

世界著名计算机教材精选

数据库系统基础教程

[美] Jeffrey D. Ullman, Jennifer Widom 著

史嘉权 等译

清华大学出版社

<http://www.tup.tsinghua.edu.cn>

PRENTICE HALL

<http://www.prenhall.com>

(京)新登字 158 号

内 容 简 介

本书是由美国斯坦福大学两位著名的计算机学者 Jeffrey D. Ullman 和 Jennifer Widom 为初学数据库的人编写的基本教材。内容以对数据库的使用为主,讲述了数据建模,关系数据模型,SQL 语言以及面向对象数据库的查询语言 OQL 的基本概念。作者根据当前数据库领域的发展,对全书内容做了较大调整,删除了大量旧内容,增加了面向对象的新技术。本书内容简洁,概念清楚,适合作大学本科生学习数据库的参考书。

A First Course in Database Systems
Jeffrey D. Ullman, Jennifer Widom

Copyright C 1997 by Prentice Hall, Inc.
Original English Language Edition Published by Prentice Hall, Inc.
All Rights Reserved.

本书中文简体字版由 Prentice Hall 出版公司授权清华大学出版社独家出版、发行。未经出版者书面许可,不得以任何方式复制或抄袭本书的任何部分。
本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。
北京市版权局著作权合同登记号:图字 01-98-006 号

图书在版编目(CIP)数据

数据库系统基础教程/(美)额尔曼,(美)威多姆著;史嘉权译.—北京:清华大学出版社,1999. 8
(世界著名计算机教材精选)
ISBN 7-302-03646-2

. 数... . 额... 威... 史... . 数据库系统-教材 . TP311.13

中国版本图书馆 CIP 数据核字(1999)第 30397 号

出版者:清华大学出版社(北京清华大学学研大厦,邮编 100084)
<http://www.tup.tsinghua.edu.cn>

印刷者:清华大学印刷厂
发行者:新华书店总店北京发行所

开 本: 787x 1092 1/16 印张: 21.75 字数: 501 千字
版 次: 1999 年 9 月第 1 版 2001 年 11 月第 4 次印刷
书 号: ISBN 7-302-03646-2/TP · 2029
印 数: 16001 ~ 19000
定 价: 36.00 元

译者前言

数据库技术近年来发展非常迅速,特别是提出信息高速公路以来,所谓“3C”即计算机、通信和信息内容(Computer、Communication 和 Contents)已成为信息技术的核心。而信息高速公路的价值正体现在信息内容上,若没有大量的数据库存放这些“内容”并提供迅速、简便、高效的查询手段,则信息高速公路就只能“跑空车”了。

面向对象的数据库技术是近年来数据库技术发展的重要方向和热点,目前国内在该领域的科研方面已在积极开展,但在教材中尚无反映,因此急需有关的教材,而本书正是雪中送炭。

本书是从斯坦福大学 1997 年的教材《数据库系统基础教程》(《A First Course in Database Systems》)翻译过来的。本书以当前的主流数据库——关系数据库——为基础,以数据库系统的最基本内容——数据库的设计与编程——为重点,以引进数据库领域的最新成果——比如面向对象的数据库技术——为特点,系统地阐述了数据库建模、关系数据库的理论和设计、结构化查询语言 SQL 及其最新的标准 SQL2 和 SQL3,阐述了递归查询等最新内容,特别是以相当多的篇幅阐述了面向对象数据库的对象定义语言 ODL 和对象查询语言 OQL。

本书的主要特点是新颖、丰富、系统、实用,把数据库技术的最新成果迅速反映到教材中。斯坦福大学是世界一流大学,世界著名的硅谷与斯坦福有不解之缘。我们及时引进国外的最新教材,对提高国内的计算机教学和科研水平会起到积极的推动作用。

为培养同学直接从英文资料获取信息的能力,清华大学出版社已于 98 年出版了本书英文原著的影印本,现在又出版该书的中译本,这样可使不同程度的读者都能从中有所收益。读者不仅可以从中学到最新的专业知识,也能从中提高英文的专业阅读能力。

本书的翻译得到了我系周立柱教授的大力支持,在此表示衷心的感谢!在本书的翻译过程中,王霞、张勇、张劲飞和武志光同学为初稿的翻译和文稿的录入协助做了很多工作。本书的译文难免有不妥之处,敬请读者予以指正。

目 录



第 1 章	数据库系统的世界	1
1. 1	数据库系统的发展	1
1. 1. 1	早期的数据库管理系统	1
1. 1. 2	关系数据库系统	3
1. 1. 3	越来越小的系统	4
1. 1. 4	越来越大的系统	4
1. 2	数据库管理系统的结构	5
1. 2. 1	DBMS 的组成概述	5
1. 2. 2	存储管理程序	7
1. 2. 3	查询处理程序	7
1. 2. 4	事务管理程序	8
1. 2. 5	客户程序-服务程序体系结构	10
1. 3	未来的数据库系统	10
1. 3. 1	类型、类和对象	10
1. 3. 2	约束和触发程序	13
1. 3. 3	多媒体数据	13
1. 3. 4	数据集成	14
1. 4	本书概要	15
1. 4. 1	设计	15
1. 4. 2	编程	15
1. 5	本章总结	16
1. 6	本章参考文献	17
第 2 章	数据库建模	18
2. 1	ODL 介绍	18
2. 1. 1	面向对象的设计	19
2. 1. 2	接口说明	20
2. 1. 3	ODL 中的属性	20
2. 1. 4	ODL 中的联系	22
2. 1. 5	反向联系	22
2. 1. 6	联系的多重性	24
2. 1. 7	ODL 中的类型	26
2. 1. 8	本节练习	27
2. 2	实体联系图	29

2. 2. 1	E/R 联系的多重性	30
2. 2. 2	联系的多向性	30
2. 2. 3	联系中的角色	31
2. 2. 4	联系中的属性	32
2. 2. 5	把多向联系转换成二元联系	33
2. 2. 6	本节练习	34
2. 3	设计原则	35
2. 3. 1	真实性	36
2. 3. 2	避免冗余	36
2. 3. 3	对简单性的考虑	36
2. 3. 4	选择合适的元素类型	37
2. 3. 5	本节练习	38
2. 4	子类	40
2. 4. 1	ODL 中的子类	40
2. 4. 2	在 ODL 中的多重继承	40
2. 4. 3	实体联系图中的子类	42
2. 4. 4	E/R 模型中的继承	42
2. 4. 5	本节练习	43
2. 5	对约束的建模	44
2. 5. 1	键码	45
2. 5. 2	在 ODL 中说明键码	46
2. 5. 3	在 E/R 模型中表示键码	47
2. 5. 4	单值约束	47
2. 5. 5	参照完整性	48
2. 5. 6	E/R 图中的参照完整性	48
2. 5. 7	其他类型的约束	49
2. 5. 8	本节练习	49
2. 6	弱实体集	50
2. 6. 1	产生弱实体集的原因	50
2. 6. 2	对弱实体集的要求	52
2. 6. 3	弱实体集的表达法	52
2. 6. 4	本节练习	53
2. 7	历史上有影响的模型	53
2. 7. 1	网状模型	53
2. 7. 2	网状模式的表示	54
2. 7. 3	层次模型	55
2. 7. 4	本节练习	56
2. 8	本章总结	56
2. 9	本章参考文献	57

第 3 章	关系数据模型	58
3. 1	关系模型的基本概念	58

3.1.1	属性	59
3.1.2	模式	59
3.1.3	元组	59
3.1.4	域	60
3.1.5	关系的等价表示法	60
3.1.6	关系实例	61
3.1.7	本节练习	62
3.2	从 ODL 设计到关系设计	62
3.2.1	从 ODL 属性到关系属性	63
3.2.2	类中的非原子属性	63
3.2.3	其他类型构造符的表示	66
3.2.4	单值联系的表示	67
3.2.5	多值联系的表示	68
3.2.6	假如没有键码	69
3.2.7	联系与反向联系的表示	70
3.2.8	本节练习	71
3.3	从 E/R 图到关系的设计	72
3.3.1	实体集到关系的转换	72
3.3.2	E/R 联系到关系的转换	73
3.3.3	处理弱实体集	75
3.3.4	本节练习	77
3.4	子类结构到关系的转换	78
3.4.1	用关系表示 ODL 子类	78
3.4.2	在关系模型中表示“属于”联系	79
3.4.3	方法的比较	80
3.4.4	使用 NULL 值合并关系	80
3.4.5	本节练习	81
3.5	函数依赖	82
3.5.1	函数依赖的定义	82
3.5.2	关系的键码	83
3.5.3	超键码	84
3.5.4	寻找关系的键码	85
3.5.5	由 ODL 设计导出的关系的键码	86
3.5.6	本节练习	87
3.6	函数依赖规则	88
3.6.1	分解/合并规则	88
3.6.2	平凡依赖	89
3.6.3	计算属性的闭包	90
3.6.4	传递规则	92
3.6.5	函数依赖的闭包	93
3.6.6	本节练习	94
3.7	关系数据库模式设计	95

3.7.1	异常	96
3.7.2	关系分解	96
3.7.3	BC 范式	98
3.7.4	分解成 BCNF	99
3.7.5	函数依赖的投影	102
3.7.6	从分解中恢复信息	103
3.7.7	第三范式	105
3.7.8	本节练习	107
3.8	多值依赖	108
3.8.1	属性的独立性及其带来的冗余	108
3.8.2	多值依赖的定义	109
3.8.3	多值依赖的推论	111
3.8.4	第四范式	112
3.8.5	分解成第四范式	113
3.8.6	范式间的联系	114
3.8.7	本节练习	114
3.9	数据库模式实例	116
3.10	本章总结	118
3.11	本章参考文献	119
第 4 章	关系模型中的运算	121
4.1	关系代数	121
4.1.1	关系的集合运算	122
4.1.2	投影	123
4.1.3	选择	124
4.1.4	笛卡尔积	124
4.1.5	自然连接	125
4.1.6	连接	127
4.1.7	查询中的复合运算	128
4.1.8	改名	129
4.1.9	基本和导出运算	130
4.1.10	本节练习	131
4.2	关系的逻辑	136
4.2.1	谓词和原子	136
4.2.2	算术原子	137
4.2.3	Datalog 规则和查询	137
4.2.4	Datalog 规则的含义	138
4.2.5	外延和内涵谓词	140
4.2.6	本节练习	140
4.3	从关系代数到 Datalog	140
4.3.1	交集	141
4.3.2	并集	141

4.3.3	差集	141
4.3.4	投影	142
4.3.5	选择	142
4.3.6	乘积	144
4.3.7	连接	144
4.3.8	用 Datalog 模拟多重运算	145
4.3.9	本节练习	146
4.4	Datalog 中的递归编程	147
4.4.1	固定点运算符	147
4.4.2	计算最小固定点	148
4.4.3	Datalog 中的固定点方程	149
4.4.4	递归规则中的求反	153
4.4.5	本节练习	156
4.5	对关系的约束	157
4.5.1	用关系代数作为约束语言	158
4.5.2	参照完整性约束	158
4.5.3	附加约束的例子	159
4.5.4	本节练习	160
4.6	包的关系运算	161
4.6.1	为什么用包?	162
4.6.2	包的并集、交集和差集	163
4.6.3	包的投影	164
4.6.4	包的选择	165
4.6.5	包的乘积	165
4.6.6	包的连接	166
4.6.7	包的运算用于 Datalog 规则	167
4.6.8	本节练习	168
4.7	关系模型的其他外延	169
4.7.1	更新	169
4.7.2	聚合	169
4.7.3	视图	169
4.7.4	空值	170
4.8	本章总结	170
4.9	本章参考文献	171

第 5 章	数据库语言 SQL	172
5.1	SQL 的简单查询	172
5.1.1	SQL 的投影	173
5.1.2	SQL 的选择	175
5.1.3	字符串的比较	176
5.1.4	日期和时间的比较	178
5.1.5	输出的排序	178

5.1.6	本节练习	179
5.2	涉及多个关系的查询	180
5.2.1	SQL 中的乘积和连接	180
5.2.2	消除属性的二义性	181
5.2.3	元组变量	182
5.2.4	多关系查询的解释	183
5.2.5	查询的并、交、差	185
5.2.6	本节练习	186
5.3	子查询	188
5.3.1	产生标量值的子查询	188
5.3.2	涉及到关系的条件	189
5.3.3	涉及到元组的条件	190
5.3.4	相关子查询	191
5.3.5	本节练习	192
5.4	副本	193
5.4.1	副本的删除	193
5.4.2	并、交、差中的副本	194
5.4.3	本节练习	195
5.5	聚合	195
5.5.1	聚合运算符	195
5.5.2	分组	196
5.5.3	HAVING 子句	198
5.5.4	本节练习	199
5.6	数据库更新	200
5.6.1	插入	200
5.6.2	删除	202
5.6.3	修改	203
5.6.4	本节练习	203
5.7	用 SQL 定义关系模式	204
5.7.1	数据类型	205
5.7.2	表的简单说明	205
5.7.3	删除表	206
5.7.4	更改关系模式	206
5.7.5	默认值	207
5.7.6	域	207
5.7.7	索引	208
5.7.8	本节练习	209
5.8	视图的定义	211
5.8.1	视图的说明	211
5.8.2	视图的查询	212
5.8.3	属性改名	213
5.8.4	视图的更新	213

5. 8. 5	对涉及到视图的查询的解释	216
5. 8. 6	本节练习	217
5. 9	空值和外部连接	218
5. 9. 1	对空值的运算	218
5. 9. 2	真值 UNKNOWN	219
5. 9. 3	SQL2 中的连接表达式	221
5. 9. 4	自然连接	222
5. 9. 5	外部连接	222
5. 9. 6	本节练习	224
5. 10	SQL3 中的递归	225
5. 10. 1	在 SQL3 中定义 IDB 关系	225
5. 10. 2	线性递归	228
5. 10. 3	在 WITH 语句中使用视图	228
5. 10. 4	分层求反	229
5. 10. 5	SQL3 递归中的未定表达式	230
5. 10. 6	本节练习	232
5. 11	本章总结	233
5. 12	本章参考文献	235

第 6 章	SQL 中的约束和触发程序	236
6. 1	SQL 中的键码	236
6. 1. 1	说明键码	236
6. 1. 2	实施键码约束	238
6. 1. 3	本节练习	238
6. 2	参照完整性和外键码	238
6. 2. 1	说明外键码约束	239
6. 2. 2	保持参照完整性	240
6. 2. 3	本节练习	241
6. 3	对属性值的约束	243
6. 3. 1	非空约束	243
6. 3. 2	基于属性的 CHECK 约束	243
6. 3. 3	域约束	244
6. 3. 4	本节练习	245
6. 4	全局约束	246
6. 4. 1	基于元组的 CHECK 约束	246
6. 4. 2	断言	247
6. 4. 3	本节练习	250
6. 5	约束的更新	251
6. 5. 1	对约束命名	251
6. 5. 2	更改表的约束	252
6. 5. 3	更改域的约束	253
6. 5. 4	更改断言	253

6. 5. 5	本节练习	253
6. 6	SQL3 中的触发程序	254
6. 6. 1	触发和约束	254
6. 6. 2	SQL3 触发程序	254
6. 6. 3	SQL3 的断言	257
6. 6. 4	本节练习	258
6. 7	本章总结	259
6. 8	本章参考文献	260
第 7 章	SQL 系统概况	261
7. 1	编程环境中的 SQL	261
7. 1. 1	匹配失衡问题	262
7. 1. 2	SQL/ 宿主语言接口	262
7. 1. 3	说明(DECLARE) 段	263
7. 1. 4	使用共享变量	263
7. 1. 5	单行查询语句	264
7. 1. 6	游标	265
7. 1. 7	通过游标的更新	267
7. 1. 8	游标选项	267
7. 1. 9	为取出的元组排序	268
7. 1. 10	防止并发更新的保护措施	269
7. 1. 11	滚动游标	269
7. 1. 12	动态 SQL	270
7. 1. 13	本节练习	271
7. 2	SQL 中的事务	273
7. 2. 1	可串行性	273
7. 2. 2	原子性	275
7. 2. 3	事务	276
7. 2. 4	只读事务	277
7. 2. 5	读脏数据	278
7. 2. 6	其他隔离性级别	279
7. 2. 7	本节练习	280
7. 3	SQL 环境	281
7. 3. 1	环境	281
7. 3. 2	模式	282
7. 3. 3	目录	283
7. 3. 4	SQL 环境中的客户程序和服务程序	284
7. 3. 5	连接	284
7. 3. 6	会话	285
7. 3. 7	模块	285
7. 4	SQL2 的安全和用户授权	286
7. 4. 1	权限	286

7. 4. 2	建立权限	287
7. 4. 3	权限检验处理	287
7. 4. 4	授予权限	289
7. 4. 5	授权图	290
7. 4. 6	取消权限	290
7. 4. 7	本节练习	293
7. 5	本章总结	294
7. 6	本章参考文献	296
第 8 章	面向对象查询语言	297
8. 1	ODL 中相关查询的特性	297
8. 1. 1	ODL 对象的操作	297
8. 1. 2	ODL 中方法署名的说明	298
8. 1. 3	类的范围	300
8. 1. 4	本节练习	300
8. 2	OQL 介绍	301
8. 2. 1	面向对象的电影实例	302
8. 2. 2	OQL 类型系统	302
8. 2. 3	路径表达式	303
8. 2. 4	OQL 中的 select-from-where 表达式	304
8. 2. 5	消除重复	305
8. 2. 6	复杂的输出类型	305
8. 2. 7	子查询	306
8. 2. 8	对结果排序	307
8. 2. 9	本节练习	307
8. 3	OQL 表达式的附加格式	308
8. 3. 1	量词表达式	308
8. 3. 2	聚合表达式	308
8. 3. 3	分组表达式	309
8. 3. 4	HAVING 子句	311
8. 3. 5	集合运算符	311
8. 3. 6	本节练习	312
8. 4	OQL 中对象的赋值和建立	313
8. 4. 1	对宿主语言变量赋值	313
8. 4. 2	从聚集中提取元素	313
8. 4. 3	获取聚集的每个成员	314
8. 4. 4	建立新对象	314
8. 4. 5	本节练习	316
8. 5	SQL3 中的元组对象	316
8. 5. 1	行类型	317
8. 5. 2	说明具有行类型的关系	317
8. 5. 3	访问行类型的分量	318

8. 5. 4	引用	318
8. 5. 5	利用引用	320
8. 5. 6	引用的作用域	320
8. 5. 7	作为值的对象标识	321
8. 5. 8	本节练习	323
8. 6	SQL3 的抽象数据类型	324
8. 6. 1	ADT 的定义	324
8. 6. 2	ADT 方法的定义	327
8. 6. 3	外部函数	329
8. 6. 4	本节练习	329
8. 7	ODL/ OQL 和 SQL3 方法的比较	330
8. 8	本章总结	331
8. 9	本章参考文献	332

前 言



本书是从斯坦福大学的“数据库入门”(CS145)的课程笔记演变而来的。CS145 是五门系列课程的第一门。由于 Arthur Keller 颇有创意的讲授,使这门课逐步发展成着重于数据库的设计和编程的课程,而这两方面内容对计算机科学专业的大多数学生来说是最有用的。这门课还包括一个内容广泛、不断滚动的课外工程项目,供学生设计并实现一个具体的数据库应用。与该工程项目相关的作业、其他课外作业、测验以及其他课程资料都可以从本书的主页上得到;请参阅“万维网(World Wide Web)上的支持”部分。

本书的使用

本书适用于讲授一学期的课程。如果像 CS145 这样按四分之一学年(译注:每学年分为四学期,这种制度的一学期)的课程来安排,就不得不省略或跳过书中的某些内容。最好由教师自己决定削减哪些内容,但以下内容显然是可以削减的:有关 Datalog 的部分,SQL 编程的高级部分以及 SQL3 的细节部分。

如果课程中安排了不断滚动的工程项目,那么,提早讲授 SQL 语句部分是很重要的。可以推后讲授的内容包括:有关 Datalog 的部分,第 5 章和第 6 章的 SQL3 部分以及第 3 章的某些理论部分(但是,如果学生们在开始进行 SQL 编程之前,想设计出优秀的关系模式的话,他们就需要规范化的知识,或许还需要多值依赖的知识)。

预备知识

我们把本书定位于“夹层”水平,即高年级本科生和低年级研究生水平。这门课程正规的预备知识相当于大学二年级的水平:(1) 数据结构、算法和离散数学;(2) 软件系统、软件工程和编程语言。学生们对以下内容至少要有初步的了解:代数的表达式和定律、逻辑、基本数据结构(如搜索树)、面向对象的编程概念以及编程环境。我们相信,按照典型的计算机科学专业的教学计划,到大学三年级结束时,学生们肯定会拥有充分的背景知识。

练 习

本书包括多方面的练习,几乎每节都有。我们把比较难的练习或练习中比较难的部分

后面的四门是:数据库系统原理,数据库系统实现的工程训练,事务和分布式数据库,以及数据库理论。

用惊叹号(!)标出。最难的练习用双惊叹号(!!)标出。

有一些练习或练习的某些部分标有星号(*)。对于这些练习,我们将尽量通过本书的主页提供解答。这些解答是公开的,并可用于自我检测。注意:在某些情况下,练习 B 要求您对另一个练习 A 的解答进行修改或改进。如果 A 的某些特定部分有解答,那么 B 的相应部分也将有解答。

万维网上的支持

本书的主页是:

<http://www-db.stanford.edu/~ullman/fcdb.html>

这里有带星号的练习的解答,对已发现的书写或印刷错误的勘误表以及辅助教材。我们希望每一个像我们一样讲授 CS145 课程的人都能获得这些课程笔记,包括课外作业、解答和工程项目的作业。

致 谢

特别感谢 Bobbie Cochrane 和 Linda DeMichiel,感谢他们在 SQL3 标准方面给予的帮助。还有其他许多人帮助我们审校手稿,他们是: Donald Aingworth, Jonathan Becker, Larry Bonham, Christopher Chan, Oliver Duschka, Greg Fichtenholtz, Bart Fisher, Meredith Goldsmith, Steve Huntsberry, Leonard Jacobson, Thulasiraman Jeyaraman, dwight Joe, Seth Katz, Brian Kulman, Le-Wei Mo, Mark Mortensen, Ramprakash Narayanaswami, Torbjorn Norbye, Mehul Patel, Catherine Tornabene, Jonathan Ullman, Mayank Upadhyay, Vassilis Vassalos, Qiang Wang, Sundar Yamunachari 和 Takeshi Yokukawa。当然,剩下的错误由我们负责。

J. D. U.

J. W.

第 1 章 数据库系统的世界

通过本书,读者可以学会如何有效地使用数据库管理系统,包括数据库的设计和对数据库操作的编程。本章主要介绍各种重要的数据库概念。简短的历史回顾之后,我们将了解数据库系统与其他软件风格的区别。在这一章里,还将介绍支持数据库及其应用的数据库管理系统的实现背景。如果我们要正确评价为什么要设计各种各样的数据库、为什么要对数据库的操作有所限制,对其“内幕”的理解是非常重要的。最后,我们将回顾一些读者可能比较熟悉,但对后续章节而言又必不可少的思想,比如“面向对象的程序设计”等。

1.1 数据库系统的发展

数据库是什么呢?实质上,数据库只不过是一些存在了很长时间——常常是许多年——的信息的聚集。通常意义下,“数据库”这个术语是指由数据库管理系统(database management system, 简称为 DBMS, 或称为数据库系统)管理的数据聚集。一个数据库系统应该是:

1. 允许用户用一种叫做数据定义语言(data definition language)的专用语言,建立新的数据库和指定它们的模式(schema)(数据的逻辑结构)。
2. 使用户能够用适当的语言查询数据(“查询”(query)是一个数据库术语,指对数据的某种询问)和更新数据,所使用的语言通常称为“查询语言”或“数据操作语言”(data manipulation language)。
3. 支持存储大量的数据——G(吉, 10^9) 字节以上——经过很长一段时间后,仍保证安全,使其免遭意外或非授权的使用,同时允许对数据库查询和更新的有效访问。
4. 控制多用户的同时访问,使得一个用户的访问不影响其他用户,保证同时访问不会损坏数据。

1.1.1 早期的数据库管理系统

第一批商用数据库管理系统出现在 20 世纪 60 年代后期。它们由文件系统演变而来。文件系统能满足以上第 3 项的要求——长时间地存储数据,并且能存储大量的数据。但是,如果没有备份的话,文件系统通常并不能保证数据不丢失;如果不知道数据所在的特定文件,文件系统也不能支持有效的数据访问。

另外,文件系统并不能直接满足第 2 项关于查询语言对数据的查询要求。它们对第 1 项——数据模式的支持也仅限于建立文件的目录结构。最后,文件系统不支持第 4 项。如果允许多个用户或进程对文件进行并发访问,由于文件系统一般不能避免两个用户几乎

同时修改同一文件的事件发生, 因此, 必然有一个用户的修改会失败, 无法在文件中反映出来。

早期 DBMS 最重要的应用是在那些数据中包含很多小的数据项并且需要许多查询和更新的场合。下面是一些应用的实例。

飞机订票系统

在这里, 每个数据包括如下数据项:

- (1) 一个旅客对一次航班的座位预定, 包括分配座位、进餐选择等信息;
- (2) 航班信息——出发地和目的地, 起飞和到达的时间, 或飞机已经起飞等等;
- (3) 机票信息——票价、要求、有无等。

典型的查询是在某一段时间内从某个给定的城市飞往另一个城市的航班, 还有什么座位可供选择, 以及机票价格。典型的数据更新包括为旅客登记航班, 分配座位, 选择餐饮。任何时刻, 都会有许多代理来访问数据的某些部分。DBMS 必须允许这种“并发访问”; 还要避免一些诸如“两个代理同时分配了同一个座位”之类的问题; 当系统突然崩溃时, DBMS 还要避免记录的丢失。

银行系统

数据项包括顾客的姓名、地址、帐号、存款、结余以及顾客与他们的帐号和存款之间的关系, 比如, 谁对那些帐号有签名权。对结余的查询固然不少, 但更多的是针对一次存款或取款所进行的修改。

正如飞机订票系统一样, 我们希望出纳员和顾客通过 ATM(Automated Teller Machine, 自动出纳机) 能同时查询、更新银行的数据。对同一帐号能同时访问, 而不影响 ATM 业务, 这一点是至关重要的。错误是不能容忍的。例如, 一旦钱从 ATM 自动出纳机中弹出, 银行必须记录这项支出, 即使立刻掉电也不例外。正确处理这种操作远不像想象的那么简单, 可以看作是 DBMS 系统结构的重大进展之一。

公司记录

许多早期应用都与公司记录有关, 比如每次销售的记录, 帐户的应付或应收信息, 雇员信息——他们的姓名、地址、工资、津贴、税款等等。查询的内容包括打印帐户收入或雇员每周工资之类的记录。每次销售、采购, 每张帐单、收据, 雇员的雇用、解雇、提升等等, 都将导致数据库的更新。

从文件系统发展而来的早期的 DBMS 希望用户把数据想象成很像它存放的样子。这些数据库系统使用几种不同的数据模型来描述数据库的信息结构, 其中主要是基于树的层次模型和基于图的网状模型。后者在 60 年代末期通过 CODASYL (数据系统和语言协会, Committee on Data Systems and Languages) 的一份报告进行了标准化。关于网状和层次模型, 读者可以参考 2.7 节, 尽管今天它们已经成了历史话题。

CODASYL Data Base Task Group April 1971 Report, ACM, New York.

这些早期的数据模型和系统的一个弊病是它们不支持高级查询语言。例如，CODASYL 查询语言允许用户通过数据元素之间的指针图从一个数据元素跳到另一个数据元素。即便查询非常简单，编写这样的程序也需要相当大的工作量。

1. 1. 2 关系数据库系统

在 Ted Codd 1970 年的那篇著名的论文 发表以后, 数据库系统发生了显著的变化。Codd 提出数据库系统应为用户提供这样一种观点, 即数据库系统是用一种称为“ 关系 ”的表来组织数据的。而在背后, 可能有一个很复杂的数据结构, 以保证对各种查询的快速响应。但与以前的数据库系统的用户不同, 关系数据库系统的用户并不关心数据的存储结构, 而是使查询能用很高级的语言来实现, 从而大大提高了数据库开发人员的效率。

从第 3 章介绍关系的基本概念开始, 本书的大部分内容都将涉及到数据库系统的关系模型。从第 5 章开始, 将介绍基于关系模型的最重要的查询语言 SQL (Structured Query Language, 结构化查询语言)。然而, 先对关系做一下简单介绍有助于读者对关系模型的了解, 同时, 我们给出一个 SQL 的例子, 以便读者了解关系模型如何支持高级语言的查询, 从而避免在数据库中“ 指引路径 ”的细节。

例 1.1 关系就是表。表的各列以属性开始, 属性是列的入口。下面是一个名为 Accounts (帐户) 的关系, 记录的是银行的帐户信息, 包括 accountNo (帐号)、balance (结余) 和 type (类型):

accountNo	balance	type
12345	1000. 00	savings
67890	2846. 92	checking
...

表的开始是三个属性: accountNo, balance 和 type。属性下面各行, 称为元组(tuple)。这里我们具体给出了两个元组, 当然, 还有许多元组被省略了, 其中每个元组对应于银行的一个帐户。第一个元组中, 帐号为 12345 的帐户上结余为 \$ 1 000. 00, 是一个存款帐户; 第二个元组中, 帐号为 67890 的帐户是一个支票帐户, 结余为 \$ 2 846. 92。

假设我们希望查询 67890 帐号的结余, SQL 的查询语句如下:

```
SELECT    balance
FROM      Accounts
WHERE     accountNo = 67890;
```

再举个例子, 我们可以通过下述语句查询出现赤字的存款帐号:

```
SELECT    accountNo
FROM      Accounts
WHERE     type = 'savings ' AND balance< 0;
```

当然,我们不指望读者通过这两个例子就能成为 SQL 编程的专家,但这两个例子已反映出 SELECT-FROM-WHERE 语句的“高级”性。一般而言,这两个例子要数据库系统做的是:

- (1) 检查 FROM 子句给出的 Accounts 关系的所有元组;
- (2) 选出满足 WHERE 子句给出的某个判别条件的元组;
- (3) 将这些元组按 SELECT 子句指定的某些属性组织成结果输出。

实际上,系统必须优化查询,并找到一条有效的途径回答查询的问题,即使查询中包含的关系非常复杂也是如此。

IBM 很早就推出了关系模型以前的和关系的数据库管理系统(DBMS)。另外,为了实现和销售关系数据库管理系统,一些新公司陆续成立。现在,它们中的一部分已经迈入世界上最大的软件商之列。

1. 1. 3 越来越小的系统

最初,DBMS 是又大又昂贵的软件,只能在大型机上运行。这种规模是必需的,因为存储上 G 字节需要大型的计算机系统。现在,一个硬盘就能容纳上 G 字节,从而使 DBMS 在微机上的实现成为可能。甚至在很小的机器上都能运行基于关系模型的数据库系统,并且,就像以前的电子表格和字处理软件那样,数据库系统也在逐渐成为各种计算机应用的一个通用工具。

1. 1. 4 越来越大的系统

另一方面,G 字节已经不够用了。公司的数据库常常有数百 G 的数据。此外,随着存储器价格的下降,人们也找到了存储更多数据的理由。比如,连锁零售往往需要 T(太, 1 000G, 也就是 10^{12}) 字节或更多的信息来保存很长一个时期以来的每次销售情况(以便计划库存量;我们将在 1. 3. 4 节详细讨论)。数据库系统不再局限于存储简单的数据——整数或短字符串,它们还可以存储图象、声音、视频和其他类型的数据,这些数据都需要相当大的空间。比如,一小时的视频数据大约需要 1G 字节的空间。预计到 2000 年,存储从人造卫星传来的图象的数据库将需要几 P(拍, 1 000T, 即 10^{15}) 字节的存储量。

处理如此巨大的数据库需要若干技术上的变革。例如,中小型数据库现在存储在磁盘阵列上,这些磁盘称为“第二存储设备”(相对于第一存储器——主存而言)。甚至,有人会说数据库系统和其他软件的最主要区别就是数据库系统一般需要处理的数据量太大,内存中根本装不下,而数据的存放通常都应以磁盘为主。下面的两种发展趋势能保证数据库系统更快地处理更大量的数据。

第三存储器

磁盘已经不能满足现在最大的数据库系统的需要了。因此,人们研制出几种第三存储设备。第三存储设备或许可以容纳上 T 字节,但要访问给定的单元需要比磁盘多得多的时间。典型的磁盘访问一个单元需要 10~20 毫秒,而第三存储设备可能需要几秒钟的时间。第三存储设备包括将存放数据的对象传输给读出设备。这种移动由某种形式的机器

人传送动作来完成。

例如, 光盘(CD)就可能是第三存储设备的存储介质。一只装在轨道上的机械手伸向某张指定的 CD, 将其取出, 送到光驱, 而后将其装入光驱。

并行计算

能存储海量的数据固然重要, 但如果不能迅速访问, 也没有太大用处。因此, 大型数据库系统需要加速装置。一种重要的加速措施是采用索引结构, 我们将在 1.2.1 节和 5.7.7 节介绍。另一种方法是利用并行操作, 在给定的时间内处理更多的数据。并行操作表现为多种形式。

比如, 磁盘的读取速度相当低, 大约每秒几兆字节, 但如果我们并行地读许多磁盘, 处理速度就会大大提高(即便数据本来是在第三存储设备上, 在 DBMS 访问它们之前也会“高速缓存”在磁盘中)。这些磁盘可能是并行计算机的组成部分, 也可能是分布式系统的一部分, 分布式系统包括许多计算机, 每台机器负责数据库的一部分, 必要时它们通过高速网络进行通信。

当然, 和数据的海量存储一样, 数据的迅速传送也不能保证查询的迅速响应。我们还需要分解查询的算法, 以便充分利用并行计算机或分布式系统的所有资源。因此, 大型数据库的并行和分布式管理一直是研究和开发的热门话题。

1.2 数据库管理系统的结构

这一节我们将简要介绍典型的数据库管理系统的结构。我们还要看看 DBMS 是如何处理用户的查询和其他数据库操作的。最后, 我们要考虑几个问题, 在设计既要容纳海量数据又要实现高速查询的数据库管理系统时将会碰到这些问题。但 DBMS 的实现技术不是本书的主题, 我们关心的是如何有效地设计和使用数据库。

1.2.1 DBMS 的组成概述

图 1.1 给出了数据库管理系统的主要组成部分。最底部表示存储数据的地方。习惯上, 用圆盘形来表示存储数据的地方。注意, 我们在这里标注的不仅有“数据”, 还有“元数据”(metadata)——有关数据结构的信息。比如, 如果这个 DBMS 是关系的, 那么元数据就包括关系名、这些关系的属性名, 以及属性的数据类型 (如整型或长度为 20 的字符串)。

图 1.1 DBMS 的主要组成部分

通常数据库管理系统需要维护数据的索引。索引是一种数据结构, 能帮助我们迅速找到数据项, 并给出其部分值; 最常见的索引的例子是对于特定关系的某一属性, 查找该属性为给定值的所有元组。例如, 存放帐号和结余的关系也许有帐号的索引, 这样我们就能很快地查到某个给定帐号的结余。索引是数据的一部分, 而说明哪些属性有索引则是元数

索引是如何实现的

读者可能在数据结构课程中学过, 哈希 (Hash) 表是建立索引的一种很有效的方法。的确, 哈希表在早期的 DBMS 中得到了广泛应用。而现在, 最常用的数据结构叫做平衡树 (balanced-tree, 简记为 B-tree)。平衡树是平衡二叉检索树的扩展。二叉树的每个节点最多有两个子节点, 而平衡树的每个节点可以有許多子节点。鉴于平衡树一般出现在磁盘上, 而不是内存中, 人们在设计时就让平衡树的每个节点占据一个磁盘块 (block)。由于典型系统中磁盘块的大小为 2^{12} (4096) 字节, 平衡树的每个块可能有几百个指向子节点的指针。所以, 平衡树的检索很少到三层以上。

磁盘操作的真正开销在于访问的磁盘块的多少。因此, 作为典型的例子, 只检测三个磁盘块的平衡树检索比二叉树检索效率高得多, 因为后者往往需要访问不同磁盘块上的许多节点。平衡树和二叉树的这种区别从一个侧面说明了磁盘上的最佳数据结构与在内存中运行的算法的最佳数据结构是不同的。

据的一部分。

在图 1.1 中, 我们还能看到存储管理程序, 它的任务是从数据存储器获得想要查询的信息, 并在接到上层的更新请求时更新相应的信息。DBMS 的另一个组成部分是查询处理程序, 不过这个名字有点不太恰当。因为它不仅负责查询, 而且负责发出更新数据或元数据的请求。它的任务是接受一个操作请求后, 找到最佳的执行方式, 然后向存储管理程序发出命令, 使其执行。

事务管理程序负责系统的完整性。它必须保证同时运行的若干个查询不互相冲突, 保证系统在出现系统故障时不丢失数据。事务管理程序要与查询处理程序互相配合, 因为它必须知道当前查询将要操作的数据 (以免出现冲突), 为了避免冲突的发生, 也许需要延迟一些查询或操作。事务管理程序也与存储管理程序互相配合, 因为保护数据模式一般需要一个“日志”文件, 记录历次数据的更新。如果操作顺序正确的话, 日志文件将会记载更新的记录, 从而使系统出现故障时根本没有执行的操作在其后能重新执行。

在图 1.1 的最上方, 是三种类型的 DBMS 的输入:

1. 查询。查询就是对数据的询问, 有两种不同的生成方式:

- (a) 通过通用的查询接口。比如, 关系数据库管理系统允许用户键入 SQL 查询语句, 然后将查询传给查询处理程序, 并给出回答。
- (b) 通过应用程序接口。典型的 DBMS 允许程序员通过应用程序调用 DBMS 来查询数据库。比如, 使用飞机订票系统的代理可以通过运行应用程序查询数据库了解航班的情况。可通过专门的接口提出查询要求, 接口中也许包括填城市名和时间之类的对话框。通过这种接口, 你并不能进行任意的查询, 但对于合适的查询, 这种方式通常比直接写 SQL 语句更容易。

2. 更新。更新是指更新数据的操作。同查询一样, 更新操作也可以通过通用的接口或应用程序接口来提出。

3. 模式更新。模式更新命令一般由被授予了一定权限的人使用, 有时我们称这些人

为数据库管理员,他们能够更改数据库模式或者建立新的数据库。比如,假设查询和报告系统(Inquiry and Reporting System, IRS)要求银行报告顾客的社会保险号 and 他们的利息,那么银行就需要在存放顾客信息的关系中加入一个新的属性——socialSecurityNo (社会保险号)。

1.2.2 存储管理程序

在简单的数据库系统中,存储管理程序也许就是底层操作系统的文件系统。但为了提高效率,DBMS 往往直接控制磁盘存储器,至少在某些情况下是这样。存储管理程序包括两个部分——缓冲区管理程序和文件管理程序。

- 1. 文件管理程序对文件在磁盘上的位置保持跟踪,并且负责取出一个或几个数据块,而数据块中含有缓冲区管理程序所要求的文件。磁盘通常划分成一个个连续存储的数据块,每个数据块能容纳许多字节,从 2^{12} 到 2^{14} (大约 4 000 到 16 000) 字节之间。
- 2. 缓冲区管理程序控制处理主存。它通过文件管理程序从磁盘取得数据块,并选择主存的一个页面存放其中的一块。缓冲区管理程序会把数据块在主存中保留一段时间,但当另一个数据块需要使用该页面时,就把原数据块写回磁盘。当然,如果事务管理程序发出请求,缓冲区管理程序也会把数据块写回磁盘(参见 1.2.4 节)。

1.2.3 查询处理程序

查询处理程序的任务是,把很高级的语言表示的查询或数据库操作(如 SQL 查询语句)转换成对存储器数据——比如某个关系的特定元组或部分索引——的请求序列。通常,查询处理任务最困难的部分是查询优化,也就是说选择好的查询计划,即对存储器系统选择好的请求序列以回答所要求的查询。

例 1.2 假设某银行有一个包含两种关系的数据库:

- 1. Customers(顾客)是一张表,给出每个顾客的姓名、社会保险号和地址。
- 2. Accounts(帐户)是一张表,给出每个帐户的帐号、结余和户主的社会保险号。注意,每个帐户都有一个主户主,其社会保险号将用于税款报告;当然,一个帐户也可能还有其他的户主,但这两种关系里不包含此类信息。

假设查询是“查找以 Sally Jones 为主户主的所有帐户的结余”。查询处理程序必须找到在这两种关系上实施的查询方案,从而得到查询结果。回答查询所需的步骤越少,查询方案就越好。通常开销大的步骤是存储管理程序把磁盘数据块复制到缓冲池的内存页面或由内存页面写回磁盘。因此,在评价查询方案的代价时只考虑这些磁盘操作是合理的。

为了回答这项查询,我们需要从 Customers 关系中找到 Sally Jones 的社会保险号(这里,我们假设只有一个顾客叫 Sally Jones,虽然事实上可能不止一个)。然后再从 Accounts 关系查找具有该社会保险号的每个帐户,并打印这些帐户的结余。

一种简单但代价很高的方案是检查 Customers 关系的所有元组,直到我们在顾客姓名域中找到 Sally Jones。平均来讲,我们需要查找半数的元组才能找到目标。鉴于银行有许多的顾客,Customers 关系会占据许多的磁盘块,这种做法代价很高。即便我们找到了 Sally Jones 的社会保险号,还远没有结束。我们还得查找帐户关系中的元组,以找到指定

的社会保险号对应的元组。由于这样的帐户可能有几个,我们不得不查找所有的元组。典型的银行可能会有许多帐户,因而 Accounts 关系也必将占据许多磁盘块。检查所有的元组付出的代价就太大了。

如果 Customers 关系中有姓名索引,那么查询就容易多了。我们只需要使用索引找到含有 Sally Jones 的元组所在的磁盘块,而不是查找整个 Customers 关系。正如我们在 1.2.1 节的方框内所讲的,为了找到想要的索引,典型的平衡树索引需要查找三个索引磁盘块。我们再访问一个磁盘块,就可找到 Sally Jones 所在的元组。

当然,我们还需要做第二步:在 Accounts 关系中寻找 Sally Jones 的社会保险号对应的帐户。这一步往往需要访问磁盘很多次。但如果针对 Accounts 关系存在一个社会保险号索引,那么检索该索引我们就能找到对应于给定社会保险号的帐户所在的每个磁盘块。为了做到这一点,就像我们讨论过的检索 Customers 关系的索引那样,必须用两到三次磁盘访问来检索索引。如果所要的元组分布在不同的磁盘块上,也许我们需要逐个访问这些磁盘块。但很可能,一个 Customers 没有那么多的帐户,因此,这步操作可能只包含几次磁盘访问。如果这两个索引都有的话,也许通过 6~10 次磁盘访问,我们就能回答上述查询了。如果只有一个索引或者两个索引都没有,而不得不用比较差的查询方案,那么,由于我们要扫描整个关系,而关系的规模又很大的话,磁盘访问的次数就可能是几百或几千的数量级。

也许例 1.2 会误导出一个结论——查询优化所做的一切不过是使用现有的索引而已。事实上,这方面还有很多的内容。复杂的查询往往允许我们改变操作顺序,也许会有很多种可能的查询方案,一般其数量是查询规模的指数函数。有时我们能选择使用一个索引,但不能使用两个。这方面的研究是数据库管理系统实现的重要方面之一,但超出了本书的讨论范围。

1.2.4 事务管理程序

我们在 1.1 节已经提到,DBMS 必须对执行数据库操作提供一些特殊的保障。例如,我们曾讨论过,即使在面临严重的系统故障时,操作结果也不能丢失的重要性。典型的 DBMS 允许用户将一个或多个查询和/或更新组成事务(transaction)。事务,非正式地讲,是一组按顺序执行的操作单位。

数据库系统常常允许许多事务并发地执行。例如,有些事情可能在一家银行的所有 ATM 机器上同时执行。保证这些事务全都正确执行是 DBMS 中事务管理程序的任务。更详细地说,事务的“正确”执行还需要通常称为 ACID 的特性。ACID 取自于事务执行的四个主要需求的首字母。这四个特性是:

- 原子性(atomicity)。我们需要整个事务或者都执行或者都不执行。例如,从 ATM 机器中取钱和记入相关借方的顾客帐户上应该是一个原子事务。如果钱已经付出了而没有记入借方帐户或者记入了借方帐户而钱并没有付出,都是不能接受的。

实际上,由于平衡树的根节点在涉及该索引的每次查找中都要用到,所以它所在的磁盘块常在主存中占据一个缓冲区页面,这样,通常两次磁盘块访问就足够了。

锁的粒度

不同的 DBMS 可能对不同种类的数据项加锁。例如,可以对关系中的单个元组加锁,也可以对单个磁盘块加锁,甚至对整个关系加锁。加锁的单元越大,一个事务必须等待另一个事务的可能性就越大,即使它们实际上并不访问同一数据。而加锁的单元越小,加锁机制就越大,越复杂。

- 一致性(consistency)。数据库通常都有“一致状态”的概念,即数据符合我们的所有期望。例如,对航班数据库而言,“一个座位不能分配给两个不同的顾客”就是一种适当的一致性条件。在事务处理过程中的某个时刻,由于旅客之间调换座位,可能会违背这种一致性条件,但事务结束后,事务管理程序必须保证数据库满足所有约定的一致性条件。
- 隔离性(isolation)。当两个或更多的事务并发运行时,它们的作用效果必须互相分开。也就是说,我们看到的两个事务并发运行的效果必须同两个事务一前一后运行时的效果完全一样。例如,如果两个机票代理正在出售同一航班的座位,而座位只剩下一个了,那么只能答应一个代理的请求,而拒绝另一个。如果由于并发操作导致同一座位卖了两次或根本没卖,都是不能接受的。
- 持久性(durability)。如果事务已经完成,即使系统出现故障,甚至事务刚刚完成,就出现了系统故障,事务的结果也不能丢失。

如何实现事务,使其具有 ACID 特性,这本身就能构成一本书的内容。我们就不准备在这里讨论这个问题了。但 7.2 节将会讨论在 SQL 语言中,如何指定属于一个事务的操作,以及 SQL 程序员能够期望从成组的操作到事务的转换中得到什么保证。我们在本节中还将提纲挈领地介绍一些实现 ACID 特性的常用技术。

加锁

造成事务间不独立的主要原因是两个或多个事务同时读写数据库中的同一数据项。例如,两个事务企图同时对同一帐户的结余进行修改,后一个写操作将会覆盖前一个,那么前一个写操作的结果就丢失了。因此,大多数 DBMS 的事务管理程序能够对事务要访问的数据项加锁。一个事务对某数据项加锁后,其他的事务就不能访问它了。例如,第一个事务锁定帐户 12345 的结余,于是在另一个事务获准访问它之前,第一个事务既能对它进行读操作,也能写入新值。第二个事务将读出新的结余,而不是旧的。这两个事务间就不会有不良的影响了。

日志

事务管理程序记录了一个日志文件,包括每个事务的开始、每个事务所引起的数据库的更新和每个事务的结束。日志总是记在非易失性存储器上,像磁盘这样的存储介质,掉电后数据仍完好保存。因此,虽然事务本身的工作区可能使用易失性的主存,而日志却总是直接写到磁盘。记录所有的操作是保证持久性的重要手段。

事务提交

为了保证持久性和原子性,事务一般以“ 试验 ”方式完成,也就是说,在试验过程中计算对数据库要做的更新,但并不真正地更新数据库本身。事务即将完成时,也就是事务提交时,更新的内容均已复制到日志记录中。该日志记录首先复制到磁盘上。然后才把更新的内容写入数据库本身。

即使系统在这两步中间出现了故障,当系统恢复正常后,我们可以阅读日志文件,看看还需要进行哪些数据库更新操作。如果系统在所有的更新操作写入日志文件之前出现了故障,我们就重新执行该事务,确保不会偶尔发生诸如两次预定航班座位或者两次记入借方帐户之类的事情。

1.2.5 客户程序-服务程序体系结构

许多种现代软件采用客户程序-服务程序体系结构,这种体系结构中,把一个进程(客户程序)发出的请求送到另一个进程(服务程序)去执行。数据库系统也不例外,通常将图1.1中各组成部分的工作分成一个服务进程和一个或多个客户进程。

在最简单的客户程序/服务程序体系结构中,除了与用户相互配合并将查询或其他命令传给服务程序的查询接口以外,整个 DBMS 就是一个服务程序。例如,关系系统通常用 SQL 语言来表达从客户程序到服务程序的各种请求。然后数据库服务程序给出回答,用表即关系的形式传给客户程序。客户程序和服务程序之间的关系可能会更复杂,尤其是答案极大的时候。关于这个问题,我们将在 1.3.3 节详细介绍。如果有很多用户同时使用数据库的话,服务程序就会成为“ 瓶颈 ”,所以也有一种趋势——让客户程序做更多的工作。

1.3 未来的数据库系统

今天,数据库的领域有很多发展趋势,这些趋势将引导这门学科沿着各种各样的新方向发展。其中有些是正在改变传统 DBMS 特性的新技术——例如面向对象的编程,约束和触发,多媒体数据,或者万维网(World Wide Web, WWW)。其他的发展趋势包括新的应用,如数据仓库或信息集成。这一节我们将简单地介绍一下未来数据库系统的几种主要发展趋势。

1.3.1 类型、类和对象

人们普遍认为面向对象的编程是更好的程序结构工具,甚至是更可靠的软件实现工具。面向对象的编程,首先在 Smalltalk 语言中推广,随着 C++ 的发展和许多以前基于 C 的软件开发向 C++ 移植,获得了很大的发展。最近,适于在万维网上进行程序共享的 JAVA 语言,也提高了人们对面向对象编程的关注。面向对象的实例对数据库领域也同样有吸引力,而一些公司正在销售所谓“ 面向对象 ”的 DBMS。这里我们将会回顾隐藏在“ 面向对象 ”之后的概念。

类型系统

面向对象的编程语言给用户提供了丰富的类型聚集。从基本类型,通常是整型、实型、布尔型和字符串开始,可以用类型构造符构造新的数据类型。类型构造符一般允许我们构造:

(1) 记录结构(record structure)。给出类型表 T_1, T_2, \dots, T_n 和对应的域名表(在 Smalltalk 中称为实例变量,即 instance variable) f_1, f_2, \dots, f_n , 就可以构造一个包含 n 个成分的记录类型。第 i 个成分的类型是 T_i , 用它的域名 f_i 来引用。记录结构其实就是 C 或 C++ 中的“结构”(struct)。

(2) 聚集类型(collection type)。给定一个类型 T , 就可以用聚集运算符将其构造成新的类型。不同的语言使用不同的聚集运算符,但有一些是通用的,包括数组、列表和集合。因此,如果 T 是基本类型整型,我们可以构造整型数组、整型列表或整型集合等聚集类型。

(3) 引用类型(reference type)。类型 T 的引用是这样一种类型,它的值用来指向类型 T 的值。在 C 或 C++ 中,引用是指向某个值的“指针”,也就是说,引用是个单元,其中存放着该值所在的虚拟存储器的地址。指针模型常用于帮助理解引用。然而,在数据库系统中,由于数据存储在很多磁盘上,或许分布在许多主机上,引用必然比指针复杂得多。比如,引用可能包括引用的值所在的主机名、磁盘号、该磁盘中的块号和在该块中的位置。

当然,记录结构和聚集运算符可以重复地使用,以构造更复杂的类型。例如,我们可以定义一个记录结构类型,它的第一个成分是字符串类型,名为“customer”,第二个成分是整型集合类型,名为“accounts”,这种类型适于将银行顾客和他们的帐户集合联系起来。

类和对象

类包括类型,可能还有一个或多个能在该类的对象上执行的函数或过程(称为方法,参看下文)。类的对象或者是属于该类型的值(称为不变对象),或者是其值属于该类型的变量(称为可变对象)。例如,如果我们定义了一个类型为“整型集合”的类 C , 则 $\{2, 5, 7\}$ 是类 C 的不变对象,然而,也可说明变量 s 属于类 C , 且赋值为 $\{2, 5, 7\}$ 。

对象标识

假定每个对象都有个对象标识(object identification, OID)。两个对象不能有相同的 OID, 一个对象也不能有两个不同的 OID。OID 是该对象的引用所拥有的值。我们通常会认为 OID 是虚拟存储器中指向该对象的指针,但正如我们在指针与引用类型的关系中所讲的,数据库系统中的 OID 实际上更复杂一些:是一个足以将在任何大量的不同机器的第二或第三存储器上的对象准确定位的位序列。而且,由于数据是持久的,只要数据存在,任何时刻 OID 都必须是有有效的。

方法

与类有关的通常还有某些函数,往往称为“方法”。类 C 的一个方法至少有一个类 C

的对象作为参数。它可能有包括 C 在内的任何类的其他参数。例如,可能有一个方法与一个类型为“整型集合”的类相关,而用该方法可以计算给定集合的幂集,求两个集合的并集,或返回一个布尔值以指出集合是否为空。

抽象数据类型

在许多情形下,类也是抽象数据类型,意思是类封装或限制对类对象的访问,因此,只有为类定义的方法才能直接修改类的对象。这种限制保证了类对象不以类的实现者所不期望的方式进行修改。人们把这种概念看作是可靠软件开发的关键工具之一。

类的分层结构

可以说明一个类 C 是另一个类 D 的子类。这样的话,类 C 将会继承类 D 的所有特性,包括 D 的类型和为 D 定义的任何函数。然而,C 可能有一些附加的特性。比如,可以为类 C 的对象定义新的方法,这些方法可以是 D 的方法的补充,也可以代替 D 的方法。甚至还可以用某些方式扩展 D 的类型。尤其是,如果 D 属于记录结构类型,我们就能在此类型的基础上为 C 类增加新的域,这些域只出现在 C 类的对象中。

例 1.3 考虑某银行帐户类的对象。我们可以非正式地将该类的类型描述为:

```
CLASS Account = {accountNo: integer;
                  balance: real;
                  owner: REF Customer;
                  }
```

也就是说,类 Account 的类型是记录结构,包含三个域:整型的 accountNo、实型的 balance 以及 owner,而户主是对类 Customer(顾客类,是我们在银行数据库中需要的另一个类,但它的类型我们在这里不作介绍)的对象的引用。

我们也可以为类定义一些方法。例如,我们可能有一个方法 deposit(存款),用来将类 Account 的对象 a 的结余增加 m:

```
deposit (a: Account, m: real)
```

最后,我们也许希望有一些 Account 的子类即明细帐目。例如,一个定期存款(time-deposit)帐户可能有一个附加的域 dueDate(支付日期),指明户主取出结余的日期。在子类 TimeDeposit 中也可能有另一个方法 penalty(违约金):

```
penalty (a: TimeDeposit)
```

该方法取出属于子类 TimeDeposit 的帐户 a,并计算因提前支取的违约金。违约金是对象 a 的 dueDate 域和当前日期的函数;当前日期可以从该方法运行的系统上得到。

本书中我们将会广泛地考虑数据库系统面向对象的概念。在 2.1 节我们会介绍面向对象的数据库设计语言 ODL。第 8 章将专门介绍面向对象的查询语言,包括正在成为面向对象 DBMS 标准的查询语言 OQL,以及为关系 DBMS 的标准查询语言 SQL 推荐的面向对象的特性。

为何使用对象？

面向对象的编程为数据库系统提供了几项重要功能：

- 利用丰富的类型系统, 可以处理比关系模型或更早的数据模型更自然的数据形式。注意: 关系模型的类型系统多少有些局限。关系就是记录的集合, 而记录结构的域(关系模型中称为“属性”)都是基本类型。
- 利用类和类的分层结构, 可以比传统系统更容易地进行软件和数据模式的共享或复用。
- 利用抽象数据类型, 可以限制对数据的访问, 从而防止数据的误用, 除非通过仔细设计的某些函数来访问数据, 已了解这些函数能正确使用数据。

1.3.2 约束和触发程序

数据库系统的另一个新发展趋势是在商用系统中广泛使用主动性元素。所谓“主动”指的是数据库的组成部分在任何时刻都是准备好了的, 无论何时, 只要时机合适就可以立刻执行。在数据库系统中, 通常有两种主动性元素:

1. 约束。约束是布尔型的函数, 要求其值为真。例如, 我们可能在银行数据库中设置一个约束: 结余不能小于 0。DBMS 会拒绝不满足该约束的数据库修改操作, 比如, 导致帐户结余为负的提取。

2. 触发程序。触发程序是一段等待某事件发生的代码; 这种可能的事件可以是某种数据项的插入或删除等。当事件发生时, 要执行(也就是要触发)相关的一系列动作。例如, 飞机订票系统可能有这样一种规则, 其触发条件是航班的状态变为“取消”。该规则的动作部分可能是查询, 查找预定了该航班座位的所有顾客的电话号码, 以便通知这些顾客。更复杂的动作也许是自动为这些顾客预订别的航班。

主动性元素并不是什么新的想法。在程序设计语言 PL/I 中, 它们就以“ON-condition”出现了。在人工智能系统中, 它们也已出现了许多年, 而且, 它们在操作系统中类似于“端口监控程序(daemons)”。然而, 当主动性元素操作的数据规模很大或主动性元素的数量很多时, 如何有效地实现主动性元素, 存在着严重的技术问题。为此, 直到 90 年代初期, 主动性元素才作为 DBMS 的标准组成部分。我们将在第 6 章讨论主动性元素。

1.3.3 多媒体数据

数据库系统的另一个重要发展趋势是包容多媒体数据。所谓“多媒体”, 指的是表示某种信号的信息。通常的多媒体数据形式包括各种编码的视频、音频、雷达信号、卫星图象以及文本或图形。这些形式的共同之处在于它们比以往形式的数据——整型、定长字符串等等——在容量上大得多, 并且其容量的变化范围也大得多。

多媒体数据的存储促使 DBMS 以几种方式扩展。比如, 在多媒体数据上进行的操作与适用于传统数据形式的简单操作不同。因此, 虽然我们可以通过比较每个结余和实数 0.0 来检索银行数据库中结余为负的帐户, 但在图象数据库中, 要检索某幅脸部像特定图

象的图,用这种方法是行不通的。于是,DBMS 必须为用户提供一种能力,允许用户引入他们自己选择的、可以应用在多媒体数据上的函数。通常,用面向对象的方法进行这种扩展,即使在关系系统中也是如此。

多媒体对象的容量也促使 DBMS 修改存储管理程序,以适应 G 字节或更大的对象或元组。容量如此大的元素如何表示,存在很多问题,其中包括查询结果的传输问题。在传统的关系数据库中,查询结果是元组的集合,可以作为一个整体由数据库服务程序传输给客户程序。

然而,如果查询的结果是 G 字节长的一段视频剪辑,服务程序要把 G 字节作为一个整体传给客户程序是不可能的。一个原因是时间太长,将会妨碍服务程序处理其他的请求。另一个原因是,客户程序可能只想要电影剪辑的一小部分,但在没有看到剪辑的起始部分之前,无法确定究竟想要哪一部分。第三个原因,即使客户程序想要整个剪辑,或许是为了在屏幕上播放,以固定的速率用一小时传输已经足够了(一小时是播放 1G 字节的压缩视频所需的时间)。因此,多媒体 DBMS 的存储系统必须准备好以交互方式传输结果,传送客户程序请求的片断或以固定的速率进行传送。

1.3.4 数据集成

由于信息在我们的工作和娱乐中变得越来越重要,我们发现人们正在以许多新的方式利用现有的信息资源。例如,设想有一个公司想为它的所有产品提供联机目录,使人们能利用万维网浏览它的产品并发出联机订单。大公司往往有很多部门,每个部门可能已经各自独立地建立了自己的产品数据库。这些部门可能使用不同的 DBMS、不同的信息结构,甚至用不同的术语表示同一事物,或者用同一术语表示不同的事物。

例 1.4 假设某磁盘制造公司由几个部门组成。一个部门的产品目录以每秒的转数来表示旋转速率,另一个则以每分钟的转数来表示旋转速率,而还有一个部门根本就忽略了旋转速度。生产软盘的部门可能将软盘称为“磁盘”;而生产硬盘的部门也可能称硬盘为“磁盘”。磁盘上的磁道数在一个部门可能叫做“磁道数”,在另一个部门可能叫做“柱面数”。

集中控制并不总是解决的办法。在各部门意识到它们之间的完整性是一个问题之前,可能已经在各自的数据库上投入了大量的资金。还有的部门可能由于需要最近已经成了独立的公司。由于这种或那种原因,并不那么容易淘汰这些所谓的“遗留数据库”。因此,公司必须在遗留数据库的上层建立某种结构,以便给顾客提供整个公司统一的视图(view)。

一种流行的方法是建立数据仓库(data warehouse),即许多遗留数据库中的信息通过适当的转换复制到中央数据库。当遗留数据库变更的时候,数据仓库也跟着更新,但不必立即更新。通常的安排是每天晚上遗留数据库不太忙的时候重构数据仓库。

这样,遗留数据库就能继续用于当初建立的场合,而诸如通过 Web 网提供联机产品目录服务等新的功能则在数据仓库上完成。我们看到,数据仓库还符合规划和分析的需要。例如,公司分析员可以在数据仓库上进行查找销售趋势的查询,以便更好地规划库存和生产。数据仓库的构造也支持数据挖掘,即在数据中寻找感兴趣的、不寻常的模式,并且

人们通过利用以这种方式发现的模式,而提高了销售额。

1.4 本书概要

与数据库系统有关的概念可以分为三个主要的范畴:

1. 数据库设计。如何建立一个有用的数据库?数据库应包含哪些类型信息?信息是如何构成的?对数据项的类型和取值有些什么假设?数据项之间是如何联系的?
2. 数据库编程。如何表达对数据库的查询和其他操作?如何使用 DBMS 的其他能力,比如事务和触发程序?
3. 数据库系统实现。如何建立一个 DBMS,包括诸如查询处理、事务处理以及为了有效的访问而进行的存储管理?

虽然数据库系统的实现是软件行业的一个主要部分,但是设计或使用数据库的人数却远远超过了建立数据库的人数。本书预计作为数据库系统的基础教程,因此关注两个基础的部分:设计和编程,是恰当的。这一章我们试图使读者对第三部分——实现——有个粗浅的了解,但以后,我们不在本书讨论这个问题了。相反地,本书的其余各章将按设计和编程的内容划分如下。

1.4.1 设计

第2章和第3章的内容是设计。第2章先介绍表达数据库设计的两种高级表示法。一种是对象定义语言(Object Definition Language, ODL),即一种面向对象的说明类的语言;另一种是实体/联系(Entity/Relationship, E/R)模型,即一种用来描述数据库组织的图形表示法。

无论是 ODL 还是 E/R 模型都不能直接用于定义数据库的结构,尽管对于面向对象的 DBMS 而言,ODL 非常接近于数据定义语言。相反地,我们更倾向于将这两种模型之一表达的设计转换成任何一种由数据定义语言所使用的正式表示法,而该数据定义语言与所使用的 DBMS 相关。鉴于大多数的 DBMS 都是关系的,我们将集中讲解如何把 ODL 或 E/R 模型转换成关系模型。所以,第3章专门讲述关系模型和转换过程。然后,5.7节将介绍如何用 SQL 语言的数据定义部分正式地描述关系数据库模式。

第3章也将为读者介绍“依赖”的概念,它是正式提出的关系中各元组之间联系的假设。依赖允许我们通过关系的所谓“规范化”的处理,以有益的方式将关系分解。依赖和规范化将在3.5节和随后的几节介绍;它们是设计关系数据库的重要部分。不管是直接用关系模型设计数据库,还是把所设计的 ODL 或 E/R 模型转换成关系后发现了设计中的某些问题,这部分内容都是很有用的。

1.4.2 编程

第4到8章涉及数据库编程。第4章从关系模型中查询的抽象处理开始,进而介绍构成“关系代数”的一组关系操作。我们也会讨论另一种描述查询的方式,该方式以逻辑表达式为基础,称为“Datalog”。

第 5 到 7 章的内容是 SQL 编程。正如我们曾提到的, SQL 是当前居支配地位的查询语言。第 5 章介绍了有关 SQL 查询的基本概念, 以及 SQL 数据库模式的表达式。这一章和随后两章的几乎所有内容都是基于称为 SQL2 的 SQL 标准版本的。不过, 一些商用系统的 SQL 编程的某些方面并未包括在 SQL2 中。这种情形下, 我们就使用新近的标准, 叫做 SQL3, 但它还未被正式采纳。

第 6 章涉及到 SQL 数据的约束和触发方面。由于 SQL2 在这些方面的局限性, 因此我们除了介绍 SQL2 以外, 还专门花些时间介绍 SQL3 中的约束和触发的处理。

第 7 章将介绍 SQL 编程中的一些高级方面。首先, 虽然 SQL 编程的最简单的模型是独立于操作系统的通用的查询接口, 但实际上, 大多数 SQL 编程都是嵌入到用 C 之类的传统语言编写的更大的程序中。在第 7 章, 我们将学会如何把 SQL 语句与外层程序结合起来, 并且在数据库和程序变量之间传递数据。这一章还包括如何使用 SQL 的如下特性: 定义事务, 连接客户程序和服务程序, 授权他人访问数据库。

第 8 章我们的注意力转移到面向对象数据库编程的正在形成的标准上。在这里, 我们考虑两个方向: 第一, OQL (对象查询语言, Object Query Language), 可以看作是使 C++ 与高级数据库编程命令兼容的一种尝试; 第二, 就是 SQL3 具有的面向对象特性, 可以看作是使关系数据库和 SQL 与面向对象编程兼容的一种尝试。在一定程度上, 这两种方法有着共同的基础。但是, 它们也存在一些本质的区别。

1.5 本章总结

数据库管理系统: DBMS 允许设计者构造自己的信息, 允许用户查询和更新这些信息, 并帮助管理海量数据和对数据的许多并发操作。

数据库语言: 有些语言或语言成份用于定义数据的结构(数据定义语言)以及查询和更新数据(数据操作语言)。

关系数据库系统: 今天, 大部分的数据库系统以数据的关系模型为基础, 也就是把信息组织成表。SQL 是这些系统中最常用的语言。

面向对象的数据库系统: 一些现代的数据库系统使用面向对象的数据建模思想, 包括类, 很大的类型系统, 对象标识和子类的继承性。将来, 大多数数据库管理系统, 包括关系数据库管理系统, 都将支持部分或全部上述概念。

第二和第三存储设备: 大的数据库存储在第二存储设备上, 通常是磁盘。极大的数据库需要第三存储设备, 它们的容量比磁盘大几个数量级, 但速度也慢几个数量级。

DBMS 的组成: 数据库管理系统的主要组成部分是查询处理程序、事务管理程序和存储管理程序。

存储管理程序: 存储管理程序既要管理第二存储设备上的数据文件, 又要管理包含这些文件一部分内容的内存缓冲区。数据库管理系统一般会维护索引, 而索引这种数据结构可支持对数据的有效访问。

查询管理程序: 查询管理程序的一个重要任务是“优化”查询, 也就是说, 为给定的查询寻找好的解答算法。

事务管理程序:事务是数据库的基本工作单元。事务管理程序在确信事务具有 ACID 特性,即原子性、一致性、隔离性和持久性的情况下,允许许多事务并发地执行。

客户程序-服务程序系统:数据库管理系统通常支持客户程序-服务程序体系结构,数据库的主要部分都在服务程序中,而客户程序则负责用户接口。

数据库的主动性元素:现代数据库系统支持某种形式的主动性元素,通常是触发程序和/或完整性约束。

未来的系统:数据库系统的主要发展方向包括支持像视频或图象那样庞大的“多媒体”对象,以及将许多独立信息源的信息集成为单一的数据库。

1.6 本章参考文献

许多书的内容涉及到数据库系统实现的重要方面。[3]和[5]涉及到事务管理程序的实现。这些著作和[7]都论述了分布式数据库的实现。[11]讨论了文件管理程序的实现。

[2]、[4]和[6]论述了面向对象的数据库系统。[1]、[9]和[10]涉及数据库系统的理论。在[8]中可以找到许多对数据库领域的研究做出贡献的论文。

- [1] Abiteboul, S., R. Hull, and V. Vianu, Foundations of Databases, Addison-Wesley, Reading, MA, 1995.
- [2] Bancilhon, F., C. Delobel, and P. Kanellakis, Building an Object-Oriented Database System, Morgan-Kaufmann, San Francisco, 1992.
- [3] Bernstein, P. A., V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.
- [4] Cattell, R. G. G., Object Data Management, Addison-Wesley, Reading, MA, 1994.
- [5] Gray, J. N. and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan-Kaufmann, San Francisco, 1993.
- [6] Kim, W. (ed.), Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press, New York, 1994.
- [7] Oszu, M. T. and P. Valduriez, Principles of Distributed Database Systems, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [8] Stonebraker, M. (ed.), Reading in Database Systems, Morgan-Kaufmann, San Francisco, 1994.
- [9] Ullman, J. D., Principles of Database and Knowledge-Base Systems, Volume I, Computer Science Press, New York, 1988.
- [10] Ullman, J. D., Principles of Database and Knowledge-Base Systems, Volume II, Computer Science Press, New York, 1989.
- [11] Wiederhold, G., Database Design, McGraw-Hill, New York, 1983.

第 2 章 数据库建模

设计数据库的过程首先是分析数据库必须保留什么信息以及该信息各成份之间的联系。数据库的结构,称为数据库模式,通常用适合表达这种设计的某种语言或表示法加以说明。经过适当的考虑之后,把设计固定为一种格式,再把按这种格式进行的设计输入到数据库管理系统中,这时数据库就有了具体的存在形式。

在这本书中,我们将使用两种设计表示法。比较传统的方法称为实体-联系(E/R, Entity-Relationship)模型,具有图的特性,用方框和箭头表示基本的数据元素和它们之间的连接。同时我们将介绍 ODL(对象定义语言, Object Definition Language),它是面向对象的数据库设计方法,是面向对象的数据库系统正在形成的标准。本章还介绍另外两种模型:网状模型和层次模型,这主要是对过去的关注。在某种意义上,它们是 ODL 的范围有限的版本,已经用在 70 年代实现的商业数据库系统中。

在第 3 章,我们把注意力转向关系模型。关系模型用表的聚集来描述世界,因而其表达力多少受到些限制。然而,这个模型非常简单实用,是现行主要的商业数据库管理系统的基础。通常,数据库设计者首先使用 E/R 模型或基于对象的模型建立模式,然后将其转换成关系模型,以利于实现。

图 2.1 给出了设计的过程。首先是对信息建模的想法。这些想法可以通过某种设计语言加以表达。E/R 和 ODL 是两种可供选择的方法,当然还有其他的方法。在大多数情况下,设计将用关系数据库管理系统来实现。这时通过完全机械的过程(我们将在第 3 章讨论),就可把抽象的设计转换成具体的关系数据库实现。我们还展示了另一种可供选择的方法,在这里,ODL 的设计成为面向对象数据库管理系统的输入。在这种情况下,转换是相当自动化的,可能涉及从 ODL 语句向相应的面向对象编程语言(例如 C++ 或 Smalltalk)的语句的简单转换。

图 2.1 数据库建模和实现的过程

2.1 ODL 介绍

ODL(对象定义语言),是用面向对象的术语(就像人们在诸如 C++ 或 Smalltalk 之类的语言中遇到的那样)说明数据库结构的推荐的标准语言。它是 IDL(Interface Definition language,接口定义语言)的扩展,又是 CORBA(Common Object Request Broker Architecture,公用对象请求代理程序体系结构)的一个组件。CORBA 是正在制订的分布

ODL 是规定数据库模式即数据库结构的语言。它不具有定义数据库中实际内容的功能,也不提供对数据的查询或操作。正如我们在 1.1 节提到的,像 ODL 这样规定模式的语言通常称为数据定义语言,而规定数据库的内容或者查询和修改其数据的语言则称为数据操纵语言。只有在第 4 章从用户的角度去看待数据库时,我们才讨论数据操纵语言。数据定义语言是从设计者的角度观察数据库的核心。

式面向对象计算的标准。

ODL 的主要用途是书写面向对象数据库的设计,进而将其直接转换成对面向对象数据库管理系统(OODBMS)的说明。因为 OODBMS 总是把 C++ 或 Smalltalk 作为它的基本语言,因此必须把 ODL 转换成其中一种语言的说明。ODL 和这两种语言相似(但和 C++ 更为相似),所以图 2.2 所提出的转换是非常直接的。相比之下,要将 ODL 或实体联系设计转换为适合更常见的关系数据库管理系统(RDBMS)的说明却相当复杂。

图 2.2 把 ODL 设计转换成对 OODBMS 的说明

2.1.1 面向对象的设计

在面向对象的设计中,人们把准备模型化的世界看作由对象组成,而对象是某种可观察的实体。例如,人、银行的帐户、航班、大学的课程、建筑物等都可以作为对象。假定对象有唯一的对象标识(OID, Object IDentity),这是它和任何其他对象的区别,就像我们在 1.3.1 节讨论的那样。

为了组织信息,我们总是将具有相似特性的对象归为一类。在数据库中,对象和类的概念与面向对象编程语言(例如 C++ 或 Smalltalk)中得到的那些概念基本相同(再回忆一下我们在 1.3.1 节对面向对象概念的讨论)。然而,当提到 ODL 面向对象设计时,我们应该从两个不同的方面考虑一类对象的相似特性。

- 由属于一类的对象所表示的现实世界在概念上应该是类似的。比如,将银行的所有顾客归为一类和将银行的所有帐户归为另一类都是有意义的。但是将顾客和帐户合并成一类是没有意义的,因为在银行世界中它们很少有或者没有共同之处,而是扮演着本质上不同的角色。
- 属于一类的对象其特性必须相同。当用面向对象的语言编程时,我们经常把对象看成记录,就像图 2.3 所假设的那样。对象含有域或槽,值将放在其中。这些值可能是普通类型,如:整型、字符串或数组,也可能是对其他对象的引用。这些值还可能是方法,也就是用于对象的函数。然而,我们在学习 ODL 时,不强调方法的使用。

OID 的特性

正如我们在 1.3.1 节提到的那样,面向对象数据库往往非常大,以至于所需要的 OID 的数量大大超过在一个地址空间中地址的数量。于是,面向对象数据库系统通常用某种模式来建立与每个对象相联系的唯一的字符串;这个字符串往往很长,可能是十六个字节。例如一个对象可能把它建立的时间(用足够小的单位来度量,以至于在一台机器上不能同时建立两个对象)和建立它的主机的标识一起作为它的标识(如果数据库系统分布在几台主机上)。

用,因为任何面向对象的编程语言对方
法的使用都是类似的。在 8.1 节,我们将
回到对 ODL 方法问题的讨论上。

虽然将对象想象成具有记录的结构往往是
有益的,但是本章主要讨论抽象层次上的设计。

图 2.3 表示帐户的对象

因此,我们首先应强调类及其特性的更抽象的概念,而不考虑实现的细节,例如记录的域是如何组织的,或者对象是否确实用记录结构来表示。

当说明 ODL 类的设计时,需要描述的三种特性是:

- 1. 属性(attribute),是一些特性,它们的类型是由基本的数据类型(例如整型或字符串)构成的。特别是,一个属性有一个和任何其他类无关的类型。在 ODL 中,属性的类型是有限的;我们将在 2.1.7 节进一步讨论。
- 2. 联系(relationship),是一些特性,它们的类型或是对某类对象的引用或是这种引用的聚集(例如,一个集合)。
- 3. 方法(method),是能用于该类对象的函数。如上所述,在这里我们不强调方法的使用。

2.1.2 接口说明

在 ODL 中,形式最简单的类的说明应包括:

- 1. 关键字 interface(接口)。
 - 2. 接口的名字(也就是类名)。
 - 3. 用花括号括起来的类的特性表。回忆一下,这些特性是属性、联系和方法。
- 也就是说,接口说明的简单形式是

```
interface 名字 {
    特性表
}
```

2.1.3 ODL 中的属性

最简单的一种特性称为属性。这些特性通过将一个对象和某个简单类型的值相连来

描述该对象的某个方面。例如,对每个人来说,对象“人”都会有一个字符串型的属性 name, 它的值是这个人的名字。对象“人”还会有一个属性 birthdate, 它是一个由三个整数组成的元组(即一个记录结构), 代表这个人的出生年、月、日。

例 2.1 图 2.4 是 ODL 中对电影类的说明。它不是完整的说明; 我们将在以后对其进行补充。1) 行说明 Movie 是一个类。在 ODL 中使用关键字 interface 来说明类, 1) 行之后是对四个属性的说明, 这四个属性是所有的电影对象所共有的。

```
1) interface Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film {color, blackAndWhite} filmType;
};
```

图 2.4 ODL 中类 Movie 的说明

第一个属性在 2) 行上, 命名为 title。其类型为 string, 即长度未知的字符串。我们希望在任何 Movie 对象中, 属性 title 的值都是电影的名字。3)、4) 行说明下面两个属性 year 和 length 都是整形, 分别代表电影的制作年份和以分钟计算的电影的 length。5) 行是另一个属性 filmType, 它说明电影是彩色的还是黑白的。其类型为枚举类型。枚举类型的名字是 Film。枚举属性的值从文字表中选择, 在这个例子中就是从 color 和 blackAndWhite 中选择。

符合上述定义的类 Movie 的对象, 可以认为是具有四个分量的记录或元组, 四个属性中的每一个都对应一个分量。例如,

```
( Gone With the Wind , 1939, 231, color)
```

就是一个 Movie 对象。

例 2.2 在例 2.1 中, 所有的属性都属于原子类型。我们也可以有类型为结构、聚集或结构聚集的属性, 正如我们将在 2.1.7 节讨论的那样。以下是非原子类型的例子:

我们可以如此定义类 Star:

```
1) interface Star{
2)     attribute string name;
3)     attribute Sruct Addr
           {string street, string city} address;
};
```

2) 行说明一个字符串的属性 name(影星的名字)。3) 行说明另一个属性 address。这个属性属于记录结构类型。这个类型的名称为 Addr, 类型包括两个域 street 和 city。两个域都是字符串型。通常, 在 ODL 中可以通过使用关键字 Struct 并用花括号将域名及其类型表括起来, 定义记录结构类型。

从技术上讲, 在 ODL 中, 类是接口以及与该接口有关的数据结构和方法的实现。本节并不讨论 ODL 接口的实现, 但将继续把接口说明作为“类”的定义。

2.1.4 ODL 中的联系

虽然我们可以通过研究属性获得关于对象的很多信息,但有时有关一个对象的关键性信息是它与同类或不同类的其他对象连接的方式。

例 2.3 现在,假定我们想在例 2.1 的 Movie 类的说明中增加一个影星集合的特性。因为影星本身是一个类,正如例 2.2 中所描述的,我们不能把这个信息作为 Movie 的属性,因为属性的类型不能是类也不能从类中构造。相反地,电影的影星集合是 Movie 和 Star 两类之间的一种联系。我们在 Movie 类的说明中,用下面一行表示这种联系:

```
relationship Set Star stars;
```

这一行可以出现在图 2.4 的 1) 至 5) 行的任一行之后。它说明在 Movie 类的每个对象中,都有一个对 Star 对象的引用集合。引用集称为 stars。关键字 relationship(联系)说明 stars 包括对其他对象的引用,而在 Star 之前的关键字 Set 表明 stars 引用 Star 对象的集合,而不是单一的对象。一般说来,如果一个类型是某个其他类型 T 作为元素的集合,那么在 ODL 中,用关键字 Set 和用尖括号括起来的类型 T 来定义它。

实际上,我们可以想象用指向 Star 对象的指针列表来表示集合 stars;而 Star 对象实际上不会出现在 Movie 对象中。但是在数据库设计阶段,实际的表示是未知的,而联系的重要方面是从 Movie 对象可以方便地找到这部电影的影星。

在例 2.3 中,我们看到用一个联系把 Star 对象的集合 stars 和单一的对象 Movie 联系起来。联系也可能把单一的对象和要说明的类的对象联系起来。例如,假定在例 2.3 中,已经给出联系的类型是 Star,而不是 Set Star,如下行所示:

```
relationship Star starOf;
```

那么这个联系将把单一的 Star 对象和每一部电影联系起来。在这里,这种方案没有太大意义,因为,一部电影通常有多个影星。然而,我们将看到在其他许多例子中,单值联系是合适的。

2.1.5 反向联系

正如我们可能希望了解给定电影的所有影星一样,我们也可能希望知道给定影星主演的所有电影。为了把这个信息放入 Star 对象中,我们可以在例 2.2 类 Star 的说明中增加以下一行:

```
relationship Set Movie starredIn;
```

然而,这一行和对 Movie 的类似说明忽略了电影和影星之间联系的一个很重要方面。我们希望如果影星 S 在电影 M 的影星集 stars 中,那么电影 M 就在影星 S 的 starredIn 集合中。我们通过将关键字 inverse 和另一个联系的名字放在每个联系的说明中表示 stars 和 starredIn 两个联系之间的这种关联机制。如果另一个联系在其他某个类中,像通常的情况那样,那么我们通过类名、随后的双冒号 和联系名来引用这个联系。

例 2.4 为了把类 Star 的联系 starredIn 定义成类 Movie 的联系 stars 的反向联系,我们把类 Star 的说明改写成图 2.5 所示的那样。在 4) 行,我们不但看到了联系 starredIn 的说明,而且事实上还有一个反向联系 Movie::stars。因为联系 stars 是在另一类 Movie

对反向联系的要求

作为抽象的设计语言, ODL 要求有反向联系。对这个要求的解释是, 每当有一个路径从一个对象出发到它的相关对象时(例如从电影对象出发到影星对象), 相反方向上的路径(从影星到他们主演的电影)也是可能的。也就是说, 如果有一个影星 Charlton Heston, 我们可以审查所有的电影对象并且检查它们的影星。然后我们可以列出由 Heston 主演的电影。ODL 要求给这个反向的过程赋予一个联系名。

然而, 当我们将 ODL 转换成实际的编程语言(例如嵌入式的 C++ 语言)的说明时, 我们知道将引用只放入电影对象中, 而在影星对象中没有对电影的引用是可能的。于是, ODL 的嵌入式 C++ 允许单向联系。因为我们在这里只关心设计, 而不是实现, 我们将期望每个联系都有反向联系。

中定义的, 因此在它之前有类名(Movie)和双冒号。在引用不同类的特性时通常使用这种表示法。

```
1) interface Star{
2)     attribute string name;
3)     attribute struct Addr
           {string street, string city} address;
4)     relationship Set< Movie> starredIn
           inverse Movie: : stars;
};
```

图 2. 5 表明联系及其反向联系的类 Star

在例 2. 4 中, 一对反向联系各自都把一个对象(一部电影或一个影星)和一个对象集相连。正如我们提到的, 有些联系是一个对象和另一类的单一对象相联系。但反向联系的概念并不改变。作为通用的规则, 如果类 C 的联系 R 把一个或多个对象 y_1, y_2, \dots, y_n 的集合和类 C 的对象 x 相连, 那么 R 的反向联系就把对象 x (可能还有其他的对象)和每个 y_i 相连。

有时, 把从类 C 到某个类 D 的联系 R 形象化地按成对的对象或关系的元组列表表示出来, 会很有帮助。每一对都包括类 C 的一个对象 x 和类 D 的一个相关对象 y。例如:

C	D
x ₁	y ₁
x ₂	y ₂
...	...

如果 R 的类型是 Set D , 那么对于同一个 C 值可能有多个对。如果 R 的类型是 D , 那么对于一个给定的 C 值只能有一个对。

于是, R 的反向联系是具有相反分量的对的集合:

联系类型之间的内涵

我们应该意识到多对一联系是多对多联系的特例,而一对一联系是多对一联系的特例。也就是说,任何多对多联系的有用的特性也适用于多对一联系,而多对一联系的有用的特性也适用于一对一联系。例如,表示多对一联系的数据结构虽然对于多对多联系并不适用,但是对于一对一联系是适用的。

还要注意,如果我们说一个联系 R 是“多对多”的,实际上意味着联系 R 有多对多的自由度。随着 R 的改变,有时它可能成为多对一的联系甚至一对一的联系。同样地,多对一的联系 R 有时可能成为一对一的联系。

D	C
y ₁	x ₁
y ₂	x ₂
...	...

注意,即使 C 和 D 同类,这个规则也成立。有些联系在逻辑上连接自己,比如“child of”,从“人”(“Persons”)的类指向自己。

2.1.6 联系的多重性

联系不论是把一个给定的对象和唯一的相关对象相连,还是把一个对象和多个其他对象相连,都是有重要意义的。在 ODL 中,我们可以通过在联系说明中使用或者不用聚集运算符(例如 Set)来说明这种选择。当我们有一对相反的联系时,有四种可能的选择:联系在一个方向上是唯一的,在两个方向上都是唯一的,在两个方向上都不是唯一的。

我们已经讨论过的 Stars 和 Movie 之间的联系在两个方向上都不是唯一的。也就是说,一部电影一般有多个影星,一个影星也可以出现在几部电影中。下例详细描述了以前的例子,在这里,我们将介绍另一个类,Studio,代表制作电影的制片公司。

例 2.5 在图 2.6 中,有三个类(Movie, Star 和 Studio)的说明。其中前两类已经在例 2.1 和 2.2 中介绍了。Studio 对象有属性 name 和 address;这些出现在 13) 行和 14) 行。注意:地址的类型在这里是字符串,而不是像 10) 行的类 Star 的 address 属性那样使用结构。在不同的类中使用同名而不同类型的属性不是什么错误。

在 7) 行,我们看到一个从电影到制片公司的联系 ownedBy。因为这个联系的类型是 Studio,而不是 Set Studio,因此我们是在说明每部电影都有唯一的所属制片公司。这个联系的反向联系出现在 15) 行。在那里我们看到了从制片公司到电影的联系 owns。这个联系的类型是 Set Movie,表示每个制片公司拥有一个电影集合——可能是 0,可能是 1,也可能是大量的电影。

联系及其反向联系的唯一性要求称为联系的多重性。三种最常见的多重性是:

- 1. 从类 C 到类 D 的多对多联系是指,在联系中,每个 C 都和 D 的集合有关,而在反

向联系中, 每个 D 都和 C 的集合有关。例如, 在图 2. 6 中, stars 是从类 Movie 到类 Star 的多对多联系, 而 starredIn 是从 Star 到 Movie 的多对多联系。在每个方向上, 都允许集合为空; 例如, 一部特定的电影可能没有知名的影星。

2. 从类 C 到类 D 的多对一联系是指, 在联系中, 每个 C 都和唯一的 D 有关, 而在反向联系中, 每个 D 都和 C 的集合有关。在图 2. 6 中, ownedBy 是从 Movie 到 Studio 的多对一的联系。我们说它的反向联系是从类 D 到类 C 的一对多的联系。例如在图 2. 6 中, 联系 owns 是从 Studio 到 Movie 的一对多的联系。

3. 从类 C 到类 D 的一对一联系是指, 在联系中, 每个 C 都和唯一的 D 有关, 而在反向联系中, 每个 D 都和唯一的 C 有关。例如, 假如我们在图 2. 6 中增加一个 President 类, 它代表制片公司的总裁。我们希望每个制片公司只有一个总裁, 而没有人可以是多个制片公司的总裁。这样, 制片公司与其总裁之间的联系在两个方向上都是一对一的。

```
1) interface Movie{
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film {color, blackAndWhite} filmType;
6)     relationship Set< Star> stars
           inverse Star :: starredIn;
7)     relationship Studio ownedby
           inverse Studio :: owns;
8) };
9) interface Star {
10)    attribute string name;
11)    attribute Struct Addr
        {string street, string city } address;
12)    relationship Set< Movie> starredIn
        inverse Movie :: stars;
13) };
14) interface Studio {
15)    attribute string name;
16)    attribute string address;
17)    relationship Set< Movie> owns
        inverse Movie :: ownedBy;
18) };
19) }
```

图 2. 6 ODL 的类及其联系

在我们使用“唯一”或“一个”之类的措辞谈论多对一或一对一时有一个微妙之处。在正常情况下要求这个“唯一”或“一个”的元素是实际存在的。例如, 对于每部电影都确实有一个制片公司, 而对于每个制片公司都确实有一个总裁。然而, 实际上, 所期望的唯一对象也可能并不存在。例如, 制片公司可能暂时没有总裁, 或者我们可能不知道哪一家制片公司拥有某一部电影。

这样, 我们应该允许所期望的有关对象唯一值为空缺的。我们以后将看到在数据库中

集合、包和列表

为了理解集合、包和列表三者之间的不同,要记住集合的元素是无序的,并且每个元素只出现一次。包允许一个元素出现多次,而元素及其出现是无序的。列表允许一个元素出现多次,但是元素的出现是有序的。于是, {1, 2, 1}和{2, 1, 1}是相同的包,而 {1, 2, 1}和{2, 1, 1}是不同的列表。

期望有真实值的地方经常出现“空”(Null)值。例如,在传统的编程术语中,我们会发现一个期望指向单一对象的指针实际上是一个空(nil)指针。在 2.5 节我们将讨论完整性约束的问题,并了解保证预期的唯一对象必须毫无例外地存在的机制。

2.1.7 ODL 中的类型

ODL 给数据库设计者提供一个类型系统,它和在 C 或其他传统的编程语言中的类型系统相似。类型系统是由单独定义的基本类型和某些递归规则构造的,利用基本类型和某些递归规则由较简单的类型构成复杂的类型。在 ODL 中,基本类型包括:

1. 原子类型:整型、浮点型、字符、字符串、布尔型和枚举型。最后一种类型是一个名字列表,它和整型是同义词。我们在图 2.6 的 5) 行看到一个枚举的例子,在效果上就是,把名字 color 和 blackAndWhite 定义成整数 0 和 1 的同义词。

2. 接口类型,例如 Movie 或者 Star,实际上表示结构类型,由属性和该接口的联系所对应的各种分量组成。这些名字代表用以下规则定义的复杂类型,但是我们可以把它们看作是基本类型。

这些基本类型可以通过下列的类型构造符组合成结构类型:

1. 集合。如果 T 是任意类型,那么 Set T 表示类型,其值为所有元素的有限集,而元素类型为 T。使用集合类型构造符的例子出现在图 2.6 的 6), 11), 15) 行。

2. 包。如果 T 是任意类型,那么 Bag T 表示类型,它的值是类型为 T 的元素的包或多重集。包允许一个元素出现多次。例如, {1, 2, 1}是包而不是集合,因为 1 出现了多次。

3. 列表。如果 T 是任意类型,那么 List T 表示类型,它的值是类型为 T 的零个或多个元素的有限列表。作为一种特例,string(字符串)类型是 List char 类型的简化形式。

4. 数组。如果 T 是一种类型而 i 是一个整数,那么 Array T, i 表示类型,是由 i 个类型为 T 的元素组成的数组。例如,Array char, 10 表示长度为十的字符串。

5. 结构。如果 T_1, T_2, \dots, T_n 是类型, F_1, F_2, \dots, F_n 是域名,那么

$$\text{Struct } N \{T_1 F_1, T_2 F_2, \dots, T_n F_n\}$$

表示其名为 N 的类型,它的元素是具有 n 个域的结构。第 i 个域名是 F_i , 类型是 T_i 。例如,图 2.6 10) 行说明一个记录类型,名字是 Addr,具有两个域。两个域都是字符串类型,名字分别为 street 和 city。

前面四种类型——集合、包、列表和数组——统称为聚集(Collection)类型。至于什么类型和属性有关,以及什么类型和联系有关,存在不同的规则。

· 属性的类型首先由原子类型或者域是原子类型的结构组成。然后我们可以随意地

把聚集类型应用于原始的原子类型或结构。

- 联系的类型是接口类型或者是应用于接口类型的聚集类型。

属性的类型不能是接口类型,而联系的类型不能是原子类型,记住这些是很重要的。正是这个差别区分了属性和联系。还要注意,复杂类型构成属性和联系的方式有所不同。属性和联系都允许用任选的聚集类型作为最终的运算符,但是只有属性允许结构类型。

例 2.6 一些可能的属性类型:

- (1) integer(整型)。
- (2) Struct N {string field1, integer field2}。
- (3) List real 。
- (4) Array Struct N {string field1, integer field2} 。

例中(1)是原子类型;(2)是由原子类型组成的结构;(3)是原子类型的聚集;(4)是由原子类型组成的结构的聚集。这是属性类型仅有的四种可能性。

现在,假如接口类型 Movie 和 Star 是可利用的基本类型。那么我们可以构造联系类型,比如 Movie 或 Bag Star 。然而,以下作为联系类型是不合法的:

- (1) Struct N {Movie field1, Star field2}, 联系类型不能涉及结构。
- (2) Set integer , 联系类型不能涉及原子类型。
- (3) Set Array Star , 联系类型不能(属性类型也不能)涉及聚集类型的两个应用。

2.1.8 本节练习

* 练习 2.1.1: 让我们为银行设计一个数据库,包括顾客及其帐户的信息。顾客的信息包括他们的姓名、地址、电话和社会保险号。帐户包括编号、类型(例如,存款,支票)和结余。我们还需要记录拥有帐户的顾客。对这个数据库用 ODL 进行描述。为所有的属性和联系选择适当的类型。

练习 2.1.2: 用下列方法修改你对练习 2.1.1 的设计。只描述变化部分,不用写出完整的新模式。

- (a) 只能列出一个顾客作为某个帐户的拥有者。
- (b) 在(a)的基础上,一个顾客不能有多个帐户。
- (c) 地址分三部分:街道(street)、城市(city)和州(state)。并且,顾客可以列出任意数目的地址和电话号码。
- (d) 顾客可以有任意数目的地址,由(c)中所示的三元组组成,任一地址都和电话集相连。也就是说,我们需要记录每个顾客的地址,而电话将连到每个地址。注意:对属性和联系的类型有不少限制,应小心。

练习 2.1.3: 给出按 ODL 设计的数据库,记录球队、队员和球迷的信息,包括:

1. 对于每个球队,球队的名字、队员、队长(队员之一)以及队服的颜色。
2. 对于每个队员,他/她的姓名。
3. 对于每个球迷,他/她的姓名,喜爱的球队,喜爱的队员以及喜爱的颜色。

练习 2.1.4: 修改练习 2.1.3,为每个队员记录他所服役的球队的历史,包括在每个球队的开始日期和结束日期(如果他们转队)。

* ! 练习 2.1.5: 假如我们想要保存一个家谱。我们应该有一个类, Person。每个人所要记录的信息包括他们的姓名(一个属性)和下列的联系: 母亲、父亲和孩子。对类 Person 进行 ODL 设计。保证指明如: mother、father 和 children 的反向联系仍是 Person 至 Person 的类。mother 的反向联系是联系 children 吗? 为什么是或为什么不是? 把每个联系及其反向联系作为一对集合来描述。

! 练习 2.1.6: 让我们在练习 2.1.5 的设计中增加一个属性“教育”。这个属性的值是每个人所获得的所有学位的聚集, 包括学位的名称(例如, 理学士(B. S., Bachelor of Science))、学校和日期。结构的聚集可以是集合、包、列表或数组。分别描述采用这四种选择的结果。采用每一种方式会得到什么信息, 会失去什么信息? 丢失的信息在实际应用中是否重要?

