`  
`{,alternative_unit_name=constant}...]`

This indicates that the named domain is a multiunit domain. It assigns a default unit, plus one or more alternative units, to the named domain. At different times, the values of the domain can be represented in different units, though the actual domain values stored in the database are always in the

default unit. At any given time, only one of the multiple units is in effect and the effective unit is called the CURRENT unit of the MULTIUNIT domain. In the definition, the `default_unit_name` and `alternative_unit_name` specify the names of the default unit and the name of each alternative unit respectively, while the constant<sup>2</sup> provides the value used to convert data represented in an alternative unit to data in the default unit and vice versa. Note that a MULTIUNIT domain must be of numerical type, usually of type real number, since the constants used in conversion are almost always real numbers.

The following example indicates that values of domain WEIGHT could be represented in one of the three units: a default unit KG (kilogram) and two alternative units LB (pound) and GRAM (gram). Suppose the CURRENT unit of domain WEIGHT is LB. Then any newly inserted WEIGHT values will be divided by 2.2046 before being stored into the database and all WEIGHT values retrieved from the database will be multiplied by the corresponding conversion constant 2.2046 before being submitted to the user.

```
CREATE DOMAIN WEIGHT real MULTIUNIT  
  
    DEFAULT='KG', 'LB'=2.2046, 'GRAM'=1000
```

In order to switch among the multiple units, we extend the SQL UPDATE statement with the following option.

```
UPDATE UNIT SET CURRENT=unit_name  
  
    WHERE DOMAIN=domain_name
```

---

<sup>2</sup>In more general cases, to convert values in one unit to values in another unit, we may need a formula or a procedure instead of a constant. For example, conversion of Celsius temperatures to Fahrenheit temperature. But for reasons of simplicity we only use constants in SQL/D.

It states that the system updates CURRENT UNIT of the domain\_name domain to unit\_name unit. It will seem to the database users that this statement changes all the values of the domain from the current unit to a new CURRENT UNIT, namely unit\_name unit. For example, to change unit of WEIGHT domain from default 'KM' to 'LB', we use:

```
UPDATE UNIT SET CURRENT='LB' WHERE DOMAIN=WEIGHT
```

- **[RANGED FROM lower\_bound TO upper\_bound]**

This indicates that the named domain is a ranged domain. The lower\_bound and upper\_bound set the interval for values of the named domain. The data type of a ranged domain can be either numerical or character string as for the domain. The lower\_bound and upper\_bound should be of the same type, too. The following example specifies that values of QTY domain cannot exceed the integer interval between 0 and 1000.

```
CREATE DOMAIN QTY int RANGED FROM 0 TO 1000
```

If a domain has several separated ranges, we will first use this option to create several separate domains for each of the ranges, and then use the UNION option to unite all the ranges to form the required “multi-ranged” domain.

- **[ENUMERATED [( {constant},... )]**

**[INSERTION [ALLOWED] {relation\_name[.attribute\_name]},...]**

This indicates that the named domain is an enumerated domain. The format ( {constant},... ) is used to enumerate the valid values of the domain. Usually such an enumeration should be a mandatory part in the definition of a

enumerated domain. But in some cases, it is possible that, at the database design stage, the valid values are not yet available and thus the enumeration is infeasible. So here this enumeration part is optional. To append new values *directly* into an enumerated domain after the completion of domain definition, we will use the amended SQL INSERT INTO statement:

```
INSERT INTO [DOMAIN] domain_name VALUES ( {constant},... )
```

Another way to add new values into an enumerated domain is *indirectly* through an ordinary relation insertion operation. But this will only happen in some exceptional cases and it should be strictly restricted. The INSERTION ALLOWED option specifies the restricted relation(s) and/or attribute(s) from which new values are allowed to be indirectly added to the enumerated domain when the data are implicitly inserted into the relation. If an attribute\_name is specified (and the attribute is defined on the named domain) then, whenever a value which is new to the enumerated domain, is inserted into the specified attribute, it will also be inserted into the domain. If relation\_name is specified, the same will apply to all attributes of the relation which are defined on the enumerated domain.

The following example specifies that CITY is an enumerated domain with three initial values and indirect insertion of new values is allowed through the attribute CITY of relation S.

```
CREATE DOMAIN CITY char(20) ENUMERATED  
( 'LONDON', 'PARIS', 'ATHENS' ) INSERTION ALLOWED S.CITY
```

- **[PICTURED picture\_description]**

This specifies the literal “picture” or pattern for the named domain. All values in the domain must match the specified picture. The special symbols used to describe the picture coincide with those used in the “LIKE” predicate of the SQL SELECT statement. In SYBASE, an underscore (.) represents any single character while a percentage mark (%) represents a sequence of  $n$  characters (where  $n$  may be zero). All the other characters simply stand for themselves. The following example indicates that values of domain S# must start with character “S” then followed by one any other character.

```
CREATE DOMAIN S# char(2) PICTURED 'S_'
```

- **[MULTITYPED {data\_type\_description},...]**

This specifies that the named domain can take values of multiple data types. The data\_type\_description here specifies the alternative data types for the domain. As indicated earlier, usually a multitype domain should also be an enumerated domain. The format to attach alternative values, which are enclosed in the parentheses, to a value of the main data type, which is specified by the VALUE clause, is:

```
INSERT INTO [DOMAIN] domain_name VALUES ( {constant},... )  
  
WHERE VALUE=constant
```

The following example specifies that domain MONTH is an enumerated multityped domain. The main data type and alternative data type are integer and character string respectively.

```
CREATE DOMAIN MONTH int ENUMERATED (1,2,3,4,5,6,7,8,9,10,11,12)
MULTITYPED char(10)
```

And the following makes “January” and “Jan” as aliases of MONTH 1:

```
INSERT INTO DOMAIN MONTH VALUES ('January', 'Jan.')
WHERE VALUE=1
```

- **[CARDINALITY integer\_cardinality**

**[OCCURRENCE integer\_occurrence]]**

This specifies the estimated cardinality of the named domain (the approximate number of distinct values in the domain) and the estimated total number of occurrences of the values database-wide. This information is useful for the system in determining whether to use internal representation storage scheme for the domain if the technique is adopted. The following example domain provides a typical case for using internal representation. The values of domain COUNTRY are one hundred characters long. The cardinality of the domain is relatively low compared with the number of occurrences of the domain; the former is only one hundredth of the latter. Note as mentioned in Section 3.4, in most cases the DBA is responsible for determining whether to store the domain or not.

```
CREATE DOMAIN COUNTRY char(100)
CARDINALITY 200 OCCURRENCE 20000
```

- **[STORED]**

This indicates that the named domain should be physically stored in the database as an independent object. Once a domain is declared to be `STORED`, direct insertion of values into the domain becomes possible while indirect insertion of values via an ordinary relation-oriented insertion is also allowed. The `STORED` property will further help in avoiding “update anomalies”. For more detail see the discussion in previous chapter.

To couple with `CREATE DOMAIN`, SQL/D provides `DROP DOMAIN` to remove a domain from database. The statement is:

**DROP DOMAIN** *domain\_name*

Note that, before dropping a domain, it is necessary to drop all relations which are defined on the domains; otherwise the domain would not be dropped to prevent unexpected loss of data.

So far we have shown that using SQL/D, various types of domains, from a plain domain with only data type specification to a complex one with several properties, could be defined. Domain definition is the most primitive step in supporting domains in databases. We will see how domains affect relation definition and data manipulation in Section 4.4.

### 4.3.2 Defining Relations in SQL/D

The SQL/D relation definition statement is similar to the SQL counterpart. The only difference is that attributes are defined on user-created domains in SQL/D and in SQL columns are defined on built-in data type domains.

**CREATE TABLE** *relation\_name*

```
( {attribute_name [ON] domain_name [[NOT] NULL [UNIQUE]]},...
  [{UNIQUE ( {attribute_name },... )}]... )
```

Each attribute must be assigned a previously created domain. Because all domains have specified data type(s), it is no longer needed to assign data types to attributes. The option `[[NOT] NULL [UNIQUE]]` specifies whether the attribute is allowed to accept null and whether duplicated values are allowed in the attribute. Note that if the underlying domain does not allow null in the entire data base, then the NULL choice is unavailable here, because the decision at the domain level applies to the entire database. The keyword ON is optional and is used for ease of reading.

```
[UNIQUE {( {attribute_name },... )},...]
```

This option is used to specify candidate keys of the relation, especially composite candidate keys, since a single attribute candidate key could be specified by choosing the UNIQUE option in the attribute definition. In SQL/D, no explicit foreign key specification is available since it is not the best way to achieve referential integrity. In SQL/D referential integrity constraints are achieved by defining a derived domain on referenced attribute(s) and making it the underlying domain of the referencing attribute. We will see such an example in the comprehensive SQL/D database definition example at the end of this section.

Like relations in standard SQL, relations in SQL/D are expandable by adding attribute(s) with the ALTER statement. The added attribute(s) must be defined on previously created domains too. The amended ALTER syntax is:

```
ALTER TABLE relation_name ADD
( {attribute_name [ON] domain_name [UNIQUE]},... )
```



The new attribute could be UNIQUE but could not be NOT NULL, since in existing tuples the newly-added attribute will necessarily contain NULL.