! 练习 2.1.7: 设计一个适合大学选课的数据库。这个数据库应当包括学生、系、教授、课程、哪个学生选了哪门课、哪个教授教哪门课、学生的分数、课程的助教(由学生兼任)、一个系提供哪些课程以及你认为适当的任何其他信息。注意, 和以上练习相比, 该练习形式上更加自由, 对联系的多重性、适当的类型、甚至所要表示的信息都要由你做出决定。

```
1) interface Ship {
2)     attribute string name;
3)     attribute integer yearLaunched;
4)     relationship TG assignedTo inverse TG::unitsOf;
5) };
6) interface TG {
7)     attribute real number;
8)     attribute string commander;
9)     relationship Set Ship unitsOf
10)         inverse Ship :: assignedTo;
11) };
```

图 2.7 关于舰艇和特遣舰队的 ODL 描述

练习 2.1.8: 图 2.7 是类 Ship 和 TG(Task Group, 特遣舰队, 是舰艇的聚集)的 ODL 定义。我们将对这个定义作一些修改。可以通过指出要修改的行, 并给出替代的内容, 或者通过在已编号的某一行之后插入一行或几行来对每个修改进行描述。描述下列修改:

- (a) 属性 commander(指挥官)的类型变成两个字符串, 其中第一个是军衔, 第二个是姓名。
- (b) 一艘舰艇可以分配给多个特遣舰队。
- (c) 姊妹舰是按相同的计划建造的相同的舰艇。我们希望对每艘舰艇都表示出姊妹舰的集合(除了它本身)。你可以假定每艘舰艇的姊妹舰都是 Ship 的对象。

* !! 练习 2.1.9: 在什么条件下, 一个联系就是它本身的反向联系? 提示: 把联系想象成一个对的集合, 正如 2.1.5 节所讨论的那样。

* ! 练习 2.1.10: 一个类型在任何时候都可以同时是合法的 ODL 的属性类型和联系类型吗? 解释为什么可以或为什么不可以。

E/ R 联系的可视化

用一个表也就是关系表示 E/ R 联系, 用每一行表示参与联系的实体对, 往往是有
益的。例如, 联系 Stars-in 可视化为由如下各对组成的一张表:

Movies(电影)	Stars(影星)
Basic Instinct	Sharon Stone
Total Recall	Arnold Schwarzenegger
Total Recall	Sharon Stone

当然, 无论是在 ODL 中, 还是在 E/ R 模型中, 都没有特定方法使联系一定能
实现。

这个表有时称为该联系所对应的联系集。表的各行就是联系集的成员。行可以表
示成元组, 包含每个参与实体集的分量。例如,

(Basic Instinct, Sharon Stone)

就是联系 Stars-in 所对应的联系集中的一个元组。

2.2 实 体 联 系 图

数据库建模有一种图形方法, 称为实体联系图(entity-relationship diagram), 它和
ODL 的面向对象的设计方法有很大的相似性。实体联系(即 E/ R)图和最初讨论的 ODL
一样具有三个主要的部分(虽然 E/ R 和 ODL 模型都有我们将在以后讨论的附加特性)。
这些部分是:

- 1. 实体集和类相似。实体是实体集的成员, 和 ODL 中的对象类似。
- 2. 属性是描述实体某个特性的值。因此, E/ R 和 ODL 中的属性本质上具有相同的
概念。
- 3. 联系是两个或多个实体集之间的连接。在 E/ R 模型中的联系和 ODL 中的联系十
分类似。然而:
 - (a) 在 E/ R 模型中, 我们对两个方向上的联系用一个名字, 而在 ODL 中, 分别称为
联系和反向联系。例如, 图 2. 6 中的反向联系 Movie::stars 和 Star::starredIn
在 E/ R 模型中将用单一的联系表示。
 - (b) E/ R 模型中的联系可以涉及两个以上的实体集, 而 ODL 的联系最多涉及两个
类。

例 2. 7 图 2. 8 是一个实体联系图, 它表示了和图 2. 6 用 ODL 说明的相同的现实世
界。实体集是 Movies(电影)、Stars(影星)和 Studios(制片公司)。我们应该用复数形式给
实体集命名, 而通常用单数给类命名, 这说明了图 2. 8 和图 2. 6 在命名上的细微差别。

电影的实体集 Movies 和图 2. 6 的类 Movie 具有相同的四个属性: title、year、length

图 2.8 电影数据库的实体联系图

和 filmType。类似地, 另两个实体集具有与它们相应的 ODL 类所说明的 name 和 address 属性。

在图 2.8 中我们也看到 E/R 联系和图 2.6 的 ODL 说明中的联系相对应。一个联系是 Stars-in(主演), 它所体现的信息分别来自 ODL 的类 Movie 和 Star 的一对相反的联系 stars 和 starredIn。图 2.8 中 E/R 联系 Owns(拥有) 代表了 ODL 的两个反向联系 Movie::ownedBy 和 Studio::owns。图 2.8 中指向实体集 Studios 的箭头表明每部电影都由唯一的制片公司所拥有。下面, 我们将讨论有关 E/R 图的多重性问题。

2.2.1 E/R 联系的多重性

正如我们在例 2.7 中所看到的那样, 箭头可以用来表示 E/R 图中联系的多重性。如果一个联系是从 E 到 F 的多对一联系, 那么我们就画一个指向 F 的箭头。这个箭头表示实体集 E 中的每个实体, 最多和实体集 F 中的一个实体有关。然而, F 中的一个实体可以和 E 中的多个实体有关。

遵循这个原则, 实体集 E 和实体集 F 之间一对一的联系通过指向 E 和 F 两个实体的两个箭头来表示。例如, 图 2.9 表示两个实体集 Studios 和 Presidents 以及它们之间的联系 Runs(经营)(把属性省略了)。我们假定一个总裁只能经营一个制片公司, 而一个制片公司也只有一个总裁, 所以这个联系是一对一的, 用两个箭头表示, 每个箭头指向一个实体集。

图 2.9 一对一的联系

2.2.2 联系的多向性

和 ODL 不同, E/R 模型使得定义涉及两个以上实体集的联系更为方便。然而, 实际上, 三元(三向)或更多元的联系是很少的。E/R 图中的多向联系通过从菱形的联系到所涉及的每个实体集之间的连线来表示。

例 2.8 图 2.10 是一个涉及到制片公司、影星和电影的联系 Contracts(签约)。该联

在多向联系中对箭头符号的限制

在有三个或多个参与者的联系中,对其连线上的箭头或非箭头没有足够的选择。因此,我们不可能通过箭头描述每一种可能的情形。例如,在图 2.10 中,制片公司实际上是电影单独的函数,而不是影星和电影共同的函数,因为只要有一个制片公司就能制作电影。然而,我们的符号不能将这种情况和一种三向联系的情况相区别,在这种三向联系中由箭头所指的实体集确实是另外两个实体集的函数。在 3.5 节,我们将提出一种正式的表示法——函数依赖——它有足够的力量来描述所有可能的选择。

系表示一个制片公司和一个特定的影星签约来表演一部特定的电影。一般来说,可以把 E/R 联系的值想象成元组的联系集,而元组的分量就是参与该联系的实体,正如我们在方框“E/R 联系的可视化”中所讨论的那样。这样,联系 Contracts 就可以用三元组的形式来描述:

(studio, star, movie)

图 2.10 三向联系

在多向联系中,指向实体集 E 的箭头意味着如果我们从该联系的每个其他实体集中选择一个实体,那么这些实体将和 E 中唯一的实体相关。(注意,这个定义简单地把我们在双向联系中使用的多重性概念加以推广。)图 2.10 中,有一个指向实体集 Studios 的箭头,表明对于特定的 star 和 movie 来说,该影星为演该电影只和一个制片公司签约。然而,却没有指向实体集 stars 和 movies 的箭头。这意味着,一个制片公司可以为一部电影和几个影星签约,而一个影星可以和一个制片公司签约主演多部电影。

2.2.3 联系中的角色

在一个联系中,一个实体集可能出现两次或多次。如果是这样,那么一个实体集在联系中出现多少次我们就从联系到这个实体集画多少条线。到实体集的每条线代表该实体集所扮演的不同角色。我们把名字标记在实体集和联系之间连线的侧面,并称它为“角色”。

例 2.9 图 2.11 给出了实体集 Movies 和它本身的联系 Sequel-of。每个联系都在两部电影之间,其中一部是另一部的续集。为了区别联系中的两部电影,一条线标记为角色 Original(首集),另一条线标记为角色 Sequel(续集),分别表示电影的首集及其续集。我们假定一部电影可能有多部续集,但是对于每部续集都只有一部首集。因此,该联系是从续集电影到首集电影的多对一联系,正如图 2.11 的 E/R 图中的箭头所表明的那样。

图 2.11 具有角色的联系

例 2.10 我们在较早的例 2.8 中曾引入 Contracts(签约)联系,在图 2.12 又给出了

该联系的更复杂的形式, 把它作为最后一个例子, 它既包括多向联系又包括具有多重角色的实体集。现在, 联系 Contracts 涉及两个制片公司、一个影星和一部电影。意图是一个和某影星已有签约的制片公司可以再和第二个制片公司签约, 让这个影星在特定的电影中扮演角色。这样, 该联系可以通过如下形式的四元组来描述:

(studio1, studio2, star, movie)

这意味着第二个制片公司和第一个制片公司签约, 让第一个制片公司的影星为第二个制片公司演这部电影。

在图 2. 12 中我们看到指向 Studios 的箭头扮演着两个角色, 一个是影星的“所有者”, 一个是电影的作者。然而, 没有指向 Stars 或 Movies 的箭头。理由如下: 给定一个影星、一部电影和一个制作该电影的制片公司, 在这里它只能是“拥有”该影星的唯一的制片公司。(我们假定一个影星正好和一个制片公司签约。) 类似地, 只有一个制片公司制作给定的电影, 所以给定一个影星、一部电影和该影星所在的制片公司, 我们就可以确定

图 2. 12 四向联系

唯一的制作电影的制片公司(Producing studio)。注意, 在两种情况下, 实际上, 我们只需要一个其他的实体就能确定这个唯一的实体——例如, 我们只要知道电影就可以确定唯一的制作电影的制片公司——但是这个事实并不改变多向联系的多重性的规定。

图中没有指向 Stars 或 Movies 的箭头。给定一个影星、该影星所在的制片公司和制作电影的制片公司, 可能有几个不同的签约使得该影星在几部电影中扮演角色。这样, 联系四元组的其他三个分量不能唯一地确定一部电影。类似地, 制作电影的制片公司可能和某个其他的制片公司签约在一部电影中使用多个影星。这样, 联系的其他三个分量就不能确定一个影星。

2. 2. 4 联系中的属性

把属性和联系相连, 而不是和任何一个与联系相连的实体集相连, 有时更为方便。例如, 考虑图 2. 10 中的联系, 它表示影星和制片公司之间为一部电影而签约。我们可能希望记录和这个签约有关的酬金。然而, 我们不能把它和这个影星相连; 因为一个影星可能从几部不同的电影中得到不同的酬金。类似地, 把酬金和一个制片公司相连(制片公司可能对不同的影星支付不同的酬金)或者和一部电影相连(一部电影中不同的影星可能得到不同的酬金)都是没有意义的。

然而, 把酬金和 Contracts 联系所对应的联系集中的三元组

(star, movie, studio)

相连是合适的。由于加上属性, 图 2. 13 看起来比图 2. 10 更加充实。联系有属性 salary(酬金), 而实体集的属性也和图 2. 8 中给出的属性相同。

把属性放在联系上决不是必须的。作为代替, 我们可以建立一个新的实体集, 让它的

实体具有属于联系的属性。然后,如果我们在联系中包含该实体集,那么就可以忽略联系本身的属性。

图 2.13 具有属性的联系

例 2.11 让我们修改图 2.13 中的 E/R 图,该图在 Contract 联系上有属性 salary。作为代替,我们建立一个实体集 Salaries,具有属性 salary。Salaries 成为联系 Contracts 的第四个实体集。图 2.14 中给出了完整的图。

图 2.14 将属性移到实体集

2.2.5 把多向联系转换成二元联系

回忆一下,与 E/R 模型不同,ODL 把我们限制在二元联系中。然而,任何连接多于两个实体集的联系都可以转换成二元、多对一的联系的聚集,而不丢失任何信息。在 E/R 模型中,我们可以引入一个新的实体集,并把它的实体看作是多向联系所对应的联系集的元组。我们称这个实体集为连接实体集。然后,我们引入从连接实体集到为最初的多向联系提供元组分量的每个实体集的多对一联系。如果一个实体集扮演多个角色,那么它将是每个角色所对应的一个联系的目标。

例 2.12 图 2.12 中的四向联系签约 Contracts 可以由也称作签约 Contracts 的实体集来取代。正如在图 2.15 所看到的那样,它参与四个联系。如果联系 Contracts 所对应的联系集有一个四元组

(studio1, studio2, star, movie)

那么实体集 Contracts(签约) 会有一个实体 e。该实体通过联系 Star-of(签约的影星) 与实体集 Stars 中的实体 star 相连。它通过联系 Movie-of(签约的电影) 和 Movies 中的实体 movie 相连。它通过联系 Studio-of-star(拥有影星的制片公司) 和 Producing-studio(制作电影的制片公司) 分别和 Studios 的实体 studio1 和 studio2 相连。

注意, 我们已经假定实体集 Contracts 没有属性, 不过图 2. 15 中的其他实体集有未给出的属性。然而, 向实体集 Contracts 中增加属性, 例如签约日期, 是可能的。

图 2. 15 用实体集和二元联系代替多向联系

在 ODL 中, 我们将用和以上描述的对 E/R 模型的转换相似的方式表示像图 2. 12 那样的多向联系。然而, 由于 ODL 中没有多向联系, 因此转换不是可有可无的, 而是必不可少的。

例 2. 13 假定我们有类 Star, Movie 和 Studio, 对应于图 2. 12 中的三个实体集中的每一个。为了表示四向联系 Contracts, 我们引入一个新的类, 比如 Contract。这个类没有属性, 但是它有四个联系, 对应于 E/R 联系的四个分量。图 2. 16 中给出了 ODL 说明, 图中省略了反向联系。E/R 联系 Contracts 中的每一个四元组对应于 ODL 类 Contract 中的一个对象。

```
interface Contract {
    relationship Studio ownerOfStar;
    relationship Studio producingStudio;
    relationship Star star;
    relationship Movie movie;
};
```

图 2. 16 用 ODL 表示签约

2. 2. 6 本节练习

* 练习 2. 2. 1: 用 E/R 模型表达练习 2. 1. 1 的银行数据库。务必在适当的地方画上箭头,

并表明联系的多向性。

练习 2.2.2: 考虑到练习 2.1.2 中提到的变化, 修改你在练习 2.2.1 对帐户的解答:

- (a) 修改你的图, 使一个帐号只能有一个顾客。
- (b) 进一步修改你的图使一个顾客只能有一个帐号。

! (c) 修改你在练习 2.2.1 中的原始图, 使顾客可以有一个地址集合(它是街道-城市-州的三元组)和一个电话集合。记住, 尽管在有限的情况下 ODL 允许属性为聚集类型, 但是在 E/R 模型中是不允许的。

! (d) 进一步修改你的图, 使顾客能有一个地址集合, 并且在每个地址中有一个电话集合。

练习 2.2.3: 用 E/R 模型表达练习 2.1.3 的球队/队员/球迷数据库。记住, 颜色的集合不是球队的合适的属性类型。你如何避免这种限制?

! 练习 2.2.4: 假定我们希望把两个队员和一个球队之间的联系 Led-by 加到练习 2.2.3 的模式中。意图是这个联系集包括三元组

(player1, player2, team)

以便当某个其他的 player2 是队长时 player1 暂时在球队中比赛。

- (a) 画出对 E/R 图的修改。
- (b) 用一个新的实体集和二元联系替代你的三元联系。

! (c) 你的新的二元联系是否和以前存在的任何联系相同? 注意, 我们假定两个队员是不同的, 也就是, 队长不能领导自己。

练习 2.2.5: 修改练习 2.2.3 中的 E/R 图以包括队员的历史, 像练习 2.1.4 那样。

! 练习 2.2.6: 用 E/R 模型表达练习 2.1.5 和 2.1.6 有关“人”的数据库。模型中包括对于母亲、父亲和孩子的联系; 当一个实体集在一个联系中多次使用时不要忘记标明角色。模型中还包括每个人获得学位的信息, 如练习 2.1.6 所描述的那样。标明每个联系的多重性。你是否需要对于母亲、父亲和孩子的单独的联系? 为什么要或为什么不要?

练习 2.2.7: 表示练习 2.1.5 中的信息的另一条途径是有一个三元联系 Family, 打算让 Family 对应的联系集中的三元组

(person, mother, father)

是某个人、他的母亲和他的父亲; 当然, 所有三个人都在实体集 People 中。

- * (a) 画出这个图(不需要关于教育的信息)。在适当的位置放上箭头。
- (b) 用一个实体集和二元联系替代三元联系 Family。再一次放上箭头来表明联系的多重性。

练习 2.2.8: 用 E/R 模型表达你在练习 2.1.7 设计的大学数据库。

2.3 设计原则

我们还需要学习关于 ODL 或 E/R 模型的许多细节, 但是我们有足够的基础开始研究什么是好的设计和应当避免什么等关键问题。在这一节, 我们将努力阐明和详细描述某些有用的原则。

冗余和反向联系

我们可以认为在 ODL 中使用联系及其反向联系是冗余设计的一个例子。然而, 我们不当假定要表示联系及其反向联系就要用两个不同的数据结构, 例如, 一个方向上的指针和另一个方向上的指针。回忆一下, 联系及其反向联系的定义仅仅反映了这样的事实, 即人们原则上可以在任一方向上遍历一个联系。

然而, 如果我们选择用两个单独的数据结构实现联系, 那么我们确实冒着冗余所带来的危险。因为当数据改变时, 希望底层的指针始终保持不变, 因此基于 ODL 的 DBMS 的实现者必须注意如何进行数据库的更新。但是, 这是系统层的问题。这里有一个假定: 实现者最终将使问题得到正确解决。到那时, 在实现层由于冗余而带来的危险很少, 而在两个方向上都存在指针将导致以很快的速度实现对联系的遍历。

2.3.1 真实性

首先, 设计应当忠于规范。也就是说, 类或实体集和它们的属性应当反映现实。你不能给影星附加一个属性 number-of-cylinders(汽缸数), 尽管它对实体集或类 Automobile(汽车)有意义。如果我们了解要建模的那部分现实世界, 那么无论声明什么样的联系手段都将是有意义的。

例 2.14 如果我们定义一个在 Star 和 Movie 之间的联系 Stars-in, 它将是多对多的联系。原因是对现实世界的观察告诉我们影星可以在多部电影中出现, 而电影也可以有多个影星。将联系 Stars-in 说明成在任一方向上多对一或说明成一对一都是不正确的。

2.3.2 避免冗余

我们应当注意任何事物只表达一次。例如, 我们已经使用了电影和制片公司之间的联系 Owns(拥有)。我们还可以让实体集 Movies 有属性 StudioName。虽然这么做没有什么不合法的, 但由于以下几个原因, 这种做法是危险的:

1. 两次表示“所属的制片公司”这一同样的事实, 比任何一次单独的表示都占用了更多的空间。
2. 如果一部电影被卖掉了, 我们可能通过 Owns 改变与该电影有关的所属制片公司, 却忘记改变该电影的 StudioName 属性值, 反之亦然。当然, 人们可以强调决不应做这样粗心大意的事, 但是实际上, 错误是频繁的, 因为企图用两种不同的方式表达同一个事物, 给我们带来了麻烦。

这些问题将在 3.7 节正式描述, 并且我们也将学习一些工具来重新设计数据库模式, 于是冗余以及随之而来的问题将迎刃而解。

2.3.3 对简单性的考虑

避免在设计中引入过多的元素是绝对必要的。

例 2.15 我们假定存在“电影所有权”, 表示一部电影的所有权(holding), 用来代替

电影和制片公司之间的联系。那么我们可以建立另一个类或实体集 Holdings。可以在每部电影和代表该电影的唯一所有权之间建立一对一的联系 Represents (代表)。从 Holdings 到 Studios 是多对一联系,如图 2.17 所示。

图 2.17 具有多余实体集的不良设计

在技术上,图 2.17 的结构确实代表了现实世界,因为经由 Holdings 从一部电影到它唯一所属的制片公司是可能的。然而,Holdings 没有起有效的作用,而没有它会更好。它使得利用电影-制片公司之间联系的程序更加复杂,浪费空间,而且容易出现错误。

2.3.4 选择合适的元素类型

有时,可以选择用来表示现实世界概念的设计元素的类型。许多这样的选择是在使用属性还是使用类或实体集之间进行的。一般来说,属性比类/实体集或联系实现起来更为简单。然而,让一切事物都成为属性通常会给我们带来麻烦。

例 2.16 让我们考虑一个特殊的问题。在图 2.6 或图 2.8 中,我们使制片公司为类或实体集是否明智?我们是否应该删除制片公司的类或实体集而把制片公司的名字和地址作为电影的属性?这样做带来的一个问题就是对于每部电影都要重复制片公司的地址。这就带来了冗余;除了 2.3.2 节所讨论的冗余的缺点之外,如果一个给定的制片公司不拥有任何一部电影,那么我们将面临失去该制片公司地址的危险。

另一方面,如果我们不记录制片公司的地址,那么把制片公司的名字作为电影的属性并没有任何损害。我们没有地址重复带来的冗余。我们必须对每部为 Disney(迪斯尼)所拥有的电影给出制片公司例如迪斯尼的名字,这种情况不是真正的冗余,因为我们必须以某种方式表示每部电影的拥有者,而给出名字正是一种合理的方式。

一般来说,我们建议,如果某个事物具有比它的名字更多的有关信息,那么可能需要的是实体集或类。然而,如果它为设计提供的只有它的名字,那么作为属性可能更好。这种差别和将在 3.7 节介绍的关系模型的模式“规范化”问题密切相关。

例 2.17 让我们考虑如下的观点:使用一个多向联系和使用具有几个二元联系的连接实体集之间有一个折衷。我们在图 2.12 看到一个影星、一部电影和两个制片公司之间的四向联系 Contracts(签约),我们机械地将它转换成实体集 Contracts。我们选择哪一个是否有影响?

在 ODL 的设计中,我们其实没有选择的余地,因为并没有提供多向联系。在 E/R 模型中,二者均合适。然而,如果我们对这个问题稍加改动,那么几乎会强制我们选择连接实体集。让我们假定签约涉及一个影星、一部电影,以及任何制片公司集合,而不是一个制片公司。这种情况比图 2.12 中的情况更为复杂,在那里有两个制片公司扮演两个角色。在这种情况下,我们会涉及任意数量的制片公司,或许一个做产品,一个搞特技,一个管销售,等等。这样,我们不能对制片公司分配角色。

看来,和联系 Contracts 对应的联系集必须包括如下形式的元组

(star(影星), movie(电影), set-of-studios(制片公司的(集合)))

并且联系 Contracts(签约)本身不仅涉及通常的影星和电影实体集,而且涉及一个其实体为制片公司集合的新的实体集。尽管这种方法是允许的,但是把制片公司的集合作为基本实体看来并不自然,因此我们并不推荐它。

一个更好的方法是将 Contracts(签约)作为实体集。如图 2.15,一个实体 Contracts 连接一个影星、一部电影和一个制片公司的集合,但是现在不必限制制片公司的数量。于是,在签约和制片公司之间的联系是多对多的,而不是多对一的,然而,如果签约是真正的“连接”实体集,那么它将是多对一的。图 2.18 画出了 E/R 图。注意,一个签约只与单个影星和一部电影有关,而与任何数量的制片公司有关。

相同的设计策略对 ODL 也完全适用。例如,以下的 ODL 接口说明:

```
interface Contract {
  relationship Star theStar;
  relationship Movie theMovie;
  relationship Set < Studio> studios;
};
```

图 2.18 连接影星、电影和制片公司集合的签约

是合适的。在这里,签约对象有三个联系,分别到一个影星、一部电影和一个制片公司集合。反向联系省略。

2.3.5 本节练习

* 练习 2.3.1: 图 2.19 是一个涉及顾客和帐户的银行数据库的 ODL 设计。假定各种联系和属性的意义正如给出的名字所期望的那样。对这个设计做出评价。它违背了哪些设计规则?为什么?你建议做哪些修改?

!! 练习 2.3.2: 在本练习和以下的练习中我们将考虑用 E/R 模型描述出生日期的两种设计选择。一个出生涉及到一个婴儿(双生儿将用两个出生表示)、一位母亲、任意数量的护士和任意数量的医生。因此,假定我们有实体集 Babies(婴儿)、Mothers(母亲)、Nurses(护士)和 Doctors(医生)。假定我们还用联系 Births(出生)和这四个实体集相连,如图 2.20 所建议的那样。注意,和 Births 对应的联系集的元素具有如下形式

(baby, mother, nurse, doctor)

如果有多个护士和/或医生给一个婴儿接生,那么将有几个元组有相同的婴儿和母亲,而一个元组对应护士和医生的一种组合。

我们可能希望把某些假设体现在我们的设计中。为了表示每种假设,说出如何把箭头或其他元素加到 E/R 图中。

(a) 对于每个婴儿,有唯一的母亲。

(b) 对于婴儿、护士和医生的每种组合, 有唯一的母亲。

(c) 对于婴儿和母亲的每种组合, 有唯一的医生。

! 练习 2.3.3: 解决练习 2.3.2 的问题的另一种方法是通过实体集 Births 连接四个实体集 Babies(婴儿)、Mothers(母亲)、Nurses(护士)和 Doctors(医生), 共有四个联系, Births 和每个其他实体集之间有一个联系, 如图 2.21 所示。用箭头(指明其中某些联系是多对一的)来表示以下情况:

(a) 每个婴儿是唯一的出生的结果, 并且每个出生属于唯一的婴儿。

(b) 在(a)的条件外, 还要求每个婴儿有唯一的母亲。

(c) 在(a)和(b)的条件外, 还要求对于每个出生有唯一的医生。

```
interface Address {
    attribute string addr;
    relationship Set < Customer> residents
        inverse Customer :: livesAt;
};

interface Customer {
    attribute string name;
    relationship Address livesAt
        inverse Address :: residents;
    relationship AcctSet accounts
        inverse AcctSet :: owner;
};

interface Account {
    attribute real balance;
    relationship Set < AcctSet> memberOf
        inverse AcctSet :: members;
};

interface AcctSet {
    attribute string ownerAddress;
    relationship Customer owner
        inverse Customer :: accounts;
    relationship Set< Account> members
        inverse Account :: memberOf;
};
```

图 2.19 对银行数据库的不良设计

图 2.20 用多向联系表示出生

图 2.21 用实体集表示出生

!! 练习 2.3.4: 假定我们允许一个出生涉及由一个母亲生的多个婴儿。你如何用练习 2.3.2 和 2.3.3 的方法表示每个婴儿仍然有唯一的母亲这一事实?

! 练习 2.3.5: 用 ODL 重做练习 2.3.2 和 2.3.3 的 E/R 设计。你发现这些练习的哪些情况容易实施? 哪些情况不能实施? 如何修改你的设计允许像练习 2.3.4 那样生多个婴儿?

2.4 子 类

类往往还包括某些具有附加特性的对象,而这些特性并不和类的所有成员相关。如果是这样,我们发现一种有效的方法就是将这个类组织成子类,除了作为整体的类的特性以外,每个子类有它自己的附加属性和/或联系。ODL 有一个简单的方法说明子类,我们将在下面予以讨论。然后我们再看看 E/R 模型如何用称为“属于”(“isa”)联系(也就是,“an A is a B”表示子类 A 属于类 B 的“属于”(“isa”)联系)的特殊联系,表示类-子类的层次关系。

2.4.1 ODL 中的子类

可能存储在我们的实例数据库中的各种电影包括卡通片、谋杀片、惊险片、喜剧片以及多种其他特殊类型的电影。对于每种电影类型,我们可以定义一个在例 2.1 中介绍的类 Movie 的子类。我们通过在类 C 说明的类名 C 后面加上冒号和另一个类 D 的名字来定义类 C 是类 D 的子类。

例 2.18 我们可以说明 Cartoon 为 Movie 的子类,于是 Movie 是 Cartoon 的超类,用以下的 ODL 说明:

```
1) interface Cartoon: Movie {
2)     relationship Set< Star> voices;
};
```

其中,1)行说明 Cartoon 为 Movie 的子类。2)行表明所有的 Cartoon 对象都有一个联系 voices,代表为卡通人物配音的人。我们没有指明联系 voices 的反向联系,虽然技术上我们可以这样做。注意,联系 voices 并不是对所有电影都有意义,只有卡通片才有意义,所以我们不想把 voices 作为类 Movie 的联系。

子类继承其超类(也就是子类从中派生的类)的所有特性。也就是说,超类的每个属性或联系自动成为子类的属性或联系。这样,在例 2.18 的每个卡通片对象都有从 Movie 中继承下来的属性 titile, year, length 和 filmType(回忆一下图 2.6),并从 Movie 中继承了联系 stars 和 ownedBy,此外,还有它自己的联系 voices。

2.4.2 在 ODL 中的多重继承

类可以有多个子类,像 2.4.1 节所描述的那样,每个子类从它的超类继承特性。进而,子类本身可能还有子类,形成类的层次,而每个类都继承祖先的特性。一个类也可能有多个超类。下例说明多重超类的隐患和问题。

例 2.19 我们可以为谋杀片定义类 Movie 的另一个子类:

- 1) interface MurderMystery: Movie {
- 2) attribute string weapon;
- };

这样,所有的谋杀片除了所有电影都有的四个属性和两个联系以外,还有一个指明谋杀武器的属性。

现在,考虑像《谁陷害了兔子罗杰?》(Who Framed Roger Rabbit?)这样的电影,它既是卡通片,又是谋杀片。这样的电影除了有普通电影的特性以外还应有联系 voices 和属性 weapon。我们可以通过说明另一个子类 Cartoon-MurderMystery(卡通-谋杀片)来描述这种情况,它是 Cartoon(卡通片)和 MurderMystery(谋杀片)两者的子类。说明如下:

```
interface Cartoon-MurderMystery: Cartoon, MurderMystery {};
```

这样的定义就使 Cartoon-MurderMystery 对象具有 Cartoon 和 MurderMystery 两个子类的所有特性。我们没有说明任何只属于 Cartoon-MurderMystery 的属性或联系。类 Cartoon-MurderMystery 的对象从类 MurderMystery 中继承属性 weapon(武器)并从 Cartoon 中继承联系 voices(配音)。另外,因为类 MurderMystery 和 Cartoon 都从类 Movie 中继承了四个属性和两个联系,因此类 Cartoon-MurderMystery 也继承了这六个特性。然而,类 Cartoon-MurderMystery 并没有继承这六个特性的两个副本;而是经由它的两个直接超类中的任何一个从 Movie 中继承这些特性。图 2.22 说明了涉及这四个类的子类-超类联系。

通常,我们可以在接口名 C 的说明之后加上冒号和任意数量的其他类的列表来说明类 C 为这些其他类的子类。例 2.19 的 Cartoon-MurderMystery 的说明就是这种格式的一个实例。当类 C 从几个类中继承时,特性名之间有发生冲突的隐患。类 C 的两个或多个超类可能有同名的属性或联系,而这些特性的类型可能不同。

图 2.22 表示多重继承的图

例 2.20 假定我们有称为 Romance(言情片)和 Courtroom(刑侦片)的类 Movie 的子类。进而假定这两个子类中每个都有称为 ending(结局)的属性。在类 Romance 中,属性结局从枚举类型{喜剧,悲剧}中取值,而在类 Courtroom 中,属性结局从枚举类型{有罪,无罪}中取值。如果我们再建立一个子类, Courtroom-Romance, 它把 Romance 和 Courtroom 两者都作为超类,那么在类 Courtroom-Romance 中继承的属性结局其类型含糊不清。

虽然 ODL 本身并不定义特定的语法,但是 ODL 的实现将至少提供以下机制之一来使用户能够规定如何处理由于多重继承而产生的冲突:

1. 指出特性的两个定义中哪一个用于子类。例如,在例 2.20 中,我们可以决定在类刑侦-言情片中,与其说关心法庭的审判结果不如说更关心电影的结局是喜剧还是悲剧。在这种情况下,我们将规定类 Courtroom-Romance 从超类 Romance 中,而不是从超类 Courtroom 中,继承属性 ending(结局)。

2. 在类 C 中,对于有相同名字的另一个特性给一个新的名字。例如,在例 2.20 中,如果 Courtroom-Romance 从超类 Romance 中继承属性 ending,那么我们可以规定类

Courtroom-Romance 有一个称为 verdict(判决)的附加属性,这是把从类 Courtroom 继承的属性 ending 改名了。

3. 为类 C 重新定义在它的一个或多个超类中已定义的某些特性。例如,在例 2. 20 中,我们可以决定属性 ending 不直接从任何一个超类中继承。相反,对于刑侦-言情片,我们重新定义属性结局(ending)为整数,表示从观众投票结果得出的对电影结局的满意程度。

注意,即使在例 2. 19 中也有冲突: Cartoon-MurderMystery 从每个直接的超类 (Cartoon 和 MurderMystery)中继承所有的六个特性,例如 title(名称)和 stars(影星),而这些特性是这两个类从类 Movie 中继承下来的。然而,因为名称和其他特性的定义在超类 Cartoon 和 MurderMystery 中是相同的,因此无论决定用哪种定义都是可以接受的。

2. 4. 3 实体联系图中的子类

回忆一下,ODL 中的类和 E/R 模型中的实体集类似。假定类 C 是类 D 的子类。为了在 E/R 模型中表达这个概念,我们通过称为“属于”(isa)的特殊联系使对应于类 C 和 D 的两个实体集相连。我们为实体集 C 和 D 画普通的方框。任何只和实体 C 有关的属性或联系都连到方框 C 上。用于 C 和 D 两者的属性则放在 D 上。

Isa 联系用两条边连同中间一个三角形来表示。三角形的顶点应指向超类。专用词“isa”可随意地放在三角形中。

例 2. 21 实体集 Movie 及其两个子类 Cartoon 和 MurderMystery 如图 2. 23 所示。该子类结构和例 2. 19 用 ODL 表示的子类结构类似。标有 isa 的两个三角形分别从 Cartoon 和 MurderMystery 指向超类 Movies。我们没有给出把 Movies 与 Stars 和 Studios 相连的联系 Stars-in 和 Owns,我们只提供了连接 Cartoon 和 stars 的联系 Voices(配音)。

图 2. 23 E/R 图中的属于(isa)联系

2. 4. 4 E/R 模型中的继承

ODL 或其他面向对象的模型和 E/R 模型在继承的概念上有细微的差别。在 ODL 中,对象必须恰好是一个类的成员。于是,在例 2. 19 我们需要定义类 Cartoon-MurderMystery 包含那些既是卡通片又是谋杀片的对象。例如,我们不能把对象“兔子罗杰”既放在卡通片类中又放在谋杀片类中。

在 E/R 模型中,我们将认为一个实体具有属于几个实体集的分量,而这几个实体集是单一的属于(isa)层次的组成部分。Isa 联系把各分量连成单一的实体。这个实体有它的所有分量的全部属性,并参与它的分量所参与的所有联系。

通常,这个观点的效果和 ODL 的效果相同,因为特性的继承性给对象的属性和联系,将和对应于该对象的实体从其分量中汇集的属性和联系相同。然而,有一个差别,我们将在例 2.22 中讨论。在 3.4 节,当我们将 ODL 和 E/R 设计转换成关系模型时,我们将看到另一个差别。

例 2.22 注意,在图 2.23 中我们不需要和卡通-谋杀片相对应的实体集。原因是,这样一个实体,如兔子罗杰,有属于所有三个实体集 Movies(电影)、Cartoons(卡通片)和 MurderMystery(谋杀片)的分量。通过 isa(属于)联系三个分量连成一个实体。同时,这些分量把 Movies 的所有四个属性(以及图 2.23 中未画出的 Movies 的两个联系)都给了实体兔子罗杰,加上实体集 MurderMystery 的属性 weapon(武器)和实体集 Cartoon 的联系 Voices(配音)。这些恰好是例 2.19 中的对象兔子罗杰所在的类 Cartoon-MurderMystery 中的特性,这些特性是从它的超类 Movie, Cartoon 和 MurderMystery 中继承下来的。

然而,注意,如果有某些特性属于卡通-谋杀片,但是既不属于卡通片也不属于谋杀片,那么我们就需要在图 2.23 中有第四个实体集 Cartoon-MurderMystery,而这些特性(属性和联系)将连向这个实体集。那么,实体兔子罗杰将有第四个分量,该分量属于 Cartoon-MurderMystery 并为实体兔子罗杰提供这些新的特性。

2.4.5 本节练习

* 练习 2.4.1: 让我们考虑军舰的数据库,并用 ODL 加以描述。每艘军舰有下列与之相关的信息:

- 1. 它的名称。
- 2. 它的排水量(重量),以吨为单位。
- 3. 它的类型,如战列舰、驱逐舰。

另外,有下列具有某些其他信息的特殊类型的舰艇:

- 1. 炮舰是携带大型火炮的舰艇,例如战列舰或巡洋舰。对于这类舰艇,我们希望记录主炮的数量和口径。
- 2. 航空母舰携带飞机。对于航空母舰,我们希望记录飞行甲板的长度和分派给它们的航空大队的集合。
- 3. 潜艇可以在水下航行。对于潜艇,我们希望记录它们的最大安全深度。你可以假定不是炮舰或航空母舰就是潜艇。
- 4. 攻击型航空母舰既是炮舰又是航空母舰,并具有和二者相关的所有信息。

回答下列问题:

可能有人会感到奇怪,确实存在这样的东西。Ise 和 Hyuga 是两艘日本的战列舰,1943 年对其进行改装使其具有一个飞行甲板和覆盖其后半部分的飞机库。

(a) 对于各类的这种层次结构给出 ODL 设计。

(b) 说明如何表示攻击型航空母舰 Ise。它有排水量 36 000 吨, 有 8 门 14 英寸的火炮, 有 200 英尺长的飞行甲板并携带“1 和 2”两个航空大队。

!! 练习 2.4.2: 对于某些子类, 例如练习 2.4.1 中的攻击型航空母舰, 只有一种可能的类型, 而对于其他子类, 例如炮舰, 却有几种类型, 例如, 战列舰和巡洋舰。这种情况是否产生某种形式的冗余? 如果是这样, 如何消除这种冗余?

* 练习 2.4.3: 用 E/R 模型重复练习 2.4.1。

! 练习 2.4.4: 修改你在练习 2.15 中对“people”数据库的设计以便包括下列特殊类型的人:

1. 女人。
2. 男人。
3. 作父母的人。

你可能也想区别某些其他类型的人, 于是用联系连接人的适当的子类。给出你的设计, 用:

(a) ODL

(b) E/R 模型

2.5 对约束的建模

现在, 我们已经看到如何使用 ODL 的类及其特性(包括属性和联系)或者使用 E/R 模型中的实体集和联系为一部分现实世界建模。我们在建模过程中感兴趣的许多结构都可以用这两种表示法中的任意一种来表示。然而, 现实世界中还有其他重要的方面不能用前面学过的工具建模。这种附加信息往往对数据具有约束的性质, 它超出了由类、属性和联系的定义而在结构和类型方面产生的约束。

以下是对常用约束的粗略分类。这里并不包括所有的约束类型。关于约束的其他内容可以在 4.5 节关系代数和第 6 章 SQL 编程中找到。

1. 键码(keys)是在类的范围内唯一标识一个对象或者在实体集的范围内唯一标识一个实体的属性或属性集。一个类中的两个对象在构成键码的属性集上取值不能相同。

2. 单值约束(single-value constraints)要求某个角色的值是唯一的。键码是单值约束的一个主要来源, 因为它们要求和键码属性值有关的类或实体集的其他属性值是唯一的。然而, 单值约束还有其他来源, 正如我们将要看到的那样。

3. 参照完整性约束(reference integrity constraints)要求由某个对象引用的值在数据库中确实存在。参照完整性类似于传统程序中对悬挂指针的禁止。

4. 域的约束(domain constraints)要求属性值必须取自特定值的集合或者处于特定的范围内。我们将在 6.3 节中涉及对 SQL 的域约束。

5. 一般约束(general constraints)是要求在数据库中保存的任意的断言。例如, 我们可能会要求对任何一部电影列出的影星不超过十个。我们将在 4.5 和 6.4 节看到一般约束表达语言。

约束是模式的一部分

我们可能将数据库看作只在某段时间存在,并且因为没有两个对象在某个属性有相同的值就错误地决定该属性构成键码。例如,当我们建立我们的电影数据库时,我们可能没有同时输入两部同名的电影。然而,如果我们在这个初步证据的基础上做出 title 是类 Movie 的键码的决定,并且按 title 为键码这样的假设为我们的数据库设计存储结构,那么我们将会发现我们自己不能向数据库中输入第二部 King Kong 电影。

因此,键码约束和一般约束都是数据库模式的一部分。数据库设计者把约束随着结构设计(例如,实体和联系)一起说明。一旦说明了某种约束,就不允许对数据库进行违背该约束的插入或修改。

因此,虽然数据库的一个特例可以满足某些约束,但是只有适用于作为现实世界正确模型的数据库的所有实例而由设计者标识的约束,才是“真正的”约束。这样的约束可以由用户假定并用于数据库的存储结构。

有几方面的约束是重要的。那些约束告诉我们为之建模的现实世界的有关方面的结构信息。例如,键码使用户能毫无混淆地识别对象或实体。如果我们知道属性 name 是类 Studio 的对象的键码,那么当我们通过制片公司的名字引用它的对象时,我们就知道我们在引用唯一的对象。另外,知道存在唯一值就可节省空间和时间,因为存储一个值比存储一个集合更容易,即使集合只有一个成员。参照完整性和键码还支持允许快速访问某个对象的某些存储结构。

2.5.1 键码

在 ODL 中,类的键码是一个或多个属性的集合 K ,于是对于该类中给定的任何两个不同的对象 O_1 和 O_2 ,都有与 O_1 和 O_2 的键码 K 对应的两组不同的属性值。在 E/R 模型中,键码完全相同,只是“实体集”代替了“类”,“实体”代替了“对象”。

例 2.23 让我们考虑例 2.1 的类 Movie(电影)。人们可能首先假定属性 title(片名)本身是键码。然而,有些片名已经被两个或多个的不同的电影重复使用了,例如,King Kong。因此,将 title 本身说明为键码是不明智的。如果这样做了,那么我们将不能在数据库中包括两部名为 King Kong 的电影的信息。

一个更好的选择将是选择两个属性 title 和 year 的集合作为键码。我们将冒着有两部同年制作的同名电影的风险(这样两部不能都存储在我们的数据库中),但这是不大可能的。

对于其他两个类,Star 和 Studio,在 2.1 节介绍过,我们必须再一次认真考虑什么可以作为键码。对于制片公司,合理的假定是不会有二个电影制片公司重名,所以我们将把 name 作为 Studio 类的键码。然而,用影星的名字唯一标识影星却容易混淆。一般而言名字并不能用于区别人。然而,传统上影星都随意地选择“艺名”,因此我们可能希望看到

注意一下类似的情况,C 程序中,即使链表只含有一个整数,表示整数也比表示整数的链表更简单。

name 也作为影星类的键码。否则, 我们可以选择一对属性 name 和 address 作为键码, 这将是令人满意的, 除非两个影星同名且住在相同的地址。

例 2. 24 根据例 2. 23 的经验可能使我们认为找出键码或者确信属性的集合构成键码是困难的。实际上, 情况通常非常简单。为设计数据库而对现实世界的环境建模时, 人们往往要在环境范围以外为重要的类建立有效键码。例如, 公司一般给所有雇员都分配雇员号, 并且认真选择使这些标识号为唯一的号码。这些标识号的一个意图是确保在公司数据库中每个雇员都可以和所有其他雇员相区别, 即使有几个雇员的名字是一样的。这样, 雇员号属性就可以作为数据库中雇员的键码。

在美国公司中, 通常每个雇员还有一个社会保险号。如果数据库有一个属性是社会保险号, 那么该属性也可作为雇员的键码。注意, 一个类的键码可有几种选择并没有错, 正如雇员类可有“雇员号”和社会保险号两个键码。

建立一个属性的目的就是作为键码, 这种想法非常普遍。除了雇员号以外, 我们看到的在大学用学号区别学生。我们看到驾驶员许可证号和机动车登记号在机动车部门分别区分驾驶员和机动车。毫无疑问, 读者可以找到许多这种属性的例子, 而建立这种属性的主要目的就是作为键码。

2. 5. 2 在 ODL 中说明键码

在 ODL 中用关键字 key 或 keys(用哪一个无关紧要)并用构成键码的属性跟在后面, 来说明一个或多个属性为一个类的键码。如果在键码中有多个属性, 那么属性表必须用括号括起来。键码说明必须紧跟在接口说明之后, 在左花括号或者任何属性或联系之前说明。说明本身用圆括号括起来。

例 2. 25 为了说明两个属性 title 和 year 的集合构成类 Movie 的键码, 我们可以用如下内容代替图 2. 6 的 1) 行:

```
interface Movie
    (key (title, year))
{
```

即使只说明一个键码, 我们也可以用 keys 代替 key。

类似地, 如果名字是类 Star 的键码, 那么我们可以在图 2. 6 的 8) 行花括号之前增加 (key name)

属性的几个集合有可能均为键码。如果是这样, 那么可以在字 key(s) 之后, 放几个用逗号分开的键码。和通常一样, 包含多个属性的键码必须用括号把其属性表括起来, 这样我们就可以把几个属性构成的一个键码和几个单一属性的键码区别开了。

例 2. 26 作为有多个键码的例子, 考虑一下类 Employee, 我们不在这里描述属性和联系的完整集合。然而, 假定它的两个属性是 empID(雇员号)和 ssNo(社会保险号)。那么, 我们可以用

```
(key empID, ssNo)
```

说明这些属性中的每一个均为键码。因为属性表没用括号括起来, ODL 把上述说明解释为两个属性中的每一个均为键码。如果我们用括号把列表(empID, ssNo)括起来, 那么

ODL 将解释成两个属性一起构成一个键码。也就是说,如下写法

```
(key (empID, ssNo))
```

的含意是不会有二个雇员既有相同的雇员号,又有相同的社会保险号,不过二个雇员可能在这两个属性之一取值一致。

2.5.3 在 E/R 模型中表示键码

实体集在本质上就是类,它可以有和 ODL 的类意义上完全相同的键码。如果一个属性集构成一个实体集的键码,那么在该实体集中将不会有这样的两个实体,它们的键码的每个属性值都一致。在 E/R 图表示法中,我们在属于实体集的键码属性下划线,例如,图 2.24 表示图 2.8 的实体集 Movies,属性 title 和 year 一起作为键码。

在有多键码的情况下,在 E/R 模型中,不提供正式的表示法来表明所有的键码。通常把一个键码称为主键码,并把该属性集看作好像是实体集唯一的键码。而在 E/R 模型中,主键码用下划线标明,其他键码,称为次键码,将不予标明或者附在图旁的注释中。

图 2.24 标出键码的电影实体集

一个不常见但是可能的情况是,一个实体集的键码不属于实体集本身。我们将推迟到 2.6 节考虑这种称为“弱实体集”的情况。

2.5.4 单值约束

通常,数据库设计的一个重要特性是最多有一个值扮演特定角色。例如,我们假定一个电影对象有唯一的名称、年份、长度和电影类型,并且一部电影属于唯一的制片公司。用 ODL 说明这些假定并不困难,因为每个属性有一个类型。如果类型不是一个聚集类型(例如,集合),那么可能对于该属性只有一个值或者对于一个联系只有一个相关的对象。另一方面,如果把属性或联系定义为聚集类型,例如图 2.6 中 6) 行的联系 stars 的类型为 Set Star,那么就允许一部给定的电影和多个影星相关。这样的联系称为多值联系。

我们还必须区别一个属性或联系最多有一个值的情况和必须正好有一个值的情况。当一个联系把一个类中的对象和另一个类中的单一对象相连,并要求后一个对象存在时,则存在一个称为“参照完整性”的约束,我们将在下面的 2.5.5 节讨论。当一个属性为单值时,我们有两种选择:

- 1. 可以要求该属性值存在。
- 2. 可以允许该属性值任选。

如果一个属性构成一个类的部分键码,那么我们一般要求每个对象中都存在该属性值。对于其他属性,当属性值不存在时,我们可以对该属性建立“null”值,来代替实际值。那么,该属性的非空值将是任选的。

例 2.27 对于类 Movie,我们在例 2.23 中已确定它的键码是 title 和 year。我们要求这两个属性在所有电影对象中都存在。另一方面,属性 length 可以有选择地省略。例如,我们可以用-1 作为 length 的空值,因为电影的长度不会为负。如果我们不知道电影的长

度,我们将把属性 length 的值设置为- 1。类似地,我们为定义属性 filmType 的各种可能值而增加第三个枚举值。除了值 color 和 blackAndWhite,我们可以选择一个值例如 Null 或 Unknown 以表明没有关于电影类型的信息。

实体/联系模型也提供了表示单值约束的方法。实体集的每个属性都隐含地有单一的值。通常我们假定,一个值可能为空。不能为空的属性将在边上标明。

表明多对一或一对一联系的箭头也表示单值约束。也就是说,如果一个联系有指向实体集 E 的箭头,那么最多有实体集 E 中的一个实体与从其他相关的每个实体集中选择的一个实体相连。

2.5.5 参照完整性

单值约束断言对一个给定的角色最多有一个值存在,而参照完整性约束断言对于这个角色正好有一个值存在。我们看到的约束是,属性有非空的单值,把这种约束作为参照完整性的一种要求,但是“参照完整性”更多地用于引用类间的联系。

让我们考虑图 2.6 的 ODL 说明中 7) 行从 Movie 到 Studio 的联系 ownedBy。人们可能会问,有一个制片公司对象作为 ownedBy 的值,而那个制片公司对象却不存在,这怎么可能呢? 答案是,在 ODL 的实现中,联系 ownedBy 将通过指针或对制片公司对象的引用来表示,而且有时可能从类 Studio 中删除制片公司对象。在这种情况下,指针成为悬挂(dangling)状态;它将不再指向实际的对象。

对联系 ownedBy 的参照完整性约束将要求所引用的制片公司对象必须存在。有几种方式来实施这种约束。

我们可以禁止删除要引用的对象(如例子中的制片公司)。

我们可以要求如果删除要引用的对象,那么也要删除引用它的所有对象。在我们的例子中,这种方法要求,如果删除一个制片公司,那么也要从数据库中删除所有属于该制片公司的电影。

除了关于删除的这两个原则之外,我们还要求,当建立电影对象时,把一个存在的制片公司对象给该电影对象作为它的联系 ownedBy 的值。进而,如果该联系的值发生了变化,那么新值也必须是一个存在的对象。实施这些原则以保证联系的参照完整性是数据库实现的一个内容,而在这里我们不讨论具体细节。

2.5.6 E/R 图中的参照完整性

我们可以扩展 E/R 图中的箭头表示法,来表示是否要求联系在一个或多个方向上支持参照完整性。假定 R 是一个从实体集 E 到实体集 F 的联系。我们将用一个指向 F 的圆箭头来表明不仅联系 R 是由 E 到 F 的多对一或一对一的联系,而且对于实体集 E 的一个给定实体要求,存在与之相关的实体集 F 的实体。当 R 是在多个实体集之间的联系时这个概念也同样有效。

例 2.28 图 2.25 表示实体集 Movies, Studios 和 Presidents 之间的某些恰当的参照完整性约束。这些实体集和联系最初在图 2.8 和 2.9 中介绍了。我们看到一个圆箭头从联系 Owns 进入 Studios。该箭头所表示的参照完整性约束就是,拥有一部电影的制片公

司必须总是存在于 Studios 实体集中。

类似地, 我们看到一个圆箭头从 Runs 进入 Studios 中。该箭头表示的参照完整性约束就是, 如果一个总裁经营一个制片公司, 那么该制片公司存在于 Studios 实体集中。

图 2. 25 表示参照完整性约束的 E/R 图

注意, 从 Runs 至 Presidents 的箭头仍为尖箭头。这种选择反映了制片公司及其总裁之间合理的假设。如果一个制片公司不再存在, 那么它的总裁也不再称为(制片公司)总裁, 于是我们将期望把该制片公司的总裁从 Presidents 实体集中删除。因此指向 Studios 的是圆箭头。另一方面, 如果把一个总裁从数据库中删除掉, 制片公司将继续存在。于是, 我们用普通的尖箭头指向 Presidents, 表明每个制片公司最多有一个总裁, 但有时可能没有总裁。

2. 5. 7 其他类型的约束

正如本节开始提到的那样, 人们可能还希望在数据库中实施其他类型的约束。在这里我们将对其他约束作简短的说明, 在第 6 章将对这个课题作详细阐述。

域约束把属性的值限制在一个有限的集合内。ODL 需要每个属性的类型, 而这个类型就是域约束的初级形式。例如, 如果属性长度是整型, 那么长度的值就不能是 101. 5 或任何其他非整数。然而, ODL 不支持更进一步的限制性约束, 例如长度在 60 和 240 之间。我们将在 6. 3 节看到 SQL 支持这种约束。

还有更多普通类型的约束不属于本节提到的任何一个范畴。例如, 关于联系度的约束, 比如一个电影对象或实体不能通过联系 stars 与十个以上的影星对象或实体相连。在 E/R 模型中, 我们可以把一个极限数附在联系和实体集的连线旁边, 以表明限制与有关实体集的任何一个实体相连的实体数。在 ODL 中, 我们可以让 Movie 的联系 stars 为长度等于 10 的数组类型, 从而限制影星的数量。然而, 没有方法规定一个集合最多有十个元素。

例 2. 29 图 2. 26 表明在 E/R 模型中如何表示这样的约束: 没有一部电影的影星数大于 10。作为另一个例子, 我们可以把箭头看作约束“ = 1 ”的同义词, 并且我们可以把图 2. 25 中的圆箭头看作表示约束“ = 1 ”。

图 2. 26 表示每部电影影星数的约束

2. 5. 8 本节练习

练习 2. 5. 1: 选择并说明你在下列练习中所做的 ODL 设计的键码:

- * (a) 练习 2. 1. 1。
- (b) 练习 2. 1. 3。
- * (c) 练习 2. 1. 5。
- (d) 练习 2. 4. 1。

练习 2.5.2: 对于你在下列练习中所做的 E/R 图:

- * (a) 练习 2.2.1。
- (b) 练习 2.2.3。
- (c) 练习 2.2.6。
- (d) 练习 2.4.3。
- (i) 选择并说明键码; (ii) 指出适当的参照完整性约束。

! 练习 2.5.3: 我们可以把 E/R 模型中的联系想象成具有键码, 正如实体集那样。令 R 为实体集 E_1, E_2, \dots, E_n 之间的联系, 则 R 的键码是选自 E_1, E_2, \dots, E_n 的属性集 K, 从而如果 (e_1, e_2, \dots, e_n) 和 (f_1, f_2, \dots, f_n) 是 R 的联系集中的两个不同的元组, 那么这两个元组不可能在 K 的所有属性上都一致。现在, 假定 $n=2$; 也就是说, R 是一个二元联系。另外, 对于每个 i, 令 K_i 是作为实体集 E_i 的键码的属性集。按如下假设, 利用 E_1 和 E_2 , 给出 R 可能的最小键码:

- (a) R 是多对多联系。
- * (b) R 是从 E_1 到 E_2 的多对一联系。
- (c) R 是从 E_2 到 E_1 的多对一联系。
- (d) R 是一对一联系。

!! 练习 2.5.4: 再考虑练习 2.5.3 的问题, 但是 n 可以为任何数, 不只是 2。仅用从 R 到 E_i 的哪条弧线有箭头这一信息, 说明如何利用 K_i 找出 R 可能的最小键码 K 的方法。

! 练习 2.5.5: 从现实生活中给出建立属性的主要目的是作为键码的其他例子(例 2.24 除外)

2.6 弱实体集

有一个奇特而又似乎合理的情况, 即组成一个实体集键码的属性中的一些或全部属于另一个实体集。这样的实体集称为弱实体集(weak entity set)。

2.6.1 产生弱实体集的原因

有两个主要的弱实体集的来源。第一, 有时实体集属于一种层次结构。如果实体集 E 的实体是实体集 F 的实体中的子单元, 那么只有把 E 的实体所从属的 F 的实体名考虑在内, E 的实体名才是唯一的。

例 2.30 引入弱实体集的层次结构的某些实例如下:

1. 一个电影制片公司可能有几个电影组(Crew), 电影组可能由给定的制片公司命名为 1 组、2 组, 等等。然而, 其他制片公司可能对电影组使用相同的名字, 所以属性 number (组号) 并不是电影组的键码。相反, 为了唯一地命名一个电影组, 我们需要同时给出它所属的制片公司的名字和组号, 如图 2.27 所示。弱实体集 Crew 的键码是它自己的 number 属性和通过多对一的联系 unit-of 与电影组相关的唯一制片公司的 name 属性。

双边菱形和双边矩形将在 2.6.3 节予以说明。

为什么在 ODL 中没有“弱类”？

在 ODL 或任何面向对象的模型中从未出现如何找出键码的问题。正如我们在 2.5.2 节看到的, 我们可以说明由一个或几个属性构成键码, 但是我们不必这样做。对象有“对象标识”(object identity), 实际上是可以找到对象的地址, 并且对象标识能唯一地区别对象, 即使它们的属性值或联系不能区别两个对象。另一方面, E/R 模型是“面向值”的, 实体仅仅通过它们的相关属性值相区别。因此, 在 E/R 设计中我们必须注意, 任何实体集的实体只用值就可以互相区别, 而不求助于任何“对象标识”。

2. 物种通过它的属名和种名来命名。例如, 人属于人类(Homo Sapiens)种; Homo 是属名, Sapiens 是种名。通常, 一个属包括几个物种, 每个物种的名字开头是属名, 随后是种名。令人遗憾的是, 种名不是唯一的。两个或多个属可能有种名相同的物种。于是, 为了唯一地命名一个物种, 我们同时需要种名和与该物种相关的属名, 而该物种则通过联系 Member-of 与它的属相连。物种是一种弱实体集, 它的键码部分来自包括它的属。

图 2.27 弱实体集及其连接

弱实体集的第二个常见的来源是连接实体集, 我们已在 2.2.5 节作为消除多向联系的方法介绍过了。这类实体集往往没有它们自己的属性。它们的键码由它们所连接的实体集的键码属性构成。

图 2.28 连接实体集是弱实体集

注意, 在 E/R 模型中没有消除多向联系的要求, 不过对于 ODL 以及某些较旧的模型, 例如, 在 2.7 节讨论的网状模型和层次模型, 我们必须替代多向联系。

例 2.31 在图 2.28 我们看到代替例 2.8 的三元联系 Contracts(签约)的连接实体集 Contracts。Contracts 有一个属性 salary(酬金),但是这个属性不属于键码。相反地,签约的键码包括制片公司的名字和所涉及的影星,加上所涉及的电影名称和年份。

2.6.2 对弱实体集的要求

我们不能不加选择地得到弱实体集的键码属性。相反,如果 E 是一个弱实体集,那么提供一个或多个 E 的键码属性的每个实体集 F 必须通过联系 R 和 E 相连。此外,还必须满足下列条件:

- 1. R 必须是从 E 到 F 的二元的多对一的联系。
- 2. F 为 E 的键码提供的属性必须是 F 的键码属性。
- 3. 然而,如果 F 本身是弱实体集,那么 F 提供给 E 的键码属性可能是通过多对一的联系与 F 相关的某个实体集的属性。
- 4. 如果有几个从 E 到 F 的多对一联系,那么每个联系都可能用于提供 F 的键码属性的一个副本来帮助构成 E 的键码属性。注意,E 的实体 e 可能通过来自 E 的不同联系和 F 的不同实体相关。这样,F 的几个不同实体的键码可能出现在标识 E 的特定实体 e 的键码值中。

为什么需要这些条件,直觉的原因如下。考虑一下弱实体集中的一个实体,比如例 2.30 中的 Crew(电影组)。理论上,每个组是唯一的。即使两个组有相同的序号,但由于属于不同的制片公司,原则上我们可以把它们区别开。只依靠有关电影组的数据来区别各个组是很困难的,因为只有组号还不够。我们可以将附加信息和组相连接的唯一方法是,假定有某种决定性过程,它所产生的附加值使组的命名唯一。找到和一个组实体相连的唯一的唯一方法是,假设或者

- 1. 该值是 Crew 实体集的属性,或者
- 2. 我们可以采用从一个 Crew 实体到某个其他实体集的唯一实体的联系,并且该其他实体有某种类型的唯一相关值。也就是说,所采用的联系必须是到另一个实体集 F 的多对一(或一对一,作为特例)联系,并且该相关值必须是 F 的键码。

2.6.3 弱实体集表示法

我们应当采用下列惯例指明实体集是弱的并说明它的键码属性。

- 1. 如果一个实体集是弱的,将用双边矩形表示。图 2.27 中的 Crews 和图 2.28 中的 Contracts 就是这个惯例的实例。
- 2. 如果一个实体集是弱的,那么连接它将和提供其键码属性的其他实体集的多对一联系将用双边菱形表示。图 2.27 中的 Unit-of 和图 2.28 中的所有三个联系都是这个惯例的实例。
- 3. 如果一个实体集为它本身的键码提供任何属性,那么这些属性将加下划线。图 2.27 给出一个实例,Crew 的序号参与了它本身的键码,不过它不是 Crew 的全部键码。

记住,一对一联系是多对一联系的一个特例。当我们说联系必须是多对一时,也总是包括一对一联系。

我们可以将惯例概括如下:

当看到一个实体集有双边时,它就是弱的。它的键码包括其属性中带下划线的部分(如果有的话),及通过具有双边的多对一联系与弱实体集相连的实体集的键码属性。

2.6.4 本节练习

* 练习 2.6.1: 一种表示学生及其各门课成绩的方法是使用与学生、课程以及“注册”相对应的实体集。“注册”实体构成学生和课程之间的“连接”实体集,它不仅可用于表示一个学生修了某门课程这一事实,而且可用于表示该学生该课的成绩。针对这种情况画一个 E/R 图,表示出弱实体集以及实体集的键码。成绩是实体集“注册”键码的一部分吗?

练习 2.6.2: 修改练习 2.6.1 以便记录学生一门课中几次作业的成绩。再次指出弱实体集和键码。

练习 2.6.3: 指出你在练习 2.3.3 设计的 E/R 图中的弱实体集和键码。

练习 2.6.4: 为涉及弱实体集的下列情况画出 E/R 图。在每种情况下指出实体集的键码。

(a) 实体集 Courses(课程)和 Departments(系),一门课由唯一的系提供,而它仅有的属性是课程号。不同的系可以提供具有相同号码的课程。每个系都有唯一的系名。

* ! (b) 实体集 Leagues(社团)、Teams(球队)和 Players(队员)。社团的名字是唯一的。一个社团不会有同名的球队。一个球队不会有同名的队员。然而,在不同的球队可以有同名的队员,并且在不同的社团可以有同名的球队。

2.7 历史上有影响的模型

在这一节我们将向读者介绍另外两种模型和它们的一些术语。“网状”和“层次”两种模型是早期为数据库系统提供基础所做的尝试。它们用于第一批商业数据库系统,开始日期为 20 世纪 60 年代后期和 70 年代。它们由基于关系模型的系统取而代之,关系模型将是第 3 章的主题。然而,它们的几个概念都注入到更新的面向对象的数据库设计方法中。

2.7.1 网状模型

我们可以把网状模型看成只限于二元的多对一联系的 E/R 模型。网状模型的两个主要元素是:

1. 逻辑记录类型。与实体集类似;它们包括类型名和属性表。逻辑记录类型的成员称为记录;它们与 E/R 模型中的实体类似。

2. 链接。是多对一的二元联系。它们连接两个实体集;其中一个是 owner(系主)类型,另一个是 member(成员)类型。链接是从成员类型到系主类型的多对一联系。也就是说,成员类型的每个记录都正好分配给系主类型的记录,而系主类型的每个记录可“拥有”0, 1 或多个成员类型的记录。

例 2.32 让我们用网状模型实现关于电影、影星和他们主演的电影的实例。每个影星和电影都构成一个逻辑记录类型。然而,在电影和影星之间的“主演”(“stars-in”)联系是多对多联系,所以我们不能在网状模型中用单个的链接表示这种联系。相反,我们必须

建立一个新的逻辑记录类型,我们将称其为 StarsIn(主演),作为“连接”逻辑记录类型,与我们在 2.25 节介绍的连接实体集类似。把每个 StarsIn 记录看成是代表一个影星-电影对,于是影星就出现在电影中。因此按网状模型设计的三个逻辑记录类型为:

```
Stars(name, address)
Movies(title, year, length, filmType)
StarsIn ()
```

Stars(影星)逻辑记录类型有两个属性:影星的姓名和地址,而 Movies(电影)类型的属性有关于电影的名称、年份、长度和电影类型等。我们并未表示和电影有关的制片公司,不过一个完整的设计应通过一个链接表示它们之间相连。连接类型 StarsIn 在我们的设计中没有属性,不过如果合适,可以给予属性。例如,如果我们要记录演一部特定的电影给影星酬金的数量,那么这个数量就是电影-影星对的函数,并且是 StarsIn 的属性。然而,在这个例子中,StarsIn 记录的效果仅仅通过它所参与的链接体现出来。

在我们的设计中有两个链接。一个链接是从 Stars 到 StarsIn;也就是说,Stars 是系主类型而 StarsIn 是成员类型。这个链接,我们称之为 TheStar,连接一个影星和该影星参与的各个电影-影星对。第二个链接是 TheMovie,有系主类型 Movies 和成员类型 StarsIn。每个电影记录都拥有包括该电影在内的电影-影星对。注意,两个链接都是多对一的。电影 m 的 Movies 记录和影星 s 的 Stars 记录都拥有一个 StarsIn 对(m, s)。

图 2.29 说明了三个逻辑记录类型的记录如何通过链接连在一起。这个图不是模式。相反,它表明个别的记录本身,并表明通过链接把它们与其他记录相连的方式。我们看到三个 StarsIn 记录。1 号表示 Sharon Stone 是电影 Basic Instinct 的影星这一事实。2 号表示 Sharon Stone/Total Recall 影星/电影对,而 3 号表示 Arnold Schwarzenegger/Total Recall 影星/电影对。这些序号实际上并不是这些记录的一部分,而是使我们便于引用 StarsIn 记录。注意,StarsIn 记录在两个链接中都是成员,而其他记录均为系主。

图 2.29 记录及其链接

2.7.2 网状模式的表示

在表示逻辑记录类型和链接的图中,我们一般用椭圆表示逻辑记录类型,用命名的箭头表示链接。箭头从成员类型指向系主类型。

例 2.33 例 2.32 中的三个逻辑记录类型和两个链接的模式图在图 2.30 中给出。

为什么采用层次模型

人们可能会想为什么层次模型的奇怪的要求一度是数据库产业中的重要推动力。最简单的论点就是用层次模型组织数据(仅当绝对必须时使用虚拟记录类型),可以通过把记录和它们的双亲群集成组的方式把数据存储在顺序文件中。这样,我们可以如图 2.32 所示意的那样存储数据,即一个像 Sharon Stone 那样的记录,其后有它“拥有的”所有记录,在这种情况下,虚拟电影指针就出现在该记录之下。在人们倾向于沿树向下查询信息的假设下,可能倾向于在附近的文件中找到所需信息,如从双亲记录移向子女记录,因此就减少了从磁盘上检索所需信息的时间。

在这个例子中,采用层次模型没有什么好处,因为沿着用虚拟记录所代表的指针将把我们带到磁盘上存放电影信息的某个随机的位置。然而,除了多对多联系以外,由于采用层次体系结构,在查询执行的效率上,往往会有显著地提高。

图 2.30 电影实例的网状模式

2.7.3 层次模型

可以把层次模型看成是一种受限制的网状模型,在这里逻辑记录类型和链接构成一个森林(树的聚集)。也就是说,如果我们把每个链接看作是在表明:系主类型是成员类型的双亲,那么逻辑记录类型就构成一个森林。这个要求的问题在于,对于某些网状模式要实现或许是不可能的。例如,我们从图 2.30 看到逻辑记录类型 StarsIn 在层次模式中需要两个双亲,Stars 和 Movies。因此,图 2.30 不是一个森林。

回忆一下,逻辑记录类型 StarsIn 确实是对应于影星和电影之间多对多联系的连接类型。在层次模型中,我们通过为每个相关类型建立虚拟副本表示多对多联系。我们可以把虚拟类型看作是代表指向实际类型记录的指针。虚拟类型使任何网状模式都能表示为层次模式。

例 2.34 图 2.31 表示电影-影星实例的层次模式。在森林中有两个简单树。第一个根为 Stars(影星),子女为虚拟电影,而第二个根为 Movies(电影),子女为虚拟影星。

我们可以把图 2.31 的模式所代表的实际数据形象化为图 2.32 那样。在该模式中,Stars 类型有子女虚拟电影。于是,类型 Stars 的每个记录有类型为虚拟电影的子女;虚拟记录用其中有单词 to 的方框表示。

图 2.31 电影实例的层次模式

例如,我们看到影星记录 Sharon Stone 有两个子女;每个都是指向 Movies 记录的指针,在这种情况下是指向 Basic Instinct 和 Total Recall 的记录。要理解影星和电影之间基本的多对多联系,我们可以从一个影星记录(例如 Sharon Stone)出发走下去,从那里到子

女虚拟电影记录,再从其中的每一个到相应的实际电影记录。

图 2.32 用层次模型表示电影/影星的实例

2.7.4 本节练习

练习 2.7.1: 用网状模型对下列练习描述的内容进行设计:

- (a) 练习 2.1.1。
- (b) 练习 2.1.3。
- (c) 练习 2.1.5。
- (d) 练习 2.3.2。

练习 2.7.2: 用层次模型重复练习 2.7.1。

* ! 练习 2.7.3: 假定一个实体-联系图有 n 个实体集和 m 个二元联系。如果我们将这个图转换为一个网状模型设计,可能需要的最大和最小的链接数是多少? 记住,每个联系都可以是多对多、多对一或一对一的。

!! 练习 2.7.4: 假定一个实体-联系图有 n 个实体集和 m 个二元联系。如果这个图用层次模型给出,那么请给出需要的最大和最小的虚拟记录类型数。

练习 2.7.5: 如果联系是 k 元的($k > 2$),那么应对你在练习 2.7.3 和 2.7.4 的回答作何修改?

2.8 本章总结

设计表示法: 数据库设计往往使用实体/联系模型或者面向对象的模型如对象定义语言(ODL, Object Definition Language)实施。人们的意图是把 E/R 模型转换为实际的数据库系统模型,通常是关系模型。ODL 设计可以用同样的方法处理面向对象的数据库系统或者(几乎)可以作为对面向对象数据库系统的直接输入。

对象定义语言: 在这种语言中我们通过给出类的属性、联系和方法描述对象的类。属性通过它们的数据类型来描述。ODL 的类型系统包括传统的基本类型,例

如整型以及具有记录结构、集合、包、列表和数组形式的结构类型。联系通过和它们相连的类描述,并允许是单值或者多值。

实体/联系图:在 E/R 模型中,我们描述实体集、它们的属性以及实体集之间的联系。实体集的成员称为实体。我们分别使用矩形、菱形和椭圆描绘实体、联系和属性。

联系的多重性:在 ODL 或者 E/R 中,通过联系的多重性来区别联系是有用的。二元联系可以是一对一、多对一或多对多的联系。在 E/R 中允许多于两个实体集(类)之间的联系,但在 ODL 中不允许。

弱实体集:在 E/R 模型中产生的一个偶然的情况是弱实体集,它要求某些相关实体集的属性以识别它自己的实体。用双边的菱形和矩形的特殊表示法来区别弱实体集。

好的设计:为了有效地设计数据库,要求我们需要选定的表示法(例如 ODL 或 E/R),使用适当的元素(例如,联系、属性),如实地表示现实世界,而且要求避免冗余——重复描述同一个事物或者用一种间接的或过于复杂的方式描述某个事物。

子类:ODL 和 E/R 两者都支持一种方法来描述类或实体集的特殊情况。ODL 有子类和继承,而 E/R 使用一个特殊的联系 `isa` 属于来表示一个实体集是另一个的特殊情况这一事实。

键码:ODL 和 E/R 两者都允许用属性集来说明键码,这意味着该属性集的值唯一地定义一个对象或实体。ODL 还有对象标识符的概念,该标识符是唯一标识对象的值,然而用户并不能访问它。

网状模型:这种模型现在很少使用。它与所有联系全都局限于二元和多对一的 E/R 图类似。

层次模型:这种模型现在也很少出现。它和将实体集排成一个森林并且只有从双亲到子女的多对一联系的 E/R 模型相似。

2.9 本章参考文献

[3]是关于实体/联系模型的原始论文。文献[4]和[1]广泛覆盖实体-联系设计的内容,也包括某些其他有助于设计的模型。

[2]是定义 ODL 的手册。这是对象数据管理组(ODMG, Object Data Management Group)正在进行的工作。该组织通过其电子邮件地址 `info@odmg.org` 和网页: `http://www.odmg.org` 提供关于 ODL 更新资料的电子访问。

- [1] Batini, C., S. Ceri, and S. B. Navathe, Conceptual Database Design, Benjamin/Cummings, Redwood City, CA, 1992.
- [2] Cattell, R. G. G (ed.), The Object Database Standard: ODMG-93 Release 1. 2, Morgan-Kaufmann, San Francisco, 1996.
- [3] Chen, P. P., The entity-relationship model: toward a unified view of data, ACM Trans. on Database Systems, 1: 1, pp. 9 ~ 36, 1976.
- [4] El Masri, R. and S. B. Navathe, Fundamentals of Database Systems, Benjamin Cummings, Menlo Park, 1994.

第 3 章 关系数据模型

我们在第 2 章讨论过两种数据建模的方法: 面向对象和实体-联系模型, 它们很有用, 适合描述数据结构。但今天的数据库实现通常总是以另一种模型——关系模型为基础。关系模型是非常有用的, 因为它只有单一的数据建模概念: 关系, 以二维表的形式排列数据。我们将在第 5 章介绍关系模型如何支持称为 SQL(结构化查询语言)的很高级的编程语言。使用 SQL, 您可以用简单的程序, 针对关系中存储的数据完成功能强大的操作。

反之, 用第 2 章学到的模型之一进行数据库设计, 通常比较容易。所以, 我们的第一个目标是了解如何把 ODL 或 E/R 模型描述的数据库设计转换成关系模型。然后, 我们会看到, 关系模型有它自己的设计理论。这种理论通常称为关系的“规范化”, 基本上以“函数依赖”为基础。函数依赖包括并扩展了我们在 2.5.1 节提到的“键码”的概念。使用规范化理论, 在设计特定的数据库时, 常常能改进对关系的选择。

3.1 关系模型的基本概念

关系模型使我们能以单一的方式来表示数据: 即以称为“关系”的二维表来表示数据。图 3.1 是一个关系的实例。其关系名是 Movie, 包含了图 2.4 所示的例 2.1 中用 ODL 简单定义的 Movie 类的信息, 我们在图 3.2 中又一次给出了该类的 ODL 定义。注意, 这并不是 Movie 类的最终定义, 该定义只包含了四个属性 title, year, length 和 filmType。

title	year	length	filmType
Star Wars	1977	124	color
Mighty Ducks	1991	104	color
Wayne's World	1992	95	color

图 3.1 关系 Movie

```
1) interface Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film ( color, blackAndWhite ) filmType;
};
```

图 3.2 类 Movie 的 ODL 描述

3.1.1 属性

关系的首行称为“属性”(attribute);图 3.1 中,属性是 title、year、length 和 filmType。关系的属性就是关系中各列的名字。通常,属性描述了所在列各项的含义。例如,属性为 length 的列存放以“分钟”计算的各种电影的长度。

注意,图 3.1 中,关系 Movie 的属性与图 3.2 ODL 定义中称为“属性”(attribute)的结构元素对应。将类的属性作为它所对应的关系的属性,这种做法是很常见的。但一般而言,并不要求关系的属性对应于用 ODL 或 E/R 描述的数据的任何特定分量。

3.1.2 模式

关系名和关系的属性集称为关系的“模式”。我们用括号把属性集括起来,并把关系名写在括号的前面来表示关系的模式。于是,图 3.1 中的关系 Movie 的模式就是

Movie (title, year, length, filmType)

请记住:关系模式中的属性是一个集合,而不是列表,为了便于讨论,我们通常需要为属性规定“标准”顺序。这样,无论何时,要引用具有一组属性的关系模式,不管是要显示该关系还是显示它的部分行,都使用这个标准顺序。

在关系模型中,数据库设计包含一个或多个关系模式。所设计的关系模式的集合称为“关系数据库模式”,或简称为“数据库模式”。

3.1.3 元组

除了由属性组成的标题栏以外,关系的各行称为“元组”。元组的各分量分别对应于关系的各个属性。例如,图 3.1 中的三个元组中的第一个,有四个分量 Star Wars、1977、124 和 color,分别对应于四个属性 title、year、length 和 filmType。当我们希望单独地、而不是作为关系的一部分写出一个元组时,一般用逗号把各分量分开,并用括号将整个元组括起来。例如,

(Star Wars, 1977, 124, color)

就是图 3.1 中的第一个元组。注意,一个元组单独出现时,属性并不伴随元组出现,因此,必须以某种方式指明该元组所属的关系,并且始终应该使用关系模式中列出的属性顺序。

通常,可以认为元组表示对象,而元组所属的关系则表示对象的类。上例中的关系 Movie 的确就是这种情形:每个元组表示一个电影对象。这些元组的分量和图 3.2 中描述的电影对象的特性是等同的。但我们应该意识到对象有同一性,而元组没有。也就是说,原则上,电影用面向对象表示法可能出现两个不同的电影对象的所有属性值完全相同的情况,尽管我们在例 2.23 中已经讨论过,对于电影,我们并不希望出现这种情况。

而关系是元组的集合,在给定的关系中一个元组不可能出现一次以上。因此,如果用关系来表示一类对象,我们必须保证该关系有足够的属性集,从而使任何两个对象的各属性值不会完全相同。对于电影,我们在例 2.23 中要求两部电影的名称和年份不能都相同。然而,在最坏的情况下,我们也许需要建立一个属性作为对象的人为标识。例如,我们可以给每部电影唯一的整数“电影标识”(movie ID),并将 movieID 加入到关系 Movie 的属性集中。

元组的形式表示法

虽然我们在本书中将元组表示成列表, 其中每个关系的属性顺序是已知的, 但是元组还有一种允许我们回避固定属性顺序的形式表示法。可以认为元组是一个函数, 该函数反映了从关系模式的各个属性到该元组对应于这些属性的各个分量值的对应关系。比如, 例 3. 1 中用两种方法表示的元组可以看作如下函数:

```
title    Star Wars
year     1977
length   124
fileType color
```

3. 1. 4 域

关系模型要求每个元组的每个分量都是原子的, 即必须属于某种基本类型, 如整型或字符串型。不允许一个值为记录结构、集合、列表、数组或者能合理地分解为更小分量的其他任何类型。这种要求是造成 ODL 中的属性不能直接转换成关系中的单个属性的原因之一。例如, 如果某 ODL 属性 name 的类型为

```
Struct Name{string first, string last }
```

那么, 对应的关系中必须用两个属性 first 和 last 与之对应。我们将在 3. 2. 2 节更详细地讨论这个问题。

此外, 假设与关系的每个属性相关的特定的基本类型称为“域”, 那么关系的任何元组的每个分量都必须在对列的域中取值。例如, 图 3. 1 中, 关系 Movie 的元组的第一个分量必须是字符串, 第二个和第三个分量必须是整数, 第四个分量必须在常量 color 和 blackAndWhite 中取一个。

3. 1. 5 关系的等价表示法

我们已经学过, 关系的模式和元组都是集合, 不是列表。因此, 它们的顺序是无关紧要的。例如, 我们可以将图 3. 1 中的三个元组以它们的六种可能顺序的任一种排列, 而得到的关系都与图 3. 1 中的一样。

而且, 我们可以按选定的顺序对关系的属性重新排列, 而关系并不改变。然而, 当我们对关系模式重新排序时, 我们必须用心, 记住属性是列的标题。因此, 当改变属性的顺序时, 也应该改变列的顺序。当列移动的时候, 元组的分量也改变了顺序。结果是每个元组分量的排列都与属性的排列一致。

例如, 图 3. 3 给出了通过改变行和列的顺序从图 3. 1 可能得到的许多关系中的一个。这两个关系可以看作是“相同的”。更确切地说, 这两张表是同一关系的不同表示法。

注意, 重排属性和列以一种独立的形式对元组产生影响。

例 3. 1 图 3. 1 中的元组

(Star Wars, 1977, 124, color)

模式和实例

我们不要忘记关系模式和关系实例之间的重要区别。模式是关系名和关系的属性集,相对比较稳定;而实例是关系中元组的集合,可能在频繁地变化。

模式/实例的区别在数据建模中是常见的。例如,在第 2 章中,我们介绍了定义类的结构的 ODL 接口定义和该类中的对象集之间的区别。接口定义类似于模式,而它所定义的类的对象集则为实例。类似地,对实体集和联系的描述是描述模式的 E/R 模型方法,而实体集和联系集则构成 E/R 模式的实例。然而,请记住,设计数据库时,数据库实例并不是设计的组成部分。当我们进行设计时,只是想象典型的实例可能是什么样子的。

和图 3.3 中的元组

(1977, Star Wars, color, 124)

表示同一个对象。然而,我们只有知道相应关系的属性顺序,才能确定它们的等价性。因此,为关系选择一个属性顺序并在对关系进行操作的时间内保持该顺序,这种做法通常是明智的。

year	title	filmType	length
1992	Wayne's World	color	95
1991	Mighty Ducks	color	104
1977	Star Wars	color	124

图 3.3 关系 Movie 的另一种表示

3.1.6 关系实例

与电影有关的关系不是静止的,而是随时间变化的。我们希望这些变化反映在关系的元组中。例如,当数据库要增加电影时,就插入新的元组;当得到电影的修订或更正信息时,就修改现有的元组;也可能由于某种原因,有的电影要从数据库中清除掉,这时就要删除相应的元组。

关系模式的变化就不那么常见了,但我们想增加或删除属性的情形也是有的。模式的变化,可能会发生在商用的数据库系统中,但其代价是很昂贵的。因为,可能要改写上百万元组中的每一个——增加或删除一些分量。如果我们增加一个属性,要找到元组的新分量的正确取值是困难的,甚至是不可能的。

我们将给定关系中元组的集合称为该关系的“实例”。例如,图 3.1 中的三个元组就构成了关系 Movie 的一个实例。可以假定,关系 Movie 已经随时间发生了变化,并将继续随时间变化下去。比如,在 1980 年,Movie 中并不包括 Mighty Ducks 或 Wayne' s World 元组。然而,传统的数据库系统只维护任何关系的一个版本:“现在”该关系中元组的集合。关系的这个实例称为“当前实例”。

3. 1. 7 本节练习

练习 3. 1. 1: 图 3. 4 中是两个关系实例, 它们可能是某银行数据库的一部分。请写出:

- (a) 每个关系的属性;
- (b) 每个关系的元组;
- (c) 从每个关系选一个元组, 写出它的分量;
- (d) 每个关系的关系模式;
- (e) 数据库模式;
- (f) 每个属性相适应的域;
- (g) 每个关系的另一种等价表示法。

练习 3. 1. 2: 如果某关系的实例满足下列条件之一, 要表示该实例, 有多少种不同的方法 (考虑元组的顺序和属性的顺序):

- * (a) 3 个属性, 3 个元组, 如同图 3. 4 中的关系 Accounts;
- (b) 4 个属性, 5 个元组;
- (c) n 个属性, m 个元组。

accNo	type	balance	firstName	lastName	idNo	account
12345	savings	12000	Robbie	Banks	901-222	12345
23456	checking	1000	Lena	Hand	805-333	23456
34567	savings	25	Lena	Hand	805-333	34567

关系 Accounts

关系 Customers

图 3. 4 银行数据库的两个关系

3. 2 从 ODL 设计到关系设计

让我们来考虑新的数据库(比如我们的电影数据库)建立的过程。我们从设计阶段开始。设计阶段中, 我们回答下述问题: 存储什么信息, 信息元素如何相互联系, 有哪些假设的约束(比如键码或参照完整性)等等。这个阶段可能会延续很长时间, 在此期间, 评估各种方案, 协调不同意见。

设计阶段之后, 是利用实际的数据库系统进行实现的阶段。因为大多数商用数据库系统都使用关系模型, 我们就假定设计阶段也是使用这种模型, 而不是用我们在第 2 章讨论的面向对象的 ODL 模型或 E/R 模型。

然而, 实际上, 首先用第 2 章所讲的模型之一进行设计, 然后转换成关系模型, 这种做法往往更容易。这么做的主要原因是关系模型中只有一个概念——关系——而没有其他辅助概念(如 E/R 模型中的实体集和联系), 因而存在一些不灵活的地方, 而这些问题最好是在设计方案选好之后再进行处理。

在本节, 我们将考虑如何把 ODL 设计转换成关系设计。3. 3 节将讨论从 E/R 模型到关系模型的转换。3. 4 节将研究子类的问题。因为 ODL 和 E/R 模型对子类(属于联系)

枚举和日期的表示

ODL 有一些原子类型——尤其是枚举和日期——在标准的关系模型中不能直接表示。但这些类型并不是本质问题。例如,枚举实际上是前几个整数的别名列表。因此,对应于一周七天的 ODL 枚举类型可以用整型的属性来表示,不过只使用 0 到 6。另外,也可以用字符串类型的属性来表示,用字符串“ Mon ”,“ Tue ”等来表示每一天。同样,ODL 中的日期在关系模型中可以用字符串型的属性来表示。当我们在第 5 章讨论关系查询语言 SQL 时,将会发现这种语言支持枚举或日期类型的属性,就像 ODL 一样。

的处理有所不同,所以它们向关系的转换也稍有不同。

我们常把约束看作是数据库模式的一部分。ODL 或 E/R 模型中的约束,比如键码约束和参照完整性约束,也能在关系模型中表示。关系模型中一类重要的约束,称为“函数依赖”,将在 3.5 节讨论。关于关系的其他类型的约束将在 4.5 节开始研究。

3.2.1 从 ODL 属性到关系属性

作为起点,假定我们的目标是为每个类建立一个对应的关系,而类的每个特性对应于该关系的一个属性。我们将会看到这种方法在很多方面必须修改,但目前,我们先考虑最简单的可能情况,即我们确实能把类转换成关系,把类的特性转换为关系的属性。我们假设的限制条件是:

1. 类的所有特性都是属性,而不是联系或方法;
2. 属性的类型是原子的,而不是结构或集合。

例 3.2 图 3.2 是满足上述条件的类的例子。它有四个属性,除此之外,没有别的特性。每个属性都是原子类型的: title 是字符串, year 和 length 是整数, filmType 是二值的枚举类型。

我们用与类名相同的名字建立一个关系,这里是 Movie。该关系包括四个属性,每个对应于类的一个属性。关系的属性名可以和相应类的属性名相同。因此,该关系的模式和 3.1.1 节的一样:

Movie (title, year, length, filmType)

该类的对象就是在类的四个属性上各取一个值。我们可以用该对象组成一个元组,只要用每个属性的值作为元组的一个分量即可。我们已经见到过这种转换的结果。图 3.1 就是一些 Movie 对象转换成元组的例子。

3.2.2 类中的非原子属性

很遗憾,即使一个类的特性都是属性,在类向关系的转换过程中,我们也还可能遇到某种困难。原因是 ODL 中的属性可能有复杂数据类型,比如结构、集合、多集或列表。反之,关系模型的一个基本原则是关系的属性具有原子类型,比如数和字符串。因此,我们必须找到某种将非原子类型转换成关系的方法。

有关数据质量的注意事项

尽管我们努力使实例数据尽可能准确,但对影星的地址和其他个人信息还是使用了假数据,以便保护演艺界成员的隐私权,因为他们中的许多人都是躲避宣传媒体的害羞的人。

记录结构的域本身都是原子类型,所以是最容易转换的。我们简单地扩展结构的定义,使结构的每个域对应于关系的一个属性就可以了。唯一可能有麻烦的是,两个结构也许有相同的域名。这种情形下,我们必须在关系中使用新的属性名,以区别它们。

```
interface Star {
    attribute string name;
    attribute Struct Addr
        { string street, string city } address;
}
```

图 3.5 属性为结构的类

例 3.3 考虑一下例 2.3 中的类 Star 的初步定义,图 3.5 复制了该定义。属性 name 是原子的,但属性 address 是具有两个域 street 和 city 的结构。因此,我们用具有三个属性的关系来表示该类。第一个属性 name 对应于同名的 ODL 属性。第二和第三个属性,称为 street 和 city,分别对应于 adress 结构的两个域,一起表示一个地址。于是,我们的关系模式为:

```
Star ( name, street, city )
```

图 3.6 中给出了这个关系可能有的一些元组构成的一个实例。

然而,记录结构并不是 ODL 类定义中可能出现的最复杂的属性类型。属性值也可以用集合(Set)、包(Bag)、数组(Array)和列表(List)等类型构造符构造出来。当移植到关系模型时,每个都会带来它自己的问题。我们将只详细讨论最常见的集合构造符。

name	street	city
Carrie Fisher	123 Maple St.	Hollywood
Mark Hamill	456 Oak Rd.	Brentwood
Harrison Ford	789 Palm Dr.	Beverly Hills

图 3.6 表示影星的关系

如果属性 A 是多个值的集合,一种表示它的方法就是针对每个值建立一个元组。除了 A 之外,该元组还包括所有其他属性的合适的取值。首先让我们来看一个例子,可以看出这种方法很有效,然后,我们将会看到这种方法的缺陷。

例 3.4 假设类 Star 已有定义,这样我们就可以为每个影星登记一个地址集。该类的 ODL 定义如图 3.7 所示。下面假设 Carrie Fisher 还有一个海滨住所,但图 3.6 提到的另外两个影星只有一个住所。然后,我们可以建立两个元组,其名字均为“Carrie Fisher”,如图 3.8 所示。其他元组保持与图 3.6 相同。

原子值: 缺点还是优点

关系模型看起来似乎在我们前进的路上设置了障碍, 因为, ODL 更加灵活, 允许用结构化的值作为特性。人们可能会彻底抛弃关系模型, 或者把它作为一种已经为更高级的“面向对象”方法(比如 ODL)所取代的早期概念。然而, 事实是基于关系模型的数据库系统在市场上占主导地位。一个原因是关系模型的简单性使得用于查询数据库的精巧而功能强大的编程语言成为可能。我们将在 4.1 和 4.2 节介绍抽象编程语言——关系代数和 Datalog(数据逻辑)。也许更重要的是它们在当前的数据库系统中最流行的标准语言 SQL 中的具体化。

```
interface Star {
    attribute string name;
    attribute Set <
        Struct Addr { string street, string city }
    > address;
};
```

图 3.7 有地址集的影星

name	street	city
Carrie Fisher	123 Maple St.	Hollywood
Carrie Fisher	5 Locust Ln.	Malibu
Mark Hamill	456 Oak Rd.	Brentwood
Harrison Ford	789 Palm Dr.	Beverly Hills

图 3.8 允许有地址集

但是读者应该认识到, 这种将集合扩展为几个元组的技术有时可能会导致一个糟糕的关系设计。在 3.7 节, 我们将考虑可能出现的问题, 也将学习如何重新设计数据库模式。现在, 让我们只考虑一个可能会带来问题的例子。

例 3.5 假设我们将出生日期 birthdate 作为属性加到 Star 类的定义中; 也就是说, 我们使用图 3.9 所示的定义。我们在图 3.7 的基础上增加了 ODL 的原子类型之一 Date 类型的属性 birthdate。该属性可以是关系 Star 的一个属性, 这时的关系模式变成:

```
Star ( name, street, city, birthdate )
```

让我们再来修改图 3.8 的数据。既然地址集可以为空, 我们就假定数据库中没有 Harrison Ford 的地址。修订后的关系如图 3.10 所示。

图 3.10 中, 出现了两个问题。首先, Carrie Fisher 的出生日期在相关的元组中重复出现。因此该关系中就有冗余信息。注意, 她的名字也是重复的, 但这种重复不是真的冗余信息, 因为如果该名字不在各元组中出现, 我们就不知道两个地址都与 Carrie Fisher 相关。区别在于影星的名字是由关系所表示的对象的“键码”, 因此, 应该在表示该影星的每个元组中都出现。相反, 出生日期是有关影星的数据, 而不是所表示的对象的键码, 因此,

它的重复出现是冗余的。

```
interface Star {
    attribute string name;
    attribute Set <
        Struct Addr { string street, string city }
    > address;
    attribute Date birthdate;
};
```

图 3.9 具有地址集和出生日期的影星

name	street	City	birthdate
Carrie Fisher	123 Maple St.	Hollywood	9/ 9/ 99
Carrie Fisher	5 Locust Ln.	Malibu	9/ 9/ 99
Mark Hamill	456 Oak Rd.	Brentwood	8/ 8/ 88

图 3.10 增加出生日期

第二个问题是由于 Harrison Ford 的地址集为空,我们丢失了关于他的所有信息。特别是,即使 Ford 的出生日期出现在与他对应的 Star 对象中,他的出生日期也不是关系的一部分。我们再次提醒读者记住,这两个问题对于从 ODL 模式到关系模式的转换方法来说并不是不可避免的。然而,我们必须意识到这些问题,并用 3.7 节描述的“规范化”方法调整关系模式。

当一个类的某些属性是聚集类型(称为多值属性)时,表示单个对象所需的元组可能有多。我们要为多值属性取值的每种组合建立一个元组。我们将在 3.2.5 节从聚集类型的联系的角度讨论这个问题。

3.2.3 其他类型构造符的表示

除了记录结构和集合,ODL 类定义还可以用包(Bag)、数组(Array)或列表(List)来构造各种值。为了表示一个包(多集),一个对象可以是包的 n 次成员,即包中的元素可以重复出现。我们不能简单地在一个关系中引入 n 个相同的元组,而是在关系模式中增加一个计数的属性 count,用以表明每个元素是包的多少次成员。例如,假设图 3.7 中的地址是一个包,而不是集合。我们可以说 123 Maple St., Hollywood 是第二次作为 Carrie Fisher 的地址,而 5 Locust Ln., Malibu 是第三次作为她的地址,表示为:

name	street	city	count
Carrie Fisher	123 Maple St.	Hollywood	2
Carrie Fisher	5 Locust Ln.	Malibu	3

确切地讲,我们不能在本章描述的抽象关系模型的关系中引入相同的元组。但基于 SQL 的关系 DBMS 却允许有重复的元组,也就是说,在 SQL 中,关系是包,而不是集合。详细内容参阅 4.6 和 5.4 节。如果元组的数目非常重要,即使 DBMS 允许有重复的元组,我们还是建议您使用这里描述的模式。

一个地址表可以用一个新属性——position(位置) 来表示, 用来表明在列表中的位置。例如, 我们可以将 Carrie Fisher 的地址表示为一个列表, Hollywood 在前, 表示如下:

name	street	city	position
Carrie Fisher	123 Maple St.	Hollywood	1
Carrie Fisher	5 Locust Ln.	Malibu	2

最后, 一个定长的地址数组可以用带有数组位置标志的属性表示。例如, 如果地址是两个元素的数组, 每个元素都是 street-city(街道-城市) 结构, Star 对象的表示如下:

name	street1	city1	street2	city2
Carrie Fisher	123 Maple St.	Hollywood	5 Locust Ln.	Malibu

3. 2. 4 单值联系的表示

一个 ODL 类定义经常包含与其他 ODL 类的联系。作为例子, 我们来考虑图 2. 6 中类 Movie 的完整定义, 图 3. 11 复制了该定义。

```
interface Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film ( color, blackAndWhite ) filmType;
    relationship Set Star stars
        inverse Star starredIn;
    relationship Studio ownedBy
        inverse Studio owns;
};
```

图 3. 11 Movie 类的完整定义

让我们首先来看联系 ownedBy, 该联系将每部电影和制作它的制片公司联系起来。我们的第一印象可能是联系像是一个属性。与表示其值为结构或结构集的 ODL 属性相类似, 我们可以建立关系的属性或属性组来表示相关类的对象。在联系为 ownedBy 的情况下, 我们可以针对类 Studio 的每个特性在 Movie 的关系模式中增加一个属性。

这种做法的一个问题是 Studio 对象有一个特性 owns, 它是指向类 Movie 的反向联系。为了表示这种联系, 每个电影元组将不得不包含它所属的制片公司制作的所有其他电影的信息。原则上, 上述每部电影的信息中都包括它的制片公司的信息, 这样, 又需要包含该制片公司制作的所有电影的信息。显然, 这种解决方案即便可行, 也未免过于复杂了。

如果电影和制片公司的系列不超出某个制片公司的所有电影, 用类似于联系 stars 及其反向联系 starredIn 的方法能让我们从一部电影找到其中的所有影星, 再找到这些影星主演的所有电影, 以及这些电影的所有影星, 等等, 很快就能访问到数据库中几乎所有的影星和电影。

如果我们想一想实际上对象是如何在计算机主存中存储的,就能找到更好的方法。当对象 O_1 包含另一个对象 O_2 的引用时,并不是把 O_2 复制到 O_1 内,而是 O_1 中有一个指针指向 O_2 。

然而,关系模型并没有指针或类似于指针的概念。作为替代,必须用表示相关对象的值来模拟指针的作用。我们需要的是相关类中构成键码的属性集。如果有了这种属性集,我们就把该联系看成好像是相关类的键码属性或属性组。下面的实例将说明这种技术。

例 3.6 假设名字 name 是类 Studio 的键码,该类的 ODL 定义如下(见图 2.6):

```
interface Studio {
    attribute string name;
    attribute string address;
    relationship Set Movie owns inverse Movie::ownedBy;
}
```

我们可以修改图 3.1 中的关系 Movie 的关系模式,增加表示所属制片公司的属性。我们可以任意地将该属性称为 studioName。在图 3.12 中,我们可以看到增加的属性 studioName 和一些元组的例子。

title	year	length	filmType	studioName
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

图 3.12 关系 Movie 用新属性表示所属制片公司

3.2.5 多值联系的表示

图 3.11 中的联系 stars 提出了一个我们在考虑联系 ownedBy 时没有看到的问题。当某联系的类型为一个类时,我们称该联系为单值的。类型为 Studio 的联系 ownedBy 就是一个单值联系的例子。然而,当联系是某个类的聚集类型时,我们称该联系为多值的。例如,stars 就是多值的,因为它的类型是 Set Star。换句话说,从类 A 到类 B 的任何一对多或多对多的联系都是多值联系。

为了表示多值联系,我们需要结合使用下面两种技术:

1. 和单值联系的处理方法一样,必须找出表示每个相关对象的键码;
2. 要表示相关对象的集合,和值为集合的属性的处理方法一样,也需要为每个值建立一个元组。也和值为集合的属性一样,这种方法也带来了冗余。因为对应于集合的每个成员,该关系的其他属性都将它们的值重复一次。3.7 节将讨论如何解决这个问题,目前,我们不妨先接受这种尚不完美的解决方案。

例 3.7 假设 name 是类 Star 的键码,那么,可以通过增加属性 starName 来扩展为

Movie 类设计的关系。该属性将记载电影中出现的某个影星的姓名。这样,一部电影就可以用许多元组来表示,元组的数量取决于数据库中列出的影星的数量。图 3.13 给出了一些实例。请注意数据的冗余;对应于每部电影的每个影星,该电影的所有其他信息都将重复一次。

title	year	length	filmType	studioName	starName
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	Emilio Estevez
Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Meyers

图 3.13 含有影星信息的关系 Movie

有时,一个类可能有多个多值联系。在这种情况下,表示单个类对象所需的元组数会爆炸性增长。假设类 C 有 k 个多值联系 R_1, R_2, \dots, R_k , 那么类 C 对应的关系需要有相应的属性对应于 C 的所有属性,并需要有表示 C 中所有单值联系的键码属性。通常,还需要有表示 R_1, R_2, \dots, R_k 的目标类的键码属性。

假设 C 的某特定对象 o 通过联系 R_1 与 n_1 个对象相联,通过 R_2 与 n_2 个对象相联,等等。那么对应于 R_1 的对象的每种选择,对应于 R_2 的对象的每种选择,等等,我们都需要为对象 o 建立一个相应的元组。从而在类 C 对应的关系中,共有 $n_1 \times n_2 \times \dots \times n_k$ 个元组对应于对象 o。

例 3.8 假设类 C 有一组单值属性 X 和两个多值联系 R_1 和 R_2 。假设这两个联系将类 C 分别与键码属性为集合 Y 和 Z 的两个类相联。现在,考虑类 C 的一个对象 c, c 通过联系 R_1 与键码为 y_1, y_2 的两个对象相联,通过 R_2 与键码为 z_1, z_2, z_3 的三个对象相联。再假设 x 表示对象 c 中属性集 X 的值。

于是,对象 c 在对应于类 C 的关系中可以用六个元组表示。我们不难想象这些元组是:

$$\begin{aligned} &(x, y_1, z_1) \ (x, y_1, z_2) \ (x, y_1, z_3) \\ &(x, y_2, z_1) \ (x, y_2, z_2) \ (x, y_2, z_3) \end{aligned}$$

也就是 Y 对应的键码与 Z 对应的键码的所有可能组合。

3.2.6 假如没有键码

ODL 之类的面向对象模型允许一个类的两个对象的所有特性具有完全相同的值。因此,我们必须对两个影星重名的现象做好思想准备。如果出现重名,姓名便不再是类 Star 的键码,因而,我们就不能用它表示 Movie 关系的元组中的影星。也许可以增加一个影星的其他属性作为键码,但原则上,我们不能保证下述现象不会出现:两个影星有相同的姓名,同一天出生,住在相同的地方.....在数据库中的其他特性也都相同。

联系的单向表示

如果两个实体集之间存在二元联系,我们就需要在两个实体集对应的关系中选择一个,并使二者之间的联系出现在该关系中。那么,究竟选择哪个呢?如果是多对多或一对一的联系,选择哪个可能是无关紧要的。但如果是多对一的联系,我们建议选择包括“多”的联系的实体集(该实体集中的多个实体对应于另一实体集的一个实体)。这样做的原因是为了避免冗余。

例如,实体集 Movies 和 Studios 之间的联系 Owns, 最好跟 Movies 放在一起。这样, Movies 对应的关系就包含了所属制片公司名字的属性,而每个元组都增加了该制片公司的名字。相反地,如果我们把该联系加入到 Studios 对应的关系中,就需要把制片公司的一个元组扩展成多个,该制片公司拥有的每部电影都对应于一个元组。结果,每个制片公司的所有其他信息在它拥有的每部电影中都要重复一次。

唯一可行的解决方案是建立一个表示“对象标识”的新属性,它标识的对象所属的类与关系相对应。例如,假设我们不能肯定 name 是否是影星的键码,我们就可以为每个影星建立一个属性“certificate number”(证书号),可能就是他们在演员协会的会员号。证书号能唯一标识一个影星,因为,有一个认证中心负责处理发放证书并记录已经用过的号码。

例 3.9 如果为每个影星建立唯一的证书号并将该证书号作为联系中表示影星的键码,采用这种方法,Movie 关系如下所示:

title	year	length	filmType	studioName	cert#
Star Wars	1977	124	color	Fox	12345

这里只列出了一个元组,并假设 12345 是 Carrie Fisher 的证书号。除了 Star 的 ODL 类定义中出现的所有信息以外,关系 Star 还必须包含“证书号”这一属性。下例给出了关系模式和 Carrie Fisher 的几个元组之一:

cert#	name	street	city	starredIn
12345	Carrie Fisher	123 Maple St.	Hollywood	Star Wars

3.2.7 联系与反向联系的表示

一般来说,直接从 ODL 转换到关系模型时,我们将 ODL 联系表示了两次——正向一次,反向一次。因此,在例 3.7 中,一部电影中出现的每个影星就放在有该电影名称的一个元组中;而在类 Star 对应的关系中,把联系 StarredIn 表示为若干个元组,即对于每个影星,都为他(她)主演的每部电影建立一个元组,记录每部电影的名称和年份(回忆一下,二者一起构成电影的键码)。

然而,同时表示影星联系及其反向联系(主演的电影)的做法带来了大量的信息冗余。二者中的任何一个都包含了另一个联系中的全部信息。因此,作为例子,完全可以从关系 Star 中省略 StarredIn 信息。或者反过来,也可以在 Star 关系中保留 StarredIn,而从 Movie 关系中删除 starName 属性。3.3.2 节将介绍第三种方法——将联系和反向联系分离出来,成为一个新的关系。这种方法有时是(但不总是)更可取的。然而,如果对应于联系的独立关系是最佳选择的话,3.7 节介绍的规范化过程将会促使我们把联系分离出来,成为独特的关系,即使我们进行关系设计的初衷并非如此。

注意:在 ODL 模型中,联系和反向联系两者都是必需的,因为它们分别指的是从电影到影星和从影星到电影的指针。指针是不能反向跟踪的,因此,两个方向的指针都是需要的。然而,在关系模型中,如同在 E/R 模型中,联系是用相关的键码来表示的。可以用包含一对相关键码(比如电影的名称、年份与影星的姓名)的元组从两个方向来跟踪联系。

3.2.8 本节练习

练习 3.2.1: 把下列练习中的 ODL 设计转换成关系数据库模式。

- * (a) 练习 2.1.1。
- (b) 练习 2.1.2(包括原练习规定的所有四种修改形式)。
- (c) 练习 2.1.3。
- (d) 练习 2.1.4。
- (e) 练习 2.1.5。
- * (f) 练习 2.1.6。

练习 3.2.2: 把图 2.7 中的 ODL 描述转换成关系数据库模式。练习 2.1.8 中的三种修改对关系模式有何影响?

! 练习 3.2.3: 图 3.14 是类“顾客”的 ODL 定义。在该类的对象中,我们保存了不同类型的电话(例如:住宅电话、办公电话、传真)的集合和“职业介绍”(referrals)的集合,在职业介绍集中对该顾客为其他顾客介绍职业给予应有的肯定。请把上述 ODL 描述转换成关系数据库模式。

```
interface Customer {
    attribute Struct Name
        {string first, string last } name;
    attribute Set<
        Struct Phone {string type, int number }
        > phones;
    relationship Customer referredBy inverse referrals;
    relationship Set< Customer> referrals
        inverse referredBy;
}
```

图 3.14 顾客记录

当然,不能肯定给出的 ODL 实现将以预期的方式表示“指针”,因此,具体的实现中这一对相反的联系可能只有一种表示。

指针: 优点还是缺点?

ODL 联系大概是通过每个对象指向一个或多个相关对象的指针或引用实现的。这种实现非常方便, 因为从一个对象出发, 可以迅速找到相关的对象。相比之下, 在关系模型中, 对象不是用指针, 而是用键码的值来表示的, 要从一个对象找到相关的对象, 似乎搜索得慢一些。

例如, 例 3.7 表示了一个 Movie 对象, 一部电影的每个影星都有一个元组; 该元组只包含了该影星的姓名, 而不是该影星的所有信息。如果我们想找到 Wayne's World 中的影星的地址, 就需要取出每个影星的姓名, 并在 Star 关系中查找该影星对应的元组。然后, 在那里才能找到该影星的地址。

也许有人会因此得出结论: 关系模型中没有指针是该模型的一个缺陷。然而, 实际上, 我们可以为关系建立索引, 这样就能很有效地搜索在给定分量上具有给定值的元组了(见 1.2.1 节和 5.7.7 节)。所以, 我们在实践中没有使用指针并未损失什么。而且, 虽然指针对于运行于主存、并且最多只需要若干秒的程序非常有用, 但是数据库与这些程序相去甚远。在将会存在若干年、并且可能分布在广泛的分布式计算机系统所连接的许多第二存储设备上的对象之间实现指针, 恐怕要困难得多。所以, 关系模型的无指针方法可能成为一种主流。

3.3 从 E/R 图到关系的设计

E/R 图到数据库模式的转换与 ODL 设计到数据库模式的转换类似。但 E/R 模型在某种意义上是面向对象的设计和关系设计的中间形式。因此, 从 E/R 图开始, 有些比较困难的工作我们已经解决了。E/R 模型与 ODL 的主要区别在于:

1. E/R 模型中, 联系作为独立的概念存在, 而不是作为特性嵌套在类定义中。这种区别有助于避免信息冗余, 而在 3.2.5 节选择嵌入多值联系时, 就像 stars 出现在表示 Movie 对象的元组中, 结果带来了冗余。
2. ODL 中, 属性可能是任意的聚集类型, 比如集合。而 E/R 模型虽然并没有严格规定允许使用的数据类型, 但通常都认为允许使用结构化数据而不允许使用集合或其他聚集类型构造符。因而, 像例 3.4 中的影星地址集之类的值为集合的属性, 在 E/R 模型中必须将地址集作为一个实体集来处理, 同时定义联系 Lives-at 来连接影星及其地址。
3. E/R 模型中, 联系可以具有属性, 而 ODL 中没有相应的概念。

3.3.1 实体集到关系的转换

我们首先来考虑不是弱实体集的情形。3.3.3 节将介绍为适应弱实体集需要做的修改。对于每个非弱实体集, 我们将建立一个与之同名而且具有相同属性集的关系。该关系

但有些 E/R 模型对属性类型的限制与 ODL 一样, 允许使用结构或任何简单类型的聚集。

并不包含与实体集有关的任何联系的信息;联系用独立的关系模式来处理,详见 3.3.2 节。

例 3.10 考虑图 3.15(同图 2.8)中的三个实体集: Movies(电影)、Stars(影星)和 Studios(制片公司)。实体集 Movies 的属性是 title、year、length 和 filmType。因此,关系 Movies 看起来就和 3.1 节介绍的图 3.1 中的关系 Movie 一样。

图 3.15 电影数据库的 E/R 图

例 3.11 现在,考虑图 3.15 中的实体集 Stars,它有两个属性: name(姓名)和 address(地址)。所以,我们预料对应的关系 Stars 应该是:

name	address
Carrie Fisher	123 Maple St. , Hollywood
Mark Hamill	456 Oak Rd. , Brentwood
Harrison Ford	789 Palm Dr. , Beverly Hills

该关系与例 3.3 中建立的关系 Star(见图 3.6)相似,而后者有三个属性,其中的两个——street 和 city——构成一个结构化的地址。二者之间的区别是次要的。我们也可以把这里的关系 Stars 设计得和图 3.6 中的关系 Star 完全相同,只要修改图 2.8 中的 E/R 模型,使实体集 Stars 的属性 address 分成两个属性: street 和 city 即可。

3.3.2 E/R 联系到关系的转换

E/R 模型中的联系也可用关系表示。对于给定的联系 R,它所对应的关系具有以下属性:

- 1. 联系 R 涉及到的每个实体集的键码属性或属性集应该是 R 对应的关系模式的一部分;
- 2. 如果 R 有属性,这些属性也应该是 R 对应的关系的属性;

如果一个实体集在联系 R 中出现多次,必须为每次出现的属性改名,以免出现重名属性。同样,如果同名属性在 R 本身或 R 涉及的实体集的属性中出现两次或两次以上,也必须改名,以免重名。

例 3.12 考虑图 3.15 中的联系 Owns。它把实体集 Movies 和 Studios 联系起来。因此,我们使用 Movies 的键码,即 title 和 year,和 Studios 的键码,即 name,来表示 Owns 的关系模式。该关系的一个例子是:

title	year	studioName
Star Wars	1977	Fox
Mighty Ducks	1991	Disney
Wayne' s World	1992	Paramount

为了清楚起见,我们采用属性 studioName,它对应于 Studios 中的 name 属性。
请注意:上面的关系加上例 3.10 中为实体集 Movies 构造的关系(见图 3.1),完全包含了为例 3.6 的类 Movie 构造的关系(见图 3.12)中的信息,只是把其 stars 特性排除在外。

例 3.13 同样,图 3.15 中的联系 Stars-In 也可以转换成具有属性 title 和 year (Movie 的键码)以及属性 starName(实体集 Stars 的键码)的关系。图 3.16 是关系 Stars-In 的一个例子。注意,该关系加上图 3.1 包含了图 3.13 中的信息,但它们却没有重复的 Movie(译注:原文误为 Star)类的非键码属性(length 和 filmType),而没有非键码属性将有损图 3.13 中的关系模式。

似乎年份 year 在图 3.16 中是冗余的。不过,这只是由于这些电影都不重名造成的。如果有几部电影重名,比如都叫“ King Kong ”,我们将会看到 year 对找出哪些影星出演某部电影的哪个版本有决定性的作用。

title	year	starName
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Mighty Ducks	1991	Emilio Estevez
Wayne' s World	1992	Dana Carvey
Wayne' s World	1992	Mike Meyers

图 3.16 联系 Stars-In 对应的关系

- 与从 ODL 设计出发相比,从 E/R 图出发转换到关系模式有以下优点:
- 关系通常都是“规范化”的,这意味着将避免某些从 ODL 描述直接进行设计时出现的冗余。
 - 双向的 ODL 联系将用表示双向联系的单个关系来代替。

例 3.14 多向的联系也容易转换为关系。让我们来考虑图 2.12(即这里的图 3.17)中的四向联系 Contracts (签约),它包括影

图 3.17 联系 Contracts(签约)

再谈 ODL 到关系的转换

正如我们所看到的, E/R 模型中的联系转换成关系和 ODL 联系转换成关系相比, 有时前者能给出更合适的关系数据库模式。我们可随意地借助于 E/R 技术把“多对一”和“多对多”的联系分解成单独的关系。这样做有助于消除冗余和元组数目激增——当我们企图实现和类一起定义的多值 ODL 联系时, 有时会遇到这种情况。但是, 我们再次提醒读者, 3.7 节将给出一种把从 ODL 描述直接得到的关系模式加以调整的机械方法。

星、电影和两个制片公司——第一个拥有影星, 第二个制作电影, 并约定了影星在电影中演什么角色。我们用关系 `Contracts` 来表示这个多向联系, 其关系模式的属性由来自下列四个实体集的键码组成:

1. 键码 `starName`(影星名)代表影星。
2. 由属性 `title`(名称)和 `year`(年份)组成的键码代表一部电影。
3. 键码 `studioOfStar`(影星所在的制片公司)表示第一个制片公司的名字; 请记住, 我们假定制片公司名是实体集 `Studio` 的键码。
4. 键码 `producingStudio`(制作电影的制片公司)表示制作该影星所主演的电影的制片公司名。

注意: 我们为关系模式中的属性取名时比较注意, 没有一个属性叫“`name`”, 因为它会引起歧义, 我们不知道它是指影星的名字还是指制片公司的名字, 或者, 对于后者——制片公司, 究竟是指哪个制片公司。

3.3.3 处理弱实体集

如果 E/R 图中出现了弱实体集, 我们需要做各不相同的三件事。

1. 弱实体集 `W` 本身所对应的关系模式必须既包含 `W` 的属性, 也包含有助于构成 `W` 的键码的其他实体集的键码属性。这些辅助实体集是很容易识别的, 因为它们通过双菱形表示的“多对一”的联系与 `W` 相联。
2. 有弱实体集 `W` 出现的任何联系都必须把 `W` 的所有键码属性(包括为 `W` 提供键码的其他实体集的键码属性)作为键码。
3. 然而, 正如我们将要看到的, 从弱实体集 `W` 到有助于为 `W` 提供键码的其他实体集的双菱形联系, 其实并不需要转换成关系。理由是这种联系的属性必然是弱实体集 `W` 自身属性集的子集, 这样的话, 这些联系除了帮助 `W` 找到它的键码之外, 并没有提供附加信息。

当然, 在引入附加的属性来构成弱实体集的键码时, 我们必须注意属性不能重名。必要时, 我们应为其中的一些属性改名。

例 3.15 考虑图 2.27 中的弱实体集 `Crews`, 我们将其复制为图 3.18。从该图我们可以得到三个关系, 关系模式分别为:

`Studios (name, addr)`

Crews (number, studioName)

Unit-of (number, studioName, name)

第一个关系 Studios 是从同名的实体集直接构造的。第二个关系 Crews 来自弱实体集 Crews。Crews 的键码属性应是该关系的属性。如果 Crews 还有非键码属性,它们也应该包含在关系 Crews 的属性集中。我们选择与实体集 Studios 的属性 name 相对应的 studioName 作为关系 Crews 的属性。

图 3.18 弱实体集 Crews

第三个关系 Unit-of, 来自与它同名的联系。和通常的做法一样, 在关系模型中我们用 一个关系表示一个 E/R 联系, 该关系的关系模式包含相关实体集的键码属性。在这里, Unit-of的属性有弱实体集 Crews 的键码属性 number 和 studioName, 还有实体集 Studios 的键码属性 name。但要注意: 由于 Unit-of 是“多对一”的联系, 制片公司的 studioName 自然与制片公司的 name 是一码事。

例如, 假设 Disney crew # 3 是 Disney 制片公司的一个电影组, 则 Unit-of 的联系集 包含下述组合:

(Disney crew # 3, Disney)

它所对应的 Unit-of 元组为:

(3, Disney, Disney)

所以, 我们可以把 Unit-of 的属性 studioName 和 name 进行合并, 从而, 给出一个更 简单的关系模式:

Unit-of (number, name)

但是, 现在我们完全可以省略 Unit-of 关系了, 因为它的属性是关系 Crews 的属性的 一个子集。

例 3.16 现在考虑 2.6.1 节例 2.31 和图 2.28 中的弱实体集 Contracts(签约)。我们 将其复制在图 3.19 中。关系 Contracts 的关系模式为:

Contracts (starName, studioName, title, year, salary)

这些属性是经过适当的改名处理的 Stars 的键码和 Sturdios 的键码, 以及构成 Movies 的键码的两个属性, 还有实体集 Contracts 自己的单个属性 salary(酬金)。没有任 何关系对应于联系 Star-of, Studio-of 或者 Movie-of。它们中间任何一个的关系模式都将 是上面的 Contracts 的真子集。

顺便提一下, 请注意: 假如我们从图 2.13 中的 E/R 图出发, 将得到与这里完全一样的 关系。回忆一下, 那幅图中将签约 contracts 视为 stars(影星)、movies(电影)和 studios (制片公司)之间的三向联系, 而 Contracts 还有一个附加的属性: salary(酬金)。

在例 3.15 和例 3.16 看到的现象——双菱形联系不需要关系——是弱实体集的一种

图 3.19 弱实体集 Contrancts

普遍现象。弱实体集 E 对应的关系模式包含了由任何双菱形联系 R 构造的关系模式——其中 R 是从 E 到某个有助于形成 E 的键码的其他实体集的“多对一”联系。原因是 E 对应的关系中既包含了 E 的键码属性,也包含了由 R 连接的两个实体集的所有键码属性。因此,我们可以得出下面两条对弱实体集的修正规则。

- 如果 E 是一个弱实体集,为 E 构造一个关系,其关系模式中包括 E 的键码属性,还包括通过“多对一”的联系与 E 相关的“辅助”实体集的的键码属性。
- 不要为任何从弱实体集到其他实体集的“多对一”联系构造关系,因为该联系是有助于为弱实体集提供键码的双菱形联系。

3.3.4 本节练习

练习 3.3.1: 将图 3.20 中的 E/R 模型转换为关系数据库模式。

图 3.20 航空公司的 E/R 图

图 3.21 “姊妹”舰的 E/R 图

练习 3.3.2: 图 3.21 中的 E/R 图表示舰艇。如果两艘舰艇是根据同一个方案设计制造的,就称它们为“姊妹”舰。把这个 E/R 图转换为关系数据库模式。

练习 3.3.3: 把下面的 E/R 图转换为关系数据库模式。

(a) 图 2.28。

- (b) 练习 2.6.1 的答案。
- (c) 练习 2.6.4(a) 的答案。
- (d) 练习 2.6.4(b) 的答案。

3.4 子类结构到关系的转换

面向对象的模型和 E/R 模型在处理子类概念方面有些许不同。这种不同导致了以两种不同的方式构成与类的层次相应的关系。我们先来回忆一下二者的区别：

- 在 ODL 中, 一个对象完全属于一个类。对象继承了它的所有超类的特性, 但是在技术上, 它并不是其超类的成员。
- 在 E/R 模型中, 一个“对象”可以由属于多个实体集的实体来表示, 其中这些实体集通过“属于”联系连接在一起。因此, 连接在一起的实体共同表示这个对象, 并把其特性——属性和联系给予该对象。

下面我们就来看看这两点如何影响不同的关系数据库模式设计策略。我们应该记住, ODL 模型或 E/R 模型并不必然导致这种或那种方法; 如果愿意, 我们可以选择适合另一种模型的方法。

3.4.1 用关系表示 ODL 子类

首先, 让我们来看看将 ODL 子类的分层结构转换成关系模式的技术。应该遵循下面的原则:

- 每个子类都有自己的关系。
- 这个关系用该子类的所有特性(包括它继承的全部特性)来表示。

例 3.17 考虑图 2.22 中的四个类的分层结构。回忆一下这些类的含义:

1. Movie(电影), 最主要的类。它是本章众多例子中所讨论的类。
2. Cartoon(卡通片), Movie 的一个子类, 附加了一个特性: 是和影星集之间的联系, 称为 voice(配音)。
3. MurderMystery(谋杀片), 是 Movie 的另一个子类, 附加了一个属性: weapon(武器)。
4. Cartoon-MurderMystery(卡通-谋杀片), 是 Cartoon 和 MurderMystery 的子类, 除了(天生的)三个超类的所有特性以外, 它没有附加的子类。

Movie 的模式和以前一样:

Movie(title, year, length, filmType, studioName, starName)

一些典型的元组见图 3.13。对于类 Cartoon, 我们在六个属性的 Movie 模式中增加了 voice 属性, 给出了七个属性的模式:

Cartoon(title, year, length, filmType, studioName, starName, voice)

对于类 MurderMystery, 我们构造了另一个关系, 它的关系模式除了 Movie 的六个属性外, 还包括 weapon。也就是说, MurderMystery 关系的模式为:

MurderMystery(title, year, length, filmType, studioName, starName, weapon)

最后, 关系 Cartoon-MurderMystery 的模式不仅包括 Movie 的六个属性, 还包括另

两个超类的属性 voice 和 weapon。因而, 该关系的模式有八个属性:
Cartoon-MurderMystery (title, year, length, filmType, studioName, starName, voice, weapon)

3. 4. 2 在关系模型中表示“属于”联系

透过 E/ R 模型中的 isa(属于) 分层结构, 我们看到这样一个特点: 分层结构通过与属于联系有关的实体集进行扩展。因此, 很自然的做法就是为每个实体集建立一个关系, 并且只将该实体集的属性赋给这个关系。然而, 为了识别与每个元组有关的实体集, 就需要包括每个实体集的键码属性。因为从 E/ R 模型到关系模型的转换方法把有关 E/ R 属性和联系的信息分成几个单独的关系, 结果, 某个子类的成员信息将分散到几个关系里, 而这种情形总有可能发生。

我们没有为属于联系建立相应的关系, 而把属于联系隐含在这样的事实里, 即与属于联系有关的实体集拥有相同的键码值。

图 3. 22 电影 E/ R 图的分层结构

例 3. 18 考虑图 2. 22 中的 E/ R 模型的分层结构。回忆一下, E/ R 图的相关部分曾见于图 2. 23, 我们将其复制成图 3. 22。表示这部分图所需的关系模式为:

- 1. 关系 Movies(title, year, length, filmType), 即例 3. 10 中讨论的关系。
- 2. 关系 MurderMysteries(title, year, weapon)。前两个属性是所有电影的键码, 而最后一个属性是所对应的实体集的唯一属性。
- 3. 关系 Cartoons(title, year)。该关系是卡通片的集合。它除了电影的键码之外, 没有其他的属性, 因为卡通片的附加属性都包括在联系 Voices 之中。
- 4. 关系 Voices(title, year, name), 对应于影星 Stars 与 Cartoons 之间的联系Voices。最后一个属性是 Stars 的键码, 而前两个属性构成了 Cartoons 的键码。

注意, 图 3. 22 中没有任何实体集对应于类 Cartoon-MurderMysteries。因此, 与例 3. 17中的关系设计不同, 没有专门为既是卡通片又是谋杀片的电影设计的关系。对于一部集二者于一身的电影, 我们从 Voices 关系中获得其配音的信息, 从 MurderMysteries 关系中获得武器的信息, 所有其他信息来自电影 Movies 关系或者来自涉及到实体集 Movies, Cartoons 和 MurderMysteries 的任一联系所对应的关系。

还要注意, 关系 Cartoons 的模式是 Voices 的关系模式的子集。在许多情形下, 我们

可能倾向于删除 Cartoons 关系, 因为看起来它没有提供 Voices 之外的任何信息。但是, 在我们的数据库中可能有一些无声卡通片。这些卡通片没有配音, 这样我们就有可能忽略它们是卡通片的事实。实际上, 同样的问题也以不同的形式出现在例 3. 17 的关系 Cartoons 中, 在那里, 没有配音的卡通片是不会提及的。这个问题可以通过规范化来解决, 我们将在 3. 7 节详细讨论。

3. 4. 3 方法的比较

3. 4. 1 节和 3. 4. 2 节的两种方法都有各自的问题。ODL 转换把一个对象的所有特性都保存在一个关系中。但我们可能不得不搜索多个关系才能找到一个对象。例如, 使用例 3. 17 的数据库模式, 要找到一部电影的长度, 我们必须搜索四个不同的关系, 直到找到该电影所在的类对应的关系为止。

另一方面, E/R 转换对于一个对象(实体) 所属的每个实体集或联系都把该对象的键码复制一次。这种重复浪费空间。而且, 我们也许不得不在几个关系中搜寻, 才能找到一个对象的信息。例如, 如果我们想在例 3. 18 的数据库模式中查找某个谋杀片的长度以及所用的武器, 就可能是这种情况。

3. 4. 4 使用 NULL 值合并关系

可有多种方法表示类的分层结构。可以使用一个特殊的“空值”, 用 NULL 表示。当 NULL 作为元组的某个属性的分量出现时, 这就非正式地表明该元组在该属性上没有合适的值。空值虽然不是传统关系模型的组成部分, 但实际上它非常有用, 而且在 SQL 查询语言中扮演着重要的角色, 详见 5. 9 节。

如果在元组中允许使用 NULL 值, 我们就可以用单个关系来表示类的分层结构。该关系拥有的属性对应于分层结构中任何类的对象所拥有的全部特性。因此, 一个对象就可以用一个元组来表示了。如果不是这个对象的类所拥有的特性, 则该元组对应的属性就取值为 NULL。

例 3. 19 如果我们将这种方法应用于例 3. 17, 将会建立一个关系, 其模式为:

Movie(title, year, length, filmType, studioName, starName, voice, weapon)

像《谁陷害了兔子罗杰》(Who Framed Roger Rabbit?) 这样既是卡通片又是谋杀片的电影, 在没有 NULL 值时可能需要几个元组才能表示; 每个“voice”对应一个元组。另一方面, 《美人鱼》(The Little Mermaid) 是一部卡通片, 但不是谋杀片, 它的 weapon 分量将取值为 NULL。《东方快车谋杀案》(Murder on the Orient Express) 在 voice 属性上将取值为 NULL, 而《乱世佳人》(Gone With the Wind) 在 voice 和 weapon 属性上将都为 NULL。

注意: 像 3. 4. 2 节中的方法那样, 这种方法允许我们在一个关系中查找来自分层结构中所有类的元组。另一方面, 像 3. 4. 1 节中的方法那样, 它也允许我们在一个关系中查找一个对象的所有信息。

事实上, 《谁陷害了兔子罗杰》中既有影星又有配音, 每个影星-配音(star-voice) 对将对应一个元组。一个纯卡通片的 starName 属性将取值为 NULL, 结果把配音和其他信息记录下来。

3.4.5 本节练习

练习 3.4.1: 请把图 3.23 中的 E/R 图转换成关系数据库模式。

图 3.23 练习 3.4.1 的 E/R 图

练习 3.4.2: 图 3.24 给出了一个模式的 ODL 描述, 该模式类似于练习 3.4.1 中的 E/R 图。请把它转换为关系数据库模式。记住, Course 对象有一个“对象标识”(object identity), 可以新建一个属性来表示该 OID, 比如, CourseID。在本练习中请不要模仿练习 3.4.1 中所用的转换弱实体集的策略(尽管在原则上, 如果愿意, 也可以这么做)。

练习 3.4.3: 请把下列练习中的 ODL 设计转换成关系数据库模式。

- * (a) 练习 2.4.1。
- (b) 练习 2.4.4。

练习 3.4.4: 请把下列练习中的 E/R 设计转换成关系数据库模式。

- * (a) 练习 2.4.3。
- (b) 练习 2.4.4。

！练习 3.4.5: 请把图 3.25 中的 E/R 图转换成关系数据库模式。

```
interface Course {
    attribute int number;
    attribute string room;
    relationship Dept deptOf inverse Dept::
        coursesOf;
};

interface LabCourse : Course {
    attribute int computerAlloc;
};

interface Dept {
    unique attribute string name;
    attribute string chair;
    relationship Set< Course> coursesOf
        inverse Course:: deptOf;
};
```

图 3.24 课程和实验课程的 ODL 描述

图 3.25 练习 3.4.5 的 E/R 图

！练习 3.4.6: 从相应的 ODL 定义出发得到的关系数据库模式, 与练习 3.4.5 得到的会有什么不同?

3.5 函数依赖

关系模型中要处理的最重要的一类约束是称为“函数依赖”的单值约束。在关系数据库的设计过程中, 为了减少冗余, 需要对数据库模式进行重新设计, 在 3.7 节读者将会看到, 了解这类约束对此是至关重要的。还有一些其他约束, 有助于我们设计出优秀的数据库模式, 比如覆盖了 3.8 节的多值依赖以及将在 4.5 节提到的存在约束 (existence constrain) 和独立性约束 (independence constrain)。

3.5.1 函数依赖的定义

关系 R 上的“函数依赖”是这种形式的陈述——“如果 R 的两个元组在属性 A_1, A_2, \dots, A_n 上一致 (也就是, 两个元组在这些属性相对应的各个分量具有相同的值), 则它们在另一个属性 B 上也应该一致”。这种依赖正式记作 $A_1A_2\dots A_n \twoheadrightarrow B$, 也可以说“ A_1, A_2, \dots, A_n 函数决定 B”。

如果一组属性 A_1, A_2, \dots, A_n 函数决定多个属性, 比方说,

$$\begin{array}{l} A_1A_2\dots A_n \twoheadrightarrow B_1 \\ A_1A_2\dots A_n \twoheadrightarrow B_2 \\ \dots \\ A_1A_2\dots A_n \twoheadrightarrow B_m \end{array}$$

则可以把这一组依赖关系简记为:

$$A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$$

图 3.26 提示我们在关系 R 中, 对于任意两个元组 t 和 u, 这种函数依赖的含义。

例 3.20 让我们考虑图 3.13 中的关系 Movie, 图 3.27 是它的一个实例。我们可以合理地断言关于 Movie 关系的几个函数依赖。例如, 我们可以断言如下三个函数依赖:

$$\begin{array}{l} \text{title year} \twoheadrightarrow \text{length} \\ \text{title year} \twoheadrightarrow \text{filmType} \\ \text{title year} \twoheadrightarrow \text{studioName} \end{array}$$

由于三个依赖的左边完全相同, 都是 title 和 year, 我们可以用简写的形式把它们汇总在一行中:

$$\text{title year} \twoheadrightarrow \text{length filmType studioName}$$

这组函数依赖具体地说明了如果两个元组在 title 分量和 year 分量都有相同的值, 则

图 3.26 函数依赖对两个元组的影响

函数依赖蕴涵着模式信息

记住, 函数依赖和任何约束一样, 都是针对关系模式、而不是针对特定实例的断言。只看一个实例, 并不能确切地断定某个函数依赖成立。例如, 看了图 3. 27, 我们可能会假设依赖 $title \rightarrow filmType$ 成立, 因为对 Movie 关系这个特例中的每个元组而言, 正好出现这种情况: 任意两个在 $title$ 上一致的元组在 $filmType$ 上也一致。

然而, 我们不能为关系 Movie 断言这个函数依赖。如果我们的实例中包含诸如 King Kong 的两个版本——彩色版本和黑白版本——的元组, 那么, 上面的函数依赖就不成立了。

它们在 $length$, $filmType$ 和 $studioName$ 分量上必然也都有相同的值。如果我们记住了导致建立 Movie 关系模式的原始设计, 那么这种断言就是有意义的。属性 $title$ 和 $year$ 构成了电影对象的键码。因此, 我们认为给定了名称和年份, 对应的电影就只有唯一的电影长度、唯一的电影类型和唯一的所属制片公司。

title	year	length	filmType	studioName	StarName
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	Emilio Estevez
Wayne' s World	1992	95	color	Par amount	Dana Carvey
Wayne' s World	1992	95	color	Par amount	Mike Meyers

图 3. 27 关系 Movie

另一方面, 我们观察到, 如下表述:

$title \rightarrow year \rightarrow starName$

就是错误的; 它不是函数依赖。已知 $title$ 和 $year$ 构成了电影的键码, 我们可能认为上面的依赖也成立。但是, 从 Movie 类定义的内容来看, 实际情况是每部电影都存在唯一确定的影星集。当我们从 ODL 设计转换到关系模型时, 必须为每部电影建立多个元组, 而每个元组都有不同的影星。所以, 即使所有的元组在 Movie 类的其他特性上都相同, 它们在影星姓名上也绝不相同。

3. 5. 2 关系的键码

如果一个或多个属性的集合 $\{A_1, A_2, \dots, A_n\}$ 满足如下条件, 我们就称该集合为关系 R 的键码:

- 1. 这些属性函数决定该关系的所有其他属性。也就是说, R 中不可能有两个不同的元组, 它们在 A_1, A_2, \dots, A_n 上的取值完全相同。
- 2. $\{A_1, A_2, \dots, A_n\}$ 的任何真子集都不能函数决定 R 的所有其他属性, 也就是说, 键码必须是最小的。

函数依赖中的“函数”是什么意思？

$A_1A_2 \dots A_n \rightarrow B$ 之所以称为“函数”依赖,是因为原则上有这样一個函数:取一组值,分别对应于属性 A_1, A_2, \dots, A_n , 结果对应于 B 产生唯一值(或者是空值)。例如,在 Movie 关系中,我们假设一个函数,取一个字符串,比如 Star Wars,再取一个整数,比如“1977”,结果产生一个唯一的长度 length 值,比如 124,这个函数就出现在关系 Movie 中。但是,该函数不是我们在数学中遇到的普通意义上的函数,因为无法从自变量来计算因变量。也就是说,我们不能对 Star Wars 这样的字符串和 1977 这样的整数做某些运算而得到正确的电影长度。而只能通过在该关系中查找来“计算”该函数。我们查找具有给定的 title 和 year 值的元组,而后看该元组的 length 值是什么。

如果键码只包含一个属性 A ,我们通常就称 A (而不是 $\{A\}$)为键码。

例 3.21 属性组 {title, year, starName} 构成了图 3.27 中的 Movie 关系的键码。首先,我们必须证明它们函数决定所有其他的属性。也就是说,假设两个元组在这三个属性——title, year 和 starName 上取值一致。由于它们在 title 和 year 上取值一致,正如我们在例 3.20 中所讨论的,它们必然在另外几个属性 length, filmType 和 studioName 上取值一致。因此,两个不同的元组不可能在 title, year 和 starName 三个属性上全都取值一致;它们实际上就是同一个元组。

现在,我们必须证明 {title, year, starName} 的任何真子集都不能函数决定所有其他的属性。为什么呢?首先我们观察到 title 和 year 不能决定 starName,因为许多电影有多个影星。因此, {title, year} 不是键码。

{ year, starName } 也不是键码,因为一个影星在同一年中可能出演两部电影;因此,

year	starName	title
1977	Star Wars	Star Wars
1977	Star Wars	Star Wars 2

不是函数依赖。同样,我们断定 {title, starName} 也不是键码,因为不同年份可能会制作两部同名电影。而且这两部电影可能有一个共同的影星,虽然坦白地讲,我们还没有想到一个例子。

有时一个关系有多个键码。这样的话,通常要指定其中的一个键码为主键码(primary key)。在商业数据库系统中,主键码的选择会影响到某些实现细节,比如关系在磁盘上如何存储。

3.5.3 超键码

包含键码的属性集称为“超键码”(superkey),是“键码的超集”(superset of a key)的缩写。因此,每个键码都是超键码。但是,某些超键码不是(最小的)键码。注意,每个超键码都满足键码的第一个条件:函数决定它所在的关系的所有其他属性。但是,超键码不必满足键码的第二个条件:最小化条件。

记住,函数依赖是我们对数据所做的假设或断言。没有任何外面的权威可以给我们一个绝对的结论,判断一个函数依赖是否成立。所以,关于依赖的成立性,我们可以做出任何似乎合理的假设。

键码的其他术语

在一些书和论文中,可能会发现关于键码的不同术语。可能会碰到把术语“键码”(key)用于我们使用术语“超键码”(superkey)的场合,也就是说,是函数决定所有属性的属性集,但没有最小化要求。这些资料一般使用术语“候选键码”(candidate key)来表示最小的键码——也就是我们用“键码”这个术语所表示的意思。

例 3.22 在例 3.21 的关系中,有许多超键码。不仅键码{title, year, starName}本身是超键码,而且该属性集的任何超集,比如{title, year, starName, length}都是超键码。

3.5.4 寻找关系的键码

如果一个关系模式是从 ODL 或 E/R 设计转换成关系而推导出来的,通常我们可以预测该关系的键码。我们在本节将考虑来自 E/R 图的关系;3.5.5 节将讨论来自 ODL 设计的关系。

如果关系来自 ODL 或 E/R 设计,通常(但不绝对)每个关系只有一个键码。当关系只有一个键码时,在关系模式中将键码属性用下划线标出是个有用的约定。

推断键码的第一条规则是:

- 来自实体集的关系的键码就是该实体集或类的键码属性。

例 3.23 在例 3.10 和 3.11 中,我们描述了实体集 Movies 和 Stars 如何转换成关系。这两个实体集的键码分别为{title, year}和{name}。因此,它们也是相应关系的键码,并且

Movie(title, year, length, filmType)
Stars(name, address)

分别是相应的关系模式,其中用下划线标出的是关系的键码。

第二条规则涉及到二元联系。如果关系 R 来自一个联系,则该联系的多样性将会影响 R 的键码。有三种情形:

- 如果联系是“多对多”的,则相连的两个实体集的键码都是 R 的键码属性。
- 如果是从实体集 E₁ 到实体集 E₂ 的“多对一”联系,那么实体集 E₁ 的键码属性是 R 的键码属性,而 E₂ 的键码属性则不是 R 的键码属性。
- 如果联系是“一对一”的,则联系两端的任何一个实体集的键码属性都是 R 的键码属性,即 R 的键码不是唯一的。

例 3.24 例 3.12 讨论了联系 Owns(拥有),它是从实体集 Movies 到实体集 Studios 的“多对一”联系。因此,关系 Owns 的键码是来自 Movies 的键码属性 title 和 year。Owns 的关系模式如下,其中用下划线标出的是键码属性:

Owns(title, year, studioName)

另一方面,例 3.13 讨论了实体集 Movies 和 Stars 之间的“多对多”联系 Stars-in(主演)。现在,相应关系的所有属性都是键码属性。

Stars-in(title, year, starName)

函数依赖的其他概念

我们的观点是函数依赖的左边可以有多个属性,但右边只能有一个属性。而且,右边的属性还不能出现在左边。但是,允许左边相同的几个函数依赖合并成简化形式,于是就得到右边有多个属性的函数依赖。我们还发现允许出现“平凡依赖”(这是指,右边的属性是左边的属性之一)有时会带来方便。

关于这个问题的其他著作通常从这样的观点出发:左右两边都是任意属性集,并且任何属性都可能在两边同时出现。这两种方式没有重大区别,但除非特别声明,我们将继续沿用这种观点——函数依赖中,不存在同时出现在两边的属性。

事实上,只有在多对多联系本身具有属性的情况下,该联系对应的关系才不能将其所有属性作为键码的组成部分。于是,联系本身的属性将从键码中略去。

最后,让我们考虑多向联系。由于我们不能用从该联系引出的箭头来描述所有可能的依赖,就存在这样的情形:如果不仔细考虑哪些实体集函数决定另一些实体集的话,键码是不那么容易看出来的。不过,我们可以保证:

- 如果多向联系 R 有一个箭头指向实体集 E ,则相应的关系中,除了 E 的键码以外,至少还存在一个键码。

3.5.5 由 ODL 设计导出的关系的键码

当我们把 ODL 设计转换成关系时,情况有点复杂。首先,一个 ODL 类可能有一个或多个说明的键码,但在属性之中也可能根本没有键码。解决后一种情况,正如我们在 3.2.6 节所讨论的,必须在相应的关系中引入一个属性,代替该类对象中的对象标识。

但是,无论是 ODL 类拥有由自己的属性组成的键码,还是必须把代替对象标识的属性作为键码,在某些情况下,类的键码属性将不是关系的键码。原因是,在使用我们介绍的从 ODL 转换到关系的方法时,有时一个关系中包含了太多的信息。这个问题在类的定义中包含联系(译注:原文误为“关系”)时尤为突出。

首先,假设类 C 有一个指向某个类 D 的单值联系 R 。然后,我们像 3.2.4 节中建议的那样,在 C 的关系中包含 D 的键码。 C 的键码依然是相应关系的键码。

当 C 有一个指向某个类 D 的多值联系 R 时,问题就出现了。如果 R 的反向联系是反向的单值联系(即, R 是个“一对多”的联系),则像 3.2.7 节的方框“联系的单向表示”中提示的那样,我们只能在 D 的关系中表示 R 的反向联系。 R 的反向联系在 D 的关系中表示是毫无问题的,因为它是 D 中的单值联系。

但是,假设 R 是“多对多”的联系,也就是说, R 及其反向联系在 C 和 D 中都是多值的,则为 C 构造的关系中可能要用多个元组来表示类 C 的一个对象。结果, C 的键码就不是相应关系的键码。而我们不得不把 D 的键码加入到 C 的键码中,来组成关系的键码。

例 3.25 在例 3.7 我们为 ODL 的类 $Movie$ 构造的关系中,除了 $Movie$ 的属性,还增加了:

1. 类 $Studio$ ($Movie$ 通过单值联系 $ownedBy$ 与之相连)的键码 $studioName$;

2. 类 Star(Movie 通过多值联系 stars 与之相连)的键码 starName。

第一项来自单值联系,对 Movie 关系的键码并没有影响。然而,第二项来自多值联系,必须加入到 Movie 关系的键码中,因此,键码就变成了:

{title, year, starName}

检验一下图 3.13 中的 Movie 关系实例,可以看出 title 和 year 本身并不是键码,但加上 starName 就足以构成键码了。

总之,如果 C 对应的关系表示了 C 中的几个多值联系,则所有这些多值联系所连接的类的键码都必须加入到 C 的键码中;得到的结果才是 C 对应的关系的键码。当然,如果多个联系将 C 与同一个类 D 相连,则 C 对应的关系中需用不同的属性表示每个从 C 到 D 的联系所连接的类 D 的键码。

由于这个悖论,如果我们按照 3.2 节所给的构成关系的方法行事的话,ODL 类的键码可能不足以构成相应关系的键码。通常我们需要修正用这种简单方法构造的关系。改善这些关系的方法将在 3.7 节介绍。我们将会看到,把“多对多”联系从任一个相连的类所对应的关系中分离出来,是可能的。最终的关系模式的聚集看起来将更像从许多 E/R 设计直接得到的关系模式。

3.5.6 本节练习

练习 3.5.1: 考虑一个关于美国人的关系,包括姓名、社会保险号、街道地址、城市、州、邮政编码(ZIP Code)、地区码以及电话号码(7 位)。你认为有哪些函数依赖?该关系的键码是什么?要回答这些问题,就需要了解这些数字分配的方法。例如,地区码能跨两个州吗?邮政编码能跨两个地区码吗?两个人的社会保险号能相同吗?他们的地址或电话号码能相同吗?

* 练习 3.5.2: 考虑一个表示封闭容器中分子当前状态的关系。属性集是一个分子的标识,包括该分子的 x, y, z 坐标和它在 x, y, z 三维空间的速度。你认为有哪些函数依赖?该关系的键码是什么?

! 练习 3.5.3: 在练习 2.3.2 中,我们讨论了联系出生(Births)的三种不同的假定。对于每种情况,标出从该联系构造的关系的键码。

! 练习 3.5.4: 假设关系 R 具有属性 A_1, A_2, \dots, A_n 。说出下列情况下, R 有多少个超键码(是 n 的函数):

- * (a) 只有键码 A_1 。
- (b) 只有键码 A_1 和 A_2 。
- (c) 只有键码 $\{A_1, A_2\}$ 和 $\{A_3, A_4\}$ 。
- (d) 只有键码 $\{A_1, A_2\}$ 和 $\{A_1, A_3\}$ 。

当然,我们也可以先把 ODL 设计转换为等价的 E/R 设计,再把 E/R 设计转换为关系设计。尽管这种方法对 3.2 节的直接转换方法固有的某些问题采取了措施,但毕竟不是根本的解决办法。而我们无论如何都需要了解的 3.7 节的关系设计技术,至少可以把这个问题解决得同样好。

3.6 函数依赖规则

这一节我们将学习如何推导函数依赖。也就是说, 假设我们已知某关系所满足的函数依赖集, 在不知道该关系的具体元组的情况下, 通常可以推断出该关系必然满足的其他某些函数依赖。在 3.7 节讨论设计良好的关系模式时, 这种揭示其他函数依赖的能力是至关重要的。

例 3.26 如果已知关系 R 拥有属性 A, B 和 C, 它满足如下函数依赖: $A \twoheadrightarrow B$ 和 $B \twoheadrightarrow C$, 则我们可以断定 R 也满足依赖 $A \twoheadrightarrow C$ 。那么我们是怎样推导的呢? 为了证明 $A \twoheadrightarrow C$, 需要考察 R 的任意两个在属性 A 上取值一致的元组, 证明它们在属性 C 上也取值一致。

设两个在属性 A 上一致的元组为 (a, b_1, c_1) 和 (a, b_2, c_2) 。假设元组中的属性顺序为 A, B, C。由于 R 满足 $A \twoheadrightarrow B$, 并且这两个元组在 A 上一致, 它们在 B 上也必然一致。也就是说, $b_1 = b_2$ 。所以这两个元组实际上就是 (a, b, c_1) 和 (a, b, c_2) , 其中 b 就是 b_1 或 b_2 。同样, 由于 R 满足 $B \twoheadrightarrow C$, 而且两个元组在 B 上一致, 则它们必然在 C 上一致。因此, $c_1 = c_2$; 也就是说, 两个元组的确在 C 上一致。我们已经证明了任意两个在 A 上一致的 R 的元组在 C 上也一致, 而这就是函数依赖 $A \twoheadrightarrow C$ 。

在不改变关系的合法实例集的条件下, 函数依赖通常可以用几种不同的方式来表示; 如果满足这种情况, 我们就称这样的两个函数依赖集是等价的(equivalent)。更一般地, 如果满足 T 中所有依赖的每个关系实例都满足 S 中的所有依赖, 我们就称函数依赖集 S “蕴含于”(follow from)函数依赖集 T。注意, 如果 S 蕴含于 T, 同时 T 也蕴含于 S, 则 S 和 T 两个函数依赖集是等价的。

本节将介绍几个有用的函数依赖规则。通常, 利用这些规则, 我们可以用等价的依赖集来代替某个依赖集, 或者在一个函数依赖集中加入蕴含于同一个函数依赖集的其他依赖集。举例来说, “转换规则”(transitive rule)就可以使我们像例 3.26 中那样跟踪函数依赖链。我们也将给出一个算法来判断一个函数依赖是否蕴含于一个或多个其他函数依赖。

3.6.1 分解/合并规则

回忆一下, 我们曾在 3.5.1 节定义了函数依赖

$$A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$$

它是下面的函数依赖集的缩写:

$$\begin{array}{l} A_1A_2...A_n \twoheadrightarrow B_1 \\ A_1A_2...A_n \twoheadrightarrow B_2 \\ \dots \\ A_1A_2...A_n \twoheadrightarrow B_m \end{array}$$

也就是说, 我们可以把每个函数依赖右边的属性分解, 从而使其右边只出现一个属性。同样, 我们也可以把左边相同的依赖的聚集用一个依赖来表示, 该依赖的左边没变, 而右边则为所有属性组成的一个属性集。两种情形下, 新的依赖集都等价于旧的依赖集。上述著

名的等价关系可用于两个方向。

- 我们可以把一个函数依赖 $A_1A_2...A_n \rightarrow B_1B_2...B_m$ 用一组函数依赖 $A_1A_2...A_n \rightarrow B_i$ ($i= 1, 2, ..., m$)来代替。这种转换称为“ 分解规则 ”。
- 我们也可以把一组函数依赖 $A_1A_2...A_n \rightarrow B_i$ ($i= 1, 2, ..., m$)用一个依赖 $A_1A_2...A_n \rightarrow B_1B_2...B_m$ 来代替。这种转换称为“ 合并规则 ”。

例如,我们在例 3. 20 中提到了下面这组函数依赖:

title year length
title year filmType
title year studioName

如何等价于一个依赖

title year length filmType studioName

也许您会想象,和右边一样,分解规则也能应用于函数依赖的左边。然而,如下例所示,并没有左边的分解规则。

例 3. 27 考虑例 3. 20 中关系 Movie 的函数依赖之一:

title year length

如果我们试图将其左边分解为:

title length
year length

我们得到了两个错误的依赖。也就是说,title 并不函数决定 length, 因为可能有两部电影同名(比如都叫 King Kong), 但长度不同。同样,year 也不函数决定 length, 因为任何一年中都必然有长度不同的电影存在。

3. 6. 2 平凡依赖

对于函数依赖 $A_1A_2...A_n \rightarrow B$ 来说,如果 B 是 A 中的某一个,我们就称之为“ 平凡的 ”。例如,

title year title

就是一个平凡依赖。

由于平凡依赖意味着“ 如果两个元组在属性 $A_1, A_2, ..., A_n$ 上全都取值一致,则它们在其中的一个属性上取值一致 ”,所以每个关系中的所有平凡依赖都保持恒真。因此,我们可以假定任何平凡依赖,而无需依靠数据来检验。

在我们最初的函数依赖定义中,并不允许出现平凡依赖。但是,鉴于它们恒为真,而且有时能简化规则的陈述,把它们包含进来并没有害处。

如果允许平凡依赖,那么也允许(作为简化形式)依赖右边的某些属性同时出现在左边。对于函数依赖 $A_1A_2...A_n \rightarrow B_1B_2...B_m$,

- 如果 B 是 A 的子集,则称该依赖为平凡的。
- 如果 B 中至少有一个属性不在 A 中,则称该依赖为非平凡的。

- 如果 B 中没有一个属性在 A 中, 则称该依赖为完全非平凡的。

因此,

$$\text{title year} \twoheadrightarrow \text{year length}$$

是非平凡的, 但不是完全非平凡的。如果把右边的 year 去掉, 我们就得到了一个完全非平凡的依赖。

如果函数依赖右边的属性中有一些也出现在左边, 那么我们就可以将右边的这些属性删除。也就是说,

- 函数依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 等价于 $A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$, 其中 C 是 B 的子集, 但属性 C 不在 A 中出现。

我们称这个规则为“平凡依赖规则”, 如图

3. 28 所示。

3. 6. 3 计算属性的闭包

在引入其他规则之前, 我们给出一个所有规则都遵循的通用原则。假设 $\{ A_1, A_2, ..., A_n \}$

图 3. 28 平凡依赖规则

是属性集, S 是函数依赖集。属性集 $\{ A_1, A_2, ...,$

$A_n \}$ 在依赖集 S 下的闭包是这样的属性集 B, 它使得满足依赖集 S 中的所有依赖的每个关系也都满足 $A_1A_2...A_n \twoheadrightarrow B$ 。也就是说, $A_1A_2...A_n \twoheadrightarrow B$ 是蕴含于 S 中的函数依赖。我们用 $\{ A_1, A_2, ..., A_n \}^+$ 来表示属性集 $A_1A_2...A_n$ 的闭包。为了简化闭包的计算, 我们允许出现平凡依赖, 所以 $A_1, A_2, ..., A_n$ 总在 $\{ A_1, A_2, ..., A_n \}^+$ 中。

图 3. 29 给出了求闭包的过程。从给定的属性集出发, 一旦包含了某函数依赖左边的属性, 就把其右边的属性增加进来, 就这样不断地扩展该集合。最终, 当该集合再也无法扩展时, 得到的结果就是闭包。下面的步骤是求解属性集 $\{ A_1, A_2, ..., A_n \}$ 在某函数依赖集下的闭包的算法的更详细描述。

1. 设属性集 X 最终将成为闭包。首先, 将 X 初始化为 $\{ A_1, A_2, ..., A_n \}$ 。
2. 然后, 重复地搜索某个函数依赖 $B_1B_2...B_m \twoheadrightarrow C$, 使得所有的 $B_1, B_2, ..., B_m$ 都在属性集 X 中, 但 C 不在其中。于是将 C 加到属性集 X 中。
3. 根据需要多次重复步骤 2, 直到没有属性能加到 X 中。由于 X 是只增的, 而任何关系的属性数目必然是有限的, 因此最终再也没有属性可以加到 X 中。
4. 最后得到的不能再增加的属性集 X 就是 $\{ A_1, A_2, ..., A_n \}^+$ 的正确值。

例 3. 28 让我们来考虑一个具有属性 A, B, C, D, E, F 的关系。假设该关系有如下的函数依赖: $AB \twoheadrightarrow C$,

图 3. 29 计算属性集的闭包

闭包算法为什么是有效的

闭包算法是有效的, 原因很简单。利用步骤 2 的运算使每个属性 D 都扩展到 X 中, 通过多次归纳, 就可以证明, 函数依赖 $A_1A_2...A_n \rightarrow D$ 是成立的(特殊情况下, 当 D 是 A 中的某个属性时, 该依赖是平凡的)。也就是说, 任何满足 S 中的所有依赖的关系 R 也满足 $A_1A_2...A_n \rightarrow D$ 。

归纳基是 0 次迭代。此时 D 必然是 A_1, A_2, \dots, A_n 中的某一个, 并且对于任何关系 $A_1A_2...A_n \rightarrow D$ 必然成立, 因为它是平凡依赖。

假设通过归纳 D 在利用依赖 $B_1B_2...B_m \rightarrow D$ 时加入 X 。由归纳假设我们知道, 对于 $i = 1, 2, \dots, m$, R 满足 $A_1A_2...A_n \rightarrow B_i$ 。另一方面, R 的任何两个在 A_1, A_2, \dots, A_n 上全都一致的元组必然在 B_1, B_2, \dots, B_m 上也全都一致。由于 R 满足 $B_1B_2...B_m \rightarrow D$, 这两个元组在 D 上也一致。因此, R 满足 $A_1A_2...A_n \rightarrow D$ 。

上面的证明说明闭包算法是有效的; 也就是说, 当把 D 加入 $\{A_1, A_2, \dots, A_n\}^+$ 时, 函数依赖 $A_1A_2...A_n \rightarrow D$ 是成立的。在这里尚未说明的是反过来的情形, 即完备性: 只要 $A_1A_2...A_n \rightarrow D$ 成立, D 必将加入 $\{A_1, A_2, \dots, A_n\}^+$ 。这个证明超出了本书的范围, 在此不予讨论。

$BC \rightarrow AD, D \rightarrow E$ 和 $CF \rightarrow B$ 。 $\{A, B\}$ 的闭包是什么, 即 $\{A, B\}^+$ 是什么?

我们从 $X = \{A, B\}$ 出发。首先, 我们注意到, 函数依赖 $AB \rightarrow C$ 左边的所有属性都在 X 中, 所以我们可以把该依赖右边的属性 C 加入到 X 中。因此, 步骤 2 的第一次迭代后, X 变成了 $\{A, B, C\}$ 。

然后, 我们看到 $BC \rightarrow AD$ 的左边现在都包含在 X 中, 因而可以把属性 A 和 D 加入到 X 中。 A 已经在 X 中了, 但 D 不在其中, 所以, X 又变成了 $\{A, B, C, D\}$ 。这时, 我们可以根据函数依赖 $D \rightarrow E$ 把 E 加入到 X 中, 现在 X 就成了 $\{A, B, C, D, E\}$ 。 X 的扩展到此为止。特别值得一提的是, 函数依赖 $CF \rightarrow B$ 没用上, 因为它的左边不包含在 X 中。因此, $\{A, B\}^+ = \{A, B, C, D, E\}$ 。

如果我们知道如何计算任意属性集的闭包的话, 就能检验给定的任一函数依赖 $A_1A_2...A_n \rightarrow B$ 是否蕴含于依赖集 S 。首先利用依赖集 S 计算 $\{A_1, A_2, \dots, A_n\}^+$ 。如果 B 在 $\{A_1, A_2, \dots, A_n\}^+$ 中, 则 $A_1A_2...A_n \rightarrow B$ 的确蕴含于 S ; 而如果 B 不在 $\{A_1, A_2, \dots, A_n\}^+$ 中, 则该依赖并不蕴含于 S 。更概括地说, 右边为属性集的依赖也可以检验, 只要我们记得它是依赖集的简化形式就有办法了。于是, 当且仅当所有的 B_1, B_2, \dots, B_m 都在 $\{A_1, A_2, \dots, A_n\}^+$ 中, $A_1A_2...A_n \rightarrow B_1B_2...B_m$ 就蕴含于依赖集 S 。

: 回忆一下 $BC \rightarrow AD$ 是一对依赖 $BC \rightarrow A$ 和 $BC \rightarrow D$ 的简化形式。愿意的话, 我们也可以分别处理这两个依赖。

闭包和键码

注意, 当且仅当 A_1, A_2, \dots, A_n 是所考虑的关系的超键码时, $\{A_1, A_2, \dots, A_n\}^+$ 才是所有属性的集合。因为只有这时, A_1, A_2, \dots, A_n 函数决定所有其他的属性。 A_1, A_2, \dots, A_n 是否是一个关系的键码, 可以这样检验, 首先检查 $\{A_1, A_2, \dots, A_n\}^+$, 它应该包含所有的属性, 然后检查从 $\{A_1, A_2, \dots, A_n\}$ 中删除任何一个属性构成的集合 S, S^+ 将不包含所有的属性。

例 3. 29 考虑例 3. 28 中的关系和函数依赖。假设我们要检验 $AB \twoheadrightarrow D$ 是否蕴含于这些函数依赖。在上例中我们计算了 $\{A, B\}^+$, 为 $\{A, B, C, D, E\}$ 。由于 D 是后一个集合的成员, 因此我们可以断定 $AB \twoheadrightarrow D$ 的确蕴含于这些函数依赖。

另一方面, 考虑函数依赖 $D \twoheadrightarrow A$ 。为了检验该依赖关系是否蕴含于给定的函数依赖, 首先计算 $\{D\}^+$ 。为此, 我们从 $X = \{D\}$ 开始计算, 利用依赖 $D \twoheadrightarrow E$ 可以把 E 加入到集合 X 中, 但 X 再也不能扩展了, 我们无法找到其左边包含在 X 中的任何其他依赖, 所以, $\{D\}^+ = \{D, E\}$ 。由于 A 不是 $\{D, E\}$ 中的成员, 因此, 我们可以断定 $D \twoheadrightarrow A$ 并不蕴含于给定的函数依赖。

3. 6. 4 传递规则

传递规则使我们级联两个函数依赖。

- 如果 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$ 和 $B_1B_2\dots B_m \twoheadrightarrow C_1C_2\dots C_k$ 在关系 R 中成立, 则 $A_1A_2\dots A_n \twoheadrightarrow C_1C_2\dots C_k$ 在 R 中也成立。

如果某个 C 在 A 中, 我们可以利用平凡依赖规则将其从右边删除。

为了说明传递规则为何成立, 我们采用 3. 6. 3 节的检验过程。要检验 $A_1A_2\dots A_n \twoheadrightarrow C_1C_2\dots C_k$ 是否成立, 就需要计算闭包 $\{A_1, A_2, \dots, A_n\}^+$ 。

函数依赖 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$ 告诉我们, B_1, B_2, \dots, B_m 全都在 $\{A_1, A_2, \dots, A_n\}^+$ 中。然后, 我们可以利用依赖 $B_1B_2\dots B_m \twoheadrightarrow C_1C_2\dots C_k$ 把 C_1, C_2, \dots, C_k 加入到 $\{A_1, A_2, \dots, A_n\}^+$ 中。因为所有的 C 都在 $\{A_1, A_2, \dots, A_n\}^+$ 中, 所以我们可以断定

$$A_1A_2\dots A_n \twoheadrightarrow C_1C_2\dots C_k$$

对于任何满足 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$ 和 $B_1B_2\dots B_m \twoheadrightarrow C_1C_2\dots C_k$ 的关系都成立。

例 3. 30 我们从图 3. 12 中的关系 *Movie* 开始, 该关系是在 3. 2. 4 节建立的, 用来表示类 *Movie* 的四个属性以及它与类 *Studio* 之间的联系 *ownedBy*。该关系及一些实例数据为:

title	year	length	filmType	studioName
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne' s World	1992	95	color	Paramount

假设我们决定在这个关系中表示有关电影所属制片公司的某些数据。为了简化起见，我们将只增加该制片公司所在的城市，表示其地址。于是，该关系就变成了这样：

Title	year	length	filmType	studioName	studioAddr
Star Wars	1977	124	color	Fox	Hollywood
Mighty Ducks	1991	104	color	Disney	Buena Vista
Wayne' s World	1992	95	color	Paramount	Hollywood

我们有理由宣称如下两个依赖成立：

$$\begin{matrix} & \text{title year} & \text{studioName} \\ & \text{studioName} & \text{studioAddr} \end{matrix}$$

第一个依赖是成立的，因为类 Movie 的联系 ownedBy 是单值的；一部电影只属于一个制片公司。第二个依赖是成立的，因为在类 Studio 中，属性 address 是单值的；它属于字符串类型(见图 2. 6)。

传递规则使我们可以把上面两个依赖组合起来，得到一个新的依赖：

$$\text{title year} \quad \text{studioAddr}$$

该依赖意味着名称和年份(即一部电影)决定一个地址——拥有该电影所有权的制片公司地址。

3. 6. 5 函数依赖的闭包

正如我们看到的那样，给定一组函数依赖，通常可以推导出某些其他依赖，既包括平凡的也包括非平凡的依赖。在后续的几节中，我们将区分“给定的”依赖——即最初说明的某个关系所满足的依赖——和“推导的”依赖——即利用本节介绍的规则之一或利用属性闭包算法推导得出的依赖。

此外，有时我们需要选择用一个关系的哪些依赖来表示它的所有依赖集。任何一个能从中导出关系的所有依赖的给定依赖集称为该关系的一个“基”。如果一个基的任何真子集都不能推导出该关系的依赖全集，则称此基为“最小的”。

例 3. 31 考虑关系 R(A, B, C)，其中，每个属性都函数决定其他两个属性。因此推导的依赖全集包括六个左、右各有一个属性的依赖；即 A → B, A → C, B → A, B → C, C → A 和 C → B。它还包括三个左边有两个属性的非平凡依赖：AB → C, AC → B 和 BC → A。还有成对依赖的简化形式，如 A → BC，也可能包含平凡依赖，如 A → A，或者不是完全非平凡的依赖，如 AB → BC(虽然在函数依赖的严格定义里，并不要求我们列出平凡或部分平凡的依赖，或者右边包含多个属性的依赖)。

上面的关系和它的依赖有几个最小基。一个是

$$\{ A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow B \}$$

另一个是

$$\{ A \rightarrow B, B \rightarrow C, C \rightarrow A \}$$

上面的关系还有许多其他的基，甚至是最小基，列举这些基，留作练习。

推导函数依赖全集的规则

如果我们想知道一个函数依赖是否蕴含于某些给定的依赖, 3. 6. 3 节的闭包计算能解决这个问题。不过, 有趣的是, 存在一组叫做“阿姆斯特朗(Armstrong)公理”的规则, 通过它可以推导出蕴含于给定依赖集的任何函数依赖。这些公理是:

自反律。如果 $\{B_1, B_2, \dots, B_m\} \subseteq \{A_1, A_2, \dots, A_n\}$, 则 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$ 。这些依赖都是平凡依赖。

增长律。如果 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$, 则对于任何属性集 $C_1, C_2, \dots, C_k, A_1A_2\dots A_n C_1C_2\dots C_k \twoheadrightarrow B_1B_2\dots B_m C_1C_2\dots C_k$ 。

传递律。如果 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$ 而且 $B_1B_2\dots B_m \twoheadrightarrow C_1C_2\dots C_k$, 则 $A_1A_2\dots A_n \twoheadrightarrow C_1C_2\dots C_k$ 。

3. 6. 6 本节练习

* 练习 3. 6. 1: 考虑关系模式为 $R(A, B, C, D)$ 的关系和函数依赖 $AB \twoheadrightarrow C, C \twoheadrightarrow D$ 和 $D \twoheadrightarrow A$ 。

- (a) 求蕴含于给定函数依赖的所有非平凡的函数依赖。
- (b) 求 R 的所有键码。
- (c) 求 R 的所有超键码(不包括键码)。

练习 3. 6. 2: 对下列关系模式和函数依赖, 按照练习 3. 6. 1 要求求解:

- (i) $S(A, B, C, D)$, 函数依赖为 $A \twoheadrightarrow B, B \twoheadrightarrow C$ 和 $B \twoheadrightarrow D$ 。
- (ii) $T(A, B, C, D)$, 函数依赖为 $AB \twoheadrightarrow C, BC \twoheadrightarrow D, CD \twoheadrightarrow A$ 和 $AD \twoheadrightarrow B$ 。
- (iii) $U(A, B, C, D)$, 函数依赖为 $A \twoheadrightarrow B, B \twoheadrightarrow C, C \twoheadrightarrow D$ 和 $D \twoheadrightarrow A$ 。

练习 3. 6. 3: 用 3. 6. 3 节的闭包检验法证明下列规则成立:

- * (a) 左边增长。如果 $A_1A_2\dots A_n \twoheadrightarrow B$ 是一个函数依赖, C 是另一个属性, 则 $A_1A_2\dots A_n C \twoheadrightarrow B$ 也成立。
- (b) 全增长。如果 $A_1A_2\dots A_n \twoheadrightarrow B$ 是一个函数依赖, C 是另一个属性, 则 $A_1A_2\dots A_n C \twoheadrightarrow BC$ 也成立。注意: 从这一规则出发, 3. 6. 5 节方框中提到的“阿姆斯特朗”(Armstrong)公理”的“增长”规则就很容易证明了。
- (c) 伪传递。假设依赖 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$ 和 $C_1C_2\dots C_k \twoheadrightarrow D$ 成立, 而且每个 B 都在 C 中, 则 $A_1A_2\dots A_n E_1E_2\dots E_j \twoheadrightarrow D$ 成立, 其中, E 是 B 中没有出现的 C 的所有属性。
- (d) 加法。如果函数依赖 $A_1A_2\dots A_n \twoheadrightarrow B_1B_2\dots B_m$ 和 $C_1C_2\dots C_k \twoheadrightarrow D_1D_2\dots D_j$ 成立, 则函数依赖

$$A_1A_2\dots A_n C_1C_2\dots C_k \twoheadrightarrow B_1B_2\dots B_m D_1D_2\dots D_j$$

也成立。上式中我们应该把同时出现在 A 和 C 或同时出现在 B 和 D 中的属性删除一份副本(即一个属性在依赖的同一侧只能出现一次)。

! 练习 3. 6. 4: 请给出满足给定依赖、但不满足假设的推导依赖的关系实例, 从而证明下

面给出的规则不是合法的函数依赖规则。

- * (a) 如果 $A \twoheadrightarrow B$, 则 $B \twoheadrightarrow A$ 。
- (b) 如果 $AB \twoheadrightarrow C$, 且 $A \twoheadrightarrow C$, 则 $B \twoheadrightarrow C$ 。
- (c) 如果 $AB \twoheadrightarrow C$, 则 $A \twoheadrightarrow C$, 或 $B \twoheadrightarrow C$ 。

！练习 3.6.5: 证明如果一个关系没有一个属性能由所有其他属性函数决定, 则该关系根本不存在非平凡的函数依赖。

！练习 3.6.6: 设 X 和 Y 是属性集。证明如果 $X \twoheadrightarrow Y$, 则 $X^+ \twoheadrightarrow Y^+$, 其中闭包是针对同一函数依赖集而言的。

！练习 3.6.7: 证明 $(X^+)^+ = X^+$ 。

！练习 3.6.8: 如果 $X^+ = X$, 则称属性集 X 是封闭的(相对于给定的函数依赖集而言)。考虑一个关系模式为 $R(A, B, C, D)$ 的关系和一个未知的函数依赖集。如果已知某些属性集是封闭的, 我们就能找出相应的函数依赖集。给出下列条件所对应的函数依赖:

- * (a) 四个属性的所有集合都是封闭的。
- (b) 仅有的闭包是 $\{A\}$ 和 $\{A, B, C, D\}$ 。
- (c) 闭包是 $\{A\}$ 、 $\{A, B\}$ 和 $\{A, B, C, D\}$ 。

！练习 3.6.9: 找出例 3.31 中的依赖和关系的所有最小基。

！！练习 3.6.10: 试说明: 如果一个函数依赖 F 蕴含于某些给定的依赖, 则我们可以从给定的依赖出发, 利用 Armstrong 公理(定义在 3.6.5 节方框内)证明 F 成立。提示: 检验计算属性集闭包的算法, 说明如何利用 Armstrong 公理推导出某些函数依赖, 来模拟该算法的每个步骤。

3.7 关系数据库模式设计

我们已经几次注意到从面向对象的 ODL 设计出发(以及在少数情况下, 从 E/R 设计出发)直接向关系数据库模式转换, 会导致关系数据库模式上的一些问题。我们发现主要的问题是冗余性, 即一个事实在多个元组中重复。而且, 我们发现造成这种冗余的最常见的原因是企图把一个对象的单值和多值特性包含在一个关系中。例如, 在 3.2.2 节, 当我们企图把电影的单值信息(如长度)和多值特性(如影星集)存储在一起的时候, 就导致了信息冗余。出现的问题见图 3.27, 现在把它复制在图 3.30 中。在那一节, 当我们试图存储一个影星的单值出生日期信息和一组地址信息时, 就发现了类似的冗余现象。

在本节, 我们将按如下步骤解决好的关系模式设计可能遇到的问题:

1. 首先, 我们应该更详细地分析关系模式有缺陷时引起的问题。
2. 然后, 我们引入“分解”的思想, 把一个关系模式(属性集)分解成两个更小的模式。
3. 其次, 我们引入“Boyce-Codd 范式”(Boyce-Codd Normal Form), 即“BCNF”, 给出一个消除这些问题的关系模式应满足的条件。
4. 在我们解释如何通过分解关系模式保证满足 BCNF 条件时, 这几点是互相联系的。

title	year	length	filmType	studioName	starName
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	Emilio Estevez
Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Meyers

图 3.30 出现异常的关系 Movie

3.7.1 异常

当我们企图把太多的信息存放在一个关系里时,出现的诸如冗余之类的问题称为“异常”(anomaly)。我们遇到的主要的几种异常是:

- 1. 冗余。信息可能在多个元组中不必要地重复。如图 3.30 中电影的长度和电影类型。
- 2. 修改异常。我们可能修改了一个元组中的信息,但另一个元组中的相同的信息却没有修改。例如,如果我们发现星球大战(Star Wars)的实际长度为 125 分钟,我们可能比较粗心,只修改了图 3.30 中的第一个元组,而没有修改第二和第三个元组。当然,也许有人会争论,不应该如此粗心。但我们将会看到,重新设计关系 Movie 从而避免出现这种错误是可能的。
- 3. 删除异常。如果一组属性的值变为空,带来的副作用可能是丢失一些其他信息。例如,如果我们从 Mighty Ducks 的影星集中删除了 Emilio Estevez,那么,数据库里就没有这部电影的影星信息了。关系 Movie 中的最后一个 Mighty Ducks 元组将会消失,同时消失的还有它的长度(104 分钟,原文误为 95 分钟)和类型(彩色)信息。

3.7.2 关系分解

消除这些异常的一个可行的方法是“分解”关系。R 的分解包括把 R 的属性分开,以构成两个新的关系模式。我们的分解规则也包括通过对 R 的元组进行投影而增加新关系的方法。描述了分解过程之后,我们将说明如何选择一个能消除异常的分解方法。

给定一个模式为{ A₁, A₂, ..., A_n}的关系 R,我们可以把 R 分解为两个关系 S 和 T,模式分别为{ B₁, B₂, ..., B_m}和{ C₁, C₂, ..., C_k} ,使得:

- 1. { A₁, A₂, ..., A_n} = { B₁, B₂, ..., B_m} ∪ { C₁, C₂, ..., C_k}。
- 2. 关系 S 中的元组是 R 的所有元组在{ B₁, B₂, ..., B_m}上的投影。也就是说,对于 R 的当前实例中的每个元组 t,取 t 在属性 B₁, B₂, ..., B_m 上的分量。这些分量构成了一个元组,它属于 S 的当前实例。然而,关系是集合,而 R 中的两个不同元组投影到 S 上,可能导致 S 中的相同元组。如果出现了这种情况,我们在 S 的当前实例中只为每个元组保留一份副本。
- 3. 类似地,关系 T 中的元组是 R 的当前实例中的元组在属性集{ C₁, C₂, ..., C_k}上的

投影。

例 3.32 让我们来分解图 3.30 中的关系 Movie。首先进行模式分解。我们选择下面两个关系,在 3.7.3 节将会看到这种选择的优点:

- 1. 一个关系称为 Movie1,其模式是除了 starName 以外的所有属性。
- 2. 另一个关系称为 Movie2,其模式包括属性 title, year 和 starName。

现在,我们通过分解图 3.30 中的采样数据来说明分解关系实例的过程。首先,我们构造在 Movie1 模式上的投影:

{ title, year, length, filmType, studioName }

图 3.30 中的前三个元组在这五个属性上的分量都相同,是:

{Star Wars, 1977, 124, color, Fox }

第四个元组的前五个分量构成一个不同的元组,而第五和第六个元组构成相同的五分量元组。结果得到的关系 Movie1 如图 3.31 所示。

title	year	length	filmT ype	studioName
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne' s World	1992	95	color	Paramount

图 3.31 关系 Movie1

其次,考虑图 3.30 在 Movie2 模式上的投影。图中的六个元组在属性 title, year 和 starName 上至少有一个分量不同,所以关系 Movie2 的结果如图 3.32 所示。

title	year	starName
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Mighty Ducks	1991	Emilio Estevez
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

图 3.32 关系 Movie2

注意,这种分解如何消除了 3.7.1 节提到的异常。冗余已经消除了;比如,每部电影的长度只在关系 Movie1 中出现一次。修改异常的危险也不存在了。例如,因为我们只需要在 Movie1 的一个元组中修改 Star Wars 的长度信息,所以该电影就不可能出现两个不同的长度。

最后,删除异常的危险也不复存在。如果我们删除了电影 Mighty Ducks 中的全部影星,也就是说,这个删除操作使得该电影在 Movie2 中消失了。但该电影的其他信息依然可以在 Movie1 中找到。

由于一部电影的名称和年份可能出现多次,看起来 Movie2 可能还存在冗余信息。然而,这两个属性构成了电影的键码,而且没有更简洁的方法来表示一部电影。此外,Movie2 并没有提供出现修改异常的机会。我们也许会这样假设:如果我们改变了 Star

Wars 的 Carrie Fisher 元组中的年份, 而另外两个元组保持不变, 于是就出现了修改异常。然而, 在我们假定的函数依赖中并不排除 1997 年存在另一部名为 Star Wars 的电影, 而 Carrie Fisher 也可能担任该电影的主演。因此, 我们不要去阻止修改一个 Star Wars 元组的年份, 而且这种修改也未必就是错误的。

3.7.3 BC 范式

分解的目的在于用几个不会导致异常的关系来代替原来的关系。已经证明, 存在一个简单的条件, 在这个条件下, 能保证避免上面所讨论的异常。该条件称为“Boyce-Codd 范式”, 或 BCNF。

- 当且仅当: 只要关系 R 有非平凡依赖 $A_1A_2...A_n \twoheadrightarrow B$, $\{A_1, A_2, ..., A_n\}$ 必然是 R 的超键码; 满足该条件的关系 R 就属于 BCNF。

也就是说, 每个非平凡函数依赖的左边必须是超键码。回忆一下, 超键码不必是最小的。因此, BCNF 条件的一个等价陈述是: 每个非平凡函数依赖的左边必须包含键码。

当我们发现一个违背 BCNF 的依赖时, 找到具有相同左边的所有其他依赖往往是有用的, 无论它们是否违背 BCNF。下面给出 BCNF 的另一种定义, 我们按这种定义查找具有相同左边的依赖集, 判断其中是否至少有一个违背 BCNF 条件的非平凡依赖。

- 当且仅当: 只要非平凡依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 在关系 R 中成立, $\{A_1, A_2, ..., A_n\}$ 必然是 R 的超键码; 满足该条件的关系 R 就属于 BCNF。

这个要求与原始的 BCNF 条件是等价的。回忆一下, 依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 是依赖集 $A_1A_2...A_n \twoheadrightarrow B_i$ (其中 $i = 1, 2, ..., m$) 的另一种形式。既然至少有一个 B_i 不在 A 中 (否则 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 就是平凡依赖), 就可以根据原始定义来判断 $A_1A_2...A_n \twoheadrightarrow B_i$ 是否违背 BCNF。

例 3.33 图 3.30 中的关系 Movie 不属于 BCNF。为了说明原因, 我们首先需要确定哪些属性是键码。在例 3.21 中我们曾讨论过为什么 $\{title, year, starName\}$ 是键码。因此, 任何包含这三个属性的属性集都是超键码。我们在例 3.21 中采用的讨论方法也可以用来解释为什么不完全包含这三个属性的属性集不可能是超键码。因此, 我们断言 $\{title, year, starName\}$ 是 Movie 的唯一的键码。

然而, 考虑函数依赖:

title year \twoheadrightarrow length filmType studioName

我们知道它在 Movie 中是成立的。回忆一下我们断言这个依赖成立的理由: 最初的 ODL 设计中有键码 $\{title, year\}$, 单值属性 length 和 filmType, 以及指向所属的制片公司的单值联系 ownedBy。

遗憾的是, 上述依赖的左边不是超键码。特别是, 我们知道 title 和 year 并不函数决定第六个属性 starName。因此, 这个依赖的存在违背了 BCNF 条件, 并告诉我们 Movie 不属于 BCNF。而且, 根据 BCNF 的原始定义, 右边只需要一个属性, 我们可以把三个函数依赖中的任何一个, 比如 title year \twoheadrightarrow length, 作为违背 BCNF 的例子。

例 3.34 另一方面, 图 3.31 中的 Movie1 属于 BCNF。因为

title year \twoheadrightarrow length filmType studioName

在该关系中成立, 并且我们曾经讨论过, title 或 year 本身都不能单独地函数决定其他任何属性, Movie1 的唯一的键码是{ title, year }。而且, Movie1 仅有的非平凡函数依赖的左边必定至少包含 title 和 year, 所以它们的左边必然是超键码。因此, Movie1 属于 BCNF。

例 3.35 我们断言任何双属性关系都属于 BCNF。为此, 需要检验右边有一个属性的可能非平凡依赖。因为需要考虑的情形并不多, 让我们逐个来讨论。在下面的讨论中, 假设这两个属性是 A 和 B。

- 1. 没有非平凡函数依赖。当然 BCNF 条件必定满足, 因为只有非平凡依赖才可能违背 BCNF 条件。顺便提一句, 这种情形下, {A, B} 是唯一的键码。
- 2. $A \twoheadrightarrow B$ 成立, 但 $B \not\rightarrow A$ 不成立。这种情形下, A 是唯一的键码, 而且任何非平凡依赖的左边必定包含 A(事实上, 左边只能是 A)。所以, 没有违背 BCNF 条件。
- 3. $B \twoheadrightarrow A$ 成立, 但 $A \not\rightarrow B$ 不成立。这种情形与情形 2 对称。
- 4. $A \twoheadrightarrow B$ 和 $B \twoheadrightarrow A$ 都成立。则 A 和 B 都是键码。当然任何依赖的左边都至少包含其中一个键码, 所以也没有违背 BCNF 条件。

值得注意的是情形 4, 这种情况下, 一个关系可能有多个键码。而 BCNF 条件只要求某个键码包含在任何非平凡依赖的左边, 并不要求所有键码都包含在左边。还应注意到有两个属性的关系, 每个都函数决定另一个, 这不是完全不可思议的。比如, 一个公司可能给每个雇员分配唯一的雇员号, 同时记录他们的社会保险号。拥有这两个属性——empID 和 ssNo 的关系中, 每个属性都函数决定另一个属性。另一方面, 每个属性都是键码, 因为我们不希望看到两个元组在任一属性上取值相同。

3.7.4 分解成 BCNF

通过不断地选择合适的分解, 我们可以把任何关系模式分解成其属性子集的聚集, 而这些子集具有如下重要特性:

- 1. 这些子集都是属于 BCNF 的关系模式。
- 2. 在某种意义上(3.7.6 节将给出确切含义), 分解后关系中的数据如实地表示原始关系中的数据。大体上讲, 我们要有能力从分解的关系准确地重构原始的关系。

例 3.35 也许给人一种印象, 我们要做的一切就是把一个关系模式分解为双属性的子集, 结果必然属于 BCNF。但我们在 3.7.6 节将会看到, 这种随意的分解并不满足条件 2。事实上, 我们必须更谨慎些, 并利用违背 BCNF 的函数依赖来指导分解过程。

我们将要采取的分解策略是寻找一个违背 BCNF 的非平凡依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$; 也就是说, { $A_1, A_2, ..., A_n$ } 不是超键码。作为试探, 我们通常在右边加入尽量多的由 { $A_1, A_2, ..., A_n$ } 函数决定的属性。图 3.33 说明如何把一个关系中的属性分为两个重叠的关系模式。一个模式包含了违背 BCNF 的依赖中的所有属性, 而另一个模式则包含了该依赖的左边以及未包含在该依赖中的所有属性, 也就是, 除了 A 以外, 再加上 B 之外的所有属性。

图 3.33 基于 BCNF 违例的关系模式分解

例 3.36 考虑我们的不断滚动的实例, 图 3.30 中的关系 Movie。我们在例 3.33 中看到,

title year length filmType studioName

是一个 BCNF 的违例。在这个实例中, 依赖的右边已经包含了由 title 和 year 函数决定的所有属性, 因此, 我们用这个 BCNF 违例把 Movie 分解成:

- 1. 包含该依赖所有属性的模式, 即:

{title, year, length, filmType, studioName}

- 2. 包含除了该依赖右边的三个属性之外的所有 Movie 属性的模式。因此, 除去 length, filmType 和 studioName, 得到了第二个模式:

{title, year, starName}

注意, 这些模式就是例 3.32 中选择的关系 Movie1 和 Movie2 的模式。我们在例 3.34 中已经看到, 它们都属于 BCNF。

例 3.37 让我们考虑例 3.30 中介绍的关系 MovieStudio。该关系存放了关于电影以及它们所属的制片公司和这些制片公司的地址等信息。其关系模式和一些典型的元组如图 3.34 所示。

title	year	length	filmType	studioName	studioAddr
Star Wars	1977	124	color	Fox	Hollywood
Mighty Ducks	1991	104	color	Disney	Buena Vista
Wayne's World	1992	95	color	Paramount	Hollywood
Addams Family	1991	102	color	Paramount	Hollywood

图 3.34 关系 MovieStudio

注意, MovieStudio 包含冗余信息。由于我们在常见的实例数据中增加了属于 Paramount(派拉蒙)的第二部电影, 因此 Paramount 的地址出现了两次。然而, 产生这个问题的原因与例 3.36 有所不同。后者是由于多值联系(给定电影中的影星)和该电影的其他信息存放在一起。而这个例子中, 所有的属性都是单值的: 表示电影长度的属性 length, 把该电影与其所属的唯一的制片公司相连的联系 ownedBy, 以及表示制片公司地址的属性 address。

在这种情况下, 存在的问题是“传递依赖”。正如我们在例 3.30 中提到的, 即关系 MovieStudio 有如下两个依赖:

title year studioName

studioName studioAddr

我们可以利用传递规则, 得到一个新的依赖

title year studioAddr

也就是说, title 和 year(即电影的键码)函数决定了制片公司的地址——拥有该电影的制片公司的地址。由于

title year length filmType

是另一个明显的依赖, 我们断定 { title, year } 是 MovieStudio 的键码; 事实上, 它是唯一的

键码。

另一方面,在上述传递规则的应用中所用到的两个依赖之一

studioName studioAddr

是非平凡的,但它的左边并不是超键码。这个事实告诉我们 MovieStudio 不属于 BCNF。利用上面的依赖,我们可以通过分解规则来解决这个问题。分解后的第一个模式是该依赖自己的属性,即:

{studioName, studioAddr }

第二个模式是除了 studioAddr(因为它是分解所用的依赖右边的属性)之外 MovieStudio 的所有属性。因此,另一个模式就是:

{title, year, length, filmType, studioName }

把图 3. 34 在这两个模式上进行投影,就得到了图 3. 35 和图 3. 36 所示的关系 MovieStudio1和 MovieStudio2。它们都属于 BCNF。MovieStudio1 的唯一键码是{ title, year },而 MovieStudio2 的唯一键码是{studioName}。在每种情形下,都没有左边不包含键码的非平凡依赖。

title	year	length	filmType	studioName
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Par amount
Addams Family	1991	102	color	Par amount

图 3. 35 关系 MovieStudio1

studioName	studioAddr
Fox	Hollywood
Disney	Buena Vista
Paramount	Hollywood

图 3. 36 关系 MovieStudio2

在前面的每个例子中,正确地应用分解规则就足以产生属于 BCNF 的关系的聚集。而一般来说,情况并不总是这样。

例 3. 38 我们把例 3. 37 中的函数依赖扩展成多于两个的函数依赖链。考虑具有如下模式的关系

{title, year, studioName, president, presAddr }

也就是说,该关系的每个元组都包含了一部电影及其制片公司、该制片公司的总裁以及该制片公司总裁的地址等信息。我们假定该关系的三个函数依赖如下:

title year studioName
studioName president
president presAddr

该关系的唯一键码是{title, year}。所以,上面的后两个依赖就违背了 BCNF。我们可

以从

studioName president

开始分解。首先, 我们应该在该函数依赖的右边增加 studioName 闭包中的任何其他属性。把传递规则用于依赖 studioName president 和 president presAddr, 我们得到:

studioName presAddr

然后把左边同为 studioName 的两个依赖组合起来, 就得到了

studioName president presAddr

该函数依赖的右边已经最大限度地扩展了, 现在我们就可以把关系模式分解为下面两个模式:

{title, year, studioName}
{ studioName, president, presAddr }

其中, 第一个模式属于 BCNF。而第二个模式唯一的键码是{studioName}, 但却拥有一个违背 BCNF 的依赖:

president presAddr

因此, 我们必须利用该依赖再次进行分解。最终我们得到了都属于 BCNF 的三个关系模式, 它们是:

{title, year, studioName}
{ studioName, president}
{ president, presAddr }

通常, 我们必须根据实际需要多次应用分解规则, 直到所有的关系都属于 BCNF 为止。我们可以肯定最终必然成功, 因为每次对关系 R 应用分解规则时, 得到的两个关系模式的属性都比 R 少。就像我们在例 3.35 中看到的, 当分解到只有两个属性的关系时, 它必然属于 BCNF。

为了说明分解为什么总是得到更小的模式, 假设我们有一个 BCNF 的违例 $A_1A_2...A_n$ $B_1B_2...B_m$ 。我们可以假定该依赖已经是扩展后的, B 中包含了由 A 函数决定的所有属性, 而且, 任何既包含在 A 中, 又包含在 B 中的属性已经从 B 中删除。

分解后的一个模式是除了 B 以外的 R 的所有属性。B 中必定至少有一个属性, 因此该模式并不会包含所有的属性。

另一个模式是 A 的所有属性和 B 的属性。这个集合不可能是 R 的属性全集, 因为, 如果是这样的话, 则{ $A_1, A_2, ..., A_n$ }就是 R 的超键码(也就是说, A 的属性能函数决定 R 的所有其他属性)。左边为超键码的依赖不是 BCNF 的违例。

因此, 可以断言分解后的两个模式都比 R 的模式小。所以, 不断分解的过程必然最终导致 BCNF 关系的聚集。

3. 7. 5 函数依赖的投影

在分解关系模式时, 我们需要检查得到的模式是否属于 BCNF。正如我们在例 3.38 中看到的, 新模式本身可能有一个甚至两个都包含 BCNF 的违例。然而, 如果我们不能确定一个关系中成立的函数依赖, 那么怎样才能判断该关系是否属于 BCNF 呢? 在例 3.38

中,我们用一种特定的方法推导了新关系中成立的函数依赖。幸运的是,有一种系统的方法可以找到分解得到的关系中成立的函数依赖。

假设把关系 R 分解为关系 S 和另一个关系。设 F 是已知的 R 中成立的依赖集。为了计算 S 中成立的函数依赖,请按照下面的步骤进行:

考虑包含于 S 的属性集的每个属性集 X 。计算 X^+ 。于是对于满足下列条件的每个属性 B , 函数依赖 $X \rightarrow B$ 在 S 中成立:

- 1. B 是 S 的一个属性,
- 2. B 属于 X^+ , 而且
- 3. B 不属于 X 。

例 3.39 设 R 的模式为 $R(A, B, C, D)$, R 中给定的函数依赖为 $A \rightarrow B$ 和 $B \rightarrow C$ 。设 $S(A, C)$ 是 R 经过某种分解得到的一个关系。我们来计算 S 中成立的函数依赖。

原则上,我们必须计算 S 的属性集 $\{A, C\}$ 的每个子集的闭包。我们从 $\{A\}^+$ 开始。很容易就能看出,该集合为 $\{A, B, C\}$ 。由于 B 不在 S 的模式中,我们并不认为 $A \rightarrow B$ 是 S 的一个依赖。然而, C 在 S 的模式中,所以我们断言依赖 $A \rightarrow C$ 在 S 中成立。

现在,我们必须考虑 $\{C\}^+$ 。因为 C 不在给定依赖的左边,它的闭包不包含任何新的属性,于是 $\{C\}^+ = \{C\}$ 。通常,一个属性集中如果不包含任何给定依赖的左边,它就不能导出 S 的任何依赖。

我们还必须考虑 $\{A, C\}^+$, 即 $\{A, B, C\}$ 。由于除了在考虑 $\{A\}^+$ 时已经得到的依赖之外,这个属性集没有引入任何新的依赖,因此,结论是 $A \rightarrow C$ 是我们需要为 S 声明的唯一依赖。当然, S 还包含从该依赖导出的其他依赖,比如 $AC \rightarrow C$ 或平凡依赖 $A \rightarrow A$ 。不过,它们可以由 3.6 节给出的规则推导出来,在我们给出 S 的函数依赖时不必特别指明。

例 3.40 现在考虑 $R(A, B, C, D, E)$ 分解为 $S(A, B, C)$ 和另一个关系。设 R 的函数依赖为 $A \rightarrow D$ 、 $B \rightarrow E$ 和 $DE \rightarrow C$ 。

首先,考虑 $\{A\}^+ = \{A, D\}$ 。由于 D 不在 S 的模式中,这个属性集没有引入任何依赖。同样, $\{B\}^+ = \{B, E\}$ 和 $\{C\}^+ = \{C\}$ 也没有为 S 引入函数依赖。

现在我们来考虑双属性子集。 $\{A, B\}^+ = \{A, B, C, D, E\}$ 。因此,我们得到了 S 的依赖 $AB \rightarrow C$ 。其他双属性子集都没有为 S 提供依赖。当然, S 的所有三个属性的集合 $\{A, B, C\}$, 也不能为 S 提供新的非平凡依赖。所以,我们需要为 S 声明的唯一依赖就是 $AB \rightarrow C$ 。

3.7.6 从分解中恢复信息

现在,把我们的注意力转移到这个问题上:为什么 3.7.4 节的分解算法能保留原始关系中包含的信息? 其思想在于,如果我们采用该算法,则原始关系的元组的投影可以重新“连接”起来,产生原始关系的所有元组并且只产生原始关系的元组。

为了把情况简化,我们考虑关系 R , 其模式为 $\{A, B, C\}$, 函数依赖为 $B \rightarrow C$, 并假定该依赖是 BCNF 的违例。像例 3.37 中的情况是可能出现的,即还有另一个函数依赖 $A \rightarrow B$, 形成一个传递依赖链。在这种情形下, $\{A\}$ 是唯一的键码, 而 $B \rightarrow C$ 的左边显然不是超键码。另一种可能性是 $B \rightarrow C$ 是唯一的非平凡依赖,在这种情形下, $\{A, B\}$ 是唯一的

简化搜索依赖的过程

当我们利用 3.7.5 节的算法从 R 的函数依赖推导关系 S 的依赖时,有时能够减少搜索过程,方法是不去计算 S 属性的所有子集的闭包。这里是有助于减少工作量的一些规则:

1. 不必考虑 S 所有属性全集的闭包。
2. 不必考虑不包含任何函数依赖左边的属性集。
3. 不必考虑包含不在任何函数依赖左边的一个属性的集合。

键码。同样, $B \rightarrow C$ 的左边也不是超键码。这两种情形下,基于依赖 $B \rightarrow C$ 进行的分解都把 R 的属性分为模式 $\{A, B\}$ 和 $\{B, C\}$ 。

设 t 是 R 的一个元组。我们可以写作 $t = (a, b, c)$, 其中, a, b, c 分别是 t 在属性 A, B, C 上的分量。元组 t 在模式为 $\{A, B\}$ 的关系上的投影为 (a, b) , 在模式为 $\{B, C\}$ 的关系上的投影为 (b, c) 。

把一个来自 $\{A, B\}$ 的元组和一个来自 $\{B, C\}$ 的元组“连接”起来是可能的,只要它们在 B 分量上一致。特别是, (a, b) 和 (b, c) 连接就又回到了最初的元组 $t = (a, b, c)$ 。无论开始时的元组 t 是什么,结论都是成立的;我们总能把 t 的投影连接起来得到原来的 t 。

但是,得到开始时的元组并不足以保证分解后的关系真实地再现原始的关系 R 。比如 R 有两个元组, $t = (a, b, c)$ 和 $v = (d, b, e)$, 情况将会如何呢? 把 t 投影到 $\{A, B\}$ 上,我们得到 $u = (a, b)$, 把 v 投影到 $\{B, C\}$ 上,得到 $w = (b, e)$, 如图 3.37 所示。

由于元组 u 和 w 在 B 分量上一致,它们可以连接,结果得到了元组 $x = (a, b, e)$ 。 x 可能是伪元组吗? 也就是说, (a, b, e) 可能不是 R 的元组吧?

由于我们假设了关系 R 存在函数依赖 $B \rightarrow C$, 结论是“否”。回忆一下,该依赖意味着在 B 分量上一致的 R 的任何两个元组,必然在 C 分量上也一致。由于 t 和 v 在 B 分量上一致(都是 b),它们在 C 分量上也一致。这意味着 $c = e$; 也就是说,我们假设不相同的两个值实际上是相同的。因此, (a, b, e) 实际上就是 (a, b, c) ; 即 $x = t$ 。

图 3.37 连接投影关系中的两个元组

既然 t 在 R 中, x 必然也在 R 中。换句话说,只要函数依赖 $B \rightarrow C$ 成立,两个投影元组连接时就不可能产生伪元组。相反地,连接生成的每个元组保证都是 R 的元组。

上面的讨论可以推广到一般。上面假定 A, B 和 C 都是单个属性,但对于它们是属性集的情况,上面的讨论也同样适用。也就是说,我们任取一个违背 BCNF 的依赖,设 B 是左边的属性, C 是右边的、而且未在左边出现的属性,而 A 是两边都没有出现的属性。我们可以得出下面的结论:

- 如果我们按照 3.7.4 节的方法对关系进行分解,则以所有可能的方式对新关系的

元组进行连接就可以准确地恢复原始关系。

如果我们不是基于一个函数依赖对关系进行分解,则可能无法恢复原始关系。下面就是一个例子。

例 3. 41 假设关系 R 的模式为(A, B, C), 这和上面一样, 但依赖 B → C 不成立。而且, R 可能包含两个元组:

A	B	C
1	2	3
4	2	5

R 在模式为{A, B}和{B, C}的关系上的投影分别是:

A	B	和	B	C
1	2		2	3
4	2		2	5

由于四个元组的 B 分量值都相同, 都是 2, 一个关系的每个元组都可以跟另一个关系的两个元组连接。于是, 当我们企图通过连接恢复 R 时, 我们得到的是:

A	B	C
1	2	3
1	2	5
4	2	3
4	2	5

也就是说, 我们得到的“太多了”; 其中有两个伪元组, (1, 2, 5) 和(4, 2, 3), 它们都不在原始关系 R 中。

3. 7. 7 第三范式

偶尔我们会遇到不属于 BCNF 的关系模式和依赖, 但却不想做进一步的分解。下面就是一个典型的例子。

例 3. 42 假设关系 Booking 的属性如下:

- 1. title, 电影名。
- 2. theater, 正在上映该电影的电影院名。
- 3. city, 电影院所在的城市。

元组(m, t, c)的意思就是一部名为 m 的电影正在 c 市的 t 影院上映。

声明该关系中有如下函数依赖是合理的:

theater → city
title city → theater

第一个依赖说明一个电影院只位于一个城市中。第二个意思不太明显, 它是基于实际

的经验,即不会在同一个城市的两个电影院预定同一部电影的票。我们声明这个依赖只是为了这个例子的需要。

首先,我们来确定键码。任何单个的属性都不是键码。例如, title 不是键码, 因为一部电影可以同时几个电影院放映, 而且可以在几个城市同时上映。 同样, theater 也不是键码, 因为虽然 theater 函数决定了 city, 但还存在多屏幕的电影院, 可以同时放映多部电影。所以, theater 并不函数决定 title。最后, city 也不是键码, 因为一个城市里通常有多个电影院, 而且会上映多部电影。

另一方面, 在三个双属性集中, 有两个是键码。显然, {title, city}是键码, 因为给定的函数依赖说明这两个属性函数决定 theater。

{theater, title}其实也是键码。为了说明原因, 我们从给定的依赖 theater → city 开始。根据练习 3. 6. 3(a) 的增长规则, 我们知道 theater → title → city 也成立。这一点是很直观的, 如果 theater 单独函数决定 city, 则 theater 加上 title 当然也函数决定 city。

剩下的一对属性, city 和 theater, 不能函数决定 title, 所以, 它们不能构成键码。我们断定只有两个键码:

{title, city}

{theater, title}

现在我们立刻就会发现一个 BCNF 的违例。给定的函数依赖中有 theater → city, 但它的左边(theater)不是超键码。因此, 我们试图根据这个违背 BCNF 的依赖把原来的关系模式分解为两个关系模式:

{theater, city}

{theater, title}

但考虑函数依赖 title → city → theater 时, 上面的分解是有问题的。可能存在与所分解的模式对应的当前关系(其模式满足依赖 theater → city, 这一点可在关系 {theater, city} 中予以检验), 但连接起来却得到了违背 title → city → theater 的关系。例如, 假设有两个关系:

theater	city	和	theater	title
Guild	Menlo Park		Guild	The Net
Park	Menlo Park		Park	The Net

根据每个关系所满足的函数依赖, 这两个关系都是允许的, 但连接时, 我们得到了违背依赖 title → city → theater 的两个元组:

Theater	city	title
Guild	Menlo Park	The Net
Park	Menlo Park	The Net

在这个例子中, 假设两部“当前的”电影不会重名, 虽然此前我们曾经考虑到不同年份制作的两部电影可能重名。

其他范式

既然存在“第三范式”，那第一、第二范式呢？实际上，人们的确定义过第一、第二范式，只不过现在很少使用。“第一范式”的条件很简单，就是每个元组的每个分量都必须是原子值。“第二范式”比 3NF 的限制稍微少一些，允许关系中存在传递依赖，但不允许非平凡依赖的左边是键码的真子集。在 3.8 节我们将看到，还有“第四范式”。

解决这个问题的方法是把 BCNF 限制稍微放宽一些，以便允许某些类似例 3.42 那样不能分解成 BCNF 关系的特殊关系模式存在，而不降低检查一个关系中的所有函数依赖的能力。放宽后的条件称为“第三范式”(third normal form)，即：

- 如果对于任何非平凡依赖 $A_1A_2...A_n \twoheadrightarrow B$ ，或者 $\{A_1, A_2, ..., A_n\}$ 是超键码，或者 B 是某个键码的组成部分，则关系 R 就属于“第三范式”(3NF)。

注意 3NF 条件和 BCNF 条件之间的区别在于句子“或者 B 是某个键码的组成部分”。这个句子“宽容了”诸如例 3.42 中的依赖 theater \twoheadrightarrow city(即允许该依赖合法存在)，因为右边的 city 是一个键码的组成部分。

证明 3NF 的确可以胜任给它的任务不在本书的讨论范围之内。也就是说，我们总可以无损地把一个关系模式分解为属于 3NF 的模式，并且所有的函数依赖都可以得到检验。不过，如果这些关系不属于 BCNF，模式中还将存在某种冗余。

有趣的是，我们观察到，我们找到的这个例子的关系模式属于 3NF，但不属于 BCNF，这个例子与以前见到的非 BCNF 的例子稍有不同。其中，有关的函数依赖 theater \twoheadrightarrow city 就是一种典型的形式，当然，这基于这样的事实：“某个电影院是定位一个城市的唯一事物”。然而，另一个依赖

title city \twoheadrightarrow theater

则来自于现实生活中电影制片公司制定的电影分配策略。通常，函数依赖分为两类：一些基于我们表示唯一事物(比如电影和制片公司)的事实，另一些基于现实世界的实践经验，比如每个城市最多在一个电影院预定一场电影票。

3.7.8 本节练习

练习 3.7.1: 对于下列的每个关系模式和函数依赖集：

- * (a) $R(A, B, C, D)$ 和函数依赖 $AB \twoheadrightarrow C, C \twoheadrightarrow D$ 和 $D \twoheadrightarrow A$ 。
- * (b) $R(A, B, C, D)$ 和函数依赖 $B \twoheadrightarrow C$ 和 $B \twoheadrightarrow D$ 。
- (c) $R(A, B, C, D)$ 和函数依赖 $AB \twoheadrightarrow C, BC \twoheadrightarrow D, CD \twoheadrightarrow A$ 和 $AD \twoheadrightarrow B$ 。
- (d) $R(A, B, C, D)$ 和函数依赖 $A \twoheadrightarrow B, B \twoheadrightarrow C, C \twoheadrightarrow D, D \twoheadrightarrow A$ 。
- (e) $R(A, B, C, D, E)$ 和函数依赖 $AB \twoheadrightarrow C, DE \twoheadrightarrow C$ 和 $B \twoheadrightarrow D$ 。
- (f) $R(A, B, C, D, E)$ 和函数依赖 $AB \twoheadrightarrow C, C \twoheadrightarrow D, D \twoheadrightarrow B$ 和 $D \twoheadrightarrow E$ 。

做如下事情：

- (i) 找出所有违背 BCNF 的依赖。切勿忘记考虑不在给定的集合中但蕴含于其中的依赖。然而，不必给出右边多于一个属性的违背 BCNF 的依赖。

- (ii) 必要时, 把给出的关系模式分解成属于 BCNF 的关系的聚集。
- (iii) 找出所有的违背 3NF 的依赖。
- (iv) 必要时, 把给出的关系模式分解成属于 3NF 的关系的聚集。

练习 3.7.2: 在 3.7.4 节我们曾经提到, 如果可能, 应该尽量扩展违背 BCNF 的函数依赖的右边。无论如何, 人们认为这是一个可选的步骤。考虑模式为属性集{A, B, C, D}的关系 R, 函数依赖为 A → B 和 A → C。因为该关系唯一的键码是{A, D}, 所以两个依赖都是 BCNF 的违例。假设我们的出发点是根据函数依赖 A → B 来分解 R。最终得到的结果跟首先把 BCNF 的违例扩展成 A → BC 一样吗? 为什么?

! 练习 3.7.3: 设 R 和练习 3.7.2 中的相同, 但函数依赖是 A → B 和 B → C。再次比较首先利用 A → B 分解和首先利用 A → BC 分解的结果。提示: 分解时, 需要考虑哪些函数依赖在分解后得到的关系中成立。只使用给定的函数依赖(仅包含分解后的一个模式中的属性)是否足够? 给定依赖的导出依赖有没有作用?