#### 4.3.3 A Comprehensive Example

To summarize our discussion on SQL/D data definition facilities, we present the SQL/D version of the definition of the “Supplier-and-Parts” database below:

```
CREATE DOMAIN S# char(2) PICTURED 'S-';
CREATE DOMAIN SNAME char(10);
CREATE DOMAIN STATUS int;
CREATE DOMAIN CITY char(20) ENUMERATED
    ('LONDON','PARIS','ATHENS','ROME','OSLO');
CREATE DOMAIN P# char(2) PICTURED 'P-';
CREATE DOMAIN PNAME char(10);
CREATE DOMAIN PRICE int;
CREATE DOMAIN WEIGHT real MULTIUNIT
    DEFAULT='KG', 'LB'=2.2046, 'GRAM'=1000;
CREATE DOMAIN S_S# DERIVED AS SELECT S# FROM S;
CREATE DOMAIN P_P# DERIVED AS SELECT P# FROM P;
CREATE DOMAIN QTY int RANGED FROM 0 TO 1000;

CREATE TABLE S (
    S# ON S# NOT NULL UNIQUE,
```

```
SNAME ON SNAME,  
STATUS ON STATUS,  
CITY CITY );  
CREATE TABLE P (  
  P# ON P# NOT NULL UNIQUE,  
  PNAME ON PNAME,  
  PRICE ON PRICE,  
  WEIGHT ON WEIGHT,  
  CITY ON CITY );  
CREATE TABLE SP (  
  S# ON S_S# NOT NULL,  
  P# ON P_P# NOT NULL,  
  QTY ON QTY,  
  UNIQUE (S#,P#) );
```

Because most of the domain definitions in this example have been explained before and the relation definitions are straightforward, we will only discuss some necessary parts of the example below.

- `S# ON S# NOT NULL` specifies that attribute `S.S#` is defined on domain `S#` and null is not allowed in this attribute. The same applies to all the other attributes.
- The primary key of relation `S` is specified by indicating `UNIQUE` on the key attribute `S#`; this also applies to `P#` in `P`. But the primary key of relation `SP` is specified by a separate `UNIQUE (S#,P#)` clause.

- Attributes S.CITY and P.CITY are defined on the same domain, namely CITY. In the definition, the optional keyword ON is omitted for S.CITY.
- The derived domain S.S# sets up a reference from attribute SP.S# to attribute S.S#. The only legal values for SP.S# are the existing values in S.S#. And the same applies to domain P.P# for SP.P# and P.P#.

## 4.4 Data Manipulation in SQL/D

The SQL/D data manipulation facilities consist of two parts: ordinary relation-oriented manipulation and special domain-oriented manipulation. In this section, we will mainly discuss the latter part since the first part is similar to ordinary SQL. As in SQL, there are four data manipulation statements in SQL/D: SELECT, INSERT, DELETE and UPDATE. They will be discussed separately.

### 4.4.1 SELECT

The only SQL/D modification to the SQL SELECT statement is the new option:

SELECT VALUE FROM domain\_name

This is used to retrieve all values of the named domain. If the domain is an enumerated domain, then the system SELECTs all values from the corresponding system relation storing the enumerated values. If the domain is a derived domain then the system SELECTs all distinct values from the attribute(s) from which the domain is derived. If values of the domain are spread over several relations then the system SELECTs all distinct values from all the relations involved. The keyword VALUE here indicates that this is a domain-oriented SELECT. The following statement re-

trieves all values of domain CITY, which provides all the legal values for attributes S.CITY and P.CITY.

```
SELECT VALUE FROM CITY
```

When discussing data comparability problems, we suggested that “forced” inter-domain comparison should be allowed in domain support systems. In SQL/D a “forced” comparator is expressed by prefixing any ordinary comparator with an ‘@’ sign. For example, the following SELECT statement compares values in S.STATUS, which is defined on domain STATUS, with values in P.WEIGHT which is defined on another domain WEIGHT:

```
SELECT SNAME FROM S,P WHERE S.STATUS @= P.WEIGHT
```

The forced comparator can also be used in other data manipulations.

#### 4.4.2 INSERT

Domain-oriented insertion is used to insert new values into some domain which is explicitly stored in the database. The domain could be an enumerated domain, a multityped domain or simply a stored domain. To insert new values into an enumerated domain or a stored domain, the format is:

```
INSERT INTO [DOMAIN] domain_name VALUES ( {constant},... )
```

Here domain\_name specifies the domain into which the values are to be inserted and the inserted values are listed enclosed in parentheses. The format to insert aliases for a value of a multityped domain is:

```
INSERT INTO [DOMAIN] domain_name VALUES ( {constant},... )  
[WHERE VALUE=constant]
```

The aliases are listed enclosed in parentheses and the WHERE clause contains the

main value to which the aliases are to be inserted. The keyword DOMAIN indicates that this is a domain oriented insertion. If domain names are distinct from relation names, then this keyword could be omitted. We recommend that every domain name be distinct from every relation name in order to simplify the syntax of this and other data manipulation statements.

#### 4.4.3 DELETE

Domain oriented deletion is used to remove values from any explicitly stored domain and it takes the following format:

```
DELETE [CASCADE] FROM domain_name [WHERE VALUE=constant]
```

Here domain\_name indicates the domain from which values are to be removed. The CASCADE option indicates whether or not to delete indirectly any value of some other domain which is a derived domain from an attribute based on the named domain. The WHERE option specifies the particular value to be deleted. If the WHERE clause is omitted then the DELETE is interpreted to remove all values of the domain. If the domain is a multityped domain then any aliases of the specified value will be removed too.

Note that the CASCADE option can also be used in a relation-oriented deletion:

```
DELETE CASCADE FROM relation_name ...
```

This indicates that if there is any derived domain defined on any attribute of the named relation then the deletion will cascade to any relation whose attribute is defined on the derived domain. This is necessary to maintain referential integrity between the named relation and the relations which are deriving values from the named relation.

As indicated before, direct deletion of values from a domain should be strictly controlled to protect the database from being damaged because such a deletion may destroy the database.

#### 4.4.4 UPDATE

The domain-oriented SQL/D UPDATE is more complicated than other domain-oriented data manipulation statements. In SQL/D, UPDATE is not only used to update values of domains but, as indicated in the discussion of multiunit domain definition, it is also used to set one of the multiple units as the current unit of a multiunit domain. The format is:

```
UPDATE UNIT SET CURRENT = unit_name
```

```
WHERE DOMAIN = domain_name
```

This could be interpreted as “change CURRENT UNIT of DOMAIN *domain\_name* to *unit\_name*”. Readers may argue that the statement is a little too obscure and farfetched. But this is the result of our way to extend SQL language. We tried to add as few new statements or new keywords as possible, while at the same time trying to obey the syntax of the original SQL in our SQL/D extension. It follows that to be able to switch among multiple units of a multiunit domain we only add two new keywords: CURRENT and UNIT<sup>3</sup> (They are definitely needed in the above-shown interpretation, too) and the syntax of the UPDATE statement is not changed at all. In fact, in our SQL/D implementation, we do have a system relation to record the current unit of each multiunit domain. The structure of the relation is as below:

```
UNIT(DOMAIN,CURRENT)
```

---

<sup>3</sup>The keyword DOMAIN is already needed elsewhere.

so that the system could interpret and execute the UPDATE UNIT statement as an ordinary UPDATE statement. We think that such a trade-off between simplifying the language extension and the meaningfulness of the language is acceptable.

To update values of a domain which is not necessarily an explicitly stored domain, the format is:

```
UPDATE domain_name SET VALUE=expression [WHERE search_condition]
```

The keyword VALUE indicates that this is a domain-oriented update while the WHERE optional clause specifies which value or values are to be updated. If the WHERE clause is omitted, then all values of the named domain and hence all the occurrences of the values in the entire database will be updated. The following example UPDATE increases all PRICE values by ten percent.

```
UPDATE PRICE SET VALUE = VALUE * 1.1
```

The amended relation-oriented UPDATE may also include the CASCADE option with a similar effect as CASCADE in DELETE. Whenever a referenced attribute is updated and the CASCADE option is selected, all the referencing attributes are updated too. For example, the following update means supplier 'S6' takes over 'S1' as well all shipments supplied by 'S1'.

```
UPDATE S CASCADE SET S#='S6' where S#='S1'
```

The cascaded update is useful to simplify the effort to maintain referential integrity constraints.

#### 4.4.5 GRANT and REVOKE

Before ending our discussion on data manipulation in SQL/D, it is necessary to mention briefly the only data control operation in our SQL/D proposal, the extended

GRANT and REVOKE statement. Since domains are regarded as objects in SQL/D, it is natural to enforce privilege control on domains. The domain-level privilege control possibilities provided by SQL/D GRANT and REVOKE are very similar to those that SQL GRANT and REVOKE provide to relations. So it is not necessary to discuss them in more detail here.

In this chapter we have proposed SQL/D, the domain extension to the SQL language. In the next chapter we will describe our current implementation of SQL/D on the top of the SYBASE database manager.



## Chapter 5

---

# The Implementation of SQL/D

---

In this chapter we describe our current implementation of SQL/D on the SYBASE database management system. This implementation is an interactive SQL/D interface built on SYBASE. For simplicity, we will call the implementation ISQLD, since the interactive SQL interface of SYBASE is called ISQL.