! 练习 3.7.4: 假设我们有一个关系模式 R(A, B, C), 具有函数依赖 A → B。再假设我们决定把该模式分解为 S(A, B)和 T(B, C)。给出一个关系 R 的实例, 应像 3.7.6 节那样, 让它在 S 和 T 上进行投影随后连接起来却得不到原来的关系实例。

! 练习 3.7.5: 假设我们把关系 R(A, B, C, D, E)分解成 S(A, B, C)和另一些关系。如果 R 的函数依赖如下, 请给出 S 中成立的函数依赖:

- * (a) AB → DE, C → E, D → C 和 E → A。
- (b) A → D, BD → E, AC → E 和 DE → B。
- (c) AB → D, AC → E, BC → D, D → A 和 E → B。
- (d) A → B, B → C, C → D, D → E 和 E → A。

每种情况只需要给出 S 的全依赖集的一个最小基即可。

3.8 多值依赖

“多值依赖”是一个断言, 指的是两个属性或属性集相互独立。我们以后将会看到, 这种情况是函数依赖概念的广义形式, 意味着每个函数依赖都包含一个相应的多值依赖。然而, 涉及属性集独立性的某些情况, 不能解释为函数依赖。在本节我们将寻找产生多值依赖的原因, 看看如何把多值依赖用于数据库模式设计。

3.8.1 属性的独立性及其带来的冗余

偶尔会遇到这样的情况, 我们设计一个关系模式并发现它属于 BCNF, 但该关系依然有和函数依赖无关的某种冗余。BCNF 模式中存在冗余, 最常见的原因是, 当我们用 3.2 节描述的方法直接把 ODL 模式转换成关系模式时, 某个类的两个或多个多值属性的独立性。

例 3.43 假设类 Star 的定义中包含姓名、地址集和该影星主演的电影集, 类似于图 2.5, 但属性 address 的类型不同。这里提出的 Star 的定义如图 3.38 所示。

在图 3.39 中, 我们给出了从图 3.38 的定义直接得到的关系中某些可能的元组。表示

地址集的方法与我们在图 3. 8 中表示的完全相同。该图中的元组已经扩展了属性 title 和 year 的相应分量,它们是类 Movie 的键码。这些属性通过联系主演(starredIn)表示与该影星相关的电影。

```
interface Star {
    attribute string name;
    attribute Set<
        Struct Addr {string street, string city}
    > address;
    relationship Set< Movie>  starredIn inverse Movie::stars;
}
```

图 3. 38 含有地址集和电影集影星定义

name	street	city	title	year
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Mailbu	Star Wars	1977
C. Fisher	123 Maple St.	Hollywood	Empire Strikes Back	1980
C. Fisher	5 Locust Ln.	Mailbu	Empire Strikes Back	1980
C. Fisher	123 Maple St.	Hollywood	Return of the Jedi	1983
C. Fisher	5 Locust Ln.	Mailbu	Return of the Jedi	1983

图 3. 39 与电影无关的地址集

让我们把注意力集中到图 3. 39 Carrie Fisher 的两个假设地址和三部最著名的电影上。把一个地址和一部电影相连,而不和另一部相连,是毫无道理的。因此,表示地址和电影都是影星的独立特性的唯一方法就是让每个地址都和每部电影一起出现。但如果在所有的组合中重复地址和电影信息时,显然存在冗余。例如,图 3. 39 把 Carrie Fisher 的每个地址重复了三次(每部电影对应一次),每部电影重复了两次(每个地址对应一次)。

然而,图 3. 39 给出的 Star 的关系模式中却没有 BCNF 的违例。事实上,根本就没有非平凡的函数依赖。例如,属性 city 之外的其他四个属性并不函数决定 city。可能有一个影星,有两个住处,在不同的城市里,但街道地址却完全相同。这样,就会有两个元组,它们在 city 之外的所有属性上一致,但在 city 上取值不同。因此,

```
name street title year city
```

不是关系 Star 的函数依赖。我们请读者自己去检验,Star 五个属性中没有一个可以由其他四个属性函数决定。这就足以判断根本不存在非平凡函数依赖了(读者也应该考虑,为什么这种推断是合理的?)。既然没有任何非平凡函数依赖,所以,所有的五个属性构成了唯一的键码,并且没有 BCNF 的违例。

3. 8. 2 多值依赖的定义

“多值依赖”是关于某个关系 R 的陈述,其含义是如果确定了 R 的一个属性集的取值,则其他某些特定属性的取值与该关系的所有其他属性的取值无关。更确切地说,如果我们自己限定 R 的元组在属于 A 的每个属性上取某特定的值,结果属于 B 的属性取值的

集合与既不属于 A 也不属于 B 但属于 R 的属性取值的集合无关, 则我们称多值依赖

$$A_1A_2...A_n \qquad B_1B_2...B_m$$

在关系 R 中成立。再确切些, 如果对于关系 R 中在 A 的所有属性上取值一致的每对元组 t 和 u, 我们可以在 R 中找到某个元组 v, 满足:

- 1. 和 t, u 在 A 上取值一致,
- 2. 和 t 在 B 上取值一致, 而且
- 3. 和 u 在除了 A 和 B 之外 R 的所有属性上取值一致。

则我们称这个多值依赖成立。

注意, 上面的规则中 t 和 u 可以交换, 意味着存在第四个元组 w, 它和 u 在 B 上一致, 和 t 在其他属性上一致。结果是, 对于 A 的任何固定值, B 和其他属性的相关值在不同的元组中以所有可能的组合出现。图 3. 40 说明了多值依赖成立时 v 如何与 t 和 u 相关。

通常, 我们可能假设多值依赖的 A 和 B 中的属性(左边和右边)是分开的。然而, 和函数依赖一样, 如果我们愿意, 也允许 A 中的某些属性出现在右边。还要注意, 函数依赖中我们从右边只有一个属性开始, 并且允许右边为属性集的简化形式, 在这一点上, 多值依赖与函数依赖不同, 我们必须直接考虑右边为属性集的情况。在例 3. 45 中我们将会看到, 把多值依赖的右边分解成单个的属性并不总是可行的。

图 3. 40 多值依赖保证 v 存在

例 3. 44 在例 3. 43 中我们遇到了一个多值依赖, 按我们的表示法可表示成:

$$\text{name} \qquad \text{street city}$$

也就是说, 对于每个影星的姓名, 其地址集伴随该影星主演的每部电影出现。作为运用该多值依赖正式定义的一个实例, 请考虑图 3. 39 的第一和第四个元组:

Name	street	city	title	year
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Mailbu	Empire Strikes Back	1980

如果假设第一个元组为 t, 第二个元组为 u, 则多值依赖断言我们在 R 中必然可以找到一个元组, 其 name 属性值为 C. Fisher, street 和 city 取值与第一个元组一致, 而其他属性(title 和 year) 取值与第二个元组一致。的确有这样一个元组; 它就是图 3. 39 的第三个元组。

类似地, 也可以假设上面的第二个元组为 t, 而第一个元组为 u, 则多值依赖告诉我们, R 中存在一个元组, 它在属性 name, street 和 city 上与第二个元组一致, 而在属性 name, title 和 year 上与第一个元组一致。这个元组也是存在的; 它就是图 3. 39 的第二个元组。

3. 8. 3 多值依赖的推论

关于多值依赖, 有许多规则, 和我们在 3. 6 节学过的有关函数依赖的规则类似。例如, 多值依赖遵守

- 平凡依赖规则, 即如果多值依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 在某个关系中成立, 则

$$A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$$

也成立, 其中, C 是 B 加上 A 中的一个或多个属性。反之, 我们也可以从 B 中删除一些属于 A 的属性, 并推导出多值依赖

$$A_1A_2...A_n \twoheadrightarrow D_1D_2...D_r$$

其中 D 是在 B 中而不属于 A 的属性。

- 传递规则, 即如果多值依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 和 $B_1B_2...B_m \twoheadrightarrow C_1C_2...C_m$ 在某个关系中成立, 则

$$A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$$

也成立。

但多值依赖并不遵守分解/ 合并规则的分解部分。

例 3. 45 再次考虑图 3. 39 中的关系, 其中我们看到多值依赖

$$name \twoheadrightarrow street \text{ city}$$

如果分解规则对于多值依赖成立, 则可以预料

$$name \twoheadrightarrow street$$

也成立。这个多值依赖的含义是每个影星的街道地址和其他属性(包括城市) 无关。然而, 这个陈述是错误的。例如, 考虑图 3. 39 的前两个元组。假定的多值依赖允许我们推导出这样的结果—— 街道属性值交换后的元组:

name	street	city	title	year
C. Fisher	5 Locust Ln.	Hollywood	Star Wars	1977
C. Fisher	123 Maple St.	Mailbu	Star Wars	1977

依然在关系中。但这些不是真正的元组, 例如, 5 Locust Ln. 的住所在 Malibu, 不在 Hollywood。

然而, 我们还要学习几个涉及多值依赖的新规则。首先,

- 每个函数依赖都是多值依赖。也就是说, 如果 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$, 则 $A_1A_2...A_n \rightarrow B_1B_2...B_m$ 。

要了解为什么, 先假设某个关系 R 中函数依赖 $A_1A_2...A_n \rightarrow B_1B_2...B_m$ 成立, 并假设 t 和 u 是在 A 上一致的 R 的元组。为了证明多值依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 成立, 我们必须证明 R 还包括一个元组 v, 它和 t, u 在 A 上一致, 和 t 在 B 上一致, 而且和 u 在所有其他属性上一致。但 v 可以是 u。显然 u 和 t, u 在 A 上一致, 因为我们最初的假设就是这两个元组在 A 上一致。函数依赖 $A_1A_2...A_n \rightarrow B_1B_2...B_m$ 保证了 u 和 t 在 B 上一致。而且, u 和它自己当然在其他属性上一致。因此, 只要函数依赖成立, 相应的多值依赖就成立。

另一个在函数依赖世界里没有相应规则的是“互补规则”(complementation rule):

- 如果 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 是关系 R 的多值依赖, 则 R 也满足 $A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$, 其中, C 是不属于 A, B 的 R 的所有其他属性。

例 3. 46 再次考虑图 3. 39 的关系, 其中我们声明了多值依赖

name street city

互补规则说明

name title year

在该关系中也必然成立, 因为 title 和 year 是第一个依赖没有涉及的属性。从直觉上, 可以看出, 第二个依赖指的是每个影星都主演了一组电影, 而这些电影与影星地址无关。

3. 8. 4 第四范式

如果把多值依赖用于新的关系分解算法中, 那么, 在 3. 8. 1 节发现的、由多值依赖引起的冗余是可以消除的。本节将介绍一种新范式, 称为“第四范式”(fourth normal form)。在这种范式里, 随着违背 BCNF 的所有函数依赖的消除, 所有的“非平凡”(nontrivial)(在下面所定义的意义下)多值依赖也都消除了。结果是分解后的关系既不含我们在 3. 7. 1 节曾讨论过的、由函数依赖引起的冗余, 也不含我们在 3. 8. 1 节所讨论的、由多值依赖引起的冗余。

如果:

1. B 中的属性都不在 A 中;
2. A 和 B 并未包含 R 的所有属性。

则关系 R 的多值依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 就是“非平凡的”(nontrivial)。

“第四范式”条件实质上是 BCNF 条件, 但它应用于多值依赖而不是函数依赖。正式地陈述如下:

- 如果 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 是非平凡的多值依赖, 且 $\{ A_1, A_2, ...A_n \}$ 是超键码, 则关系 R 就属于第四范式(4NF)。

也就是说, 如果一个关系属于 4NF, 则每个非平凡多值依赖实际上就是一个左边为超键码的函数依赖。注意, 键码和超键码的概念只和函数依赖有关; 增加多值依赖并不改变“键码”的定义。

例 3. 47 图 3. 39 的关系违背了 4NF 条件。例如,

name street city

是非平凡的多值依赖, 而 name 本身不是超键码。实际上, 该关系的所有属性是唯一的键码。

第四范式实际上是 BCNF 的广义形式。回忆一下, 我们在 3. 8. 3 节讲过, 每个函数依赖也是一个多值依赖。因此, 每个 BCNF 的违例也是一个 4NF 的违例。换句话说, 属于 4NF 的每个关系都必然属于 BCNF。

然而, 有些属于 BCNF 的关系却不属于 4NF。图 3. 39 就是一个很好的例子。该关系

多值依赖的投影

进行第四范式分解时, 我们需要找到在分解后的关系中成立的多值依赖。我们希望有更简单的方法找到这些依赖。然而, 像函数依赖中计算属性集的闭包之类的简单检验方法(见 3.6.3 节)是不存在的。实际上, 即使是关于函数和多值依赖集的推导规则的完备集, 就已经相当复杂了, 并已超出本书的讨论范围。参考文献部分提到了一些处理该主题文献。

幸运的是, 我们常常能利用传递规则、互补规则和交集规则(intersection rule) [练习 3.8.7(b)] 得到与分解的结果相应的多值依赖。我们建议读者通过实例和练习对此作进一步尝试。

的唯一键码是所有五个属性, 而且没有非平凡函数依赖。因此, 它确实属于 BCNF。然而, 正如我们在例 3.47 中所看到的, 它不属于 4NF。

3.8.5 分解成第四范式

4NF 的分解算法与 BCNF 的分解算法非常类似。首先, 我们要找到一个 4NF 的违例, 比如 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$, 而 $\{A_1, A_2, ..., A_n\}$ 不是超键码。注意: 这个多值依赖可能的确是一个多值依赖, 也可能是从相应的函数依赖 $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ 导出的多值依赖, 因为每个函数依赖都是一个多值依赖。然后, 我们把含有 4NF 违例的关系 R 的模式分解为两个模式:

- 1. A 和 B 中的属性;
- 2. A 中的属性以及既不属于 A 也不属于 B 的 R 的所有其他属性。

例 3.48 让我们继续研究例 3.47。我们观察到

name street city

是一个 4NF 的违例。上面的分解规则告诉我们, 可用两个模式来代替原来的包含五个属性的关系模式—— 一个模式只包含上面的多值依赖所涉及的三个属性, 另一个模式由该依赖的左边, 即 name, 加上未在该依赖中出现的属性组成。这些属性是 title 和 year。因此, 下面的两个模式

{ name, street, city }
{ name, title, year }

就是分解的结果。每个模式中都没有非平凡多值(或函数)依赖, 所以它们属于 4NF。注意, 在模式为 { name, street, city } 的关系中, 多值依赖

name street city

是平凡的, 因为它包含了所有的属性。类似地, 在模式为 { name, title, year } 的关系中, 多值依赖

name title year

也是平凡的。如果有一个或两个分解后的模式不属于 4NF, 我们就必须把非 4NF 的模式继续分解。

对于 BCNF 分解, 每一步分解所得的模式的属性数目都绝对比分解前更少, 因此, 最终我们将得到不需要进一步分解的模式; 也就是说, 它们属于 4NF。而且, 我们在 3. 7. 6 节给出的证明分解正确的论证也适用于多值依赖。当我们由于多值依赖

$$A_1A_2...A_n \qquad B_1B_2...B_m$$

而分解关系模式时, 该依赖足以用来证明这一断言是正确的, 即我们可以从分解后的关系重构原始的关系。

3. 8. 6 范式间的联系

正如我们曾经提到的, 4NF 隐含 BCNF, 而 BCNF 又隐含 3NF。因此, 对于任何关系模式而言, 满足三个范式的关系实例集之间的联系如图 3. 41 所示。也就是说, 如果一组元组满足 4NF 条件, 那么它必然满足其他两个范式条件, 而且如果它满足 BCNF 条件, 那么它必然属于 3NF。然而, 随着该模式所假设的函数依赖不同, 有一些元组的集合可能属于 3NF, 而不属于 BCNF。类似地, 对于假设的函数和多值依赖的特定集合, 有一些元组的集合属于 BCNF, 而不属于 4NF。

另一种对范式进行比较的方法是, 比较它们为分解成相应的范式后得到的一组关系所做的保证。这方面的观察结果汇总在图 3. 42 的表中。也就是说, BCNF(从而 4NF) 消除了由函数依赖带来的冗余和其他异常, 而只有 4NF 消除了由于非平凡多值依赖、而不是函数依赖的存在而带来的附加冗余。

图 3. 41 4NF 隐含 BCNF 隐含 3NF

通常, 3NF 就足以消除这种冗余了, 但有些例子中, 它也无能为力。我们总是可以选择分解成 3NF, 从而保持函数依赖; 也就是说, 它们在分解后的关系中依然成立(虽然本书中并未讨论这么做的算法)。BCNF 不保证函数依赖的保持性, 并且, 没有一个范式保证多值依赖的保持性, 虽然在典型的情形下, 依赖是保持的。

特性	3NF	BCNF	4NF
消除函数依赖引起的冗余	大部分	是	是
消除多值依赖引起的冗余	否	否	是
保持函数依赖	是	可能	可能
保持多值依赖	可能	可能	可能

图 3. 42 范式及其分解的特性

3. 8. 7 本节练习

* 练习 3. 8. 1: 假设我们有一个关系 $R(A, B, C)$, 存在多值依赖 $A \twoheadrightarrow B$ 。如果已知元

组 (a, b_1, c_1) , (a, b_2, c_2) 和 (a, b_3, c_3) 在 R 的当前实例中, 我们可以判断还有哪些元组必然也在 R 中?

* 练习 3.8.2: 假设我们有一个关系, 用以记录每个人的名字、社会保险号和出生日期。还包括他/她的每个孩子的名字、社会保险号和出生日期, 以及他/她所拥有的每辆车的序号和型号。更确切地说, 该关系的所有元组形式如下:

$$(n, s, b, cn, cs, cb, as, am)$$

其中,

- 1. n 是社会保险号为 s 的人的名字。
- 2. b 是 n 的出生日期。
- 3. cn 是 n 的一个孩子的名字。
- 4. cs 是 cn 的社会保险号。
- 5. cb 是 cn 的出生日期。
- 6. as 是 n 的一辆车的序号。
- 7. am 是序号为 as 的汽车的型号。

对于这个关系

- (a) 写出我们期望保持的函数依赖和多值依赖。
- (b) 给出把该关系分解成 4NF 的结果。

练习 3.8.3: 对于下面的每个关系模式和依赖:

- (a) $R(A, B, C, D)$, 存在多值依赖 $A \twoheadrightarrow B$ 和 $A \twoheadrightarrow C$ 。
- (b) $R(A, B, C, D)$, 存在多值依赖 $A \twoheadrightarrow B$ 和 $B \twoheadrightarrow CD$ 。
- (c) $R(A, B, C, D)$, 存在多值依赖 $AB \twoheadrightarrow C$ 和函数依赖 $B \rightarrow D$ 。
- (d) $R(A, B, C, D, E)$, 存在多值依赖 $A \twoheadrightarrow B$, $AB \twoheadrightarrow C$ 和函数依赖 $A \rightarrow D$ 和 $AB \rightarrow E$ 。

做如下练习:

- (i) 找出所有 4NF 的违例。
- (ii) 把关系分解为属于 4NF 的关系模式的聚集。

! 练习 3.8.4: 在练习 2.3.2 中, 我们讨论了关于联系 Births 的四个不同假设。对于每个假设, 指出希望在结果关系中保持的多值依赖(而不是函数依赖)。

练习 3.8.5: 为什么不希望例 3.43 中的五个属性中的任一个由其他四个函数决定, 请给出非正式论证。

! 练习 3.8.6: 用多值依赖的定义, 证明互补规则为什么成立。

! 练习 3.8.7: 证明下列多值依赖规则:

- * (a) 并集规则(union rule)。如果 X, Y 和 Z 是属性集, $X \twoheadrightarrow Y$ 并且 $X \twoheadrightarrow Z$, 则 $X \twoheadrightarrow (Y \cup Z)$ 。
- (b) 交集规则(intersection rule)。如果 X, Y 和 Z 是属性集, $X \twoheadrightarrow Y$ 并且 $X \twoheadrightarrow Z$, 则 $X \twoheadrightarrow (Y \cap Z)$ 。
- (c) 差集规则(difference rule)。如果 X, Y 和 Z 是属性集, $X \twoheadrightarrow Y$ 并且 $X \twoheadrightarrow Z$, 则 $X \twoheadrightarrow (Y - Z)$ 。

- (d) 平凡多值依赖。如果 $Y \twoheadrightarrow X$, 则 $X \twoheadrightarrow Y$ 在任何关系中都成立。
- (e) 平凡多值依赖的另一个来源。如果 $X \twoheadrightarrow Y$ 是关系 R 的全部属性, 则 $X \twoheadrightarrow Y$ 在 R 中成立。
- (f) 消去左、右边共同的属性。如果 $X \twoheadrightarrow Y$ 成立, 则 $X \twoheadrightarrow (Y - X)$ 也成立。

！练习 3.8.8: 给出反例关系, 说明下列多值(或函数)依赖规则为什么不成立。

- * (a) 如果 $A \twoheadrightarrow BC$, 则 $A \twoheadrightarrow B$ 。
- (b) 如果 $A \twoheadrightarrow B$, 则 $A \twoheadrightarrow B$ 。
- (c) 如果 $AB \twoheadrightarrow C$, 则 $A \twoheadrightarrow C$ 。

！练习 3.8.9: 从 ODL 到关系的转换常常会引入多值依赖。给出采用 3.2.2 节和 3.2.5 节的关系模式策略时, 从多值的属性和联系中发现多值依赖的一些原则。

3.9 数据库模式实例

我们已经看到了直接从 ODL 或 E/R 设计构造关系时可能产生的各种问题, 也已经看到了如何处理有时出现的异常, 现在让我们专注于一个关系的数据库模式上, 在本书下一部分(致力于用户的数据库编程方面)的实例中将会用到这一数据库模式。我们的数据库模式建立在电影、影星和制片公司不断滚动的实例的基础上, 采用规范化的关系, 类似于前面章节建立的那些关系。然而, 它还包含了前面的实例中没有出现的某些属性, 而且还包含一个以前没有出现的关系——MovieExec。这些改动的目的是为了在第 4 到第 8 章的实例中, 给我们提供一些学习表示信息的不同数据类型和不同方法的机会。图 3.43 给出了这个数据库模式。

我们的关系模式包括五个关系。每个关系的属性及其指定的域都列出来了。其中, 每个关系的键码属性在图 3.43 中都用大写表示, 不过在叙述过程中提到它们时, 它们还像以前一样, 都是小写的。例如, 关系 StarsIn 的所有三个属性一起构成键码。关系 Movie 具有六个属性: title 和 year 一起构成了它的键码, 这都和以前一样。属性 title 是字符串, 而 year 是整数。

与我们以前见过的关系模式相比, 该模式的主要改动如下:

- 对于电影行政长官(制片公司总裁和电影制片人)而言, 有 certificate number(证书号)的概念。这个证书号是我们假想的唯一的整数, 它由某个外部的权威机构维护, 该机构或许是行政长官注册处或“协会”。
- 我们将证书号作为电影行政长官的键码。即使是电影明星也不一定都有证书号, 所以我们仍然用 name 作为影星的键码。这种决定可能有些不切实际, 因为两个影星可能重名。但为了说明某些不同的选项, 我们还是采用这种方法。
- 我们为电影引入了另一个特性——制片人, 他的信息由关系 Movie 的新属性 producerC# 表示。该属性应是制片人的证书号。制片人应该是和制片公司总裁一样的电影的行政长官。在 MovieExec 关系中也可能有其他的行政长官。
- Movie 的属性 filmType 由枚举类型改成布尔值的属性, 称为“inColor”: 彩色电影, 其值为“真”(true); 黑白电影, 其值为“假”(false)。这么做的原因是有些数据

库语言不支持枚举类型。

- 为电影明星增加了属性 gender(性别)。它的类型是“字符”,或者是‘M’,表示“男”,或者是‘F’,表示“女”。还增加了属性 birthdate(出生日期),它的类型是 date(日期)——许多商用数据库系统支持的一种特殊类型,或者,如果我们愿意,也可以只是一个字符串。
- 所有的地址都使用字符串,而不是街道和城市的组合。这样做使得不同关系中的地址容易比较,而且简化了地址操作。

最后作为总结,我们对五个关系及其属性以及这些关系从以前的 ODL 或 E/R 设计的演变过程给出简单的注释。

```
Movie (
    TITLE: string,
    YEAR: integer,
    length: integer,
    inColor: boolean,
    studioName: string,
    producerC# : integer )

StarsIn (
    MOVIE TITLE: string,
    MOVIE YEAR: integer,
    STARNAME: string )

MovieStar (
    NAME: string,
    address: string,
    gender: char,
    birthdate: date )

MovieExec (
    name: string,
    address: string,
    CERT# : integer,
    netWorth: integer )

Studio (
    NAME: string,
    address: string,
    presC# : integer )
```

图 3.43 关于电影的数据库模式实例

1. Movie 是例 3.36 的关系 Movie 分解得到的关系之一,我们在其中增加了属性 producerC# ,表示该电影的制片人。

2. StarsIn 是例 3.36 的关系分解得到的另一个关系。如果我们从同名的 ODL 类定义出发构造关系 Star,然后将其分解为 BCNF,也需要同样的关系 StarsIn。也就是说,从图 2.5 的 ODL 定义出发,我们将会得到具有属性 name, address, title 和 year 的关系 Star。后两个属性代表联系 starredIn。我们发现{ name, title, year }是键码,但存在函数依

赖 name address。因此, 该关系将分解为模式 { name, address }(已扩展为关系 MovieStar)和 { name, title, year }(实质上就是关系 StarsIn)。我们的关系 StarsIn 也表示了图 2.8 的 E/R 图中的联系 Stars-in。

3.10 本章总结

关系模型: 关系是表示信息的表。列以属性开头; 每个属性都有相关的域或数据类型。行称为元组, 对于关系的每个属性, 元组都有一个分量与之对应。

模式: 关系名和关系的属性一起构成了关系模式。关系模式的聚集构成了关系数据库模式。一个关系或关系聚集对应的特定数据称为该关系模式或该数据库模式的实例。

实体集转换为关系: 实体集对应的关系中, 对于实体集的每个属性, 关系中都有一个属性与之对应。弱实体集是例外, 弱实体集 E 对应的关系中还必须包含有助于标识 E 的实体的其他实体集的键码属性。

联系转换为关系: E/R 联系对应的关系中包含参与该联系的每个实体集的键码属性所对应的属性。

ODL 类转换为关系: ODL 类 C 对应的关系中, 对于类的每个属性, 关系中都含有一个属性与之对应。该关系中还可能包含通过某个联系与 C 相连的类 D 的键码属性。由于 ODL 的联系存在反向联系, 我们建议把 C 到 D 的多对一联系放在 C 中, 而不放在 D 中。

ODL 多对多联系转换为关系: 多对多的联系可以放在两个类的任一个中, 但它们会造成关系的元组数目激增。关系设计中的这个缺陷可以通过规范化过程来消除。另一种方法是, ODL 中的多对多联系可以像 E/R 模型那样, 用单独的关系来表示。

子类结构转换为关系: 一种方法是把实体或对象分散在不同的子类中, 并为每个子类建立一个关系, 其中包含所有必要的属性。第二种方法是用一个主关系来表示所有的实体或对象, 这种主关系只包含最通用的类的属性。而子类的实体或对象则用对应于它们所属的子类的特定关系来表示。这些关系只包含通用类的键码属性和该子类的特有属性。

函数依赖: 函数依赖是对于关系中在某个特定的属性集上一致的两个元组必然在某个其他特定属性上也一致的陈述。

键码: 一个关系的超键码是函数决定该关系所有属性的属性集。键码是其任何真子集都不能函数决定所有属性的超键码。

函数依赖的推论: 存在许多规则, 我们可以藉此推出函数依赖 $X \twoheadrightarrow A$ 在满足其他某个给定函数依赖集的任何关系实例中成立。证明 $X \twoheadrightarrow A$ 成立, 通常最简单的方法是计算 X 的闭包, 即利用给定的函数依赖扩展 X , 直到包含 A 。

分解关系: 我们可以把一个关系模式分解为两个。只要分解得到的两个关系模式中至少有一个, 其属性构成一个超键码, 就可能不损失信息。

BC 范式: 如果一个关系中仅有的非平凡函数依赖都表明是某个超键码函数决定其他某个属性, 则该关系属于 BCNF。把任何关系分解为 BCNF 关系的聚集而不损失信息是可能的。BCNF 的主要优点是消除了由函数依赖带来的冗余。

第三范式: 有时, 分解成 BCNF 会妨碍我们检验某些特定的函数依赖。BCNF 的一种宽松形式, 称为“3NF”, 即使 X 不是超键码, 但只要 A 是某个键码的组成部分, 就允许存在函数依赖 $X \rightarrow A$ 。3NF 并不能保证消除由函数依赖带来的所有冗余, 但通常情况下能做到全部消除。

多值依赖: 多值依赖是对于一个关系的两个属性集取值的集合出现各种可能组合的陈述。产生多值依赖的常见原因是所设计的关系代表了含有两个或多个多值的属性或联系的 ODL 类。

第四范式: 关系中的多值依赖也可能带来冗余。4NF 类似于 BCNF, 但还禁止非平凡多值依赖(除非它们实际上就是 BCNF 允许的函数依赖)。把一个关系分解为 4NF 而不损失任何信息是可能的。

3.11 本章参考文献

Codd 的关于关系模型的经典论文是[4]。该论文除了基本关系概念之外, 还介绍了函数依赖的思想。其中也描述了第三范式, 而 Boyce-Codd 范式是 Codd 在稍后的一篇文章[5]中描述的。

多值依赖和第四范式是 Fagin 在[7]中定义的。不过, 多值依赖的思想也独立地出现在[6]和[9]中。

Armstrong(阿姆斯特朗)首先研究了函数依赖推导规则(见[1])。本章所涉及的函数依赖规则(包括我们称之为“Armstrong(阿姆斯特朗)公理”的规则)以及多值依赖推导规则, 都来自[2]。通过计算属性集的闭包来检验函数依赖的技术来自[3]。

还有许多算法和/或该算法的可行性证明在本书中并未给出。这些内容包括对以下各种问题的解释: 推导函数依赖的闭包算法何以成立, 如何推导多值依赖, 如何把多值依赖投影到分解后的关系上, 以及在不损失检验函数依赖的能力的情况下如何分解成 3NF。这些和其他有关依赖的内容在[8]中均有解释。

- [1] Armstrong, W. W., Dependency structures of database relationships, Proceedings of the 1974 IFIP Congress, pp. 580 ~ 583.
- [2] Beeri, C., R. Fagin, and J. H. Howard, A complete axiomatization for functional and multivalued dependencies, ACM SIGMOD International Conference on Management of Data, pp. 47 ~ 61, 1977.
- [3] Bernstein, P. A., Synthesizing third normal form relations from functional dependencies, ACM Transactions on Database Systems, 1:4, pp. 177 ~ 298, 1976.
- [4] Codd, E. F., A relational model for large shared data banks, Comm. ACM 13:6, pp. 377 ~ 387, 1970.
- [5] Codd, E. F., Further normalization of the database relational model, in Database Systems (R.

Rustin, ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972.

- [6] Delobel, C. , Normalization and hierarchical dependencies in relational data model, ACM Transactions on Database Systems, 3:3, pp. 201 ~ 222, 1978.
- [7] Fagin, R. , Multivalued dependencies and a new normal form for relational databases, ACM Transactions on Database Systems, 2:3, pp. 262 ~ 278, 1977.
- [8] Ullman, J. D. , Principles of Database and Knowledge-Base Systems, Volume I, Computer Science Press, New York, 1988.
- [9] Zaniolo, C. , and M. A. Melkanoff, On the design of relational database schemata, ACM Transactions on Database Systems, 6:1, pp. 1 ~ 47, 1981.

第 4 章 关系模型中的运算

在这一章, 我们开始从用户的观点学习数据库。通常, 用户的主要问题是查询数据库, 也就是编写程序来回答关于数据库当前实例的询问。在这一章, 我们将从抽象的观点出发学习数据库查询的问题, 定义主要的查询运算符。

虽然 ODL 使用原则上可以对数据进行任何运算的方法, 而 E/R 模型不包含数据操作的特定方式, 但关系模型有对数据进行“标准”运算的具体集合。因此, 我们对数据库运算的学习理论上将集中于关系模型及其运算。这些运算可以用代数(称为关系代数)表示, 也可以用逻辑的形式(称为“Datalog”)表示。我们将在本章学习这两种表示法中的每一种。

以后章节使我们看到今天的商业数据库系统向用户提供的语言和特性。抽象查询运算符将主要出现在第 5 至 7 章讨论的 SQL 查询语言的运算中。然而, 它们也在第 8 章提到的 OQL 语言中出现。

4.1 关系代数

为了开始关于关系运算的学习, 我们应该学习一种特殊的代数, 称为关系代数 (relational algebra), 它包含某些简单而有效的方法来从旧关系构造新关系。关系代数中的表达式 (expression) 首先从作为运算数的关系开始; 关系可以用关系名 (例如, R 或 Movies) 表示, 也可以显式地表示为其元组的列表。然后, 我们可以通过把下面将要描述的任何运算符用于关系或用于关系算术的较简单的表达式, 逐渐构造更为复杂的表达式。一个查询 (query) 就是关系算术的一个表达式。因此, 关系代数就是查询语言的第一个具体实例。

关系代数运算分为四大类:

1. 普通的集合运算——并、交和差——用于关系。
2. 删除部分关系的运算:“选择”将删除某些行(元组), 而“投影”则删除某些列。
3. 合并两个关系元组的运算, 包括笛卡尔积——把两个关系的元组以所有可能的方式组成对, 以及各种“连接”运算——从两个关系的元组中有选择地组成对。
4. 称为“改名”的运算并不影响关系的元组, 但是改变关系模式, 也就是改变属性的名字和/或关系本身的名字。

这些运算并不足以对关系做任何可能的计算; 实际上, 它们是很有限的。但是, 它们包括了我们真正想对数据库进行的大部分处理, 它们构成了标准关系查询语言 SQL 的主要部分, 正如我们将在第 5 章看到的那样。然而, 我们将在 4.6 和 4.7 节简要讨论存在于实际查询语言(如 SQL)之中但尚不属于关系代数部分的某些计算能力。

4. 1. 1 关系的集合运算

集合的三个最普通的运算是并、交和差。我们假定读者都熟悉这些运算, 对于任意集合 R 和 S, 这些运算定义如下:

- $R \cup S$, R 和 S 的并, 是在 R 或 S 或两者中元素的集合。一个元素在并集中只出现一次, 即使它在 R 和 S 中都存在。
- $R \cap S$, R 和 S 的交, 是 R 和 S 中都存在的元素的集合。
- $R - S$, R 和 S 的差, 是在 R 中而不在 S 中的元素的集合。注意: $R - S$ 不同于 $S - R$, 后者是在 S 中而不在 R 中的元素的集合。

当我们对关系进行这些运算时, 需要把某些条件加在 R 和 S 上,

1. R 和 S 的模式必须具有相同属性集。
2. 在计算元组集合的集合论并集、交集或差集之前, R 和 S 的列需要排序, 以使两个关系的属性顺序相同。

有时我们希望对属性数相同但属性名不同的关系进行并、交或差运算。如果这样, 就可以利用将在 4. 18 节讨论的改名运算符来改变一个或两个关系的模式给它们以相同的属性集。

例 4. 1 假定我们有两个关系 R 和 S, 这是 3. 9 节关系 MovieStar 的实例。R 和 S 的当前实例在图 4. 1 中给出。

名字	地址	性别	出生日期
Carrie Fisher	123 Maple St. , Hollywood	F	9/ 9/ 99
Mark Hamill	456 Oak Rd. , Brentwood	M	8/ 8/ 88

关系 R

名字	地址	性别	出生日期
Carrie Fisher	123 Maple St. , Hollywood	F	9/ 9/ 99
Harrison Ford	789 Palm Dr. , Beverly Hills	M	7/ 7/ 77

关系 S

图 4. 1 两个关系

并集 $R \cup S$ 如下所示:

名字	地址	性别	出生日期
Carrie Fisher	123 Maple St. , Hollywood	F	9/ 9/ 99
Mark Hamill	456 Oak Rd. , Brentwood	M	8/ 8/ 88
Harrison Ford	789 Palm Dr. , Beverly Hills	M	7/ 7/ 77

注意, 两个关系中有关 Carrie Fisher 的两个元组在结果中只出现一次。

交集 $R \cap S$ 如下所示:

名字	地址	性别	出生日期
Carrie Fisher	123 Maple St. , Hollywood	F	9/ 9/ 99

现在, 只有一个关于 Carrie Fisher 元组, 因为只有它出现在两个关系中。差集 R—S 如下所示:

名字	地址	性别	出生日期
Mark Hamill	456 Oak Rd. , Brentwood	M	8/ 8/ 88

也就是说, Fisher 和 Hamill 元组出现在 R 中, 因此是 R—S 的候选元组。然而, Fisher 元组也出现在 S 中, 因此不在 R—S 中。

4. 1. 2 投影

投影运算符用来从关系 R 产生一个只有 R 的某些列的新关系。表达式 $\pi_{A_1, A_2, \dots, A_n}(R)$ 的值是一个关系, 该关系只有 R 的属性 A_1, A_2, \dots, A_n 所对应的列。结果值的模式是属性集 $\{A_1, A_2, \dots, A_n\}$, 按照惯例以列出的顺序表示它。

例 4. 2 考虑具有在 3. 9 节描述的关系模式的关系 Movie。该关系的实例在图 4. 2 中给出。我们可以用表达式

$$\pi_{\text{title, year, length}}(\text{Movie})$$

将该关系投影到前三个属性上。

title	year	length	inColor	studioName	producerC#
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne' s World	1992	95	true	Paramount	99999

图 4. 2 关系 Movie

结果关系如下所示:

title	year	length
Star Wars	1977	124
Mighty Ducks	1991	104
Wayne' s World	1992	95

作为另一个例子, 我们可以用表达式 $\pi_{\text{inColor}}(\text{Movie})$ 投影到属性 inColor 上。结果是单列关系

$$\frac{\text{inColor}}{\text{true}}$$

注意, 在结果关系中只有一个元组, 因为图 4. 2 的所有三个元组在属性 inColor 的分量值相同。

4. 1. 3 选择

选择运算符应用到关系 R, 将产生作为 R 元组的子集的新关系。结果关系中的元组是满足某种条件 C 的元组, 该条件与关系 R 属性有关。我们把这种运算记为 $c(R)$ 。结果关系的模式和 R 的模式相同, 并按照惯例用和 R 所用的同样的顺序表示属性。

C 是我们从传统的编程语言熟悉的条件表达式, 例如, 在编程语言 C 或 Pascal 中, 条件表达式在关键字 if 的后面。唯一的不同在于条件 C 中运算项是常量或 R 的属性。我们通过用 R 的元组 t 中每个属性 A 对应的分量, 代替出现在条件 C 中的每个属性 A, 把条件 C 用于 R 的每个元组 t。如果代替条件 C 的每个属性以后条件 C 为真, 那么, 元组 t 就是出现在 $c(R)$ 的结果中的元组之一; 否则 t 不出现在结果中。

例 4. 3 让关系 Movie 如图 4. 2 那样。那么表达式 $length \geq 100 (Movie)$ 的值是

title	year	length	inColor	studioName	producerC#
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890

第一个元组满足条件 $length \geq 100$, 因为当我们用第一个元组中属性 length 的分量值 124 来代替 length, 条件就变成 $124 \geq 100$ 。后面的条件为真, 所以我们取第一个元组。可以用同样的理由说明图 4. 2 的第二个元组为什么也出现在结果中。

第三个元组的 length 分量为 95。于是, 当我们代替 length 时, 条件变成 $95 \geq 100$, 为假。因此, 图 4. 2 的最后一个元组未出现在结果中。

例 4. 4 假设我们想要在关系

$Movie(title, year, length, inColor, studioName, producerC#)$

中表示 Fox 公司最少 100 分钟长的电影, 我们可以用一个较复杂的条件得到它, 该条件包含两个子条件的与(AND)。表达式如下:

$$length \geq 100 \text{ AND } studioName = 'Fox' (Movie)$$

如下元组

Title	Year	length	inColor	studioName	ProducerC#
Star Wars	1977	124	true	Fox	12345

是结果关系中仅有的一个。

4. 1. 4 笛卡尔积

两个集合 R 和 S 的笛卡尔积(或只是乘积)是元素对的集合, 该元素对是通过选择 R 的任何元素作为第一个元素, S 的元素作为第二个元素构成的。该乘积用 $R \times S$ 表示。当 R 和 S 是关系时, 乘积本质上相同。然而, 因为 R 和 S 的成员是元组, 通常包含多个分量, 由 R 的元组和 S 的元组构成的元组对是一个更长的元组, 其中每个分量都对应于组成元组的一个分量。按现在的顺序, R 的分量在 S 的分量之前。

结果关系的关系模式是 R 和 S 的模式 的并集。然而, 如果 R 和 S 偶然有某些公共属性, 那么, 我们需要为每个相同属性对的至少一个属性引入新名。为了区别既在 R 的模式中又在 S 的模式中的属性 A, 我们对来自 R 的属性用 R. A 表示, 对来自 S 的属性用 S. A 表示。

例 4.5 为了简明扼要, 让我们使用一个解释乘积运算的抽象例子。假设关系 R 和 S 具有图 4.3 中给出的模式和元组。那么, 乘积 $R \times S$ 包括该图中给出的六个元组。注意, 我们如何将两个 R 元组中的每一个和三个 S 元组中的每一个组成对。因为 B 是两个模式中的属性, 我们已经在 $R \times S$ 的模式中使用了 R. B 和 S. B。其他属性不会混淆, 于是, 它们的 名字未加改变地出现在结果模式中。

关系 R		关系 S		
A	B	B	C	D
1	2	2	5	6
3	4	4	7	8
		9	10	11
结果 $R \times S$				
A	R. B	S. B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

图 4.3 两个关系及其笛卡尔积

4.1.5 自然连接

我们发现, 只把两个关系中在某个方面匹配的元组成对地连接起来比得到两个关系的乘积有更多的需求。匹配的最简单的类型是两个关系 R 和 S 的自然连接, 表示为 $R \bowtie S$, 只有在 R 和 S 的模式 的任何公共属性上一致的 R 和 S 的元组才会成对地出现在自然连接的结果中。更确切地说, 假设 A_1, A_2, \dots, A_n 是在 R 和 S 的模式中都有的属性, 当且仅当 R 的元组 r 和 S 的元组 s 在 A_1, A_2, \dots, A_n 每个属性上都一致时, R 的元组 r 和 S 的元组 s 才能成功地组成一对。

如果在连接 $R \bowtie S$ 中元组 r 和 s 成功地匹配成对, 那么, 成对的结果就是一个元组, 称为连接元组, 其中每个分量都对应于 R 和 S 的模式并集 中的一个属性。连接元组在 R 的模式 的每个属性上和元组 r 一致, 而在 S 的模式 的每

图 4.4 连接元组

个属性上和元组 s 一致。因为 r 和 s 成功地组成一对, 我们就知道 r 和 s 在 R 和 S 的模式公共属性上一致。因此, 连接元组在属于两个模式的公共属性上与 r 和 s 都一致是可能的。连接元组的建立可以通过图 4.4 表示出来。

另外还要注意, 这个连接运算和我们在 3.7.6 节中使用的连接运算是一样的, 在那里用连接运算重新组合已经投影到其属性的两个子集上的关系。其动机是解释 BCNF 分解为什么有意义。在 4.1.7 节, 我们将看到自然连接的另一个用途: 连接两个关系以使我们能够写出与每个关系的属性都有关的查询。

例 4.6 在图 4.3 中, 关系 R 和 S 的自然连接是

A	B	C	D
1	2	5	6
3	4	7	8

R 和 S 唯一的公共属性是 B 。因此, 要成功地匹配成对, 元组只需要在 B 分量上一致即可。如果这样, 结果元组有属性 A (来自 R)、 B (来自 R 或 S)、 C (来自 S) 和 D (来自 S) 对应的分量。

在这个例子中, R 的第一个元组只和 S 的第一个元组成功地匹配; 在它们的公共属性 B 上它们有公共值 2。这一对产生了第一个结果元组: (1, 2, 5, 6)。 R 的第二个元组只和 S 的第二个元组成功地匹配, 这一对产生的元组是 (3, 4, 7, 8)。注意, S 的第三个元组不能和 R 的任何元组匹配。因此对 $R \bowtie S$ 的结果没有影响。没能在连接中和另一个关系的任何元组匹配的元组有时称为悬挂元组。

例 4.7 前面的例子并不能说明为自然连接运算符所特有的所有可能性。例如: 没有一个元组成功地和多个元组匹配, 并且两个关系模式只有一个公共属性。在图 4.5, 我们看到另外两个关系 U 和 V , 在它们的模式中, 有两个公共属性 B 和 C 。我们还给出了一个元组和几个元组连接的实例。

A	B	C	B	C	D
1	2	3	2	3	4
6	7	8	2	3	5
9	7	8	7	8	10

关系 U

关系 V

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

结果 $U \bowtie V$

图 4.5 关系的自然连接

为了元组成功匹配, 它们必须在 B 和 C 的分量都一致。因此, U 的第一个元组成功地和 V 的前两个元组匹配, 而 U 的第二个和第三个元组成功地和 V 的第三个元组匹配。这四对的结果在图 4.5 中给出。

4.1.6 连接

自然连接要求我们使用一个特定的条件匹配元组。虽然公共属性相等这种方式是关系相连的最普通的基础, 但有时希望把两个关系的元组按其他某个基础匹配成对。为此, 我们有一个相关的表示法, 称为 连接, 指一个任意的条件。但在表达式中, 我们用 C 而不是 表示它。

R 和 S 基于条件 C 的 连接用 $R \bowtie_C S$ 表示。该运算的结果按如下步骤建立:

- 1. 获得 R 和 S 的积。
- 2. 从乘积中只选择满足条件 C 的元组。

因为使用了乘积运算, 结果模式是 R 和 S 模式的并集, 如果需要指出属性来自哪个模式, 则可在属性前面加上前缀“R.”或“S.”。

例 4.8 考虑运算 $U \bowtie_{A < D} V$, 其中, U 和 V 是来自图 4.5 的关系。我们必须考虑由每个关系的一个元组组成的所有 9 个元组对, 并要看 U 元组的 A 分量是否小于 V 元组的 D 分量。U 的第一个元组, A 分量为 1, 和 V 的每个元组成功地匹配成对。然而, U 的第二和第三个元组, A 分量分别为 6 和 9, 只和 V 最后的元组成功地匹配成对。于是, 结果只有 5 个元组, 由 5 个成功的匹配构造而成。该关系在图 4.6 中给出。

注意, 图 4.6 中的结果模式包括所有 6 个属性, 并用 U 和 V 作为前缀加在它们相应的属性 B 和 C 的前面以区别它们。于是, 连接与自然连接形成对照, 因为后者的公共属性合并成一个副本。当然, 在自然连接的情况下这样做是有意义的, 因为如果两个元组不在它们的公共属性上一致, 那它们就不能组成一对。在 连接的情况下, 不能保证在结果中相比较的属性一致, 因为此运算符可能不是“=”。

A	U.B	U.C	V.B	V.C	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

图 4.6 $U \bowtie_{A < D} V$ 的结果

例 4.9 这里是相同关系 U 和 V 的 连接, 具有更加复杂的条件:

$$U \bowtie_{A < D \text{ AND } U.B \neq V.B} V$$

也就是说, 对于成功的匹配, 不仅要求 U 元组的 A 分量小于 V 元组的 D 分量, 而且要求两个元组在它们相应的 B 分量上不一致。

等价表达式和查询优化

所有的数据库系统都有查询-应答系统, 其中许多都基于一种在表达能力上与关系代数类似的语言。因此, 用户提出的查询可能有许多等价表达式(当给定相同的关系作为运算数将产生同一个答案的表达式), 其中某些计算可能更快。1. 2. 3 节简要讨论的查询优化的一个重要工作, 就是用计算更有效的等价表达式取代关系代数的一个表达式。

元组

A	U. B	U. C	V. B	V. C	D
1	2	3	7	8	10

是满足两个条件的唯一元组, 因此该关系是上述 连接的结果。

4. 1. 7 查询中的复合运算

当查询时, 如果我们所做的一切就是在一个或两个关系上写单一的运算, 那么关系代数不会像它现在那样有用。然而, 关系代数像所有代数一样, 允许我们通过将运算符用于给定关系或者结果关系从而形成任意复杂的表达式。所谓结果关系就是将一个或多个运算符用于关系得到的结果。

可以通过将运算符用于子表达式, 必要时用括号表明运算项分组, 来构造关系代数表达式。把表达式表示为表达树也是可能的; 后者对我们来说读起来往往比较容易, 不过表达树作为机器可读的表示法不是太方便。

例 4. 10 让我们重新考虑例 3. 32 分解的电影关系。假定我们想知道“由 Fox 公司制作的至少有 100 分钟长的电影的名称和年份是什么?”, 要计算这个查询的答案, 一个途径是:

- (1) 选择长度 length ≥ 100 的 Movie 元组。
- (2) 选择 StudioName= 'Fox' 的 Movie 元组。
- (3) 计算(1)和(2)的交集。
- (4) 将来自(3)的关系投影到属性 title 和 year 上。

我们在图 4. 7 看到表示以上步骤的表达树。两个选择节点相当于步骤(1)和(2)。交集节点相当于步骤(3), 而投影节点是步骤(4)。

图 4. 7 关系代数表达式
对应的表达树

作为另一种方法, 我们可以用常规的、具有括号的线性表示法来表示相同的表达式。公式

$$title, year (\text{length} \geq 100 (Movie) \cap StudioName = 'Fox' (Movie))$$

代表相同的表达式。

顺便提一下, 经常有多个关系代数表达式代表相同的计算。例如, 用单一的选择运算

中的逻辑与(AND)代替交集,也可以写出上述查询。也就是说,

$$\text{title, year}(\text{length} \geq 100 \text{ AND StudioName} = \text{'Fox'}(\text{Movie}))$$

是等价的查询形式。

例 4.11 自然连接运算的一个用途是重组被分解的关系,分解的目的是使关系成为 BCNF。回忆一下例 3.32 中分解的关系:

Movie1 具有模式 {title, year, length, filmType, studioName}

Movie2 具有模式 {title, year, starName}

让我们写一个表达式来回答查询“找出主演的电影至少 100 分钟长的影星”。该查询将 Movie2 的 starName 和 Movie1 的 length 属性相连。我们可以通过连接这两个关系来连接这些属性。自然连接只把那些在 title 和 year 上一致的元组成功地匹配成对;也就是把指向同一部电影的元组匹配成对。因此, $\text{Movie1} \bowtie \text{Movie2}$ 是关系代数的一个表达式,它产生例 3.32 中我们称为 Movie 的关系。该关系是非 BCNF 关系,它的模式是所有 6 个属性,当一部电影有几个影星时它将包含涉及该电影的几个元组。

对于 Movie1 和 Movie2 的连接,我们必须使用一个选择,其条件是要求电影的长度至少是 100 分钟。然后,我们投影到所要求的属性 starName 上。表达式

$$\text{starName}(\text{length} \geq 100(\text{Movie1} \bowtie \text{Movie2}))$$

用关系代数表达了所要求的查询。

4.1.8 改名

为了调整由一个或多个关系代数运算构造的关系所用的属性名,使用显式的把关系改名的运算符往往是很方便的。我们将使用运算符 $s(A_1, A_2, \dots, A_n)(R)$ 把关系 R 改名。结果关系和 R 有完全相符的元组,但关系名是 S。此外,还把结果关系 S 的属性从左至右依次命名为 A_1, A_2, \dots, A_n 。如果我们只想把关系改名为 S,而让属性和 R 中的一样,那我们可以只是用 $s(R)$ 。

例 4.12 在例 4.5 中,我们产生了图 4.3 的两个关系 R 和 S 的乘积,并按惯例为属性改名,若一个属性出现在两个运算数中,就用关系名作为它的前缀而使之改名。这两个关系 R 和 S 在图 4.8 中重复给出。

然而,假设我们并不希望称 B 的两种描述为名字 R.B 和 S.B;而想继续对来自 R 的属性使用名字 B,但希望使用 X 作为来自 S 的属性 B 的名字。我们可以把 S 的属性改名,以便第一个属性称为 X。表达式 $s(X, C, D)(S)$ 的结果是名为 S 的关系,看起来和图 4.3 中的关系 S 几乎一样,但是其第一列的属性为 X,取代了原来的 B。

当我们求 R 和这个新关系的乘积时,属性之间没有名字的冲突,因此不必进一步改名。也就是说,除了五列从左到右依次标记为 A, B, X, C, D 之外,表达式 $R \bowtie s(X, C, D)(S)$ 的结果就是图 4.3 的关系 $R \bowtie S$ 。这个关系在图 4.8 中给出。

作为另一种方法,我们求乘积时可以不改名,如在例 4.5 所做的那样,然后把结果改名。表达式 $RS(A, B, X, C, D)(R \bowtie S)$ 将产生和图 4.8 中相同的关系,具有相同的属性集,但是关

记住,例 3.32 中的关系 Movie 与 3.9 节介绍的并用于例 4.2、4.3 和 4.4 的关系 Movie,其关系模式稍有不同。

系名为 RS, 而图 4. 8 中的结果关系并非如此。

关系 R		关系 S		
A	B	B	C	D
1	2	2	5	6
3	4	4	7	8
		9	10	11

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

结果 $R \bowtie_{S(X,C,D)}(S)$

图 4. 8 两个关系及其乘积

4. 1. 9 基本和导出运算

在 4. 1 节我们已经讨论过的某些运算可以用其他关系代数运算的形式表达。例如，交集可以用集合差的形式表达

$$R \cap S = R - (R - S)$$

也就是说，如果 R 和 S 是具有相同模式的任何两个关系，要计算 R 和 S 的交集，可以首先从 R 中减去 S，形成的关系 T 包括所有在 R 中而不在 S 中的元组。然后从 R 中减去 T，就只剩下既在 R 中也在 S 中的元组。

连接的两种形式也可以用其他运算的形式表达。 连接可以通过乘积和选择表达。

$$R \Join S = \sigma_{\theta}(R \bowtie S)$$

R 和 S 的自然连接可以首先用乘积 $R \bowtie S$ 表达。然后应用选择运算符并带有如下形式的条件 C 进行选择

$$R \Join S = \sigma_{A_1 = S.A_1 \wedge A_2 = S.A_2 \wedge \dots \wedge A_n = S.A_n}(R \bowtie S)$$

其中， A_1, A_2, \dots, A_n 是出现在 R 和 S 的两个模式中的所有属性。最后，我们必须对每个等值属性的一个副本进行投影。假设 L 是一个属性表，其中前面的属性在 R 的模式中，后面的属性在 S 的模式中而不在 R 的模式中。于是

$$R \Join S = \pi_L(\sigma_{\theta}(R \bowtie S))$$

例 4. 13 图 4. 5 的关系 U 和 V 的自然连接可以用乘积、选择和投影的形式写出：

$$\pi_{A, U.B, U.C, D}(\sigma_{U.B = V.B \wedge U.C = V.C}(U \bowtie V))$$

也就是说，我们得到乘积 $U \bowtie V$ ，然后在每对同名属性(在这个例子中是 B 和 C)之间进行等值选择。最后，我们投影到所有的属性上，但只保留其中一个 B 和一个 C；我们的选择是

取消 V 的属性, 如果该属性的名字也在 U 的模式中出现的话。

对于另一个例子, 例 4. 9 的 连接可以写作

$$A < D \text{ AND } V.B = V.B(U \times V)$$

也就是说, 我们求关系 U 和 V 的乘积, 然后应用出现在 连接中的条件。

在本节提到的冗余只是我们已经介绍的运算之间的“冗余”。剩下 6 个运算——并集、差集、选择、投影、乘积和改名——构成独立集, 其中每个都不能由其他 5 种形式导出。

4. 1. 10 本节练习

练习 4. 1. 1: 在这个练习中, 我们介绍关系数据库模式的一个不断滚动的实例和某些采样数据。数据库模式包括四个关系, 它们的模式是:

- Product (maker, model, type)
- PC(model, speed, ram, hd, cd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)

Product 关系给出不同产品的制造商、型号和类型(PC、便携式电脑或打印机)。为了方便, 我们假定型号对于所有制造商和产品类型是唯一的; 这个假设并不现实, 实际的数据库将把制造商代码作为型号的一部分。PC 关系对于每个 PC 型号给出速度(处理器的速度, 以兆赫计算)、RAM 的容量(以兆字节计算)、硬盘容量(以 G 字节计算)、光盘驱动器的速度(例如, 4 倍速)和价格。便携式电脑(Laptop)关系和 PC 是类似的, 除了屏幕尺寸(用英寸计算)记录在原来记录 CD 速度的地方。打印机(Printer)关系对于每台打印机的类型记录打印机是否产生彩色输出(真, 如果是的话)、工艺类型(激光、喷墨或干式)和价格。

关系 Product 的某些采样数据在图 4. 9 中给出。其他三个关系的采样数据在图 4. 10 中给出。制造商和型号均已“妥善处理”, 但是数据取自 1996 年底出售的典型产品。

maker(制造商)	model(型号)	type(类型)	maker(制造商)	model(型号)	type(类型)
A	1001	个人电脑(PC)	D	2001	便携式电脑
A	1002	个人电脑	D	2002	便携式电脑
A	1003	个人电脑	D	2003	便携式电脑
B	1004	个人电脑	D	3001	打印机
B	1006	个人电脑	D	3003	打印机
B	3002	打印机	E	2004	便携式电脑
B	3004	打印机	E	2008	便携式电脑
C	1005	个人电脑	F	2005	便携式电脑
C	1007	个人电脑	G	2006	便携式电脑
D	1008	个人电脑	G	2007	便携式电脑
D	1009	个人电脑	H	3005	打印机
D	1010	个人电脑	I	3006	打印机

图 4. 9 产品的采样数据

写出关系代数的表达式, 回答下列查询。对于图 4. 9 和 4. 10 中的数据, 给出你的查询结果。然而, 你的答案应该适用于任意数据, 而不仅是这些图中的数据。

model(型号)	speed(速度)	ram(内存)	hd(硬盘)	cd(光驱)	price(价格)
1001	133	16	1. 6	6X	1595
1002	120	16	1. 6	6X	1399
1003	166	24	2. 5	6X	1899
1004	166	32	2. 5	8X	1999
1005	166	16	2. 0	8X	1999
1006	200	32	3. 1	8X	2099
1007	200	32	3. 2	8X	2349
1008	180	32	2. 0	8X	3699
1009	200	32	2. 5	8X	2599
1010	160	16	1. 2	8X	1495

(a) 关系 PC(个人电脑) 的采样数据

model(型号)	speed(速度)	ram(内存)	hd(硬盘)	screen(屏幕)	price(价格)
2001	100	20	1. 10	9. 5	1999
2002	117	12	0. 75	11. 3	2499
2003	117	32	1. 00	11. 2	3599
2004	133	16	1. 10	11. 2	3499
2005	133	16	1. 00	11. 3	2599
2006	120	8	0. 81	12. 1	1999
2007	150	16	1. 35	12. 1	4799
2008	120	16	1. 10	12. 1	2099

(b) 关系 Laptop(便携式电脑) 的采样数据

model(型号)	color(彩色)	type(类型)	price(价格)
3001	真	喷墨	275
3002	真	喷墨	269
3003	假	激光	829
3004	假	激光	879
3005	假	喷墨	180
3006	真	干式	470

(c) 关系 Printer(打印机) 的采样数据

图 4. 10 练习 4. 1. 1 中各关系的采样数据

提示: 对于比较复杂的表达式, 利用给定的关系(Product 等) 定义一个或更多中间关系, 然后在最终表达式中使用这些关系, 也许是有意义的。此后, 可以替换最终表达式中的中间关系, 以得到用给定关系表示的表达式。

* (a) 什么型号的 PC 速度至少为 150?

- (b) 哪个厂商生产的便携式电脑具有最少 1G 字节的硬盘。
- (c) 找出厂商 B 生产的所有产品(任一类型)的型号和价格。
- (d) 找出所有彩色激光打印机的型号。
- (e) 找出销售便携式电脑但不销售 PC 的厂商。
- * ! (f) 找出在两个或两个以上 PC 中出现的硬盘容量。
- ! (g) 找出速度相同且 RAM 相同的成对的 PC 型号。一对型号只列出一次。例如, 列出((i, j) 就不再列出(j, i)。
- !! (h) 找出至少生产两种不同的计算机(PC 或便携式电脑)且机器速度至少为 133 的厂商。
- !! (i) 找出生产最高速度的计算机(PC 或便携式电脑)的厂商。
- !! (j) 找出至少生产三种不同速度 PC 的厂商。
- !! (k) 找出只卖三种不同型号 PC 的厂商。

练习 4.1.2: 把练习 4.1.1 列出的每个表达式画成表达树。

练习 4.1.3: 这个练习介绍另一个不断滚动的实例, 是有关第二次世界大战中的主力舰的。它用到以下关系:

Classes(class, type, country, numGuns, bore, displacement)

Ships(name, class, launched)

Battles(name, date)

Outcomes(ship, battle, result)

舰艇都是按具有相同设计的“等级”制造的, 而等级总是以该等级的第一艘舰艇命名。关系 Classes 记录 class(等级名)、type(类型: bb 代表战列舰, 或 bc 代表巡洋舰)、country(制造舰艇的国家)、numGuns(主要火炮的数量)、bore(主要火炮的口径: 炮管的直径, 以英寸计算)和 displacement(排水量: 重量, 以吨计算)。关系 Ships 记录 name(舰艇名)、class(舰艇等级名)和 launched(舰艇下水的年份)。关系 Battles 给出涉及这些舰艇的战役的 name(名字)和 date(日期), 关系 Outcomes 给出每艘舰艇在每次战役中的 result(结果: 沉没、损坏或完好)。

图 4.11 和 4.12 给出这四个关系的某些采样数据。注意, 和练习 4.1.1 中的数据不同, 在这些数据中有某些“悬挂的元组”, 例如, 在 Outcome 中提到而在 Ships 中没有提到的舰艇。

写出关系代数的表达式回答下列查询。对于图 4.11 和图 4.12 的数据, 给出你的查询结果。然而, 你的答案应该适用于任意的数据, 而不仅是这些图中的数据。

- (a) 对于配备至少 16 英寸口径火炮的等级, 给出等级名和国家。
- (b) 找出 1921 年以前下水的舰艇。
- (c) 找出在北大西洋战役中沉没的舰艇。
- (d) 1921 年的华盛顿条约禁止主力舰的重量在 35 000 吨以上。列出违背华盛顿条约的舰艇。

来源: J. N. Westwood, Fighting Ships of World War II, Follett Publishing, Chicago, 1975 and R. C. Stern, US Battleships in Action, Squadron/Signal Publications, Carrollton, TX, 1980.

class	type	country	numGuns	bore	displacement
Bismarck	bb	德国	8	15	42000
Iowa	bb	美国	9	16	46000
Kongo	bc	日本	8	14	32000
North Carolina	bb	美国	9	16	37000
Renown	bc	大不列颠	6	15	32000
Revenge	bb	大不列颠	8	15	29000
Tennessee	bb	美国	12	14	32000
Yamato	bb	日本	9	18	65000

(a) 关系 Classes 的采样数据

name	data
North Atlantic	5/ 24-27/ 41
Guadalcanal	11/ 15/ 42
North Cape	12/ 26/ 43
Surigao Strait	10/ 25/ 44

(b) 关系 Battles 的采样数据

ship(舰艇)	battle(战役)	result(结果)
Bismarck	North Atlantic	沉没
California	Surigao Strait	完好
Duke of York	North Cape	完好
Fuso	Surigao Strait	沉没
Hood	North Atlantic	沉没
King George V	North Atlantic	完好
Kirishima	Guadalcanal	沉没
Prince of Wales	North Atlantic	毁坏
Rodney	North Atlantic	完好
Scharnhorst	North Cape	沉没
South Dakota	Guadalcanal	毁坏
Tennessee	Surigao Strait	完好
Washington	Guadalcanal	完好
West Virginia	Surigao Strait	完好
Yamashiro	Surigao Strait	沉没

(c) 关系 Outcomes 的采样数据

图 4.11 练习 4.1.3 的数据

- (e) 列出参与瓜达尔卡纳尔岛战役的舰艇的名字、排水量以及火炮的数量。
- (f) 列出数据库中提到的所有主力舰(记住,可能不是所有的舰艇都出现在 Ships 关系中)。

name	class	launched
California	Tennessee	1921
Haruna	Kongo	1915
Hiei	Kongo	1914
Iowa	Iowa	1943
Kirishima	Kongo	1915
Kongo	Kongo	1913
Missouri	Iowa	1944
Musashi	Yamato	1942
New Jersey	Iowa	1943
North Carolina	North Carolina	1941
Ramillies	Revenge	1917
Renown	Renown	1916
Repulse	Renown	1916
Resolution	Revenge	1916
Revenge	Revenge	1916
Royal Oak	Revenge	1916
Royal Sovereign	Revenge	1916
Tennessee	Tennessee	1920
Washington	North Carolina	1941
Wisconsin	Iowa	1944
Yamato	Yamato	1941

图 4. 12 关系 Ships 的采样数据

- ! (g) 找出那些其成员只有一个的等级。
- ! (h) 找出既有战列舰又有巡洋舰的国家。
- ! (i) 找出那些“ 来日再战斗 ”的舰艇。它们在一次战役中受损, 但以后又投入另一次战役。

练习 4. 1. 4: 将你在练习 4. 1. 3 中写的每个表达式画成表达树。

* 练习 4. 1. 5: 自然连接 $R \bowtie S$ 和 $\bowtie_c R \bowtie S$ (对于出现在 R 和 S 的模式中的每个属性 A , 条件 C 是 $R.A = S.A$) 有什么不同?

! 练习 4. 1. 6: 如果把一个元组加到某个关系运算符的一个自变量中, 得到的结果将包括在增加该元组前它所包含的所有元组, 还可能加上更多的元组, 则认为该关系运算符是单调的。在这一节描述的什么运算符是单调的? 对于每个运算符, 或者说明它为什么是单调的, 或者给出一个表明不是的例子。

! 练习 4. 1. 7: 假定关系 R 和 S 分别有 n 个元组和 m 个元组, 给出下列表达式的结果中可能的最小和最大元组数

- * (a) $R \bowtie S$
- (b) $R \bowtie_c S$
- (c) $\bowtie_c(R) \bowtie S$, 其中 C 为某个条件。

(d) $\pi_L(R) \bowtie S$, 其中 L 为某个属性表。

！练习 4.1.8: 关系 R 和 S 的半连接记为 $R \ltimes S$ 。它是在 R 和 S 的模式的所有公共属性上, 至少和一个 S 元组一致的 R 元组的集合。给出三个不同的与 $R \ltimes S$ 等价的关系代数表达式。

！！练习 4.1.9: 假设 R 是模式为 $(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ 的关系, 并且假设 S 是模式为 (B_1, B_2, \dots, B_m) 的关系。也就是说, S 的属性是 R 的属性的子集。 R 和 S 的商, 表示为 $R \div S$, 是属性 A_1, A_2, \dots, A_n (即 R 属性中不是 S 属性的部分) 上满足如下条件的元组 t 的集合, 条件为对于 S 中的每个元组 s , 由 t 对应于 A_1, A_2, \dots, A_n 的分量和 s 对应于 B_1, B_2, \dots, B_m 的分量组成的元组 ts , 都是 R 的成员。用本节前面定义的运算符写出与 $R \div S$ 等价的关系代数表达式。

4.2 关系的逻辑

另一种表达关系查询的方法基于逻辑而非代数。有趣的是, 两种方法(逻辑和代数)可以表达相同类型的查询。这一节介绍的逻辑查询语言称为 Datalog(“数据逻辑”), 它是由 if-then 规则组成的。我们用这些规则之一表达这样的想法: 从某些关系中的某些元组组合出发, 可以推测出某个其他元组在某个其他关系中, 或在查询的答案中。

4.2.1 谓词和原子

关系在 Datalog 中通过称为谓词的符号表示。每个谓词都有固定数量的参数, 而谓词及其随后的参数称为原子。原子的语法很像传统编程语言中的函数调用。例如 $P(x_1, x_2, \dots, x_n)$ 是由谓词 P 和参数 x_1, x_2, \dots, x_n 组成的原子。

本质上, 谓词是返回一个布尔值的函数名。如果 R 是以某种固定顺序排列的 n 个属性的关系, 那么, 我们也应当用 R 作为相应于这个关系的谓词的名字。如果 (a_1, a_2, \dots, a_n) 是 R 的一个元组, 则原子 $R(a_1, a_2, \dots, a_n)$ 的值为 TRUE; 否则原子的值为 FALSE。

例 4.14 假设 R 是图 4.3 中的关系

A	B
1	2
3	4

那么 $R(1, 2)$ 为真, 并且 $R(3, 4)$ 也为真。然而, 对于任何其他的 x, y 值, $R(x, y)$ 为假。谓词可以用变量以及常量作为参数。如果一个原子的一个或多个参数为变量, 那么它就是一个布尔值函数, 该函数随变量取值不同而取真或假。

例 4.15 如果 R 是例 4.14 中的谓词, 那么对于任意的 x 和 y , $R(x, y)$ 是指出元组 (x, y) 是否属于关系 R 的函数。对于例 4.14 中 R 的具体实例, 当

1. $X = 1$ 且 $y = 2$, 或者
2. $X = 3$ 且 $y = 4$

时, $R(x, y)$ 返回真, 否则返回假。作为另一个例子, 当 $z = 2$ 时, 原子 $R(1, z)$ 返回真, 否则返回假。

4.2.2 算术原子

Datalog 中, 另一种重要的原子是算术原子。这种原子是两个算术表达式的比较, 比如 $x < y$ 或 $x + 1 \leq y + 4 * z$ 。与此相对, 我们将把 4.2.1 节介绍的原子称为关系原子; 这两者都是“原子”。

注意, 算术原子和关系原子都以任意变量的值为参数, 并返回布尔值。在效果上, 算术比较(比如 $<$ 或 \leq) 就像某个关系名, 其关系包括比较结果为真的所有的“对”。于是, 我们可以把关系“ $<$ ”具体化, 成为包括 $(1, 2)$ 或 $(-1.5, 65.4)$ 在内的所有元组, 其第一个分量小于第二个分量。然而, 数据库的关系总是有限的, 并且通常随时间变化。而算术比较关系(例如 $<$) 则是无限的, 并且不变。

4.2.3 Datalog 规则和查询

类似于关系代数中的那些运算在 Datalog 中通过规则描述。规则包括:

1. 一个称为头部(head)的关系原子, 随后是
2. 符号 `:-`, 通常读作“if”, 随后是
3. 由一个或多个原子组成的体(body), 其原子称为子目标(subgoal), 它可以是关系的, 也可以是算术的。各子目标用 AND 连接, 并且, 子目标前面可以有逻辑运算符 NOT, 也可以没有。

例 4.16 Datalog 规则

$\text{LongMovie}(t, y) \text{ :- Movie}(t, y, l, c, s, p) \text{ AND } l \geq 100$

可以用来计算至少 100 分钟长的“长电影”。它引用我们在 3.9 节定义的标准关系 Movie, 其模式为

$\text{Movie}(\text{title}, \text{year}, \text{length}, \text{inColor}, \text{studioName}, \text{producerC\#})$

规则的头部是原子 $\text{LongMovie}(t, y)$ 。规则体包括两个子目标:

1. 第一个子目标有谓词 Movie 和 6 个参数, 对应于 Movie 关系的 6 个属性。这些参数中的每一个都有不同的值: t 对应于 title 分量, y 对应于 year 分量, l 对应于 length 分量, 等等。我们可以将这个子目标看作是在表明:“假设 (t, y, l, c, s, p) 是关系 Movie 的当前实例中的元组。”更确切地说, 当 6 个变量具有某一个 Movie 元组的 6 个分量的值时, $\text{Movie}(t, y, l, c, s, p)$ 为真。
2. 当 Movie 元组的长度分量至少为 100 时第二个子目标 $l \geq 100$ 为真。

规则作为一个整体是在表明: 当可以在 Movie 中找到符合下列条件的元组时, $\text{longMovie}(t, y)$ 为真:

- (a) t 和 y 为前两个分量(对应于 title 和 year),
- (b) 第三个分量 l (对应于 length) 至少为 100, 并且
- (c) 分量 4 至 6 为任意值。

因此, 这个规则等价于关系代数中的“赋值语句”。

匿名变量

通常, Datalog 规则中某些变量只出现一次。因此, 这些变量的名字是无关紧要的; 仅当一个变量多次出现时, 我们才关注它的名字以便将其视为第二次以及后来出现的同一个变量。于是, 我们按常规以下划线“_”作为原子的参数, 代表只出现在那里的变量。多条下划线“_”代表不同的变量, 决不是相同的变量。例如, 例 4. 16 的规则可以写为

longMovie(t, y) Movie(t, y, l, _ , _ , _) AND l ≥ 100

每个只出现一次的变量 c, s 和 p 都用下划线替代。我们不能替代其他变量, 因为它们在规则中都出现了两次。

longMovie = title, year(length ≥ 100(Movie))

它的右边是关系代数表达式。

Datalog 中的查询是一个或多个规则的聚集。如果规则头部只有一个关系出现, 那么, 就认为该关系的值是查询的答案。因此, 在例 4. 16 中, longMovie 就是查询的答案。如果规则头部有多个关系, 那么这些关系中的一个为查询的答案, 而其他关系在答案定义中起辅助作用。我们必须指定哪个关系是查询要求的答案, 可以通过给出一个名字(例如 Answer)来指明这个关系。

4. 2. 4 Datalog 规则的含义

例 4. 16 给出一个 Datalog 规则含义的提示。更确切地说, 想象一下在所有可能值范围内变化的规则变量。当这些变量都具有使所有子目标为真的值时, 那么, 我们对于这些变量检查头部的值是什么, 并且将结果元组加入谓词在头部的关系中。

例如, 我们可以想象例 4. 16 的在所有可能值范围内变化的 6 个变量。只有(t, y, l, c, s, p)的值按顺序构成一个 Movie 元组, 才是使所有子目标为真的值的组合。并且, 因为 l ≥ 100子目标必须也为真, 所以这个元组必须是长度分量的值 l 至少为 100 的元组。当我们找到这样的值的组合时, 就将元组(t, y)放入头部的关系 LongMovie 中。

然而, 我们必须对规则中变量的使用方法加以限制, 以使规则的结果是有限关系, 并且使得带算术子目标或求反子目标(前面有 NOT 的子目标)的规则具有直观意义。这个条件称为安全条件, 即:

- 出现在规则中任何地方的变量必须出现在某个非求反的关系子目标中。

特别要指出的是, 在头部、求反关系子目标或任何算术子目标中出现的变量, 也必须出现在非求反的关系子目标中。

例 4. 17 考虑例 4. 16 的规则

LongMovie(t, y) Movie(t, y, l, _ , _ , _) AND l ≥ 100

第一个子目标是非求反的关系子目标, 并且它包括出现在规则中的所有变量。特别要注意的是, 出现在头部中的两个变量 t 和 y 也出现在体的第一个子目标中。变量 l 不但出

现在算术子目标中,而且也出现在第一子目标中。

例 4. 18 以下规则有三处违反了安全条件:

$$P(x, y) \quad Q(x, z) \text{ AND NOT } R(w, x, z) \text{ AND } x < y$$

- 1. 变量 y 出现在头部,但是没有出现在任何非求反的关系子目标中。注意, y 出现在算术子目标 $x < y$ 中的事实无助于将 y 的可能值限制在有限集中。只要我们找到分别对应于 w, x 和 z 的值 a, b 和 c 满足前两个子目标,数量无限的元组(a, d)(其中 $d > a$)就会对头部的关系 P“纠缠不休”。
- 2. 变量 w 出现在求反的关系子目标中而不在非求反的关系子目标中。
- 3. 变量 y 出现在算术子目标中,但不在非求反的关系子目标中。

因此,它不是安全的规则,不能用于 Datalog。

还有另一种方法定义规则。不考虑对变量所有可能的赋值,而考虑和每个非求反的关系子目标相对应的关系中元组的集合。如果对于每个非求反的关系子目标的元组的某个赋值在这样的意义上是一致的,即对于一个变量的每个出现赋予同一个值,那么,就考虑把结果的值赋予规则的所有变量。注意,因为规则是安全的,所以对每个变量赋予一个值。

对于每个一致的赋值,我们考虑求反的关系子目标和算术子目标,以便检查对变量的赋值是否使它们都为真。记住,如果求反子目标的原子为假,则求反子目标为真。如果所有的子目标都为真,那么,我们将检查对变量的这种赋值使头部变成了什么元组。这个元组将加入其谓词是头部的关系中。

例 4. 19 考虑 Datalog 规则:

$$P(x, y) \quad Q(x, z) \text{ AND } R(z, y) \text{ AND NOT } Q(x, y)$$

假设关系 Q 包括两个元组(1, 2)和(1, 3)。假设关系 R 包括元组(2, 3)和(3, 1)。这里有两个非求反的关系子目标, $Q(x, z)$ 和 $R(z, y)$, 所以,我们必须考虑分别来自关系 Q 和 R 的元组对这些子目标赋值的所有组合。图 4. 13 考虑了所有的四种组合。

	$Q(x, z)$ 的元组	$R(z, y)$ 的元组	赋值一致?	NOT $Q(x, y)$ 为真?	头部的结果
(1)	(1, 2)	(2, 3)	Yes	No	-
(2)	(1, 2)	(3, 1)	No; $z = 2, 3$	不相关	-
(3)	(1, 3)	(2, 3)	No; $z = 3, 2$	不相关	-
(4)	(1, 3)	(3, 1)	Yes	Yes	$P(1, 1)$

图 4. 13 元组对 $Q(x, z)$ 和 $R(z, y)$ 的所有可能的赋值

图 4. 13 中第二和第三种选择方案不一致。每种都把两个不同的值给变量 Z。这样,我们就不能进一步考虑这些元组赋值。

第一种选择方案,对子目标 $Q(x, z)$ 赋予元组(1, 2)并对子目标 $R(z, y)$ 赋予元组(2, 3),产生了一致的赋值, x, y 和 z 分别赋值 1, 3 和 2。于是,我们进入对其他子目标的测试,即对不是非求反的关系子目标的测试。只有一个:NOT $Q(x, y)$ 。对于变量的这种赋值,这个子目标变成 NOT $Q(1, 3)$ 。因为(1, 3)是 Q 的一个元组,这个子目标为假,对于元组赋值选择方案(1)没有产生头部元组。

最终的选择方案为(4)。这里,赋值是一致的; x, y 和 z 分别赋值 1, 1 和 3。子目标 $\text{NOT } Q(x, y)$ 取值为 $\text{NOT } Q(1, 1)$ 。因为元组(1, 1)不是 Q 的元组, 故该子目标为真。于是, 我们按变量的这种赋值计算头部 $P(x, y)$, 得到 $P(1, 1)$ 。元组(1, 1)在关系 P 中。我们已经详细讨论了所有元组赋值方案, 因此这是 P 中唯一的元组。

4.2.5 外延和内涵谓词

对以下两者进行区别是有益的:

- 外延谓词, 其关系存储在数据库中。
- 内涵谓词, 其关系通过应用一个或多个 Datalog 规则计算而求得。

两者的差别和关系代数表达式的运算项(它们是外延的, 也就是, 由它们的外延定义, 它是关系的当前实例的另一个名字)与用关系代数表达式计算的关系(或者作为最终结果, 或者作为对应于某个子表达式的中间结果; 这些关系是“内涵的”, 即由编程者的“意图”定义)之间的差别相同。

说到 Datalog 规则时, 如果谓词分别是“内涵”或“外延”的, 那么, 我们将引用与“内涵”或“外延”谓词相对应的关系。我们也将用“内涵数据库”的缩写 IDB 来引用内涵谓词或相应关系。与此类似, 我们把“外延数据库”的缩写 EDB 用于外延谓词或相应关系。

这样, 在例 4.16 中, Movie 是一个 EDB 关系, 通过它的外延定义。谓词 Movie 同样是一个 EDB 谓词。关系和谓词 LongMovie 都是内涵的。

EDB 谓词永远不能出现在规则的头部, 不过, 它可以出现在规则体中。IDB 谓词可以出现在规则的头部或者体中, 或者两者兼有。通过使用在头部具有相同谓词的几个规则, 构造单个的关系也是常见的。我们将在涉及两个关系并集的例 4.21 看到对这个概念的解释。

我们可以用一系列的内涵谓词逐渐建立 EDB 关系的更复杂的函数。这个过程类似于用几个运算符建立关系代数表达式。我们在下一节还将看到使用几个内涵谓词的例子。

4.2.6 本节练习

练习 4.2.1: 用 Datalog 写出练习 4.1.1 的每个查询。应当只使用安全规则, 但是你可能希望使用与复杂的关系代数表达式的子表达式相对应的几个 IDB 谓词。

练习 4.2.2: 用 Datalog 写出练习 4.1.3 的每个查询。仍然只使用安全规则, 但是如果愿意, 也可以使用几个 IDB 谓词。

!! 练习 4.2.3: 如果关系子目标的谓词具有有限关系, 那么, 给予 Datalog 规则安全的要求足以保证头部谓词有一个有限关系。然而, 这个要求太严了。给出违背这个条件的 Datalog 的一个例子, 而不管我们赋予关系谓词的有限关系是什么, 头部关系将是有限的。

4.3 从关系代数到 Datalog

每个关系代数运算符都可以用一个或几个 Datalog 规则来模拟。在这一节, 我们将依次考虑每个运算符, 然后, 考虑如何对 Datalog 规则进行组合以模拟复杂的代数表达式。

规则中的变量是局部的

注意,我们为规则中的变量所选择的字是任意的,并且和其他任何规则中使用的变量没有联系。没有联系的原因是每个规则单独计算,并且为与其他规则无关的头部关系提供元组。例如,我们可以用

$$U(w, x, , z) \quad S(w, x, y, z)$$

替代例 4.21 中的第二个规则而第一个规则保持不变,这两个规则仍将算出 R 和 S 的并集。然而要注意,当用变量 a 替代写在规则中的另一个变量 b 时,我们必须用 a 替代 b 在规则中的所有出现,并且,我们选择的替换变量 a 一定不是规则中已有的变量。

4.3.1 交集

两个关系的交集可用一个规则来表示,该规则具有与两个关系对应的子目标,并对相应的参数使用相同的变量。

例 4.20 以图 4.1 中的关系 R 和 S 为例。回忆一下这些关系,每个关系的模式都具有四个属性: name, address, gender 和 birthdate。这样,它们的交集用 Datalog 规则计算

$$I(n, a, g, b) \quad R(n, a, g, b) \text{ AND } S(n, a, g, b)$$

其中, I 是一个 IDB 谓词,当我们应用这个规则时,它的关系变成 R S。也就是说,对于某个元组(n, a, g, b),为了使两个子目标都为真,该元组必须同时在 R 和 S 中。

4.3.2 并集

两个关系的并集要用两个规则来构造。每个规则都有一个与其中一个关系相对应的原子作为它唯一的子目标,并且,两个规则的头部都有相同的在头部的 IDB 谓词。每个头部的参数都和它的规则子目标中的完全相同。

例 4.21 为了得到例 4.20 中关系 R 和 S 的并集,我们使用两个规则

$$\begin{aligned} (1) \quad & U(n, a, g, b) \quad R(n, a, g, b) \\ (2) \quad & U(n, a, g, b) \quad S(n, a, g, b) \end{aligned}$$

规则(1)表明,R 中的每个元组都是 IDB 关系 U 中的一个元组。类似地,规则(2)表明,S 中的每个元组都在 U 中。这样,两个规则合在一起意味着 R S 中的每个元组都在 U 中。如果我们在头部没有写 U 的更多的规则,那么,任何其他元组都没有进入关系 U 的途径,在这种情况下,我们可以得出结论, U 正好是 R S。回忆一下,因为关系是集合,所以,一个元组即使同时出现在 R 和 S 中,在关系 U 中也只出现一次。

4.3.3 差集

关系 R 和 S 的差集用具有求反子目标的单一规则计算。也就是说,非求反子目标是

实际上,我们应假设,在本节的每个例子中,除了显式地给出的规则之外,没有其他 IDB 谓词规则。如果有其他规则,那么就不能把其他元组排斥在该谓词的关系之外。

谓词 R, 求反子目标是谓词 S。这些子目标和头部对于相应的参数都有相同的变量。

例 4.22 如果 R 和 S 是例 4.20 中的关系, 那么规则

$$D(n, a, g, b) \leftarrow R(n, a, g, b) \text{ AND NOT } S(n, a, g, b)$$

将 D 定义为关系 R—S。

4.3.4 投影

为了计算关系 R 的投影, 我们使用一个具有谓词 R 的单一子目标的规则。该子目标的参数是不同的变量, 每个变量对应于关系的一个属性。头部有带参数的原子, 这些参数是按要求的顺序与投影的属性表对应的变量。

例 4.23 假定我们想把关系

$$\text{Movie}(\text{title}, \text{year}, \text{length}, \text{inColor}, \text{studioName}, \text{produceC\#})$$

投影到它的前三个属性——title, year 和 length 上, 正如例 4.2 那样, 规则

$$P(t, y, l) \leftarrow \text{Movie}(t, y, l, c, s, p)$$

的作用就是定义一个称为 P 的关系作为投影的结果。

4.3.5 选择

在 Datalog 中要表达选择比较困难。简单的情况是选择条件为一个算术比较或多个算术比较的“与”(AND)。在这种情况下, 我们建立一个具有如下子目标的规则:

- 1. 一个关系子目标, 对应于将对其进行选择的关系。该原子对于每个分量有不同的变量, 而每个分量都对应于关系的一个属性。
- 2. 算术子目标, 每个算术子目标对应于选择条件中的一个比较, 且与这个比较是等同的。虽然在选择条件中使用属性名, 但在算术子目标中却使用相应的变量, 并与关系子目标中建立的变量保持一致。

例 4.24 例 4.4 的选择

$$\text{length} \geq 100 \text{ AND studioName} = \text{'Fox'}(\text{Movie})$$

可以写成如下的 Datalog 规则

$$S(t, y, l, c, s, p) \leftarrow \text{Movie}(t, y, l, c, s, p) \text{ AND } l \geq 100 \text{ AND } s = \text{'Fox'}$$

结果是关系 S。注意, l 和 s 是对应于属性 length 和 studioName 的变量, 而这些属性则按 Movie 中属性的标准顺序排列。

现在, 让我们来考虑涉及条件的 OR(或)的选择。我们不必用单一 Datalog 规则替换该选择。但是, 对两个条件 OR 或的选择等价于对每个条件单独的选择, 然后取结果的并集。这样, n 个条件的 OR 或可用 n 条规则表示, 其中每条规则都定义相同的头部谓词。第 i 条规则对 n 个条件中的第 i 个进行选择。

例 4.25 让我们用 OR 替换 AND 来修改例 4.24 的选择, 得到以下结果

$$\text{length} \geq 100 \text{ OR studioName} = \text{'Fox'}(\text{Movie})$$

也就是, 找出或者长度至少 100 或者由 Fox 公司制作的所有电影。我们可以写两个规则, 每个都对应于下面两个条件中的一个:

$$1. S(t, y, l, c, s, p) \leftarrow \text{Movie}(t, y, l, c, s, p) \text{ AND } l \geq 100$$

$$2. S(t, y, l, c, s, p) \quad \text{Movie}(t, y, l, c, s, p) \text{ AND } s = 'Fox'$$

规则 1 将找出至少 100 分钟长的电影,而规则 2 将找出由 Fox 制作的电影。

即使更复杂的选择条件也可以通过以任何顺序多次使用逻辑运算符 AND, OR 和 NOT 而形成。然而,有一个众所周知的技术(我们将不在这里介绍它)可以把任何这样的逻辑表达式重新组成“析取范式”,即表达式是“合取式”的 OR。合取式是直接项的 AND,而直接项则是比较或者求反的比较。

可以用子目标表示任何直接项,在它前面可能有 NOT。如果是算术子目标,NOT 就可以并入比较运算符。例如,NOT $x \geq 100$ 可以写成 $x < 100$ 。于是,任何合取式都可以用单一 Datalog 规则表示,而每个子目标对应一个比较。最后,每个析取范式表达式都可以用几个 Datalog 规则写出,而每个规则对应一个合取式。这些规则取每个“合取式”结果的“并”,也就是 OR。

例 4.26 我们在例 4.25 中给出了这个算法的一个简单例子。一个更加复杂的例子可以用例 4.25 的条件取反形成,于是我们得到表达式:

$$\text{NOT}(\text{length} \geq 100 \text{ OR } \text{studioName} = 'Fox')(\text{Movie})$$

也就是说,找出长度小于 100 而又不属于 Fox 的所有电影。

在这里,把 NOT 用于本身不是简单比较的表达式中,因此,我们必须将 NOT 下推到表达式中,利用德·摩尔根定律(DeMorgan's Law)的一种形式,即 OR 的反是反的 AND。也就是说,选择可以重写为

$$(\text{NOT}(\text{length} < 100)) \text{ AND } (\text{NOT}(\text{studioName} = 'Fox'))(\text{Movie})$$

现在,我们可以在表达式内部取反,得到表达式

$$\text{length} < 100 \text{ AND } \text{studioName} \neq 'Fox'(\text{Movie})$$

这个表达式可以转换成 Datalog 规则

$$S(t, y, l, c, s, p) \quad \text{Movie}(t, y, l, c, s, p) \text{ AND } l < 100 \text{ AND } s \neq 'Fox'$$

例 4.27 让我们考虑一个类似的例子,其中的选择含有 AND 的反。现在,我们利用德·摩尔根定律的第二种形式,即 AND 的反是反的 OR。我们先从代数表达式开始

$$\text{NOT}(\text{length} \geq 100 \text{ AND } \text{studioName} = 'Fox')(\text{Movie})$$

也就是说,找出不是既长又出自 Fox 的所有电影。

利用德·摩尔根定律(DeMorgan's law)将 NOT 推到 AND 下面,可得

$$(\text{NOT}(\text{length} < 100)) \text{ OR } (\text{NOT}(\text{studioName} = 'Fox'))(\text{Movie})$$

再在比较内部取反,可得

$$\text{length} < 100 \text{ OR } \text{studioName} \neq 'Fox'(\text{Movie})$$

最后,可写出两个规则,每个规则对应 OR 的一部分。所得到的 Datalog 规则为:

1. $S(t, y, l, c, s, p) \quad \text{Movie}(t, y, l, c, s, p) \text{ AND } l < 100$
2. $S(t, y, l, c, s, p) \quad \text{Movie}(t, y, l, c, s, p) \text{ AND } s \neq 'Fox'$

可参阅 A. V. Aho and J. D. Ullman, Foundations of Computer Science, Computer Science Press, New York, 1992.