In this chapter we will not show any results from running the ISQLD program, in order to keep the length manageable; instead we will present some results in the Appendix of the thesis.

### 5.1 Overview of SYBASE

Produced by Sybase Inc., Berkeley, California, SYBASE is one of the newest SQL-based mainframe RDBMSs. SYBASE is primarily designed for DEC VAX computers under the VMS and UNIX operating systems, and SUN computers under UNIX. Our ISQLD is written in the C language on the SUN version of SYBASE.

SYBASE is acknowledged [VaGa89] as the leading RDBMS for distributed on-line applications. It is also characterized by its ability to handle data integrity in the database itself rather than in each application. Like the other SQL-based systems, SYBASE provides both an interactive interface and an application programming interface to the SQL language, but the programming interface of SYBASE is distinctly

different from so-called “embedded SQL” of most other systems. First, while most RDBMSs embed SQL into a programming language via a precompiler, SYBASE employs a different approach by providing a unique programming language interface, namely DB-Library, which is a set of C routines and macros to manage the communication between any front-end process and the system. Second, in SYBASE there is no similar notation as so-called “cursors” of the other SQL-based systems, which provide some kind of bridge between the set-at-a-time level SQL language and the record-at-a-time level “host” language. SYBASE always keeps the set level result of a DB-Library routine in the system buffer and lets the application program fetch the result one-record-at-a-time with some special DB-Library routines, such as “dbnextrow”.

The TRANSACT-SQL language, SYBASE’s version of SQL, while providing standard SQL data definition and data manipulation facilities, does not support domains at all. Although SYBASE has some outstanding data integrity maintenance facilities, like rule, trigger, primary/foreign key, it still suffers from the domain-related problems discussed in previous chapters. For example, the following retrieval:

```
SELECT * FROM S,P WHERE S.STATUS > P.WEIGHT
```

will yield a nonsensical result in SYBASE, as described earlier.

The lack of domain support in SYBASE makes it an ideal candidate to implement our SQL/D proposal. In the following sections we will discuss the details of our ISQLD implementation.

## 5.2 Overview of ISQLD

The main purpose of ISQLD is to investigate the feasibility and ease of supporting domains, especially with the features proposed in our SQL/D, in relational databases. Since the time for this thesis research was limited, it was not practical to implement the entire SQL/D proposal. What we did is build an interactive SQL/D interface on top of SYBASE with part of the SQL/D facilities implemented. That means the domain features of some SQL/D commands are available only through the ISQLD interface. For example, if the foregoing retrieval is submitted from ISQLD then it will be rejected because of the inter-domain data comparison, but if it is issued via a SYBASE program then it will be executed because our current SQL/D implementation has no effect on SYBASE programs.

Simply speaking, our ISQLD is a SYBASE/C program which can read an SQL/D command from a terminal, interpret the command, reduce the command to one or more TRANSACT-SQL command(s) and finally pass the commands to SYBASE for execution. To achieve the SQL/D domain features, we did not use any special SYBASE facilities like rules and primary/foreign keys; instead we simply coded the features in C, since we believe this is a better way to meet our feasibility study goal. We did not modify the SYBASE source files because we were not allowed to do so.

To make our ISQLD more compatible with ISQL, the interactive TRANSACT-SQL interface on SYBASE, we also implemented some other non-database features of ISQL. Examples of the features include: the command to call a UNIX editor to edit the SQL command buffer<sup>1</sup>; the command to read SQL commands from an

---

<sup>1</sup>Unlike ORACLE and the other SQL-based systems, SYBASE does not provide a built-in editor. To edit the SQL command buffer, an editor must be called.

operating system file into the SQL command buffer; the command to clear or reset the SQL command buffer, the command to quit SYBASE, etc. Our ISQLD parser can also detect some syntactic/semantic errors. For example, you cannot CREATE a new domain with a name conflicting with the name of an existing domain.

Our ISQLD works in the same manner as ISQL except for some minor limitations on the ISQLD parser. For example, while the “joined” multiple-relation SELECT is implemented in ISQLD, a “nested” SELECT or a SELECT with “subquery” will not be interpreted as an SQL/D SELECT, because in most cases they are just different ways to express a data request from multiple relations and they make no difference to our goal: examining the impact of domains on the data requests from multiple relations. In ISQLD, to find the names of suppliers who supply part 'P1', the corresponding SQL/D command should be:

```
SELECT S.SNAME FROM S,SP
WHERE S.S# = SP.S# AND SP.P# = 'P1'
```

rather than a “nested” one like:

```
SELECT SNAME FROM S WHERE S# IN
(SELECT S# FROM SP WHERE P# = 'P1')
```

In the next two sections, we will describe the details of the ISQLD data definition facilities and data manipulation facilities respectively.

### 5.3 Data Definition in ISQLD

In ISQLD, database definition takes two steps: first you define domains and then you define the relations. A relation can be defined only after all the underlying

domains of the relation are defined, otherwise ISQLD will not accept the relation definition. In contrast, to drop (delete) a domain you have to drop all the relations with attribute defined on the domain first, and then you can drop the domain.

### 5.3.1 Types of Domains Implemented in ISQLD

ISQLD does not support all the domain types proposed in SQL/D. Currently we have only implemented the following:

- Plain domains. A plain domain is the simplest domain which takes any values that match the specified underlying data type for the domain. Usually values of a plain domain are not stored separately from relations unless “STORED” option is explicitly specified. Note: in ISQLD we do not store values for any plain domain because the option is not implemented.
- ENUMERATED domains. The legal values of an enumerated domain are either enumerated at the domain definition stage or are explicitly inserted into the domain via domain-oriented insertions. Each enumerated domain is physically stored in a separate system relation whose name is prefixed by “ED\_” meaning Enumerated Domain. Any attempt to insert new values into the domain via a relation-oriented insertion will be rejected, because the SQL/D optional “INSERTION” clause in derived domain definitions is not implemented.
- MULTIUNIT domains. The values of a multiunit domain can be represented in any of the pre-defined units when the data are input into the database or are output from the database. Although all the data values stored in the database are in the default unit of the domain, by changing the “current” unit

of the domain to any of the alternative units specified in the domain definition with the UPDATE UNIT command, the data values could be automatically converted back and forth between the default unit and the current unit. The conversion is transparent to the users and may take place whenever needed. Multiunit domains are not stored.

- **RANGED domains.** The values of a ranged domain are limited to within an interval and the interval is specified in the domain definition by a lower bound and an upper bound. Whenever a new value is to be inserted into any attributes based on the domain, it is automatically verified against the interval. Usually it is not necessary to physically store ranged domains separately from relations and we did not store any ranged domain in our implementation.
- **DERIVED domains.** This is the most important domain to be implemented in ISQLD. A derived domain is composed of data values from a specified attribute of a relation. The derived domain is used to specify and maintain the “referential” integrity constraint between relations. Since the values of a derived domain are kept in the referenced attribute of the referenced relation, it is therefore not necessary to store them again in any system relations.

We chose the above domain types for implementation because we believe they are either relatively hard to implement, like the MULTIUNIT domains, or more interesting to try, like the DERIVED domains. The rest of the domain types seem easier to implement or can be implemented in ways similar to the implementation of the above types.

### 5.3.2 Define Domains and Relations

In ISQLD, a domain is defined by the “CREATE DOMAIN” command. A domain can be assigned to one or several of the implemented domain types. Once a domain type is assigned to a domain the corresponding domain integrity constraints will be automatically maintained on the domain by ISQLD. When a “CREATE DOMAIN” command is executed, ISQLD will record the domain definition in the relevant system relation and, in some necessary cases, create a system relation to physically store the values of the domain. For example, a system relation will be created automatically for each enumerated domain to store its values.

The proposed “CREATE TABLE” SQL/D command, which is used to define relations on the pre-defined domains, is almost completely implemented. The only exception is the UNIQUE clause which is used to specify a candidate key of the relation. This is partly because the option itself is not available in SYBASE and partly because we do not need the concept of primary key to specify the critical “referential” integrity constraints in ISQLD. An ISQLD “CREATE TABLE” command is actually translated into an equivalent TRANSACT-SQL “CREATE TABLE” command and is executed to generate a new relation. The major modification from an ISQLD command to a TRANSACT-SQL command is that domain\_name in the ISQLD command is replaced by the data type of the corresponding domain, which could be found in the system relation “sysdomains”.

### 5.3.3 Drop Domains and Relations

The DROP DOMAIN and DROP TABLE commands are implemented in ISQLD as proposed in SQL/D with two things to note: First, a domain can be dropped only if all the relevant relations which have attributes defined on that domain are dropped beforehand. Second, if there is a derived domain acquiring values from an attribute of a relation, then the relation cannot be dropped because the derived domain needs the relation to survive in the database.

In this section, we outline the main data definition features of ISQLD. The data manipulation aspects of the implemented domain types will be discussed later.

## 5.4 The ISQLD Data Dictionary

To implement the proposed SQL/D domain features on top of SYBASE it is certainly necessary to have some new entries in the system catalogue (or the system data dictionary), but in order to keep the system simple, we must try to add as few system relations as possible to the data dictionary<sup>2</sup> and only maintain the necessary information in the relations. The following are the system relations created and used exclusively by ISQLD:

- **sysdomains** — In this relation some very basic information about each domain is recorded: the name of the domain and the data type of the domain. If the domain is a DERIVED domain then the data type is set to the name of the corresponding parent domain. And if the domain is a MULTITYPED one

---

<sup>2</sup>Actually, we could not add any relation to the SYBASE data dictionary. Instead we created some user relations and treat them as if they were system relations.



then the data type would be set to the main data type of the domain, though currently ISQLD does not support this type of domain. A new tuple is inserted into this relation when an SQL/D CREATE DOMAIN command is successfully executed.

- **sysattdom** — In this relation there are three attributes: name of relation, name of attribute and name of domain. Each tuple in this relation records an attribute of a relation and its underlying domain. Whenever a new relation is successfully created by an SQL/D CREATE TABLE command, each attribute of the relation will cause a tuple to be inserted into this system relation.
- **sysderived** — In this relation the information about each derived domain is recorded. The structure of the relation is similar to that of the relation “sysdomains”, but the data in the relation represent different information. Each tuple in this relation indicates from where (the attribute of the relation) the domain is deriving values. Each time a derived domain is successfully created a new tuple will be inserted into this relation showing the source of the derived domain.
- **sysunit** — In this relation the information about MULTIUNIT domains is kept. The three attributes in the relation represent the name of the domain, the name of each alternative unit of the domain, and the constant used in conversion of data values between the default unit and each of the multiple units. The default unit is identified by the special conversion constant: 1. In addition to the sysunit relation, there is another system relation relevant to

multiunit domains: **UNIT**, which is used for accommodating the format of the **UPDATE UNIT** command, the command to set the current unit of a multiunit domain.

- **sysranged** — In this relation there are three attributes to record the name of a ranged domain, the lower bound and the upper bound of the values in the domain. Each ranged domain has one tuple in this relation.
- System relations for enumerated domains — For each enumerated domain, a unary system relation will be created to store all the enumerated values of the domain. As mentioned before, the name of such a relation is the name of domain prefixed with string “ED\_”. For example, the system relation for enumerated domain **CITY** is named **ED\_CITY**. When a value is to be inserted into any attribute which is based on an enumerated domain, the value will be looked up in the corresponding system relation. The insertion will be executed only if the value exists in the system relation; otherwise the insertion will be rejected.

Our ISQLD program does not modify the structure of any SYBASE system relations. But the data in the relations might be affected by some SQL/D command executed in ISQLD. For example, a successful **CREATE TABLE** command will cause new tuples to be inserted into the system relation **sysobjects**, which contains information about relations of a SYBASE database.

## 5.5 Data Manipulation in ISQLD

The ways to manipulate data in ISQLD are quite different from those in a standard SQL-based system like SYBASE because, as proposed in SQL/D, it becomes possible to manipulate data via domains in ISQLD. Besides that, relation-oriented data manipulation in ISQLD is also enhanced with some new features. The following are the main data manipulation features implemented in ISQLD.

### 5.5.1 Restricted Data Comparison

Data comparisons are restricted to data from the same domain or data from two compatible domains. In ISQLD, if an SQL/D command contains a comparison of data from two different domains, then the command will not be accepted unless the two domains are compatible, which means the two domains must be a derived domain and its parent domain. This restriction is one of the basic requirements of domain support in relational databases. ISQLD checks every data comparison for any incompatibility on domains and rejects SQL/D commands which violate the restriction.

To make the system more flexible, we also implemented the “forced” inter-domain comparators proposed in SQL/D, so that if data comparisons between two incompatible domains are really needed, we can just prefix the corresponding comparison operator with a ‘@’ sign to make the data comparison a “forced” one and override the restriction. The following are four typical data comparisons and their fate in ISQLD:

S.CITY = P.CITY — Same domain, accepted.

S.S# = SP.S# — Compatible domains, accepted.

S.STATUS > P.WEIGHT — Incompatible domains, rejected.

S.STATUS @> P.WEIGHT — “Forced” comparison, accepted.

### 5.5.2 Domain-oriented Data Manipulation

In ISQLD, we implemented most of the domain-oriented data manipulation facilities proposed in SQL/D. Here are the major ones:

- INSERT new values into a domain. New values can be inserted directly into an ENUMERATED domain via an explicit domain-oriented INSERT command as proposed in SQL/D. The newly inserted value is actually inserted into the corresponding “ED\_”-prefixed system relation and may exist there regardless of whether there is any occurrence of the value in any relation.
- SELECT all values from a domain. Retrieval of all the (distinct) values of a domain is completely implemented for all domain types supported in ISQLD. If the values of the domain are physically stored in a separate system relation, then the retrieval will be carried out on the system relation and all the values in the system relation will be retrieved. If the values of the domain are not stored separately from relations but are spread out in a user relation, or even in several user relations, then the retrieval will be carried out on all the relations concerned. The result will be all the distinct values in the relation(s) which are in fact the current occurrences of some legal values of the domain rather than all the possible legal values of the domain. For example, the retrieval of all the values of domain QTY, though it might contain all the integers between 0 and

1000, will only result in the distinct values in attribute QTY of relation SP, which is the sole relation with an attribute defined on the domain. The actual result of the domain-oriented retrieval command “SELECT VALUES FROM QTY” would be the set of {100, 200, 300, 400} in our example database. And in contrast, the retrieval command “SELECT VALUES FROM CITY” will result in all the five distinct values of the enumerated domain even if, for example, the value 'BERLIN' has no occurrence in the relations.

- Domain integrity is well maintained in ISQLD. The various domain integrity constraints of the implemented domain types are properly maintained. For example, any attempt to insert into relation SP a new QTY value which exceeds the interval of the ranged domain QTY will be rejected. Whenever a database updating command, including UPDATE, INSERT and DELETE, is issued in ISQLD, the program will automatically check all the domains involved for any possible violations of the domain integrity constraints. If any potential integrity violation is detected, the command will be rejected. The so-called “referential integrity” is also well maintained in ISQLD, and we will discuss it in the next section.
- Switching among multiple units for a MULTIUNIT domain. With the proposed UPDATE UNIT command which deals with the MULTIUNIT domains, the current unit of a MULTIUNIT domain can be changed in ISQLD. Despite the fact that all data stored in the database are in the default unit, any data retrieved from or to be inserted into the database are represented in the current unit. As long as the current unit is not the default unit, ISQLD will automati-

cally convert every piece of data between the current unit and the default unit before displaying or storing the data.

### 5.5.3 Other Features

The following are some other domain-related data manipulation features which have been implemented in ISQLD.

- Interrogation of the system data dictionary for domain information. As mentioned before, information about domains is recorded in some specially created system relations. In ISQLD, this information can be retrieved by an ordinary `SELECT` command in the same manner as retrieving system information in other SQL-based systems.
- Referential Integrity in ISQLD. Although we had pointed out before that the so-called “referential integrity” problem is exactly a special case of the domain integrity problem on derived domains, we prefer to continue using the term “referential integrity” to address the problem for reasons of accuracy and clarity.

Referential integrity constraints can be well maintained in ISQLD provided the constraints are correctly specified. To specify a referential integrity constraint between a referencing attribute and its referenced counterpart, a derived domain on the referenced attribute must be created and the referencing attribute must be defined on the derived domain. Then ISQLD will take the responsibility of maintaining the integrity between the two attributes.

When a value is to be inserted into the referencing attribute, ISQLD will look the value up in the derived domain which in fact is the referenced attribute. Only if the value exists in the domain will the insertion be carried out; otherwise it will be rejected. Similarly, when a value is to be deleted from the referenced attribute, ISQLD will check if there is any occurrence of the value in the referencing attribute, and if there is no such occurrence then the deletion will be carried out, otherwise it will be rejected. There is one exception in the case of deletion: if the CASCADE option is included in the DELETE command then ISQLD will simply delete the value together with all its occurrences in both of the referencing and the referenced attributes without any referential checking because such a cascaded deletion will not result in violations of referential integrities.

We selected the above data manipulation facilities to implement in ISQLD because we believe that they are more essential than the other features in a domain-supported system. There are fewer features left for further implementation in the data manipulation aspect of SQL/D than in the data definition aspect.

## 5.6 Remarks

Although our ISQLD is not a complete domain-supported relational database system, and is even incomplete in regard to the SQL/D proposal, the result of the program is positive and encouraging. It is positive in that the proposed domain features are fulfilled in the program, and it is encouraging because the program and the associated data structures are not very complex. The experimental program convinces us that

full support of domains in relational database systems is achievable and with domains being supported, relational databases will become more usable and some long-time perplexing domain-related problems of relational database systems will be eventually solved.

In the Appendix of the thesis we will show some script files of the ISQLD program. Most of them have been edited to include comments and explanation. The files include:

- The contents of the ISQLD system relations and the example database.
- Create/drop domains and relations;
- Data comparability;
- Derived domains and “referential integrities”;
- Implemented domain types;
- Domain-oriented data manipulation;
- Typical error messages and other interface features;

It took approximate 500 man-hours to develop the entire ISQLD program and the size of the executable file of our current implementation is shown as:

```
-rwxr-xr-x  1 zhang      376832 May 15 02:57 isqld`
```

This project represents approximately 2000 lines of code including a few program comments. Here is the list of the source programs.