4.3.6 乘积

两个关系的乘积 $R \bowtie S$ 可以用单一 Datalog 规则表示。这个规则有两个子目标, 一个对应于 R , 一个对应于 S 。这些子目标中的每一个都有不同的变量, 每个变量对应于 R 或 S 的一个属性。头部的 IDB 谓词把出现在两个子目标中的所有变量作为参数, 出现在 R 子目标中的变量排列在 S 子目标的变量之前。

例 4.28 让我们考虑例 4.20 中的两个四属性关系 R 和 S 。规则

$$P(a, b, c, d, w, x, y, z) \leftarrow R(a, b, c, d) \text{ AND } S(w, x, y, z)$$

将 P 定义为 $R \bowtie S$ 。我们使用任意变量, R 的参数对应的变量取自字母表的开始, S 的参数对应的变量取自字母表的结尾。这些变量都出现在规则头部。

4.3.7 连接

我们可以用很像乘积的规则 Datalog 规则取两个关系的自然连接。差别是如果我们想取 $R \bowtie S$, 那么要注意, 对于 R 和 S 中有相同名字的属性, 必须使用相同的变量, 否则必须使用不同的变量。例如, 我们可以使用属性名本身作为变量。头部是每个变量都出现一次的 IDB 谓词。

例 4.29 考虑模式为 $R(A, B)$ 和 $S(B, C, D)$ 的两个关系。它们的自然连接可由如下规则定义:

$$J(a, b, c, d) \leftarrow R(a, b) \text{ AND } S(b, c, d)$$

注意, 子目标中使用的变量如何以显而易见的方式与关系 R 和 S 的属性相对应。

我们也可以用直接的方式将 连接转换为 Datalog。回忆一下 4.1.9 节, 连接如何用带选择的乘积来表示。如果选择条件是合取式, 也就是比较的与 (AND), 那么, 我们可以简单地从乘积对应的 Datalog 规则开始, 然后, 加上附加的算术子目标, 每个子目标对应于一个比较。

例 4.30 让我们考虑例 4.9 的关系 $U(A, B, C)$ 和 $V(B, C, D)$, 当时我们应用 连接

$$U \bowtie_{A < D \text{ AND } U.B = V.B} V$$

我们可以用 Datalog 规则

$$J(a, ub, uc, vb, vc, d) \leftarrow U(a, ub, uc) \text{ AND } V(vb, vc, d) \text{ AND } a < d \text{ AND } ub = vb$$

实现相同的运算。我们使用 ub 作为与 U 的属性 B 相对的变量, 类似地使用 vb, uc 和 vc 。不过, 对于两个关系的 6 个属性用任何 6 个不同的变量都行。前两个子目标引入两个关系, 后两个子目标进行两个比较, 这是出现在 连接条件中的两个比较。

如果 连接的条件不是合取式, 那么, 如在 4.3.5 节讨论的那样, 我们将把它转换为析取范式。然后为每个合取式建一个规则, 在这个规则中, 我们从乘积对应的子目标开始, 接着加上合取式中每个直接项对应的子目标。所有规则的头部是相同的, 而每个参数对应于进行 连接的两个关系中的一个属性。

例 4.31 在这个例子中, 我们将对例 4.30 中的代数表达式作一个简单的修改。AND 将由 OR 取而代之。在这个表达式中没有求反运算, 所以它已经属于析取范式。有两个合

取式, 每个都有单一的直接项。表达式是:

$$U \quad A < D \text{ OR } U.B \quad V.B \quad V$$

使用和例 4. 30 中一样的变量命名模式, 我们将得到两个规则:

- 1. $J(a, ub, uc, vb, vc, d) \quad U(a, ub, uc) \text{ AND } V(vb, vc, d) \text{ AND } a < d$
- 2. $J(a, ub, uc, vb, vc, d) \quad U(a, ub, uc) \text{ AND } V(vb, vc, d) \text{ AND } ub \quad vb$

每个规则都有与所涉及的两个关系对应的子目标, 以及与两个条件 $A < D$ 或 $U.B \quad V.B$ 中的一个对应的子目标。

4. 3. 8 用 Datalog 模拟多重运算

Datalog 规则不仅可以模拟关系代数的单一运算, 实际上还可以模拟任何代数表达式。方法就是检查关系代数表达式对应的表达树, 并为树的每个内部节点建立一个 IDB 谓词。每个 IDB 谓词对应的一个或几个规则是把运算符用于树的相应节点所需要的。作为外延的树的运算项(也就是, 它们是数据库中的关系)由相应的谓词表示。本身是内部节点的运算项由相应的 IDB 谓词表示。

例 4. 32 考虑例 4. 10 的代数表达式

$$title, year \left(\quad length \geq 100 (Movie) \quad \quad \quad studioName = 'Fox' (Movie) \right)$$

它的表达树在图 4. 7 中给出。我们在图 4. 14 中重复了这棵树。它有四个内部节点, 所以我们需要建立四个 IDB 谓词。每个谓词都有一个 Datalog 规则, 图 4. 15 汇总了所有的规则。

最底下的两个内部节点对 EDB 关系 Movie 进行简单的选择, 所以, 我们可以建立 IDB 谓词 W 和 X 来表示这两个选择。图 4. 15 的规则 1 和 2 描述这两个选择。例如, 规则 1 定义 W 为长度至少有 100 的电影元组。

然后, 规则 3 定义谓词 Y 为 W 和 X 的交集, 使用我们在 4. 3. 1 节学过的交集对应的规则形式。最后, 规则 4 定义 Z 为 Y 在属性 title 和 year 上的投影。在这里, 使用了在 4. 3. 4 节所学的模拟投影的技术。谓词 Z 是“回答”谓词; 也就是说, 由 Z 定义的关系和本例开始的代数表达式的结果相同, 而与关系 Movie 的值无关。

图 4. 14 表达树

- 1. $W(t, y, l, c, s, p) \quad Movie(t, y, l, c, s, p) \text{ AND } l \geq 100$
- 2. $X(t, y, l, c, s, p) \quad Movie(t, y, l, c, s, p) \text{ AND } s = 'Fox'$
- 3. $Y(t, y, l, c, s, p) \quad W(t, y, l, c, s, p) \text{ AND } X(t, y, l, c, s, p)$
- 4. $Z(t, y) \quad Y(t, y, l, c, s, p)$

图 4. 15 进行几个代数运算的 Datalog 规则

注意, 在这个例子中, 我们可以用规则 3 的规则体取代图 4. 15 的规则 4 中的子目标 Y。再者, 我们可以用规则 1 和 2 的规则体取代 W 和 X 子目标。因为 Movie 子目标都出现在这两个规则体中, 所以, 我们可以取消一个副本。作为结果, 可以把 Z 定义为单一规则:

$$Z(t, y) \quad Movie(t, y, l, c, s, p) \text{ AND } l \geq 100 \text{ AND } s = 'Fox'$$

然而, 一个复杂的关系代数表达式与单一的 Datalog 规则等价, 这种情况并不常见。

4.3.9 本节练习

练习 4.3.1: 假设 $R(a, b, c)$, $S(a, b, c)$ 和 $T(a, b, c)$ 为三个关系。写出一个或多个定义下列各个关系代数表达式结果的 Datalog 规则:

- (a) $R \bowtie S$
- (b) $R \cup S$
- (c) $R - S$
- * (d) $(R \bowtie S) - T$
- ! (e) $(R - S) \bowtie (R - T)$
- (f) $\pi_{a,b}(R)$
- * ! (g) $\pi_{a,b}(R) \cup \pi_{a,b}(S)$

练习 4.3.2: 假设 $R(x, y, z)$ 为一个关系, 写出一个或多个定义 $\pi_c(R)$ 的 Datalog 规则, 其中 C 为以下条件:

- (a) $x = y$
- * (b) $x < y \text{ AND } y < z$
- (c) $x < y \text{ OR } y < z$
- (d) $\text{NOT } (x < y \text{ OR } x > y)$
- * ! (e) $\text{NOT } ((x < y \text{ OR } x > y) \text{ AND } y < z)$
- ! (f) $\text{NOT } ((x < y \text{ OR } x < z) \text{ AND } y < z)$

练习 4.3.3: 假设 $R(a, b, c)$, $S(b, c, d)$ 和 $T(d, e)$ 为三个关系。对每个自然连接写出单一的 Datalog 规则。

- (a) $R \bowtie S$
- (b) $S \bowtie T$
- ! (c) $(R \bowtie S) \bowtie T$ (注意: 因为自然连接是相关的和可交换的, 所以这三个关系的连接顺序无关。)

练习 4.3.4: 假设 $R(x, y, z)$ 和 $S(x, y, z)$ 是两个关系。写出一个或多个 Datalog 规则来定义每个 $\pi_C(R \bowtie S)$, 其中 C 是练习 4.3.2 的条件之一。对于每个条件, 把每个算术比较解释为左边 R 的属性和右边 S 的属性的比较。例如, $x < y$ 代表 $R.x < S.y$ 。

! 练习 4.3.5: Datalog 规则可以转换为等价的关系代数表达式。虽然我们还没有给出一般性的方法, 但你可以给出很多简单问题的解。对以下每个 Datalog 规则, 写出关系代数表达式来定义与规则头部相同的关系:

- * (a) $P(x, y) \leftarrow Q(x, z) \text{ AND } R(z, y)$
- (b) $P(x, y) \leftarrow Q(x, z) \text{ AND } Q(z, y)$
- (c) $P(x, y) \leftarrow Q(x, z) \text{ AND } R(z, y) \text{ AND } x < y$

4. 4 Datalog 中的递归编程

虽然关系代数可以表达对于关系有用的很多运算,但也存在某些不能写成关系代数表达式的计算。关系代数不能表达的对数据的一种常见运算涉及一个相似的但是不断增长的代数表达式的非限定序列,称为递归。

例 4. 33 一个取自电影产业的递归运算的例子是有关续集(sequel)的问题。通常,每部成功的电影都会制作续集;如果续集制作得好,那么续集之后还有续集,等等。这样,一部电影可能是很长的系列电影的祖先。假定我们有一个关系 SequelOf(Movie, Sequel) 包含有电影和它的直接续集组成的对。在这个关系中元组的例子是:

Movie	sequel
Naked Gun	Naked Gun 2 _{1/2}
Naked Gun 2 _{1/2}	Naked Gun 33 _{1/3}

我们还可以有一个更普遍的电影后集(follow-on) 的概念,表明是续集、续集的续集等等。在上述关系中,Naked Gun 331/3 是 Naked Gun 的一个后集,但在严格意义上不是我们在这里使用的术语“ 续集 ”所指的续集。如果我们在关系中存储直接续集,而当需要后集时,再建立后集,这样就可节省空间。在上述例子中,我们只少存储一个对,但是对于五集的“ Rocky ”电影我们将少存储六个对,而对于 18 集的电影“ Friday the 13th ”则少存储 136 个对。

然而,我们如何从关系 SequelOf 构造后集关系不是立即能看出来的。我们可以把 SequelOf和它本身做一次连接来构造续集的续集。可以通过改名使连接成为自然连接,这样的表达式的例子如下:

$$first, third \left(R(first, second)(SequelOf) \quad s(second, third) (SequelOf) \right)$$

在这个表达式中,SequelOf 两次改名,一次改名使其属性称为 first 和 second,再改名使其属性称为 second 和 third。这样,自然连接对 SequelOf 中的元组(m1, m2) 和(m3, m4)的要求是 m2= m3。于是,将产生元组(m1, m4)。注意,m4 是 m1 续集的续集。

与此类似,我们可以连接 SequelOf 的三个副本以得到续集的续集的续集(例如, Rocky 和 Rocky IV)。事实上,对于任何固定值 i,通过把 SequelOf 和它本身连接 i- 1 次,就可产生第 i 个续集。因此,我们可以取 SequelOf 和这种连接的有限序列的并集,以获得直到某个固定限制的所有续集。

我们在关系代数中不能做的是对每个 i 给出第 i 个续集的表达式非限定序列求出“ 非限定并集 ”。注意,关系代数的并集只允许我们取两个关系的并集,并非不限数量。通过把并集运算符以任何有限的次数用于代数表达式中,就可以得到任何有限数量关系的并集,但是在代数表达式中我们从来不能得到非限定数量关系的并集。

4. 4. 1 固定点运算符

其实,我们不必为了表达“ 相似 ”表达式非限定并集而向关系代数中增加杂乱的约

定。有一个常见的方法来表达通过一个非限定然而正规的过程从其他关系(例如 SequelOf)建立的关系(例如 FollowOn(x, y))(也就是, 电影 y 在例 4. 33 的意义上是电影 x 的一个后集)。我们写一个 FollowOn 用其本身和 SequelOf 描述的方程式, 然后表明 FollowOn 的值是满足方程式的最小关系(最小固定点(least-fixedpoint))。我们将用符号表示要取方程式的最小固定点。

例 4. 34 这里是用于描述关系 FollowOn(x, y)的方程式最小固定点运算符:

$$\begin{aligned} &(\text{FollowOn} = \text{SequelOf}(x, y) (\text{SequelOf}) \\ &\text{R}(x, y) (\text{movie}, y (\text{SequelOf} \bowtie_{\text{sequel} = x} \text{FollowOn}))) \end{aligned}$$

这个方程式一个直觉的陈述是“如果电影 y 是电影 x 的续集或者 y 是 x 续集的后集, 那么电影 y 就是电影 x 的后集”。

为了理解方程式, 我们应当首先注意 FollowOn 的属性是 x 和 y。关系 FollowOn 等于两项的并集。第一项, $\text{SequelOf}(x, y) (\text{SequelOf})$ 是 SequelOf 的一个副本, 经过重命名后它的属性和 FollowOn 的属性相符。第二项是 连接 $\text{SequelOf} \bowtie_{\text{sequel} = x} \text{FollowOn}$, 它把来自 SequelOf 的所有的对(a, b)和来自 FollowOn 的对(b, c)相连接。结果是元组(a, b, b, c), 它的属性分别是 movie, sequel, x 和 y。并集的第二项继续做下去, 首先投影到第一和第四个分量 movie 和 y 上, 然后将属性改名为 x 和 y。

因此, 按照上面的固定点方程, FollowOn 等于关系 SequelOf 和计算续集的后集的第二项结果的并集。也就是说, FollowOn 包含的所有(x, y)对, 或者在 SequelOf 中, 或者符合 y 是 x 的续集的后集。换句话说, y 是 x 的续集的续集的续集……, 把“的续集”这几个字用若干次。

4. 4. 2 计算最小固定点

然而, 我们可能不太清楚为什么根据例 4. 34 的方程式求出的 FollowOn 的最小解恰好是我们所认为的电影后集对的集合。要了解固定点运算符的意义, 就必须了解如何计算最小固定点。在 4. 4. 4 节中, 我们将讨论当差集运算符出现在方程式中时产生的问题, 但是对于没有差集的方程式, 以下方法可行。

- 1. 首先假定方程式左边的关系 R 为空。
- 2. 通过用 R 的旧值计算右边, 重复计算关系 R 的新值。
- 3. 当一次迭代之后, 如旧值和新值相同就停止。

例 4. 35 当关系 SequelOf 包括以下三个元组时, 给出 FollowOn 的计算过程:

movie	Sequel
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

在计算的第一次循环, 假定 FollowOn 为空。这样, 在固定点方程中 SequelOf 和 FollowOn 的连接为空, 而仅有的元组来自并集的第一项 SequelOf。于是, 第一次循环之后, FollowOn 的值和上述的 SequelOf 关系相同。第一次循环之后的情况如图 4. 16 所示。

x	y
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

图 4. 16 第一次循环之后的关系 FollowOn

在第二次循环中, 我们用图 4. 16 的关系作为 FollowOn, 再一次计算固定点方程的右边。并集的第一项 SequelOf 给出三个已有的元组。对于第二项, 我们必须把关系 SequelOf (它有图 4. 16 所示的 3 个元组)和当前的关系 FollowOn(它是相同的关系)连接起来。为此, 我们寻找这样的元组对, 以便来自 SequelOf 的元组的第二个分量等于来自 FollowOn 的元组的第一个分量。

这样, 我们可以从 SequelOf 中得到元组(Rocky, Rocky II), 把它和来自 FellowOn 的元组(Rocky II, Rocky III)匹配成对, 从而为 FollowOn 得到新的元组(Rocky, RockyIII)。与此类似, 我们可以从 SequelOf 中得到元组(RockyII, Rocky III), 并从 FollowOn 中得到元组(RockyIII, RockyIV), 从而为 FollowOn 得到新的元组(RockyII, RockyIV)。然而, 没有其他元组对——来自 SequelOf 的元组和来自 FollowOn 的旧值的另一元组——能连接起来。于是, 第二次循环之后, FollowOn 有 5 个元组, 如图 4. 17 所示。直觉上, 正如图 4. 16 只包含这样的事实: 后集(FollowOn) 基于一个续集, 而图 4. 17 包含的事实是: 后集(FollowOn) 基于一个或两个续集。

x	y
Rocky	Rocky II
Rocky II	RockyIII
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV

图 4. 17 第二次循环之后的关系 FollowOn

在第三次循环中, 我们为 FollowOn 使用来自图 4. 17 的关系, 并再次计算固定点方程的右边。当然, 由此可得到已有的所有元组和另外一个元组。当我们连接来自 SequelOf 的元组(Rocky, Rocky II)和来自 FollowOn 的当前值的元组(Rocky II, Rocky IV)后, 将得到新的元组(Rocky, Rocky IV)。这样, 第三次循环之后, FollowOn 的值如图 4. 18 所示。

当进入第四次循环时, 并没有得到新的元组, 于是过程结束。用该固定点计算定义的真正关系 FollowOn 如图 4. 18 所示。

4. 4. 3 Datalog 中的固定点方程

有用的固定点方程所需要的关系代数表达式往往会很复杂。用 Datalog 规则的聚集表达它们通常比较容易, 从这一节开始将使用这种表示法。正如我们将在 5. 10 节中所看到的那样, 在 SQL3 中, 用得更多的是代数的而不是逻辑的固定点表示法来实现这些想

法, 因为这种风格和 SQL 句法更为一致。

x	y
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV
Rocky	Rocky IV

图 4. 18 第三次循环之后的关系 FollowOn

在逻辑固定点方程的背后, 一般的想法首先是, 假定一个或多个关系其值已知; 这些是外延数据库关系, 即 EDB 关系。其他关系通过出现在规则头部来定义。这些关系是内涵数据库关系, 即 IDB 关系。这些规则的体可能包含其谓词是 EDB 或者 IDB 关系, 以及代数原子的子目标。如果一个或多个 IDB 关系用体中使用相同关系的规则来定义, 那么, 规则用固定点方程就能有效定义这些 IDB 关系, 正如例 4. 34 的关系代数方程中那样。

例 4. 36 我们可以用以下两个 Datalog 规则定义 IDB 关系 FollowOn:

- 1. FollowOn (x, y) SequelOf(x, y)
- 2. FollowOn (x, y) SequelOf(x, z) AND FollowOn(z, y)

第一个规则是基础, 它告诉我们每个续集都是后集。这个规则对应于例 4. 34 的方程中并集的第一项。

第二项规则表明电影 x 续集的每个后集也是 x 的后集。更确切地说, 如果 z 是 x 的续集, 并且已经发现 y 是 z 的后集, 那么 y 也是 x 的后集。

例 4. 36 的规则正好表明和例 4. 35 的固定点方程同样的事情。因此, 对这些规则计算出的 FollowOn 值和例 4. 35 中计算的一样。一般来说, 我们可以用没有求反子目标的 Datalog 规则的任何聚集定义 IDB 关系, 再从所有的 IDB 关系为空开始计算 IDB 关系的值, 并通过把规则用于 EDB 关系和 IDB 关系的以前值重复计算 IDB 关系的新值, 直到 IDB 关系不再改变。

例 4. 37 使用递归的更复杂的例子可以在研究图的路径问题中找到。图 4. 19 是一个示意图。它给出了两个假想的航线 Untried AirLines(UA) 和 Arcane AirLines(AA) 在旧金山(San Francisco)、丹佛(Denver)、达拉斯(Dallas)、芝加哥(Chicago) 和纽约(New York)之间的某些航班。

我们可以想象航班用 EDB 关系表示:

Flights(airline, from, to, departs, arrives)

针对图 4. 19 中的数据, 该关系的元组如图 4. 20 所示。

我们可以提出的最简单的递归问题是“找出通过乘坐一个或多个航班可以从城市 x 到城市 y 的城市对(x, y) ”。以下两个规则描述的关系 Reaches(x, y) 恰好包含这些城市对:

- 1. Reaches(x, y) Flights(a, x, y, d, r)

2. Reaches(x, y) Reaches(x, z) AND Reaches(z, y)

图 4.19 某些航线的航班图

航线	起点站	终点站	起飞时间	到达时间
U A	SF	DEN	930	1230
AA	SF	DAL	900	1430
U A	DEN	CHI	1500	1800
U A	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
U A	CHI	NY	1830	2130

图 4.20 关系 Flights 中的元组

第一个规则表明 Reaches 包含有从第一个城市到第二个城市的直达航班的城市对；在这个规则中，航线 a、起飞时间 d 和到达时间 r 是任意的。第二个规则表明，如果你从城市 x 到达城市 z 并能从 z 到达 y，那么你就能从 x 到达 y。注意，我们在这里已经使用了递归的非线性形式，正如关于“递归的其他形式”的方框中所描述的那样。在这里这种形式多少更方便些，因为在递归规则中，Flights 的另一个使用将涉及 Flights 中未用的分量对应的更多(三个)变量。

为了计算关系 Reaches，我们仿照 4.4.2 节介绍的迭代过程。首先利用规则 1 得到 Reaches 中的下列城市对：(SF, DEN)，(SF, DAL)，(DEN, CHI)，(DEN, DAL(译注：原文误为 CHI))，(DAL, CHI)，(DAL, NY) 和 (CHI, NY)。这些是图 4.19 中用弧线表示的 7 个城市对。

在下一次循环中，我们应用递归规则 2，将这样的弧线对放在一起，即一段弧线的头是另一段弧线的尾。这样做使我们得到附加的城市对 (SF, CHI)，(DEN, NY) 和 (SF, NY)。下一次循环组合这些两段弧线对和所有的单一弧线对共同形成长度最多达到四段弧线的路径。在这个特定的图中，我们没有得到新的城市对。这样，关系 Reaches 包括十个城市对 (x, y)，使得在图 4.19 的图中可以从 x 到达 y。因为我们画图的方式，这些城市对

递归的其他形式

在例 4.34 和 4.36 中, 我们对递归使用右递归的形式, 其中所用的递归关系 FollowOn 出现在 EDB 关系 SequelOf 之后。我们也可以将递归关系放在开始而写出相似的左递归规则。这些规则是:

- 1. FollowOn(x, y) SequelOf(x, y)
- 2. FollowOn(x, y) FollowOn(x, z) AND SequelOf(z, y)

可以非正式地说, 如果 y 是 x 的续集或者 x 的后集的续集, 那么, y 就是 x 的后集。

我们甚至可以两次使用递归关系, 正如在非线性递归中那样:

- 1. FollowOn(x, y) SequelOf(x, y)
- 2. FollowOn(x, y) FollowOn(x, z) AND FollowOn(z, y)

可以非正式地说, 如果 y 是 x 的续集或者 x 的一个后集的后集, 那么, y 就是 x 的后集。所有这三种形式都为关系 FollowOn 给出相同的值: 对 (x, y) 的集合, 其中 y 是 x 的续集,(若干次续集)的续集。

碰巧正是这样的 (x, y), 使得在图 4.19 中 y 在 x 的右边。

例 4.38 当两个航班可以合并成一个更长的航班序列时, 更复杂的定义要求第二个航班离开机场的时间在第一个航班到达该机场之后至少一小时。现在, 使用一个称为 Connects(x, y, d, r) 的 IDB 谓词, 它表明我们可以乘坐一个或多个航班, 起飞的城市为 x、时间为 d, 到达的城市为 y、时间为 r。如果有任何的连接, 那么至少有一个小时建立该连接。

连接的规则是:

- 1. Connects(x, y, d, r) Flights(a, x, y, d, r)
- 2. Connects(x, y, d, r) Connects(x, z, d, t1) AND Connects(z, y, t2, r) AND t1 <= t2 - 100(译注: 原文误为 t2+ 100)

在第一次循环中, 规则 1 给我们的结果是图 4.21 中所示的 8 个连接(Connect)。每个连接对应于图 4.19 中所表明的一个航班。注意, 该图的 7 条弧线中, 有一条表示不同时间的两个航班。

现在, 我们试着使用规则 2 合并这些元组。例如, 这些元组中的第二个和第五个共同给出元组(SF, CHI, 900, 1730)。然而, 第二和第六个元组没有合并, 因为到达达拉斯(Dallas)的时间是 1430, 而离开达拉斯的时间是 1500, 仅在半小时之后。图 4.22 表示第二次循环之后的 Connects 元组。虚线以上是来自第一次循环的原有元组, 而第二次循环增加的 6 个元组在虚线以下表示。虚线本身不是关系的一部分。

在第三次循环中, 原则上必须将图 4.22 中的所有元组对看作是规则 2 的体中两个 Connects 元组的候选者。然而, 如果两个元组都在虚线以上, 那么, 在第二次循环就已经

这些规则仅在午夜没有航班运营的假定下有效。

把它们考虑在内了, 因此, 不会产生以前未见过的 Connects 元组。得到新元组的唯一途径是假设在规则 2 的体中使用的两个 Connects 元组中至少有一个是前一次循环中新增加进来的, 也就是在图 4. 22 的虚线以下。

x	y	d	r
SF	DEN	930	1230
SF	DAL	900	1430
DEN	CHI	1500	1800
DEN	DAL	1400	1700
DAL	CHI	1530	1730
DAL	NY	1500	1930
CHI	NY	1900	2200
CHI	NY	1830	2130

图 4. 21 关系 Connects 的基本元组

x	y	d	r
SF	DEN	930	1230
SF	DAL	900	1430
DEN	CHI	1500	1800
DEN	DAL	1400	1700
DAL	CHI	1530	1730
DAL	NY	1500	1930
CHI	NY	1900	2200
CHI	NY	1830	2130
SF	CHI	900	1730
SF	CHI	930	1800
SF	DAL	930	1700
DEN	NY	1500	2200
DAL	NY	1530	2130
DAL	NY	1530	2200

图 4. 22 第二次循环之后的关系 Connects

第三次循环仅给我们三个新元组, 这些在图 4. 23 的底部给出。这个图中的两条虚线把第一次循环的 8 个元组、第二次循环的 6 个附加元组和来自第三次循环的 3 个新元组分开。第四次循环没有新元组, 至此我们的计算就结束了。这样, 整个关系 Connects 如图 4. 23所示。

4. 4. 4 递归规则中的求反

有时, 在涉及递归的规则中还要求反。混合递归和求反的方法有安全的和不安全的。通常认为只有求反不出现在固定点运算的内部, 使用求反才是合适的。为了了解差别, 我们考虑两个递归和求反的例子, 一个是合适的, 另一个是矛盾的。我们将看到当有递归

时, 只有“ 分层 ”求反是有用的, 术语“ 分层 ”的确切定义将在例子之后给出。

x	y	d	r
SF	DEN	930	1230
SF	DAL	900	1430
DEN	CHI	1500	1800
DEN	DAL	1400	1700
DAL	CHI	1530	1730
DAL	NY	1500	1930
CHI	NY	1500	2200
CHI	NY	1830	2130
SF	CHI	900	1730
SF	CHI	930	1800
SF	DAL	930	1700
DEN	NY	1500	2200
DAL	NY	1530	2130
DAL	NY	1530	2200
SF	CHI	900	2130
SF	NY	900	2200
SF	NY	930	2200

图 4.23 第三次循环之后的 Connects 关系

例 4.39 假定我们想找出图 4.19 中的城市对(x, y), 使得通过 UA 可以从 x 飞到 y (可能通过其他几个城市), 但是通过 AA 却不可以。我们可以递归定义谓词 UA reaches, 正如我们在例 4.37 定义 Reaches 那样, 只是将我们自己限制在 UA 航班。定义如下:

- 1. UA reaches(x, y) Flights(UA, x, y, d, r)
- 2. UA reaches(x, y) UA reaches(x, z) AND UA reaches(z, y)

与此类似, 我们可以递归定义谓词 AA reaches 为这样的城市对(x, y), 即人们仅利用 AA 航班从 x 旅行到 y。定义如下:

- 1. AA reaches(x, y) Flights(AA, x, y, d, r)
- 2. AA reaches(x, y) AA reaches(x, z) AND AA reaches(z, y)

现在, 看一个简单的问题, 计算 UAonly 谓词, 它包含通过 UA 航班而不能通过 AA 航班从 x 到 y 的城市对(x, y), 用非递归规则:

UAonly(x, y) UA reaches(x, y) AND NOT AA reaches(x, y)

这个规则计算 UA reaches 和 AA reaches 的集合差。

对于图 4.19 的数据, 可以看出 UA reaches 包含下列城市对: (SF, DEN), (SF, DAL), (SF, CHI), (SF, NY), (DEN, DAL), (DEN, CHI), (DEN, NY) 和 (CHI, NY)。这个集合是按照 4.4.2 节描述的迭代的固定点过程计算出来的。与此类似, 对这些数据我们可以计算出 AA reaches 的值: (SF, DAL), (SF, CHI), (SF, NY), (DAL, CHI), (DAL, NY) 和 (CHI, NY)。当取这些对的集合差时, 我们得到: (SF, DEN), (DEN, DAL), (DEN,

CHI) 和 (DEN, NY)。这四对的集合就是谓词 UAonly 的值。

例 4. 40 现在, 让我们考虑一个工作情况不好的抽象的例子。假定我们有一个 EDB 谓词 R。这个谓词是一元的(一个参数), 有一个元组(0)。有两个 IDB 谓词, P 和 Q, 也是一元的。它们用以下两个规则来定义:

- 1. $P(x) \leftarrow R(x) \text{ AND NOT } Q(x)$
- 2. $Q(x) \leftarrow R(x) \text{ AND NOT } P(x)$

可以非正式地说, 两个规则告诉我们, R 中的元素 x 或在 P 中, 或在 Q 中, 但不同时在两者之中。注意 P 和 Q 相互递归定义。

当我们在 4. 4. 1 节对递归规则意味着什么下定义时, 我们说需要最小固定点, 也就是作为代数方程使规则为真的最小关系。规则 1 表明, 作为关系, $P = R - Q$, 而规则 2 表示 $Q = R - P$ 。因为 R 仅包括元组(0), 所以 P 或 Q 中只能为(0)。但是(0)在哪里呢? 它不可能两处都不在, 否则方程不满足。比如 $P = R - Q$ 将意味着 $\{(0)\} = \{(0)\} - \{(0)\}$, 此式为假。

如果假设 $P = \{(0)\}$, 而 $Q = \{(0)\}$, 那么, 我们确实得到两个方程的解。 $P = R - Q$ 成为 $\{(0)\} = \{(0)\} - \{(0)\}$, 此式为真, 而 $Q = R - P$ 变成 $\{(0)\} = \{(0)\} - \{(0)\}$, 此式也为真。

然而, 我们也可以假设 $P = \{(0)\}$, 而 $Q = \{(0)\}$ 。这种选择也满足两个规则。这样我们有两个解:

- (a) $P = \{(0)\} \quad Q = \{(0)\}$
- (b) $P = \{(0)\} \quad Q = \{(0)\}$

两个都是最小的, 在某种意义上, 如果我们使任何元组离开任何关系, 那么, 结果关系都不再满足这两个规则。因此, 不能在两个最小固定点(a)和(b)之间作出决定, 这样, 就不能回答比如“P(0)为真吗?”这样简单的问题。

在例 4. 40 中, 我们看到, 当递归和求反太紧密地纠缠在一起时, 通过找出最小固定点来定义递归规则或固定点方程的想法不再有效。可能有多个最小固定点, 并且这些固定点可能互相矛盾。如果用某个其他方法定义递归性求反能更有效就好了, 但遗憾的是, 没有关于这些规则或方程意味着什么的普遍一致的意见。

因此, 常规的做法是限制我们自己在递归中分层求反。例如, 在 5. 10 节讨论的关于递归的 SQL3 标准就做了这种限制。下面我们将看到, 当分层求反时, 存在计算一个特定最小固定点(可能出自许多这样的固定点)的算法, 该最小固定点符合直觉的关于规则的含义。我们如下定义分层的性质:

- 1. 画一个图, 它的节点对应于 IDB 谓词。
- 2. 如果在规则的头部有谓词 A, 并在求反子目标有谓词 B, 那么就画一个从节点 A 至节点 B 的弧线。用符号“-”标记这个弧线以标明它是求反的弧线。
- 3. 如果规则的头部有谓词 A, 而非求反子目标有谓词 B, 那么就画一个从节点 A 至节点 B 的弧线。这个弧线不用负号作为符号。

如果这个图有一个环包含一个或更多的求反弧线, 那么表明递归没有分层。否则, 就表明图已分层。我们可以用层划分 IDB 谓词。谓词 A 的层是从 A 开始的一条路径上求反弧线的最大数量。

如果把递归分层, 那么, 我们可以从它们的最底层开始计算 IDB 谓词分层的次序。这

个策略将产生规则的最小固定点之一。更重要的是,由它们的层隐含的次序计算 IDB 谓词好像总是有意义的,并给我们“正确的”固定点。与此相对,正如我们在例 4.40 中看到的那样,即使有许多固定点可以从中选择,但未分层的递归可能使我们完全没有“正确”的固定点。

例 4.41 例 3.49 的谓词对应的图如图 4.24 所示。AAreaches 和 UAreaches 在 0 层,因为没有从它们的节点开始的路径涉及求反的弧线。UAonly 在 1 层,因为存在这样的路径,它从该节点开始具有一条求反弧线,但不存在具有多条求反弧线的路径。这样,在开始计算 UAonly 之前,我们必须全面地计算 AAreaches 和 UAreaches。

图 4.24 由分层递归构造的图

图 4.25 由未分层递归构造的图

比较一下我们为例 4.40 的 IDB 谓词构造图的情况,这个图在图 4.25 中给出。因为规则 1 具有和求反子目标 Q 有关的头部 P,所以有一条从 P 至 Q 的求反弧线。因为规则 2 具有和求反子目标 P 有关的头部 Q,所以,在相反方向上,也有一条求反弧线。于是就有一个求反的环,表明规则没有分层。

4.4.5 本节练习

练习 4.4.1: 如果我们针对图 4.19 的图增加或删除弧线,就可能改变例 4.37 的关系 Reaches、例 4.38 的关系 Connects 或例 4.39 的关系 UAreaches 和 AAreaches 的值。如果做出如下改变,请给出这些关系的新值。

- * (a) 增加一条标记为 AA 1900-2100 的从 CHI 至 SF 的弧线;
- (b) 增加一条标记为 UA 900-1100 的从 NY 至 DEN 的弧线;
- (c) 同时增加(a)和(b)中提到的两条弧线;
- (d) 删除从 DEN 至 DAL 的弧线。

练习 4.4.2: 写出 Datalog 规则(若需求反,则用分层求反),以便对例 4.33 中后集(“Follow-On”)概念的以下几种修改加以描述。你可以使用例 4.36 中定义的 EDB 关系 SequelOf 和 IDB 关系 FollowOn。

- * (a) $P(x, y)$ 意味着电影 y 是电影 x 的后集,但不是 x 的续集(正如用 EDB 关系 SequelOf 所定义的)。
- (b) $Q(x, y)$ 意味着 y 是 x 的后集,但不是 x 的续集或续集的续集。
- ! (c) $R(x)$ 意味着电影 x 至少有两个后集。注意,两个都可能是续集,而不是一个是续集,而另一个是续集的续集。
- ! (d) $S(x, y)$ 意味着 y 是 x 的后集,但是 y 最多有一个后集。

练习 4.4.3: ODL 类及其联系可以用一个关系 $Rel(class, rclass, mult)$ 描述。其中, mult

给出联系的多重性, 或用 multi(多)对应多值联系, 或用 single(单)对应单值联系。前两个属性是相关的类, 联系从 class 至 rclass(相关类)。例如, 图 4. 26 中给出的关系 Rel 表示图 2. 6 中不断滚动的电影实例的三个 ODL 类。

类(Class)	相关类(rclass)	多(mult)
Star	Movie	多
Movie	Star	多
Movie	Studio	单
Studio	Movie	多

图 4. 26 用关系数据表示 ODL 联系

我们也可以将这些数据看成一个图, 其中节点是类, 而弧线从一个类到一个相关类, 弧线边上标记着合适的多(multi) 或单(single)。图

4. 27 描绘出图 4. 26 的数据所对应的图。

对于下列各种情况, 写出 Datalog 规则, 如果必须求反, 就用分层求反来表示所描述的谓词。你可以用 Rel 作为一个 EDB 关系。给出用你的规则对图

图 4. 27 用图表示联系

4. 26 的数据进行计算时每次循环的结果。

- (a) 谓词 P(class, eclass), 意味着从 class 至 eclass 的类对应的图中有一条路径。可以把后面的类想象成是嵌入在 class 中, 因为在某种意义上它是第一个类的对象的一部分的一部分的.....一部分。
- * ! (b) 谓词 S(class, eclass) 和 M(class, eclass)。第一个意味着在 class 中有一个 eclass“单值嵌入”, 也就是, 沿着从 class 到 eclass 的路径每条弧线都标记着单(single)。第二个, M, 意味着在 class 中有一个 eclass 的“多值嵌入”, 也就是说, 沿着从 class 到 eclass 的路径至少有一条弧线标记为多(multi)。
- (c) 谓词 Q(class, elcass) 说明说有一条从 class 到 eclass 的路径但没有单值路径。你可以用本练习中前面定义 IDB 谓词。

4. 5 对关系的约束

关系模型提供了一种方法来表示通常的约束, 例如在 2. 5 节介绍的参照完整性约束。实际上, 我们将看到关系代数给我们提供了简便的方法来表示种类繁多的其他约束, 即使函数依赖也可以用关系代数表达, 正如我们在例 4. 44 中看到的那样。约束在数据库编程中十分重要, 我们将在第 6 章讨论 SQL 数据库系统如何加强在关系代数中表达的相同种类的约束。

在本练习中不认为空路径为“路径”。

4.5.1 用关系代数作为约束语言

用关系代数表达式来表达约束共有两种方法。

1. 如果 R 是关系代数表达式, 那么 $R = \emptyset$ 就是一个约束, 它表明“ R 的值必须为空”, 或相当于, “在 R 的结果中没有元组”。

2. 如果 R 和 S 是关系代数表达式, 那么 $R \subseteq S$ 就是一个约束。它表明“ R 结果中的每个元组也必须在 S 的结果中”。当然, S 的结果可能包括不是由 R 产生的附加元组。

这些表示约束的方法在它们所能表示的内容方面实际上是等价的, 但有时一个或另一个比较清楚或比较简洁。也就是说, 约束 $R \subseteq S$ 也可以仅仅写为 $R - S = \emptyset$ 。原因是, 如果 R 中的每个元组也在 S 中, 那么 $R - S$ 肯定为空。相反地, 如果 $R - S$ 不包含元组, 那么 R 中的每个元组必然在 S 中(否则它将在 $R - S$ 中)。

另一方面, 第一种形式的约束, $R = \emptyset$, 也可以仅仅写为 $R \subseteq \emptyset$ 。在技术上, \emptyset 不是关系代数表达式, 但是因为有关计算结果为 \emptyset 的表达式, 例如 $R - R$, 所以用 \emptyset 作为关系代数表达式没有什么危害。

在随后几节, 我们将看到如何用这两种形式之一表达重要的约束。正如我们将在第 6 章看到的, 第一种形式——等于空集——最广泛地用于 SQL 编程。然而, 如上所述, 如果我们愿意就按照集合-包含来进行思考, 并在以后把约束转换为等于空集的形式, 一切悉听尊便。

4.5.2 参照完整性约束

在 2.5 节称为“参照完整性约束”的一种普通类型约束断言, 出现在一个环境中的值也出现在另一个相关的环境中。我们将参照完整性视为联系“有意义”的一个要素。也就是说, 如果对象或实体 A 与对象或实体 B 相关, 那么 B 必须实际存在。例如, 用 ODL 的术语, 如果对象 A 中的联系实际上用一个指针表示, 那么指针必须不为空, 并且必须指向一个真正的对象。

在关系模型中, 参照完整性约束看起来有些不同。如果一个关系 R 的一个元组有个分量值为 v , 那么因为我们的设计意图的缘故, 可以期望 v 将出现在另一个关系 S 的某个元组的特定分量中。下面的例子将说明如何用关系代数表示关系模型中的参照完整性。

例 4.42 让我们考虑我们的不断滚动的电影数据库模式, 特别是两个关系

```
Movie(title, year, length, inColor, studioName, producerC# )
MovieExec(name, address, cert# , networth)
```

我们可能合理地假定每部电影的制片人必须出现在 `MovieExec` 关系中。如果不是, 就有点问题了, 这时至少要让实现关系数据库的系统告诉我们: 我们有一部电影, 而系统并不知道它的制片人。

更确切地说, 每个 `Movie` 元组的 `producerC#` 分量也必须出现在某个 `MovieExec` 元组的 `cert#` 分量中。因为行政长官由证件号唯一地标识, 这样系统将保证电影的制片人将在电影的行政长官中找到。我们可以用集合-包含表达这个约束:

$$\text{producerC\# (Movie)} \quad \text{cert\# (MovieExec)}$$

左边表达式的值是出现在 Movie 元组中 producerC# 分量的所有证件号的集合。同样地, 右边表达式的值是 MovieExec 元组的 Cert# 分量中所有证件号的集合。我们的约束表明前面集合中的每个证件号也必须出现在后面的集合中。

随便提一下, 我们可以用空集的等式表达相同的约束:

$$\text{producerC\# (Movie)} - \text{cert\# (MovieExec)} =$$

例 4. 43 我们可以类似地表达一个参照完整性约束, 其中所涉及的“值”用多个属性表示。例如, 我们可能想声明在关系

$$\text{starsIn(MovieTitle, movieYear, starName)}$$

中提到的任何电影也出现在下面的关系中

$$\text{Movie(title, year, length, inColor, studioName, producerC\#)}$$

在两个关系中电影都用 title-year 对表示, 因为我们一致认为这两个属性中的任何一个都不足以单独识别一部电影。通过比较把两个关系投影到合适的分量表上而产生的 title-year 对约束

$$\text{movieTitle, movieYear (starsIn)} \quad \text{title, year (Movie)}$$

表达了这样的参照完整性约束。

4. 5. 3 附加约束的例子

同样的约束表示法允许我们表达比参照完整性多得多的东西。例如, 我们可以把任何函数依赖表示成代数约束, 不过这种表示法比我们已经使用的函数依赖表示法更为麻烦。

例 4. 44 让我们把关系

$$\text{MovieStar(name, address, gender, birthdate)}$$

的函数依赖

$$\text{name} \quad \text{address}$$

表示成代数约束。

想法是如果我们构造 MovieStar 的所有元组对 (t₁, t₂), 就一定不能出现一个对在 name 分量上一致而在 address 分量上不一致。为了构造元组对, 我们使用笛卡尔 (Cartesian) 积, 而为了寻找违背函数依赖的元组对, 我们使用选择。然后, 我们用结果等于 来声明约束。

首先, 因为我们取一个关系和它本身的乘积, 所以为了使乘积的属性都有名字, 就需要至少把一个副本改名。为了简洁, 让我们使用两个新名字, MS1 和 MS2, 以便引用 MovieStar 关系。于是函数依赖就可以用如下的代数约束来表示:

$$\text{MS1.name = MS2.name AND MS1.address} \quad \text{MS2.address}(\text{MS1} \bowtie \text{MS2}) =$$

上式中, 乘积 MS1 ⋈ MS2 中的 MS1 是改名运算

$$\text{MS1(name, address, gender, birthdate) (MovieStar)}$$

的简写, 而 MS2 是 MovieStar 的类似的改名。

我们有时需要的另一种约束是域约束。通常, 域约束就是简单地要求属性的值有特定

的数据类型, 例如整型或长度为 30 的字符串。这些约束不能用关系代数声明, 因为像整型这种类型不是关系代数的一部分。然而, 通常, 域约束涉及到属性所需要的特定值。如果可取值的集合可以用选择条件语言表达, 那么这个域约束就可以用代数约束语言表达。

例 4. 45 假定我们希望规定 MovieStar 性别属性的仅有的合法值为 ' F' 和 ' M' 。我们可以用如下的代数方法表示这个约束。

$$gender \neq 'F' \text{ AND } gender \neq 'M' (MovieStar) =$$

也就是说, MovieStar 中 gender 分量既不等于 ' F' 也不等于 ' M' 的元组集合为空。

最后, 有某些约束不在 2. 5 节描述的任何一个范畴内。代数约束语言允许我们表达许多新种类的约束。这里是一个例子。

例 4. 46 假定我们希望要求一个人必须至少有净资产 \$ 10 000 000 才可以做电影制片公司的总裁。这个约束不能归类为域、单值或参照完整性约束。然而我们可以用如下的代数方法表示它。

首先, 我们需要用 连接把两个关系

$$\begin{aligned} &MovieExec(name, address, cert\#, \text{networth}) \\ &Studio(name, address, presC\#) \end{aligned}$$

连起来, 利用的条件是来自 studio 的 presC# 和来自 MovieExec 的 cert# 相等。该连接把包含制片公司和行政长官的两个元组组合成对, 以使行政长官就是制片公司的总裁。如果我们从这个关系中选择净资产少于 \$ 10 000 000 的元组, 那么根据我们的约束, 就会有一个必须为空的集合。这样, 我们可以把该约束表达为:

$$netWorth < 10000000 (studio \bowtie_{presC\# = cert\#} MovieExec) =$$

表达同一约束的另一种方法是, 把代表制片公司总裁的证件号的集合和代表具有净资产至少 \$ 10 000 000 的行政长官的证件号集合进行比较; 前者必须是后者的子集。包含

$$presC\# (studio) \subseteq cert\# (\text{networth} \geq 10\,000\,000 (MovieExec))$$

表达了上述想法。

4. 5. 4 本节练习

练习 4. 5. 1: 表达对练习 4. 4. 1 的关系(在这里再次给出)的下列约束

$$\begin{aligned} &Product(maker, model, type) \\ &PC(model, speed, ram, hd, cd, price) \\ &Laptop(model, speed, ram, hd, screen, price) \\ &Printer(model, color, type, price) \end{aligned}$$

可以用包含的形式或者表达式等于空集的形式写出你的约束。对于练习 4. 1. 1 的数据, 指出对你的约束的任何违背之处。

- * (a) 处理器速度小于 150 的 PC 销售价格一定不超过 \$ 1500。
- (b) 屏幕尺寸小于 11 英寸的便携式电脑一定至少有 1G 字节的硬盘或者销售价格一定低于 \$ 2000。
- ! (c) PC 的制造商不会同时制造便携式电脑。
- * !! (d) PC 的制造商还必须制造处理器速度至少一样快的便携式电脑。

! (e) 如果便携式电脑的主存比 PC 更大, 那么便携式电脑的价格必然比 PC 更高。

练习 4.5.2: 用关系代数表达下列约束。约束基于练习 4.1.3 的关系:

Classes(class, type, country, numGuns, bore)

Ships(name, class, launched)

Battles(name, date)

Outcomes(ship, battle, result)

可以用包含的形式或者表达式等于空集的形式写出你的约束。对于练习 4.1.3 的数据, 指出对你的约束的任何违背之处。

(a) 任何等级的舰艇都不会有大于 16 英寸口径的火炮。

(b) 如果某个等级的舰艇火炮多于 9 门, 那么它们的口径一定不大于 14 英寸。

! (c) 任何等级的舰艇都不会多于两艘。

! (d) 没有一个国家既有战列舰又有巡洋舰。

!! (e) 多于 9 门火炮的舰艇不可能在和少于 9 门火炮的舰艇的战役中被击沉。

练习 4.5.3: 用 Datalog 以及关系代数表达约束是可能的。我们写出一个或几个 Datalog 规则来定义一个特殊的其值限定为空的 IDB 谓词。用 Datalog 写出下列各个约束。

* (a) 例 4.42 的约束。

(b) 例 4.43 的约束。

(c) 例 4.44 的约束。

(d) 例 4.45 的约束。

(e) 例 4.46 的约束。

! 练习 4.5.4: 假定 R 和 S 是两个关系。假定 C 为这样的参照完整性约束: 当 R 有一个元组在特定属性 A_1, A_2, \dots, A_n 上有某些值 v_1, v_2, \dots, v_n 时, 必定有 S 的一个元组在特定属性 B_1, B_2, \dots, B_n 上有相同值 v_1, v_2, \dots, v_n 。说明如何用关系代数表达约束 C。

!! 练习 4.5.5: 假定 R 为关系, 并假定函数依赖 $A_1, A_2, \dots, A_n \rightarrow B$ 是涉及 R 的属性的函数依赖。用关系代数写出表明该函数依赖在 R 中一定成立的约束。

4.6 包的关系运算

虽然元组的集合(也就是关系)是数据的简单自然模型, 正如它可能出现在数据库中的那样, 但商业数据库系统很少完全基于集合。在某些情况下, 关系正如它们出现在数据库系统中那样, 允许有重复元组。回忆一下, 如果一个“集合”允许一个成员多次出现, 那么该集合就称为包或多集。在这一节, 我们将把关系作为包而不是集合; 也就是说, 我们将允许相同的元组在关系中多次出现。当我们指“集合”时, 意味着没有重复元组的关系; 而“包”则意味着可能有(也可能没有)重复元组的关系。

例 4.47 图 4.28 中的关系是元组的包。其中, 元组(1, 2)出现三次而元组(3, 4)出现一次。如果图 4.28 是集合-值关系, 我们将不得不消除元组(1, 2)的两次出现。在一个包-值关系中, 我们的确允许同一元组的多次出现, 但类似于集合, 元组的顺序无关紧要。

A	B
1	2
3	4
1	2
1	2

图 4. 28 一个包

4. 6. 1 为什么用包？

当考虑关系实现的有效性时, 我们可以看到在几种情况下, 允许关系为包而非集合就可以加速对集合的运算。例如, 当进行投影时, 允许结果关系为包就使我们可以独立地处理每个关系。如果我们希望用集合作为结果, 就需要把每个元组通过投影去掉不想要的分量得到的结果和其他所有已投影的元组的结果进行比较, 以确保我们以前未见到这个投影。然而, 如果我们允许用包作为结果, 那么就可以简单地对每个元组进行投影, 并将其加到结果中; 而不需要和其他已投影的元组进行比较。

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

图 4. 29 例 4. 48 的包

例 4. 48 如果我们允许结果为包而不消除(1, 2) 的重复出现, 那么图 4. 28 的包可以是图 4. 29 所示的关系在属性 A 和 B 上投影的结果。如果我们使用关系代数通常的投影运算符, 从而消除重复, 结果仅为

A	B
1	2
3	4

注意用包作为结果, 虽然数据量较大, 却可以计算得更快, 因为不需要把每个元组(1, 2) 或(3, 4) 与以前产生的元组进行比较。

进而, 如果我们对关系进行投影, 以得到一个聚合(如同 5. 5 节所讨论的那样), 例如“找出图 4. 29 中 A 的平均值”, 就不能用集合模型来想象投影的关系。作为一个集合, A 的平均值为 2, 因为图 4. 29 中 A 的值只有两个——1 和 3, 而它们的平均值为 2。然而, 如果我们把图 4. 29 中的 A 列作为包(1, 3, 1, 1) 来处理, 那么将得到图 4. 29 的四个元组之间 A 的正确平均值 1. 5。

允许用包作为结果就可以节省时间的另一种情况是, 取两个关系的并集。如果我们计算并集 R ∪ S 并坚持用集合作为结果, 那么就必须检查 S 中的每个元组是否为 R 的成员。

如果 S 中的某个元组在 R 中找到了, 那么 S 中的该元组就不加入并集中; 否则才加入并集中。然而, 如果我们允许用包作为结果, 那么只要把 R 和 S 的所有元组都复制到答案中即可, 而不考虑它们是否出现在两个关系中。

4. 6. 2 包的并集、交集和差集

当我们取两个包的并集时, 每个元组出现的次数就增加了。也就是, 如果 R 是一个包, 其中元组 t 出现了 n 次, 而 S 也是一个包, 其中元组 t 出现了 m 次, 那么在包 R ∪ S 中, 元组 t 将出现 n+ m 次。注意, n 或 m(或两者)可以为 0。

当我们取两个包 R 和 S 的交集时, 如果在 R 和 S 中元组 t 分别出现 n 和 m 次, 那么在 R ∩ S 中元组 t 将出现 min(n, m) 次。当我们计算 R − S(包 R 和 S 的差集) 时, 元组 t 在 R − S 中出现 max(0, n − m) 次。也就是说, 如果 t 在 R 中出现的次数比它在 S 中出现的次数多, 那么在 R − S 中元组 t 出现的次数为它在 R 中出现的次数, 减去它在 S 中出现的次数。然而, 如果 t 在 S 中出现的次数至少和它在 R 中出现的次数一样多, 那么 t 就根本不会出现在 R − S 中。直觉上, t 在 S 中的每个出现都“抵消”R 中的一个出现。

例 4. 49 假设 R 是图 4. 28 的关系, 也就是, R 是一个包, 其中元组(1, 2)出现 3 次、(3, 4)出现一次。假设 S 为包

A	B
1	2
3	4
3	4
5	6

那么包的并集 R ∪ S 仍为包, 其中(1, 2)出现 4 次(三次对应于它在 R 中的出现, 一次对应于它在 S 中的出现); (3, 4)出现三次, (5, 6)出现一次。

包的交集 R ∩ S 是包

A	B
1	2
3	4

其中(1, 2)和(3, 4)各出现一次。也就是, (1, 2)在 R 中出现 3 次, 在 S 中出现一次, 而 min(3, 1) = 1, 所以(1, 2)在 R ∩ S 中出现一次。类似地, (3, 4)在 R ∩ S 中出现 min(1, 2) = 1 次。元组(5, 6)在 S 中出现一次, 但在 R 中出现 0 次, 在 R − S 中出现 min(0, 1) = 0 次。

包的差集 R − S 是包

A	B
1	2
1	2

对集合按包运算

设想我们有两个集合 R 和 S 。可以把每个集合看作这样一个包: 该包只是碰巧任何元组最多出现一次。假定计算交集 $R \cap S$, 但是我们把 R 和 S 看作包, 并使用包的交集规则。那么我们得到的结果将与把 R 和 S 看作集合应得到的结果相同。也就是说, 把 R 和 S 看作包, 一个元组 t 在 $R \cap S$ 中的出现次数是它在 R 和 S 中出现次数的最小值。因为 R 和 S 是集合, 所以 t 在每个集合中只能出现 0 或 1 次。不管使用包还是集合的交集规则, 我们发现 t 在 $R \cap S$ 中最多只能出现一次, 如果它同时在 R 和 S 中, 那么它正好在 $R \cap S$ 中出现一次。与此类似, 如果使用包的差集规则比较 $R - S$ 或 $S - R$, 我们将得到和使用集合规则完全相同的结果。

然而, 并的表现却不同, 这取决于我们把 R 和 S 看作集合还是包。如果我们使用包规则计算 $R \cup S$, 即使 R 和 S 是集合, 结果可能并不是集合。特别是, 如果元组 t 在 R 和 S 中都出现, 而且我们对并集使用包规则, 那么 t 在 $R \cup S$ 中将出现两次, 但如果使用集合规则, 那么 t 在 $R \cup S$ 中就只出现一次。因此, 当取并集时, 必须特别指明, 我们正在使用并集的包定义还是集合定义。

要了解原因, 注意, $(1, 2)$ 在 R 中出现三次, 而在 S 中出现一次, 所以在 $R - S$ 中它出现 $\max(0, 3 - 1) = 2$ 次。元组 $(3, 4)$ 在 R 中出现一次, 而在 S 中出现两次, 所以在 $R - S$ 中, 它出现 $\max(0, 1 - 2) = 0$ 次。在 R 中没有其他元组出现, 所以在 $R - S$ 中就不会再有其他元组。

作为另一个例子, 包的差集 $S - R$ 是包

A	B
3	4
5	6

元组 $(3, 4)$ 出现一次, 因为这是它出现在 S 中的次数减去它出现在 R 中的次数之差。由于同样的原因, 元组 $(5, 6)$ 在 $S - R$ 中出现一次。结果包在这种情况下碰巧为一个集合。

4. 6. 3 包的投影

我们已经解释了包的投影。正如我们在例 4. 48 所看到的那样, 每个元组在投影过程中独立进行。如果 R 是图 4. 29 的包, 当我们计算包的投影 $\pi_{A, B}(R)$ 时, 将得到图 4. 28 的包。

如果在投影过程中消除一个或多个属性使得从几个元组产生出相同的元组, 这些重复元组并不从包投影的结果中消除。于是, 图 4. 29 关系 R 的三个元组 $(1, 2, 5)$, $(1, 2, 7)$ 和 $(1, 2, 8)$ 在投影到属性 A 和 B 上之后, 每个都产生出相同的元组 $(1, 2)$ 。在包的结果中, 元组 $(1, 2)$ 出现三次, 而在集合的投影中, 该元组只出现一次。

包的代数定律

代数定律表示两个关系代数表达式之间等价, 而表达式的参数为代表关系的变量。等价意味着无论用什么关系替代这些变量, 两个表达式都定义相同的包。一个众所周知的定律就是并集交换律: $R \cup S = S \cup R$ 。不管我们认为关系变量 R 和 S 是代表集合还是代表包, 该定律碰巧都成立。然而, 另外有些定律当关系代数用传统方式解释——关系作为集合——时成立, 但是当把关系解释为包时, 并不成立。这类定律的一个简单例子就是并集上集合差的分配律, $(R \cup S) - T = (R - T) \cup (S - T)$ 。该定律对集合成立, 而对包却不成立。要了解为什么对包不成立, 假定 R 、 S 和 T 各有元组 t 的一个副本。那么左边的表达式有一个 t , 而右边的表达式则没有。作为集合, 两边都没有 t 。对包的代数定律的某些探索将出现在练习 4. 6. 4 和 4. 6. 5 中。

4. 6. 4 包的选择

为了把选择用于包, 我们把选择条件独立地用于每个元组。由于总是用包, 因此我们并不消除结果中的重复元组。

例 4. 50 如果 R 是包

A	B	C
1	2	5
3	4	6
1	2	7
1	2	7

那么包选择 $\sigma_6(R)$ 的结果是

A	B	C
3	4	6
1	2	7
1	2	7

也就是, 除了第一个元组以外其他所有元组都符合选择条件。最后两个元组在 R 中是重复的, 它们中的每一个都包含在结果中。

4. 6. 5 包的乘积

包的笛卡尔积规则是意料之中的规则。一个关系的每个元组和另一个关系的每个元组匹配成对, 而不考虑它是否重复。作为结果, 如果元组 r 在关系 R 中出现 m 次, 元组 s 在关系 S 中出现 n 次, 那么在乘积 $R \bowtie S$ 中, 元组 rs 将出现 mn 次。

例 4. 51 假设 R 和 S 为图 4. 30 所表示的包, 那么乘积 $R \bowtie S$ 包含 6 个元组, 如图 4. 30(c) 所示。注意, 我们为集合关系提出的关于属性名的常用习惯可完全一样地用到包

上。因此,属于 R 和 S 两个关系的属性 B, 在乘积中出现两次, 每次都用一个关系名作为前缀。

A	B
1	2
1	2

(a) 关系 R

B	C
2	3
4	5
4	5

(b) 关系 S

A	R.B	S.B	C
1	2	2	3
1	2	2	3
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

(c) 乘积 $R \times S$

图 4.30 计算包的乘积

4.6.6 包的连接

连接包也没有出现意想不到的事。我们把一个关系中的每个元组和另一个关系中的每个元组进行比较,判断该元组对是否连接成功,如果成功就将结果元组放在答案中。生成答案时,并不消除重复元组。

例 4.52 图 4.30 中所见到的关系 R 和 S 的自然连接 $R \bowtie S$ 是

A	B	C
1	2	3
1	2	3

也就是, R 的元组(1, 2)和 S 的元组(2, 3)相连接。因为在 R 中有(1, 2)的两个副本并且在 S 中有(2, 3)的一个副本, 所以有两个元组对连接得到元组(1, 2, 3)。没有其他来自 R 和 S 的元组连接成功。

作为在相同关系 R 和 S 上的另一个例子, 连接

$$R \bowtie_{R.B < S.B} S$$

产生包

A	R.B	S.B	C
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

连接的计算如下,R 的元组(1, 2)和 S 的元组(4, 5)满足连接条件。由于每个元组在各自的关系中都出现两次,因此连接元组在结果中出现的次数是 2×2 即 4 次。其他可能的元组连接——R 的(1, 2)和 S 的(2, 3)——不满足连接条件,所以这种组合并不出现在结果中。

4. 6. 7 包的运算用于 Datalog 规则

如果没有求反的关系子目标,那么也可把计算包的选择、投影和连接技术用于 Datalog 规则。大致的步骤是,首先对由不同子目标表示的关系进行连接,然后把结果按算术子目标所包含的内容进行选择,再把结果按头部的格式进行投影。在每一步,我们都使用适于包的算法。

把 4. 2. 4 节给出的 Datalog 规则的第二种计算方法一般化,在概念上比较简单。回忆一下,该技术涉及到查找每个非求反的关系子目标,并用该子目标的谓词对应的关系的所有元组替代它。如果通过对每个子目标元组的选择给每个变量一致的赋值,并且算术子目标全都为真,那么随着对变量的这种赋值我们就会看到头部变成了什么。结果元组就放在头部关系中。

因为我们现在处理的是包,所以不消除头部的重复元组。而且,当我们考虑子目标元组的所有组合时,一个子目标对应关系中出现 n 次的元组在与其他子目标对应元组的所有组合形成的合取式中,将作为该子目标的元组考虑 n 次。

例 4. 53 考虑规则

$$H(x, z) \leftarrow R(x, y) \text{ AND } S(y, z)$$

并假设 R 和 S 为图 4. 30 中的关系。子目标的元组赋值一致(也就是,每个子目标的 y 值赋值相同)的唯一的的情况是,用 R 的元组(1, 2)对第一个子目标赋值,并用 S 的元组(2, 3)对第二个子目标赋值。由于(1, 2)在 R 中出现两次,而(2, 3)在 S 中出现一次,因此将有两个元组赋值,对变量的赋值为 $x = 1, y = 2$ 和 $z = 3$ 。头部的元组为(x, z),每个元组的赋值均为(1, 3)。于是在头部关系 H 中,元组(1, 3)出现两次,而没有出现其他元组。也就是说,关系

H 1	H 2
1	3
1	3

是用这个规则定义的头部的关系,在这里我们把关系的属性随意命名为 H 1 和 H 2。更概括地说,如果元组(1, 2)在 R 中出现 n 次,元组(2, 3)在 S 中出现 m 次,那么元组(1, 3)将在 H 中出现 nm 次。

如果一个关系用几个规则定义,那么结果是由每个规则产生的所有元组的包构成的并集。

例 4. 54 考虑用两个规则:

注意,规则中不能有任何求反的关系子目标。在包的模式下,不能显式地定义具有求反关系子目标的任意 Datalog 规则。

$$H(x,y) = S(x,y) \text{ AND } x > 1$$
$$H(x,y) = S(x,y) \text{ AND } y < 5$$

定义的关系 H 。假定关系 S 有图 4. 30(b) 的值, 它是

B	C
2	3
4	5
4	5

第一个规则将 S 的三个元组都放入 H , 因为这三个元组的第一个分量都大于 1。第二个规则只把元组(2, 3)放入 H , 因为(4, 5)并不满足条件 $y < 5$ 。于是, 结果关系 H 有元组 (2, 3) 的两个副本, 以及元组(4, 5)的两个副本。

4. 6. 8 本节练习

* 练习 4. 6. 1: 假设 PC 为图 4. 10(a) 的关系, 并假定我们计算投影 $speed(PC)$ 。该表达式作为集合, 它的值是什么? 作为包, 它的值又是什么? 什么是这个投影的元组平均值, 当作为集合处理时, 该投影的元组平均值是什么? 作为包, 其平均值又是什么?

练习 4. 6. 2: 对投影 $hd(PC)$ 重复练习 4. 6. 1。

练习 4. 6. 3: 该练习引用练习 4. 1. 3 的“战列舰”关系。

(a) 表达式 $bore(Classes)$ 产生具有各种等级的火炮口径的单列关系。对于练习 4. 1. 3 的数据, 这个关系作为一个集合是什么? 作为一个包又是什么?

! (b) 写出关系代数表达式以给出舰艇(不是等级)的火炮口径。你的表达式必须对包有意义; 也就是, 数值 b 出现的次数 a 必须是火炮口径为 b 的舰艇数。

! 练习 4. 6. 4: 某些关系代数定律对作为集合的关系成立, 对作为包的关系也成立。解释为什么以下每个定律既对包成立又对集合成立。

- * (a) 并的结合律: $(R \cup S) \cup T = R \cup (S \cup T)$
- (b) 交的结合律: $(R \cap S) \cap T = R \cap (S \cap T)$
- (c) 自然连接的结合律: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- (d) 并的交换律: $(R \cup S) = (S \cup R)$
- (e) 交的交换律: $(R \cap S) = (S \cap R)$
- (f) 自然连接的交换律: $(R \bowtie S) = (S \bowtie R)$
- (g) $L(R \cup S) = L(R) \cup L(S)$ 。其中, L 是任意的属性表。

- * (h) 交上并的分配律: $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$
- (i) $C \text{ AND } D(R) = C(R) \cap D(R)$ 。其中, C 和 D 是关于 R 中元组的任意条件。

!! 练习 4. 6. 5: 下列代数定律对集合成立, 而对包不成立。解释为什么对集合成立, 并给出它们对包不成立的反例。

- * (a) $(R \cup S) - T = R \cup (S - T)$
- (b) 并上交的分配律。 $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$
- (c) $C \text{ OR } D(R) = C(R) \cup D(R)$ 。其中, C 和 D 是关于 R 中元组的任意条件。

4.7 关系模型的其他外延

还有其他一些概念和运算,它们不是正式的关系模型的一部分,但是出现在实际的查询语言中。在这一节中,我们提到的运算和概念包括更新关系、计算聚合(比如对关系按列求和),以及定义关系的“视图”或命名的函数。这些运算和概念都出现在数据库语言 SQL 中,并将在第 5 章进一步讨论。我们也将第 8 章对查询语言 OQL 的讨论中看到其中的某些部分。

4.7.1 更新

关系代数或 Datalog 在某种意义上都是“查询语言”,其中每一个都给我们求出一个关系(即答案),也就是某些给定关系的函数。虽然查询很重要,但是不能改变的数据库不会令人感兴趣。因此,所有实际的数据库语言都既包括查询数据库的能力也包括更新数据库的能力。至少,我们需要如下命令。

1. 向关系中插入元组;
2. 从关系中删除元组;
3. 通过改变一个或多个分量修改已有元组。

4.7.2 聚合

关系代数运算符可独立于同一关系中的其他元组而对某些元组进行运算。通常,我们希望组合单一关系的元组以产生某个聚合值,也就是以某种方式组合的所有元组的函数。例如,在不断滚动的电影实例中我们可能希望:

- 统计 Movie 关系中提到的不同电影的数量。
- 产生一个表格给出每个制片公司制作的电影长度的总和。
- 找出具有最大净资产的电影行政长官。

于是,实际的数据库查询语言允许我们对关系中的列应用聚合运算符(主要是统计、求和、求平均值、求最小值和最大值)。

4.7.3 视图

我们可以把关系代数表达式看作一个“程序”,它计算关系 R 并打印结果或另外产生 R 作为结果。然而,对关系表达式还有另一种解释。我们可以把它看作是定义一个关系的公式,而在该公式应用于实际的关系之前该关系并未产生。这样的公式在数据库术语体系中称为视图。我们将看到,通常把视图作为给定的名字,并用这些名字作为其他关系表达式的参数,好像视图就是实际的关系。

Datalog 规则也说明了查询和视图的区别。回忆一下,我们把用 Datalog 规则定义的谓词或关系看作是“内涵的”;也就是说,它们是一种关系的定义,而这种关系不需要以“外延的”或存储的形式存在。视图等价于内涵谓词。正如内涵谓词可以在规则体内使用一样,视图可以用在代数表达式中作为参数。同样地,正如 Datalog 规则的聚集可以用于由存储

关系组成的数据库, 当需要时, 对视图同样可以计算。

4.7.4 空值

在许多情况下我们必须对元组的分量赋值, 但又不能给出具体的值。例如, 我们可能知道 Kevin Costner 是影星, 但不知道他的出生日期。因为所有的 MovieStar 元组都有 birthdate 分量, 我们怎么办呢? 答案是, 我们对该分量可以使用一个称为空值的特殊值 NULL。NULL 值在某种意义上就像任何其他值一样。然而, 在其他方面它不是一个值。特别是, 当我们连接两个关系时, 并不认为两个 NULL 分量彼此相等。例如, 两个影星都用 NULL 作为他们的 birthdate 分量的值, 但并不能假定他们的出生日期相同。

使用 NULL 值可以有许多不同的解释。这里是最常见的几种:

1. 值未知: 也就是说, “我知道有某个值属于它, 但不知道它是什么。”如同上面所讨论的, 未知的出生日期就是一个例子。
2. 值不适用: “没有值在这里有意义。”例如, 如果 MovieStar 关系具有 spouse 属性, 那么未婚的影星将需要用空值对应该属性, 不是因为不知道配偶姓名, 而是因为没有。
3. 值隐瞒: “我们无权知道属于这的值。”例如, 一个未列出的电话号码可能作为 NULL 出现在 phone 属性对应的分量中。

4.8 本章总结

关系代数: 这种代数是关系模型查询语言的一种重要形式。它的主要运算是并、交、差、选择、投影、笛卡尔积、自然连接、连接和改名。

Datalog: 这种逻辑的形式是关系模型查询语言的另一种重要形式。在 Datalog 中, 人们写出规则, 在规则中通过由子目标组成的体定义谓词或关系。头部和子目标均为原子, 而原子包括一个用到某些参数的谓词(求反是任选的)。可以用关系代数表达的所有查询都可以用 Datalog 表达。

递归 Datalog: Datalog 规则也可以是递归的, 即允许一个关系通过自己来定义。递归 Datalog 规则的含义是, 所定义关系元组的最小固定点、最小集合, 也就是使规则头部恰好等于规则体所隐含的内容。

分层求反: 当递归涉及求反时, 最小固定点可能不是唯一的, 在某些情况下没有可接受的 Datalog 含义。因此, 必须禁止在递归内部使用求反, 从而导致对分层求反的需求。对于这类规则有一个(或许是几个中的一个)最小固定点是规则普遍接受的含义。

作为包的关系: 在商业数据库系统中, 实际上关系是包, 其中相同的元组允许多次出现。对集合的关系代数运算可扩展到包, 但某些代数定律不能成立。

商业系统中的关系: 除了使用关系的包模型之外, 商业系统还提供未出现在关系代数或 Datalog 中的运算。这些运算包括对关系中元组的插入、删除和修改, 关系上的聚合, 以及元组的空值。

4.9 本章参考文献

关系代数是关系模型基础论文[4]的另一个贡献。逻辑查询语言的来历不太直接。Codd 在关系模型的一篇早期论文[5]中介绍了称为关系演算的前序逻辑形式。关系演算是一种表达式语言,非常像关系代数,实际上表达能力等价于关系代数,在[5]中已经证明。

Datalog 受到编程语言 Prolog 的启发,看上去更像逻辑规则。因为它允许递归,比关系演算表达力更强。[6]首先把大量的逻辑推导作为查询语言,而[2]则把这种想法放在数据库系统的环境中。论文[8]最早使用了表达约束的查询。

用分层的方法给出固定点的正确选择,这一想法来自[3],不过用这种方法计算 Datalog 规则的想法也是[1]、[7]和[10]的独立想法。另外关于分层求反,关于关系代数、Datalog 和关系演算之间的联系以及关于 Datalog 规则的计算,无论是否求反均可在[9]中找到。

- [1] Apt, K. R., H. Blair, and A. Walker, Towards a theory of declarative knowledge, in Foundations of Deductive Databases and Logic Programming (J. Minker, ed.), pp. 89 ~ 148, Morgan-Kaufmann, San Francisco, 1998.
- [2] Bancilhon, F. and R. Ramakrishnan, An amateur's introduction to recursive query-processing strategies, ACM SIGMOD Intl. Conf. on Management of Data, pp. 16 ~ 52, 1986.
- [3] Chandra, A. K. and D. Harel, Structure and complexity of relational queries, J. Computer and System Sciences 25: 1, pp. 99 ~ 128.
- [4] Codd, E. F., A relational model for large shared data banks, Comm. ACM 13: 6, pp. 377 ~ 387, 1970.
- [5] Codd, E. F., Relational completeness of database sublanguages, in Database Systems (R. Rustin, ed.), Prentice Hall, Englewood Cliffs, NJ, 1972.
- [6] Gallaire, H. and J. Minker, Logic and Databases, Plenum Press, New York, 1978.
- [7] Naqvi, S., Negation as failure for first-order queryies, Proc. Fifth ACM Symp. on Principles of Database Systems, pp. 114 ~ 122, 1986.
- [8] Nicolas, J. - M., Logic for improving integrity checking in relational databases, Acta Informatica, 18: 3, pp. 227 ~ 253, 1982.
- [9] Ullman, J. D., Principles of Database and Knowledge-Base Systems, Volume I, Computer Science Press, New York, 1988.
- [10] Van Gelder, A., Negation as failure using tight derivations for general logic programs, in Foundations of Deductive Databases and Logic Programming (J. Minker, ed.), pp. 149 ~ 176, Morgan-Kaufmann, San Francisco, 1988.

第 5 章 数据库语言 SQL

最常用的关系数据库系统通过称为 SQL(可以读做“sequel”)的语言对数据库进行查询和更新。SQL 的含义是“结构化查询语言(Structured Query Language)”。尽管 SQL 的许多重要的特征已经超越了关系代数的范畴,例如聚合运算(如求和、统计)以及数据库更新等,但是,SQL 的重要核心是和关系代数等价的。

目前有许多不同版本的 SQL 语言。首先,存在两个不同的主要标准:ANSI(American National Standards Institute,美国国家标准协会)的 SQL 和 1992 年采用的修正后的标准,称为 SQL-92 或 SQL2。还有一种正在制订的标准称为 SQL3,它在 SQL2 的基础上扩展了许多新的特性,如递归(recursion)、触发(trigger)以及对象(object)等。其次,还存在数据库管理系统的主要供货商所开发的不同版本的 SQL。这些版本都支持最初的 ANSI 标准。它们还在很大程度上遵循新近的 SQL2 标准,尽管各自都有一些超越了 SQL2 的变化和扩展,包含了前面提到的 SQL3 标准的某些特性。

在本章及随后的两章中,我们将讨论作为数据库查询语言的 SQL 的使用。本章集中讨论 SQL 的通用查询接口。也就是说,当坐在终端前发出数据库查询命令或更新请求时,我们把 SQL 看成是独立于操作系统的查询语言。对查询的回答,将在终端上显示出来。在本章和随后两章所讨论的 SQL 中,我们通常将遵循 SQL2 标准,着重分析存在于几乎所有的商业系统以及早期 ANSI 标准中的特性。在某些情况下,如果 SQL2 标准不能充分地适用于所讨论的问题,我们将遵循可以得到的最新的正在制订的 SQL3 标准。

本章和随后两章的意图是对于“SQL 是什么”这样一个问题给读者提供感性的认识,更多的是从“教学”的角度而不是从“手册”的角度来进行描述。因此我们只集中于讨论 SQL 最常用的一些特性。关于语言及其“方言”的更多的细节在引用的参考书中可以找到。

5.1 SQL 的简单查询

或许 SQL 中最简单的查询就是从某个关系中查找满足某种条件的一些元组(tuple)。这种查询类似于关系代数中的选择。这种简单的查询,同几乎所有的 SQL 查询一样,使用了具有 SQL 特性的三个关键字:SELECT、FROM 以及 WHERE。

例 5.1 在本例和随后的例子中,我们会用到在 3.9 节描述过的数据库模式。回顾一下,这些关系模式如图 5.1 所示。在 5.7 节,我们将看到如何用 SQL 来表述模式信息,但是现在,让我们假设在 3.9 节所提及的每个关系和每个域在 SQL 中都有其相应的位置。

作为我们的第一个查询,询问在关系

```
Movie(title, year, length, inColor, studioName, producerC# )
```

中由 Disney 制片公司在 1990 年制作的所有电影。在 SQL 中, 我们这样表示:

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

这个查询语句展示了大多数 SQL 查询语句特有的 select-form-where 格式。

```
Movie( title, year, length, inColor, studioName, producerC# )
StarsIn( movieTitle, movieYear, starName)
MovieStar( name, address, gender, birthdate)
MovieExec( name, address, cert# , netWorth )
Studio( name, address, presC# )
```

图 5.1 数据库模式的实例(复制的)

- FROM 子句给出了查询所涉及的关系。在我们的例子中, 是对关系 Movie 进行查询。
- WHERE 子句给出了查询的条件, 这很像关系代数中的选择条件, 要与查询相匹配则选出的元组必须满足这个条件。在这里条件就是, 元组的 studioName 属性取值为 'Disney', 元组的 year 属性取值为 1990。所有符合这两条约定的元组满足条件, 而其他的则不满足。
- SELECT 子句说明了满足条件的元组的哪些属性将作为回答的组成部分。例中的星号(*)表明把整个元组作为回答的内容。查询的结果是由这个处理过程产生的所有元组组成的关系。

解释这种查询的一种方法是, 考虑 FROM 子句中所提及的关系中的每个元组。将 WHERE 子句中的条件应用于这一元组。更确切地说, WHERE 子句中所提及的每个属性都用这一元组中相应属性的值来代替。然后对条件进行求值, 如果为真, 则 SELECT 子句中出现的各属性将构成答案中的一个元组。

比如当 SQL 查询处理程序遇到了一个 Movie 元组

title	year	length	inColor	studioName	producerC#
Pretty Woman	1990	119	true	Disney	999

(这里, 999 是虚构的该电影制片人的证书号), 对于 WHERE 子句的条件, 用值 'Disney' 来代替属性 studioName, 而用值 1990 来代替属性 year, 由于这些值是上述元组相应的属性值, 因此, WHERE 子句就变成了

```
WHERE 'Disney' = 'Disney' AND 1990 = 1990
```

这个条件显然为真, 因此, Pretty Woman 这一元组将通过 WHERE 子句的检测, 从而成为查询结果的一部分。

5.1.1 SQL 的投影

如果愿意的话, 我们能够删除选定元组中的某些分量; 也就是说, 我们可以将 SQL 查询

所产生的关系投影(projection)到它自己的某些属性上。我们可以列出 FROM 子句中提到的关系的任何属性来代替 SELECT 子句中的星号(*)。结果将投影到所列的属性上。

例 5. 2 假设我们希望修改例 5. 1 中的查询, 只产生电影的名称和长度, 可以这样写:

```
SELECT title, length
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

结果是以 title 和 length 为表头的两列表。这个表中的元组都有两个属性, 即电影的名称及其长度, 并且电影是由 Disney 在 1990 年制作的。例如, 关系模式及其中的一个元组就像这样:

title	length
Pretty Woman	119

有时, 我们希望生成的关系中列的标题与 FROM 子句所提到的关系中的属性能够有所区别。我们可以在属性名的后面加上关键字 ‘ AS’ 和将在结果关系中出现的 ‘ 别名’ 。关键字 AS 是任选的, 有些较早的的 SQL 系统经常将其省略。这意味着, 别名可以直接跟在它所代表的属性的后面, 中间不需要任何标点符号。

例 5. 3 我们可以修改例 5. 2 来生成一个分别由 name 和 duration 来代替 title 和 length 的关系, 如下所示:

```
SELECT title AS name, length AS duration
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

结果是和例 5. 2 相同的元组集合, 只是由属性 name 和 duration 作为列的标题。例如, 结果关系可能这样开始:

name	duration
Pretty Woman	119

SELECT 子句的另一个选项是用表达式来代替属性。

例 5. 4 假如我们希望得到像例 5. 3 那样的输出, 但是电影的长度以小时为单位。我们可以将上例中的 SELECT 子句替换为

```
SELECT title AS name, length* 0. 016667 AS lengthInHours
```

于是将产生相同的 name-length 结果对, 但是电影的长度将以小时计算, 同时第二列将以属性 lengthInHours 为标题。

例 5. 5 我们甚至能够允许常量作为 SELECT 子句的一项。这看起来好象像有意

因此, SQL 中的关键字 SELECT 实际上密切地对应关系代数的投影运算符, 而关系代数中的选择运算符则对应于 SQL 查询的 WHERE 子句。

大小写无关性

SQL 是大小写无关的, 也就是说它把大写字母和小写字母看成相同的字母。例如, 尽管我们选择将关键字以大写的形式写出, 如 FROM, 其实将它写成 From 或者 from, 甚至 FrOm 都同样正确。同样, 属性、关系、别名等的名字也是大小写无关的。只有在引号之内, SQL 才区别大小写字母。因此, 'FROM' 和 'from' 是不同的字符串; 当然它们都不是关键字 FROM。

义, 但是应用程序有时会要求在 SQL 的输出显示中加入一些有用的信息。像如下的查询:

```
SELECT title, length* 0.016667 AS length, 'hrs.' AS inHours
FROM Movie
WHERE studioName = 'DISNEY' AND year = 1990;
```

生成的元组如下:

title	length	inHours
Pretty Woman	1.98334	hrs.

我们安排了称为 inHours 的第三列, 这同第二列的标题 length 相对应。回答中的每个元组在第三列都有一个常量 'hrs.', 于是, 可把它看作是附加在第二列数值上的单位。

5.1.2 SQL 的选择

通过 SQL 的 WHERE 子句, 我们能够获得关系代数中的所有选择运算, 甚至更多。跟在 WHERE 后面的表达式包括和普通的 C 或 PASCAL 语言一样的条件表达式。

在构造表达式的时候, 我们可以使用 6 种通用的比较运算符对值进行比较: = 、< > 、< > 、< = 和 > = 。这些运算符同它们在 PASCAL 中的用法完全一样, 有着明显的字面意义(如果你不是一个 PASCAL 迷的话, 可以告诉你, < > 是“不等于”的意思)。

可以进行比较的值包括常量以及在 FROM 后面提到的关系的属性。我们也可以在对这些值进行比较之前用 + 、* 等通用算术运算符对这些值进行数值计算。例如, (year - 1930) * (year - 1930) < 100 对于 1930 前后 9 年的年份(year)都为真。我们可以对字符串用并置运算符 @@ 进行连接操作; 例如, 'foo' @@ 'bar' 得到值 'foobar' 。

一个比较的例子是例 5.1 中的

```
studioName = 'Disney'
```

检测关系 Movie 中的属性 studioName 是否等于常量 'Disney' 。这个常量是字符串类型的; 在 SQL 中字符串通过在其前后加单引号来表示。在 SQL 中也允许有数值常量、整数和实数, 并用通常的表示法来表示实数, 如- 12.34 或 1.23E45。

比较的结果是布尔值: 或者为真(TRUE)或者为假(FALSE)。布尔值可以用逻辑运算符组合起来, AND(与)、OR(或)、NOT(非)的用法和在 PASCAL 中的用法相同。例如, 我们在例 5.1 中可以看到两个条件如何用 AND 组合起来。当且仅当两个比较都满足的

布尔值和位串的表达

我们可以把 SQL 中的布尔值表示为位串的特殊情况。二进制位串由 B 加上引号内 0 和 1 的串来表示。因此, B '011' 表示一个三位的串, 其中第一位是 0 而其余两位是 1。也可以采用十六进制表示法, X 后面加上引号内的十六进制数字串(0 到 9 和 a 到 f, 后者表示数字 10 到 15)。例如, X '7ff' 表示 12 位的串, 一个 0 后面有 11 个 1。注意: 每个十六进制数字表示 4 位二进制数, 并允许前几位为 0。

布尔值 TRUE 可以用 1 位二进制数表示, 即, B '1'。类似地, FALSE 由 B '0' 表示。

时候, 例中 WHERE 子句的值才为真; 也就是说, 制片公司的名字为 'Disney' 并且制作的年份是 1990, 结果才为真。下面是一些有复杂 WHERE 子句的查询例子。

例 5.6 下面的查询找出所有 1970 年以后制作的黑白电影(black-and-white)。

```
SELECT title
FROM Movie
WHERE year > 1970 AND NOT inColor;
```

在这个条件中, 我们又遇到了两个布尔值的 AND。前者是普通的比较, 而后者是对属性值 inColor 取反。因为属性 inColor 是布尔类型, 因此对其进行布尔运算是有意义的。

接下来, 考虑查询

```
SELECT title
FROM Movie
WHERE (year > 1970 OR length < 90)
      AND studioName = 'MGM'
```

此查询找出由米高梅制片公司制作的或者年份在 1970 年以后或者长度小于 90 分钟的电影的名称。注意在这里, 比较可以用圆括号来分组。这里之所以要用括号是因为 SQL 中的逻辑运算符同其他语言中所用到的一样, 有优先级: AND 的优先级高于 OR, 而 NOT 的优先级则高于前两者。

5.1.3 字符串的比较

如果两个字符串具有相同的字符序列, 那么它们是相等的。SQL 允许说明不同类型的字符串, 例如固定长度的字符数组(array)和可变长度的字符列表(list)。这样, 我们就希望能够在不同的字符串类型之间进行合理的强制转换。比如说, 一个字符串如 foo 可能加上 7 个填充(pad)字符存储为固定长度为 10 的字符串, 也可能存储为可变长度的字符串。这样我们就希望这两种类型的值彼此相等, 并且也等于字符串常量 'foo'。

当我们对两个字符串用“大于或小于”这一类运算符(如< 或> =)进行比较时, 我们

至少可以认为字符串分别存储在数组或列表中, 至于它们实际上是如何存储的则是依赖于实现的问题, 而这些并没有在任何 SQL 标准中予以规定。

LIKE 表达式中的换码字符

如果想在 LIKE 表达式的模式中包含字符 % 或 _ , 怎么办呢? SQL 允许我们给单个模式指定换码字符, 这个字符由我们任意选定, 而不是用某个特定字符作为换码字符(例如大多数 UNIX 命令中用反斜杠作为换码字符)。我们可以在模式后面加上关键字 ESCAPE 和带引号的选定的换码字符来实现这一点。模式中如果在 % 或 _ 之前加上换码字符, 就会从字面上分别解释为 % 或 _ , 而不是分别代表任意字符序列或任意某个字符的通配符了, 例如, 表达式

s LIKE 'x% % x%' ESCAPE 'x'

中 x 作为模式 x% % x% 的换码字符。字符序列 x% 指的是单独的字符 % 。于是, 这个模式就将匹配所有以 ' % ' 开始并结束的字符串。

是想知道这两个字符串按字典顺序(即按词典顺序或按字母顺序)是否其中一个先于另一个。也就是说, 如果有两个字符串 $a_1a_2...a_n$ 和 $b_1b_2...b_m$, 如果 $a_1 < b_1$, 或者 $a_1 = b_1$ 而 $a_2 < b_2$, 或者 $a_1 = b_1, a_2 = b_2$ 而 $a_3 < b_3, ...$, 依此类推, 那么前者“小于”后者。如果 $n < m$ 并且 $a_1a_2...a_n = b_1b_2...b_n$, 那么 $a_1a_2...a_n < b_1b_2...b_m$, 也就是说, 第一个字符串是第二个字符串的真前缀。例如, 'fodder' < 'foo', 因为这两个字符串的前两个字符 fo 均相同, 而 fodder 的第三个字符先于 foo 的第三个字符; 同样, 'bar' < 'bargain', 因为前者恰好是后者的真前缀。在字符串相等的场合, 我们或许希望在不同的字符串类型之间进行合理的强制转换。

SQL 还提供了基于简单模式(pattern)匹配的字符串比较功能。下面是另一种形式的比较表达式

s LIKE p

其中, s 是字符串, 而 p 是模式, 表示包含 ' % ' 和 ' _ ' 两种任选的特殊字符的字符串。p 中普通的字符只和 s 中完全相同的字符匹配, 而 ' % ' 却能同 s 中的 0 个或多个字符序列匹配, ' _ ' 能同 s 中的任意一个字符匹配。类似地, 只有当字符串 s 不匹配模式 p 的时候, s NOT LIKE p 才为真。

例 5.7 我们记得一部电影的名称是“Star x”, 并记得 x 是 4 个字母的单词。那么这部电影到底是什么呢? 我们可以通过查询找出所有这类电影名:

```
SELECT title
FROM Movie
WHERE title LIKE 'Star _ _ _ _ ' ;
```

该查询检测 Movie 的 title 属性是否是长度为 9 个字符的字符串, 并且前五个字符是 Star 和一个空格。而后四个字符可以是任意的, 因为任意四个字符的序列都能同四个连续的符号 ' _ ' 相匹配。查询的结果是完全匹配的电影名的集合, 比如“Star Wars”和“Star Trek”。

例 5.8 让我们来搜索电影名中含有所有格(' s)的所有电影。想要的查询是这样的:


```
SELECT title
FROM Movie
WHERE title LIKE '% ' 's% ' ;
```

要想理解这个模式,我们就必须首先考察单引号,在 SQL 语句中用单引号把字符串括起来,这时单引号并不表示它本身。SQL 的习惯用法是在字符串中用两个连续的单引号表示一个单独的单引号,而不表示字符串的结束。这样,模式中的 's 将与电影名中的一个单引号和紧跟着的一个 s 相匹配。

's 两边的% 字符将匹配两个任意的字符串。因此,所有以 's 作为子串的电影名都将匹配这个模式,查询的结果将包括“ Logan's Run ”或“ Alice's Restaurant ”之类的电影。

5.1.4 日期和时间的比较

SQL 语言的不同实现版本通常都将日期和时间作为专用数据类型来支持。它们的值常常表示成多种格式,如 5/ 14/ 1948 或 14 May 1948。在这里我们仅描述对格式有着非常明确规定的 SQL2 标准表示法。

日期(date)由关键字 DATE 及其后面带引号的特殊格式的字符串表示。例如,DATE '1948-05-14' 就符合这种格式。前面 4 个字符是表示年份的阿拉伯数字。接下来是连字符和表示月份的 2 位数字。值得注意的是,就像我们的例子那样,在 1 位数字表示的月份前面加上了一个 0。最后又是一个连字符以及表示日的 2 位数字。对于日,就像月那样,我们有必要为了构成两位的数字而在前面补 0。

与此类似,时间(time)由关键字 TIME 和带引号的字符串来表示。字符串中小时占两位数字,按军用时钟计时(24 小时制)。然后依次是一个冒号、两位数字的分,再一个冒号、两位数字的秒。如果需要秒的小数部分,我们可以在后面加上一个小数点,以及我们想要的多个有效位。例如,TIME '15 00 02.5' 表示下午 3 点过两秒半,这时所有的学生都下课了,课在下午 3 点结束。

我们可以使用对数字或字符串进行比较运算的比较运算符来对日期或时间进行比较运算。也就是说,日期的小于(<) 意味着第一个日期早于第二个日期;时间的小于(<) 意味着前者早于后者(在同一天内)。

5.1.5 输出的排序

我们可能会要求查询生成的元组按照一定的顺序表示出来。排序可以基于某个属性的值,相等时根据第二个属性的值,若还相等,可以根据第三个属性的值,依此类推。我们只要在 select-from-where 语句中加入一个子句就可以实现对输出的排序:

```
ORDER BY 属性表
```

默认的顺序为升序,但是我们可以通过附加关键字 DESC(指“ 降序 ”)实现高序数先输出。同样,我们可以用关键字 ASC 来指定升序,但是实际上这个关键字是没有必要的。

例 5.9 下面是对例 5.1 中我们原来查询的改写,从关系

```
Movie (title, year, length, inColor, studioName, producerC# )
```

中查询 1990 年 Disney 公司的电影。要使列出的电影先按长度排序,短的在前,对于长度

相同的电影,再按字母排序,我们可以这样描述:

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

如果存在易于理解的属性顺序(由于 SQL 中的关系是由属性表说明的(如 5.7.2 节所述),所以应该有这种顺序),只要我们愿意,就可以用属性号来代替属性名。因此,上面的 ORDER BY 子句可以根据我们在关系 Movie 中所列出的属性的标准顺序而写成:

```
ORDER BY 3, 1;
```

5.1.6 本节练习

练习 5.1.1: 如果查询中的 SELECT 子句为

```
SELECT A B
```

我们如何才能知道其中的 A 与 B 是两个不同的属性,或者 B 是 A 的别名?

练习 5.1.2: 基于不断滚动的电影数据库的实例:

```
Movie(title, year, length, inColor, studioName, producerC# )
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert# , netWorth)
Studio(name, address, presC# )
```

请写出下列 SQL 查询语句:

- * (a) 找出米高梅制片公司(MGM studios)的地址。
- (b) 找出桑德拉·布洛克(Sandra Bullock)的出生日期(birthdate)。
- * (c) 找出在 1980 年拍摄过电影的所有影星,或者拍摄过电影名中含有“ Love ”的电影的所有影星。
- (d) 找出净资产至少 1 000 万美元的所有行政长官。
- (e) 找出所有的男影星或者是住址中含有 Malibu 的影星。

练习 5.1.3: 用 SQL 写出下列的查询。查询将引用练习 4.1.1 中所描述的数据库模式:

```
Product (maker, model, type)
PC ( model, speed, ram, hd, cd, price)
Laptop (model, speed, ram, hd, screen, price)
Printer (model, color, type, price)
```

使用练习 4.1.1 的数据,请写出查询的结果。

- * (a) 找出价格低于 1 600 美元的所有个人计算机(PC)的型号(model)、速度(speed)以及硬盘容量(hd)。
- * (b) 同(a)的要求,另外将“ 速度 ”(speed)改为“ 兆赫 ”(megahertz),将“ 硬盘 ”(hd)改为“ 吉字节 ”(gigabytes)。

- (c) 找出打印机(Printer)的制造商(maker)。
- (d) 找出费用高于 2 000 美元的便携式电脑(laptop)的型号、内存容量(ram)以及屏幕尺寸(screen)。
- (e) 从 Printer 关系中找出所有彩色打印机的元组。记住 color 是布尔值的属性。
- (f) 找出具有 6 倍速或 8 倍速光驱(6x or 8x cd)而价格低于 2 000 美元的所有个人计算机的型号、速度以及硬盘容量。你可以把属性 cd 看作是字符串类型的。

练习 5.1.4: 基于练习 4.1.3 的数据库模式:

Classes (class, type, country, numGuns, bore, displacement)

Ships (name, class, launched)

Battles (name, date)

Outcome (ship, battle, result)

写出下列查询, 并根据练习 4.1.3 中的数据给出查询的结果:

- (a) 列出至少拥有十门火炮(numGuns)的所有舰艇等级(class)的名称(name)和所属国家(country)。
- (b) 找出所有在 1918 年以前下水的舰艇的 name, 而结果的名称用 shipName 来表示。
- (c) 找出所有在作战中沉没的舰艇的 name, 同时给出使它们沉没的 battle(战役名)。
- (d) 找出所有 name 和 class 同名的舰艇。
- (e) 找出 name 以字母 R 开头的所有舰艇的名称。
- ! (f) 找出舰名中包含三个或三个以上单词(如 King George V)的所有舰艇的名称。

5.2 涉及多个关系的查询

关系代数的许多功能来自于它能够通过连接(joins)、乘积(products)、并(unions)、交(intersections)、差(differences)等运算将两个或两个以上的关系连起来。我们在 SQL 中可以使用这五种运算之中的任何一个。集合论中的运算——并(union)、交(intersection)和差(difference)可以直接出现在 SQL 中, 我们将在 5.2.5 节学习有关的内容。首先, 让我们来了解一下 SQL 的 select-form-where 语句是如何允许我们使用乘积(product)和连接(join)的。

5.2.1 SQL 中的乘积和连接

SQL 中有一个简单的方法用来把几个关系连到一个查询中: 在 FROM 子句中列出每个关系。这样, SELECT 和 WHERE 子句就可以引用 FROM 子句中列出的任何关系的属性了。

例 5.10 假设我们需要得知《星球大战》(Star Wars)这部电影的制片人(producer)的姓名。要想回答这个问题我们需要用到不断滚动的实例中的下列两个关系:

Movie (title, year, length, inColor, studioName, producerC#)

MovieExec (name, address, cert# , netWorth)

在关系 Movie 中给出了制片人的证书号(producerC#), 因此我们可以通过对 Movie

进行简单查询得到这个号码。然后我们就可以通过对关系 MovieExec 进行的第二个查询找到拥有这个证书号的人的姓名。

然而, 我们可以通过对一组关系 Movie 和 MovieExec 的一次查询来实现这两个步骤:

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert# ;
```

这个查询要求我们考虑所有的元组对——其中一个元组来自关系 Movie, 而另一个来自关系 MovieExec。元组对的条件在 WHERE 子句中得到了说明:

- 1. 来自 Movie 的元组的 title 属性值必须是“ Star Wars ”。
- 2. Movie 中元组的 producerC# 属性值必须和 MovieExec 中元组的 cert# 属性值相同, 即具有相同的证书号。也就是说, 这两个元组一定是指向同一个制片人。

每当我们找到满足这两个条件的一对元组后, 就从来自 MovieExec 的元组中取出 name 属性作为结果的一部分。如果数据是我们所要求的, 那么只有当来自 Movie 元组的对应值是“ Star Wars ”, 并且来自 MovieExec 元组的对应值是“ George Lucas ”(乔治·卢卡斯), 两个条件才能同时满足。当且仅当这个时候, 电影的名称是正确的, 同时证书号也是符合的。这样, “ George Lucas ”将是查询所生成的唯一的值。

5. 2. 2 消除属性的二义性

有时我们的查询涉及到几个关系, 并且在这些关系当中有两个或两个以上的属性有着相同的名称。这样, 我们就需要找到一种方法, 使得能够指出这些拥有同样名称的属性到底指的是哪一个。SQL 允许我们在属性的前面加上关系名和一个小圆点来解决这个问题。按照这种方法, R. A 指的是关系 R 的属性 A。

例 5. 11 下面两个关系

```
MovieStar (name, address, gender, birthdate)
MovieExec ( name, address, cert# , netWorth)
```

都有属性 name 和 address。假设我们要求找出所有具有相同地址的影星和行政长官的组合, 下面的查询将完成这项工作:

```
SELECT MovieStar. name, MovieExec. name
FROM MovieStar, MovieExec
WHERE MovieStar. address = MovieExec. address;
```

在这个查询中, 我们想要寻找一对元组, 其中一个来自 MovieStar, 而另一个来自 MovieExec, 以便使它们的 address 分量相同。WHERE 子句的作用就是要求这两个元组中的 address 属性值相同。于是, 对于每个匹配的元组对, 我们从中取出两个 name 属性, 首先从 MovieStar 的元组中提取, 然后从另一个元组中提取。结果将会是这种姓名组合的集合, 比如:

MovieStar. name	MovieExec. name
Jane Fonda	Ted Turner

元组变量和关系名

从技术上来说, SELECT 和 WHERE 子句中对属性的引用总是针对某个元组变量的。然而, 如果一个关系只在 FROM 子句中出现一次, 那么我们就可以用关系名作为它自己的元组变量。因此, 我们可以把 FROM 子句中的关系名 R 作为 R AS R 的简写。

即使在没有二义性的场合, 我们也可以加上关系名和一个小圆点。例如, 我们可以很自然地将例 5.10 中的查询写成这样:

```
SELECT MovieExec.name
FROM Movie, MovieExec
WHERE Movie.title = 'Star Wars'
      AND Movie.producerC# = MovieExec.cert#
```

换句话说, 我们可以在这个查询中所用到的属性的任何子集之前加上关系名和圆点。

5.2.3 元组变量

在查询涉及到几个不同关系的组合时, 通过在属性前面加上关系作为前缀来避免属性的二义性是很成功的。然而, 有时我们需要做的查询涉及到同一个关系中的两个或多个元组。我们可以根据需要在 FROM 子句中多次列出关系 R, 但是需要找到一种方法来引用 R 的每一次出现。SQL 允许我们为 FROM 子句中关系 R 的每一次出现定义一个别名, 用于作为引用的元组变量(tuple variable)。FROM 子句中每一次用到 R 都可以在后面加上(任选的)关键字 AS 和一个元组变量名。

在 SELECT 和 WHERE 子句中, 我们可以在属性的前面加上适当的元组变量和一个小圆点以避免属性的二义性。因此, 元组变量提供了关系 R 的另一个名字, 并在我们希望的时候把它放在关系 R 的位置上。

例 5.12 例 5.11 要求查询具有同样地址的影星和行政长官的信息, 在这里, 我们想要知道具有同样地址的两位影星的信息。查询从本质上来说是一样的, 但是现在我们必须考虑从关系 MovieStar 中选择两个元组, 而不是分别从 MovieStar 和 MovieExec 各选择一个元组。在两次用到 MovieStar 时, 我们用元组变量作为别名, 从而可以将查询写成这样:

```
SELECT Star1.name, Star2.name
FROM MovieStar AS Star1, MovieStar AS Star2
WHERE Star1.address = Star2.address
      AND Star1.name < Star2.name;
```

我们看到在 FROM 子句中两个元组变量 Star1 和 Star2 的说明, 每个元组变量都是关系 MovieStar 的别名。元组变量在 SELECT 子句中用于引用两个元组的 name 分量。在 WHERE 子句中也用这些别名来说明它们所代表的两个 MovieStar 元组在 address 分量上有相同的值。

WHERE 子句中的第二个条件, Star1.name < Star2.name, 指出第一名影星的姓名按照字母顺序应当在第二名影星的姓名之前。如果漏掉这个条件, 那么元组变量 Star1 和

Star2 就可能都指向同一个元组。我们当然会发现那些元组的地址是相同的, 于是就会生成同一影星的一对对的姓名。 第二个条件还能使我们将每一对拥有共同地址的影星只按字母顺序输出一次。如果我们用< > (不等于)作为比较运算符, 那么我们会将已结婚的一对影星生成两次, 像这样:

Star 1. name	Star 2. name
Alec Baldwin	Kim Basinger
Kim Basinger	Alec Baldwin

5. 2. 4 多关系查询的解释

要定义我们已经用过的 select -from -where 表达式的含义可有几种方法。每个用于相同的关系实例的查询按每种方法都给出相同的答案, 在这个意义上, 这几种方法是等价的。我们将依次介绍它们。

嵌套循环

到目前为止, 我们在实例中隐含使用的语义是关于元组变量的。通过回顾, 我们可以知道关系名的别名是覆盖了相应关系的所有元组的元组变量。没有别名的关系名同样也是覆盖了该关系本身的元组变量。如果有几个元组变量, 我们就可以设想一个嵌套的循环, 其中每个元组变量对应一层循环, 在每层循环中元组变量覆盖了相应关系的所有元组。对元组变量所对应的元组的每一个赋值, 我们都要判断 WHERE 子句是否为真。如果为真, 我们就生成一个元组, 它包含 SELECT 后面各项目的值(注意: 每项都会按照元组变量所对应的当前元组的赋值给出一个值)。解答查询的算法在图 5. 2 中给出。

```
假设 FROM 子句中的元组变量覆盖关系 R1, R2, ..., Rn;
FOR  关系 R1 中每个元组 t1 DO
  FOR  关系 R2 中每个元组 t2 DO
    ...
    FOR  关系 Rn 中每个元组 tn DO
      IF 当用 t1, t2, ..., tn 的值代替所有引用的属性时, WHERE
        子句得到满足
      THEN 根据 t1, t2, ..., tn 对 SELECT 子句中的属性进行求
        值并生成结果的元组值
```

图 5. 2 对简单 SQL 查询的解答

并行赋值

在另一种等价的定义中, 我们不必显式地建立覆盖元组变量的嵌套循环。相反, 我们将考虑以任意的顺序或者并行的方式由适当关系的元组给相应的元组变量所有可能的赋

如果一个人既是影星同时又是行政长官, 那么同样的问题也会出现在例 5. 11 中。通过要求这两个姓名不同, 可以简单地解决这个问题。

读者应该注意到我们在 4.2.4 节所描述的解释 Datalog 规则的第二种方法同我们给 SQL 的 select-from-where 语句的第二种解释是非常类似的。对于 Datalog, 我们说过要考虑由适当关系的元组给规则体中的关系子目标所有可能的赋值。在 SQL 中, 我们将要考虑元组给相应元组变量的所有可能的赋值。在这两种情况下, 算术子目标 (SQL 中 WHERE 子句的一部分) 限制了这些元组的赋值, 并且结果元组是通过对规则头部 (SQL 中的 SELECT 子句) 的计算得到的。

值。对每一种这样的赋值, 我们考虑 WHERE 子句是否为真。每个使 WHERE 子句为真的赋值都为最后的结果提供一个元组; 该元组是由 SELECT 子句所列出的属性按照它的赋值构造出来的。

转换到关系代数

第三种方法是把 SQL 查询和关系代数联系起来。我们从 FROM 子句中的元组变量开始并计算它们的笛卡尔积。如果两个元组变量指向同一关系, 那么该关系在乘积中会出现两次, 于是就把其属性改名以便所有的属性都有唯一的名字。同样, 不同关系中具有相同名字的属性也将改名以避免二义性。

得到乘积以后, 我们通过将 WHERE 子句以明显的方式转换为选择条件就可以把选择运算符加到乘积上了。也就是, 将 WHERE 子句中每个对属性的引用用笛卡尔积中相应的属性来代替。最后, 我们由 SELECT 子句生成用于最后投影运算的属性表。投影运算的属性由选择运算来决定; 我们用笛卡尔积的相应属性来代替 SELECT 中每个对属性的引用。

例 5.13 让我们把例 5.12 的查询转换为关系代数。首先, 在 FROM 子句中有两个元组变量, 都指向关系 MovieStar。因此, 表达式开始于

$$\text{MovieStar} \times \text{MovieStar}$$

得到的关系有 8 个属性, 前 4 个和关系 MovieStar 的第一个副本中的属性 name, address, gender 和 birthdate 相对应, 后 4 个和关系 MovieStar 的另一个副本中相同的属性相对应。我们可以通过在这些属性之前加上元组变量的别名和小圆点构成它们的名称, 如 Star1.gender, 但是为简明起见, 我们引进新的符号, 简单地称这些属性为 A_1, A_2, \dots, A_8 。于是, A_1 对应着 Star1.name, 而 A_5 对应着 Star2.name, 等等。

按照这种对属性命名的策略, 从 WHERE 子句中得到的选择条件就是 $A_2 = A_6$ 和 $A_1 < A_5$ 。投影的列是 A_1, A_5 。这样,

$$\pi_{A_1, A_5} (\sigma_{A_2 = A_6 \text{ AND } A_1 < A_5} (\text{M}(A_1, A_2, A_3, A_4)(\text{MovieStar}) \times \text{N}(A_5, A_6, A_7, A_8)(\text{MovieStar}))))$$

从技术上来说, 关系代数并不允许在 SELECT 子句中进行算术运算, 而 SQL 允许 (如例 5.4 所示)。因此, 对关系代数的投影运算符进行扩展就显得很必要了, 只是由于历史的原因, 投影是以更严格的方式定义的。

SQL 语义非直观的后果

假设 R, S 和 T 都是一元(一个分量)关系, 每个关系都只有一个属性 A, 我们希望找到那些既在 R 中出现同时又在 S 或 T(或 S 和 T 两者)中出现的元素。也就是说, 我们希望计算 $R \cap (S \cup T)$ 。我们可以认为如下的 SQL 查询可以完成这项任务:

```
SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A;
```

然而, 考虑 T 为空的情况。由于 $R.A = T.A$ 永远不能得到满足, 基于我们对 OR 运算的直观上的理解, 我们可能希望查询能准确地生成 $R \cap S$ 。但是, 无论用 5.2.4 节所述的三种等价定义中的哪一个, 我们都会发现不管 R 和 S 中有多少共同的元素, 结果都为空。如果我们用图 5.2 的嵌套循环的语义分析, 就会看到对元组变量 T 的循环只进行了 0 次, 这是因为该关系中没有元组为元组变量所覆盖。因此, FOR 循环中的 IF 语句永远不会执行, 不能生成任何结果。同样, 如果我们查看元组对元组变量的赋值, 由于没有办法把元组赋给 T, 结果不存在赋值的情况。最后, 如果我们使用笛卡尔积的方法, 首先计算 $R \times S \times T$, 由于 T 为空, 所以笛卡尔积也为空。

将以关系代数的形式再现全部的查询。

5.2.5 查询的并、交、差

有时我们希望利用关系代数中的集合运算: 并、交和差将若干个关系综合在一起。SQL 提供用于查询结果的相应运算符, 但要求这些查询生成具有相同属性集的关系。这里使用关键字 UNION, INTERSECT 和 EXCEPT 分别代表 \cup 、 \cap 和 $-$ 。UNION 这类关键字用于两个查询之间, 而查询必须用括号括起来。

例 5.14 假定我们想要身为电影行政长官并且净资产在 1 000 万美元以上的所有女影星的姓名和地址。用到下面两个关系:

```
MovieStar (name, address, gender, birthdate)
MovieExec (name, address, cert# , netWorth)
```

我们可以写出如图 5.3 所示的查询。

```
1) (SELECT name, address
2) FROM MovieStar
3) WHERE gender = 'F' )
4) INTERSECT
5) (SELECT name, address
6) FROM MovieExec
7) WHERE netWorth > 10000000);
```

图 5.3 女影星和富有的行政长官的交集

1) 到 3) 行生成了女影星的集合, 得到的关系其模式以 name 和 address 为属性。

SQL 查询的可读性

通常,人们在书写 SQL 查询语句的时候总是将每个重要的关键字如 FROM 或 WHERE 另起一行。这种风格给读者提供了查询结构的直观线索。然而,当查询或者子查询很短的时候,我们有时也会像例 5.15 那样,把它写在单独的一行。这种风格,既保持了完整查询的紧凑性,也提供了很好的可读性。

同样,5)到 7)行生成了富有的行政长官的集合,他们的净资产超过 1 000 万美元。这个查询也产生了一个关系,其模式仅有 name 和 address 属性。由于二者的模式相同,我们可以把它们相交,并用 4)行的运算符来实现。

例 5.15 依照同样的思路,可以对分别从两个关系中选出的两个人的集合求差。查询

```
(SE LECT name, address FROM MovieStar)
EXCEPT
(SELECT name, address FROM MovieExec);
```

给出了不同是电影行政长官的影星的姓名和地址,而不考虑性别和净资产。

在上面的两个例子中,为方便起见,用于相交或者求差的关系的属性都相同。但是,如果需要得到通用的属性集,我们可以像例 5.3 那样把属性改名。

例 5.16 假定我们希望得到关系 Movie 或 StarsIn 中出现的所有电影的名称和年份,而这两个关系是在不断滚动的实例中给出的:

```
Movie (title, year, length, inColor, studioName, producerC# )
StarsIn (movieT itle, movieYear, starName)
```

理想情况下,这些电影的两个集合应该是相同的,但实际上两个关系不一致也是很常见的事情;例如可能有些电影并没有列出影星,也可能有的 StarsIn 元组提到了关系 Movie 中没有找到的电影。 因此,我们可以这样写:

```
(SELECE title, year FROM Movie)
UNION
(SELECT movieT itle AS title, movieYear AS year FROM StarsIn);
```

结果将是两个关系中提到的所有电影,结果关系的属性为 title 和 year。

5.2.6 本节练习

练习 5.2.1: 使用我们不断滚动的电影实例的数据库模式

```
Movie(title, year, length, inColor, studioName, producerC# )
StarsIn(movieT itle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert# , netWorth)
```

可以设法避免这种不一致,见第 6 章。

Studio(name, address, presC#)

用 SQL 写出下列查询:

- * (a) 电影“Terms of Endearment”中的男影星都有谁?
- (b) 哪些影星出现在米高梅公司(MGM)于 1995 年制作的电影中?
- (c) 谁是米高梅制片公司的总裁?
- * ! (d) 哪些电影比《乱世佳人》(Gone With the Wind) 更长?
- ! (e) 哪些行政长官比 Merv Griffin 更富有?

练习 5.2.2: 基于练习 4.1.1 中的数据库模式

Product (maker, model, type)

PC (model, speed, ram, hd, cd, price)

Laptop (model, speed, ram, hd, screen, price)

Printer (model, color, type, price)

写出下列查询:

- * (a) 给出配置了容量至少为 1G 字节的硬盘(hd)的便携式电脑(laptop)的生产厂商(maker)及其速度(speed)。
- * (b) 找出由生产厂商 B 生产的所有产品的型号(model)和价格(price)。
- (c) 找出所有出售便携式电脑(而不出售 PC 机)的生产厂商。
- ! (d) 找出在两种或两种以上 PC 机上出现的硬盘的容量。
- ! (e) 找出拥有相同速度和内存的 PC 机的成对的型号(model)。每对只列出一;
例如, 列出了(i, j)就不要列出(j, i)。
- !! (f) 找出所有这样的生产厂商, 其产品中至少有两种不同类型的计算机(PC 机或
便携式电脑)速度最低为 133MHz。

练习 5.2.3: 基于练习 4.1.3 中的数据库模式

Classes (class, type, country, numGuns, bore, displacement)

Ships (name, class, launched)

Battles (name, date)

Outcome (ship, battle, result)

写出下列查询, 并用练习 4.1.3 中的数据对你写出的查询求值:

- (a) 找出排水量(displacement)大于 35 000 吨的舰艇。
- (b) 列出参加瓜达尔卡纳尔岛(Guadalcanal, 简称瓜岛)战役的舰艇的名称、排水量以及火炮的数量。
- (c) 列出数据库中所有提到的舰艇。(记住: 所有提到的舰艇不一定都出现在关系 Ships 中)。
- ! (d) 找出所有既拥有战列舰又拥有巡洋舰的国家。
- ! (e) 找出在一次战役中受损, 而后又在另一次战役中投入战斗的那些舰艇。
- ! (f) 找出参战的舰艇中至少有三艘属于同一国家的所有战役。

练习 5.2.4: 关系代数查询的通用格式为

$$L(c(R_1 \times R_2 \times \dots \times R_n))$$

其中, L 是任意的属性表, 而 C 是任意的条件。关系表 R_1, R_2, \dots, R_n 可以包括重复多次的同一个关系, 在这种情况下可以假设为 R_i 适当改名。说明如何表达 SQL 中这种格式的任意查询。

! 练习 5.2.5: 关系代数查询的另一种通用格式为

$$L(c(R_1 \bowtie R_2 \dots R_n))$$

这里用了练习 5.2.4 中同样的假定; 唯一不同的就是用自然连接代替了乘积。说明如何表达 SQL 中这种格式的任意查询。

5.3 子 查 询

在这一节, 我们将加深对 WHERE 子句中可能出现的表达式的理解。在此之前, 我们知道在条件中可以比较各种标量值(诸如整数、实数、字符串或日期等简单值, 或者代表这些值的表达式)。现在, 我们要扩展这种观点, 允许把整个元组甚至整个关系作为比较的内容。我们的第一步是学习如何在条件中使用子查询。子查询是对关系进行求值的表达式, 例如 select-from-where 表达式就可以是子查询。首先了解如何产生关系的值, 然后将考虑 SQL 提供的某些运算符从而使我们能够在 WHERE 子句中比较元组和关系。

5.3.1 产生标量值的子查询

表达式 select-from-where 可以生成其模式中有任意数量属性的关系, 并且关系中可以有任意数量的元组。然而, 我们往往只对单一的属性值感兴趣。此外, 有时我们可以从键码的信息推断出某个属性只生成单一的值。

如果这样, 我们可以利用由括号括起来的 select-from-where 表达式, 就像它是一个常量那样。尤其是, 它可以出现在 WHERE 子句中我们想要找到表示元组分量的常量或属性的任何地方。例如, 我们可以把这类子查询的结果同常量或属性相比较。

例 5.17 让我们回忆一下例 5.10。在例 5.10 中我们要查询《星球大战》(Star Wars) 的制片人。必须对下面两个关系进行查询:

Movie (title, year, length, inColor, studioName, producerC#)

MovieExec (name, address, cert# , netWorth)

因为只有前者有电影名称的信息, 而只有后者有制片人的姓名, 这些信息是由证书号联系起来的。证书号是制片人的唯一标识。我们拟定的查询是:

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#
```

我们可以换一种方法考虑这个查询。我们仅仅需要从关系 Movie 中得到电影《星球大战》的制片人的证书号。一旦有了证书号, 我们就可以对关系 MovieExec 进行查询, 从而找到拥有这个证书号的制片人的姓名。第一个问题, 得到证书号, 可以写成子查询; 而子查询的结果, 正如我们所预期的那样, 是单一的值, 可以用于主查询中达到和上述查询相

同的效果。图 5.4 中描述了这个查询。

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# =
4)          ( SELECT producerC#
5)          FROM Movie
6)          WHERE title = 'Star Wars'
7)          );
```

图 5.4 用嵌套子查询得到《星球大战》的制片人

图 5.4 中的 4)到 6)行是子查询。只看这个简单的查询本身,我们留意到其结果是属性为 producerC# 的一元关系,并且,我们希望在这个关系中只有一个元组。这个元组看起来形如(12345),也就是说,是由某个整数构成的单一分量,它可能是 12345,也可能是乔治·卢卡斯(George Lucas)的别的证书号。如果 4)至 6)行的子查询生成 0 个或者多于一个元组,就会出现运行时错误(run-time error)。

执行子查询以后,我们可以执行图 5.4 的 1)至 3)行,好像用值 12345 代替了整个子查询。也就是说,“主”查询将好像执行如下内容:

```
SELECT name
FROM MovieExec
WHERE cert# = 12345;
```

查询的结果将是 George Lucas。

5.3.2 涉及到关系的条件

有许多 SQL 运算符可以用于关系 R 并且生成布尔值的结果。作为典型情况,关系 R 就是 select-from-where 子查询的结果。某些运算符——IN, ALL 和 ANY——还涉及到标量值 s,在这种情况下,要求关系 R 是只有一列的关系。这里给出这些运算符的定义:

- 1. 当且仅当 R 非空时,条件 EXISTS R 为真。
- 2. 当且仅当 s 和 R 中的某一个值相等时, s IN R 为真。而且,当且仅当 s 和 R 中的任一个值都不等时, s NOT IN R 为真。这里,我们假定 R 是一元关系。我们将在 5.3.3 节讨论当 R 相应的模式有一个以上的属性并且 s 是一个元组时,对 IN 和 NOT IN 运算符的扩展。
- 3. 当且仅当 s 比一元关系 R 中的每个值都大时, s > ALL R 为真。与此类似,“>”运算符可以用其他 5 个比较运算符中的任何一个来代替而具有类似的含义。例如, s < > ALL R 和 s NOT IN R 是相同的。
- 4. 当且仅当 s 比一元关系 R 中的至少一个值大时, s > ANY R 为真。与此类似,其他 5 个比较运算符中的任何一个都可以用来代替“>”。例如, s = ANY R 和 s IN R 是相同的。

EXISTS, ALL 和 ANY 运算符都可以通过在整个表达式前面加上 NOT 来取反,就像其他布尔值的表达式一样。于是,当且仅当 R 为空时,NOT EXISTS R 为真;当且仅当

s“不大于”(译注:原文误为“不是”)R 中的最大值时, NOT s > ALL R 为真;当且仅当 s “不大于”(译注:原文误为“是”)R 中的最小值时, NOT s > ANY R 为真。我们将很快看到使用这些运算符的几个例子。

5.3.3 涉及到元组的条件

SQL 中的元组可以用括号里的标量值表来表示,如(123, 'foo') 或者(name, address, networth)。前者由常量作为分量,后者由属性作为分量。将常量和属性混合起来也是允许的。

如果元组 t 和关系 R 具有相同数量的分量,那么在 5.3.2 节所列出类型的表达式中对 t 和 R 进行比较就有意义。如 t IN R 或者是 t < > ANY R。后一个比较指的是在 R 中存在不同于 t 的元组。注意在将一个元组和关系 R 中的成员相比较时,我们必须按照 R 中假定的标准属性顺序对其分量进行比较。

例 5.18 图 5.5 中是基于以下三个关系的 SQL 查询,

```
Movie(title, year, length, inColor, studioName, producerC# )
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert# , netWorth)
```

要求寻找由 Harrison Ford(哈里森·福特)主演的所有电影的制片人。它包含一个主查询,嵌套于其中的第二级查询,以及嵌套于第二级查询中的第三级查询。

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# IN
4)      (SELECT prodecerc#
5)      FROM Movie
6)      WHERE ( title, year ) IN
7)          (SELECT movieTitle, movieYear)
8)          FROM StarsIn
9)          WHERE starName = 'Harrison Ford'
10)     )
11) );
```

图 5.5 查找 Harrison Ford 主演的电影的制片人

我们应当从内到外地分析每个嵌套查询。因此,让我们从最内层的嵌套子查询,即 7) 到 9) 行,开始。该子查询检查关系 StarsIn 中的元组,并从中找出 starName 分量是 'Harrison Ford' 的所有元组。该子查询返回电影的名称和年份。回忆一下,电影的键码是名称和年份而不仅仅是名称,因此我们需要生成具有两个属性的元组来唯一地标识一部电影。这样,我们会希望 7) 到 9) 行生成的值看起来如图 5.6 所示。

现在,让我们考虑中层子查询,4) 到 6) 行。中层子查询搜索关系 Movie 以求找到所要的元组,即元组中的名称和年份在图 5.6 所假定的关系中。对找到的每个元组,返回制片人的证书号,所以中层子查询的结果是 Harrison Ford 主演电影的制片人的证书号的集合。

Title	year
Star Wars	1997
Raiders of the Lost Ark	1981
The Fugitive	1993
...	...