---

```
-rw-r--r--  1 zhang      1856 May 15 03:23 isqld.c
-rw-r--r--  1 zhang      7702 Feb  4 03:34 creation.c
-rw-r--r--  1 zhang      2340 Feb  4 02:05 deletion.c
-rw-r--r--  1 zhang      2171 Jan 16 15:05 drop.c
-rw-r--r--  1 zhang      7723 May 15 03:50 imp.h
-rw-r--r--  1 zhang      3975 Jan 26 23:52 insertion.c
-rw-r--r--  1 zhang     10427 Jan 28 20:52 selection.c
-rw-r--r--  1 zhang      1360 Jan 16 20:42 syb.h
-rw-r--r--  1 zhang      2033 Jan 27 00:09 update.c
```

The main program of our implementation of ISQLD is called “isqld.c” which includes two header files “imp.h” and “syb.h” that in turn contain most basic functions used in parsing SQL/D commands, especially the command editor and the functions emulating SYBASE ISQL user interface. The SQL/D data definition and data manipulation functions are contained in the following files:

- *creation.c*: the functions to create various types of domains and the functions to create relations on pre-defined domains; the functions to insert domain definition information into ISQLD system relations.
- *deletion.c*: the functions to delete tuples from relations while maintaining the referential integrity constraints, especially the “CASCADE” deletion of referencing attribute values from relations.
- *drop.c*: the functions to drop domains and the functions to drop relations.

- *insertion.c*: the functions to insert values into domains and the functions to insert tuples into relations; the functions to maintain domain integrity constraints for various types of domains, like DERIVED domains, RANGED domains; the function to convert values in the current unit to values in the default unit for MULTIUNIT domains.
- *selection.c*: the functions to select values from domains and the functions to select tuples from relations; functions to check data comparability and functions to conduct “forced” inter-domain comparisons; functions to convert values from the default unit to values in the current unit for MULTIUNIT domains.
- *update.c*: the functions to directly update values in domains.

The main program calls the functions contained in the above files according to the type of command typed in by the user and the corresponding result or error message is displayed on the screen. The functions to conduct ordinary SQL operations, like SELECTION from relations, are written to include domain features. The specially created data dictionary relations are used to get information about domains and their relationships with relations and attributes of relations.

## Chapter 6

---

# Conclusions

---

In this chapter we will review and summarize the contributions that this thesis made to the relational data model. As well we will suggest the possible directions of future research on the subject of supporting domains in relational database systems.

### 6.1 Summary and Conclusion

Theoretically, the domain concept is the most fundamental concept of the relational data model. In relational database systems, relations are just subsets of the Cartesian product of domains, attributes draw actual values from underlying domains, and the most data integrity constraints are related to domains. Almost all aspects of the data model involve domains, so that without domain support, a database system would not be fully relational.

Unfortunately, the significance of domains has, to a large extent, not been recognized in the database community and hence no existing relational database systems support the concept of domains. Therefore the systems inevitably suffer from many domain-related problems, especially those described in this thesis. Although some of the problems (only a few) had been studied before, the real causes of the problems had typically not been properly pinpointed: the lack of support for domains in relational databases.

In this thesis, we first examined the major domain-related problems in existing relational database systems of which none fully supports domains. The main problems discussed include: data comparability; domain-oriented data manipulation; domain update anomalies; domain integrity and “referential integrity”. To get rid of the problems in a natural way, we strongly recommend the support of domains in relational database systems. The main features of domain support have been described and SQL/D, a domain extension to the SQL language which enhances the SQL language with the domain features has been proposed. Whereas Date claimed that his recent paper on the subject of domains[Date90A] is a comprehensive answer to the question “What is a domain?”, we would suggest that this thesis is the first to answer the question “Why and how should domains be supported in relational databases?”. In addition to discovering and solving domain-related problems, this thesis has also clarified several misconceptions about domains and misconceptions about the relational data model. For example, the so-called “referential integrity” problem is identified in this thesis as a special case of domain integrity problem on derived domains.

With domains being supported in relational database systems, not only are the overall data structures of the systems changed, but so are the ways of handling the databases themselves. In a domain-supported system, domains are precisely defined and (at least some of them) are physically stored so that data manipulation via domains becomes possible. Also relations are properly built on their theoretic basis, domains, and are treated differently than their counterparts in ordinary databases. New facilities of domain-supported systems, from data definition to data manipula-

tion, make it reasonable to expect that the systems will have these properties:

- Easier to use. For example, the complicated “referential integrity” constraints become easier domain integrity constraints on derived domains. Neither foreign keys nor indices on key attributes are needed to maintain such integrity constraints. Database designers no longer need to establish indexes for the purpose of maintaining this type of data integrities.
- More reliable. Domains are precisely defined so that the correctness of data can be more easily guaranteed by the system. Database integrity is thus improved significantly and databases become more reliable. For example, semantically incorrect inter-domain comparisons will be prohibited by the system.
- More flexible. The physical existence of domains makes it possible to manipulate data via domains instead of only via relations as in ordinary systems. Domain-oriented data inquiries are carried out on the corresponding domains. It is no longer necessary to translate (if possible) the domain-oriented inquiries into relation-oriented data inquiries and carry them out on relations.
- More natural: Domain-related problems are solved naturally by explicit support for domains. For example, complex domain update anomalies no longer exist in domain-supported systems. The systems are more relational than “table” systems which do not really have underlying domains for the tables to acquire values.

Although currently only some of the proposed features of domain support are implemented in our experimental SQL/D interactive interface on the SYBASE database

manager, the running result is very positive.<sup>1</sup> All the implemented features confirm the expected results and the system resources used to support the features are relatively low. From our experience with the current implementation of SQL/D, we can predict that full support of domains in relational database systems is feasible and beneficial, and with domain support, a relational database system will be fully relational and the potential of the relational data model can be fulfilled.

It must be noted, however, that our implementation was not intended to be more than a prototype or feasibility study, and that much more work needs to be done before the success of our approach can be considered proven.

## 6.2 Future Research

There are some topics about domain support in relational database systems which are not mentioned or are not elaborated on in this thesis. Here are the most extensions that could be made to our proposal in order to consider these topics.

1. *The order of data values in domains.* It is obviously necessary to be able to specify the order of data values of a domain. While data values of some domains, like integers, are naturally ordered, data values in some other domains are not initially ordered. For example, there is no particular order for suppliers' names, except the lexical order of the character strings representing the names.

The order of data values is useful in comparisons of one value with another and

---

<sup>1</sup>The program was demonstrated to some of graduate and undergraduate students of the Department of Computer Science at The University of Calgary. They were impressed in the program's improvement over SYBASE. A major SQL vendor also became interested in the SQL/D proposal and our implementation.

might be needed in calculation of the “distance” between values.

2. *More built-in domains.* There are several “popular” domains which will be used in almost every database. Examples include the domain of month and the domain of person’s names. The data values of the domains likely have similar properties in every databases. Such domains should become built-in ones so that the users could avoid redundant defining of the domains.
3. *The ability to convert (or combine) values of one or more domains into value(s) of some other domain(s).* Examples include the ability to convert a value of domain CENTURY to 100 values of domain YEAR; to convert values of domain MONTH into values of domain CALENDAR\_YEAR and values of domain FISCAL\_YEAR; to yield a value of domain DISTANCE by multiplying a value of domain VELOCITY with a value of domain TIME, and so on.
4. *Composite (compound) domains.* It is perhaps not as widely recognized that the relational data model permits composite domains. For example, the domain DATE may have three components, say MONTH, DAY, and YEAR. There are many problems to be investigated about composite domains.
5. *More domain-oriented data manipulation facilities.* For example, the operations to split values of a composite domain into atomic values and to compare data values decomposed from composite domains, etc.

According to “The Committee for Advanced DBMS function”[CADF90], the next generation of database systems will not be of a brand new data model, for example the object-oriented database systems, but will be relational database systems *with*

*advanced function*. The relational database systems are neither finished nor perfect. The concept of domain is certainly one of the aspects that should be taken into account in functional enhancement of relational database systems.

Although as the title suggested, this thesis mainly discusses why and how to support domains in relational database systems, we must point out that the principles of domain support discussed in the thesis also apply to database systems of other data models, since databases of the other models are not free of domain-related problems. Even in an object-oriented database system [Alag89] domains should also be regarded as independent classes of objects which therefore could be handled in a similar manner as relations.

Let us conclude this thesis with the Chinese idiom “**Cast a brick to attract jade**”. We wish our introductory remarks about supporting domains in relational database systems, as a little brick to us, could draw more valuable opinions on the subject, as jade to database community, and could eventually bring the relational database systems to support domains.



---

# Appendix

---

fsb: sqld

```
*****
*                               *
*   Welcome to Interactive SQL/D on SYBASE   *
*                               *
*           Developed by Zhao Zhang           *
*           Supervised by Anton Colijn       *
*                               *
*****
```

=====

The example database with additional relation PART;  
The ISQLD system relations for implemented domain types.

=====

1> select \* from S

2> go

SNUM	SNAME	STATUS	CITY
S2	JONES	10	PARIS
S4	CLARK	20	LONDON
S1	SMITH	20	LONDON
S3	BLAKE	30	PARIS
S5	ADAMS	30	ATHENS

(5 rows affected)

1> select \* from P

2> go

PNUM	PNAME	COLOR	WEIGHT	CITY
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P5	CAM	BLUE	12	PARIS
P1	NUT	RED	12	LONDON
P4	SCREW	RED	14	LONDON
P6	COG	RED	19	LONDON

(6 rows affected)

1> select \* from SP

2> go

SNUM	PNUM	QTY
S2	P1	300
S2	P2	400
S4	P2	200
S4	P4	300
S4	P5	400
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S3	P2	200

(12 rows affected)

1> select \* from PART

2> go

PID	PNA	COL	WEI	LOC
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P5	CAM	BLUE	12	PARIS
P1	NUT	RED	12	LONDON
P4	SCREW	RED	14	LONDON
P6	COG	RED	19	LONDON

(6 rows affected)

1> sp\_help

2> go

Name	Owner	Object_type
currentunit	zhang	view
ranged	zhang	view
samedomain	zhang	view
ED_CITY	zhang	user table
P	zhang	user table
PART	zhang	user table
S	zhang	user table
SP	zhang	user table
UNIT	zhang	user table
sysattdom	zhang	user table
sysderived	zhang	user table
sysdomains	zhang	user table

sysranged	zhang	user table
sysunit	zhang	user table
sysalternates	dbo	system table
syscolumns	dbo	system table
syscomments	dbo	system table
sysdepends	dbo	system table
sysindexes	dbo	system table
syskeys	dbo	system table
syslogs	dbo	system table
sysobjects	dbo	system table
sysprocedures	dbo	system table
sysprotects	dbo	system table
syssegments	dbo	system table
systypes	dbo	system table
sysusers	dbo	system table

1> select \* from sysdomains

2> go

DOMAIN	DATATYPE
SNUM	char(4)
SNAME	char(6)
STATUS	smallint
WEIGHT	tinyint
PNUM	char(4)
PNAME	char(6)
COLOR	char(6)
QTY	smallint
SSNUM	SNUM
PPNUM	PNUM
CITY	char(20)

(11 rows affected)

1> select \* from sysattdom

2> go

ATT	REL	DOM	NUM
SNUM	S	SNUM	1
SNUM	SP	SSNUM	1
PNUM	P	PNUM	1
PNUM	SP	PPNUM	2
QTY	SP	QTY	3
STATUS	S	STATUS	3
CITY	S	CITY	4
CITY	P	CITY	5
WEIGHT	P	WEIGHT	4
COLOR	P	COLOR	3
PNAME	P	PNAME	2
SNAME	S	SNAME	2

PNA	PART	PNAME	2
PID	PART	PNUM	1
COL	PART	COLOR	3
WEI	PART	WEIGHT	4
LOC	PART	CITY	5

(17 rows affected)

1> select \* from sysranged

2> go

DOM	LOW	UP
QTY	0	1000

(1 rows affected)

1> select \* from sysderived

2> go

DOM	REL	ATT
SSNUM	S	SNUM
PPNUM	P	PNUM

(2 rows affected)

1> select \* from sysunit

2> go

DOM	UNIT	CON
WEIGHT	KG	1.000000
WEIGHT	LB	2.204600
WEIGHT	GRAM	1000.000000

(3 rows affected)

1> select \* from ED\_CITY

2> go

VALUE
LONDON
PARIS
ROME
ATHENS
BERLIN

(5 rows affected)

Domain-oriented data manipulation (retrieval & update)

```
1> select VALUES from CITY
```

```
2> go
```

```
VALUE
```

```
-----
```

```
LONDON
```

```
PARIS
```

```
ROME
```

```
ATHENS
```

```
BERLIN
```

```
(5 rows affected)
```

```
1> select VALUES from SNUM
```

```
2> go
```

```
VALUE
```

```
-----
```

```
S1
```

```
S2
```

```
S3
```

```
S4
```

```
S5
```

```
(5 rows affected)
```

```
1> select VALUES from SSNUM
```

```
2> go
```

```
VALUE
```

```
-----
```

```
S1
```

```
S2
```

```
S3
```

```
S4
```

```
S5
```

```
(5 rows affected)
```

```
1> select SNUM from S
```

```
go
```

```
2> SNUM
```

```
----
```

```
S2
```

```
S4
```

```
S1
```

```
S3
```

```
S5
```

```
(5 rows affected)
```

```
1> select distinct SNUM from SP
```

```
2> go
```

```
SNUM
```

```
----
```

S1  
S2  
S3  
S4

(4 rows affected)

1> select VALUES from QTY

2> go

VALUE

-----  
100  
200  
300  
400

(4 rows affected)

1> select QTY from SP

2> go

QTY

-----  
300  
400  
200  
300  
400  
300  
200  
400  
200  
100  
100  
200

(12 rows affected)

1> select VALUES from WEIGHT

2> go

VALUE

-----  
12  
14  
17  
19

(4 rows affected)

1> select \* from P

2> go

PNUM PNAME COLOR WEIGHT CITY

-----

```

P2  BOLT  GREEN    17 PARIS
P3  SCREW BLUE     17 ROME
P5  CAM   BLUE     12 PARIS
P1  NUT   RED      12 LONDON
P4  SCREW RED      14 LONDON
P6  COG   RED      19 LONDON

```

(6 rows affected)

```
1> update DOMAIN WEIGHT set VALUE = VALUE + 1
```

```
2> go
```

(12 rows affected)

```
1> select * from P
```

```
2> go
```

PNUM	PNAME	COLOR	WEIGHT	CITY
P2	BOLT	GREEN	18	PARIS
P3	SCREW	BLUE	18	ROME
P5	CAM	BLUE	13	PARIS
P1	NUT	RED	13	LONDON
P4	SCREW	RED	15	LONDON
P6	COG	RED	20	LONDON

(6 rows affected)

```
1> select * from PART
```

```
2> go
```

PID	PNA	COL	WEI	LOC
P2	BOLT	GREEN	18	PARIS
P3	SCREW	BLUE	18	ROME
P5	CAM	BLUE	13	PARIS
P1	NUT	RED	13	LONDON
P4	SCREW	RED	15	LONDON
P6	COG	RED	20	LONDON

(6 rows affected)

```
1> update DOMAIN WEIGHT set VALUE = VALUE - 1
```

```
2> go
```

(12 rows affected)

```
1> select * from P
```

```
2> go
```

PNUM	PNAME	COLOR	WEIGHT	CITY
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P5	CAM	BLUE	12	PARIS
P1	NUT	RED	12	LONDON
P4	SCREW	RED	14	LONDON
P6	COG	RED	19	LONDON

(6 rows affected)

1> select \* from PART

2> go

PID	PNA	COL	WEI	LOC
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P5	CAM	BLUE	12	PARIS
P1	NUT	RED	12	LONDON
P4	SCREW	RED	14	LONDON
P6	COG	RED	19	LONDON

(6 rows affected)

Define domains of the implemented types

d1: plain character string domain;

d2: derived domain from PART.LOC;

d3: ranged domain (integers from between 1 and 9);

d4: multiunit domain with units u1, u2 and u3;

d5: enumerated domains {'A', 'B', 'C', 'D'}.

1> create domain CITY char(4)

2> go

Cannot create domain 'CITY', because there already exists a domain 'CITY'.

1> create domain d1 char(4)

2> go

This domain d1 is created.

1> select \* from sysderived

2> go

DOM	REL	ATT
SSNUM	S	SNUM
PPNUM	P	PNUM

(2 rows affected)

1> create domain d2 as select LOC from PART

2> go

This DERIVED domain d2 is created.

1> select \* from sysderived

2> go

DOM	REL	ATT
SSNUM	S	SNUM



PPNUM	P	PNUM
d2	PART	LOC

(3 rows affected)

1> select value from d2

2> go

VALUE

-----

LONDON

PARIS

ROME

(3 rows affected)

1> select \* from sysranged

2> go

DOM	LOW	UP
-----	-----	-----
QTY	0	1000

(1 rows affected)

1> create domain d3 int ranged ( 1,9 )

2> go

This RANGED domain d3 is created.

1> select \* from sysranged

2> go

DOM	LOW	UP
-----	-----	-----
QTY	0	1000
d3	1	9

(2 rows affected)

1> select \* from sysunit

2> go

DOM	UNIT	CON
-----	-----	-----
WEIGHT	KG	1.000000
WEIGHT	LB	2.204600
WEIGHT	GRAM	1000.000000

(3 rows affected)

1> create domain d4 int multiunit ( 'u1',1,'u2',10 , 'u3',100 )

2> go

This MULTI-UNIT domain d4 is created.

1> select \* from sysunit

2> go

DOM	UNIT	CON
-----	-----	-----
WEIGHT	KG	1.000000

WEIGHT	LB	2.204600
WEIGHT	GRAM	1000.000000
d4	u1	1.000000
d4	u2	10.000000
d4	u3	100.000000

(6 rows affected)

1> create domain d5 char(5) enumerated ( 'A','B','C','D' )

2> go

This ENUMERATED domain d5 is created.