图 5.6 内层子查询返回的名称-年份对

最后, 考虑从 1) 到 3) 行的“主”查询。它搜索关系 MovieExec 来寻找符合条件的元组, 这样的元组的 cert# 分量是中层子查询所返回的集合中的证书号之一。每个元组都返回一个制片人的姓名, 最终给我们的是 Harrison Ford 主演电影的制片人的集合, 这正是我们所需要的。

顺便说明一下, 图 5.5 中的嵌套查询和许多嵌套查询一样, 可以写成单一的 select - from-where 表达式, 至于在主查询或者子查询中提到的每个关系都将置于 FROM 子句中。运算符 IN 将用 WHERE 子句中的等式来代替。例如, 图 5.7 中的查询和图 5.5 中的查询是等价的。但对制片人(如 George Lucas)重复出现的处理方式存在差异, 有关内容将在 5.4.1 节讨论。

```
SELECT name
FROM MovieExec, Movie, StarsIn
WHERE cert# = producerC# AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford' ;
```

图 5.7 不采用嵌套子查询的 Ford 的制片人

5.3.4 相关子查询

最简单的子查询在整个过程中只求一次值, 并把结果用于较高一层的查询。嵌套子查询更复杂的应用场合则要求子查询多次求值, 每次对子查询中来自子查询外部元组变量的某一项赋一个值。这类子查询称为相关子查询。让我们从一个例子开始。

例 5.19 我们要找出为两部或两部以上的电影所采用的电影名。我们从对关系

Movie (title, year, length, inColor, studioName, producerC#)

中的所有元组进行外层查询开始。对于每个这样的元组, 我们在子查询中查找是否存在具有相同名称而年代更晚的电影。完整的查询如图 5.8 所示。

```
1) SELECT title
2) FROM Movie AS Old
3) WHERE year < ANY
4)      ( SELECT year
5)      FROM Movie
6)      WHERE title = Old.title
7)      );
```

图 5.8 找出重复出现的电影名

像分析其他嵌套查询那样, 让我们从 4) 至 6) 行最内层的子查询开始。如果将 6) 行的 Old. title 替换为常量字符串比如 'King Kong', 我们就很容易理解这是为找出电影“King Kong”制作的年份而进行的查询。当前的子查询稍有不同。唯一的问题是我们不知道 Old. title 的值到底是什么。然而, 当我们在 1) 至 3) 行的外查询中覆盖关系 Movie 的元组时, 每个元组都提供了 Old. title 的值。然后我们就利用 Old. title 的这个值执行 4) 至 6) 行的查询来判断从 3) 行扩展到 6) 行的 WHERE 子句的真假。

如果任何同 Old. title 具有相同名称的电影的年份比元组变量 Old 的当前值所对应的元组中的电影年份晚一些, 那么, 3) 行中的条件就为真。除了 Old 元组中的年份是制作同名电影的最后年份, 其他年份该条件都为真。结果, 1) 到 3) 行生成的电影名比同名电影少一次。制作两次的电影将列出一次, 制作三次的电影将列出两次, 依此类推。

写相关查询时, 知道名称的作用范围是相当重要的。一般而言, 如果某个元组变量对应的关系在其模式中具有某种属性, 那么子查询中的该属性就属于该子查询的 FROM 子句中的该元组变量。否则, 我们检查相邻的外层子查询, 然后检查再外层的子查询, 等等。于是, 图 5. 8 中 4) 行的 year 和 6) 行的 title 指的是这样的元组变量的属性, 该元组变量覆盖由 5) 行引入的关系 Movie 的副本(也就是, 4) 到 6) 行的子查询所访问的关系 Movie 的副本)的所有元组。

然而, 如果我们在属性前面加上某个元组变量和小圆点就可以使属性属于该元组变量。这就是我们为外层查询中的 Movie 关系引入别名 Old, 并在 6) 行引用 Old. title 的原因。注意: 如果在 2) 行和 5) 行的 FROM 子句中的两个关系不同, 我们就不需要用别名了。相反地, 在子查询中我们就可以直接引用在 2) 行提到的关系的属性。

5. 3. 5 本节练习

练习 5. 3. 1: 基于练习 4. 1. 1 中的数据库模式

Product (maker, model, type)
PC (model, speed, ram, hd, cd, price)
Laptop (model, speed, ram, hd, screen, price)
Printer (model, color, type, price)

写出下列查询。每个解答应该至少用一个子查询, 每个查询都用两种明显不同的方法(如, 用运算符 EXISTS, IN, ALL 和 ANY 的不同的集合)。

- * (a) 找出速度至少为 160MHz 的 PC 机的制造商(maker)。
- (b) 找出价格最高的打印机。
- ! (c) 找出速度低于任何 PC 机的便携式电脑(laptop)。
- ! (d) 找出具有最高价格的机器(PC 机、便携式电脑或打印机) 的型号(model)。
- ! (e) 找出具有最低价格的彩色打印机的制造商。

该例是我们必须正视关系代数中的关系和 SQL 中的关系之间重要区别的第一个场合。所有的 SQL 关系都可以有副本; 也就是说, 它们是包, 而不是集合。副本在 SQL 关系中出现的有若干种。我们将在 5. 4 节详细讨论这个问题。

!! (f) 找出在具有最小内存(ram) 容量的所有 PC 机中具有最快处理器的 PC 机制造商。

练习 5.3.2: 基于练习 4.1.3 中的数据库模式

Classes (class, type, country, numGuns, bore, displacement)

Ships (name, class, launched)

Battles (name, date)

Outcome (ship, battle, result)

写出下列查询。每个解答应该至少用一个子查询, 每个查询都用两种明显不同的方法(如, 用运算符 EXISTS, IN, ALL 和 ANY 的不同的集合)。

(a) 找出其舰艇拥有最大数量火炮的国家。

* ! (b) 找出其中至少有一艘舰艇在战役中沉没的舰艇等级。

(c) 找出具有 16 英寸口径(bore) 火炮的舰艇名称。

(d) 找出 Kongo 级舰艇参战的战役。

!! (e) 在具有相同口径火炮的舰艇中找出火炮数量最多的舰艇的名称。

! 练习 5.3.3: 不用任何子查询, 写出图 5.8 的查询。

! 练习 5.3.4: 考虑一下关系代数表达式 $\pi_L((R_1 \bowtie R_2 \dots R_n))$, 其中 L 是属于 R_1 的所有属性表。说明该表达式可以只用于子查询用 SQL 写出。更确切地说, 写出等价的 SQL 表达式, 其中 FROM 子句的元组变量表中都只有一个元组变量。

! 练习 5.3.5: 不使用交集或差集运算符写出下列查询:

* (a) 图 5.3 的交集查询。

(b) 例 5.15 的差集查询。

!! 练习 5.3.6: 我们已经注意到 SQL 的某些运算符是冗余的, 某种程度上它们总可以用其他运算符来代替。例如, 我们看到 $s \text{ IN } R$ 可以用 $s = \text{ANY } R$ 来代替。通过用不涉及到 EXISTS(不考虑可能在 R 所代表的表达式中出现)的表达式来代替任何具有 EXISTS R 和 NOT EXISTS R 形式的表达式来说明 EXISTS 和 NOT EXISTS 是冗余的。提示: 尽管很少使用, 但是, 允许在 SELECT 子句中包含常量。

5.4 副本

到现在为止, 我们所研究的关系运算大部分是每次对一个元组的运算。例外的情况是在 5.2.5 节讨论的并、交和差运算符。在本节和下一节我们将研究一些作为整体作用于关系上的运算。在这里, 我们要面对这个事实: SQL 把关系当成包(bag)而不是集合来使用, 在一个关系中一个元组可以出现多次。在 5.4.1 节我们将看到怎样强制性地使运算的结果以集合的形式出现, 在 5.4.2 节我们将看到防止副本(duplicate)的删除同样是可能的。

5.4.1 副本的删除

正如我们在 5.3.4 节提到的, SQL 中关系的概念不同于第 3 章描述的关系的抽象概念。关系作为一个集合, 任何给定元组的副本都不能超过一份。然而, 当 SQL 的查询生成一个新的关系时, SQL 系统通常并不删除副本。这样, SQL 对于一个查询的应答可能把

删除副本的代价

可能有人试图在所有的 SELECT 后面都放上 DISTINCT, 理论上这没什么坏处。实际上, 从关系中删除副本的代价是昂贵的。通常, 必须把关系排序, 以便相同的元组一个接一个地出现。只有通过这样的方法把元组分成组, 我们才能决定是否应该删除给定的元组。为删除副本而对关系排序所用的时间通常要比查询本身所用的时间长。因此, 如果我们希望查询迅速运行, 那么就要谨慎地使用删除副本的操作。

同一元组列出几次。

回顾一下 5.2.4 节, 对 SQL select-from-where 查询含义的几种等价定义之一就是从 FROM 子句中引用的关系的笛卡尔积开始的。每个这样的元组用 WHERE 子句中的条件检测, 通过检测的那些元组根据 SELECT 子句进行投影输出。该投影可能导致由不同的乘积元组产生的相同元组, 如果这样, 结果元组的每个副本将依次打印。此外, 由于 SQL 关系具有副本并不存在什么问题, 所以笛卡尔积所形成的关系也可能具有副本, 并且每个同样的副本与来自其他关系的元组成对出现, 这就导致乘积中副本的激增。

如果我们不希望产生副本, 可以在关键字 SELECT 的后面加上关键字 DISTINCT。这个关键字告诉 SQL, 对于任何元组, 只生成一份副本。这样查询结果将保证没有重复。

例 5.20 让我们再考虑一下图 5.7 中的查询, 在那里, 我们要求不使用子查询来找出 Harrison Ford(哈里森·福特)主演的电影的制片人。用图 5.7 中的查询语句, George Lucas(乔治·卢卡斯)将在输出中出现多次。如果对于每个制片人我们只想看到一次, 可以把查询中的 1) 行修改成:

1) SELECT DISTINCT name

这样, 在打印前, 将删除列出的制片人中所有重复出现的名字。

顺便说一下, 图 5.5 中的查询, 由于使用了子查询, 因而不存在副本的问题。确实, 图 5.5 中 4) 行的子查询将多次产生乔治·卢卡斯的证书号。但是, 在 1) 行的“主”查询中, 我们对关系 MovieExec 中的每个元组都检查一次。假定在这个关系中只有一个关于乔治·卢卡斯的元组, 这样的话, 只有这个元组能够满足 3) 行中的 WHERE 子句。于是, 乔治·卢卡斯这个名字只打印一次。

5.4.2 并、交、差中的副本

在 SELECT 语句中, 保留副本将作为默认的情况, 只有在使用 DISTINCT 关键字明确指出时才删除副本; 与此不同, 我们在 5.2.5 节介绍的并、交和差运算, 通常将自动地删除副本。为了防止删除副本, 我们必须在 UNION、INTERSECT 或 EXCEPT 运算符的后面加上关键字 ALL。这样, 我们将得到这些运算符的包的语义, 就像在 4.6.2 节讨论的那样。

例 5.21 再次考虑例 5.16 中的 UNION 表达式, 但现在加上关键字 ALL, 如下:

```
(SELECT title, year FROM Movie)
UNION ALL
```

(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);

现在, 结果中名称和年份的出现次数将等于这组数据分别在关系 Movie 和 StarsIn 中出现次数的和。如果一部电影在 Movie 关系中出现一次, 并且这部电影有 3 个影星列入关系 StarsIn 中(这样, 这部电影出现在 StarsIn 的 3 个不同的元组中), 那么这部电影的名称和年份将在并运算的结果中出现 4 次。

同并运算一样, 运算符 INTERSECT ALL 和 EXCEPT ALL 都是包的交和差。于是, 如果 R 和 S 都是关系, 则表达式 R INTERSECT ALL S 的结果是这样一个关系, 在该关系中, 元组 t 的出现次数是它在 R 中的出现次数和在 S 中的出现次数中的最小者。

表达式 R EXCEPT ALL S 的结果中, 元组 t 的出现次数是它在 R 中的出现次数减去它在 S 中出现次数的差, 规定这个差为正数。这些定义都是我们在 4.6.2 节关于包的讨论中出现过的。

5.4.3 本节练习

练习 5.4.1: 用 SQL 写出练习 4.1.1 中的每个查询, 保证删除副本。

练习 5.4.2: 用 SQL 写出练习 4.1.3 中的每个查询, 保证删除副本。

! 练习 5.4.3: 对于练习 5.3.1 的每个答案, 判断你的查询结果中是否存在副本。如果存在, 则重写查询, 删除副本; 如果不存在, 则写一个不用子查询的查询, 要求有同样的、不存在副本的结果。

! 练习 5.4.4: 对于练习 5.3.2 的答案, 重复练习 5.4.3 中的工作。

5.5 聚 合

把一列中的值进行聚合(aggregation)是另一类把关系看作整体的运算。所谓聚合指的是把一列中出现的一系列的值形成单一值的运算。比如说一列中各个值的和或平均值。在 SQL 中, 我们不但可以按列进行聚合, 而且可以根据某个条件, 比如别的某一行中的值, 对关系中的元组进行分组, 然后按组进行聚合。

5.5.1 聚合运算符

SQL 提供了 5 种适用于关系中的列的运算符, 这些运算符可以产生该列的汇总或聚合信息。这些运算符是:

1. SUM, 该列中各个值的和;
2. AVG, 该列中各个值的平均值;
3. MIN, 该列中的最小值;
4. MAX, 该列中的最大值;
5. COUNT, 值的个数(如果没有用 DISTINCT 明确删除副本, 则也包括副本)。

这些运算符用于标量表达式, 典型的是 SELECT 子句中的列的名称。

例 5.22 下面的查询找出所有电影的行政长官的平均净资产

```
SELECT AVG(netWorth)
```

FROM MovieExec;

请注意, 根本没有 WHERE 子句, 所以把关键字 WHERE 省略是正确的。该查询检查关系
MovieExec(name, address, cert# , netWorth)

中的 netWorth 列, 对找到的值求和, 每个元组 (即使该元组是另一个元组的副本) 取一个值, 然后用元组的个数来除总和。如果没有重复的元组, 则查询给出我们期望的净资产的平均值。如果存在重复的元组, 那么, 一个元组出现 n 次的电影行政长官的净资产将在总和中统计 n 次。

例 5.23 下面的查询

SELECT COUNT(*)
FROM MovieExec;

统计 MovieExec 关系中元组的个数。假定 name 是 MovieExec 的键码并且该关系中没有重复的元组, 则统计的元组数和统计的数据库中提到的电影行政长官的人数是相同的。

上面的星号(*) 表示整个元组。把 COUNT 这个聚合运算符用于整个元组, 是 COUNT 特有的。把其他任何聚合运算符用于超过一列是没有意义的。

如果我们希望确保不把重复的元组多次计数, 我们可以只对 name 这个属性计数, 并在它的前面使用关键字 DISTINCT, 像下面这样:

SELECT COUNT(DISTINCT name)
FROM MovieExec;

即使 name 不是键码(例如, 电影的同一个行政长官可以有两个不同的元组, 或者两个行政长官有相同的名字), 上面的查询也只对每个名字统计一次。

5.5.2 分组

通常我们需要的并不仅仅是一列的平均值或其他某种聚合。相反地, 我们需要考虑关系中根据其他一列或多列的值划分成组的元组。例如, 假设我们想要计算每个制片公司生产的电影的总长度(以分钟计), 那么我们必须根据制片公司对关系 Movie 中的元组进行分组, 并对每个分组中的 length 列求和。我们还希望把结果作成表, 把制片公司和它们的总和联系起来, 表的格式如下:

Studio	SUM(length)
Disney	12345
MGM	54321

要生成上面的表, 就要在 WHERE 子句的后面跟随一个 GROUP BY 子句。关键字 GROUP BY 的后面将跟随一个分组属性表。在最简单的情况下, FROM 子句只引用一个关系, 该关系按分组属性的值对其元组进行分组。所有在 SELECT 子句中使用的聚合运算符都只在分组内起作用。

例 5.24 在关系

Movie(title, year, length, inColor, studioName, producerC#)

中找出各制片公司制作的所有电影的总长度的问题, 可用下面的查询来表达:

```
SELECT studioName, SUM(length)
FROM Movie
GROUP BY studioName;
```

我们可以想象关系 Movie 中的元组将重新组织并分组, 于是迪斯尼制片公司的所有元组放在一起, 米高梅公司的所有元组也放在一起, 等等, 就像图 5.9 假设的那样。然后按组计算所有元组中长度分量的总和, 并且按组打印制片公司的名称和总长度。

	StudioName	
	Disney	
	Disney	
	Disney	
	MGM	
	MGM	
	o	
	o	
	o	

图 5.9 具有假设分组的关系

观察一下例 5.24 中 SELECT 子句是怎样包含两类术语的。

- 1. 聚合: 聚合运算符用于属性或者涉及到属性的表达式。如上所述, 按这些术语在每组的基础上进行计算。
- 2. 属性: 比如在这个实例中的 studioName, 同时也出现在 GROUP BY 子句中。在包含聚合运算的 SELECT 子句中, 只有在 GROUP BY 子句中出现的属性才可以在 SELECT 子句中以非聚合的形式出现。

虽然在涉及到 GROUP BY 的查询中, 分组属性和聚合通常都出现在 SELECT 子句中, 但技术上两者并不都是必须的。例如, 我们可以写出这样的查询:

```
SELECT studioName
FROM Movie
GROUP BY studioName
```

这个查询将根据制片公司的名称对关系 Movie 中的元组进行分组并按组打印制片公司的名称, 而不管和某个给定的制片公司名称相关的元组有多少。因此, 上面的查询和下面这个查询具有相同的效果:

```
SELECT DISTINCT studioName
FROM Movie
```

在对多个关系进行的查询中同样可以用 GROUP BY 子句。这样的查询将按以下步骤解释:

- 1. 计算 FROM 和 WHERE 子句涉及到的关系 R。也就是, 关系 R 是 FROM 子句中提到的关系的笛卡尔积, 而 WHERE 子句中的选择运算则用于 R。
- 2. 根据 GROUP BY 子句中的属性对关系 R 中的元组进行分组。
- 3. 把查询看作是对关系 R 进行的, 把生成的 SELECT 子句中的属性和聚合作为结果。

SQL 查询中子句的顺序

到目前为止,我们已经遇到了可以出现在 SQL“select -from -where”查询中的所有 6 种子句: SELECT, FROM, WHERE, GROUP BY, HAVING 和 ORDER BY。只有前两个是必须的。其他任何附加的子句都必须按照上面列出的顺序出现。

例 5.25 假设我们想打印一张列出每个制片人制作的电影总长度的表。我们需要从如下两个关系中获取信息。

```
Movie(title, year, length, inColor, studioName, producerC# )
MovieExec(name, address, cert# , netWorth)
```

以便首先通过使这两个关系中元组的证书号(certificate number)相等对这两个关系进行连接。通过这一步我们得到一个关系,该关系中 MovieExec 的每个元组分别和 Movie 中该制片人的所有电影的元组组成对。然后我们根据制片人的姓名对该关系的元组进行分组。最后,我们按组计算电影的总长度。这个查询见图 5. 10。

```
1) SELECT name, SUM(length)
2) FROM MovieExec, Movie
3) WHERE producerC# = cert#
4) GROUP BY name;
```

图 5.10 计算每个制片人制作的电影的总长度

5. 5. 3 HAVING 子句

假设我们并不想在例 5. 25 的表格中把所有的制片人都包括进去。我们可以在分组之前对元组加以限制,使得不需要的组为空。比如说,如果我们希望得到净资产超过一千万美元的制片人制作的电影的总长度,我们可以把图 5. 10 中的 3)行修改成:

```
WHERE producerC# = Cert# AND netWorth > 10000000
```

然而,有时我们希望在分组本身的某个聚合特性的基础上来选择分组。这样,我们就可以在 GROUP BY 子句的后面加上 HAVING 子句。后者由关键字 HAVING 及随后的分组条件组成。

例 5. 26 假设我们想要打印 1930 年以前至少制作过一部电影的制片人制作的所有电影的总长度。我们可以在图 5. 10 后面加上子句:

```
HAVING MIN(year)< 1930

SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

图 5.11 计算早期制片人制作的电影的总长度

最后的查询, 如图 5. 11 所示, 将从分组的关系中去掉所有这样的分组, 即这些分组中每个元组的 year 分量都是 1930 或者更晚。

5. 5. 4 本节练习

练习 5. 5. 1: 在练习 4. 1. 1 的数据库模式的基础上, 写出下列查询, 并用练习 4. 1. 1 中的数据计算你的查询结果。数据库模式为:

```
Product (make, model, type)
PC (model, speed, ram, hd, cd, price)
Laptop (model, speed, ram, hd, screen, price)
Printer (model, color, type, price)
```

查询:

- * (a) 找出 PC 机的平均速度。
- (b) 找出价格超过 2 500 美元的便携式电脑(laptop)的平均速度。
- (c) 找出厂商“ A ”生产的 PC 机的平均价格。
- ! (d) 找出厂商“ D ”生产的 PC 机和便携式电脑的平均价格。
- (e) 找出各种不同速度的 PC 机的平均价格。
- * ! (f) 找出各厂商生产的便携式电脑的显示器平均尺寸。
- ! (g) 找出至少生产三种不同型号的 PC 机的厂商。
- ! (h) 找出各厂商生产的 PC 机的最高价格。
- * ! (i) 找出速度超过 150MHz 的各种速度的 PC 机的平均价格。
- !! (j) 找出所有生产打印机的厂商生产的 PC 机的硬盘平均容量。

练习 5. 5. 2: 在练习 4. 1. 3 中的数据库模式的基础上, 写出下列查询, 并用练习 4. 1. 3 的数据计算你的查询结果。数据库模式为:

```
Classes (class, type, country, numGuns, bore, displacement)
Ships (name, class, launched)
Battles (name, date)
Outcomes (ship, battle, result)
```

查询:

- (a) 找出战列舰(battleship)的等级数。
- (b) 找出各等级战列舰火炮的平均数。
- ! (c) 找出战列舰火炮的平均数。请注意(b)和(c)之间的差别; 我们是否根据某一等级舰艇的数量而对该等级加权?
- ! (d) 找出各等级舰艇中的第一艘下水的年份。
- ! (e) 对于每个等级都找出在战役中沉没的舰艇数。
- !! (f) 对于至少有三艘舰艇的各个等级找出在战役中沉没的舰艇数。
- !! (g) 舰炮发射的炮弹的重量(以磅为单位)约等于口径(以英寸为单位)的立方的 1/ 2。找出每个国家舰艇上所用的炮弹的平均重量。

5.6 数据库更新

到现在为止,我们一直把重点放在标准的 SQL 查询格式上,这就是: select-from-where 语句。SQL 中还有一些其他的并不返回结果但会改变数据库状态的语句格式。在这一节,我们将把重点放在三类语句上,它们可以使我们做到:

1. 在关系中插入元组。
2. 从关系中删除某些元组。
3. 修改某些已有元组中某些分量的值。

我们把这三类操作统称为数据库更新。

5.6.1 插入

插入语句的基本格式包括:

1. 关键字 INSERT INTO;
2. 关系名 R;
3. 括在括号中的关系 R 的属性表;
4. 关键字 VALUES; 以及
5. 一个元组表达式,也就是括在括号中的一组具体的值,每个值都对应属性表 3 中的一个属性。

因此,插入的基本格式为:

```
INSERT INTO R(A1, ..., An) VALUES (v1, ..., vn);
```

通过把值 v_i 赋给对应的属性 A_i ($i = 1, 2, \dots, n$), 可以生成一个元组。如果属性表并不包括关系 R 中的所有属性, 那么在生成的元组中没有包括进来的属性将采用默认值。关于默认值我们将在 5.7.5 节讨论。最常见的默认值是空(NULL), 我们已经在 4.7.4 节进行了初步的讨论, 在 5.9 节我们将在 SQL 的范围内对它作进一步的讨论。我们可以暂时把空(NULL) 看作是, 当不知道分量的确切值时, 作为占位符使用的一个值。

例 5.27 假如我们想把 Sydney Greenstreet 加入到电影 The Maltese Falcon 的影星名单中, 我们可以用:

- 1) INSERT INTO StarsIn(movieTitle, movieYear, starName)
- 2) VALUES('The Maltese Falcon' , 1942, 'Sydney Greenstreet');

执行这个语句的效果就是, 2) 行上具有三个分量的元组将插入到关系 StarsIn 中。由于 StarsIn 的所有属性都在 1) 行列出来了, 所以就不需要再加默认的分量了。2) 行的值按照给定的顺序与 1) 行的属性相匹配, 于是 'The Maltese Falcon' 成为属性 movieTitle 对应的分量的值, 依此类推。

如果像例 5.27 那样为关系中所有的属性都提供了值, 那么我们可以省略关系名后面的属性表。这就是说, 也可以写成:

```
INSERT INTO StarsIn
VALUES( 'The Maltese Falcon', 1942, 'Sydney Greenstreet' );
```

插入的时间

图 5.12 说明了 SQL 语句语义上的微妙之处。原则上, 计算 2) 到 6) 行的查询应该在执行 1) 行的插入操作之前就完成。这样, 就不可能发生在 1) 行加入到 Studio 中的新元组影响到 4) 行中的条件。然而, 为了提高效率, 有的实现可能这样来执行这个语句, 从而在执行 2) 到 6) 行的过程中, 只要一找到新的制片公司, 就立刻改变 Studio。

在这个特例中, 插入操作是否延迟到查询完全计算完无关紧要。但是, 有些其他查询, 如果改变插入的时间, 查询的结果就可能发生变化。例如, 设想我们去掉图 5.12 中 2) 行的 DISTINCT。如果我们在执行任何插入操作之前, 先计算 2) 到 6) 行的查询, 那么新的制片公司名在 Movie 的元组中出现几次就会在查询的结果中出现几次, 从而也就在关系 Studio 中插入几次。可是, 如果我们在计算 2) 到 6) 行的查询的过程中只要一找到新的制片公司就将之立刻插入到关系 Studio 中, 那么就不会把相同的新制片公司插入两次了。相反地, 一旦把新的制片公司插入一次, 它的名称就不再满足 4) 到 6) 行的条件了, 从而也不会在 2) 到 6) 行的查询结果中第二次出现了。

然而, 如果选择这种写法, 就必须保证值的顺序与关系中属性的标准顺序相同。在 5.7 节我们将看到如何说明关系模式, 并将看到在说明时会给属性提供一个顺序。如果在 INSERT 语句中省略了属性表, 那么值就按照这个假定的顺序和属性进行匹配。如果你不能确定属性的标准顺序, 那么最好是把它们按照你喜欢的顺序列出来。

上面描述的简单的 INSERT 仅仅把一个元组加入到关系中。我们可以用子查询计算出要插入的元组集, 用来代替为每个元组赋以显式的值。该子查询代替上面描述的关键字 VALUES 和 INSERT 语句格式中的元组表达式。

例 5.28 假如我们想要在关系

Studio (name, address, presC#)

中插入所有在关系

Movie (title, year, length, inColor, studioName, producerC#)

中提到, 但并没有在 Studio 中出现的电影制片公司。由于无法确定这样一个制片公司的地址和总裁, 因此我们只能满足于用 NULL 值和要插入的 Studio 元组的 address 和 presC# 属性相对应。图 5.12 中给出了实现该插入的一种方法。

```
1) INSERT INTO Studio(name)
2)   SELECT DISTINCT studioName
3)   FROM Movie
4)   WHERE studioName NOT IN
5)         (SELECT name
6)         FROM Studio);
```

图 5.12 加入新的制片公司

和大多数嵌套 SQL 语句相似, 从内向外检查图 5.12 是最方便的。5) 行和 6) 行生成关系 Studio 中所有的制片公司名。然后, 4) 行以来自关系 Movie 的制片公司名不属于上

插入、删除和副本

关于插入和删除与 SQL 关系中所允许的副本的相互影响呈现出一些棘手的情况。首先, 插入操作将指定的元组加入到关系中, 而不管插入之前关系中是否存在该元组。这样, 如果已经存在一个与 The Maltese Falcon 和 Sydney Greenstreet 相对应的 StarsIn 元组, 那么, 例 5. 27 的插入操作将增加同一个元组的另一个副本。如果我们在插入之后执行例 5. 29 的删除操作, 由于两个元组都满足 WHERE 子句的条件, 因此就把两个元组都删掉了。我们得到一个令人吃惊的结论, 对关系 StarsIn 的插入操作和随后的删除操作得到的结果和这两个操作之前的关系不同。另外就是删除语句, 似乎在说要删掉单个元组, 而实际上删掉了不只一个元组。事实上, 在 SQL 中无论如何也没有办法从两个完全相同的元组中删掉一个。

述制片公司为条件进行检查。

现在, 我们观察 2) 到 6) 行生成的, 在关系 Movie 中而不在关系 Studio 中的制片公司名的集合。在 2) 行用 DISTINCT 保证了在这个集合中每个制片公司只出现一次, 而不管它同多少部电影相关联。最后, 1) 行把每个这样的制片公司插入到关系 Studio 中, 而对其中的属性 address 和 presC# 则赋以 NULL 值。

5. 6. 2 删除

删除语句包括:

1. 关键字 DELETE FROM,
2. 关系名, 比如说 R,
3. 关键字 WHERE, 以及
4. 一个条件。

即删除的格式为

DELETE FROM 关系名 WHERE 条件 ;

该语句执行的结果就是从给定的关系中删除满足条件 4 的所有元组。

例 5. 29 我们可以从关系

StarsIn (movieTitle, movieYear, starName)

中删除 Sydney Greenstreet 主演电影 The Maltese Falcon 这样一个事实, 通过 SQL 语句来实现:

```
DELETE FROM StarsIn
WHERE movieTitle = ' The Maltese Falcon ' AND
      MovieYear = 1942 AND
      StarName = 'Sdney Greenstreet';
```

注意: 和例 5. 27 中的插入语句不同, 我们不能简单地指定要删除的元组。相反地, 我们必须通过 WHERE 子句准确地描述出该元组。

例 5. 30 这里有另一个删除的实例。这次, 我们用不只一个元组能满足的条件从关系

MovieExec (name, address, cert# , netWorth)

中一次删除多个元组。语句

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

将删除所有净资产较低——不到 1 000 万美元的电影的行政长官。

5. 6. 3 修改

尽管我们会认为对元组的插入和删除都是对数据库的“修改”，但 SQL 中的修改操作却是对数据库的一种非常具体的更改：数据库中已经存在的一个或多个元组的某些分量有所改变。修改语句的一般格式为：

- 1. 关键字 UPDATE，
- 2. 关系名，比如为 R，
- 3. 关键字 SET，
- 4. 一系列公式，每个公式都将关系 R 的某个属性设置成表达式或常量的值，
- 5. 关键字 WHERE，以及
- 6. 条件。

即，修改的格式为

```
UPDATE R SET 新的赋值 WHERE 条件；
```

每个新的赋值(上面第 4 项)都是由属性、等号和公式组成。如果有多个赋值，中间用逗号分开。

该语句的作用首先是在关系 R 中找到所有满足条件 6 的元组；然后通过对 4 中的公式进行求值，以及对关系 R 中相应属性所对应的元组分量进行赋值，从而使每个满足条件的元组有所改变。

例 5. 31 让我们对关系

```
MovieExec(name, address, cert# , netWorth)
```

进行修改，在所有的身为制片公司总裁的电影行政长官前加上头衔“ Pres. ”。所要求的元组应满足的条件是，其证书号出现在关系 Studio 中某个元组的 presC# 分量中。我们这样描述该修改语句：

- 1) UPDATE MovieExec
- 2) SET name = 'Pres. ' @@name
- 3) WHERE cert# IN (SELECT presC# FROM Studio);
- 3) 行检测 MovieExec 中元组的证书号是否是关系 Studio 中总裁的证书号之一。

2) 行执行对选定的元组的修改操作。回顾一下运算符@@，它表示字符串的拼接，结果根据 2) 行等号右边的表达式将字符 Pres. 和一个空格放到该元组 name 分量的原有值之前。新的字符串成为该元组 name 分量的值；结果头衔 'Pres.' 就附加在 name 的原有值之前。

5. 6. 4 本节练习

练习 5. 6. 1: 基于练习 4. 1. 1 中的数据库模式

Product (make, model, type)
 PC (model, speed, ram, hd, cd, price)
 Laptop (model, speed, ram, hd, screen, price)
 Printer (model, color, type, price)

写出下列数据库更新操作, 描述对练习 4. 1. 1 中的数据进行更新的效果。

- (a) 用两个 INSERT 语句将下述事实存入数据库: 生产厂商 C 制造的型号为 1100 的 PC 机, 速度 240M, 内存 32M, 硬盘 2. 5G, 12 倍速光驱, 售价 2 499 美元。
- ! (b) 插入这些事实: 对于每一种 PC 机, 都有一种具有相同速度、内存和硬盘的便携式电脑, 而屏幕为 11 英寸, 型号大于 1100, 价格超过 500 美元。
- (c) 删掉所有硬盘容量不到 2G 字节的 PC 机。
- (d) 删掉由不制造打印机的厂商所制造的所有便携式电脑。
- (e) 厂商 A 收购了厂商 B, 将所有由 B 制造的产品改为由 A 制造。
- (f) 对每种 PC 机, 将内存的容量加倍, 硬盘的容量增加 1G 字节。(记住: 一些属性可以通过 UPDATE 语句进行更改。)
- ! (g) 对由生产厂商 E 制造的所有便携式电脑, 将屏幕尺寸增加 1 英寸, 并且将价格降低 100 美元。

练习 5. 6. 2: 基于练习 4. 1. 3 中的数据库模式

Classes (class, type, country, numGuns, bore, displacement)
 Ships (name, class, launched)
 Battles (name, date)
 Outcomes (ship, battle, result)

写出下列数据库更新操作。描述对练习 4. 1. 3 中数据进行更新操作的效果。

- * (a) 两艘 Nelson 级的英国战列舰——Nelson 号和 Rodney 号——均于 1927 年下水, 拥有 9 门 16 英寸的火炮以及 34 000 吨的排水量。将这些事实插入到数据库中。
- (b) 三艘意大利 Vittorio Veneto 级战列舰中的两艘——Vittorio Veneto 号和 Italia 号——均于 1940 年下水; 该等级的第三艘战列舰, Roma 号, 于 1942 年下水。它们都有 9 门 15 英寸的火炮以及 41 000 吨的排水量。将这些事实插入到数据库中。
- * (c) 从关系 Ships 中删除所有在战役中沉没的舰艇。
- * (d) 修改关系 Classes, 从而用厘米来计量火炮口径(1 英寸= 2. 5 厘米), 用公吨来计量排水量(1 公吨= 1. 1 吨)。
- (e) 删掉所有舰艇数量少于 3 艘的等级。

5. 7 用 SQL 定义关系模式

在本节我们开始讨论数据定义(data definition), 即 SQL 中涉及到描述数据库中信息结构的部分。相反, 以前讨论的 SQL 的诸方面——查询和更新——通常称为数据操作

(data manipulation)。

本节的主题是关系模式的说明。我们将看到如何描述新的关系,即 SQL 中所谓的“表”(table)。我们能够描述的方面包括属性名,属性的数据类型,以及诸如“键码”之类的某些有限定的约束类型。5.8 节涉及到“视图”(view),视图是虚拟关系而不是数据库中实际存储的真正关系,至于一些关于关系约束的更复杂的问题则留到第 6 章。

5.7.1 数据类型

首先,我们介绍 SQL 系统支持的主要的数据类型。所有的属性都必须具有数据类型。

1. 固定或可变长度的字符串。类型 CHAR(n) 表示固定长度为 n 个字符的字符串。这意味着,如果属性类型为 CHAR(n),则任何元组中对应于该属性的分量都是 n 个字符的串。VARCHAR(n) 表示顶多为 n 个字符的串。该类属性所对应的分量可以是 0 到 n 个字符的串。SQL 允许字符串类型的值之间进行合理的强制转换。如果字符串成为更长的固定长度字符串类型的分量值,通常就用尾随的空格来填充。例如,字符串 'foo',如果成为类型为 CHAR(5) 的属性所对应的分量值,则采用的值为 'foo' (在第 2 个 o 后面有两个空格)。如果该分量的值同另一个字符串相比较(见例 5.1.3),则填充的空格就会忽略不计。

2. 固定或可变长度的位串。同固定和可变长度字符串相似,但是其值为位串而不是字符串。类型 BIT(n) 表示长度为 n 的位串,而 BIT VARYING(n) 则表示长度顶多为 n 的位串。

3. 类型 INT 或 INTEGER(它们是同义词)表示典型的整型值。类型 SHORTINT 也表示整型,但是根据具体的实现,允许的位数可能会少一些(这同 C 中的类型 int 和 short int 相似)。

4. 浮点数可以通过多种方式来表示。我们可以用类型 FLOAT 或 REAL(它们是同义词)来表示典型的浮点数。可以通过类型 DOUBLE PRECISION 得到高精度浮点数;这些类型之间的区别也同 C 中的情形相似。SQL 还有固定小数点的实数类型,如,DECIMAL(n, d) 允许包含 n 个十进制数字,小数点位于从右数第 d 位。例如,0123.45 就是类型 DECIMAL(6, 2) 的一个可能值。

5. 日期和时间可以通过数据类型 DATE 和 TIME 来表示。请回忆我们在 5.1.4 节对日期和时间值的讨论。这些值本质上是特殊格式的字符串。事实上,我们可以将日期和时间强制转换为字符串类型,如果字符串是有意义的日期和时间,也可以反过来进行强制转换。

5.7.2 表的简单说明

关系模式说明的最简单的格式包括关键字 CREATE TABLE 和随后的关系名以及括号内的一系列属性名及其类型。

例 5.32 我们实例中的 MovieStar 关系的模式,在 3.9 节非正式地描述过,它可以通过图 5.13 中的语句建成 SQL 的表。我们将前两个属性, name 和 address, 都说明为字符串。但是对于姓名,我们决定使用 30 个字符的固定长度字符串,如果需要就在姓名后面填补空格,如果姓名太长则将其截至 30 个字符。相反,我们将地址说明为最多具有

255 个字符的可变长度字符串。还不清楚这两种选择是否是可能实现的最好选择,但是,我们可以用它们来说明两种字符串数据类型。

```
1) CREATE TABLE MovieStar (  
2)     Name CHAR( 30),  
3)     address VARCHAR(255),  
4)     gender CHAR( 1),  
5)     birthdate DATE  
);
```

图 5.13 说明关系 MovieStar 的关系模式

属性 gender 具有单个字母(M 或 F)的值。这样,我们能够可靠地使用单个字符作为该属性的类型。最后,属性 birthdate 很自然地采用数据类型 DATE。如果该类型无法从不遵循 SQL2 标准的系统中得到,那么既然所有的 DATE 值实际上都是 10 个字符的字符串——8 个数字和 2 个连字符,我们就可以用 CHAR(10)来代替。

5.7.3 删除表

关系作为长期存在的数据库的一部分,往往只建立一次,然后与元组共存若干年。关系可能通过把与 SQL 表不同的某种格式的现有数据转换为插入命令序列来进行成批写入;为此,数据库管理系统(DBMS)或许会提供成批写入工具。另外,关系可能随时间的推移而扩展,日复一日地积累元组。例如,关系 Movie 最初可能从早期的某个数据库中成批写入,然后每当一部新的电影发行时通过 INSERT 操作不断更新。

然而,有时我们希望从数据库模式中删掉某个关系。由于客观条件的变化,可能不再需要维持表中的信息。或者关系就是暂时的,也许是在某个复杂的查询中无法用单个 SQL 语句来表达时产生的中间结果。如果是这种情形,我们可以用下面的 SQL 语句来撤消关系 R

```
DROP TABLE R;
```

5.7.4 更改关系模式

关系作为长期存在的数据库的一部分,可以撤消,而更常见的是,不得不更改现有的关系模式。这种更改可以通过以关键字 ALTER TABLE 和关系名开始的语句来完成。有几个选项,其中最重要的是:

- 1. ADD 加上列名及其数据类型。
- 2. DROP 加上列名。

例 5.33 例如,我们通过增加属性 phone 来更改关系 MovieStar

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

作为结果,模式 MovieStar 现在具有 5 个属性,图 5.13 中提到的 4 个再加上属性

数字 255 并不是什么典型地址的某种神秘表示法的答数。单个字节能够存储 0 到 255 之间的整数,所以能用单个字节来表示最多为 255 个字符的可变长度字符串的字符数,另外再加上存储字符串本身的字节。然而,商业系统通常能支持更长的可变长度字符串。

phone——固定长度为 16 个字节的串。在实际的关系中, 元组都会有 phone 对应的分量, 但我们知道现在并没有电话号码放在那里。因此, 所有这些分量值都将置为 NULL。在 5.7.5 节, 我们将看到如何选用其他的“默认”值来代替未知值 NULL。

作为另一例子, 我们可以通过

```
ALTER TABLE MovieStar DROP birthdate;
```

来撤消属性 birthdate。

5.7.5 默认值

当我们建立或修改元组时, 有时并不能为所有的分量赋值。例如, 上面提到当我们在关系模式中增加一列时, 现有的元组没有已知值, 有人提出用特定值 NULL 来代替“实际”值。或者, 在例 5.28 中我们提出可以向关系 Studio 中插入只知道制片公司名而不知道地址和总裁证书号的新元组。还有, 需要用“不知道”的某个值来代替后两个属性的实际值。

为了说明这些问题, SQL 提供了 NULL 值。除了某些情况下不允许用 NULL 值(见 6.2 节)以外, 该值成为没有给出具体值的所有分量的值。但是, 有时我们宁愿选择其他的默认值, 如果不知道其他值就用该值作为分量值。

一般而言, 我们可以在说明属性及其数据类型的地方加上关键字 DEFAULT 和一个适当的值。这个值或者为 NULL, 或者为某个常量。当然系统也会提供某些其他值, 比如也可以选用当前时间。

例 5.34 让我们考虑例 5.32, 我们可能希望用字符 '?' 作为未知的 gender 的默认值, 也可能希望用可能存在的最早的日期, DATE '0000-00-00' 作为未知 birthdate 的默认值。我们可以将图 5.13 中的 4) 行和 5) 行替换为:

```
4) gender CHAR(1) DEFAULT '?',
5) birthdate DATE DEFAULT DATE '0000-00-00'
```

作为另一个实例, 当我们在例 5.33 中增加新属性 phone 时, 将这个新属性的默认值设为 'unlisted', 则更改语句如同:

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

5.7.6 域

到目前为止, 我们已经为所有的属性定义了数据类型。另一种选择是首先定义域 (domain), 给数据类型起一个新的名称。诸如默认值或将在 6.3.3 节讨论的对值的约束等其他信息也可以通过域来说明。然后, 在我们说明属性的时候, 就可在属性名后面用域名取代数据类型。几个属性可以使用同一个域, 以有效的方式把属性联系起来。例如, 如果我们对两个属性使用同一个域, 我们就知道一个属性的值总可以成为另一个属性的值。

域定义的格式是关键字 CREATE DOMAIN、域名、关键字 AS 以及数据类型。其域说明如下:

```
CREATE DOMAIN < 域名> AS < 数据类型描述>;
```

在这些内容之后可以加上默认值说明, 也可以加上将在 6.3.3 节讨论的对域的其他

约束。

例 5. 35 我们为电影的名称定义一个域 MovieDomain。该域可以用于关系 Movie 的 title 属性中, 也可以用于关系 StarsIn 的 movieTitle 属性中。可以这样定义该域:

```
CREATE DOMAIN MovieDomain AS VARCHAR(50) DEFAULT 'unkown';
```

于是, MovieDomain 域的值为最多 50 个字符的可变长度字符串, 同时, 对于未知的名称其默认值为 'unknown' 。

当我们说明关系 Movie 的模式时, 可以用:

```
title MovieDomain
```

说明属性 title, 而不用等价的

```
title VARCHAR(50) DEFAULT 'unknown'
```

同样, 我们可以用

```
movieTitle MovieDomain
```

说明关系 StarsIn 中的属性 movieTitle。

可以用如下的语句

```
ALTER DOMAIN MovieDomain SET DEFAULT 'no such title';
```

更改域的默认值:

该更改语句用字符串 'no such title' 代替我们刚刚在例 5. 35 中说明的域 MovieDomain 的默认值 'unknown' 。还可以采用其他的更改选项, 例如将在第 6 章讨论的对域的约束。

我们可以用如下的撤消语句

```
DROP DOMAIN MovieDomain;
```

来删除域的定义。

结果是, 要说明属性时不能再用该域了。但是, 已经用该域定义的属性将如同撤消该域之前一样具有相同的类型和默认值。

5. 7. 7 索引

关系中属性 A 的索引(index)是一种数据结构, 它能更有效地查找对应于属性 A 具有固定值的元组。索引通常有助于将属性 A 和常量进行比较(例如 $A = 3$, 甚至于 $A > 3$) 的查询。当关系非常大时, 为了找出与给定条件匹配的元组(或许很少), 扫描关系中所有元组的开销将变得很大。例如, 考虑我们在例 5. 1 考察的第一个查询:

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

可能会有 10 000 个电影元组, 其中只有 200 个制作于 1990 年。

实现该查询的直截了当的方法是取出所有 10 000 个元组, 并对每个元组测试 WHERE 子句中的条件。如果我们用某种方法只取出制作于 1990 年的 200 个元组, 并对其中每个进行测试看其制片公司是否是 Disney, 将有效得多。如果我们能够直接得到满足 WHERE 子句中的两个条件——制片公司为 Disney 同时制作年份为 1990 年——的大

约 10 个左右元组, 甚至会更有效。但这已超过从通常使用的数据结构所能期望得到的了。

尽管建立索引并不是包括 SQL2 在内的任何 SQL 标准的一部分, 但大多数商业系统有办法向数据库设计者表明, 系统能够为某个关系中的某个属性建立索引。下面的句法是有代表性的。假如我们想要得到关系 Movie 中的属性 year 的索引, 就可以如下表达:

```
CREATE INDEX YearIndex ON Movie(year);
```

结果是建立了关系 Movie 的 year 属性的索引, 索引名为 YearIndex。从此, 对于指定了年份的 SQL 查询, SQL 查询处理程序将采用只检查关系 Movie 中具有指定年份的元组的方法, 这样执行的结果是减少了回答查询所需要的时间。

通常, 还可以利用多属性索引。该索引将取出若干属性的值, 并且能有效地查找对于这些属性具有给定值的元组。表面上看多属性索引没有单属性索引有用, 因为当前者并没有对每个属性都赋值时就要使用后者。但是, 当可以用多属性索引时, 就会更有效地找到所要求的元组。

例 5.36 由于属性 title 和 year 构成关系 Movie 的键码, 通常我们可能希望这两个属性值都指定或者都不指定。下面是关于这两个属性的索引的典型说明:

```
CREATE INDEX KeyIndex ON Movie( title, year );
```

由于(title, year)是键码, 那么每当给出一组名称和年份, 我们确信索引只能找到一个元组, 这也正是我们所要求的元组。相反, 如果查询既指定了名称, 又指定了年份, 但只能利用 YearIndex 索引, 那么系统顶多能返回该年份的所有电影, 然后从中对给定的名称进行检查。

如果我们有单独的 title 索引, 情形就要比只有 year 索引好。原因就是, 对于给定的年份会有许多电影, 而对于给定的名称通常只有很少的电影。在此特例中, 对于给定的名称, 返回所有的元组然后从中检查给定的年份, 比使用既包括 title 又包括 year 的多属性索引花费的时间稍多一点。

如果我们想要撤消索引, 只需简单地把索引名用在如下的语句中:

```
DROP INDEX YearIndex;
```

数据库设计者需要折衷考虑索引的选择:

- 某一属性索引的存在能够大大加快该属性值已然确定的查询。
- 另一方面, 所有基于某个关系的某个属性的索引都使得对该关系的插入、删除和修改操作变得更加复杂和费时。

索引的选择是数据库设计中最困难的部分之一, 因为需要估计出数据库中查询和其他操作有代表性的混合方式。如果对关系的查询比更新频繁得多, 那么对最频繁指定的属性建立索引将很有意义。如果更新是主要的行为, 那么我们对建立索引就应非常谨慎。即使如此, 对频繁使用的属性建立索引也会提高效率。事实上, 由于某些更新命令涉及到对数据库的查询(如带有 select-from-where 子查询的 INSERT 操作, 或带有条件的 DELETE 操作), 人们必须很仔细地对更新和查询操作的相关频率进行估算。

5.7.8 本节练习

练习 5.7.1: 在本节, 我们只对不断滚动的实例中五个关系中的 MovieStar 关系给出了正

式的说明。给出其余四个关系的适当的说明:

```
Movie (title, year, length, inColor, studioName, producerC# )
StarsIn (movieTitle, movieYear, starName)
MovieExec (name, address, cert# , netWorth)
Studio (name, address, presC# )
```

练习 5.7.2: 下面,我们又一次重复了练习 4.1.1 中非正式的数据库模式:

```
Product (make, model, type)
PC (model, speed, ram, hd, cd, price)
Laptop (model, speed, ram, hd, screen, price)
Printer (model, color, type, price)
```

写出下列说明:

- (a) 与关系 Product 相对应的模式。
- (b) 与关系 PC 相对应的模式。
- * (c) 与关系 Laptop 相对应的模式。
- (d) 与关系 Printer 相对应的模式。
- (e) 对域 ModelType 的适当定义,其值为型号(model)。给出在从(a)到(d)的模式中如何使用该域。
- * (f) 对(c)中你写的 Laptop 模式进行更改,增加属性 cd。如果便携式电脑没有配置光驱,则令该属性对应的默认值为 'none'。
- (g) 更改(d)中的 Printer 模式,删掉其属性 color。

练习 5.7.3: 这里给出了练习 4.1.3 中的非正式的模式:

```
Classes (class, type, country, numGuns, bore, displacement)
Ships (name, class, launched)
Battles (name, date)
Outcomes (ship, battle, result)
```

写出下列说明:

- (a) 与关系 Classes 相对应的模式。
- (b) 与关系 Ships 相对应的模式。
- (c) 与关系 Battles 相对应的模式。
- (d) 与关系 Outcomes 相对应的模式。
- (e) 对域 ShipNames 的适当定义,使之能够用于舰艇名称和等级名称。利用该域更改(a)、(b)和(c)中的模式。
- (f) 对(b)中的关系 Ships 进行更改,使之包含属性 yard,用于给出建造舰艇的造船厂。
- (g) 更改(a)中的关系 Classes,删掉属性 bore。

! 练习 5.7.4: 解释下列两条语句之间的区别。

```
DROP R;
DELETE FROM R;
```

关系(relation)、表(table)和视图(view)

SQL 程序员倾向于用术语“表”来代替“关系”。原因是对实际存储的关系(即“表”)和虚拟的关系(即“视图”)二者作出明确的区别是非常重要的。既然我们知道了表和视图之间的区别,那么我们应该仅在或者使用表或者使用视图的情况下才使用“关系”。当我们想要强调是存储的关系而不是视图时,有时会使用术语“基本关系”(base relation)或“基本表”(base table)。

还有第三种关系,既不是视图也不是永久存储的表。这些关系是临时结果,可能是为某个子查询而构造出来的。以后把临时结果也称为“关系”。

5.8 视图的定义

通过 CREATE TABLE 语句定义的关系实际存在于数据库中。也就是说,SQL 系统把表(table)存储在某种物理的体系结构中。可以认为表能够无限期地存在并且不发生变化,除非明确表示用 INSERT 或 5.6 节讨论的其他更新语句之一对其进行改变。在这个意义上来说表是持久不变的。

还有另一类并不实际存在的 SQL 关系称为“视图”(view)。相反,它们通过非常类似于查询的表达式来定义。反过来,又可以对视图进行查询,就如同它们实际存在一样,并且在某些情况下,甚至可以更新视图。

5.8.1 视图的说明

定义视图的最简单格式为:

1. 关键字 CREATE VIEW,
2. 视图名,
3. 关键字 AS, 以及
4. 查询 Q。该查询就是视图的定义。每当我们对视图进行查询时,SQL 的作用就好像当时在执行 Q,并对 Q 所生成的关系进行查询。

即,简单的视图定义格式如下:

```
CREATE VIEW 视图名 AS 视图定义 ;
```

例 5.37 假定我们想要把关系

Movie (title, year, length, inColor, studioName, producerC#)

的一部分,确切地说,由 Paramount(派拉蒙)制片公司制作的电影的名称和年份作为一个视图,我们可以通过以下语句来定义视图。

- 1) CREATE VIEW ParamountMovie AS
- 2) SELECT title, year
- 3) FROM Movie
- 4) WHERE studioName = 'Paramount';

首先,就像我们在 1) 行看到的,视图名为 ParamountMovie,视图的属性在 2) 行列出,名称为 title 和 year。视图的定义在 2) 到 4) 行。

5.8.2 视图的查询

关系 ParamountMovie 并不包含通常意义下的元组。相反,如果我们查询 ParamountMovie,将会从基本表 Movie 中得到适当的元组,所以查询能够得到回答。即使我们并没有表现出对 ParamountMovie 有什么改变,只是因为其基本表可能在两次查询期间发生变化,结果,我们对 ParamountMovie 进行两次同样的查询可能得到不同的回答。

例 5.38 我们可以对视图 ParamountMovie 进行查询,就如同它是一个存储的表,例如:

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

用视图 ParamountMovie 的定义将上面的查询转换为只访问基本表 Movie 的新查询。我们将在 5.8.5 节阐明如何将基于视图的查询转换为基于基本表的查询。然而,在这种情况下,不难推断出该例中视图查询的意义。我们观察到 ParamountMovie 同 Movie 的区别只在于以下两点:

- (1) ParamountMovie 只生成属性 title 和 year。
- (2) 条件 studioName = 'Paramount' 是关于 ParamountMovie 的任何 WHERE 子句的一部分。

由于我们的查询只想生成 title, (1) 不存在问题。对于(2),我们只需要将条件 studioName = 'Paramount' 引入到查询的 WHERE 子句中。这样,我们就可以在 FROM 子句中用 Movie 来代替 ParamountMovie,从而保证把查询的含义保存下来。于是,查询

```
SELECT title
FROM Movie
WHERE studioName = 'Paramount' AND year = 1979;
```

就成为对基本表 Movie 的查询,与我们最初对视图 ParamountMovie 的查询具有相同的效果。注意,做这种转换是 SQL 系统的工作。我们说明该推导过程只是想指出对视图的查询的含义。

例 5.39 也可以写出既涉及到视图又涉及到基本表的查询。例如

```
SELECT DISTINCT starName
FROM ParamountMovie, StarsIn
WHERE title = movieTitle AND year = movieYear;
```

该查询寻找由派拉蒙制作的电影中所有影星的姓名。注意,即使影星们出现在多部派拉蒙制片公司制作的电影中,由于使用了 DISTINCT 就保证影星只列出一次。

例 5.40 让我们来考虑用于定义视图的更复杂的查询。我们的目标是得到具有电影名及其制片人名的关系 MovieProd。定义视图的查询既涉及到关系

```
Movie( title, year, length, inColor, studioName, producerC# )
```

从中得到制片人的证书号,又涉及到关系

```
MovieExec( name, address, cert# , netWorth )
```

在此我们将证书号和姓名联系起来。我们可以这样写:

```
1) CREATE VIEW MovieProd AS
2)     SELECT title, name
3)     FROM Movie, MovieExec
4)     WHERE producerC# = cert# ;
```

我们可以对该视图进行查询,就如同它是一个存储的关系。例如,要找到“Gone With the Wind”《乱世佳人》的制片人,可询问

```
SELECT name
FROM MovieProd
WHERE title = 'Gone With the Wind';
```

就像对任何视图一样,把该查询看作只是针对基本表的等价查询,如同

```
SELECT name
FROM Movie, MovieExec
WHERE producerC# = cert# AND title = 'Gone With the Wind';
```

5.8.3 属性改名

有时,我们会更愿意给出自己选择的视图属性名,而不愿使用由定义视图的查询产生的属性名。我们可在 CREATE VIEW 语句的视图名之后用括号内的属性表来指定视图的属性。例如,我们可以这样重新写出例 5.40 的视图定义:

```
CREATE VIEW MovieProd ( movieTitle, prodName) AS
SELECE title, name
FROM Movie, MovieExec
WHERE producerC# = cert# ;
```

视图是一样的,但各列的标题用属性 movieTitle 和 prodName 来代替 title 和 name。

5.8.4 视图的更新

在有限的情形下,可以对视图进行插入、删除或者修改操作。首先,由于视图并不像基本表(存储的关系)那样实际存在,这种概念毫无意义。比如说,将新的元组插入到视图中意味着什么呢?该元组去向如何?数据库系统如何记住该元组应该在视图中呢?

对于许多视图,回答仅仅是“你不能这样操作”。然而,对于足够简单的视图,称为“可更新”视图,就可能将对视图的更新转换为等价的对基本表的更新,于是可以用对基本表的操作来代替更新操作。SQL2 对于何时允许对视图进行更新提供了正式的定义。SQL2 的规则很复杂,但是粗略地讲,它允许更新从单个关系 R 中选出某些属性(用 SELECT 而不用 SELECT DISTINCT)定义的视图(而 R 本身也可以为可更新的视图)。两个重要的技术要点为:

- WHERE 子句绝对不能在子查询中涉及到 R。

- SELECT 子句中的属性必须包括足够的属性,以便对于每个要插入到视图中的元组,我们都可以将其他的属性赋以 NULL 值或其他适当的默认值,就能得到基本关系的一个元组,而该元组将生成要插入到视图中的元组。

例 5. 41 假定我们要向例 5. 37 的视图 ParamountMovie 中插入如下的元组:

```
INSERT INTO ParamountMovie
VALUES('Star Trek', 1979);
```

由于视图 ParamountMovie 只对单个基本表

```
Movie( title, year, length, inColor, studioName, producerC# )
```

中的某些元组的某些分量进行查询,因此基本上满足 SQL2 可更新的条件。

唯一的问题是,由于关系 Movie 中的属性 studioName 并不是视图的属性,那么插入到 Movie 中的元组将用 NULL 而不是 'Paramount' 作为 studioName 的值。

因此,为使视图 ParamountMovie 可更新,我们应该在视图的 SELECT 子句中增加属性 studioName,即使对我们来说制片公司的名称显然为派拉蒙。视图 ParamountMovie 修订后的定义为:

```
1) CREATE VIEW ParamountMovie AS
2)     SELECT studioName, title, year
3)     FROM Movie
4)     WHERE studioName = 'Paramount';
```

于是,我们可以将对可更新视图 ParamountMovie 进行插入的操作写为:

```
INSERT INTO ParamountMovie
VALUES( 'Paramount', 'Star Trek', 1979 );
```

为了实现该插入,当视图定义用于关系 Movie 时,我们构造出一个 Movie 元组使之生成插入到视图中的元组。对于上面特定的插入操作,其 studioName 分量为 'Paramount',title 分量为 'Star Trek' ,而 year 分量为 1979。

在插入的 Movie 元组中必须有视图没有出现的其他三个属性——length, inColor 和 producerC# 。可是,我们无法推断出它们的值。结果,新的 Movie 元组必须为这三个属性中的每个相应分量赋以适当的默认值: 或者为 NULL 或者为某个其他的对属性或相应域所说明的默认值。例如,如果对属性 length 说明的默认值为 0,而其他两个用 NULL 作为其默认值,那么结果插入的 Movie 元组就会是:

title	Year	Length	inColor	studioName	producerC#
'Star Trek'	1979	0	NULL	'Paramount'	NULL

我们还可以对可更新视图进行删除操作。删除操作,和插入操作一样,将传递给底层的关系 R,从而导致删除和视图中要删除的元组相对应的 R 中的所有元组。

例 5. 42 假设我们希望从可更新视图 ParamountMovie 中删除所有名称中含有“Trek ”的电影。我们可以写出删除语句:

```
DELETE FROM ParamountMovie
WHERE title LIKE '% Trek % ';
```

为何有些视图不可更新

请考虑例 5.40 中的视图 MovieProd, 它将电影名称和制片人的姓名联系到一起。由于在 FROM 子句中有两个关系: Movie 和 MovieExec, 根据 SQL2 的定义, 该视图不可更新。假设我们试图插入如下一个元组:

```
('Greatest Show on Earth', 'Cecil B. DeMille')
```

我们就不得不将元组既插入到 Movie 中, 又插入到 MovieExec 中。我们可以对 length 或 address 这类属性使用默认值, 但是对于都表示 DeMille 未知证书号的等价属性 producerC# 和 cert# 怎么办呢? 我们可能对这两个属性都使用 NULL 值。但是, 当连接含有 NULL 的两个关系时, SQL 无法识别出两个 NULL 值是否相等(见 5.9.1 节)。因此, 在视图 MovieProd 中, 'Greatest Show on Earth' 就无法同 'Cecil B. DeMille' 联系起来, 这表明我们的插入操作不能正确执行。

该删除语句将转换为等价的对基本表 Movie 的删除; 唯一的区别就是把视图 ParamountMovie 定义中的条件加到 WHERE 子句的条件中。

```
DELETE FROM Movie
```

```
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

这就是最终的删除语句。

与此类似, 对可更新视图的修改也将传递给底层的关系。因此, 修改视图的结果就是对导致修改视图元组的底层关系的所有元组进行修改。

例 5.43 视图的修改

```
UPDATE ParamountMovie
```

```
SET year = 1979
```

```
WHERE title = 'Star Trek the Movie';
```

将转换为对基本表的修改

```
UPDATE Movie
```

```
SET year = 1979
```

```
WHERE title = 'Star Trek the Movie' AND StudioName = 'Paramount';
```

最后一种对视图的更新就是完全撤消视图。无论视图是否可更新, 这种更新都可以进行。典型的 DROP 语句为

```
DROP VIEW ParamountMovie;
```

注意该语句撤消了视图的定义, 因此我们再进行查询或发布更新命令时都不能涉及到该视图。然而, 撤消视图并不影响底层的 Movie 关系中的任何元组。相反,

```
DROP TABLE Movie
```

不仅会使得表 Movie 消失, 而且也会使视图 ParamountMovie 不能再用, 这是由于对该视图的查询会间接地引用已经不存在的关系 Movie。

5. 8. 5 对涉及到视图的查询的解释

虽然如何由 SQL 系统实现对视图的查询超出了本书的范围,但是我们可以通过下面的解释过程来理解视图查询的含义。尽管通过增加反映 SQL 的附加特征的运算符能处理完整的 SQL,如分组和聚合,我们还是应该将自己限制在能用关系代数表示的查询和视图的范围之内。

图 5. 14 说明了基本的想法。图中,查询 Q 由关系代数中的表达树来表示。该表达树用表示视图的一些关系作为叶子。我们假定有两片这样的叶子,视图 V 和 W。为了从基本表的观点解释 Q,我们就要找出视图 V 和 W 的定义。这些定义也是由关系代数中的表达树来表示的。

要想形成对基本表的查询,我们将 Q 树中作为视图的每片叶子用视图定义树的副本的根来代替。因此,在图 5. 14 中我们描绘出标明 V 和 W 的叶子由这些视图的定义来代替。结果生成的树成为与最初对视图的查询等价的对基本表的查询。

例 5. 44 让我们考虑例 5. 38 中的视图的定义和查询。回顾一下,视图 ParamountMovie 的定义为:

```
1) CREATE VIEW ParamountMovie AS
2)     SELECT title, year
3)     FROM Movie
4)     WHERE sutdioName = 'Paramount';
```

图 5. 15 给出了定义该视图的查询表达树。

图 5. 14 用视图的定义代替视图的引用

图 5. 15 视图 ParamountMovie 的表达树

例 5. 38 的查询为:

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

查找派拉蒙制片公司 1979 年制作的电影。该查询可用图 5. 16 中给出的表达树来描述。注意该树的一片叶子表示视图 ParamountMovie。

因此,我们可以用图 5. 15 中的树代替图 5. 16 中的叶子 ParamountMovie 来解释查询。结果树在图 5. 17 中给出。

用图 5. 17 中的树对查询进行解释是可以接受的。但是,这种解释方式过于复杂。SQL 系统会对这种树进行转换,目的是使其看起来像我们在例 5. 38 中提出的查询表达式(译注:原文误为表达树):

```
SELECT title
FROM Movie
WHERE sutdioName = 'Paramount' AND year = 1979;
```

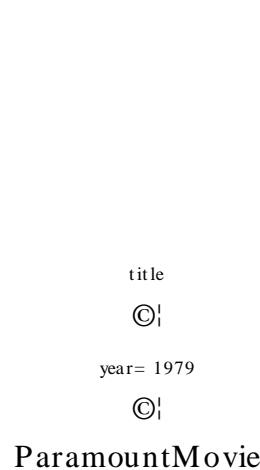


图 5.16 查询表达树

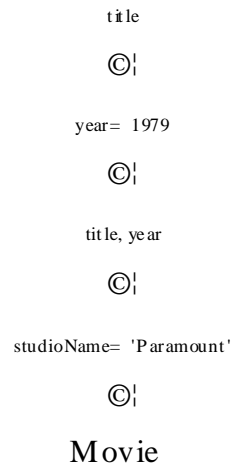


图 5.17 根据基本表查询的示意图

例如, 我们可以把投影 `title, year` 移到选择 `year = 1979` 之上。原因就是投影的延时绝对不会改变表达式的含义。这样, 就在一行中有了两个投影, 首先投影到 `title` 和 `year`, 然后仅投影到 `title`。显然第一个投影是冗余的, 我们可以将其删除。于是, 这两个投影就可以用 `title` 上的单个投影所代替。

两个选择也可以组合在一起。一般而言, 连续的两个选择可以用两个条件的“与” (AND) 构成的一个选择来代替。结果的表达树如图 5.18 所示。我们可以从如下查询直接得到该树:

```
SELECT title
FROM Movie
WHERE studioName = 'Paramount' AND year = 1979;
```

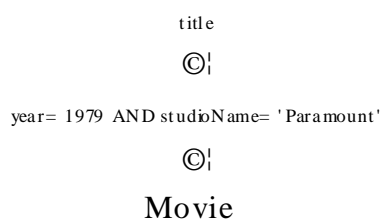


图 5.18 简化对基本表的查询

5.8.6 本节练习

练习 5.8.1: 根据我们不断滚动的实例中的基本表:

```
Movie(title, year, length, inColor, studioName, producerC# )
MovieExec(name, address, cert# , netWorth)
Studio(name, address, presC# )
```

构造下列视图:

- * (a) 视图 RichExec 给出净资产至少 10 000 000 美元的所有行政长官的姓名、地址、证书号和净资产。

- (b) 视图 StudioPres 给出身为制片公司总裁的所有行政长官的姓名、地址和证书号。
- (c) 视图 ExecutiveStar 给出既是行政长官又是影星的人的姓名、地址、性别、出生日期、证书号以及净资产。

练习 5.8.2: 练习 5.8.1 的视图中, 哪些是可更新的?

练习 5.8.3: 用练习 5.8.1 中的一个或多个视图而不用基本表, 写出下列所有查询:

- (a) 找出既是影星又是行政长官的女性的姓名。
- * (b) 找出既是制片公司总裁而资产又至少 10 000 000 美元的行政长官的姓名。
- ! (c) 找出既是影星而资产又至少 50 000 000 美元的制片公司总裁的姓名。
- * ! 练习 5.8.4: 对于例 5.40 中的视图和查询:
 - (a) 给出视图 MovieProd 的表达树。
 - (b) 给出该例的查询表达树。
 - (c) 按照你写的(a)和(b)的答案构造出根据基本表的查询表达式。
 - (d) 说明如何改变你写的(c)的表达式使之成为符合例 5.40 中给出的解答的等价表达式。

! 练习 5.8.5: 对于练习 5.8.3 中的每个查询, 用关系代数表达式表示出查询和视图, 代替查询表达式中使用的视图, 并尽可能简化结果表达式。写出与你写的基于基本表的结果表达式相对应的 SQL 查询。

练习 5.8.6: 用练习 4.1.3 中的基本表

Classes (class, type, country, numGuns, bore, displacement)
Ships (name, class, launched)

- (a) 定义视图 BritishShips, 为每一艘英国舰艇给出其等级、类型、火炮数量、口径、排水量和下水年份。
- (b) 利用(a)中的视图写一个查询, 找出 1919 年以前下水的所有英国战列舰的火炮数量和排水量。
- ! (c) 将(b)中的查询和(a)中的视图表示成关系代数表达式, 用来代替查询表达式中使用的视图, 并尽可能简化结果表达式。
- ! (d) 写出与(c)中的基于基本表 Classes 和 Ships 的表达式相对应的 SQL 查询。

5.9 空值和外部连接

在本节我们将进一步学习用 NULL 作为 SQL 元组中的一个值的有关内容。NULL 值特别重要的应用在于定义变体的连接运算, 这种运算在一个关系的某个元组未能和另一个关系的任一元组连接的情况下并不丢失信息。这种变体的连接称为“外部连接”。在本节我们将涉及到外部连接运算符的几种变体。尽管 NULL 存在于早期的 SQL 标准中, 但外部连接运算符一般只存在于支持 SQL2 的系统中。

5.9.1 对空值的运算

我们已经多次看到 SQL 支持称为 NULL 的特殊值。在 4.7.4 节讨论了 NULL 值的

关于空值的缺陷

这是一个很诱人的假设,即 SQL2 中的 NULL 总可以用来表示这样的含义:“一个确实存在但是我们还不知道的值”。然而,有一些情形是和直觉相违背的。例如,假设 x 是某元组的一个分量,该分量的域是整数。由于无论 x 是什么整数,它和 0 的乘积都是 0,因此我们可以推断出 $0 * x$ 的值必定为 0。但是,如果 x 的值为 NULL,并采用 5.9.1 节的规则 1,那么 0 和 NULL 的乘积是 NULL。与此相似,因为无论 x 是什么整数,它和自己的差都是 0,因此我们可推断出 $x - x$ 的值为 0。但是,又一次采用规则 1,结果却是 NULL。

一些用途,比如表示未知的或不存在的值。例如,当我们向关系中插入元组时如果所用的命令只提供元组的某些分量而不是所有分量,那么就会生成 NULL 值。除非对没有指定值的分量有另外说明的默认值,否则该分量就会是 NULL 值。我们将看到外部连接也是 NULL 值的另一种来源。

当我们对 NULL 值进行运算时必须记住两条重要的规则:

1. 当我们对 NULL 值和其他任何值(包括另一个 NULL 值)进行运算时,使用类似于 x 或 $+$ 的算数运算符,结果为 NULL。
2. 当我们将 NULL 值和任何值(包括另一个 NULL 值)进行比较时,使用类似于 $=$ 或 $>$ 的比较运算符,结果为 UNKNOWN。值 UNKNOWN 和 TRUE 或 FALSE 类似,是另一种真值;我们将简略地讨论如何使用 UNKNOWN 值。

但是,我们必须记住,尽管 NULL 值能够出现在元组中,但它不是常量。因此,当我们试图对值为 NULL 的表达式进行运算而使用上述规则时,不能显式地将 NULL 作为操作数使用。

例 5.45 假设 x 值为 NULL。那么 $x + 3$ 的值同样为 NULL。但是, $NULL + 3$ 不是合法的 SQL 表达式。与此相似, $x = 3$ 的值为 UNKNOWN,这是由于我们无法说出 x 的值(为 NULL)是否等于 3。然而,比较运算 $NULL = 3$ 不是正确的 SQL 表达式。

顺便提一下,询问 x 是否具有 NULL 值的正确方法是采用表达式 $x \text{ IS NULL}$ 。如果 x 具有 NULL 值,则表达式取 TRUE 值,否则表达式取 FALSE 值。与此相似, $x \text{ IS NOT NULL}$ 取值为真,除非 x 的值为 NULL。

5.9.2 真值 UNKNOWN

在 5.1.2 节我们观察到比较的结果或者为 TRUE,或者为 FALSE,并且这些真值(truth-value)可以用逻辑运算符 AND、OR 和 NOT 以显式的方法组合起来。我们刚才看到当 NULL 值出现时,比较运算可以产生第三种真值:UNKNOWN。现在我们必须了解在所有三种真值组合中,逻辑运算符的具体运算结果。

如果我们将 TRUE 看作 1(也就是完全的真),将 FALSE 看作 0(也就是完全没有真的成分),而将 UNKNOWN 看作 $1/2$ (也就是在真与假之间),规则就很容易记住了。即:

1. 两个真值的 AND 结果是二者的最小值。也就是说,如果 x 或者 y 为 FALSE,那么

x AND y 结果为 FALSE; 如果二者都不为 FALSE 但至少有一个为 UNKNOWN, 则结果为 UNKNOWN; 只有当 x 和 y 都为 TRUE 时结果才为 TRUE。

2. 两个真值的 OR 结果是二者的最大值。也就是说, 如果 x 或者 y 为 TRUE, 那么 x OR y 结果为 TRUE; 如果二者都不为 TRUE 但至少有一个为 UNKNOWN, 则结果为 UNKNOWN; 只有当二者都为 FALSE 时结果才为 FALSE。

3. 真值的非是 1 减去该真值。也就是说, 当 x 为 FALSE 时 NOT x 结果为 TRUE, 当 x 为 TRUE 时结果为 FALSE, 而当 x 为 UNKNOWN 时结果为 UNKNOWN。

图 5.19 汇总了操作数 x 和 y 的 9 种不同的真值组合用 3 种逻辑运算符运算的结果。最后一个运算符 NOT 的值仅取决于 x。

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

图 5.19 三值逻辑的真值表

把 select -from-where 语句或 DELETE 语句的 WHERE 子句中出现的 SQL 条件用于某个关系的每个元组, 对于每个元组, 都会产生 3 个真值 TRUE、FALSE 或 UNKNOWN 之一。但是, 只有条件值为 TRUE 的元组才能成为回答的一部分, 而把具有 UNKNOWN 或 FALSE 值的元组从回答中排除掉。这种情形导致了与在方框“关于空值的缺陷”中讨论的情形相似的另一种出乎意外的状态。

例 5.46 假定我们向不断滚动的实例中的关系

```
Movie(title, year, length, inColor, studioName, producerC# )
```

进行下面的查询:

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

直觉上看, 我们会期望得到关系 Movie 的一个副本, 因为每部电影的长度均为或者 120 分钟或更短, 或者比 120 分钟更长。

但是, 假定一些 Movie 元组的 length 分量为 NULL。那么比较运算 length <= 120 和 length > 120 的值都为 UNKNOWN。根据图 5.19, 两个 UNKNOWN 的 OR 还是 UNKNOWN。因此, 对于 length 分量为 NULL 的任何元组, WHERE 子句的值为

UNKNOWN。这样的元组不能作为查询结果的分量返回。结果, 查询真正的含义为“找出所有具有非 NULL 长度的 Movie 元组”。

5.9.3 SQL2 中的连接表达式

在介绍 SQL2 的外部连接运算之前, 我们先来考虑传统连接的简单情况。SQL2 中有几种连接运算符可以利用; 而以前的 SQL 标准并未明确包含这些运算符, 不过, 通过 select-from-where 查询可能获得相同的效果。在 SQL2 中, 连接表达式可以代替 select-from-where 并且可以用在允许 select-from-where 查询的任何地方。此外, 由于连接表达式可以生成关系, 所以, 也可以用在 select-from-where 查询的 FROM 子句中。

连接表达式最简单的格式为 CROSS JOIN; 该术语与我们在 4.1.4 节所称的笛卡尔积或简称的“乘积”(product)是同义词。例如, 如果我们想要如下两个关系的乘积

```
Movie( title, year, length, inColor, studioName, producerC# )
StarsIn( movieTitle, movieYear, starName )
```

我们可以表示为

```
Movie CROSS JOIN StarsIn;
```

结果为具有关系 Movie 和 StarsIn 所有属性的 9 列的关系。由 Movie 的一个元组和 StarsIn 的一个元组组成的每一对都构成结果关系的一个元组。

乘积关系中的属性可以称为 R.A, 其中 R 是两个相连的关系之一, A 是其属性之一。如果只有一个关系具有命名为 A 的属性, 那么 R 和圆点照常可以省略掉。该例中, 由于 Movie 和 StarsIn 没有共同的属性, 在乘积中 9 个属性名足够了。

然而, 和自己的乘积却几乎是无意义的操作。用关键字 ON 可以得到更常用的 连接。我们在两个关系名 R 和 S 之间写上 JOIN, 并在它们后面加上 ON 和条件。先做 R x S 的乘积运算, 随后按 ON 和后面的任何条件进行选择。

例 5.47 假定我们想要连接关系

```
Movie( title, year, length, inColor, studioName, producerC# )
StarsIn( movieTitle, movieYear, starName )
```

条件为要连接的元组都指向同一部电影。也就是说, 来自两个关系的名称和年份都必须相同。可以通过

```
Movie JOIN StarsIn ON
title = movieTitle AND year = movieYear;
```

进行该查询。结果又是具有显式属性名的 9 列关系。但是, 现在只有当一个来自 Movie 的元组和一个来自 StarsIn 的元组在名称和年份上都一致时, 两个元组才能组合形成一个结果元组。由于每个结果元组的 title 和 MovieTitle 两个分量具有相同的值, 并且 year 和 movieYear 两个分量也具有相同的值, 结果, 其中的两列是冗余的。

前面已经提到过, 连接表达式可以出现在 select-form-where 查询的 FROM 子句中。如果这样, 由连接表达式所表示的关系就将如同 FROM 子句中的基本表或视图一样处理。这种运用的一个实例如下。

例 5.48 如果我们关注例 5.47 中有两个冗余分量的事实, 我们可以将该例中的整

个表达式放在 FROM 子句中, 并用 SELECT 子句除去不必要的属性。于是, 我们可以这样写:

```
SELECT title, year, length, inColor, studioName, producerC# , starName
FROM Movie JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

从而得到 7 列的关系, 该关系是把关系 Movie 中的每个元组都通过该电影中的影星以所有可能的方式进行扩充而得到的。

5. 9. 4 自然连接

我们回顾 4. 1. 5 节, 就会看到自然连接(natural join)与 连接的不同之处在于:

- 1. 连接条件是两个关系中具有公共名字的所有属性对都相等, 不需要其他条件。
- 2. 对每个相等属性对之一进行投影。

SQL2 的自然连接就完全按这种方式工作。关键字 NATURAL JOIN 出现在两个关系之间表示运算符 。

例 5. 49 假定我们想要计算关系

```
MovieStar ( name, address, gender, birthdate )
MovieExec ( name, address, cert# , netWorth )
```

的自然连接。结果将是一个关系, 其模式中包括属性 name 和 address 加上出现在两个关系之一的所有属性。结果元组所表示的人既是影星又是行政长官, 元组中含有与两者有关的所有信息: 姓名、地址、性别、出生日期、证书号和净资产。表达式

```
MovieStar NATURAL JOIN MovieExec;
```

简明地描述了所需要的关系。

5. 9. 5 外部连接

外部连接是 SQL2 标准提供的一种连接的变体, 用来处理在特定的连接情况下的下列问题。假设我们希望计算连接 $R \bowtie S$ 。如果 R 中的一个元组 t 与 S 中的任何元组都不匹配, 那么 t 将从关系 $R \bowtie S$ 中消失。这种情况由于种种原因而显得不灵活实用。例如, 如果连接产生视图, 而我们只按照属于模式 R 的属性对视图进行查询, 那么直观上我们希望看到 t 出现在查询结果中。但实际上, 通过视图 $R \bowtie S$, t 变得不可见了, 所以同样的基于 R 的查询可能会产生出不同于基于 $R \bowtie S$ 的查询的结果。

外部连接不同于通常的(或称为“内部的”)连接, 它在结果中加上了每个关系中并没有和另一个关系中至少一个元组相连的任何元组。回顾一下例 4. 6, 那些未能与另一关系中任何元组相连的元组称为“悬浮元组”。由于连接的关系中的元组必须具有两个关系的所有属性, 因此对每个悬浮元组都要用 NULL 填充只属于另一个关系的属性, 然后再把悬浮元组加入到连接的结果关系中。

例 5. 50 假设我们希望连接两个关系

```
MovieStar( name, address, gender, birthdate )
MovieExec( name, address, cert# , netWorth )
```

但要保留那些是影星而不是行政长官或者是行政长官而不是影星的人。我们可以对两个关系执行 SQL2 中所谓的“自然完全外部连接”(natural full outerjoin)。其句法并不出乎意外：

MovieStar NATURAL FULL OUTER JOIN MovieExec;

该运算的结果是与例 5.49 具有相同的 6 属性模式的关系。该关系的元组有 3 种。描述同时是影星和行政长官的元组具有所有 6 个非 NULL 属性。这些也是例 5.49 结果中的那些元组。

第二种元组描述的是非行政长官的影星。这些元组在取自 MovieStar 元组的属性 name, address, gender 和 birthdate 上有对应值, 而只属于 MovieExec, 称作 cert# 和 netWorth 的属性其对应值为 NULL。

第三种元组描述非影星的行政长官。这些元组在取自 MovieExec 元组的属性 MovieExec 上有对应值, 而只来自 MovieStar 的 gender 和 birthdate 属性其对应值为 NULL。例如, 图 5.20 中所示的结果关系的三个元组分别对应于三种人。

name	address	gender	birthdate	cert#	networth
Mary Tyler Moore	Maple St.	'F'	9/9/99	12345	\$ 100...
Tom Hanks	Cherry Ln.	'M'	8/8/88	NULL	NULL
George Lucas	Oak Rd.	NULL	NULL	23456	\$ 200...

图 5.20 MovieStar 和 MovieExec 的外部连接中的三个元组

SQL2 中有许多外部连接的变体可以利用。首先, 除完全外部连接——其中两个关系的悬浮元组都用空值填充——以外, 我们可以得到左侧(left)外部连接, 其中只有左侧(第一个)关系中的悬浮元组用 NULL 填充并包含在结果中。例如,

MovieStar NATURAL LEFT OUTER JOIN MovieExec;

将生成图 5.20 中的前两个元组而不生成第三个。

与此类似, 右侧(right)外部连接只填充和包含来自右侧(第二个)关系的悬浮元组。因此,

MovieStar NATURAL RIGHT OUTER JOIN MovieExec;

将生成图 5.20 中的第一个和第三个元组而不生成第二个。

外部连接的第二种变体是我们如何指定匹配元组必须满足的条件。我们可以在连接后面加上 ON 以及匹配元组必须遵循的条件, 而不用关键字 NATURAL。如果我们还指定 FULL OUTER JOIN, 那么在匹配了来自于两个连接关系的元组之后, 还要为每个关系中的悬浮元组填充空值并把填充后的元组包含在结果之中。

例 5.51 让我们重新研究例 5.47, 在那里我们用如下条件将关系 Movie 和 StarsIn 连接起来, 条件就是两个关系中的 title 和 movieTitle 属性一致, 而且两个关系中的 year 和 movieYear 属性也一致。如果我们修改该例进行完全外部连接:

```
Movie FULL OUTER JOIN StarsIn ON
title = movieTitle AND year = movieYear;
```

那么我们将不但得到 StarsIn 中至少提到一个影星的电影元组, 而且得到没有列出影星的电影元组, 对应于属性 movieTitle, movieYear 和 starName 则用 NULL 填充。同样, 对于关系 Movie 所列出的任何电影中都没有出现的影星, 我们将得到 Movie 中 6 个属性的对应值均为 NULL 的元组。

在例 5. 51 提到的这类外部连接中, 关键字 FULL 可以用 LEFT 或者 RIGHT 来代替。例如,

```
Movie LEFT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

将给我们至少列出一个影星的 Movie 元组和没有列出影星而填上 NULL 的 Movie 元组, 但是不包括没有列出电影的影星。反过来,

```
Movie RIGHT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

将略掉没有列出影星的电影元组, 但是将包括没有列出任何电影而用 NULL 来填充的影星元组。

5. 9. 6 本节练习

练习 5. 9. 1: 对于我们不断滚动的电影数据库模式中的关系

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert# , netWorth)
Studio(name, address, presC# )
```

描述将在下列 SQL 表达式中出现的元组:

- (a) Studio CROSS JOIN MovieExec;
- (b) StarsIn FULL NATURAL OUTER JOIN MovieStar;
- (c) StarsIn FULL OUTER JOIN MovieStar ON name = starName;

* ! 练习 5. 9. 2: 应用数据库模式

```
Product (maker, model, type)
PC ( model, speed, ram, hd, cd, price)
Laptop ( model, speed, ram, hd, screen, price)
Printer (model, color, type, price)
```

写出 SQL 查询, 该查询将生成关于所有产品——PC 机、便携式电脑以及打印机——的信息, 包括其生产厂商(如果可以得到)以及与该产品有关的一切信息(就是说, 在该类产品的关系中寻找)。

练习 5. 9. 3: 用练习 4. 1. 3 的数据库模式中的两个关系

```
Classes (class, type, country, numGuns, bore, displacement)
Ships (name, class, launched)
```

写出 SQL 查询, 该查询将生成所有可得到的舰艇信息, 包括可从关系 Classes 中得到的信息。如果在 Ships 中没有提到关于某一等级的舰艇, 就不必生成该等级的有关信息。

！练习 5.9.4: 重复练习 5.9.3, 但是对于没有在 Ships 中提到的任一等级 C, 当舰艇的名称和等级同样为 C 时, 该舰艇的信息也要包含在结果当中。

！练习 5.9.5: 在例 5.46 中我们讨论了查询

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

当电影长度为 NULL 时, 其执行情况很不直观。找一个更简单的等价的查询, 该查询在 WHERE 子句中只有单一条件(在条件中没有 AND 或 OR)。

！练习 5.9.6: 我们在本节学到的连接运算符是冗余的, 在这个意义上, 它们总可以用 select-from-where 表达式来代替。说明如何用 select-from-where 写出下列表达式:

- * (a) R CROSS JOIN S;
- (b) R NATURAL JOIN S;
- (c) R JOIN S ON C; 其中 C 是 SQL 条件。

！！练习 5.9.7: 外部连接运算符也可以用涉及到其他 SQL 运算符的 SQL 查询来代替。不显式地使用连接或外部连接运算符, 说明如何重写下列语句:

- (a) R NATURAL LEFT OUTER JOIN S;
- (b) R NATURAL FULL OUTER JOIN S;
- (c) R FULL OUTER JOIN S ON C; 其中 C 是 SQL 条件。

5.10 SQL3 中的递归

在这一节, 我们将把重点放在 SQL3 的一个特性——递归查询上。这个特性刚刚在商业系统中崭露头角。相比之下, 本章前面各节以存在于 SQL2 中的特性为基础, 或者以能在几乎所有的商业系统中找到的类似特性为基础。此外, 当 SQL2 标准被正式采纳时, 本节描述的递归查询还是基于 SQL3 标准的一个草案, 而且可能在该草案的基础上有所发展。

SQL3 实现递归的方法以 4.4 节描述的递归 Datalog 规则为基础, 但是也有一些修改。首先, SQL3 标准建议, 只有线性递归, 也就是说, 最多有一个递归子目标的规则, 才是必须遵循的; 其次, 层次化的需求, 如我们在 4.4.4 节针对否定运算符所讨论的那样, 同样适用于 SQL 中能引起类似问题的其他运算符, 如聚合。

5.10.1 在 SQL3 中定义 IDB 关系

回忆一下 4.4 节的内容, 有助于我们区别作为存储表的外延数据库 (EDB, Extensional DataBase) 关系和用 Datalog 规则定义的内涵数据库 (IDB, Intensional DataBase) 关系。在 SQL3 中, 我们可以通过由关键字 WITH 引导的语句来定义等价的 IDB 关系。然后可以在 WITH 语句内部使用这些定义。WITH 语句的简单格式如下:

```
WITH R AS R 的定义 涉及到 R 的查询
```

这就是说, 可以定义一个名称为 R 的临时关系, 然后在某个查询中使用 R。更一般地, 在

WITH 之后可以定义几个关系,用逗号将它们的定义分开。这些定义中的任何一个都可以是递归的。定义的几个关系可以相互递归,这就是说,每个关系都可以根据某个其他关系(包括它本身在内)来定义。但是,涉及到递归的任何关系前都必须加上关键字 RECURSIVE(递归)。这样,WITH 语句的格式就是:

- 1. 关键字 WITH。
- 2. 一个或多个定义。定义之间用逗号分开,每个定义包括:
 - (a) 一个任选的关键字 RECURSIVE,要定义的关系是递归的则需要它。
 - (b) 要定义的关系的名称。
 - (c) 关键字 AS。
 - (d) 定义该关系的查询语句。
- 3. 一个查询,可以引用前面的任何定义,并形成该 WITH 语句的结果。

注意到这一点是很重要的:和关系的其他定义不同,在 WITH 语句中定义的关系只能用在语句内部,在其他地方不能使用。如果希望得到一个持久的关系,则应该在 WITH 语句之外,在数据库的模式中定义关系。

例 5.52 让我们重新考虑一下 4.4 节作为实例的航空公司航班信息。关于航班的数据存储于关系 Flights(airline, frm, to, departs, arrives) 中。实例中的实际数据在图 4.19 中给出,现在我们把它重新复制成图 5.21。

图 5.21 航空公司航班(重复图 4.19)

在例 4.37 中,我们利用图 5.21 中所表示的航班计算能从第一个城市飞到第二个城市这样的城市对的集合。在那个例子中,我们利用这样的两个规则:

- 1. Reaches (x, y) Flights (a, x, y, d, r)
- 2. Reaches (x, y) Reaches (x, z) AND Reaches (z, y)

计算出能把所需信息提供给我们的 IDB 关系 Reaches。

从这两个规则中,我们可以开发出同样意义的 SQL 的 Reaches 定义。该 SQL 查询成为 WITH 语句中的 Reaches 定义,而我们用所需要的查询来结束该 WITH 语句。在例 4.37 中,结果是整个 Reaches 关系,但是我们可以对 Reaches 关系进行某种查询,比如从丹佛(Denver)可以到达的城市的集合。

由于在 SQL 中 from 是关键字,所以我们把第二个属性名改为 frm。

相互递归

可以用图论的方法检查两个关系或者谓词是否是相互递归。构造一个依赖图, 节点对应于关系(如果是使用 Datalog 规则则对应于谓词)。如果关系 B 的定义直接依赖于关系 A 的定义, 则画一条从 A 指向 B 的弧。也就是说, 如果使用 Datalog 规则, 则 A 将连同前面的 B 一起出现在规则体中。在 SQL 中, A 将出现在 B 的定义中, 一般出现在 FROM 子句中, 但也可能作为一项出现在并、交或差中。

如果存在一个包含节点 R 和 S 的环, 那么 R 和 S 就是相互递归的。最常见的情况是一个从 R 到 R 的循环, 表明 R 递归地依赖于它本身。

请注意, 依赖图类似于我们在 4.4.4 节定义分层求反时引入的图。但是, 在那里我们必须区别肯定和否定依赖, 而这里不需要做这种区别。

```
1)  WITH RECURSIVE Reaches(frm,to) AS
2)      (SELECT frm,to FROM Flights)
3)      UNION
4)      (SELECT R1.frm,R2.to
5)      FROM Reaches AS R1,Reaches AS R2
6)      WHERE R1.to = R2.frm)
7)  SELECT * FROM Reaches;
```

图 5.22 对可到达城市对的 SQL3 查询

图 5.22 给出了如何用 SQL3 的查询来计算 Reaches。 1)行引入了 Reaches 的定义, 该关系实际的定义在 2)到 6)行给出。

该定义由两个查询的并集组成, 对应于例 4.37 中定义 Reaches 的两个规则。2)行是并集的第一项, 对应于第一个规则, 也就是基本规则。它表明, Flights 关系中每个元组的第二和第三分量(frm 和 to 分量)构成 Reaches 的一个元组。

4)到 6)行对应于 Reaches 定义中第二个规则, 也就是归纳规则。FROM 子句中 Reaches 的两个别名 R1 和 R2 代表规则中的两个 Reaches 子目标。R1 的第一个分量对应于规则 2 中的 x, R2 的第二个分量对应于 y。R1 的第二个分量和 R2 的第一个分量共同代表变量 z; 注意, 在 6)行这两个分量相等。

最后, 7)行描述了由整个查询生成的关系。它是 Reaches 关系的一个副本。作为替代, 我们可以用更复杂的查询来代替 7)行。比如:

```
7) SELECT to
FROM Reaches
WHERE frm = 'DEN' ;
```

将产生所有从丹佛可以到达的城市。

有一个技术细节, 那就是 SQL3 标准只要求线性递归, DBMS 所支持的定义递归关系的查询要求在 FROM 子句中递归关系只能出现一次。图 5.22 的 5)行两次用到递归关系 Reaches。虽然, 该查询极力模仿例 4.37 也许是最自然的方式。但是这样写 SQL3 DBMS 也许支持, 也许不支持。

5. 10. 2 线性递归

就像我们随同例 5. 52 一起提到的那样, 解答中存在一个技术上的缺点, 因为 SQL3 标准只要求 SQL3 的实现支持线性递归。形式上, 如果在任何要定义的关系的 FROM 子句中, 和该关系相互递归的关系最多只出现一次, 那么递归就是线性的。最常见的情况是, 要定义的关系本身在 FROM 子句中出现一次, 但也可能是, 出现一次的递归关系是与要定义的关系相互递归的其他某个关系。

例 5. 53 有趣的是, 我们可以简单地用 Flights 取代图 5. 22 中 5) 行的任一个 Reaches 来修改图 5. 22 中的代码。这将使递归具有 4. 4. 3 节的方框中讨论的左递归或右递归的形式。另一种方法是在 WITH 语句中定义一个附加的关系 Pairs, 代表 Flights 在 frm 和 to 分量上的投影, 并且在并集的两个部分都使用该关系。

对图 5. 22 的重新编码如图 5. 23 所示。在这里, 我们选择了右递归定义, 虽然我们也可以交换 7) 行的 Pairs 和 Reaches 的次序, 从而使定义为左递归。

```
1) WITH
2)      Pairs AS SELECT frm,to FROM Flights,
3)      RECURSIVE Reaches(frm,to) AS
4)          Pairs
5)      UNION
6)          (SELECT Pairs.frm, Reaches.to
7)          FROM Pairs, Reaches
8)          WHERE Pairs.to = Reaches.frm)
9) SELECT * FROM Reaches;
```

图 5. 23 可到达城市对的线性递归查询

请注意, 4. 4. 2 节概述的迭代固定点计算也能用于像这样的 SQL 查询, 就像把它用于 Datalog 规则一样。因为递归是线性的, Reaches 中的事实可能以某种不同的顺序找到, 但所有像(x, y)这样的能从 x 飞到 y 的城市对, 最终都会找到。

用我们的采样数据考虑一下第一次循环。因为 Reaches 初始化为空, 只有并集的第一项——4) 行的 Pairs——构成了由图 5. 21 的地图所表示的基本城市对。第二次循环, Pairs 和 Reaches 都提供一段航程, 所以我们得到了两段航程的像(SF, CHI)这样的城市对。下一次循环, Reaches 同时提供两段航程, 但是没有更多的城市对要加入进来。一般而言, 在第 i 次循环, 我们可以得到所有这样的城市对(x, y), 即由 i 条弧线所组成的从 x 到 y 的最短路径。

5. 10. 3 在 WITH 语句中使用视图

视图也可以像表那样在 WITH 语句中定义。句法上的唯一差别是在关系的定义中使用关键字 VIEW。

例 5. 54 在图 5. 23 中, 2) 行的关系 Pairs 也可以定义成视图。我们仅仅需要将 2) 行修改成:

```
2) VIEW Pairs AS SELECT frm,to FROM Flights
```

甚至有很好的理由将 Pairs 定义为视图。如果我们把它定义为真正的关系,那么当执行 WITH 语句时,它将首先作为一个整体构造出来,而与从 4) 行开始用它来构造 Reaches 无关。如果它是一个视图,那么可以用 Flights 中的元组的分量来代替它构造 Reaches。

5. 10. 4 分层求反

并不是任意的 SQL 查询都能以递归关系定义的形式出现。相反,对查询必须在某些方面加以限制。最重要的一个要求就是相互递归关系的否定必须是分层的,就像在 4. 4. 4 节所讨论的那样。在 5. 10. 5 节,我们将看到层次化的原则如何扩展到 SQL 中可以找到、而 Datalog 或关系代数中不存在的其他结构,如聚合。

例 5. 55 让我们重新看一下例 4. 39,我们要求找出这样的城市对(x,y),可以通过航空公司 UA,而不能通过 AA,从 x 到达 y。我们需要用递归来表达一个航空公司采用不确定的航线进行飞行的想法。但是,否定的情况是以分层的方式出现的:在例 4. 39 中,我们在使用递归计算出两个关系 UAreaches 和 AAreaches 以后,取它们的差集。

我们可以采用同样的策略写出 SQL3 的查询,虽然我们 must 像例 5. 53 那样用左线性或右线性的形式来取代例 4. 39 的非线性递归。然而,为了描述一种不同的处理方法,我们可以另外递归地定义单一的关系 Reaches(airline, frm, to),它的三元组(a, f, t)表示可以从城市 f 飞到城市 t,可能经过几段航程但是只乘坐一个航空公司 a 的班机。我们还将用到一个关系 Triples(airline, frm, to),它是 Flights 的三个相关分量的投影。查询如图 5. 24 所示。

关系 Reaches 的定义在 3) 到 9) 行由两项的并集组成。基本项是 4) 行中的关系 Triples。归纳项是 6) 到 9) 行的查询,该查询将对 Triples 和 Reaches 本身进行连接。这两项的结果是将所有的元组(a, f, t)都放到关系 Reaches 中,于是我们可以经过一段或几段航程从城市 f 到达城市 t,但是所有的旅程都乘坐航空公司 a 的班机。

10) 到 12) 行是查询本身。10) 行给出通过 UA 能够到达的城市对,12) 行给出通过 AA 能够到达的城市对。查询结果是这两个关系的差集。

```
1)  WITH
2)      Triples AS SELECT airline, frm, to FROM Flights,
3)      RECURSIVE Reaches (airline, frm, to) AS
4)          Triples
5)      UNION
6)          (SELECT Triples. airline, Triples. frm, Reaches. to
7)              FROM Triples, Reaches
8)              WHERE Triples. to = Reaches. frm AND
9)                  Triples. airline = Reaches. airline
          )
10)     (SELECT frm, to FROM Reaches WHERE airline = 'UA' )
11)  EXCEPT
12)     (SELECT frm, to FROM Reaches WHERE airline = 'AA' );
```

图 5. 24 对两个航空公司之一能到达的城市的分层查询

例 5.56 在图 5.24 中, 11) 行中 EXCEPT 所表示的否定明显是分层的, 因为只有 3) 到 9) 行的递归结束后它才适用。另一方面, 我们在例 4.40 观察到的对非分层求反的使用, 将转换成在递归关系的定义中对 EXCEPT 的使用。该实例可直接转换成 SQL3, 如图 5.25 所示。该查询只要求 P 的值, 虽然我们可以查询 Q 或者 P 和 Q 的某个函数。

图 5.25 中, 4) 行和 8) 行使用的两个 EXCEPT 在 SQL3 中是非法的, 因为在每种情况下第二个参数都和要定义的关系为相互递归的关系。于是, 这里所用的否定不是分层求反, 因此是不允许的。事实上, 在 SQL3 中没有这个问题的解决方案, 而且也不需要, 因为图 5.25 中的递归并不真正地定义关系 P 和 Q 的值。

```
1) WITH
2)      RECURSIVE P(x) AS
3)          (SELECT * FROM R)
4)      EXCEPT
5)          (SELECT * FROM Q),
6)      RECURSIVE Q(x) AS
7)          (SELECT * FROM R)
8)      EXCEPT
9)          (SELECT * FROM P)
10) SELECT * FROM P;
```

图 5.25 非分层查询, 在 SQL3 中非法

5.10.5 SQL3 递归中的未定表达式

我们在例 5.56 中已经看到, 在递归定义中使用 EXCEPT 违背了 SQL3 所要求的分层求反原则。然而, 还有另一些不使用 EXCEPT 的查询形式也是无法令人接受的。例如, 关系的否定可以用 NOT IN 来表示。因此, 图 5.25 的 2) 到 6) 行也可以写为:

```
RECURSIVE P(x) AS
    SELECT x FROM R WHERE x NOT IN Q
```

这种改写仍保留了非分层的递归, 因此是非法的。

另一方面, 在 WHERE 子句中仅仅使用 NOT, 如 NOT x= y(当然也可以写成 x< > y)并不会自动和分层求反的条件相违背。那么, 在 SQL3 中可以用哪些种类的 SQL 查询来定义递归关系? 有什么通用规则?