1> select \* from ED\_d5

2> go

VALUE

-----

A

B

C

D

(4 rows affected)

1> select \* from sysdomains

2> go

DOMAIN	DATATYPE
-----	
SNUM	char(4)
SNAME	char(6)
STATUS	smallint
WEIGHT	tinyint
PNUM	char(4)
PNAME	char(6)
COLOR	char(6)
QTY	smallint
SSNUM	SNUM
PPNUM	PNUM
CITY	char(20)
d1	char(4)
d2	CITY
d3	int
d4	int
d5	char(5)

(16 rows affected)

Define a relation (t) on the predefined domains;

Insert some tuples into the relation;

The effect of the domain integrity constraints of various domain types.

```
=====
```

```
1> create table t
```

```
2> ( a1 d1,
```

```
3> a2 d2,
```

```
4> a3 d3,
```

```
5> a4 d4,
```

```
6> a5 d5 )
```

```
7> go
```

```
Table 't' created.
```

```
1> select * from sysattdom
```

```
2> go
```

ATT	REL	DOM	NUM
SNUM	S	SNUM	1
SNUM	SP	SSNUM	1
PNUM	P	PNUM	1
PNUM	SP	PPNUM	2
QTY	SP	QTY	3
STATUS	S	STATUS	3
CITY	S	CITY	4
CITY	P	CITY	5
WEIGHT	P	WEIGHT	4
COLOR	P	COLOR	3
PNAME	P	PNAME	2
SNAME	S	SNAME	2
PNA	PART	PNAME	2
PID	PART	PNUM	1
COL	PART	COLOR	3
WEI	PART	WEIGHT	4
LOC	PART	CITY	5
a1	t	d1	1
a2	t	d2	2
a3	t	d3	3
a4	t	d4	4
a5	t	d5	5

```
(22 rows affected)
```

```
1> select * from t
```

```
2> go
```

a1	a2	a3	a4	a5
-----	-----	-----	-----	-----

```
(0 rows affected)
```

```
1> insert into t values ( 'R1','LONDON',5,5,'A' )
```

```
2> go
```

```
(1 rows affected)
```

```
1> select * from t
```

```

2> go
a1  a2              a3              a4              a5
-----
R1  LONDON              5              5 A

(1 rows affected)
1> insert into t values ( 'R2','ATHENS',20,10,'E' )
2> go
Insertion failed, value of t.a2 does not exist in referenced relation PART.
1> insert into t values ( 'R2','PARIS',20,10,'E' )
2> go
Insertion failed, value of t.a3 exceeds the range of domain d3.
1> insert into t values ( 'R2','PARIS',8,10,'E' )
2> go
Insertion failed, value of t.a5 does not exist in domain d5.
1> insert into t values ( 'R2','PARIS',8,10,'B' )
2> go
(1 rows affected)
1> select * from t
2> go
a1  a2              a3              a4              a5
-----
R1  LONDON              5              5 A
R2  PARIS              8              10 B

(2 rows affected)

=====

Multiunit domain

=====

1> update UNIT SET CURRENT = 'u2' where
2> DOMAIN = 'd4'
3> go
(1 rows affected)
1> select * from t
2> go
a1  a2              a3              a5
-----
R1  LONDON              5 A      a4 in u2      50.000000
R2  PARIS              8 B      a4 in u2      100.000000

(2 rows affected)
1> insert into t values ( 'R3','ROME',2,200,'C' )
2> go
(1 rows affected)
1> select * from t

```

```

2> go
a1  a2              a3              a5
-----
R1  LONDON              5 A      a4 in u2      50.000000
R2  PARIS              8 B      a4 in u2      100.000000
R3  ROME              2 C      a4 in u2      200.000000

```

(3 rows affected)

```
1> update UNIT set CURRENT = 'u1' where DOMAIN = 'd4'
```

```
2> go
```

(1 rows affected)

```
1> select * from t
```

```
2> go
```

```

a1  a2              a3              a4              a5
-----
R1  LONDON              5              5 A
R2  PARIS              8              10 B
R3  ROME              2              20 C

```

(3 rows affected)

```
1> select * from P
```

```
2> go
```

```

PNUM PNAME  COLOR  WEIGHT CITY
-----
P2  BOLT    GREEN   17  PARIS
P3  SCREW   BLUE    17  ROME
P5  CAM     BLUE    12  PARIS
P1  NUT     RED     12  LONDON
P4  SCREW   RED     14  LONDON
P6  COG     RED     19  LONDON

```

(6 rows affected)

```
1> update UNIT SET CURRENT = 'LB'
```

```
2> go
```

(1 rows affected)

```
1> select * from P
```

```
2> go
```

```

PNUM CITY          COLOR  PNAME
-----
P2  PARIS          GREEN  BOLT  WEIGHT in LB      37.478200
P3  ROME           BLUE   SCREW WEIGHT in LB      37.478200
P5  PARIS          BLUE   CAM   WEIGHT in LB      26.455200
P1  LONDON         RED    NUT   WEIGHT in LB      26.455200
P4  LONDON         RED    SCREW WEIGHT in LB      30.864400
P6  LONDON         RED    COG   WEIGHT in LB      41.887400

```

(6 rows affected)

```
1> update UNIT set CURRENT = 'KG'
```

```
2> go
(1 rows affected)
1> select * from P
```

```
2> go
PNUM PNAME  COLOR  WEIGHT CITY
-----
P2  BOLT    GREEN   17  PARIS
P3  SCREW   BLUE    17  ROME
P5  CAM     BLUE    12  PARIS
P1  NUT     RED     12  LONDON
P4  SCREW   RED     14  LONDON
P6  COG     RED     19  LONDON
```

```
(6 rows affected)
```

```
=====
Data comparability:
restricted and forced inter/intra domain comparisons.
```

```
=====
1> select * from S, P
2> where S.STATUS > P.WEIGHT
3> go
Attributes S.STATUS and P.WEIGHT are not in the same domain.
1> select * from S, SP
2> where SP.QTY > S.STATUS * 10
3> go
Attributes SP.QTY and S.STATUS are not in the same domain.
1> select * from P, SP
2> where SP.QTY > P.WEIGHT * 10
3> go
Attributes SP.QTY and P.WEIGHT are not in the same domain.
1> select * from P, SP
2> where SP.QTY @< P.WEIGHT * 10
3> go
```

PNUM	PNAME	COLOR	WEIGHT	CITY	SNUM	PNUM	QTY
P2	BOLT	GREEN	17	PARIS	S1	P5	100
P2	BOLT	GREEN	17	PARIS	S1	P6	100
P3	SCREW	BLUE	17	ROME	S1	P5	100
P3	SCREW	BLUE	17	ROME	S1	P6	100
P5	CAM	BLUE	12	PARIS	S1	P5	100
P5	CAM	BLUE	12	PARIS	S1	P6	100
P1	NUT	RED	12	LONDON	S1	P5	100
P1	NUT	RED	12	LONDON	S1	P6	100
P4	SCREW	RED	14	LONDON	S1	P5	100
P4	SCREW	RED	14	LONDON	S1	P6	100

```
P6 COG RED 19 LONDON S1 P5 100
P6 COG RED 19 LONDON S1 P6 100
```

(12 rows affected)

```
1> select * from S, P
2> where S.CITY = P.CITY
3> go
```

SNUM	SNAME	STATUS	CITY	PNUM	PNAME	COLOR	WEIGHT	CITY
S2	JONES	10	PARIS	P2	BOLT	GREEN	17	PARIS
S2	JONES	10	PARIS	P5	CAM	BLUE	12	PARIS
S4	CLARK	20	LONDON	P1	NUT	RED	12	LONDON
S4	CLARK	20	LONDON	P4	SCREW	RED	14	LONDON
S4	CLARK	20	LONDON	P6	COG	RED	19	LONDON
S1	SMITH	20	LONDON	P1	NUT	RED	12	LONDON
S1	SMITH	20	LONDON	P4	SCREW	RED	14	LONDON
S1	SMITH	20	LONDON	P6	COG	RED	19	LONDON
S3	BLAKE	30	PARIS	P2	BOLT	GREEN	17	PARIS
S3	BLAKE	30	PARIS	P5	CAM	BLUE	12	PARIS

(10 rows affected)

```
1> select * from S, PART
2> where S.CITY = PART.LOC
3> go
```

SNUM	SNAME	STATUS	CITY	PID	PNA	COL	WEI	LOC
S2	JONES	10	PARIS	P2	BOLT	GREEN	17	PARIS
S3	BLAKE	30	PARIS	P2	BOLT	GREEN	17	PARIS
S2	JONES	10	PARIS	P5	CAM	BLUE	12	PARIS
S3	BLAKE	30	PARIS	P5	CAM	BLUE	12	PARIS
S4	CLARK	20	LONDON	P1	NUT	RED	12	LONDON
S1	SMITH	20	LONDON	P1	NUT	RED	12	LONDON
S4	CLARK	20	LONDON	P4	SCREW	RED	14	LONDON
S1	SMITH	20	LONDON	P4	SCREW	RED	14	LONDON
S4	CLARK	20	LONDON	P6	COG	RED	19	LONDON
S1	SMITH	20	LONDON	P6	COG	RED	19	LONDON

(10 rows affected)

```
1> select * from S, SP
2> where S.SNUM = SP.SNUM
3> go
```

SNUM	SNAME	STATUS	CITY	SNUM	PNUM	QTY
S2	JONES	10	PARIS	S2	P1	300
S2	JONES	10	PARIS	S2	P2	400
S4	CLARK	20	LONDON	S4	P2	200
S4	CLARK	20	LONDON	S4	P4	300
S4	CLARK	20	LONDON	S4	P5	400

S1	SMITH	20	LONDON	S1	P1	300
S1	SMITH	20	LONDON	S1	P2	200
S1	SMITH	20	LONDON	S1	P3	400
S1	SMITH	20	LONDON	S1	P4	200
S1	SMITH	20	LONDON	S1	P5	100
S1	SMITH	20	LONDON	S1	P6	100
S3	BLAKE	30	PARIS	S3	P2	200

(12 rows affected)

```
1> select * from S, P
2> where S.CITY = P.CITY
3> and S.STATUS > P.WEIGHT
4>
5> go
```

Attributes S.STATUS and P.WEIGHT are not in the same domain.

```
1> select * from S, P
2> where S.CITY = P.CITY
3> and S.STATUS @> P.WEIGHT
4>
5> go
```

SNUM	SNAME	STATUS	CITY	PNUM	PNAME	COLOR	WEIGHT	CITY
S4	CLARK	20	LONDON	P1	NUT	RED	12	LONDON
S4	CLARK	20	LONDON	P4	SCREW	RED	14	LONDON
S4	CLARK	20	LONDON	P6	COG	RED	19	LONDON
S1	SMITH	20	LONDON	P1	NUT	RED	12	LONDON
S1	SMITH	20	LONDON	P4	SCREW	RED	14	LONDON
S1	SMITH	20	LONDON	P6	COG	RED	19	LONDON
S3	BLAKE	30	PARIS	P2	BOLT	GREEN	17	PARIS
S3	BLAKE	30	PARIS	P5	CAM	BLUE	12	PARIS

(8 rows affected)

Effects of various domain types on the example database.

Insert tuples into a relation;

Delete tuples from a relation;

Drop relations and domains;

```
1> insert into SP VALUES ('s0', 'p0', 10000 )
```

```
2> go
```

Insertion failed, SP.SNUM value does not exist in referenced relation S.

```
1> insert into SP VALUES ('S1', 'p0', 10000 )
```

```
2> go
```

Insertion failed, SP.PNUM value does not exist in referenced relation P.



```
1> insert into SP values ( 'S1', 'P1', 10000 )
2> go
Insertion failed, value of SP.QTY exceeds the range of domain.
1> insert into SP values ( 'S1', 'P1' ,-1 )
2> go
Insertion failed, value of SP.QTY exceeds the range of domain.
1> select * from SP
```

```
go
```

```
2> SNUM PNUM QTY
```

SNUM	PNUM	QTY
S2	P1	300
S2	P2	400
S4	P2	200
S4	P4	300
S4	P5	400
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S3	P2	200

```
(12 rows affected)
```

```
1> insert into SP values ( 'S1', 'P1', 999 )
```

```
2> go
```

```
(1 rows affected)
```

```
1> select * from SP
```

```
2> go
```

```
SNUM PNUM QTY
```

SNUM	PNUM	QTY
S2	P1	300
S2	P2	400
S4	P2	200
S4	P4	300
S4	P5	400
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S3	P2	200
S1	P1	999

```
(13 rows affected)
```

```
1> delete from P where PNUM = 'P1'
```

```
2> go
```

Deletion denied, because of referencial integrity violation.

1> delete cascade from P where PNUM = 'P1'

2> go

(4 rows affected)

1> select \* from P

2> go

PNUM	PNAME	COLOR	WEIGHT	CITY
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P5	CAM	BLUE	12	PARIS
P4	SCREW	RED	14	LONDON
P6	COG	RED	19	LONDON

(5 rows affected)

1> select \* from SP

2> go

SNUM	PNUM	QTY
S2	P2	400
S4	P2	200
S4	P4	300
S4	P5	400
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S3	P2	200

(10 rows affected)

1> delete from S

2> where CITY = 'LONDON'

3> go

Deletion denied, because of referencial integrity violation.

1> delete cascade from S

2> where CITY = 'LONDON'

3> go

(10 rows affected)

1> select \* from S

2> go

SNUM	SNAME	STATUS	CITY
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S5	ADAMS	30	ATHENS

(3 rows affected)

```
1> select * from SP
2> go
  SNUM PNUM QTY
-----
S2 . P2      400
S3  P2      200

(2 rows affected)
1> drop domain SNUM
2> go
Cannot drop domain 'SNUM', because there are relations defined on it.
1> drop DOMAIN QTY
2> go
Cannot drop domain 'QTY', because there are relations defined on it.
1> drop table SP
2> go
1> drop domain QTY
2> go
1> select VALUES from QTY
2> go
Domain QTY does not exists.
1> select * from SP
2> go
Msg 208, Level 16, State 1:
Line 1:
Invalid object name 'SP'.
1> drop DOMAIN WEIGHT
2> go
Cannot drop domain 'WEIGHT', because there are relations defined on it.
1> drop DOMAIN d1
2> go
Cannot drop domain 'd1', because it does not exist.
1> exit
fsb:
```

=====

For reasons of thesis length, we did not include the script file of running SYBASE ISQL in this appendix.

=====

---

# Bibliography

---

- [Alag89] Alagic, S. "Object-oriented Database Programming", Springer-Verlag, New York, New York, (1989).
- [Brad82] Bradley, J. "File and Data Base Techniques", Holt, Rinehart and Winston, New York, New York; (1982).
- [CADF90] The Committee for Advanced DBMS Functions "Third Generation Database System Manifesto", *SIGMOD Record* Vol. 19, No. 3, (Sept. 1990), 31-43.
- [Codd82] Codd, E. "Relational Database: A Practical Foundation for Productivity — The 1981 ACM Turing Award Lecture", *CACM*, Vol. 25, No. 2, (Feb. 1982), 109-117.
- [Codd70] Codd, E. "A Relational Model of Data for Large Shared Data Banks", *CACM* Vol. 13, No. 6, (June 1970), 377-387.
- [Croo91] Crooks, T. "Using ORACLE", QUE Corp., Carmel, Indiana, (1991).
- [Date91] Date, C. "An Introduction to Database Systems, Volume I", 5th Edition, Addison-Wesley, Reading, Mass., (1991).
- [Date90A] Date, C. "What is a Domain?", in Date, C., "*Relational Database Writings 1985-1989*", Addison-Wesley, Reading, Mass., (1990) 27-57.

- 
- [Date90B] Date, C. "Referential Integrity and Foreign Keys"; in Date, C., "*Relational Database Writings 1985-1989*", Reading, Mass., (1990) 99-169.
- [Date89] Date, C. "A Guide to the SQL Standard", Addison-Wesley, Reading, Mass., (1989).
- [Date87] Date, C. "A Guide to INGRES", Addison-Wesley, Reading, Mass., (1987).
- [Date86] Date, C. "An Introduction to Database Systems, Volume I", 4th Edition, Addison-Wesley, Reading, Mass., (1986).
- [Date83] Date, C. "An Introduction to Database Systems, Volume II", Addison-Wesley, Reading, Mass., (1983).
- [DaWh88] Date, C. and White, C. "A Guide to DB2", 2nd Edition, Addison-Wesley, Reading, Mass., (1988).
- [HeHe89] Heydt, R. & Heydt, D. "DB2 Database Design and Administration, Version 2", John Wiley & Sons, New York, (1989).
- [Koch89] Kocharekar, R. "Nulls in Relational Databases: Revisited", *ACM SIGMOD Record* Vol. 18 No. 1, (March 1989), 68-73.
- [Krug86] Kruglinski, D. "Data Base Management Systems, MS-DOS: Evaluating MS-DOS Data Base Software", Osborne McGraw-Hill, Berkeley, California, (1986).

- 
- [LaPi77] Lacroix, M. and Pirotte, A. "Domain-oriented Relational Languages", in *Proceedings of 3rd International Conference on Very Large Data Bases* (October 1977), 370-378.
- [McLe76] McLeod, D. J. "High Level Domain Definition in a Relational Data Base System", *ACM SIGPLAN Notices* Vol. 11 Special Issue, (1976), 47-52.
- [Mair83] Maier, D. "The Theory of Relational Databases", Computer Science Press, Rockville, Maryland, (1983).
- [OsHe86] Osborn, S. and Heaven T. "The Design of a Relational Database System with Abstract Data Types for Domains," *ACM TOD* Vol 11 No. 3, (Sept. 1986), 357-373.
- [Ullm88] Ullman, J. D. "Principles of Database and Knowledge-base Systems", Volumn 1, Computer Science Press, Rockville, Maryland, (1988).
- [VaGa89] Valduriez, P and Gardarin, G. "Analysis and Comparison of relational Database Systems", Addison Wesley, Reading, Matt. (1989).
- [YaCh88] Yannakoudakis, E. J. and Cheng, C. P. "Standard Relational and Network Database Languages", Springer-Verlag, Berlin, (1988).
- [Yang86] Yang, C. "Relational Databases", Prentice-Hall, Englewood Cliffs, New Jersey, (1986).