其原则就是, 对于合法的 SQL3 递归, 递归关系 R 的定义只能涉及到使用相互递归关系 S(S 可以是 R 本身), 条件是 S 的使用是单调的。如果在 S 中加入任意的元组会在 R 中加入一个或多个元组, 或者使 R 保持不变, 但决不能导致删除 R 中的任何元组, 则 S 的使用是单调的。

这条规则在考虑 4.4.2 节描述的最小固定点计算概要时是有意义的。我们从递归定义 IDB 关系为空开始, 然后依次循环地重复加入元组。如果在某一循环中加入的元组

在技术上, 非线性递归即使没有否定在 SQL3 中也是无法接受的, 尽管 SQL3 的文档中约定非线性递归将“出现在 SQL4 中”。但是, 在这里讨论无意义的荒谬的递归形式, 总强于讨论纯粹非 SQL3 标准的递归形式。

会导致我们不得不在下一循环中删掉元组,那么就会出现振荡的危险,而固定点计算可能永远不会收敛了。在下例中,我们将看到某些非单调的、在 SQL3 递归中非法的构造。

例 5.57 回顾一下例 4.40,这是一个非分层求反的例子,图 5.25 是例 4.40 中 Datalog 规则的一个实现。规则允许两个不同的最小固定点。正如我们所预料的,在图 5.25 中 P 和 Q 的定义是非单调的。以 2) 到 5) 行的 P 的定义为例。P 依赖于 Q,与之相互递归,但向 Q 中加入元组会从 P 中删除元组。看看为什么,假定 R 包含两个元组(a)和(b),而 Q 包含元组(a)和(c)。那么 $P = \{(b)\}$ 。然而,如果我们将(b)加入 Q,则 P 为空。加入一个元组导致删除一个元组,所以我们得到一个非单调的非法构造。

当我们试图通过计算最小固定点对关系 P 和 Q 求值时,缺乏单调性将直接导致振荡的状态。例如,假定 R 有两个元组{(a),(b)}。最初,P 和 Q 均为空。这样,在第一次循环,图 5.25 的 3)到 5)行计算出 P 值为{(a),(b)},由于 9)行用的是 P 原有的空值,因此 7)到 9)行计算出的 Q 与 P 具有同样的值。

现在,R,P 和 Q 的值均为{(a),(b)}。因此,在下一次循环中,3)到 5)行和 7)到 9)行分别计算出 P 和 Q 均为空。在第三次循环,二者均取值为{(a),(b)}。这个过程将永远延续下去,在偶数次循环两个关系均为空而在奇数次循环均为{(a),(b)}。因此,从 P 和 Q 在图 5.25 中的“定义”出发我们永远不会得到这两个关系的明确的值。

例 5.58 聚合也会导致非单调性,尽管在最初这种关系并不很明显。假定我们有由下面两个条件定义的一元(单属性)关系 P 和 Q:

- 1. P 是 Q 和 EDB 关系 R 的并集。
- 2. Q 中的一个元组为 P 的成员的总和。

我们可以用 WITH 语句表示这两个条件,不过该语句和 SQL3 的单调性要求相违背。图 5.26 中给出寻找 P 值的查询。

```
1) WITH
2)      RECURSIVE P(x) AS
3)          (SELECT * FROM R)
4)      UNION
5)          (SELECT * FROM Q)
6)      RECURSIVE Q(x) AS
7)          SELECT SUM(x) FROM P
8) SELECT * FROM P;
```

图 5.26 涉及到聚合的非分层查询,在 SQL3 中非法

假设 R 包含元组(12)和(34),由于 P 和 Q 必须处于固定点计算的起点,因此它们最初均为空。图 5.27 汇总了前 6 次循环计算出的值。回忆一下,我们采用的策略就是,在一次循环中所有关系的计算都使用前一次循环的值。因此,第一次循环的计算结果 P 和 R 相同,而由于 P 原有的空值用在 7)行,结果 Q 为空。

递归非单调时,在 WITH 子句中对关系求值的顺序会影响最后的结果,而递归单调时结果与顺序无关。在本例和下面的例子中,我们将假定在每次循环中,P 和 Q 均“并行”求值。即每次循环都用一个关系的旧值计算另一个关系。

在固定点计算中使用新值

人们可能会问我们为什么在例 5.57 和 5.58 中用 P 的旧值计算 Q, 而不用 P 的新值。结果将可能依赖于我们在 WITH 子句中列出的递归谓词定义的顺序。在例 5.57 中, P 和 Q 将收敛于两个可能的固定点之一, 这取决于计算的顺序。在例 5.58 中, P 和 Q 仍不会收敛, 事实上它们在每次循环而不是每隔一次循环都会发生变化。

循环次数	P	Q
1)	{(12), (34)}	
2)	{(12), (34)}	{(46)}
3)	{(12), (34), (46)}	{(46)}
4)	{(12), (34), (46)}	{(92)}
5)	{(12), (34), (92)}	{(92)}
6)	{(12), (34), (92)}	{(138)}

图 5.27 非单调聚合的固定点迭代计算

在第二次循环, 3) 到 5) 行的并是集合 $R = \{(12), (34)\}$, 这也就是新的 P 值。P 的原有值和新值是相同的, 所以在第二次循环 $Q = \{(46)\}$, 46 是 12 与 34 的和。

在第三次循环, 我们通过 2) 到 5) 行得到 $P = \{(12), (34), (46)\}$ 。用原有的 P 值, $\{(12), (34)\}$, Q 通过 6)、7) 两行再一次定义为 $\{(46)\}$ 。

在第四次循环, P 具有和上次相同的值, $\{(12), (34), (46)\}$, 而由于 $12 + 34 + 46 = 92$, Q 取值为 $\{(92)\}$ 。注意 Q 尽管得到了元组 (92), 但是失去了元组 (46)。也就是说, 将元组 (46) 加入到 P 中导致从 Q 中删除掉一个元组 (碰巧是同一元组)。这种情形就是 SQL3 在递归定义中所禁止的非单调性, 可以确认图 5.26 中的查询是非法的。一般来说, 在第 $2i$ 次循环, P 包含元组 (12)、(34) 和 $(46i - 46)$, 而 Q 仅包含元组 $(46i)$ 。

5.10.6 本节练习

练习 5.10.1: 在 4.36 节, 我们讨论了关系

$SequelOf(movie, sequel)$

它给出电影直接的续集。我们还定义了一个 IDB 关系 FollowOn, 其 (x, y) 对是这样的电影, y 或者是 x 的续集, 或者是其续集的续集, 以此类推。

- (a) 写出作为 SQL3 递归式的 FollowOn 的定义。
- (b) 写出递归 SQL3 查询, 返回 (x, y) 对的集合, 其中电影 y 是电影 x 的后集, 但不是续集。
- (c) 写出递归 SQL3 查询, 返回 (x, y) 对的集合, 其含义为 y 是 x 的后集, 但既不是续集也不是续集的续集。
- ! (d) 写出递归 SQL3 查询, 返回电影 x 的集合, 而 x 至少有两个后集。注意两个后集不能其中一个是续集, 另一个是续集的续集。

! (e) 写出递归 SQL3 查询, 返回 (x, y) 对的集合, 其中电影 y 是 x 的后集, 而 y 最多有一个后集。

练习 5.10.2: 在练习 4.4.3 中, 我们介绍的关系

$\text{Rel}(\text{class}, \text{eclass}, \text{mult})$

描述了一个 ODL 类是如何同其他的类联系在一起的。特别是, 如果存在一个从类 c 到类 d 的关系, 该关系具有元组 (c, d, m) 。如果 $m = \text{'multi'}$ 则该关系是多值的, 而如果 $m = \text{'single'}$ 则该关系是单值的。我们在练习 4.4.3 还提到可以将 Rel 看作图的定义, 图的节点是类, 当且仅当 (c, d, m) 为 Rel 的元组时, 图中将有一条从 c 到 d 标记为 m 的弧。

(a) 写出生成 (c, d) 对的集合的递归 SQL3 查询, 从而使得上面描述的图中存在从 c 到 d 的路径。

* (b) 写出生成 (c, d) 对的集合的递归 SQL3 查询, 从而使得存在从 c 到 d 的路径, 而每条沿着路径的弧都标记为 single 。

* ! (c) 写出生成 (c, d) 对的集合的递归 SQL3 查询, 从而使得存在从 c 到 d 的路径, 而至少有一条沿着路径的弧标记为 multi 。

(d) 写出生成 (c, d) 对的集合的递归 SQL3 查询, 从而使得存在从 c 到 d 的路径, 但是不存在所有沿着路径的弧都标记为 single 的路径。

! (e) 写出生成 (c, d) 对的集合的递归 SQL3 查询, 从而使得存在从 c 到 d 的路径, 而所有沿着路径的弧交替地标记着 single 与 multi 。

(f) 写出生成 (c, d) 对的集合的递归 SQL3 查询, 从而使得存在从 c 到 d 的路径和从 d 到 c 的路径, 而每条沿着路径的弧都标记为 single 。

* ! 练习 5.10.3: 假设我们用非线性递归来修改图 5.23 中的 Reaches 计算。特别是, 6) 到 8) 行可以替换成:

6) (SELECT First.frm, Second.to

7) FROM $\text{Reaches AS First, Reaches AS Second}$

8) WHERE First.to = Second.frm)

在这里, 为得到新的对而把元组变量 First 和 Second 表示的 Reaches 的两份副本连接起来。在固定点计算的第 i 次循环中, 加到 Reaches 的新路径有多长?

5.11 本章总结

SQL: SQL 语言是关系数据库系统的主要查询语言。1997 年对商业系统产生最大影响的标准称为 SQL2。该语言的一个正在制订的标准, SQL3, 有望在不久完成。

select-from-where 查询: SQL 查询最常见的形式为 select-from-where 形式。它允许我们得到几个关系的积(FROM 子句), 把条件用于结果的元组(WHERE 子句)以及生成需要的分量(SELECT 子句)。

子查询: 也可以把 select-from-where 查询作为子查询用于另一个查询的 WHERE 子句中。可以对子查询结果的关系使用运算符 EXISTS、IN、ALL 以及 ANY 来表示布尔值条件。

设置关系运算：我们可以分别使用关键字 UNION、INTERSECT 和 EXCEPT 把关系或者定义关系的查询联系起来从而得到关系的并、交和差。

关系的包模型：SQL 实际上将关系看作是元组的包，而不是元组的集合。我们可以用关键字 DISTINCT 强制除掉重复元组，而在某些情况下包并不是默认的，这时可用关键字 ALL 令结果为包。

聚合：出现在关系的某一系列的值可以用关键字 SUM、AVG(平均值)、MIN、MAX 或 COUNT 之一来汇总(聚合)。元组可以在聚合之前用关键字 GROUP BY 进行分组。某些组可以用关键字 HAVING 引入的子句略掉。

更新语句：SQL 允许我们改变关系中的元组。我们可以 INSERT(插入新元组)、DELETE(删除元组)或 UPDATE(修改某些已有的元组)——通过写出利用这三个关键字之一的 SQL 语句来实现。

数据定义：SQL 有有来说明数据库模式的各个元素的语句。CREATE TABLE 语句允许我们说明存储的关系(称为表)的模式，指定其属性和类型。我们也可以用 CREATE DOMAIN 语句来定义数据类型的名称，然后该名称可以用在关系模式的说明中。这些 CREATE 语句还允许我们说明属性和域的默认值。

更改模式：我们可以用 ALTER 语句更改数据库模式的外观。这些更改包括在关系模式中增加和删除属性，以及更改与属性或域相关的默认值。我们还可以用 DROP 语句完全撤消关系、域或其他的模式元素。

索引：虽然不是 SQL 标准的一部分，但是商业 SQL 系统允许对属性说明索引；这些索引加速了某些涉及到索引属性指定值的查询或更新。

视图：视图的定义说明如何从存储在数据库中的表来构造一个关系(视图)。可以像对表一样对视图进行查询，而 SQL 系统将修改对视图的查询，从而用对定义视图的表的查询来代替该查询。

空值：SQL 提供了特殊值 NULL，出现在无法得到具体值的元组分量中。NULL 的算术和逻辑运算具有特殊性。任何值(甚至另一个 NULL)和 NULL 进行比较，都得到真值 UNKNOWN。反过来，在布尔值表达式中该真值就好像处于 TRUE 和 FALSE 之间的中间状态。

连接表达式：SQL 有诸如可以应用于关系的 NATURAL JOIN 之类的运算符，或者本身作为查询，或者在 FROM 子句中定义关系。

外部连接：SQL 还提供了 OUTER JOIN 运算符来连接关系，但是也在结果中包括了来自一个或两个关系中的悬浮元组；在结果关系中悬浮元组将用 NULL 来填充。

SQL3 递归：SQL3 标准将包含递归定义临时关系并在查询中使用这些关系的方法。推荐标准要求递归中涉及到的否定和聚合都是分层的；也就是说，递归定义的关系不能根据其本身的否定或者聚合来定义。

5.12 本章参考文献

可以通过全国科学技术学会(NIST, 其前身为美国国家标准局(National Bureau of Standards))联机获得 SQL2 和 SQL3 标准。这些文档可以通过匿名的 FTP 或 HTTP 来获取。主机名为 speckle.ncsl.nist.gov。SQL2 标准和 SQL3 标准的当前版本可以在目录

isowg3/dbl/BASEdocs

中找到。若对 SQL2 的正式语法特别感兴趣, 则可查阅文件

isowg3/dbl/BASEdocs/sql-92.bnf

目录 isowg3/x3h2 中包含了许多解释 SQL2 和 SQL3 标准的现行的和历史的文档。这些文档全都具有以 X3H2 开头的报告编号。

要想通过 HTTP 获取 SQL 文档, 使用 URL

<http://speckle.ncsl.nist.gov/~ftp>

后面加上一个上面提到的目录的路径。

有些书给出更多 SQL 编程细节。我们最感兴趣的是[2], [3]和[6]。

SQL 最初在[4]中定义。它作为系统 R^[1](第一代关系数据库原型之一)的一部分而实现。参考文献[5]是 SQL3 递归的原始资料。

- [1] Astrahan, M. M. et al., System R: a relational approach to data management, ACM Transactions on Database Systems, 1: 2, pp. 97 ~ 137, 1976.
- [2] Celko, J., SQL for Smarties, Morgan-Kaufmann, San Francisco, 1995.
- [3] Date, C. J. and H. Darwen, A Guide to the SQL Standard, Addison-Wesley, Reading, MA, 1993.
- [4] Chamberlin, D. D., et al., SEQUEL 2: a unified approach to data definition, manipulation, and control, IBM Journal of Research and Development, 20: 6, pp. 560 ~ 575, 1976.
- [5] Finkelstein, S. J., N. Mattos, I. S. Mumick, and H. Pirahesh, Expressing recursive queries in SQL, ISO WG3 report X3H2-96-075, March, 1996.
- [6] Melton, J. and A. R. Simon, Understanding the New SQL: A Complete Guide, Morgan-Kaufmann, San Francisco, 1993.

第 6 章 SQL 中的约束和触发程序

本章将涉及关于建立“主动性”(active)元素的 SQL 的各个方面。主动性元素是这样的表达式或语句,即一旦写出就保存在数据库中,并希望在适当的时候予以执行。动作的时机可能是某个事件发生的时候,比如对特定的关系进行插入操作的时候,也可能是数据库发生变化导致某个布尔值条件为真的时候。

编写更新数据库应用程序的人所面对的重要问题之一就是新的信息可能在很多方面都是错误的。数据库更新过程中,为了保证关系中不出现不适当的元组,最直接的方法就是编写应用程序,从而对每个插入、删除、修改命令都进行与之相关的保证正确性的必要检查。不幸的是,正确性检查往往都很复杂,并且总是重复的;应用程序必须在每次更新操作之后进行相同的检验。

幸运的是,SQL2 将表示完整性约束的各种技术作为数据库模式的一部分提出来。在本章中,我们将研究其主要方法。首先是键码约束,把某个属性或属性集说明为关系的键码。其次,我们考虑参照完整性,即要求某个关系中的属性或属性组的值(如 Studio 中的 presC#)还必须作为另一个关系中的属性或属性组的值(如 MovieExec 中的 cert#)出现。然后,我们会看到对域的一些约束,包括唯一性(“键码”),将域限制为某些特定值,不允许为 NULL 值。接下来,我们将研究把元组或关系作为整体的约束,以及关系之间称为“断言”(assertion)的约束。每当对相关关系进行更新时,都要对这些约束进行检验。

最后,我们讨论“触发程序”(trigger),它是由某些特定事件(例如对特定的关系进行插入)启动的主动性元素的一种形式。在 SQL2 标准中没有出现触发,而在后续的标准 SQL3 中,则包含了触发。尽管在本书的写作过程中 SQL3 并没有最终完成,但一些商用数据库系统为用户提供了某种形式的触发程序。

6.1 SQL 中的键码

也许在数据库中最重要约束类型就是说明某个属性或属性集构成关系的键码。也就是,禁止关系的两个元组在说明为键码的属性上一致,或者禁止在共同构成键码的属性集的所有属性上一致。就像其他许多约束一样,键码约束在 SQL 的 CREATE TABLE 命令中说明。有两种相似的说明键码的方法:使用关键字 PRIMARY KEY 或关键字 UNIQUE。然而,表中只能有一个主键码,但可以有任意数量的“unique”说明。

6.1.1 说明键码

主键码可以由关系的一个或多个属性组成。在定义存储关系的 CREATE TABLE 语

句中有两种方式来说明主键码。

- 1. 可以在关系模式中列出属性时说明某个属性为主键码。
- 2. 可以在模式中说明的项目表(到目前为止仅为属性表)中加入额外的说明,说明某个特定的属性或属性集构成主键码。

对于方法 1,我们在属性及其类型之后加上关键字 PRIMARY KEY。对于方法 2,我们将在属性表中引入新元素,该元素由关键字 PRIMARY KEY 以及括号内的构成该键码的属性或属性集组成。注意,如果键码中有多个属性,则要求用方法 2。

例 6.1 让我们重新考虑例 5.32 中关系 MovieStar 的模式。该关系的主键码为 name。因此,我们可以在说明 name 的行中加入该事实。图 6.1 是反映这种变化的图 5.13 的修订版。

作为选择,我们也可以用主键码的单独的定义。在图 5.13 的 5) 行后面加上主键码的说明,而不必在 2) 行中说明。最终的模式说明如图 6.2 所示。

```
1) CREATE TABLE MovieStar(  
2)     name CHAR(30) PRIMARY KEY,  
3)     address VARCHAR(255),  
4)     gender CHAR(1),  
5)     birthdate DATE  
    );
```

图 6.1 使 name 为主键码

```
1) CREATE TABLE MovieStar(  
2)     name CHAR(30),  
3)     address VARCHAR(255),  
4)     gender CHAR(1),  
5)     birthdate DATE,  
6)     PRIMARY KEY (name)  
    );
```

图 6.2 单独说明主键码

注意,在例 6.1 中,由于主键码是单一属性,图 6.1 和图 6.2 的格式都是可以接受的。但是,在主键码超过一个属性的情况下,我们必须用图 6.2 的格式。例如,如果我们说明关系 Movie 的模式,其键码为属性对 title 和 year,就要在属性表后面加上一行

PRIMARY KEY (title, year)

另一种说明键码的方法是用关键字 UNIQUE。这个单词恰好可以出现在 PRIMARY KEY 可以出现的地方:或者跟在属性及其类型后面,或者作为 CREATE TABLE 语句中一个单独的项目。它和主键码的说明具有同样的含义。但是,表中可以有任意数量的 UNIQUE 说明,却只能有一个主键码。

例 6.2 图 6.1 的 2) 行也可以写成

```
2) name CHAR(30) UNIQUE,
```

如果我们认为两个影星不可能具有同样的地址(值得怀疑的假设),那么也可以将 3) 行改为

```
3) address VARCHAR(255) UNIQUE,
```

同样,如果愿意选择另一种格式,那么也可以将图 6.2 中的 6) 行改为

```
6) UNIQUE (name)
```

键码约束是既可以用 PRIMARY KEY 也可以用 UNIQUE 说明的约束。关于这两种说明之间区别的要点,可参看 6.2.2 节的方框“主键码和唯一值属性”。

6.1.2 实施键码约束

回顾我们在 5.7.7 节讨论的索引,可以从中了解到,尽管索引并不是 SQL 标准的一部分,但是每种 SQL 实现都有办法把建立索引作为数据库模式定义的一部分。为了支持指定主键码值的普通类型的查询,建立基于主键码的索引是很正常的。我们也会想到在说明为 UNIQUE 的其他属性上建立索引。

于是,当查询的 WHERE 子句中包含了使键码等于特定值的条件时,例如在例 6.1 的 MovieStar 关系的情形下使 name = 'Audrey Hepburn',不用查找关系中所有的元组,就可以很快地找到匹配的元组。

许多 SQL 的实现提供了带关键字 UNIQUE 的索引建立语句,在为属性建立索引的同时将属性说明为键码。例如,语句

```
CREATE UNIQUE INDEX YearIndex ON Movie(year);
```

和 5.7.7 节的例子中的建立索引语句具有相同的效果,但是它还为关系 Movie 的属性 year 说明了唯一性约束(这不是一个合理的假设)。

让我们研究一下 SQL 系统是如何实施键码约束的。原则上,每当我们试图改变数据库时必须对约束进行检验。但是,必须明确的是,只有更新关系 R 时关系 R 的键码约束才可能发生违背。实际上,R 的删除操作不会导致违背;只有插入或修改操作才会。因此,只有对关系进行插入或修改时才检验键码约束,这对于 SQL 系统已经成为通用惯例了。

SQL 系统要想有效地实施键码约束,在说明为键码的属性上建立索引是非常重要的。如果能够得到索引,那么无论向关系中插入元组或修改某个元组的键码属性值,我们都应利用索引来检验在说明为键码属性上具有相同值的元组是否已经存在。如果存在,系统就必须阻止该更新的发生。

如果在键码属性上没有索引,原则上依然可以实施键码约束。但是,系统为查找具有相同键码值的元组就必须搜索整个关系。这个过程是极其费时的,因此,对大型关系数据库的更新实际上变得不可行了。

6.1.3 本节练习

- * 练习 6.1.1: 在 3.9 节中我们不断滚动的电影数据库实例为其所有关系定义了键码。修改你在练习 5.7.1 中的 SQL 模式说明,使之包括所有这些关系的键码说明。回顾一下,StarsIn 的所有三个属性构成其键码。
- 练习 6.1.2: 为练习 4.1.1 中的 PC 机数据库的关系列出合适的键码。修改练习 5.7.2 中的 SQL 模式,使之包括这些键码的说明。
- 练习 6.1.3: 为练习 4.1.3 中的战列舰数据库的关系列出合适的键码。修改你在练习 5.7.3 中的 SQL 模式,使之包括这些键码的说明。

6.2 参照完整性和外键码

数据库模式的第二种重要的约束类型就是某些属性的值必须有意义。也就是,如关系

Studio 中的 presC# 属性要求表示某个特定的电影行政长官。“参照完整性”约束不言而喻,如果某个制片公司元组在 presC# 分量中有确定的证书号 c,那么 c 就不会是假的:它是某个实际的电影行政长官的证书号。按照数据库的术语,“实际”的行政长官指的是在 MovieExec 关系中提到的行政长官。因此,必然存在某个 MovieExec 元组,其 cert# 属性值为 c。

6.2.1 说明外键码约束

在 SQL 中我们可以将一个关系的属性或属性组说明为外键码,它参照第二个关系(可以是同一个关系)的某个(些)属性。该说明的含意是双重的:

- 1. 必须说明第二个关系的被参照属性为该关系的主键码。
- 2. 出现在第一个关系的外键码属性中的任何值也必须出现在第二个关系的相应属性中。也就是存在着连接这两个属性或属性集的参照完整性约束。

像对主键码一样,我们采用两种方法来说明外键码。

- (a) 如果外键码是单一属性,我们可以在其名称和类型后面加上“参照”某个表的某个属性(必须为主键码)的说明。说明的格式为:

REFERENCES 表 (属性)

- (b) 另一种方法是,在 CREATE TABLE 语句的属性表后面加上一个或多个表明属性集为外键码的说明。然后,我们给出外键码所参照的表及其属性(必须为主键码)。说明的格式为:

FOREIGN KEY 属性 REFERENCES 表 (属性)

例 6.3 假设我们想要说明关系

Studio(name, address, presC#)

其主键码为 name, 并且有一个外键码 presC# 参照如下关系的 cert# :

MovieExec(name, address, cert# , netWorth)

我们可以直接说明 presC# 参照 cert# , 如下所示:

```
CREATE TABLE Studio(  
    name CHAR( 30) PRIMARY KEY,  
    address VARCHAR( 255),  
    presC# INT REFERENCES MovieExec(cert# )  
);
```

另一种格式是对外键码单独说明, 如

```
CREATE TABLE Studio(  
    name CHAR( 30) PRIMARY KEY,  
    address VARCHAR( 255),  
    presC# INT,  
    FOREIGN KEY presC# REFERENCES MovieExec(cert# )  
);
```

注意, 外键码所参照的属性, MovieExec 中的 cert# , 实际上是所在关系的主键码。如

主键码和唯一值属性

PRIMARY KEY 的说明几乎是 UNIQUE 说明的同义词。最明显的区别就是,对于表只能有唯一的一个主键码,但是可以有任意数量的 UNIQUE 属性或属性集。然而,还有一些细微的差别。

1. 外键码只能参照某个关系的主键码。
2. 数据库管理系统的实现有给“主键码”概念赋予某种特定含义的选项,但这并不是 SQL2 标准的一部分。例如,就像在 6.1.2 节讨论的那样,数据库销售商可能总是在说明为主键码的键码上设置索引,即使该键码包含不只一个属性,但在其他属性上就需要用户显式地请求建立索引。另外,如果表有主键码的话,可能总是按其主键码排序。

果我们要将 presC# 合法地说明为 Studio 的外键码,而所要参照的是 MovieExec 中的 cert#, 那么就需要将 cert# 说明为 MovieExec 的主键码。

这两种外键码说明的含义为,无论何时只要有某个值出现在 Studio 元组的 presC# 分量中,该值也必须出现在某个 MovieExec 元组的 cert# 分量中。一个例外就是,即使某个特定的 Studio 元组将 NULL 作为其 presC# 分量的值,也不必将 NULL 作为 cert# 分量的值(实际上,无论如何也不允许在主键码属性中出现 NULL 值,这听起来更现实一些;参见 6.3.1 节)。

6.2.2 保持参照完整性

我们已经看到如何说明外键码,并且也了解到这种说明意味着外键码中的任何非 NULL 值也必须出现在被参照关系的相应属性中。但是,当面临对数据库的更新时如何保持这种约束呢?数据库实现者可以在三个可选方案中选择。

默认策略:拒绝违法更新

SQL 的默认策略就是任何与参照完整性约束相违背的更新均为系统所拒绝。例如,考虑例 6.3,要求关系 Studio 中的 presC# 值也是 MovieExec 中的 cert# 值。如下的动作将为系统所拒绝(也就是产生运行时错误)。

1. 我们尝试插入一个新的 Studio 元组,其 presC# 值不是 NULL 并且也不是任何 MovieExec 元组的 cert# 分量。该插入将为系统所拒绝,该元组决不会插入到 Studio 中。
2. 我们尝试修改一个 Studio 元组,将其 presC# 分量改为不是任何 MovieExec 元组 cert# 分量的非 NULL 值。该修改将遭到拒绝,该元组不会改变。
3. 我们尝试删除一个 MovieExec 元组,其 cert# 分量作为 presC# 分量出现在一个或多个 Studio 元组中。该删除将遭到拒绝,该元组将保留在 MovieExec 之中。
4. 我们尝试通过改变 cert# 的值来修改一个 MovieExec 元组,而原有的 cert# 是某个电影制片公司的 presC# 值。所要做的修改将再次遭到拒绝,MovieExec 也保持原样。

级联策略

要对 MovieExec 这样的被参照关系进行删除或修改(也就是上述的第三和第四类更新)还有另一种处理方法,称为级联策略(cascade policy)。在这种策略下,当我们删除对应于某制片公司总裁的 MovieExec 元组时,为了保持参照完整性,系统必须从 Studio 中删除参照的元组。修改也以类似方式处理。如果我们将某个电影行政长官的 cert# 从 c1 改为 c2,而存在其 presC# 分量值为 c1 的某个 Studio 元组,那么系统也将该 presC# 分量值改为 c2。

置空策略

还有一种处理方法,就是将 presC# 的值从所删除或修改的制片公司总裁的对应值改为 NULL;这种策略称为置空(set-null)。

这些选项都可以单独地用于删除和修改,这些选项是和外键码的说明一起描述的。可以通过在 ON DELETE 或 ON UPDATE 后面加上所选择的 SET NULL 或 CASCADE 来说明相应的选项。

例 6.4 让我们看看如何修改例 6.3 中的如下关系的说明

Studio(name, address, presC#)

来规定对如下关系的删除和修改操作

MovieExec(name, address, cert# , netWorth)

图 6.3 采用了例 6.3 中的第一个 CREATE TABLE 语句,并用 ON DELETE 和 ON UPDATE 子句对其扩展。5)行指出当我们删除 MovieExec 元组时,将身为总裁的他(或她)所在的制片公司的 presC# 置为 NULL。6)行指出,如果我们修改 MovieExec 元组的 cert# 分量,那么,Studio 中其 presC# 分量具有相同值的任何元组将进行同样的修改。

注意,在本例中,置空策略对删除操作更有意义,而级联策略看起来对修改操作更为可取。例如,如果某制片公司总裁退休,我们会期望该制片公司将暂时“没有”总裁而存在。但是,对制片公司总裁证书号的修改很可能是一种事务性的变化。人继续存在而且仍是制片公司总裁,所以我们希望 Studio 中的 presC# 属性值将随之变化。

```
1) CREATE TABLE Studio (  
2)     name CHAR(30) PRIMARY KEY,  
3)     address VARCHAR(255),  
4)     presC# INT REFERENCES MovieExec(cert# )  
5)         ON DELETE SET NULL  
6)         ON UPDATE CASCADE  
);
```

图 6.3 选择保持参照完整性的策略

6.2.3 本节练习

练习 6.2.1: 针对电影数据库

悬浮元组和更新策略

外键码值并未出现在被参照关系中的元组称为悬浮元组。回顾一下, 未能参与到连接中的元组也称为“悬浮”的。这两个概念密切相关。如果元组的外键码值在被参照关系中遗漏了, 那么, 该元组就不会参与它所在的关系和被参照关系的连接运算。

悬浮元组恰好是针对外键码约束而与参照完整性相违背的那些元组。

- 默认策略就是, 当且仅当在参照关系中构造出一个或多个悬浮元组时, 对于被参照关系的删除和修改操作将予以禁止。
- 级联策略就是, 删除或修改(分别取决于对被参照关系的更新是删除还是修改) 所有构造出来的悬浮元组。
- 置空策略就是, 把每个悬浮元组中的外键码值都设置为 NULL。

Movie(title, year, length, inColor, studioName, producerC#)

StarsIn(movieTitle, movieYear, starName)

MovieStar(name, address, gender, birthdate)

MovieExec(name, address, cert# , netWorth)

Studio(name, address, presC#)

说明下列参照完整性约束:

- (a) 电影制片人(producerC#)必须是 MovieExec 中提到的某个人。与该约束相违背的对 MovieExec 的更新将予以拒绝。
- (b) 重复(a), 但是违例将导致把 Movie 中的 producerC# 置空。
- (c) 重复(a), 但是违例将导致删除或修改相违背的 Movie 元组。
- (d) 出现在 StarsIn 中的电影也必须出现在 Movie 中。通过拒绝更新来处理违例。
- (e) 出现在 StarsIn 中的影星也必须出现在 MovieStar 中。通过删除相违背的元组来处理违例。

* ! 练习 6.2.2: 我们希望说明这样的约束: 在关系 Movie 中的每部电影都必须伴随 StarsIn 中的至少一名影星出现。我们能通过外键码约束来实现吗? 为什么行, 或为什么不行?

练习 6.2.3: 基于练习 4.1.3 中的数据库模式

Classes(class, type, country, numGuns, bore, displacement)

Ships(name, class, launched)

Battles(name, date)

Outcomes(ship, battle, result)

写出下列参照完整性约束。对键码进行合理的假设并且通过设置参照的属性值为空来处理所有的违例。

- * (a) Ships 中提到的每个等级都必须在 Classes 中提到。
- (b) Outcomes 中提到的每个战役都必须在 Battles 中提到。
- (c) Outcomes 中提到的每艘舰艇都必须在 Ships 中提到。

6.3 对属性值的约束

我们已经看到了键码约束, 它强制某些属性在关系的所有元组中具有截然不同的值, 我们还看到了外键码约束, 它强制两个关系的属性之间保持参照完整性。现在, 我们将看到第三种重要的约束: 它限制了某些属性分量中出现的值。这些约束可以由如下两者之一来表示:

1. 在关系模式的定义中对属性的约束, 或者
2. 对域的约束, 随后把域说明为上述属性的域。

在 6.3.1 节我们将引入一类简单的对属性值的约束: 属性值不为 NULL 的约束。然后, 在 6.3.2 节我们将涉及到第一类约束的主要形式: 基于属性的 CHECK(检验) 约束。第二类(对于域)的约束在 6.3.3 节讨论。我们将在 6.4 节看到其他更通用的约束类型。这些约束如同对单一属性值的约束一样, 能够用来限制整个元组甚至整个关系或几个关系的变化。

6.3.1 非空约束

和某个属性相关的一类简单约束就是 NOT NULL。其效果就是不接受该属性为空的元组。该约束通过在 CREATE TABLE 语句中属性说明后面的关键字 NOT NULL 来说明。

例 6.5 假设关系 Studio 要求 presC# 不为 NULL, 可以通过将图 6.3 的 4) 行改为

4) presC# INT REFERENCES MovieExec(cert#) NOT NULL

来实现。这种改变导致几种后果。例如:

- 我们不能将元组的 presC# 分量的值修改为 NULL。
- 我们不能仅指定名称和地址就把元组插入到 Studio 中, 这是由于插入的元组在 presC# 分量上为 NULL。
- 我们不能像图 6.3 的 5) 行的情况那样采用置空策略, 5) 行的情况告诉系统通过使 presC# 为 NULL 来改正外键码的违例。

6.3.2 基于属性的 CHECK 约束

更复杂的约束可以在属性说明中附加上关键字 CHECK(检验), 随后加上括号内的条件, 该属性的每个值必须满足该条件。实际上, 基于属性的 CHECK 约束就像是对值(比如, 枚举的合法值或者算术不等式)的简单限制。然而, 原则上, 条件可以是 SQL 查询中跟在 WHERE 后面的任何内容。该条件可能引用所约束的属性。但是, 如果它引用其他任何关系或关系的属性, 那么, 必须在子查询的 FROM 子句中引入该关系(即使所引用的关系是所检验的属性所在的关系)。

每当任何元组得到某个属性新的值时, 就对基于该属性的 CHECK 约束进行检验。可以通过对元组的修改而引入新值, 或者把新值作为插入元组的一部分。如果新值与约束相违背, 那么就拒绝更新。如果数据库更新并没有改变同约束相关的属性值, 而且这种限制会导致与约束相违背, 那么就不必检验基于该属性的 CHECK 约束, 如同我们将在例 6.7

中看到的那样。首先, 让我们考虑一个基于属性检验的简单例子。

例 6.6 假设我们要求证书号至少为 6 位。我们可以将图 6.3 的关系

```
Studio(name, address, presC# )
```

的模式说明的 4) 行修改为

```
4) presC# INT REFERENCES MovieExec(cert# )
CHECK (presC# >= 100000)
```

对于另一个例子, 关系

```
MovieStar(name, address, gender, birthdate)
```

的属性 gender 在图 5.13 中说明为数据类型 CHAR(1), 即单个字符。然而, 实际上我们期望出现在那里的字符仅为 'F' 和 'M'。用下面内容来替代图 5.13 的 4) 行即可。

```
4) gender CHAR(1) CHECK (gender IN ( 'F' , 'M' )),
```

上述条件用到一个显式的二值关系, 并表明任何 gender 分量的值必须在此集合之中。

允许在要检验的条件中提到该关系的其他属性或元组, 甚至提到其他关系, 但要这样做, 就需要在条件中有子查询。正像我们所说的那样, 条件可以是 select-from-where 这样的 SQL 语句中跟在 WHERE 后面的任何内容。但是, 约束的检验只和所讨论的属性相关, 而不是和约束中提到的每个关系或属性都相关。因此, 如果要检验的属性之外的某些元素发生变化, 则条件可以为假。

例 6.7 假设我们可以通过一个要求所参照的值存在的基于属性的 CHECK 约束来模拟参照完整性约束。下面是模拟这种要求的错误尝试, 其要求是, 关系

```
Studio(name, address, presC# )
```

的元组中的 presC# 值, 必须出现在关系

```
MovieExec(name, address, cert# , netWorth)
```

的某个元组的 cert# 分量中, 假设把图 6.3 的 4) 行替换成

```
4) presC# INT CHECK
    (presC# IN (SELECT cert# FROM MovieExec))
```

该语句是合法的基于属性的 CHECK 约束, 但让我们来看一看其效果。

- 倘若我们试图向 Studio 中插入一个新元组, 而该元组的 presC# 值不是任何电影行政长官的证书号, 则插入被拒绝。
- 倘若我们试图修改某个 Studio 元组的 presC# 分量, 而新值并不是电影行政长官的 cert#, 则修改被拒绝。
- 然而, 如果我们改变 MovieExec 关系, 比如删除某制片公司总裁的元组, 这种改变对于上述的 CHECK 约束却是不可见的。因此, 即使现在违背了 presC# 上基于属性的 CHECK 约束, 该删除操作也是允许的。

我们将在 6.4.2 节看到如何用更强的约束形式正确地表述该条件。

6.3.3 域约束

我们也可以通过说明带有类似约束的域(见 5.7.6 节)并说明该域为属性的数据类型来约束属性值。唯一重要的区别就是当我们试图描述对域值的约束时, 没有该值的称谓。

何时检验约束

在一般情况下, SQL 系统不会允许导致违背约束的数据库更新。但是, 有时需要进行几个相关更新, 其中一个会导致违例而另一个则会予以弥补。例如, 在例 6.3 中我们规定 presC# 为 Studio 的外键码, 将参照 MovieExec 中的 cert#。如果我们想要插入新的制片公司及其总裁, 那么, 首先插入制片公司将会违背外键码约束。

看起来我们可以通过首先向 MovieExec 中插入总裁来解决这个问题。但是, 假设还有一个约束, MovieExec 元组中的证书号必须或者作为制片公司总裁出现或者作为制片人(在 Movie 中)出现。于是, 没有一种顺序正确。

幸运的是, SQL2 赋予我们将约束说明为 DEFERRED(延迟)的能力。于是, 直到“事务”(数据库操作的基本单元, 见 7.2 节)完成, 才会对约束进行检验。我们可以将制片公司及其总裁的两个插入都包括在一个事务中, 于是就避免了这种不合逻辑的约束违例。

而当我们描述对属性的约束时, 有指向该值的属性的名称。SQL2 通过提供指向域值的特殊的关键字 VALUE 解决了这个问题。

例 6.8 我们可以通过

```
CREATE DOMAIN GenderDomain CHAR(1)
      CHECK (VALUE IN ( 'F' , 'M' ));
```

说明仅以两个字符 'F' 和 'M' 为值的域 GenderDomain。于是, 我们就可以修改图 5.13 的 4) 行使其内容为

4) gender GenderDomain,

与此类似, 在例 6.6 中, 我们要求属性 presC# 为 6 位的证书号, 通过把域说明为

```
CREATE DOMAIN CertDomain INT
      CHECK(VALUE >= 100000);
```

可以得到同样的效果。如果我们写出属性 presC# 的说明

4) presC# CertDomain REFERENCES MovieExec(cert#)

就可以得到所要求的约束。

6.3.4 本节练习

练习 6.3.1: 对于关系

Movie(title, year, length, inColor, studioName, producerC#)

中的属性, 写出下列约束

- * (a) 年份不能在 1895 年以前。
- (b) 电影长度不能短于 60, 也不能长于 250。
- * (c) 制片公司名只能为迪斯尼(Disney)、福克斯(Fox)、米高梅(MGM) 或派拉蒙(Paramount)。

练习 6.3.2: 针对练习 4.1.1 的实例模式

Product(maker, model, type)
PC(model, speed, ram, hd, cd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)

写出其中属性的下列约束

- (a) 便携式电脑的速度必须至少为 100。
- (b) CD 的速度只能为 4, 6, 8 或 12 倍速。
- (c) 打印机的类型只能为激光(laser)、喷墨(ink-jet)和干式(dry)。
- (d) 产品类型只能为 PC 机、便携式电脑和打印机。
- (e) PC 机 RAM 的容量至少为其硬盘容量的 1% 。

6.4 全局约束

现在, 我们将对涉及到几个属性甚至几个不同关系之间的联系这类更加复杂的约束进行说明。该题目分成两个部分:

- 1. 基于元组的 CHECK 约束, 用于限制单个关系中元组的诸方面;
- 2. 断言, 即涉及到整个关系或覆盖同一关系的几个元组变量的约束。

6.4.1 基于元组的 CHECK 约束

我们用 CREATE TABLE 语句定义单个表 R 时, 要想对该表内的元组说明约束, 可以在属性表和键码或外键码的说明之后加上关键字 CHECK 和随后括起的条件。该条件可以是能出现在 WHERE 子句中的任何内容。把它看作表 R 中元组的条件。但是, 像基于属性的约束那样, 该条件可以在子查询、其他关系或同一关系 R 中的其他元组中提到。

每当把元组插入到 R 中或者修改 R 中的元组时, 就对基于元组的 CHECK 约束中的条件进行检验。对于所有的新元组或者修改的元组都要对条件求值。如果对某个元组为假, 则违背了约束并拒绝导致违例的插入或修改操作。但是, 如果条件涉及到子查询中的某个关系(即使是 R 本身), 并且该关系的改变导致对于 R 的某个元组条件变为假, 则该检验并不阻止这种改变。也就是说, 像例 6.7 中讨论的基于属性的 CHECK 一样, 基于元组的约束对于其他关系也是不可见的。

尽管基于元组的检验可以涉及到一些非常复杂的条件, 但通常最好将复杂的检验留给 SQL 的“断言”来解决, 关于断言我们将在 6.4.2 节讨论。就像我们在上面讨论的那样, 其原因是在特定条件下, 可能会违背基于元组的检验。然而, 如果基于元组的检验仅涉及到要检验的元组的属性并且元组没有子查询, 那么约束总是有效的。这里有一个涉及到一个元组中几个属性的基于元组的 CHECK 约束的简单例子。

例 6.9 回顾一下例 5.32, 在那里我们说明了表 MovieStar 的模式。图 6.4 重复了 CREATE TABLE 语句, 并附加了用关键字 UNIQUE 说明的键码以及一个额外的约束, 它是我们可能希望检验的几种可能的“一致性条件”之一。该约束指出如果影星的性别为男性, 那么, 其姓名前面绝对不能加上 'Ms.' 。

正确书写约束

许多约束都和例 6.9 的情形相类似,在例中我们想要禁止满足两个或更多条件的元组。符合检验要求的表达式是对每个条件否定(即取反)的或(OR)。因此,例 6.9 中第一个条件是影星为男性,于是我们用 `gender = 'F'` 作为适当的否定(尽管 `gender = 'M'` 或许是表示否定的更普通的方式)。第二个条件是姓名前面带有 'Ms.', 于是我们用 `NOT LIKE` 这种比较运算作为它的否定。该比较运算对条件本身取反,而条件在 SQL 中则为 `name LIKE 'Ms. %'`。

```
1) CREATE TABLE MovieStar(  
2)     name CHAR(30) UNIQUE,  
3)     address VARCHAR( 255),  
4)     gender CHAR(1),  
5)     birthdate DATE,  
6)     CHECK (gender = 'F' OR name NOT LIKE 'Ms. %' )  
       );
```

图 6.4 表 MovieStar 的约束

在 2) 行,说明 `name` 为关系的键码。接下来 6) 行说明一个约束。对于所有的女性影星以及姓名前面不加 'Ms.' 的所有影星,该约束条件均为真。只有对性别为男性而姓名前面却带有 'Ms.' 的元组,条件不为真。那些恰恰是我们希望从 MovieStar 中排除的元组。

6.4.2 断言

我们已经将对属性的约束扩展为对元组的约束。但即使这些形式有时也还不够。有时我们需要的约束涉及到作为整体的关系,例如对一系列中的值求和或者进行其他聚合运算的约束。还会用到涉及到多个关系的约束。事实上,外键码约束是我们已经看到的连接两个关系的约束的一个例子,但外键码约束有其局限性。

SQL2 的断言(assertion)(也称为“通用约束”)允许我们施加任何条件(可以跟在 WHERE 后面的表达式)。其他类型的约束同其他模式元素相关,通常基于表或域,而断言本身就是模式元素。

和其他模式元素一样,我们用 CREATE 语句说明断言。断言的格式为:

- 1. 关键字 CREATE ASSERTION,
- 2. 断言的名称,
- 3. 关键字 CHECK, 以及
- 4. 括起的条件。

即,语句的格式为

```
CREATE ASSERTION 名称 CHECK ( 条件 )
```

断言中的条件必须一直为真,任何使其为假的数据库更新操作都会遭到拒绝。回顾

对限定约束的检验: 缺点还是优点?

人们可能会感到奇怪, 如果基于属性和基于元组的检验引用其他关系或同一关系的其他元组, 为什么就允许违约? 其原因是, 实现这样的约束比实现断言更有效。用基于属性或基于元组的检验, 只要对插入或修改的元组的约束进行判断。反之, 每当断言中提到的任何一个关系发生变化, 都必须对断言进行判断。这样额外的判断使数据库更新增加了运行时间, 这样做是否值得, 数据库设计者要对此作出评估。然而, 为了代码的长期可靠性, 我们建议, 设计者不用可能违约的基于属性或基于元组的检验。

一下, 我们已经研究过的其他类型的 CHECK 约束, 在某些条件下只要涉及到子查询, 就会出现违例。

我们书写基于元组的 CHECK 约束和书写断言的方法有所不同。基于元组的检验可以引用关系说明中出现的属性。例如, 在图 6.4 的 6) 行中我们用到了属性 gender 和 name, 而没有指出它们来自何处。由于表 MovieStar 正是在 CREATE 语句中要说明的表, 因此这些属性是指 MovieStar 元组的分量。

断言的条件就没有这种特权了。条件中引用的任何属性都必须在断言中指明, 典型的方式是在 select-from-where 表达式中指出属性所在的关系。由于条件必须为布尔值, 通常以某种方法将条件的结果聚合起来得到单一的真/假选择。例如, 我们可以将条件写成生成某个关系的表达式, 对其应用 NOT EXISTS; 也就是, 约束为该关系总为空。作为选择, 我们可以对关系中的一列用 SUM 这样的聚合运算符并将之与常量相比较。例如, 用这种方法我们可以要求和总是小于某个限定的值。

例 6.10 假设我们要求其净资产没达到 10 000 000 美元的人不能成为制片公司的总裁。为此目的我们说明一个断言, 其总裁净资产不足 10 000 000 美元的电影制片公司的集合为空。此断言涉及到两个关系:

MovieExec(name, address, cert# , netWorth)
Studio(name, address, presC#)

断言如图 6.5 所示。

```
CREATE ASSERTION RichPres CHECK
  (NOT EXISTS
    (SELECT *
      FROM Studio, MovieExec
      WHERE presC# = cert# AND netWorth < 10000000
    )
  );
```

图 6.5 承认富有的制片公司总裁的断言

顺便说一下, 尽管该约束涉及到两个关系, 却毫无价值, 我们可以将它写成针对这两个关系的基于元组的 CHECK 约束, 而不写成单独的断言。例如, 我们可以在例 6.3

约束的比较			
下表列出了基于属性的检验、基于元组的检验和断言之间的主要区别。			
约束类型	说明位置	激活时刻	肯定保持?
基于属性的约束	与属性一起	对关系进行插入或修改属性时	如果有子查询则不保持
基于元组的约束	关系模式的元素	对关系进行插入或修改元组时	如果有子查询则不保持
断言	数据库模式的元素	当提到的任何关系发生改变时	保持

的 CREATE TABLE 语句中加上如图 6. 6 所示的对 Studio 的约束。

```

1) CREATE TABLE Studio(
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     presC# INT REFERENCES MovieExec(cert# ),
5)     CHECK (presC# NOT IN
6)           (SELECT cert# FROM MovieExec
7)           WHERE netWorth < 10000000)
           )
);

```

图 6. 6 和断言相对应的对 Studio 的约束

但是要注意, 只有在关系 Studio 发生变化时才对图 6. 6 中的约束进行检验。它不会检验到这种情形: 已记录在关系 MovieExec 中的某个制片公司总裁的净资产降到 10 000 000 美元以下。为达到断言的全部效果, 我们不得不给表 MovieExec 的说明加上另一个约束, 要求如果行政长官为制片公司总裁, 则净资产至少为 10 000 000 美元。

例 6. 11 这里是断言的另一个例子。它仅影响到关系

```

Movie(title, year, length, inColor, studioName, producerC# )
并且指出给定的制片公司所制作的所有电影的总长度不应超过 10 000 分钟。
CREATE ASSERTION SumLength CHECK ( 10000 > = ALL
    (SELECT SUM(length) FROM Movie GROUP BY studioName)
);

```

该约束碰巧仅涉及到关系 Movie。也可以把它表示成 Movie 模式中的基于元组的 CHECK 约束而不表示成断言。即, 我们可以在表 Movie 的定义中加上基于元组的 CHECK 约束

```

CHECK (10000 > = ALL
    (SELECT SUM(length) FROM Movie GROUP BY studioName));

```

注意, 原则上, 这个条件应用于表 Movie 的所有元组。然而, 它并没有明确地提到元组的任何属性, 而所有的工作都是在子查询中完成的。

还应该注意, 如果作为基于元组的约束来实现, 当从关系 Movie 中删除元组时, 不会进行检验。在本例中, 如果约束在删除操作之前能满足, 那么, 保证删除操作之后也满足, 所以这种区别无关紧要。但是, 如果本例中约束是总长度的下限而不是上限, 那么我们就发现作为基于元组的检验而不是断言所写的约束会发生违例。

6.4.3 本节练习

练习 6.4.1: 我们在例 6.10 中提到, 图 6.6 中的基于元组的 CHECK 约束只能完成图 6.5 中断言所完成的工作的一半。写出完成该项工作所需要的对 MovieExec 的 CHECK 约束。

练习 6.4.2: 对于我们不断滚动的电影实例的如下几个关系之一, 写出下列基于元组的 CHECK 约束。

```
Movie(title, year, length, inColor, studioName, producerC# )
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert# , netWorth)
Studio(name, address, presC# )
```

倘若约束实际上涉及到两个关系, 那么, 你应该把约束放在两个关系中, 从而保证无论哪个关系发生变化, 都将对插入和修改操作的约束进行检验。假定没有删除操作, 那么保留对删除操作的基于元组的约束就不合理了。

- * (a) 1939 年以前制作的电影不会是彩色的。
- (b) 影星不会出现在他们出生之前制作的电影中。
- ! (c) 两个制片公司不会有同一个地址。
- * ! (d) 在 MovieStar 中出现的姓名绝对不能出现在 MovieExec 中。
- ! (e) 出现在 Studio 中的制片公司名称必须至少出现在一个 Movie 元组中。
- !! (f) 如果某电影的制片人同时也是某制片公司的总裁, 那么他或她必须是制作该电影的制片公司的总裁。

练习 6.4.3: 将下列约束表示成 SQL 断言。约束基于练习 4.1.1 中的关系:

```
Product(maker, model, type)
PC(model, speed, ram, hd, cd, price)
Laptop( model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

- * (a) PC 机制造商不可能也制造便携式电脑。
- * ! (b) PC 机制造商必须也制造其处理器速度至少同样快的便携式电脑。
- ! (c) 如果便携式电脑的主存容量比 PC 机大, 那么便携式电脑的价格也一定比 PC 机高。
- !! (d) 在关系 PC, Laptop 和 Printer 中, 型号不可能出现两次。
- !! (e) 如果关系 Product 提到了某型号及其类型, 那么该型号一定出现在适合该类型的关系之中。

练习 6.4.4: 写出下列关于我们的“PC”模式的基于元组的 CHECK 约束。

- (a) 处理器速度不到 150MHz 的 PC 机一定不能以超过 1 500 美元的价格出售。
- (b) 屏幕尺寸不到 11 英寸的便携式电脑必须具有至少 1G 字节的硬盘, 否则价格必须低于 2 000 美元。

练习 6.4.5: 将下列约束表示成 SQL 断言。约束基于练习 4.1.3 中的关系:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

- (a) 每个等级不超过两艘舰艇。
- ! (b) 没有哪个国家能够同时有战列舰和巡洋舰。
- ! (c) 配备超过 9 门火炮的舰艇在与配备不到 9 门火炮的舰艇的交战中不可能被击沉。
- ! (d) 没有哪艘舰艇能够在具有该等级名称的那艘舰艇下水之前下水。
- ! (e) 对于每个等级, 都有一艘以等级命名的舰艇。

练习 6.4.6: 对于我们的“战列舰”模式, 将下列约束写成基于元组的 CHECK 约束。

- (a) 没有什么等级的舰艇其火炮口径超过 16 英寸。
- (b) 如果某等级的舰艇具有 9 门以上的火炮, 那么, 其火炮口径一定不超过 14 英寸。
- ! (c) 没有哪艘舰艇能够在下水之前参加战役。

6.5 约束的更新

可以在任何时候增加、修改或删除约束。描述这种更新的方法取决于所涉及的约束是和域、属性、表还是数据库模式有关。

6.5.1 对约束命名

为了修改或删除已经存在的约束, 约束需要有个名称。作为数据库模式的一部分, 断言的命名总是作为其 CREATE ASSERTION 语句的一部分。然而, 也可以为其他约束命名。为此, 可以在约束之前加上关键字 CONSTRAINT 和该约束的名称。

例 6.12 我们也可以对主键码或外键码的说明命名。例如, 我们可以重写图 6.1 的 2) 行, 以便对指出属性 name 为主键码的约束命名, 像这样:

```
2) name CHAR(30) CONSTRAINT NameIsKey PRIMARY KEY,
```

同样, 我们可以通过

```
4) gender CHAR(1) CONSTRAINT NoAndro
    CHECK (gender IN ( 'F' , 'M' ))
```

为出现在例 6.6 中的基于属性的 CHECK 约束命名。

例 6.8 中的域约束可以命名为:

```
CREATE DOMAIN CertDomain INT
    CONSTRAINT SixDigits CHECK (VALUE >= 100000);
```

为约束命名

记住, 为你写的每个约束命名是一个好主意, 即使你并不相信会用到它。一旦建立了没有名称的约束, 以后当你想以任何方式更改它时再命名就太晚了。

最后, 下面的约束:

6) CONSTRAINT RightTitle

CHECK (gender = 'F' OR name NOT LIKE 'Ms. %')

是为给约束命名而对图 6.4 中 6) 行的基于元组的 CHECK 约束的改写。

6.5.2 更改表的约束

我们可以用 ALTER 语句来更改与域、属性或表有关的约束的集合。在 5.7.4 节我们讨论了 ALTER TABLE 语句的某些用法, 在那里是用于增加或删除属性。与此类似, 我们在 5.7.6 节讨论了 ALTER DOMAIN, 用它来改变默认值。

这些语句也能用来更改约束; ALTER TABLE 既用于基于属性的检验也用于基于元组的检验。我们可以用关键字 DROP 和要撤消的约束名来撤消约束。我们也可以用关键字 ADD 随后加上要增加的约束名来增加约束。

例 6.13 让我们看看如何撤消和增加例 6.12 中的约束。首先, 可以通过

ALTER TABLE MovieStar DROP CONSTRAINT NameIsKey;

把表明 name 为关系 MovieStar 的主键码的约束撤消。

同一关系中, 限制属性 gender 取值的基于属性的 CHECK 约束, 可以通过

ALTER TABLE MovieStar DROP CONSTRAINT NoAndro;

来撤消。

另外, 关系 MovieStar 中限于女影星的称呼 'Ms.' 这一约束可以通过

ALTER TABLE MovieStar DROP CONSTRAINT RightTitle;

来撤消。

如果我们想要恢复这些约束, 可以通过增加同样的约束来更改关系 MovieStar 的模式, 例如:

ALTER TABLE MovieStar ADD CONSTRAINT NameIsKey

PRIMARY KEY (name);

ALTER TABLE MovieStar ADD CONSTRAINT NoAndro

CHECK (gender IN ('F' , 'M'));

ALTER TABLE MovieStar ADD CONSTRAINT RightTitle

CHECK (gender = 'F' OR name NOT LIKE 'Ms. %')

现在这些约束是基于元组的, 而不是基于属性的。尽管属性类型为域, 我们可以用更改域的约束来代替更改 MovieStar 表, 却无法使之恢复为属性约束。

这些重新引入的约束其名称是任选的。但是, 我们不能依赖 SQL 来记住同约束名相关的约束。因此, 当我们增加的是以前的约束时, 还需要重新写出约束; 不能仅通过名称来

引用它。

6.5.3 更改域的约束

从本质上来说, 撤消和增加关于域的约束同撤消或增加基于元组的检验, 其方法是一样的。要想撤消对域的约束, 用 ALTER 语句, 其中关键字 DROP 后面加上约束名。要想增加对域的约束, ALTER 语句中要有关键字 ADD、约束名以及定义约束的 CHECK 条件。

例 6.14 证书号至少有六位, 该域约束可以通过

```
ALTER DOMAIN CertDomain DROP CONSTRAINT SixDigits;
```

来撤消。

反之, 我们可以通过

```
ALTER DOMAIN CertDomain ADD CONSTRAINT SixDigits
CHECK (VALUE >= 100000);
```

来恢复该约束。

6.5.4 更改断言

要撤消断言可以在关键字 DROP ASSERTION 后面加上断言名。

例 6.15 例 6.10 中的断言可以通过语句

```
DROP ASSERTION RichPres;
```

来撤消。要想恢复该约束, 像在例 6.10 中所作的那样对其重新说明。

6.5.5 本节练习

练习 6.5.1: 说明如何用下列方法更改电影实例中的关系模式:

```
Movie(title, year, length, inColor, studioName, producerC# )
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert# , netWorth)
Studio(name, address, presC# )
```

- * (a) 使 title 和 year 成为 Movie 的键码。
- (b) 要求的参照完整性约束是, 每部电影的制片人都出现在 MovieExec 中。
- (c) 要求电影的长度不能短于 60 也不能长于 250。
- * ! (d) 要求一个名字不能同时作为影星和电影行政长官出现(该约束在删除的情况下不必保持)。
- ! (e) 要求两个制片公司不能有相同的地址。

练习 6.5.2: 说明如何更改“ 战列舰 ”数据库模式

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

来得到下列基于元组的约束。

- (a) 等级和国家构成关系 Classes 的键码。
- (b) 要求的参照完整性约束是, 出现在 Battles 中的每艘舰艇也都出现在 Ships 中。
- (c) 要求的参照完整性约束是, 出现在 Outcomes 的每艘舰艇也都出现在 Ships 中。
- (d) 要求没有舰艇配置 14 门以上的火炮。
- ! (e) 不允许舰艇在下水之前参战。

6.6 SQL3 中的触发程序

本章中我们所研究的不同形式的约束都遵循 SQL2 标准。它们有其执行模式, 每当它们所约束的元素发生变化时就按其执行模式调用这些约束。例如, 每当某个元组中的某个属性发生变化时(包括插入元组的情况), 就调用基于该属性的检验。

由于约束的实现涉及到对相应事件的检验进行“触发”, 所以, 人们很自然地会问, 能否由数据库编程人员而不是由系统来选择触发事件。这种方法会给用户某些附加的选项, 以便有目的地触发数据库操作, 而不是防止出现约束的违例。因此, SQL3 的推荐标准还包括了“触发”, 触发使人联想到约束, 但触发程序要明确指定触发事件, 并明确指定基于条件结果而要做的动作。有趣的是, 目前的商业系统在其强大的主动性元素方面往往更接近 SQL3 而不是 SQL2。原因可能是对商业开发商来说, 在某种意义上, 触发程序比断言更容易实现。

6.6.1 触发和约束

触发(trigger)有时也称为事件-条件-动作规则(event-condition-action rules)或 ECA 规则, 在三个方面不同于前面讨论的约束类型。

1. 当数据库编程人员所指定的某些事件发生时才对触发程序进行测试。允许的事件种类通常为对特定关系的插入、删除或修改。在许多 SQL 系统中另一种允许的事件为事务结束(参见 7.2 节对用于批量数据库操作的称为事务的原子工作单元的讨论)。

2. 不是直接阻止事件的发生, 而是由触发程序对条件进行测试。如果条件不满足, 则什么也不做, 否则, 为响应该事件就会进行与该触发相关的处理。

3. 如果触发条件得到满足, 就由 DBMS 执行与该触发相关的动作。于是该动作可能阻止事件的发生或撤消事件(如删除插入的元组)。实际上, 动作可能是数据库操作的任何序列, 甚至可能是和触发事件毫无关联的操作。

下面, 我们将首先考虑 SQL3 中的触发程序。然后, 我们将简单地讨论对于 SQL2 中称为“断言”的约束在 SQL3 中的扩展。这些约束也含有触发的某些方面。

6.6.2 SQL3 触发程序

SQL3 触发语句在事件、条件和动作部分给用户许多不同的选项。这里是主要特点。

1. 动作可以在触发事件之前、之后执行, 甚至可以不用触发事件而执行。
2. 动作可以引用在触发该动作的事件中插入、删除或修改的元组的旧值或新值。
3. 修改事件可以指定特定的属性或属性集。

4. 条件可以通过 WHEN 子句指定,而只有在对规则进行触发并且当触发事件发生时条件满足的情况下动作才会执行。

5. 编程人员可以对规定执行的动作进行选择:

- (a) 对于每个更新的元组都执行一次,或
- (b) 对于在一个数据库操作中发生变化的所有元组执行一次。

在给出触发程序的语法细节之前,让我们研究一个能阐明最重要的语法以及语义特点的例子。在本例中,对于每个修改的元组都执行一次触发程序。

例 6.16 我们将写出应用于表

MovieExec(name, address, cert# , netWorth)

的 SQL3 触发程序。触发程序是由对 netWorth 属性的修改而启动的。该规则的作用是对降低电影行政长官净资产的任何尝试加以阻止。触发程序的说明如图 6.7 所示。

```
1) CREATE TRIGGER NetWorthTrigger
2) AFTER UPDATE OF netWorth ON MovieExec
3) REFERENCING
4)     OLD AS OldTuple,
5)     NEW AS NewTuple
6) WHEN(OldTuple.netWorth > NewTuple.netWorth)
7)     UPDATE MovieExec
8)     SET netWorth = OldTuple.netWorth
9)     WHERE cert# = NewTuple.cert#
10) FOR EACH ROW
```

图 6.7 SQL3 的触发程序

1)行给出具有关键字 CREATE TRIGGER 和触发程序名的说明。然后,2)行给出了触发事件,即修改关系 MovieExec 中的属性 netWorth。3)到 5)行是为该触发程序的条件和动作部分如何引用旧元组(修改前的元组)和新元组(修改后的元组)提供一种方法。根据 4)行和 5)行中的说明,将用 OldTuple 和 NewTuple 分别引用这两个元组。在条件和动作部分,这些名称可以像在普通 SQL 查询的 FROM 子句中说明的元组变量一样使用。

6)行是触发程序的条件部分。它表明只有在新的净资产低于旧的净资产(也就是某行政长官的净资产缩减)时才执行动作。

7)行到 9)行构成动作部分:它们是普通的 SQL 修改语句,而具有的功能是将该行政长官的净资产恢复到修改以前的值。注意,原则上,会考虑每个 MovieExec 元组,但是 9)行的 WHERE 子句保证只影响到修改的元组(具有特定 cert# 的元组)。

最后,10)行表明了要求,即每当修改元组,该触发程序都会启动一次。如果没有这一行,那么,一个 SQL 语句使触发程序执行一次而无论发生多少改变元组的触发事件。

当然,例 6.16 仅说明了 SQL3 触发程序的某些特点。在下面的要点中,我们将概述触发程序提供的选项以及如何描述这些选项。

- 图 6.7 中的 2)行如关键字 AFTER 所指,其规则的动作将在触发事件之后执行。可供选择来代替 AFTER 的有

- (a) BEFORE。WHEN 中的条件在触发事件之前检验。如果条件为真,则执行触

发程序的动作。此外, 无论条件是否为真, 都将执行触发修改的事件。

(b) INSTEAD OF。(如果符合 WHEN 中的条件)会执行动作, 而永远不会执行触发事件。

- 除 UPDATE 以外, 其他可能的触发事件有 INSERT 和 DELETE。图 6.7 的 2) 行中的 OF netWorth 子句是 UPDATE 事件的选项, 并且现在假定把事件仅定义为修改关键字 OF 后面列出的属性。OF 子句不支持 INSERT 或 DELETE 事件; 这些事件只对整个元组有意义。
- 尽管我们用单个的 SQL 语句来表示动作, 其实可以有任意数量用分号分开的这类语句。
- 当触发事件为修改时, 有旧元组和新元组, 分别为修改前后的元组。我们用在 4) 行和 5) 行看到的 OLD AS 和 NEW AS 子句为这些元组命名。如果触发事件是插入, 那么我们可以用 NEW AS 子句为插入的元组命名, 而 OLD AS 则不予接受。相反, 对于删除操作, 将用 OLD AS 为删除的元组命名, 而 NEW AS 则不予接受。
- 倘若我们略掉 10) 行中的 FOR EACH ROW, 那么像图 6.7 这样的行级触发程序 (row-level trigger) 就变成了语句级的触发程序。对于生成一个或多个触发事件的一个语句, 语句级的触发程序只执行一次。例如, 如果我们用 SQL 的修改语句修改整个表, 那么, 语句级的修改触发程序只执行一次, 而元组级的触发程序对每个元组都执行一次。在语句级的触发程序中, 我们不能像在 4) 和 5) 行所做的那样直接引用旧的或新的元组。相反, 我们可以将旧元组的集合(删除的元组或修改的元组的旧版本)和新元组的集合(插入的元组或修改的元组的新版本)作为两个关系来引用。我们用诸如 OLD- TABLE AS OldStuff 或 NEW- TABLE AS NewStuff 这样的说明来代替图 6.7 中 4) 行和 5) 行的说明。像上面定义的那样, OldStuff 命名了包含所有旧元组的关系, 而 NewStuff 则指向包含所有新元组的关系。

例 6.17 假设我们想要防止电影行政长官的平均净资产降到 500 000 美元以下。对如下关系

MovieExec(name, address, cert# , netWorth)

中 netWorth 列的插入、删除或修改操作可能会违背该约束。我们需要为这三个事件中的每一个写一个触发程序。图 6.8 描述了修改事件的触发程序。插入和删除元组的触发程序与此类似, 但稍微简单一些。

3) 到 5) 行说明 NewStuff 和 OldStuff 是关系名, 这两个关系包含触发我们的规则的数据库操作所涉及到的新元组和旧元组。注意, 一个数据库语句可以修改一个关系中的许多元组, 而如果执行了这样的语句, 那么, 在 NewStuff 和 OldStuff 中就会有许多元组。

如果是修改操作, 那么, NewStuff 和 OldStuff 分别为修改的元组的新版本和旧版本。如果为删除操作写出类似的触发程序, 那么, 删除的元组应该在 OldStuff 中, 而不会有像该触发程序中对应于 NEW- TABLE 的 NewStuff 这样的关系名的说明。同样, 在插入操作的类似的触发程序中, 新元组在 NewStuff 中, 而不会有 OldStuff 的说明。

6) 到 8) 行是条件。如果修改后的平均净资产至少为 500 000 美元, 则条件满足。注意, 8) 行中的表达式计算的是如果修改操作已经做完的 MovieExec 关系。

```

1) CREATE TRIGGER AvgNetWorthTrigger
2) INSTEAD OF UPDATE OF netWorth ON MovieExec
3) REFERENCING
4)         OLD- TABLE AS OldStuff
5)         NEW- TABLE AS NewStuff
6) WHEN ( 500000 < =
7)         ( SELECT AVG(netWorth)
8)         FROM((MovieExec EXCEPT OldStuff) UNION NewStuff)
9) )
9) DELETE FROM MovieExec
10) WHERE(name, address, cert# , netWorth) IN OldStuff;
11) INSERT INTO MovieExec
12)         (SELECT * FROM NewStuff);

```

图 6.8 对平均净资产的约束

但是, 由于 2) 行中规定了 INSTEAD OF, 任何对 MovieExec 的 netWorth 列进行修改的尝试都将截取下来。永远不会执行修改操作。作为替代, 触发程序利用其条件来判断该做什么。在我们的实例中, 如果修改操作保留电影行政长官的平均净资产至少五十万, 那么动作就会得到修改操作所期望得到的效果。也就是说, 9) 行和 10) 行删除修改操作想要修改的元组, 而 11) 行和 12) 行则插入这些元组的新版本。

6.6.3 SQL3 的断言

SQL3 还在两个重要的方面扩展了 SQL2 断言。

1. 由程序员规定的事件触发断言, 而不是由系统决定、可能和约束相违背的事件触发断言。

2. 断言就像元组级的检验, 可以任意指向表中的每个元组, 而不是指向作为整体的表或多个表。

例 6.18 用 SQL3 的表示法, 例 6.10 中的 RichPres 断言如图 6.9 所示。像通常那样, 1) 行开始说明。在 2) 到 6) 行, 我们看到可以触发对断言进行检验的多个事件。

```

1) CREATE ASSERTION RichPres
2) AFTER
3)     INSERT ON Studio,
4)     UPDATE OF presC# ON Studio,
5)     UPDATE OF netWorth ON MovieExec,
6)     INSERT ON MovieExec
7) CHECK (NOT EXISTS
8)         (SELECT * FROM Studio, MovieExec
9)         WHERE presC# = cert# AND netWorth < 10000000
10) )

```

图 6.9 SQL3 断言

回顾一下, 为了截取与 7) 到 9) 行的约束相违背的对数据库所有可能的改变, 我们需

要注视新任的制片公司总裁或者某个行政长官净资产的变化。因此,每当插入 Studio 元组或者修改制片公司总裁的证书号(也就是总裁人选发生变化)时,3)行和4)行将导致对断言的检验。当修改任何行政长官的净资产或者插入某个行政长官时,两种情况中的任何一种都可能导致约束为假,这时5)行和6)行就会触发检验。要检验的约束位于7)行到9)行,而在本质上和例6.10是相同的。

SQL3 和 SQL2 在断言方法上的主要区别在于,当需要进行检验时图6.9将断言显式地表示出来。这种情形使 SQL3 断言对系统实现者更加容易,但对如下的用户就比较困难了:

- 1. 必须发现所有可能触发约束的事件;
- 2. 当事件选择不恰当时,甘于冒允许数据库进入不一致状态的风险。

6.6.4 本节练习

练习 6.6.1: 为 MovieExec 的插入和删除事件写出与图6.8相似的 SQL3 触发程序。

练习 6.6.2: 基于练习4.1.1中的“PC”实例:

```
Product(maker, model, type)
PC(model, speed, ram, hd, cd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

写出下列 SQL3 触发程序或断言:

- * (a) 当修改 PC 机的价格时,检验没有速度相同而价格更低的 PC 机。
- (b) 当插入新的打印机时,检验其型号存在于 Product 中。
- ! (c) 当对 Laptop 关系进行任何更新时,检验每个厂商的便携式电脑的平均价格至少为 2 000 美元。
- ! (d) 当对任何 PC 机的 RAM 或硬盘进行修改时,检验修改的 PC 机的硬盘至少为 RAM 容量的 100 倍。
- ! (e) 当插入新的 PC 机、便携式电脑或打印机时,保证该型号以前没有出现在 PC, Laptop 或 Printer 中。

练习 6.6.3: 基于练习4.1.3中的数据库模式

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

写出一个或多个 SQL3 触发程序或断言来完成下列要求:

- * (a) 当向 Classes 中插入新的等级时,还要插入一艘以等级命名而下水日期为 NULL 的舰艇。
- (b) 当插入排水量大于 35 000 吨的新等级时,允许插入,但要将排水量改为 35 000。
- ! (c) 如果把元组插入到 Outcomes 中,应检验舰艇和战役分别列在 Ships 和 Battles 中,如果没有,则将元组插入到这两个关系中或其中之一,必要的分量

为 NULL 值。

! (d) 当向 Ships 插入或者对 Ships 的 class 属性进行修改时, 检验没有国家具有 20 艘以上的舰艇, 否则取消插入。

!! (e) 在所有可能导致违例的情况下, 检验没有舰艇能够参加比导致该舰艇沉没的那场战役更晚的一场战役。否则防止更新操作的发生。

! 练习 6.6.4: 将下列要求写成合适的 SQL3 触发程序或者 SQL3 断言。问题基于我们不断滚动的电影实例:

```
Movie(title, year, length, inColor, studioName, producerC# )
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert# , netWorth)
Studio(name, address, presC# )
```

可以假定在尝试对数据库做任何改变之前, 要求的条件都成立。同样, 即使意味着插入带有 NULL 值或默认值的元组, 也宁愿更新数据库而不愿拒绝更新尝试。

- (a) 确保在任何时候, 出现在 StarsIn 中的任何影星也出现在 MovieStar 中。
- (b) 确保在任何时候, 所有的电影行政长官或者作为制片公司总裁出现, 或者作为电影制片人出现, 或者作为二者出现。
- (c) 确保所有的电影都至少有一个男影星和一个女影星。
- (d) 确保任何制片公司在任何年份制作的电影数量都不超过 100。
- (e) 确保在任何年份制作的所有电影的平均长度都不超过 120。

! 练习 6.6.5: 在例 6.17 中处理不利的更新的方法是, 首先检验, 然后更新——如果它不和条件相违背的话。另一种方法是允许更新, 然后复原——如果它和条件相违背的话。写出该触发程序。

6.7 本章总结

键码约束: 我们可以用关系模式中的 UNIQUE 或 PRIMARY KEY 说明来说明属性或属性集为键码。

参照完整性约束: 我们可以用关系模式中的 REFERENCES 或 FOREIGN KEY 说明来说明出现在某个属性或属性集中的值也必须出现在另一个关系某个元组的主键码属性中。

基于属性的检验约束: 我们可以在关系模式中某个属性说明的后面加上关键字 CHECK 和要检验的条件来检验对该属性值的约束。作为选择, 我们可以将域作为属性类型并在域的说明中指定要检验的条件。

基于元组的检验约束: 我们可以在关系本身的说明中加上关键字 CHECK 和要检验的条件来检验关系中元组的一些或所有分量的条件。

断言: 我们可以用关键字 CHECK 和要检验的条件来说明断言为数据库模式的元素。该条件可能涉及到数据库模式的一个或多个关系, 还可能涉及到作为整体的

关系(例如,聚合)以及单个元组的条件。

检验的调用:每当断言所涉及的关系之一发生变化而使得可能和约束发生违背时,就对断言进行检验。仅当对元组的插入或修改操作使得基于值的和基于元组的检验约束所作用的属性或关系发生变化时,才对约束进行检验。因此,如果约束有涉及到其他关系或同一关系的其他元组的子查询时,就可能违背这些约束。

SQL3 触发程序:SQL3 的推荐标准包含详细说明某些事件(如对特定关系的插入、删除或修改)的触发程序,而这些事件将启动触发程序。一旦启动触发程序,就对条件进行检验,如果为真,则执行特定的动作序列(诸如查询和数据库更新之类的 SQL 语句)。

SQL3 断言:SQL3 标准包含了一种不同于 SQL2 断言的断言概念。像 SQL3 触发程序一样,这些断言将为一个或多个诸如向关系中进行插入之类的事件所启动。一旦启动,SQL3 断言就将检验关系或元组的条件,若条件不满足,则拒绝更新。

6.8 本章参考文献

读者应该回到第 5 章的文献目录评述中以得到如何获取 SQL2 或 SQL3 标准文档的信息。参考文献[4]是关于数据库系统中主动性元素所有情况的信息源。[1]讨论了关于 SQL3 和未来标准中主动性元素的新近想法。参考文献[2]和[3]讨论了一个提供主动性数据库元素的早期原型系统 HiPAC。

- [1] Cochrane, R. J., H. Pirahesh and N. Mattos, Integrating triggers and declarative constraints in SQL database systems. Intl. Conf. on Very Large Database Systems, pp. 567 ~ 579, 1996.
- [2] Dayal, U., et al., The HiPAC project: combining active databases and timing constraints. SIGMOD Record, 17: 1, pp. 51 ~ 70, 1988.
- [3] McCarthy, D. R., and U. Dayal, The architecture of an active database management system, Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 215 ~ 224, 1989.
- [4] Widom, J., and S. Ceri, Active Database Systems, Morgan Kaufmann, San Francisco, 1996.

第 7 章 SQL 系统概况

现在我们来谈谈如何把 SQL 用于完整的编程环境中。下面提到的每个问题,都将遵循 SQL2 标准。在 7.1 节,我们将看到 SQL 通常用于由一般编程语言(例如 C)编写的程序中。SQL 的许多特性允许我们在它内在的关系和外在的宿主语言变量之间传递数据。

接下来,7.2 节介绍“事务”——工作的原子单位。许多数据库的应用,例如银行业务,即使可能同时进行大量的并发操作,也需要对数据的操作表现出原子性即不可分性。SQL 提供的特性允许我们来描述事务,并且 SQL 系统具有这样的机制来保证事务操作能够真正原子化地执行。

7.3 节涉及到其他的系统问题,例如,对客户程序/服务程序计算模型的支持。接下来,7.4 节将讨论 SQL 如何控制对数据的非法访问,以及我们如何告诉 SQL 系统什么样的访问是合法的。

7.1 编程环境中的 SQL

到目前为止,我们在例子中都在使用直接的 SQL。也就是说,我们假定已经有一个 SQL 解释程序,它接受和执行各种已经学过的 SQL 查询和命令。这种操作方式现实中很少使用。实际上,大多数 SQL 语句都是某种更大的程序或者函数集的一部分。一个更现实的方式是用某种传统的宿主语言(例如 C)编写程序,但是该程序中的某些函数或者 C 程序中的某些语句实际上是 SQL 语句。在这一节,我们将叙述在传统的程序内部进行 SQL 操作的方法。

包括 SQL 语句的典型编程系统的示意图如图 7.1 所示。在这里我们可以看到程序员用宿主语言来编写程序,但是其中某些特殊的嵌入式 SQL 语句不是宿主语言的一部分。首先把整个程序送到预处理程序,该预处理程序把嵌入式 SQL 语句转换成在宿主语言中有意义的内容。

图 7.1 处理具有嵌入式 SQL 语句的程序

SQL 语句的这种表示法能够简化成函数调用,该函数调用把 SQL 语句作为字符串参数并且执行该 SQL 语句。我们还在图 7.1 中表示了程序员直接用宿主语言写程序的可能性,如果需要就使用这些函数调用。

SQL2 标准支持的语言

SQL2 的实现至少要支持以下七种宿主语言: ADA, C, Cobol, Fortran, M(以前称作 Mumps), Pascal 和 PL/I。可能除了 M(即 Mumps, 一种主要用于医学界的语言)以外, 学习计算机科学的学生应该熟悉以上的每种语言。在我们的例子中将使用 C。

然后, 预处理过的宿主语言以通常的方式编译。数据库管理系统供应商通常提供函数库以提供必要的函数定义。这样, 就能执行实现 SQL 的函数, 而整个程序则表现为一个整体。

7.1.1 匹配失衡问题

连接 SQL 语句和传统编程语言的一个基本问题是匹配失衡, 就是说 SQL 的数据模型与其他语言的数据模型差别非常大。我们知道, SQL 使用关系数据模型作为它的核心。然而, C 和其他普通的编程语言使用的数据模型具有整数、实数、算术运算、字符、指针、记录结构、数组等等。C 和其他一些语言不能直接表示集合, 而另一方面, SQL 又不直接使用指针、数组或者其他普通的编程语言结构。从而 SQL 和其他语言之间的转换不是直接的, 所以要研究一种机制允许同时使用 SQL 和另一种语言来开发程序。

首先一个可能的假设是最好使用单一的语言, 或者用 SQL 完成所有的计算, 或者忘记 SQL 而用传统的语言完成所有的计算。然而, 当涉及到数据库操作时我们就会很快抛弃无视 SQL 的想法。SQL 系统可以给程序员编写既高效率执行又高级别表达的数据库操作以极大的帮助。SQL 可以减少程序员的如下工作: 了解数据在存储器中是如何组织的或者如何利用存储结构来有效地操作数据库。

在另一方面, 有许多重要的事情 SQL 并不能做。例如, 不能利用 SQL 查询来计算一个数 n 的阶乘 [$n! = n \times (n-1) \times \dots \times 2 \times 1$], 但是这种事情很容易用 C 或者其他类似的语言实现。SQL 不能把它的输出直接格式化为图形那样方便的形式。所以, 实际的数据库编程同时需要 SQL 和传统的语言, 而后者通常称为宿主语言(host language)。

7.1.2 SQL/ 宿主语言接口

在数据库(只能通过 SQL 语句访问)和宿主语言程序之间是通过宿主语言变量来传递信息的(可以用 SQL 语句读或写宿主语言变量)。当 SQL 语句引用所有这些共享变量时, 变量前面都加上冒号, 但是在宿主语言语句中变量前没有冒号。

当我们想在宿主语言程序中使用 SQL 语句时, 我们要警告: 在 SQL 语句之前要加上关键字 EXEC SQL。一个典型的系统将利用与 SQL 有关的库对这些语句进行预处理并用宿主语言中相应的函数调用来代替它们。

在 SQL2 标准中称为 SQLSTATE 的特殊变量用于连接宿主语言程序和 SQL 执行系统。SQLSTATE 的类型是 5 个字符的数组。每当调用 SQL 库中的函数时, 就把一个

没有实现 SQL2 标准的系统可能使用不同于 SQLSTATE 的名字, 但是我们预期会找到起该作用的某个变量。

代码放入变量 SQLSTATE 中, 以表示调用期间发生的任何问题。例如, '00000' (5 个 0) 表示没有错误发生, '02000' 表示没有找到作为 SQL 查询应答所要的元组。宿主语言程序能够读出 SQLSTATE 的值并且基于该值作出判断。

7.1.3 说明(DECLARE)段

要说明共享的变量, 就要把变量的说明放在两个嵌入式 SQL 语句之间:

```
EXEC SQL BEGIN DECLARE SECTION;
...
EXEC SQL END DECLARE SECTION;
```

中间的内容称为说明段。说明段中变量说明的格式是宿主语言所要求的。而且, 只有所说明的变量属于宿主语言和 SQL 都能处理的类型才有意义, 例如, 整数、实数以及字符串即字符数组。

例 7.1 下列语句可能出现在更新 Studio 关系的 C 函数中。

```
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[ 15], studioAddr[ 50];
    char SQLSTATE[ 6];
EXEC SQL END DECLARE SECTION;
```

前后两句是说明段的开始和结束所必需的。中间的一个语句说明了两个变量 studioName 和 studioAddr。它们都是字符数组, 像我们将会看到的那样, 它们将用来保存制片公司的名字和地址, 名字和地址将作为一个元组插入到 Studio 关系中。第三句说明 SQLSTATE 是 6 个字符的数组。

7.1.4 使用共享变量

在 SQL 语句中可以用共享变量来代替具体的值。使用共享变量时需要在前面加上冒号。这里有一个例子, 我们使用例 7.1 的变量作为将要插入到关系 Studio 中的元组分量。

例 7.2 在图 7.2 中, 我们看到 C 函数 getStudio 的概要, 该函数提示用户输入制片公司的名字和地址, 读取数据, 并把合适的元组插入到关系 Studio 中。1) 到 4) 行是我们在例 7.1 中学过的说明。我们省略了 C 的代码, 这些代码将打印请求信息并输入用于填写数组 studioName 和 studioAddr 的数据。

接下来, 5) 和 6) 行是一个由普通的 INSERT 语句构成的嵌入式 SQL 语句。该语句用关键字 EXEC SQL 作为前导以表明它实际上是嵌入式 SQL 语句而不是不合语法的 C 代码。在图 7.1 中所提到的预处理程序将查看 EXEC SQL 以检测那些必须预处理的语句。

由 5) 和 6) 行所插入的值不是显式常数, 如前面的例 5.27 那样的显式常数。相反地, 6) 行中出现的参数是共享变量, 其当前值将成为插入元组的分量。

我们用 6 个字符来表示 5 个字符的 SQLSTATE 的值, 因为在随后的程序中, 我们要用 C 的函数 strcmp 来检验 SQLSTATE 是否具有某个值。由于 strcmp 要求字符串以 '\0' 结束, 因此需要第 6 个字符作为结束标志。第 6 个字符必须初始化为 '\0', 但是我们不会在随后的程序中给出该赋值。

```

void getStudio() {
    1) EXEC SQL BEGIN DECLARE SECTION;
    2)      char studioName[ 15], studioAddr[ 50];
    3)      char SQLSTATE[ 6];
    4) EXEC SQL END DECLARE SECTION;
        /* 打印请求信息, 输入制片公司的名字和地址, 并把回答填
        写到变量 studioName 和 studioAddr 中 */
    5) EXEC SQL INSERT INTO Studio(name, address)
    6)      VALUES(:studioName, :studioAddr);
}

```

图 7.2 使用共享变量插入新的制片公司

除了 INSERT 语句之外, 还有许多 SQL 语句都可以用共享变量作为接口嵌入到宿主语言中。每个嵌入式 SQL 语句在宿主语言程序中都以 EXEC SQL 作为前导, 并可引用共享变量代替常数。任何不返回结果的 SQL 语句(即, 不是查询语句)都可以嵌入到宿主语言中。嵌入式 SQL 语句的例子包括删除和修改语句以及建立、更改或撤消类似于表和视图那样的模式元素的语句。

然而, select-from-where 查询是不能直接嵌入的。因为匹配失衡, 查询不能简单地嵌入到宿主语言中。查询产生元组集合作为结果, 然而没有任何一种主要的宿主语言直接支持集合数据类型。这样, 嵌入式 SQL 必须使用以下两种机制之一以便把查询结果和宿主语言程序连接起来。

1. 只产生一个元组的查询可以把该元组存储在共享变量中, 元组的每个分量对应一个变量。为了这样做, 我们使用 select-from-where 语句的变形, 叫做单行查询(single-row select)。

2. 如果我们为查询说明一个游标, 就可以执行查询结果超过一个元组的查询。游标将覆盖回答关系的所有元组, 每个返回的元组都可以取到共享变量中并由宿主语言程序予以处理。

我们将依次考察每种机制。

7.1.5 单行查询语句

除了跟在 SELECT 子句后面的关键字 INTO 和共享变量表以外, 单行查询的形式与普通的 select-from-where 语句相同。这些共享变量前面都有冒号, 在 SQL 语句中所有的共享变量都是这样的。如果查询的结果是单一的元组, 那么该元组的分量就成为这些变量的值。如果结果没有元组或者超过一个元组, 那么就不会对共享变量赋值, 而在变量 SQLSTATE 中写入相应的代码。

例 7.3 我们将写一个 C 函数来读取一个制片公司的名字并打印其总裁的净资产。该函数的概要如图 7.3 所示。为了说明我们所需要的变量, 函数以说明段作为开始, 见 1) 到 5) 行。接下来, 我们未明确给出的 C 语句将从标准输入设备读入制片公司的名字。再下面, 6) 到 9) 行是单行查询语句。它和我们已经看到的查询非常相似。两者的不同之处在于: 在 9) 行的条件中, 用变量 studioName 代替了常量字符串, 并在 7) 行有个 INTO 子句

告诉我们把查询结果放到哪里。在这种情况下,我们要求单个元组,而元组只有与属性 netWorth 对应的一个分量。一个元组的该分量的值将存放在共享变量 presNetWorth(类型为整数)中。

```
void printNetWorth() {
    1) EXEC SQL BEGIN DECLARE SECTION;
    2)      char studioName[15];
    3)      int presNetWorth;
    4)      char SQLSTATE[6];
    5) EXEC SQL END DECLARE SECTION;
        /* 打印请求信息,输入制片公司的名字。将输入的名字放到 studioName 中 */
    6) EXEC SQL SELECT netWorth
    7) INTO :presNetWorth
    8) FROM Studio, MovieExec
    9) WHERE presC# = cert# AND Studio.name = :studioName;
        /* 检验 SQLSTATE 是否为全 0,如果是,就打印 presNetWorth 的值 */
}
```

图 7.3 嵌入到 C 函数中的单行查询

7.1.6 游标

把 SQL 语句连接到宿主语言的最通用的方式是使用在一个关系的各个元组上移动的游标(cursor)。该关系可以是已存储的表,也可以是查询产生的结果。要建立和使用游标,我们需要下列语句:

- 1. 游标说明。游标说明的最简单格式组成如下:
 - (a) 用 EXEC SQL 引导,就像所有嵌入式 SQL 语句那样。
 - (b) 关键字 DECLARE。
 - (c) 游标的名字。
 - (d) 关键字 CURSOR FOR。
 - (e) 表达式,例如关系的名字或者 select-from-where 表达式,它的值是一个关系。已说明的游标将覆盖该关系的所有元组;也就是说,当游标向前“推进”(fetch)时,该游标可以依次指向该关系的每个元组。

综上所述,游标说明的格式为

```
EXEC SQL DECLARE 游标 CURSOR FOR 查询
```

- 2. EXEC SQL OPEN 语句,其后跟着游标的名字。该语句将游标初始化到某个位置,从该位置可以检索到游标所覆盖的关系的第一个元组。
- 3. 一次或者多次使用推进语句。推进语句的目的是得到该游标所覆盖的关系的下一个元组。如果已经把元组取完了,那么就没有元组返回,结果 SQLSTATE 的值就设置成 '02000',该代码意味着“没有找到元组”。推进语句由以下几个部分组成:
 - (a) 关键字 EXEC SQL FETCH FROM。
 - (b) 游标的名字。
 - (c) 关键字 INTO。

(d) 由逗号分开的共享变量表。如果推进到一个元组, 那么该元组的各个分量将依次放到这些变量中。

也就是说, 推进语句的格式是:

EXEC SQL FETCH FROM 游标 INTO 变量表

4. EXEC SQL CLOSE 语句, 其后跟着游标的名字。该语句关闭现在不再覆盖关系元组的游标。然而, 该游标可以用另一个 OPEN 语句重新初始化, 在这种情况下它将重新覆盖该关系的元组。

例 7.4 假定我们要确定电影行政长官的净资产分布情况, 他们将按净资产划分成按指数增长的若干段, 每段与他们的净资产有几位数相对应。我们将设计一个查询, 它取出 MovieExec 的每个元组的 netWorth 域并且放到共享变量 worth 中。游标 execCursor 将覆盖这些单一分量的元组。每当取出来一个元组时, 我们就计算在整型变量 worth 中净资产的位数, 并把数组 counts 中相应的元素加 1。

C 函数 worthRanges 从图 7.4 的 1) 行开始。2) 行说明了一些只用于 C 函数而不用于嵌入式 SQL 的变量。数组 counts 保存不同段内行政长官的数目, digits 计算净资产有几位数, i 是覆盖数组 counts 的所有元素的下标。

3) 到 6) 行是嵌入式 SQL 的说明段, 在此说明了共享变量 worth 和常见的 SQLSTATE。7) 和 8) 行说明 execCursor 为游标, 它将覆盖 8) 行中的查询所产生的值。该查询只是简单地请求 MovieExec 的所有元组的 netWorth 分量。然后在 9) 行打开该游标。10) 行通过将数组 counts 的元素置 0 来完成初始化。

```
1) void worthRanges() {
2)     int i, digits, counts[15];
3)     EXEC SQL BEGIN DECLARE SECTION;
4)         int worth;
5)         char SQLSTATE[6];
6)     EXEC SQL END DECLARE SECTION;
7)     EXEC SQL DECLARE execCursor CURSOR FOR
8)         SELECT netWorth FROM MovieExec;
9)     EXEC SQL OPEN execCursor;
10)    for (i= 0; i< 15; i++ ) counts[i]= 0;
11)    while(1){
12)        EXEC SQL FETCH FROM execCursor INTO :worth;
13)        if (NO-MORE-TUPLES) break;
14)        digits = 1;
15)        while( (worth /= 10) > 0) digits++ ;
16)        if(digits <= 14) counts[digits]++ ;
17)    }
18)    EXEC SQL CLOSE execCursor;
19)    for (i= 0; i< 15; i++ )
20)        printf( digits = %d: number of execs = %d\n , i, counts[i]);
21) }
```

图 7.4 把行政长官的净资产分成指数段

主要的工作是由 11) 到 16) 行的循环完成的。在 12) 行把一个元组取到共享变量

worth 中。因为由 8) 行的查询产生的元组只有一个分量, 所以我们只需要一个共享变量, 不过在通常情况下会有多个变量, 而变量数应该和检索到的元组的分量数相同。13) 行检测推进操作是否成功。在这里, 我们使用了宏 NO- MORE- TUPLES, 可以认为它们是由下列语句定义的:

```
# define NO- MORE- TUPLES ! (strcmp(SQLSTATE, 02000 ))
```

回忆一下, 当 SQLSTATE 的内容是 02000 时意味着没有找到元组。13) 行检测是否查询返回的所有元组前面都已经找到而不存在“下一个”元组了。如果是这样, 我们就跳出循环转到 17) 行。

如果取出一个元组, 那么在 14) 行就把净资产的位数 digits 初始化为 1。15) 行是一个循环, 它重复地把净资产除以 10 并把 digits 加 1。当除以 10 之后净资产为 0 时, digits 就保存原来检索到的 worth 值的正确位数。最后, 16) 行将数组 counts 中相应的元素加 1。我们假定净资产的位数不会超过 14。然而, 即使净资产有 15 位或者更多, 16) 行将因为没有相应的作用域而不会把数组 counts 的任何元素加 1; 也就是, 将舍弃巨大的净资产而不会影响统计结果。

从 17) 行开始函数进入收尾阶段。关闭游标, 18) 和 19) 行打印数组 counts 中的值。

7. 1. 7 通过游标的更新

当游标覆盖基本表(也就是数据库中存储的关系, 而不是由查询所构建的视图或者关系)的元组时, 人们不仅能读出并处理每个元组的值, 而且也能修改或删除元组。除了 WHERE 子句之外, 这些 UPDATE 和 DELETE 语句的句法和我们在 5. 6 节中遇到的一样。WHERE 子句只能是在 WHERE CURRENT OF 之后跟着游标的名字。当然, 在决定是否删除或者修改元组之前, 读取元组的宿主语言程序可能把它喜欢的任何条件用于该元组。

例 7. 5 在图 7. 5 中我们看到与图 7. 4 类似的 C 函数。二者都说明了游标 execCursor 将覆盖 MovieExec 的元组。然而, 图 7. 5 查找每个元组并决定是删除该元组还是将净资产加倍。

我们又把宏 NO- MORE- TUPLES 用于以下这种情况: 变量 SQLSTATE 具有表示“已无元组”的代码 02000。在 12) 行的测试中, 我们查询净资产是否少于 \$ 1 000。如果是这样, 就在 13) 行用删除语句把该元组删除。如果净资产至少 \$ 1 000, 那么就在 15) 行把净资产加倍。

7. 1. 8 游标选项

SQL2 为游标提供了各种各样的选项。下面是一个总结。详细介绍可以在 7. 1. 9 节到 7. 1. 11 节找到。

- 1. 可以指定从关系中取出元组的顺序。
- 2. 可以限制游标所覆盖的关系发生改变所产生的影响。
- 3. 可以改变游标在元组列表上的移动方式。

```

1) void changeWorth() {
2)     EXEC SQL BEGIN DECLARE SECTION;
3)         int worth;
4)         char SQLSTATE[ 6];
5)     EXEC SQL END DECLARE SECTION;
6)     EXEC SQL DECLARE execCursor CURSOR FOR
7)         SELECT netWorth FROM MovieExec;
8)     EXEC SQL OPEN execCursor;
9)     while(1) {
10)         EXEC SQL FETCH FROM execCursor INTO :worth;
11)         if (NO-MORE-TUPLES) break;
12)         if (worth < 1000)
13)             EXEC SQL DELETE FROM MovieExec
14)                 WHERE CURRENT OF execCursor;
15)         else
16)             EXEC SQL UPDATE MovieExec
17)                 SET netWorth = 2 * netWorth
18)                 WHERE CURRENT OF execCursor;
19)     }
20)     EXEC SQL CLOSE execCursor;
21) }

```

图 7.5 更新行政长官的净资产

7.1.9 为取出的元组排序

让我们先来考虑一下元组的顺序。我们可以把取出的元组按照任一分量的值排序。为了规定一个顺序,我们在游标所覆盖的关系的定义中加上关键字 ORDER BY 和用于排序的分量表,就像我们在 5.1.5 节为查询所做的那样。先按分量表的第一分量排序,第一个分量相同就按第二个分量排序,又相同,再按第三个分量排序,依次类推。可以用属性或者数字来规定分量。在后一种情况下,数字指该属性在关系的所有属性中的位置。

例 7.6 假定我们希望检验一个关系的元组,该关系是通过连接和投影而构成的,先连接 Movie 和 StartIn 两个关系,然后投影产生的关系只有电影的名称、年份、影星和制片公司。我们同样希望按年份对这些关系进行排序,而在同一年的元组中,我们将按名称对这些元组进行排序(按字母顺序)。图 7.6 说明游标 movieStarCursor 覆盖了所构成的关系。

```

1) EXEC SQL DECLARE movieStarCursor CURSOR FOR
2)     SELECT title, year, studioName, starName
3)     FROM Movie, StarsIn
4)     WHERE title = movieTitle AND year = movieYear
5)     ORDER BY year, title;

```

图 7.6 用 ORDER BY 子句控制取出元组的顺序

2)到4)行是普通的 SELECT 子句,1)行说明一个游标来覆盖该关系的元组。5)行是说当我们通过游标 movieStarCursor 取出元组时,将首先得到年代最早的元组。同一

年份的元组则按照第二个属性名称来分组。同一年中的名称将按照字母顺序排序, 因为字符串的值就是这样排序的。此处并没有限定同一部电影中表示不同影星的元组也排序。

7. 1. 10 防止并发更新的保护措施

下面, 让我们来考虑这种可能性: 当某个函数通过例 7. 6 的 movieStarCursor 游标来读取元组时, 并发执行的某个函数(或者甚至是同一函数)正在改变底层的 Movie 或者 StarsIn 关系。在 7. 2 节, 我们将更多地介绍几个进程同时访问一个数据库的情况。然而, 现在, 让我们只是接受这种可能性, 即当我们使用一个关系时其他进程可能更新它。

对于这种可能性我们能做什么呢? 也许什么也做不了。我们可能只是为了某个或者某些影星而检索元组, 至于影星所在的元组是否正在插入或者删除并不重要。于是, 我们只是简单地接受通过游标得到的元组。

然而, 我们也许不希望由于并发产生的变化影响我们通过游标查看的元组。例如, 如果我们的函数查看的元组导致函数把新的元组增加到 StarsIn, 我们就可能会进入一个反馈环, 也就是新的元组通过游标产生附加的元组, 结果又产生更多的新元组, 迅速激增。如果有这种或那种不希望发生的危险, 我们不妨说明游标对并发产生的变化不敏感 (insensitive)。

例 7. 7 我们可以把图 7. 6 中的 1) 行改为:

1) EXEC SQL DECLARE movieStarCursor INSENSITIVE CURSOR FOR

如果这样说明 movieStarCursor, 那么 SQL 系统将确保在打开和关闭游标之间关系 Movie 或 StarsIn 发生的变化将不会影响取出元组的集合。

考虑到 SQL 系统也许要花费更多时间来管理对数据的访问以确保游标不受影响, 因此不敏感的游标可能开销较大。另外, 对数据库中管理并发操作的讨论将推后到 7. 2 节。然而, 一个支持不敏感游标的简单方法是, 让 SQL 系统把可能访问使用不敏感游标的底层关系(比如 Movie 或 StarsIn)的任何进程挂起。

还有一些覆盖在关系 R 上的游标, 我们可以肯定地说它们将不会改变关系 R。这种游标可以和 R 的不敏感游标同时运行, 而不会发生不敏感游标看到关系 R 改变的危险。如果我们说明一种 FOR READ ONLY 的游标, 那么数据库系统就会由于是通过这种游标访问数据库而确保底层的关系不会改变。

例 7. 8 我们可以在图 7. 6 的 5) 行后面加上 6) 行

6) FOR READ ONLY;

如果我们这样做了, 那么通过游标 movieStarCursor 来执行 UPDATE 或者 DELETE 的任何尝试都会发生错误。

7. 1. 11 滚动游标

游标选项的最后一类是选择如何在关系的元组上移动。默认的也是最常见的选择是从头开始, 依次取出元组, 直到结束。然而, 还可按其他次序取出元组, 并在关闭游标之前, 对元组多次扫描。为了利用这些选项, 我们需要做两件事情。

1. 在说明游标时,把关键字 SCROLL 放在关键字 CURSOR 之前。这种改变告诉 SQL 系统,游标可能不以按元组顺序向前移动的方式使用。

2. 在推进语句中,在关键字 FETCH 之后跟上几种选项之一来告诉到哪里找到想要的元组。这些选项是:

- (a) NEXT 或者 PRIOR 将依次取下一个或者上一个元组。记住这些元组以游标的当前位置为基准。如果没有指定选项, NEXT 就是默认的选择,而且也是通常的选择。
- (b) FIRST 或者 LAST 将依次取第一个或者最后一个元组。
- (c) 跟着一个正整数或者负整数的 RELATIVE(相对)表示依次向下(如果是正整数)移动多少元组或者向上(如果是负整数)移动多少元组。例如, RELATIVE 1 是 NEXT 的同义词,而 RELATIVE - 1 是 PRIOR 的同义词。
- (d) 跟着一个正整数或者负整数的 ABSOLUTE(绝对),表示从前面(如果正)或者从后面(如果负)计算的所要元组的位置。例如, ABSOLUTE 1 是 FIRST 的同义词,而 ABSOLUTE - 1 是 LAST 的同义词。

例 7.9 让我们重写图 7.5 中的函数,从最后一个元组开始,通过元组的列表向回移动。首先,我们需要说明游标 execCursor 是滚动型的,这就需要在 6) 行中加上关键字 SCROLL 如下:

6) EXEC SQL DECLARE execCursor SCROLL CURSOR FOR

7) SELECT netWorth FROM MovieExec;

此外,我们还需要用 FETCH LAST 语句把取出元组的操作初始化,并在循环中使用 FETCH PRIOR。把图 7.5 中 9) 到 15) 行的循环重写如下:

```
EXEC SQL FETCH LAST FROM execCursor INTO: worth;
while( 1) {
    /* 与 11) 到 15) 行相同 */
    EXEC SQL FETCH PRIOR FROM execCursor INTO : worth;
}
```

读者不应该认为 SELECT netWorth FROM MovieExec 产生的倒序的元组有什么好处。事实上,让系统提供反向的元组开销会更大,因为在游标 execCursor 推进到第一个位置之前就不得不把全部元组生成并存储起来。

7.1.12 动态 SQL

我们嵌入到宿主语言中的 SQL 模型都是较大的宿主语言程序中特定的 SQL 查询和命令。然而有一种更通用的模型可以把 SQL 嵌入到另外一种语言中。语句本身可以由宿主语言所计算。这样的语句在编译的时候并不知道,所以不能由 SQL 预处理程序或者宿主语言编译程序来处理。

这种情况的例子是这样一个程序:提示用户为 SQL 查询输入信息,然后读入查询,再执行该查询。在第 5 章中我们所假设的特定 SQL 查询的解释程序就是这种程序的一个例子;每个商业的 SQL 系统都提供这类解释程序。如果在运行时读入并执行查询要求,那么

在编译时就没什么可做的了。SQL 系统读入查询要求之后,就立即进行分析并以合适的方式予以执行。

宿主语言程序必须指导 SQL 系统处理读入的字符串,把它转换成可执行的 SQL 语句,并且最终执行该语句。这里有两个动态的 SQL 语句来执行这两个步骤。

1. EXEC SQL PREPARE, 后跟 SQL 变量 V、关键字 FROM 以及字符串类型的宿主语言变量或表达式。这个动态的 SQL 语句使字符串变成一个 SQL 语句,而 V 的值就变成该 SQL 语句。可以推测,SQL 系统将对 SQL 语句进行语法分析,并且找到执行该语句的好方法,但是还没有执行该语句。

2. EXEC SQL EXECUTE, 后跟一个 SQL 变量如步骤 1 中的 V。该语句的作用是执行 V 所代表的 SQL 语句。

通过在如下语句

EXEC SQL EXECUTE IMMEDIATE

后面跟上字符串型的共享变量或者字符串型的表达式,这两个步骤可以合并成一个步骤。在一个语句作一次预处理,然后执行多次的情况下,就可以看出把这两部分合起来的缺点了。若用 EXECUTE IMMEDIATE 这种形式,每当执行该语句时都要花费预处理的时间,不如只预处理一次。

例 7.10 图 7.7 中给出一个 C 程序的概要,它从标准输入设备把文本读入到变量 query 中,进行预处理并且执行它。SQL 变量 SQLquery 保存预处理过的查询语句。因为这样的查询只执行一次,用下面这样一个语句来代替图 7.7 中的 6) 和 7) 行是可以接受的:

EXEC SQL EXECUTE IMMEDIATE :query;

```
1) void readQuery() {
2)     EXEC SQL BEGIN DECLARE SECTION;
3)         char * query;
4)     EXEC SQL END DECLARE SECTION;
5)     /* 提示用户输入一个查询,分配空间(例如,使用 malloc)
        并且使共享变量: query 指向该查询的第一个字 */
6)     EXEC SQL PREPARE SQLquery FROM :query;
7)     EXEC SQL EXECUTE SQLquery;
}
```

图 7.7 预处理并执行动态 SQL 查询

7.1.13 本节练习

练习 7.1.1: 基于练习 4.1.1 中的数据库模式,写出下列嵌入式的 SQL 查询。

```
Product(maker, model, type)
PC(model, speed, ram, hd, cd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

你可以使用任何一种你熟悉的宿主语言,如果愿意,你可以用清晰的注释来代替宿主语言程序的具体细节。

- * (a) 询问用户所要求的价格(price), 找出与要求的价格最接近的 PC。打印 PC 的厂商(maker)、型号(model) 和速度(speed)。
- (b) 询问用户所能接受的速度(speed)、内存(RAM)、硬盘容量(hd) 和屏幕尺寸(screen) 的最小值。查找所有满足以上要求的便携式电脑(laptop)。打印它们的规格(laptop 的所有属性) 和它们的厂商。
- ! (c) 要求用户指定一个厂商。打印该厂商的所有产品(product) 的规格。也就是, 打印型号(model)、类型(type) 和适合该类型的任何关系的所有属性。
- !! (d) 要求用户给出预算(“ budget ”)(指的是一台 PC 和一台打印机的总价), 以及 PC 的最低的速度(speed)。找出在预算和最低速度范围内最便宜的“ 系统 ”(PC 加上打印机), 但是如果可能就使打印机是一台彩色打印机。打印所选择系统的型号(model)。
- (e) 要求用户给出一台新 PC 的厂商、型号、速度、内存、硬盘容量, 光驱速度(CD) 和价格。检验是否有这种型号的 PC。如果有就打印一个通知信息, 否则将信息插入到 Product 和 PC 表中。
- * ! (f) 使所有“ 旧 ”PC 的价格调低 \$ 100。要确保在你的程序运行期间插入的任何“ 新 ”PC 的价格不会调低。

练习 7.1.2: 基于练习 4.1.3 中的数据库模式, 写出下列嵌入式的 SQL 查询。

Classes(class, type, country, numGuns, bore, displacement)

Ships(name, class, launched)

Battles(name, date)

Outcomes(ship, battle, result)

- (a) 舰艇的火力大致和火炮的数量(numGuns) 与火炮口径(bore) 的立方之积成正比。找出具有最大火力的舰艇等级(class)。
- ! (b) 要求用户给出一次战役(battle) 的名称(name)。找出参加该战役的舰艇所属的国家。打印沉没舰艇最多的国家和损坏舰艇最多的国家。
- (c) 要求用户给出一个等级的名字和表 Classes 的一个元组所需要的其他信息。接下来查询那种等级的舰艇的名字的列表和它们的下水(launched) 日期。然而, 用户不必给出全称中的第一个名字, 因为它必须是该等级的名字。
- ! (d) 检验 Battles, Outcomes 和 Ships 关系以找出在下水之前就参加战役的舰艇。如果发现错误, 就提示用户, 在改变下水日期或者战役日期之间提供选择。做所请求的任何一种改变。

* ! 练习 7.1.3: 在本练习中, 找出如下关系中满足条件的所有 PC。

PC (model, speed, ram, hd, cd, price)

条件是: 至少有两种速度相同而价格更贵的 PC。尽管我们有许多办法来解决该问题, 但是在该练习中读者应该使用滚动游标。读取先按速度(speed) 再按价格(price) 排序的 PC 的元组。提示: 对于每个读出的元组, 往前跳两个元组看看速度是否没变。

!! 练习 7.1.4: 在 7.1.1 节中, 我们提到用 SQL 不能写出求阶乘的程序。那种断言对 SQL2 是正确的。然而, 就像在 5.10 节所讲的 SQL3 递归允许我们做一些近似的事情。写

出用于关系 M 的递归 SQL3 查询, M 包含单个元组(m), 其中 m 是一个整数。查询的结果将是元组的集合(n, n!), 其中 $1 \leq n \leq m$ 。

7.2 SQL 中的事务

到目前为止, 我们对数据库进行操作的模型都是一个用户对数据库进行查询或者更新。因此, 在某个时刻对数据库只执行一个操作, 而前一个操作遗留的数据库状态就是下一个操作所处的状态。此外, 我们假定操作作为一个整体都执行了; 也就是, 不可能出现如下情况: 在一个操作的执行过程中由于硬件或者软件的故障, 而使数据库处于在数据库正常操作下无法解释的状态。

真正的情况通常要复杂得多。我们将首先考虑什么样的操作使数据库处于不反映在它上面所执行的操作的状态, 然后, 我们将考虑 SQL 提供给读者什么样的工具以保证这些问题不会发生。

7.2.1 可串行性

在像银行业和飞机订票的应用中, 每秒钟都会对数据库执行数以百计的操作。这些操作可能在成百甚至成千的节点中的任何一个上开始执行, 例如在自动出纳机或者旅游代理、航班职员或航班顾客自己的台式机上执行。因为操作时间上的重叠, 两个操作完全有可能同时作用于同一帐户或者同一航班。如果这样, 它们可能以非常奇怪的方式互相影响。这里有一个例子说明, 如果数据库管理系统对在数据库上的操作顺序完全没有限制可能会发生什么样的错误。我们强调数据库系统在正常情况下不是按照这种方式工作的, 当使用商业数据库管理系统时, 不得不想尽办法以避免发生这类错误。

例 7.11 假定我们编写了函数 chooseSeat() 来读取关于航班和空座位的关系, 找出是否还有特定的座位空着, 如果有, 就把它置成非空。我们操作的关系称为 Flights, 其属性有 fltNum, fltDate, fltSeat 和 occupied, 这些属性都具有有很明显的意义。座位选择程序的概要如图 7.8 所示。

图 7.8 中 9) 到 11) 行是一个单行查询, 它根据指定的座位是否已占用设置共享变量 occ 为真或假(1 或 0)。12) 行检验座位是否已占用, 如果没有, 就把该座位的元组修改为占用。修改操作由 13) 到 15) 行完成, 在 16) 行将把座位分配给需要的用户。实际上, 我们可以把座位分配信息存放在另一个关系中。最后在 17) 行, 如果座位已占用, 就要通知用户。

现在, 请记住, 两个或者更多的用户可能同时执行函数 chooseSeat()。假定碰巧两个代理几乎在同一时间试图预定同一天、同一航班的同一座位, 如图 7.9 所示。在同一时间它们都执行到 9) 行, 并且它们的本地变量 occ 的副本都取值为 0; 也就是, 座位尚未分配。从 12) 行起 chooseSeat() 的每个执行都将此位修改为 1, 也就是, 占用该座位。这些操作可能一个接一个地执行, 每个执行都在 16) 行告诉用户“ 该座位属于你 ”。

如我们从例 7.11 中看到的那样, 两个操作各自执行都正确, 但整体的结果却不正确, 这种情况是可能的: 两个用户都认为自己请求的座位已落实。该问题可以这样解决: 通过

保证可串行化特性

实际上, 要求操作串行运行往往是不可能的; 因为操作太多, 需要一定的并行性。于是数据库管理系统就采取一定机制来保证可串行化的特征; 即使执行不是串行的, 但是对用户来说, 结果看起来好像操作是串行执行的。

像我们在 1.2.4 节所讨论的那样, 一个通用的方法是数据库管理系统锁住数据库的元素从而两个函数不能同时访问这些元素。例如, 若把例 7.11 的函数 chooseSeat() 写成锁住关系 Flights 的其他操作, 则不访问 Flights 的操作就可以和 chooseSeat() 的该请求并行运行, 但是 chooseSeat() 的另一请求不能运行。实际上, 像 1.2.4 节中提到的那样, 锁住比整个关系更小的元素, 例如单个磁盘块或者单个元组, 将允许更多的并行性, 包括同时运行 chooseSeat() 的某些请求的能力。

```
1) EXEC SQL BEGIN DECLARE SECTION;
2)      int flight; /* 航班号 */
3)      char date[10]; /* 用 SQL 格式表示的航班日期 */
4)      char seat[3]; /* 两个数字和一个字母表示一个座位 */
5)      int occ; /* 用布尔值表明座位是否已占用 */
6) EXEC SQL END DECLARE SECTION;
7) void chooseSeat() {
8)      /* C 代码提示用户输入航班号、日期和座位,
          并把它们保存在具有相应名字的三个变量中 */
9) EXEC SQL SELECT occupied INTO :occ
10)          FROM Flights
11)          WHERE fltNum = :flight AND fltDate = :date
              AND fltSeat = :seat;
12) if (! occ) {
13)      EXEC SQL UPDATE Flights
14)          SET occupied = 'B1'
15)          WHERE fltNum = :flight
              AND fltDate = :date
              AND fltSeat = :seat;
16)      /* C 和 SQL 代码登记座位分配情况并通知用户分配结果 */
17)      }
18) else /* C 代码通知用户座位已占用并请求另选座位 */
19)      }
```

图 7.8 选择座位

几个 SQL 机制使两个函数的执行实现串行化。如果在一个函数开始执行以前另一个函数已经执行完了, 我们就说对同一数据库进行操作的两个函数其执行是串行的。即使两个函数的执行在时间上可能是重叠的, 只要它们执行的情况如同是在串行执行, 我们就说这种执行是可串行化的(serializable)。

很明显, 如果 chooseSeat() 的两个请求是串行(或者可串行化地)运行的, 我们所看到的错误就不会发生。一个用户首先发出请求。该用户看到一个空座位并把它预订了。然

后另外一个用户的请求开始执行并且看到座位已经占用了。对于用户来说谁得到座位是有关系的,但是对数据库来说,重要的是一个座位只能分配一次。

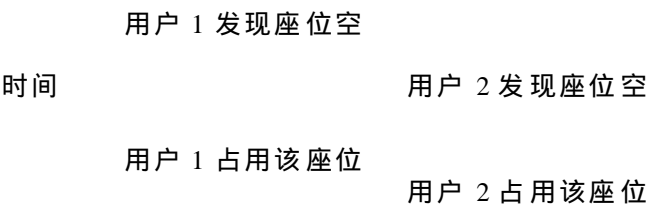


图 7.9 两个用户试图同时预订同一座位

7.2.2 原子性

除了两个或者更多的数据库操作同时执行可能引发非串行化行为以外,当单个操作正在执行时如果出现硬件或者软件的“崩溃”,那么该操作也可能把数据库置于不可接受的状态。这里有另外一个例子提醒我们可能会发生什么。如例 7.11 所示,我们应当记住实际的数据库系统不允许在设计合理的应用程序中发生这种错误。

例 7.12 让我们来设想另一种常见的数据库类型:银行的帐户记录。我们可以用具有属性 acctNo 和 balance 的关系 Accounts 来表示这种情形。在该关系中帐户号和该帐户的余额组成一对。

我们希望写一个函数 transfer(),它读取两个帐户和一定数量的钱,检验第一个帐户是否至少有那么多钱,如果是,就把这些钱从第一个帐户转到第二个帐户。图 7.10 是函数 transfer()的概要。

```
1) EXEC SQL BEGIN DECLARE SECTION;
2)      int acct1, acct2; /* 两个帐户 */
3)      int balance1; /* 第一个帐户的结余 */
4)      int amount; /* 转帐的金额 */
5) EXEC SQL END DECLARE SECTION;
6) void transfer() {
7)      /* C 代码。提示用户输入帐户 1 和帐户 2 以及转帐的金额,
           放到变量 acct1, acct2 和 amount 中 */
8)      EXEC SQL SELECT balance INTO :balance1
9)      FROM Accounts
10)     WHERE acctNo = acct1;
11)     if (balance1 >= amount) {
12)         EXEC SQL UPDATE Accounts
13)         SET balance = balance + amount
14)         WHERE acctNo = acct2;
15)         EXEC SQL UPDATE Accounts
16)         SET balance = balance - amount
17)         WHERE acctNo = acct1;
           }
18)     else /* C 代码。如果没有足够的钱进行转帐,就打印一个信息。 */
           }
```

图 7.10 从一个帐户向另一个帐户转帐

在事务处理过程中数据库是如何改变的

不同的系统可能以不同的方式来实现事务。当事务执行时,它可以改变数据库。如果该事务失败了,某个其他事务可能看到那些改变。最常用的解决办法是让数据库系统锁住变化的数据项直到选择 COMMIT(提交)或 ROLLBACK(退回)为止。这样可以避免其他事务看到这种暂时的改变。如果用户想让事务以可串行化的方式运行,那么一定要使用锁或者等效的东西。

然而,就像我们从 7.2.4 节开始看到的那样,SQL2 提供给我们几个选项来处理暂时的数据库改变。即使随后的退回使改变不可见,改变的数据也可能没有锁住而变得可见。事务的设计者要决定是否避免暂时改变的可见性。如果是这样,那么所有的 SQL 实现将提供像加锁这样的方法来保持在提交之前的改变是不可见的。

图 7.10 的处理过程简单明了。8)到 10)行检索第一个帐户的余额。在 11)行确定余额是否足够多从而可以从中减去所要求的数量。如果是这样,那么 12)到 14)行就把该数加到第二个帐户上,而 15)到 17)行则从第一个帐户减去该数。如果第一个帐户的钱不够,那么就不做转帐,并在 18)行打印通知信息。

现在考虑,如果在 14)行后面发生了故障;也许是计算机故障或者是实际执行转帐的连接数据库和处理机的网络故障,那么数据库就将处于这种状态:钱已经转到第二个帐户,但是还没有从第一个帐户取走。银行实际上损失了要转帐的这笔钱。

例 7.12 说明的问题是数据库操作的某些组合(如图 7.10 中的两个修改)需要以原子的方式完成,也就是,它们或者都做,或者都不做。例如一个通常的解决方法是让对数据库的所有改变在本地的工作区完成,并且只有当所有的工作完成以后我们才把这种改变提交给数据库,于是所有的改变都成为数据库的一部分并且对其他操作是可见的。

7.2.3 事务

7.2.1 和 7.2.2 节中提出的串行化和原子性问题的解决方法是把这些数据库操作组合成事务(transaction)。事务是在数据库上的一个或者多个操作的聚集,它必须以原子的方式执行;也就是,所有的操作要么都做,要么都不做。另外,早期的 SQL 标准还要求事务好像以串行的方式执行;也就是,它们是可串行化的。然而,SQL2 具有更加灵活的观点。在 SQL2 中,可串行性是默认的,但是用户可以在两个或者更多的交叉事务中规定稍加严格的限制。在以后的几节中我们将讨论这些对可串行性条件的更改。

当对数据库或者模式进行查询或操作的任何 SQL 语句开始时,事务也就开始了。在 SQL 中我们不必给出任何专门的事务开始语句。然而,我们必须明确地结束一个事务。我们可以用两种方法来结束事务。

1. SQL 语句 COMMIT(提交)使事务成功地结束。自从当前事务开始后 SQL 语句所造成的数据库的任何改变将永久地放置在数据库中(也就是,把改变的内容提交了)。在

尽管某些实现对此稍加限制。

COMMIT 语句执行以前, 改变都是暂时的, 对其他事务可能可见也可能不可见。

2. SQL 语句 ROLLBACK(退回)使事务异常终止, 即不成功地终止。响应该事务的 SQL 语句所造成的任何改变都作废了(也就是, 把改变的内容复原了), 所以数据库不会发生改变。

例 7. 13 假定我们想让图 7. 10 中的函数 transfer() 以单个事务的形式执行。在 8) 行当我们读取第一个帐户的余额时, 事务开始。如果 11) 行的测试为真, 而且完成了资金的转帐, 那么就要提交所做的改变。所以我们把附加的 SQL 语句

```
EXEC SQL COMMIT;
```

放在 12) 到 17) 行的 if 子句的后面。

如果 11) 行的测试为假—— 也就是, 没有足够的资金来转帐—— 我们宁愿取消该事务。通过把语句

```
EXEC SQL ROLLBACK;
```

放在 18) 行所示的 else 子句后面就可以做到这一点。实际上, 既然在该分支中, 没有执行数据库更新语句, 是提交还是异常终止都没有关系, 因为没有什么变化需要提交。

7. 2. 4 只读事务

例 7. 11 和例 7. 12 各自包含一个事务, 先读然后(可能) 写一些数据到数据库中。这类事务容易引发串行化问题。于是我们在例 7. 11 中看到如果函数的两个执行都想在同一时间订同一座位会发生什么情况, 同时我们在例 7. 12 中看到如果在函数执行过程中发生崩溃会发生什么情况。然而, 当一个事务只是读数据而不写数据时, 我们就有更大的灵活性让该事务和其他事务并行执行。

例 7. 14 假定我们写了一个函数来读取数据以确定是否还有特定的座位空着; 该函数就像图 7. 8 中 1) 到 11) 行那样运行。我们能够同时执行对该函数的许多请求, 而不用担心对数据库造成永久的损害。可能发生的最坏的情况是, 当我们读取一个特定的座位判断其是否空着时, 其他某个函数执行正在预订或者正在退掉该座位。这样, 我们可能会得到“ 空着 ”或者“ 不空 ”的回答, 这取决于我们执行查询时在时间上的细微差别, 但是这种答案有时是有意义的。

如果我们告诉 SQL 执行系统当前的事务是只读的, 也就是它绝对不会改变数据库, 那么 SQL 系统很可能会利用该信息。尽管我们在此不讨论详细的机制, 然而让许多访问同样数据的只读事务并行运行一般来说是可能的, 但是不允许它们和写相同数据的事务并行运行。

我们用

```
SET TRANSACTION READ ONLY;
```

来告诉 SQL 系统下一个事务是只读的。该语句必须在事务开始之前执行。例如, 如果我们有一个由图 7. 8 中的 1) 到 11) 行组成的函数, 就可以把

可在事务和游标的管理之间进行比较。例如, 我们在 7. 1. 10 节提到只读游标比一般游标具有更多的并行性。类似地, 只读的事务允许并行操作。

EXEC SQL SET TRANSACTION READ ONLY;

刚好放在事务开始执行的 9) 行之前来说明它是只读的。在 9) 行之后做只读说明就太晚了。

我们同样可以用如下语句

SET TRANSACTION READ WRITE;

来通知 SQL 系统接下来的事务将要写或者可能写数据。然而, 该选项一般是默认的, 因此也是不必要的。

7.2.5 读脏数据

脏数据是个常用术语, 用来表示已由事务写完但尚未提交的数据。读脏数据(dirty read)就是对脏数据的读取。读脏数据的危险是写脏数据的事务最终可能异常终止。如果这样, 那么将把脏数据从数据库中清除出去, 一切都好像那些数据从来没有存在过一样。如果其他某个事务读取了脏数据, 那么该事务可能提交或者用其他某种操作来反映脏数据的内容。

读脏数据有时有问题, 有时没问题。在影响足够小的情况下, 值得冒险来避免数据库管理系统为防止读脏数据而进行耗费时间的必要操作。这里有一些例子说明当允许读脏数据时会发生什么事。

例 7.15 让我们重新考虑例 7.12 中的帐户转帐。然而, 假定转帐是通过执行下面一系列步骤的程序 P 实现的:

1. 把钱加到帐户 2。
2. 检验帐户 1 是否有足够的钱。
 - (a) 如果没有足够的钱, 从帐户 2 去掉这笔钱并且异常终止。
 - (b) 如果有足够的钱, 从帐户 1 减掉这笔钱并且提交。

如果程序 P 可串行化执行, 那么我们暂时把钱放入帐户 2 就不会有问题。没有人会看到这笔钱, 而且如果不能转帐就会把这笔钱去掉。

然而, 假设可能读脏数据。假定有三个帐户: A1, A2 和 A3 分别存有 \$ 100, \$ 200 和 \$ 300。假定事务 T₁ 执行程序 P 从 A1 到 A2 转帐 \$ 150。几乎同时, 事务 T₂ 执行程序 P 从 A2 到 A3 转帐 \$ 250。可能的事件顺序如下:

1. T₂ 执行步骤 1, 往 A3 加 \$ 250, A3 现在有 \$ 550。
2. T₁ 执行步骤 1, 往 A2 加 \$ 150, A2 现在有 \$ 350。
3. T₂ 执行步骤 2 的检测, 发现 A2 具有足够的资金(\$ 350) 用于从 A2 到 A3 转帐 \$ 250。
4. T₁ 执行步骤 2 的检测, 发现 A1 没有足够的资金(\$ 100) 用于从 A1 到 A2 转帐 \$ 150。
5. T₂ 执行步骤 2(b), 从 A2 减 \$ 250, A2 现在有 \$ 100, 于是 T₂ 提交。
6. T₁ 执行步骤 2(a), 从 A2 减 \$ 150, A2 现在有 - \$ 50, 于是 T₁ 异常终止。

总钱数没有变, 在三个帐户中仍然有 \$ 600。但是因为事务 T₂ 在以上 6 个步骤中的第

三步读取了脏数据,因此我们没能防止一个帐户变负,这恐怕就是检测第一个帐户看它是否有足够资金的目的。

例 7.16 让我们假想例 7.11 中 seat-choosing 函数的变形。在新的方法中:

1. 我们找到一个空座位,通过把该座位的 occupied 置成 1 来预订它。如果没有座位,就异常终止。
2. 我们询问顾客是否同意该座位。如果同意,我们就提交。否则,我们就通过设置 occupied 为 0 释放该座位,然后重复步骤 1 去找另一个座位。

如果两个事务几乎同时执行该算法,一个事务可能预订座位 S,而过后顾客又退掉了。如果第二个事务在座位 S 标记为已占用的时刻执行了步骤 1,那么该事务的顾客对座位 S 将没有选择的机会。

像例 7.15 那样,读脏数据的问题发生了。第二个事务看到一个元组(其中座位 S 标记为已占用)由第一个事务写过然后又由第一个事务修改了。

读脏数据到底有多大影响呢?在例 7.15 中,读脏数据会造成严重影响;尽管显而易见地想防止发生意外,但是读脏数据还是使一个帐户变成了负的。在例 7.16 中,问题看起来没有那么严重。的确,第二个旅行者没有得到他(她)想要的座位,或者甚至被告知没有座位了。然而,在后一种情况下,再次执行该事务时几乎可以肯定会显示出座位 S 是空的。为了减少订票请求的平均处理时间,以允许读脏数据的方式来实现 seat-choosing 函数是有意义的。

SQL2 允许我们对给定的事务规定可以读取脏数据。我们将使用在 7.2.4 节讨论的 SET TRANSACTION 语句。像在例 7.16 中描述的事务的合适形式是:

- 1) SET TRANSACTION READ WRITE
- 2) ISOLATION LEVEL READ UNCOMMITTED;

上面的语句做两件事情:

1. 1) 行说明了事务同时读和写数据。
2. 2) 行说明了下列事务可以用隔离性级别(isolation level)为读-不提交的方式运行。在 7.2.6 节我们将讨论 4 种隔离性级别。迄今为止,我们看到了其中的两种:可串行化和读-不提交。

注意:如果事务不是只读的(也就是,它至少往数据库中写入一个数据项),而且我们规定隔离性级别为 READ UNCOMMITTED,那么我们必须同时规定 READ WRITE。回想一下在 7.2.4 节中默认的假定为事务是读-写的。然而,SQL2 把允许读脏数据的情况作为一个异常。于是,默认的假定为事务是只读的,因为就像我们看到的那样,读脏数据时读-写事务要冒很大的风险。如果我们想让读-写事务以读-不提交的方式作为隔离性级别来运行,那么我们就要像上面那样,明确规定 READ WRITE。

7.2.6 其他隔离性级别

SQL2 总共提供了 4 种隔离性级别。我们已经看到了其中的两种:可串行化和读-不提交(允许读脏数据)。其他两种是读-提交和可重复读。对于给定的事务可如下规定这两种级别

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

或者

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

对于每种情况,都默认事务是读-写的,所以在适当的时候,我们可以给每个语句增加 READ ONLY。顺便提一下,我们也可以规定下列选项:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

然而,这是 SQL2 的默认设置,不必明确指出。

读-提交隔离性级别,就像它的名字所隐含的那样,禁止读脏的(没有提交的)数据。然而,它确实允许一个事务发出好几次同样的请求而得到不同的结果,只要结果反映的是已经提交的事务所写的数据就可以。

例 7.17 让我们再来考虑例 7.16 中的 seat-choosing 函数,但是假定函数是在读-提交的隔离性级别上运行的。于是,当函数在第一步寻找座位时,如果某个其他事务正在预订但是还未提交,那么,它就看不到预订的座位。然而,如果旅客拒绝预订的座位,并且该函数的一个执行过程可多次查询空座位,那么当其他与该事务并行的事务成功地预订到座位或者退掉预订的座位时,每次查询该函数将会看到不同的空座位集合。

现在,让我们考虑可重复读这种隔离性级别。可重复读多少有点误称,因为同样的查询执行多次并不能完全保证得到同样的结果。在隔离性为可重复读的情况下,如果第一次检索到一个元组,那么我们就能够确定如果重复查询的话,将再一次检索到该元组。然而,同样查询的第二次或者以后的执行也可能会检索到幻象(phantom)元组。后者是当我们的事务在执行时,插入到数据库中的元组。

例 7.18 让我们继续讨论在例 7.16 和 7.17 中的 seat-choosing 问题。如果我们在可重复读的隔离性级别下执行该函数,那么在第一次查询的步骤 1 中的空座位在以后的查询中将依然为空。

然而,假定一些新的元组进入了关系 Flights。例如,航线可能把班机换成比较大的飞机,从而产生了一些以前没有的元组,或者有的预订被取消了。于是,在隔离性为可重复读的情况下,随后对空座位的查询可能会检索到这些新座位。

7.2.7 本节练习

练习 7.2.1: 本练习和下一个练习包括对不断滚动的 PC 练习中的如下两个关系进行操作的某些程序:

Product (maker, model, type)

PC (model, speed, ram, hd, cd, price)

用嵌入式 SQL 和合适的宿主语言来描述下列程序。不要忘了在合适的时候给出

因为我们还没有描述实施不同隔离性级别的算法,所以实际发生的事情可能是难以理解的。两个事务可能同时看到一个座位空着并且都想预订它,其中一个事务将会被系统强制异常终止,即使它不希望执行 ROLLBACK 语句。

COMMIT 和 ROLLBACK 语句, 并告诉系统你的事务是只读的(如果它们是的话)。

- (a) 给出速度(speed)和 RAM 的容量(作为函数的参数), 查看具有该速度和 RAM 容量的 PC, 并打印每台 PC 的型号(model)和价格(price)。
- * (b) 给出一个型号, 从 PC 和 Product 两个关系中删除该型号的元组。
- (c) 给出一个型号, 将该型号的 PC 的价格减少 \$ 100。
- (d) 给出厂商(maker)、型号、处理器速度、RAM 容量、硬盘容量、光驱(CD)速度和价格, 检查有没有该型号的产品。如果有这种型号, 就给用户打印出错信息。如果没有这种型号, 就把该型号的信息输入到表 PC 和 Product 中。

! 练习 7.2.2: 对于 7.2.1 中每个程序, 讨论是否存在原子性问题。如果有的话, 在程序执行中, 一旦系统崩溃就会发生问题。

! 练习 7.2.3: 假定我们把练习 7.2.1 中的 4 个程序之一作为事务 T 来执行, 而执行 4 个程序中相同或者不同程序的其他事务也可能几乎同时执行。如果所有的事务不可能都运行在 SERIALIZABLE 隔离性级别下, 那么如果所有的事务都运行在 READ UNCOMMITTED 隔离性级别下, 可能看到事务 T 的什么情况呢? 分别考虑 T 是练习 7.2.1 中从(a)到(d)的任一个程序的情况。

* !! 练习 7.2.4: 假定我们有一个事务 T(一个“永远”运行的函数), 每小时都要检验是否有速度等于或高于 200 而售价低于 \$ 1 000 的 PC。如果找到了, 就打印出该信息并且终止该事务。在此期间, 执行练习 7.2.1 中的 4 个程序之一的其他事务都可以运行。对于 4 个隔离性级别中的每个——可串行化、可重复读、读-提交和读-不提交——说出在该隔离性级别上运行的事务 T 会有什么影响。

7.3 SQL 环境

在本节中, 我们将尽可能广泛地描述数据库管理系统和数据库以及数据库管理系统所支持的程序。我们将看到数据库是如何定义以及组织群集(cluster)、目录(catalog)和模式(schema)。我们也将看到程序是如何与它们要操作的数据联系起来的。许多细节取决于特定的实现, 所以我们将把重点放在 SQL2 标准所包含的一般概念上。

7.3.1 环境

SQL 环境是数据在其中可以存在和对数据的 SQL 操作可以执行的框架。实际上, 我们应该把 SQL 环境看作是运行在某个设备上的数据库管理系统。例如, ABC 公司买了 Dandy-DB 公司的 SQL 数据库管理系统的许可证, 从而使之能够运行在 ABC 公司的机器系列上。运行在这些机器上的系统就构成了 SQL 环境。

我们讨论过的所有数据库元素——表、视图、域和断言——都是在 SQL 环境中定义的。这些元素组成了层次结构, 在该体系结构中, 每个元素都起着不同的作用。SQL2 标准定义的结构如图 7.11 所示。

图 7.11 SQL 环境中数据库元素的体系结构

简单来说, 该组织由以下结构组成:

- 1. 模式(schema)。 它们是表、视图、断言、域和本书中没有讨论的其他类型信息(见 7.3.2 节的方框“ 模式中还有什么 ”)的聚集。模式是体系结构的基本单位, 与我们所想象的数据库类似, 但是事实上比下面第 3 点(群集)中我们将看到的数据库要小。
- 2. 目录(catalog)。它们是模式的聚集, 是支持唯一的、可访问的术语的基本单位。每个目录都有一个或者多个模式; 在一个目录中模式的名字必须是唯一的, 并且每个目录都包含着一个叫做 INFORMATION- SCHEMA 的特殊模式, 该模式包含着该目录中的所有模式的信息。
- 3. 群集(cluster)。它们是目录的聚集。每个用户都有相关的群集, 即该用户可以访问的所有目录的集合(要了解如何控制对目录和其他元素的访问可参看 7.4 节)。SQL2 没有明确描述什么是群集, 例如, 不同用户的群集如果不相同是否可以互相重叠。群集是可以查询的最大范围, 因此, 在某种意义上, 对于特定的用户来说, 群集就是“ 数据库 ”。

7.3.2 模式

模式说明的最简单格式由以下几部分组成:

- 1. 关键字 CREATE SCHEMA。
- 2. 模式的名字。
- 3. 基本表、视图、断言和域之类的模式元素的说明的列表。

也就是, 一个模式可以如下说明:

```
CREATE SCHEMA  模式名  元素说明
```

注意, “ 模式 ”这个术语在这段上下文中是指数据库模式, 而不是关系模式。

模式中还有什么

除了已经提到的表、视图、域和断言,还有 4 种其他模式元素。首先,一个模式可以指定字符集,它是符号的集合以及对其进行编码的方法。ASCII 是最著名的字符集,但是 SQL2 的实现可以支持许多其他字符集,例如各种外国语言的字符集。

第二,模式可以为一个字符集指定一个核对项(collation)。回忆一下 5.13 节,字符串按字典顺序进行比较,假定任何两个字符都能用我们表示为 < 的“小于”来进行比较。核对项规定哪些字符“小于”其他字符。例如,我们可以使用 ASCII 码隐含的顺序,或者我们可以对小写和大写字母同样处理,而且不比较任何不是字母的东西。

第三,模式可以有翻译,它是把一个字符集的字符转换为其他字符集的字符的方法。可能出现在模式中的最后一种元素是涉及到谁访问该模式的“授权语句”。在 7.4 节我们将讨论权限的授予。

元素说明的格式如 5.7 节、5.8 节和第 6 章所述。有些元素我们还没有描述其特性,但是 SQL2 允许在模式中对其加以说明;见方框“模式中还有什么”。

例 7.19 我们可以说明一个模式,它包括在不断滚动的实例中已经用到的关于电影的 5 个关系,加上我们已经介绍的某些其他元素,例如视图。图 7.12 简要地描述了这样一个说明的格式。

```
CREATE SCHEMA MovieSchema
CREATE DOMAIN CertDomain... 如例 6.8
    其他域的说明
CREATE TABLE MovieStar... 如图 6.4
    其他 4 个表的建表语句
CREATE VIEW MovieProd... 如例 5.40
    其他视图说明
CREATE ASSERTION RichPres... 如例 6.10
```

图 7.12 说明一个模式

不需要把模式说明全放在一起。读者可以用合适的 CREATE, DROP 或者 ALTER 语句来增加或者更改模式,例如,CREATE TABLE 跟着该模式的一个新表的说明。问题是 SQL 系统需要知道新表属于哪个模式。如果我们更改或者撤消一个表或者其他模式元素,那么,我们可能还需要使元素的名字无二义性,因为两个或者更多的模式可能有同样名字的不同元素。

我们用 SET SCHEMA 语句来改变“当前的”模式。例如,

```
SET SCHEMA MovieSchema;
```

使图 7.12 中描述的模式成为当前的模式,以便模式元素的任何说明都加到该模式中或者更改已经属于该模式的元素。

7.3.3 目录

在目录中建立和更改模式就像在模式中建立表之类的模式元素一样。原则上,我们

模式元素的全称

形式上, 模式元素(例如表)的名字是由它的目录名、它的模式名和它自己的名字用圆点依次连成的。于是, 其目录名为 MovieCatalog、模式名为 MovieSchema 的表 Movie 可以如下引用:

MovieCatalog. MovieSchema. Movie

如果目录是默认的或者是当前目录, 那么我们就可以省略这一部分。如果模式也是默认的或者是当前模式, 那么这一部分也可以省略, 从而就像通常那样只剩下元素自己的名字。然而, 如果需要访问在当前模式或者目录之外的某些东西, 可以选择使用全称。

希望建立和移植目录的过程类似于建立和移植模式的过程。遗憾的是, SQL2 没有定义一个标准的方式来这样做, 例如语句

CREATE CATALOG 目录名

后面跟着属于该目录的模式和这些模式的说明的列表。

然而, SQL2 确实规定了语句

SET CATALOG 目录名

该语句允许我们设置当前的目录, 于是新的模式将进入该目录并且模式更新将引用该目录中的模式, 而该目录的名字可能会有二义性。

7.3.4 SQL 环境中的客户程序和服务程序

SQL 环境不只是目录和模式的聚集。它还包括这样一些元素, 它们的目的是支持数据库的操作或者由这些目录和模式所体现的数据库的操作。在 SQL 环境中有两种特殊类型的处理程序: SQL 客户程序和 SQL 服务程序。服务程序支持对数据库元素的操作, 而客户程序则允许用户连接到服务程序上。可以想象, 服务程序运行在存有数据库的大型主机上, 而客户程序则运行在另一台主机上, 或许运行在离服务程序很远的个人工作站上。然而, 也有可能客户程序和服务程序运行在同一台主机上。

7.3.5 连接

如果我们希望在具有 SQL 客户程序的主机上运行包含 SQL 的某个程序, 那么可以通过执行 SQL 语句

CONNECT TO 服务程序名 AS 连接名

在客户程序和服务程序之间打开连接。服务程序名取决于设备。DEFAULT 这个词可以代替名字, 它将把用户连接到其设备是“默认服务器”的任何 SQL 服务程序上。

连接名可以在以后用来对连接进行引用。我们不得不引用连接的原因是 SQL2 允许用户打开几个连接, 但是在任何时间只能有一个是活动的。为了在连接之间切换, 我们用语句

SET CONNECTION conn1;

使 conn1 变成活动的连接。曾是当前活动的任何连接将变成待用状态,直到用另一个明确提到它的 SET CONNECTION 语句重新激活它。

当我们取消该连接时也要使用该名字。我们可以用

```
DISCONNECT conn1;
```

取消连接 conn1。现在,conn1 结束了;它不是待用而且不能重新激活。

然而,如果我们永远不需要引用正在建立的连接,那么 AS 和连接名就可以从 CONNECT TO 语句中省略掉。也允许完全跳过连接语句。如果我们在具有 SQL 客户程序的主机上单纯地执行 SQL 语句,那么默认的连接就会按我们的需要建立起来。

7.3.6 会话

当连接活动时,执行的 SQL 操作形成会话(session)。会话与建立它的连接同时扩展。例如,当连接待用时,它的会话也转为待用,而且 SET CONNECTION 语句所引发的连接的重新激活也使会话激活。所以在图 7.13 中我们把连接和会话表示成在客户程序和服务程序之间进行联络的两个方面。

图 7.13 SQL 客户程序-服务程序的相互作用

每个会话都有当前目录和该目录中的当前模式。像在 7.3.2 节和 7.3.3 节讨论的那样,它们可以用 SET SCHEMA 和 SET CATALOG 语句来设置。我们在 7.4 节将会讨论到,对于每个会话都有一个授权的用户。

7.3.7 模块

模块是用于应用程序的 SQL2 术语。SQL2 标准建议三种类型的模块,但是只坚持 SQL 实现至少为用户提供其中一种类型。

- 1. 通用 SQL 接口。用户可以坐下来并且敲入由 SQL 服务程序执行的 SQL 语句。在这种方式下,每个查询或其他语句都是独立的模块。我们为本书中大多数例子设想的就是这种方式,尽管实际上很少使用这种方式。
- 2. 嵌入式 SQL。这种类型在 7.1 节已经讨论了,SQL 语句出现在宿主语言程序中,并且由 EXEC SQL 引导。预处理程序把嵌入式 SQL 语句转换为适于 SQL 系统的函数或过程调用,当编译好的宿主语言程序执行时,在合适的时间执行这些调用。
- 3. 实际模块(true modules)。SQL2 所设想的模块的最常用的形式是存储函数或过程的聚集,其中一些是宿主语言代码,而另一些是 SQL 语句。它们之间通过传递参数也可能通过共享变量来进行通信。

模块的执行称为 SQL 代理(agent)。在图 7.13 中,我们给出了一个模块和一个 SQL 代理,并把两者作为一个整体调用 SQL 客户程序来建立连接。然而,我们应该记住,模块和 SQL 代理之间的差别类似于程序和处理之间的差别;第一个是代码,而第二个是执行那些代码。

7.4 SQL2 的安全和用户授权

SQL 要求存在授权 ID, 授权 ID 基本上用户名。SQL 还有一个特定的授权 ID, PUBLIC, 它包括任何用户。可以授予授权 ID 权限, 很像在文件系统环境中由操作系统维护权限那样。例如, UNIX 系统通常控制三种类型的权限: 读、写和执行。因为 UNIX 系统的保护对象是文件, 而且这三种操作正好描述了文件的典型操作, 所以该权限表是有意义的。然而数据库比文件系统复杂得多, 从而 SQL2 中使用的权限类型相应地更复杂。

在这一节中, 首先我们将了解对于数据库元素 SQL2 允许什么权限。然后我们将看到用户(也就是授权 ID)如何获得权限。最后, 我们将看到如何取消权限。

7.4.1 权限

SQL2 定义了 6 种类型的权限:

1. SELECT
2. INSERT
3. DELETE
4. UPDATE
5. REFERENCES
6. USAGE

其中前 4 种用于关系, 这里的关系或者是基本表或者是视图。就像它们的名字所隐含的那样, 它们分别给权限的拥有者以下权力: 查询(select from)关系, 插入到关系中, 从关系中删除, 以及修改关系的元组。如果没有与 SQL 语句相应的权限, 那么就不能执行包含该 SQL 语句的模块。例如, 一个 select -from -where 语句对它所访问的每个表都需要 SELECT 权限。我们很快就会看到模块如何得到这些权限。

REFERENCES 权限是引用在完整性约束下的关系的权力。这些约束可能采用第 6 章提到的任何一种形式, 例如断言、基于属性或元组的检验, 或者参照完整性约束。如果约束所在的模式不是对该约束有关的所有数据都有引用(REFERENCES)权限, 那么就不能对约束进行检验。

对于域, 或者除了关系和断言(见 7.3.2 节)以外的其他几种模式元素来说, USAGE 权限就是在自己的说明中使用该元素的权力。

人们可以给三种权限——INSERT, UPDATE 和 REFERENCES——加上单独的属性作为参数。在这种情况下, 权限只能引用提到的属性。人们可以控制每个都涉及一个属性的几种权限; 通过这种方式, 我们可以授权访问一个关系的列的任何子集。

例 7.20 让我们考虑执行图 5.12 的插入语句(我们在这里重新生成, 如图 7.14 所示)需要什么权限。首先, 因为要插入到关系 Studio 中, 所以我们需要对 Studio 的 INSERT 权限。然而, 因为插入只是针对属性 name 的相应分量, 所以具有关系 Studio 的 INSERT 权限或者 INSERT(name) 权限都是可以的。后一种权限允许我们插入 Studio 元组中的 name 分量, 同时让其他分量取它们的默认值或者 NULL, 也就是图 7.14 所做的。

```

1)  INSERT INTO Studio(name)
2)      SELECT DISTINCT studioName
3)      FROM Movie
4)      WHERE studioName NOT IN
5)          (SELECT name
6)              FROM Studio);

```

图 7.14 增加新的制片公司

然而注意: 图 7.14 中的插入语句包括从 2) 行和 5) 行开始的两个子查询。为了执行这些选择, 我们要求子查询所需的权限。这样, 我们就需要对 FROM 子句中包含的两个关系 Movie 和 Studio 的 SELECT 权限。注意, 这只是因为我们拥有对 Studio 的插入 (INSERT) 权限并不意味着我们拥有对 Studio 的选择 (SELECT) 权限, 反之亦然。

7.4.2 建立权限

我们已经明白 SQL2 权限是什么而且注意到它们对于执行 SQL 操作是需要的。现在我们必须学习如何获得执行某个操作所需要的权限。授权有两个方面: 权限最初是如何建立的, 以及它们是如何从一个用户传递到另一用户的。在这里我们将讨论初始化, 在 7.4.4 节我们将讨论权限的传递。

首先, 像模式或模块这样的 SQL 元素都有一个拥有者。某个事物的拥有者拥有与该事物有关的所有权限。在 SQL2 中有三个时刻可以建立所有权。

1. 当建立模式时, 就假定模式以及其中所有的表和其他模式元素都为建立它的用户所拥有。于是该用户对该模式的元素就具有所有可能的权限。

2. 当 CONNECT 语句把会话初始化时, 有机会用 USER 子句来指定用户。例如, 连接语句

```
CONNECT TO Starfleet-sql-server AS conn1 USER kirk;
```

将为用户 kirk 建立一个到 SQL 服务程序的连接, 连接名为 conn1, SQL 服务程序的名字叫 Starfleet-sql-server。通常, SQL 的实现将验证用户名是否合法, 例如查询一个口令。

3. 当建立模块时, 可以用 AUTHORIZATION 子句作为选项给模块指定一个拥有者。我们将不深入到模块建立的细节中, 因为 SQL2 标准在这方面允许其实现有相当大的灵活性。然而, 我们可以想象模块建立语句中的子句

```
AUTHORIZATION picard;
```

将使用户 picard 成为该模块的拥有者。当模块可公开执行时, 可以不指定模块的拥有者, 但是在该模块中执行任何操作所必须的权限必须从其他某个来源 (例如在该模块执行期间与连接和会话有关的用户) 得到。

7.4.3 权限检验处理

像我们在上面所看到的, 每个模块、模式和会话都有一个相关的用户; 用 SQL 术语来说, 其中每个都有一个相关的授权 ID。任何 SQL 操作都有两个部分:

1. 在其上执行操作的数据库元素;
2. 使操作得以执行的代理。

代理可行使的权限从称为当前授权 ID 的特定授权 ID 得到。如果代理执行的模块具有授权 ID, 那么特定授权 ID 就是(a), 否则就是(b):

- (a) 模块授权 ID;
- (b) 会话授权 ID。

只有当当前授权 ID 拥有了执行操作所需要的一切权限时, 我们才能执行 SQL 操作。

例 7.21 为了理解检验权限的机制, 让我们再来看一下例 7.20。我们可以假定引用的表——Movie 和 Studio——是模式 MovieSchema 的一部分, 用户 janeway 建立并拥有该模式。此时, 用户 janeway 拥有对模式 MovieSchema 中的这些表和任何其他元素的一切权限。她可以用 7.4.4 节所描述的机制把一些权限授予其他用户, 但是我们假定还没有授予其他用户。例 7.20 中的插入操作可以用几种方式执行。

1. 插入可以作为由用户 janeway 建立并含有 AUTHORIZATION janeway 子句的模块的一部分来执行。模块授权 ID(如果有的话)总是成为当前的授权 ID。于是, 该模块及其 SQL 插入语句就具有与用户 janeway 完全相同的权限, 包括对表 Movie 和 Studio 的一切权限。

2. 插入可以成为没有拥有者的模块的一部分。用户 janeway 在 CONNECT TO 语句中用 USER janeway 子句打开连接。现在, janeway 同样成为当前授权 ID, 于是插入语句具有所需要的一切权限。

3. 用户 janeway 将对表 Movie 和 Studio 的所有权限授予用户 sisko, 或者也许给特殊用户 PUBLIC(它代表“所有用户”)。插入语句处于具有子句

AUTHORIZATION sisko

的模块中。既然当前授权 ID 现在是 sisko, 并且该用户具有所需要的权限, 所以插入也是允许的。

4. 像在方案 3 中那样, 用户 janeway 给用户 sisko 所需要的权限。插入语句在没有拥有者的模块中; 它是在一个会话中执行的, 而该会话的授权 ID 是通过 USER sisko 子句设置的。这样当前的授权 ID 是 sisko, 而且该 ID 拥有所需要的权限。

例 7.21 阐明了几个原则。我们总结如下。

- 如果拥有数据的用户, 其 ID 就是当前授权 ID, 那么所需要的权限总是可用的。上面的方案 1 和方案 2 说明了这一点。
- 如果用户的 ID 是当前授权 ID, 而且数据的拥有者已经把所需要的权限授予了该用户, 或者如果已经把所需要的权限授予了用户 PUBLIC, 那么这些权限就是可用的。方案 3 和方案 4 说明这一点。
- 执行数据的拥有者所拥有的模块或者执行对数据已有授权的某个用户所拥有的模块, 就使得所需要的权限是可用的。方案 1 和方案 3 说明了这一点。
- 在会话(它的授权 ID 是具有所需要权限的用户)期间执行公用模块是合法执行该操作的另一种方法。方案 2 和方案 4 说明了这一点。

7.4.4 授予权限

在例 7.21 中, 我们看到拥有所需要的权限对用户(也就是授权 ID)的重要性。但是到现在为止, 我们看到对数据库元素拥有权限的唯一方法就是成为该元素的建立者和拥有者。SQL2 提供了 GRANT 语句允许一个用户把权限授予另一个用户。第一个用户还保留所授予的权限; 于是可以把 GRANT 看成是“复制权限”。

在授予的和复制的权限之间有一个重要的区别。每个权限都有相关的授权选项。也就是, 一个用户可能对表 Movie 具有带“授权选项”的 SELECT 权限, 尽管第二个用户可能具有同样的权限, 但是不带授权选项。于是, 第一个用户可以把对 Movie 的 SELECT 权限授予第三个用户, 而该授权可能具有也可能不具有授权选项。然而, 没有授权选项的第二个用户就不能把对 Movie 的 SELECT 权限授予其他任何用户。如果第三个用户后来得到了带授权选项的同样权限, 那么该用户可以把权限授予第四个用户, 仍然可能具有也可能不具有授权选项, 依此类推。授权语句由以下几部分组成:

- 1. 关键字 GRANT。
- 2. 一个或多个权限的列表, 例如, SELECT 或者 INSERT(名字)。可选的关键字 ALL PRIVILEGES 可以在这里出现, 作为授权者对所讨论的数据库元素(在下面第 4 项提到的元素)合法地授予所有权限的简写。
- 3. 关键字 ON。
- 4. 一个数据库元素。典型的元素是关系, 也就是, 或者是基本表或者是视图。它也可能是域或者我们没讨论过的其他元素(见 7.3.2 节的方框“模式中还有什么”), 但是在这些情况下, 元素名之前必须是关键字 DOMAIN 或者另一个合适的关键字。
- 5. 关键字 TO。
- 6. 一个或多个用户(授权 ID)的列表。
- 7. 可选的关键字 WITH GRANT OPTION。

也就是, 授权语句的格式是:

GRANT 权限表 ON 数据库元素 TO 用户表

可能还跟着 WITH GRANT OPTION。

为了合法地执行授权语句, 执行该语句的用户必须拥有所要授予的权限, 而且这些权限必须带授权选项。然而, 授权者可以拥有比所要授予的权限更一般的权限(带授权选项)。例如, 授权者可以授予对表 Studio 的 INSERT(名字)权限, 然而仍保留对 Studio 更一般的带授权选项的 INSERT 权限。

例 7.22 MovieSchema 模式包含表

Movie(title, year, length, inColor, studioName, producerC#)
Studio(name, address, presC#)

它的拥有者用户 janeway 把对 Studio 表的 INSERT 和 SELECT 权限以及对 Movie 表的 SELECT 权限授予用户 kirk 和 picard。而且, 授权的内容还包括这些权限的授权选项。授权语句是:


```
GRANT SELECT, INSERT ON Studio TO kirk, picard
WITH GRANT OPTION;
GRANT SELECT ON Movie TO kirk, picard
WITH GRANT OPTION;
```

现在, picard 将同样的权限授予用户 sisko, 但是不带授权选项。语句如下:

```
GRANT SELECT, INSERT ON Studio TO sisko;
GRANT SELECT ON Movie TO sisko;
```

同样, kirk 授予 sisko 为进行图 7. 14 的插入所需要的最小权限, 也就是对 Studio 表的 SELECT 和 INSERT(名字) 以及对 Movie 表的 SELECT 权限。语句如下:

```
GRANT SELECT, INSERT(名字) ON Studio TO sisko;
GRANT SELECT ON Movie TO sisko;
```

注意 sisko 从两个不同的用户得到了对表 Movie 和 Studio 的 SELECT 权限。他同样两次得到对 Studio 的 INSERT(名字) 权限: 直接从 kirk 得到以及通过一般性权限 INSERT 从 picard 得到。

7. 4. 5 授权图

因为一系列授权可能产生由授权和重叠的权限形成的复杂网络, 因此用授权图来表示授权是十分有用的。SQL 系统维护着这张图的映象以记录权限及其来龙去脉;(取消权限的情况, 见 7. 4. 6 节)。这张图的节点对应于一个用户和一个权限。如果用户 U 把权限 P 授予用户 V, 而且该授权基于这样一个事实: U 拥有权限 Q(Q 是带授权选项的 P, 或者也可以是 P 的某种广义形式, 同样带授权选项), 那么我们就从节点 U/Q 往节点 V/P 画一个弧。

例 7. 23 图 7. 15 是由例 7. 22 的授权语句序列产生的授权图。我们使用在用户-权限组合后面跟一个星号(*) 这样的约定来表示带授权选项的权限。同样, 在用户-权限组合后面跟两个星号(**) 表示权限是从所讨论的数据库元素的所有权派生出来的, 而不是靠其他途径的授权。有两个星号的权限自动包括授权选项。

7. 4. 6 取消权限

任何时候都可以把授予的权限取消。实际上, 带授权选项的权限可能已传递给其他用户, 要取消这种权限就需要把这些用户的权限也取消, 在这个意义上, 权限的取消可能需要级联过程(cascade)。取消语句的简单格式是:

1. 关键字 REVOKE。
2. 一个或者多个权限的列表。
3. 关键字 ON。
4. 一个数据库元素, 就像在授权语句组成部分的第 4 项所讨论的那样。
5. 关键字 FROM。
6. 一个或者多个用户(授权 ID)的列表。

也就是, 取消语句的格式如下:

```
REVOKE 权限表 ON 数据库元素 FROM 用户表
```

图 7.15 授权图

然而, 该语句也可能包含如下各项:

- 语句可以用 CASCADE 结尾。如果这样, 那么当取消指定的权限时, 我们也取消了只由要取消的权限所授予的任何权限。更确切地说, 如果用户 U 取消了用户 V 的权限 P, 而权限 P 基于属于 U 的权限 Q, 那么我们将删除授权图中从 U/Q 到 V/P 的弧线。现在, 还要删除不能从某个所有权节点(双星号节点)访问到的任何节点。
- 语句可以用 RESTRICT(限制)作为结尾, 这意味着, 如果由于要取消的权限已传递给其他用户, 而按前一项描述的级联规则将导致取消这些权限的话, 那么该取消语句将不能执行。
- 允许用 REVOKE GRANT OPTION FOR 代替 REVOKE, 在这种情况下, 权限本身仍然保留, 但是把授权给其他用户的选项取消了。这种选择可以与 CASCADE 或者 RESTRICT 相结合, 在这种情况下, 将检验授权图看是否需要取消其他已授予的权限。

例 7.24 继续例 7.22, 假定 janeway 用下列语句

```
REVOKE SELECT, INSERT ON Studio FROM picard CASCADE;  
REVOKE SELECT ON Movie FROM picard CASCADE;
```

取消了授予 picard 的权限。

我们删除图 7.15 中从 janeway 的这些权限到 picard 的相应权限的弧线。由于级联(CASCADE)规则, 我们还应该看图中是否有任何权限不能到达双星号的(基于所有权的)权限。检验图 7.15, 我们看到从双星号的节点不再能到达 picard 的权限(如果有另一条路径

到达 pircard 节点, 那么权限可能还存在)。同样, 不再能到达 sisko 对表 Studio 的 INSERT 权限。! 这样, 我们不仅从授权图中删除 picard 的权限, 而且删除 sisko 的插入权限。

注意, 我们没有删除 sisko 对表 Movie 和 Studio 的 SELECT 权限以及对 Studio 的 INSERT(名字)权限, 因为它们通过 kirk 的权限都能到达 janeway 的基于所有权的权限。得到的授权图如图 7.16 所示。

图 7.16 取消了 picard 权限后的授权图

例 7.25 我们将用抽象的例子来说明一些细微的区别。首先, 当我们取消一个一般权限 p 时, 并没有同时取消权限 p 的特殊情况。例如, 考虑下列步骤序列, 通过用户 U(关系 R 的所有者), 把对关系 R 的 INSERT 权限授予用户 V, 并且把对同一关系的INSERT (A)权限授予 V。

步骤	通过	动 作
1	U	GRANT INSERT ON R TO V
2	U	GRANT INSERT(A) ON R TO V
3	U	REVOKE INSERT ON R FROM V RESTRICT

当 U 取消 V 的 INSERT 权限时, INSERT(A) 权限仍然保留。步骤 2 和 3 之后的授权图如图 7.17 所示。

注意: 第二步之后出现两个独立的节点代表用户 V 的相似但有区别的权限。同样可以发现步骤 3 中的 RESTRICT 选项没有阻止取消权限, 因为 V 并没有给其他任何用户授权的选项。事实上, V 不能授予任何权限, 因为 V 得到的是不带授权选项的权限。

图 7.17 取消一般的权限留下更具体的权限

例 7.26 现在, 让我们考虑一个相似的例子: U 授予 V 带授权选项的权限 p, 然后只取消授权选项。在这种情况下, 我们必须改变 V 的节点来反映授权选项的丢失, 并且必须通过去掉来自 V/P 节点的弧线来表明取消由 V 授予的任何权限 p。步骤序列如下:

步骤	通过	动 作
1	U	GRANT p TO V WITH GRANT OPTION
2	U	GRANT p TO W
3	U	REVOKE GRANT OPTION FOR p FROM V CASCADE

步骤 1, U 把带授权选项的权限 p 授予 V。步骤 2, V 利用授权选项把 p 授予 W。于是授权图如图 7.18(a) 所示。

然后是步骤 3, U 取消了 V 的权限 p 的授权选项, 但是没有取消权限本身。这样, 星号就从 V/p 的节点去掉了。然而, 没有星号的节点就不会有弧线引出, 因为这样的节点不可能成为授权的来源。所以我们还必须去掉从 V/p 节点到 W/p 节点的弧线。

现在, 节点 W/p 没有一条从双星号节点(它代表权限 p 的起点)来的路径。作为结果, 把节点 W/p 从图中删除。然而, 节点 V/p 仍然保留, 只是通过去掉代表授权选项的星号做了更改。最后得到的授权图如图 7.18(b) 所示。

图 7.18 取消授权选项留下基本的权限

7.4.7 本节练习

练习 7.4.1: 指出执行下列查询需要什么样的权限。对于每种情况, 都要提到最具体的权限和一般的权限。

(a) 图 5.3 的查询。

- (b) 图 5.5 的查询。
- * (c) 图 5.12 的插入。
- (d) 例 5.29 的删除。
- (e) 例 5.31 的修改。
- (f) 图 6.4 基于元组的检验。
- (g) 例 6.10 的断言。

* 练习 7.4.2: 画出图 7.19 列出的动作序列的步骤 4 到 6 以后的授权图。假定 A 是权限 p 所涉及的关系的拥有者。

步骤	通过	动 作
1	A	GRANT p TO B WITH GRANT OPTION
2	A	GRANT p TO C
3	B	GRANT p TO D WITH GRANT OPTION
4	D	GRANT p TO B,C,E WITH GRANT OPTION
5	B	REVOKE p FROM D CASCADE
6	A	REVOKE p FROM C CASCADE

图 7.19 练习 7.4.2 的动作序列

练习 7.4.3: 画出图 7.20 列出的动作序列的步骤 5 和 6 执行完以后的授权图。假定 A 是权限 p 所涉及的关系的拥有者。

步骤	通过	动 作
1	A	GRANT p TO B,E WITH GRANT OPTION
2	B	GRANT p TO C WITH GRANT OPTION
3	C	GRANT p TO D WITH GRANT OPTION
4	E	GRANT p TO C
5	E	GRANT p TO D WITH GRANT OPTION
6	A	REVOKE GRANT OPTION FOR p FROM B CASCADE

图 7.20 练习 7.4.3 的动作序列

! 练习 7.4.4: 画出下面这些步骤之后的最后授权图, 假定 A 是权限 p 所涉及的关系的拥有者。

步骤	通过	动 作
1	A	GRANT p TO B WITH GRANT OPTION
2	B	GRANT p TO B WITH GRANT OPTION
3	A	REVOKE p FROM B CASCADE

7.5 本 章 总 结

嵌入式 SQL: 不使用通用查询接口来表达 SQL 查询和更新, 而是把 SQL 查询嵌

入到传统的宿主语言中,这样编写的程序通常更有效。

匹配失衡: SQL 的数据模型与传统的宿主语言的数据模型有很大的不同。因此,信息就通过共享变量在 SQL 和宿主语言之间进行传递,而共享变量代表程序中 SQL 部分的元组分量。

游标: 游标是指向关系的一个元组的 SQL 变量。尽管可以把当前元组的分量取到共享变量并通过宿主语言进行处理,但是通过游标覆盖该关系的每个元组使得宿主语言和 SQL 之间的连接变得更加方便。

动态 SQL: 不是在宿主语言程序中嵌入特定的 SQL 语句,而是宿主程序可以产生字符串,这些字符串由 SQL 系统作为 SQL 语句进行解释和执行。

并发控制: SQL2 提供两种机制以防止并发操作互相干扰: 事务和对游标的限制。对游标的限制包括说明游标为“不敏感的”(insensitive),在这种情况下,游标看不到关系的任何变化。

事务: SQL 允许程序员把 SQL 语句组成事务,它可能提交,也可能退回(异常终止)。在后一种情况下,将取消事务对数据库所做的任何改变。

隔离性级别: SQL2 允许事务在四种隔离性级别上运行,从最严格的到最不严格的分别称为:“可串行化”(事务必须看来好像完全在某个其他事务之前或者之后运行),“可重复读”(如果重复查询,那么响应该查询而读出的每个元组都将再次出现),“读-提交”(只有已经提交的事务才能看到该事物所写的元组),和“读-不提交”(对于事务可能看到什么没有限制)。

只读游标和事务: 可以把游标或者事务说明为只读的。该说明是该游标或者事务将不改变数据库的保证,于是就可以通知 SQL 系统: 它不会在违背不敏感性、可串行性或者其他要求的情况下影响其他游标或者事务。

数据库的体系结构: 利用具有 SQL2 DBMS 的设备建立 SQL 环境。在该环境中,像关系这样的数据库元素将组成数据库模式、目录和群集。目录是模式的聚集,群集是用户可以看到的元素的最大聚集。

客户程序/服务程序系统: SQL 客户程序连接到 SQL 服务程序,建立一个连接(两个进程之间的连接)和一个会话(操作的序列)。会话期间执行的代码来自模块,该模块的执行称为 SQL 代理。

权限: 出于安全的考虑,SQL2 允许对数据元素有许多不同种类的权限。这些权限包括查询(读)、插入、删除或修改关系的权限以及引用关系(在约束中引用它们)的权限。还可以按关系的特定列获得插入、修改和引用的权限。

授权图: 权限可以由拥有者授予其他用户或者一般用户 PUBLIC。如果授权时带授权选项,那么这些权限可以再传给其他用户。也可以取消权限。授权图是一种有用的方式,它可以完全记住授权和取消的历史,从而确定谁拥有什么样的权限以及他们是从哪里得到这些权限的。

7.6 本章参考文献

读者要再一次参考第 5 章文献目录的注释以获得 SQL2 标准的信息。[1] 中给出了在事务和游标领域关于该标准问题的讨论。

实现事务的最重要的“二段锁”思想在[3]中提及。关于事务管理和实现的更多信息,看[2]和[5]。关于 SQL2 授权的概念来源于[6]和[4]。

- [1] Berenson, H., P. A. Bernstein, J. N. Gray, J. Melton, E. O' Neil, and P. O' Neil, A critique of ANSI SQL isolation levels, Proceedings of ACM SIGMOD Intl. Conf. on Management of Data, pp. 1 ~ 10, 1995.
- [2] Bernstein, P. A., V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.
- [3] Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, The notions of consistency and predicate locks in a database system., Communications of the ACM, 19: 11, pp. 624 ~ 633, 1976.
- [4] Fagin, R., On an authorization mechanism, ACM Transactions on Database Systems, 3: 3, pp. 310 ~ 319, 1978.
- [5] Gray, J. N. and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan-Kaufmann, San Francisco, 1993.
- [6] Griffiths, P. P. and B. W. Wade, An authorization mechanism for a relational database system, ACM Transactions on Database Systems, 1: 3, pp. 242 ~ 255, 1976.

第 8 章 面向对象查询语言

在这一章,我们将讨论把面向对象的编程引进数据库世界的两种尝试。OQL 和 SQL3 这两种语言的标准正在形成之中,并没有广泛实现,但是它们正为人们所接受,同时其思想正在迅速地渗入到商业系统中。

OQL,即对象查询语言(Object Query Language),是这样一种尝试,它希望把面向对象的查询语言标准化为一种语言,这种语言能将高级的、说明性的 SQL 编程和面向对象的编程范例结合起来。我们在本章开始先讨论 ODL 中的方法和范围(ODL 即第 2 章中作为建模工具介绍的对象定义语言)。这两个特性对查询语言 OQL 有着重要的影响。然后我们将介绍 OQL 编程的许多内容。

如果说 OQL 试图把 SQL 的精华引进面向对象的世界,那么就可以把 SQL3 描述为把面向对象的精华引进关系的世界。在某种意义上,这两种语言“在中间相遇”,但是两者在方法上也存在显著差别,以至于某些东西用一种语言比用另一种更容易实现。所以,在介绍了 SQL3 的推荐标准的面向对象特性之后,我们将对这两种语言的能力进行比较。

实质上,这两种面向对象方法的不同之处在于,它们对“关系有多么重要”这个问题的回答不同。对于以 ODL 和 OQL 为中心的面向对象的群体,答案是“不很重要”。于是,在这种方法中,我们可以找到所有类型的对象,其中一部分是集合或者结构(也就是关系)的包。对于 SQL3 群体,答案是关系仍然是基本的数据建构概念。在通常称为对象关系的 SQL3 方法中,把关系模型扩充了,它允许关系的元组和属于关系属性的域有更为复杂的类型。这样,对象和类都引入到关系模型中,但总是处在一个关系中。

8.1 ODL 中相关查询的特性

在这一节,我们将继续第 2 章中关于 ODL 的讨论。首先我们将讨论 ODL 类与其所处的更大编程环境之间交互方式的问题。然后我们将讨论类扩充的问题,对于 OQL 来说它将起到类似于关系在 SQL 中的作用。

8.1.1 ODL 对象的操作

我们很快将会看到,OQL 是这样一种语言,它允许我们像 SQL 那样来表示具有关系的或者基于集合行为的操作。然而在许多情况下,还需要执行不基于集合的其他操作。例如,如果对象是文档,我们可能希望检验在给定的文档中是否包含给定的关键字。如果对象是地图或者图片元素,我们也许希望在合适的位置显示该对象。即使是传统的面向记录的数据(比如我们不断滚动的电影实例)也受益于某些特殊的操作(比如产生某个给定影

为什么要署名?

提供的署名值是当我们用真正的编程语言来实现模式时,我们能够自动检验实现与用模式表示的设计是否匹配。我们不能检验实现能够准确完成操作的“意义”,但是我们至少能够检验输入和输出参数的准确数目和正确类型。

星每年所主演电影数量的统计图表)。

由传统的或者宿主的编程语言(例如 C)编写的程序执行这些 SQL 操作,在程序中 SQL 语句是嵌入式的。值通过 7.1 节所介绍的机制在 SQL 变量和宿主语言变量之间进行传递。

ODL 定义和宿主语言之间是紧密结合的。假定宿主语言是 C++ 或者 Smalltalk 这样面向对象的语言。每种语言都非常类似于 ODL,可以把 ODL 说明直接转换为宿主语言说明。更进一步说,表示对象的宿主语言变量,很容易表示 ODL 语句中说明的对象。

为了使 ODL 说明和 OQL 查询与宿主语言之间的结合更加方便,ODL 允许第三种特性(除了属性和联系):方法。方法是与类相关的函数。它应用于该类的对象并可能有一个或多个其他参数。我们将会看到方法可以用在 OQL 中,似乎它们是类的属性。

8.1.2 ODL 中方法署名的说明

在 ODL 中,我们可以说明与类相关的方法的名字和这些方法的输入/输出类型。这些说明称为署名,如同 C 或者 C++ 中的函数说明(与函数定义不同,函数定义给出实现该函数的代码)。方法的实际代码是用宿主语言写的,它的代码不是 ODL 的一部分。

方法说明与接口说明中的属性和联系一起出现。每种方法都和一个类(也就是和一个接口)相关,同时由该类的对象所引用,这种形式是面向对象语言的标准形式。这样,对象就是该方法的隐式自变量。这种形式允许同一方法名用于几个不同的类,因为执行操作的对象决定了方法的特定意义。这样的方法名称为重载(作为多个类的方法出现)。

方法说明的语法类似于 C 的函数说明,这里有两点重要补充:

1. 规定函数参数为输入、输出或者输入输出,意味着它们分别作为输入参数、输出参数或者输入输出参数。函数可以修改后两种参数;而不能修改输入参数。实质上,输出和输入输出参数是通过引用传递的,而输入参数则通过值来传递。注意,函数可能有返回值,它是由函数产生的结果,而不是由赋值给输出或者输入输出参数产生的。

2. 函数可能引发异常,这些异常是正常的参数传递之外的特殊响应和函数间互相通信的响应机制。异常通常指非正常或者非希望的情况,这种情况将通过调用它(或许通过一系列的调用间接调用它)的某个函数来“处理”。除以 0 就是可以作为异常情况的例子。ODL 函数说明后面可以跟着关键字 `raises`(引发),随后的括号里是该函数可能引发的一个或者多个异常的列表。

回忆一下 SQL,宿主语言的变量类型(像整数)不能很好地匹配 SQL 的基本数据类型(元组和关系)。如 7.1 节中我们所看到的,SQL 和宿主语言的结合相当不便。

例 8.1 在图 8.1 中,我们将看到图 2.6 所示的类 Movie 的扩充的接口定义。这里有两个与方法无关的改动。

- 1. 2) 行的“范围说明”。该语句的目的将在 8.1.3 节加以说明。
- 2. 3) 行说明 title 和 year 是 Movie 的键码。

接口说明中包括的方法如下所示。10) 行说明了方法 lengthInHours。我们可以认为该方法将产生一个返回值,该返回值就是应用该方法的电影对象的长度,而长度为从用分钟表示(和属性 length 所表示的一样)转换为用小时表示的等值浮点数。注意该函数没有参数。应用该方法的 Movie 对象是“隐式”自变量,根据该对象 lengthInHours 的可能实现将获得用分钟表示的电影长度。

我们还将看到该函数可能引发名为 noLengthFound 的异常。如果应用方法 lengthInHours 的对象的 length 属性值无定义或者不代表有效长度(例如是负数),那么估计就会引发这种异常。

```
1) interface Movie
2)   (extent Movies
3)   key(title, year))
4)   {
5)   attribute string title;
6)   attribute integer year;
7)   attribute integer length;
8)   attribute enumeration(color, blackAndWhite) filmType;
9)   relationship Set Star stars
      inverse Star starredIn;
10)  relationship Studio ownedBy
      inverse Studio owns;
11)  float lengthInHours() raises( noLengthFound);
12)  starNames(out Set String );
13)  otherMovies(in Star, out Set Movie )
      raises(noSuchStar);
14)  };
```

图 8.1 向 Movie 类增加方法署名

记住,说明中并不需要方法来完成其名字所隐含的事情。例如,不管方法应用于什么样的 Movie 对象,都用总是返回 3.14159 的函数来实现方法 lengthInHours,人们都会认为是正确的。我们也可以这样实现该方法:返回转换为浮点数的长度的平方。只要无自变量(除了应用它的对象)、返回一个浮点数而且除了 noLengthFound 之外没有其他异常,任何函数都可以接受。

我们在 11) 行看到另一个方法署名;该署名是名为 starNames 的函数。该函数没有任何返回值,但是有一个其类型是字符串集合的输出参数。我们可以假定输出参数的值由该函数计算成字符串的集合,该字符串集是应用该函数的电影明星的属性 name 的值。然而同以前一样,这里并没有保证所实现的函数以这种特定的方式执行。

最后, 12) 行是第三个方法, otherMovies。该函数的输入参数类型为 Star。该函数的可能实现如下。我们可以假定 otherMovies 希望该影星成为这部电影中的明星之一,

如果不是这样, 那么就会引发异常 noSuchStar。如果他是应用该方法的电影的影星之一, 那么就将给出该影星所有其他电影的集合, 而把它作为其类型为电影集合的输出参数的值。

8.1.3 类的范围

每个 ODL 类(接口)可以有一个说明的范围, 它是该类对象的当前集合名。说明的格式是关键字 extent 跟着为范围选择的名称。范围说明必须紧跟在接口(类)名说明之后。

在某种意义上, 类的范围类似于关系名, 而类定义本身则类似于该关系属性的类型说明。我们将会看到 OQL 查询引用的是类的范围, 而不是类名本身。

例 8.2 图 8.1 的 2) 行给出了类 Movie 的范围定义。该范围的名称是 Movies。Movies 的值在任何时候都是数据库中此刻存在的所有 Movie 对象的集合。

8.1.4 本节练习

练习 8.1.1: 图 8.2 是我们不断滚动的产品练习的 ODL 描述。我们已经使得这 3 种类型产品的每类都成为主 Product 类的子类。读者将会发现: 产品的类型既可以从 type 属性得到, 也可以从它所属的子类得到。这种安排不是一个出色的设计, 因为它允许这种可能性, 比如说, PC 对象将具有与“laptop”或者“printer”相同的类型(type)属性。然而, 这种安排对如何表达查询给出一些有趣的选项。

因为 Printer 从它的超类 Product 继承了类型(type), 所以我们将不得不将 Printer 的类型属性改名为 printerType。后一个属性给出了打印机所使用的处理方式(例如, 激光或者喷墨), 而 Product 的类型则具有像 PC、便携式电脑或者打印机之类的值。

把适合于做以下事情的函数作为方法署名加到图 8.2 的 ODL 代码中。

- * (a) 从产品的价格中减去 x。假设 x 是函数的输入参数。
- * (b) 如果产品是“PC”或者“laptop”, 则返回它的速度。否则就引发异常“notComputer”。
- (c) 设置便携式电脑的的屏幕尺寸为特定的输入值 x。
- ! (d) 给出输入产品 p, 确定应用该方法的产品 q 是否比 p 的速度更快而价格更低。
如果 p 是没有速度的产品(也就是, 不是 PC 机或便携式电脑)就引发异常 badInput, 而如果 q 是没有速度的产品, 则引发异常 noSpeed。

练习 8.1.2: 图 8.3 是我们不断滚动的战列舰(battleships)数据库的 ODL 描述。增加下列方法署名:

- (a) 计算一艘舰艇的火力, 也就是火炮的数量乘以火炮口径的立方。
- (b) 找出一艘舰艇的姐妹舰艇。如果该等级的舰艇只有一艘就引发异常 noSisters。
- (c) 给出一次战役 b 作为参数, 并且把该方法应用于舰艇 s。找出战役 b 中沉没的舰艇, 假设 s 参加该战役。如果舰艇 s 没有参加战役 b, 就引发异常 didNotParticipate。
- (d) 给出名字和下水年份作为参数, 把该名字和年份的舰艇加到应用该方法的类中。

```
interface Product
    (extent Products
     key model)
{
    attribute integer model;
    attribute          string
    manufacturer;
    attribute string type;
    attribute real price;
};
interface PC      Product
    (extent PCs)
{
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
    attribute string cd;
};
interface Laptop   Product
    (extent Laptops)
{
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
    attribute real screen;
};
interface Printer  Product
    (extent Printers)
{
    attribute boolean color;
    attribute string printertype;
};
```

图 8.2 ODL 的产品(Product)模式

```
interface Class
    (extent Classes
     key name)
{
    attribute string name;
    attribute string country;
    attribute integer numGuns;
    attribute integer bore;
    attribute integer displacement;
    relationship Set Ship ships inverse Ship classOf;
};
interface Ship
    (extent Ships
     key name)
{
    attribute string name;
    attribute integer launched;
    relationship Class classOf inverse Class ships;
    relationship Set Outcome inBattles
                        inverse Outcome theShip;
};
interface Battle
    (extent Battles
     key name)
{
    attribute string name;
    attribute Date dateFought;
    relationship Set Outcome results
                        inverse Outcome theBattle;
};
interface Outcome
    (extent Outcomes)
{
    attribute enum Stat {ok, sunk, damaged} status;
    relationship Ship theShip inverse Ship inBattles;
    relationship Battle theBattle inverse Battle results;
};
```

图 8.3 ODL 的战列舰数据库

8.2 OQL 介绍

在这一节,我们将介绍 OQL(Object Query Language),即对象查询语言。我们在本节和下面两节涉及的范围比 SQL 中的范围稍有扩大。我们将解释那些最重要的语句和特性,但是 OQL 还有许多其他能力。通常这些能力类似于 SQL 或者典型的面向对象传统编程语言中的相应特性。

OQL 不允许我们像传统编程语言 C 那样表达任意函数。相反,OQL 提供给我们类似 SQL 的表示法,这种表示法比传统语言的典型语句在更高级的抽象层次上表达特定的查询。人们的意图是把 OQL 作为某个面向对象的宿主语言(例如 C++、Smalltalk 或者

Java) 的扩充。这些对象将由 OQL 查询和宿主语言的传统语句进行操作。将宿主语言语句和 OQL 查询混合起来而不是在两种语言之间显式传递值的能力比起将 SQL 嵌入到宿主语言中(如 7.1 节所讨论的)这种方式是一个进步。

8.2.1 面向对象的电影实例

为了说明 OQL 的句法, 我们需要不断滚动的电影实例。它包括熟悉的类 Movie、Star 和 Studio。我们将使用图 8.1 中 Movie 的定义。另一方面, 我们从图 2.6 中得到 Star 和 Studio 的定义, 用键码和范围说明来扩充它们, 但是没有用方法; 见图 8.4。

```
interface Star
    (extent Stars
     key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set < Movie> starredIn
        inverse Movie starts;
};

interface Studio
    (extent Studios
     key name)
{
    attribute string name;
    attribute string address;
    relationship Set < Movie> owns
        inverse Movie ownedBy;
};
```

图 8.4 面向对象的电影数据库的一部分

8.2.2 OQL 类型系统

OQL 中的类型与 ODL 说明中的一样多(见 2.1.7 节)。然而 OQL 对于类型构造符的嵌套深度没有限制。

当我们讨论编程语言的类型系统时, 需要区别说明变量的类型(有时称为可变对象)和表达常量的值(有时称为不可变对象)。OQL 语句所用的变量将在外层的宿主语言中说明, 可能使用在 2.1.7 节介绍的 ODL 表示法或者类似的某种方法。ODL 作为数据定义语言不需要常量, 但是 OQL 程序需要。这样, 我们就需要了解在 OQL 中如何构造任意类型的常量。常量是由下列的基本类型和类型构造符构成的。

- 1. 基本类型, 包括两部分:
 - (a) 原子类型: 整数、浮点数、字符、字符串和布尔类型。它们像 SQL 那样表示, 除了字符串要用双引号括起来。

用箭头代替点

OQL 把箭头- > 作为点的同义词。这种约定有点 C 的风格, C 的点和箭头都能获得结构的分量。然而, C 的箭头和点运算符的含义稍微有些不同;但是在 OQL 中它们是相同的。在 C 中, 表达式 `a.f` 要求 `a` 是一个结构, 而 `p->f` 要求 `p` 是结构的指针。两者都产生该结构的域 `f` 的值。

(b) 枚举类型: 枚举类型中的值实际上是在 ODL 中说明的。任何这样的值都可以作为常量。

2. 由下列类型构造符构成复杂类型。

- (a) `Set(...)`。
- (b) `Bag(...)`。
- (c) `List(...)`。
- (d) `Array(...)`。
- (e) `Struct(...)`。

前四种称为聚集类型。聚集类型和结构(struct)可以用来作为任何合适类型(基本的或者复杂的)的值。然而, 当应用 Struct 运算符时需要指定域名及其相应的值。每个域名都跟着一个冒号和它的值, 域-值对之间用逗号分开。

例 8.3 表达式 `bag(2, 1, 2)` 表示在包(bag)中整数 2 出现 2 次, 整数 1 出现 1 次。表达式

```
struct(foo bag(2, 1, 2), bar baz)
```

表示具有两个域的结构。一个名为 `foo`, 用上面描述的包作为它的值; 另一个名为 `bar`, 其值为字符串 `baz`。

8.2.3 路径表达式

我们通过采用点表示法的复杂类型来访问变量的分量, 点表示法类似于 C 中所用的点, 也与 SQL 中所用的点有关。一般规则如下。如果 `a` 表示属于类 `C` 的对象, `p` 是该类的某个特性——可以是该类的属性、联系或者方法——那么 `a.p` 就表示把 `p` 用于 `a` 的结果。也就是:

- 1. 如果 `p` 是属性, 那么 `a.p` 就是对象 `a` 的该属性值。
- 2. 如果 `p` 是联系, 那么 `a.p` 就是通过联系 `p` 与 `a` 相连的对象或者对象的聚集。
- 3. 如果 `p` 是方法(或许带参数), 那么 `a.p` 就是把 `p` 用于 `a` 的结果。

例 8.4 假定 `myMovie` 是宿主语言变量, 它的值是 `Movie` 对象。那么

- `myMovie.length` 的值是该电影的长度, 也就是, 由 `myMovie` 所表示的 `Movie` 对象中 `length` 属性的值。
- `myMovie.lengthInHours()` 的值是实数(用小时表示的电影长度), 是通过把方法 `lengthInHours` 用于对象 `myMovie` 计算出来的。
- `myMovie.stars` 的值是通过联系 `stars` 与电影 `myMovie` 相连的 `Star` 对象的集合。

- 表达式 `myMovie.starNames(myStars)` 不返回任何值(也就是,在 C++ 中该表达式的类型是 `void`)。然而,作为附带的效果,它设置方法 `starNames` 的输出变量 `myStars` 的值为字符串的集合;这些字符串是电影的影星名。

为了有意义,我们可以用几个点形成表达式。例如,如果 `myMovie` 表示电影对象,那么 `myMovie.ownedBy` 就表示拥有该电影的 Studio 对象,而 `myMovie.ownedBy.name` 则表示作为该制片公司名字的字符串。

8.2.4 OQL 中的 select-from-where 表达式

OQL 允许我们用与众所周知的 SQL 查询格式类似的 `select-from-where` 句法来写表达式。这里是查询电影《乱世佳人》(`Gone With the Wind`) 年份的例子。

```
SELECT m.year
FROM Movies m
WHERE m.tittle =  'Gone With the Wind'
```

注意,除了字符串常量前后的双引号以外,该查询更像 SQL 而不是 OQL。唯一不明显的差别就是 SQL 中希望将 `FROM` 子句写成

```
FROM Movies AS m
```

然而,OQL 中关键字 `AS` 是可选的,就像 SQL 一样。OQL 中省略它显得更有意义,因为短语“`Movies m`”的含义是:`m` 是依次引用范围 `Movies` 中每个对象的变量;后者的范围是 `Movie` 类的当前对象集合。

一般说来,OQL 的 `select-from-where` 表达式的组成如下:

1. 关键字 `SELECT` 后面跟着表达式的列表。
2. 关键字 `FROM` 后面跟着一个或多个变量说明的列表。变量通过给出以下三项来说明:
 - (a) 其值为聚集类型(例如集合或包)的表达式,
 - (b) 可选关键字 `AS`, 和
 - (c) 变量名。

典型的情况是,(a)的表达式是某个类的范围,例如上面例子中的范围 `Movies` 对应于类 `Movie`。范围类似于 SQL `FROM` 子句中关系。然而,可以将产生聚集的任何表达式(例如另一个 `select-from-where` 表达式)用在变量说明中。对于这种能力 SQL2 不能直接模拟,不过某些商业的 SQL 允许 `FROM` 子句包含子查询。

3. 关键字 `WHERE` 和布尔值表达式。该表达式类似于 `SELECT` 之后的表达式,只是用常量和 `FROM` 子句中说明的那些变量作为运算数。比较运算符类似于 SQL,除了用“`!=`”而不是“`<>`”来表示“不等于”。逻辑运算符类似于 SQL,也是 `AND`, `OR` 和 `NOT`。

查询将产生对象的包。在嵌套的循环中,我们通过考查 `FROM` 子句中变量的所有可能的值来计算这个包。如果这些变量值的任何组合都满足 `WHERE` 子句的条件,那么就把 `SELECT` 子句所描述的对象加到包中,该包就是 `select-from-where` 语句的结果。

例 8.5 下面是说明 `select-from-where` 结构的更加复杂的 OQL 查询。

```
SELECT s.name
```

```
FROM Movies m, m.stars s
WHERE m.title = Casablanca
```

该查询检索 Casablanca(《卡萨布兰卡》)中的影星名。注意 FROM 子句中各项的顺序。首先,我们通过说明 m 在 Movie 类的 Movies 范围中来定义 m 是 Movie 类的任意对象。然后,对于每个 m 值,我们令 s 为电影 m 的影星集 m.stars 中的一个 Star 对象。也就是,我们考虑两个嵌套循环的所有(m, s)对,其中 m 是一部电影, s 是该电影的影星。计算过程可以描述成:

```
FOR Movies 中的每个 m DO
  FOR m.stars 中的每个 s DO
    IF m.title = Casablanca THEN
      把 s.name 加到输出包中
```

WHERE 子句把我们的考虑范围限制到一些对中,这样的对使 m 等于其名为 Casablanca(《卡萨布兰卡》)的 Movie 对象。然后,SELECT 子句产生包(在这种情况下它应该是一个集合),包中为满足 WHERE 子句的(m, s)对的影星对象 s 的所有姓名属性。这些姓名是集合 m.c.stars 中的影星名,其中 m.c 是电影对象 Casablanca。

8.2.5 消除重复

在技术上,查询的结果更像例 8.5 那样产生包,而不是集合。也就是,OQL 遵循 SQL 的默认:如果不提出要求,就不会消除结果中重复的部分。和 SQL 一样,消除重复的方法是在 SELECT 之后加上关键字 DISTINCT(互异)。

例 8.6 让我们查询 Disney(迪斯尼)电影中的影星名。下面查询执行这样的操作:当一个影星在几部 Disney 电影中出现时删除重名。

```
SELECT DISTINCT s.name
FROM Movies m, m.starts s
WHERE m.ownedBy.name = Disney
```

该查询的策略类似于例 8.5。像例 8.5 那样,我们再一次考虑在两个嵌套循环中由一部电影和该电影的一个影星组成的所有的对。但是现在(m, s)对的条件是:“ Disney ”是其 studio 对象为 m.ownedBy 的制片公司的名字。

8.2.6 复杂的输出类型

SELECT 子句中的表达式不必都是简单的变量,可以是任何表达式(包括用类型构造符构成的表达式)。例如,我们可以用 Struct 类型构造符构造几个表达式,并得到产生结构集或者结构包的 select-from-where 查询。

例 8.7 假设我们要得到地址相同的影星对的集合。我们可以用如下查询来得到该集合。

```
SELECT DISTINCT Struct(star1 s1, star2 s2)
FROM Stars s1, Stars s2
WHERE s1.addr = s2.addr AND s1.name < s2.name
```


也就是,我们考虑所有的影星对, s1 和 s2。WHERE 子句检验他们是否具有同一地址。它同时检验第一个影星名按字母顺序是否在第二个影星的前面,所以不会产生由同一影星重复两次组成的对,并且也不会产生以两种不同顺序组成的同一影星对。

经过这两种检验的每个对都产生一个记录结构。该结构类型是具有两个域的记录,其域名为 star1 和 star2。由于为这两个域提供值的变量 s1 和 s2 的类型都是 Star 类,因此每个域的类型也都是 Star 类。正式地说,也就是对于某个名字 N 该结构的类型是

```
Struct N {star1 Star, star2 Star}
```

查询结果的类型是类型 N 这种结构的集合,也就是:

```
Set Struct N {star1 Star, star2 Star}
```

注意,该查询结果的类型是一种类型的实例,它可以在 OQL 程序中出现,但不能作为属性或联系的类型在 ODL 说明中出现。

顺便提一下,如果我们在关键字 SELECT 之后只列出分量和域名,而不显式地定义结构类型,也可以得到和例 8.7 一样的效果。也就是,我们可以把例 8.7 查询中的第一行改写成

```
SELECT DISTINCT star1 s1, star2 s2
```

8.2.7 子查询

我们可以在适于聚集(例如集合)的任何地方使用 select-from-where 表达式。一个意想不到的地方是,子查询可以出现在 FROM 子句中,在该子句中,作为变量范围的聚集可以通过 select-from-where 表达式生成。实际上,按照 SQL2 标准, FROM 子句中允许有带括号的子查询。顺便提一下,同种类型的能力——用表达式来定义表而不是表名——是 SQL3 推荐标准的一部分,并且在某些商业 SQL 系统中得到了应用。

例 8.8 让我们重做例 8.6 的查询,查询 Disney(迪斯尼)公司制作的电影中的影星。首先,迪斯尼电影的集合可以通过下面的查询获得:

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = Disney
```

我们现在可以用该查询作为子查询来定义代表迪斯尼电影的变量 d 所能覆盖的集合。

```
SELECT DISTINCT s.name
FROM (SELECT m
      FROM Movies m
      WHERE m.ownedBy.name = Disney) d,
d.stars s
```

查询“找出迪斯尼电影的影星”的这种表达式不比例 8.6 更简洁,可能更差。然而,它确实说明了 OQL 中建立查询的新形式。在上面的查询中, FROM 子句具有两个嵌套的循环。在第一个循环中,变量 d 覆盖了所有的迪斯尼电影,这是 FROM 子句中子查询的结果。对于嵌套在第一个循环外的第二个循环,变量 s 覆盖电影 d 的所有影星。注意,不需要 WHERE 子句。

8.2.8 对结果排序

OQL 中的 select-from-where 表达式的结果或者是包, 或者(如果使用了 DISTINCT)是集合。如果我们在 select-from-where 后面使用 ORDER BY 子句, 就可以使输出成为列表, 并能同时选择该列表的元素顺序。OQL 中的 ORDER BY 子句非常类似于 SQL 的同样的子句。关键字 ORDER BY 的后面跟着表达式的列表。根据查询结果中的每个对象计算其中第一个表达式的值, 然后把对象按该值进行排序。如果为等序, 那么它们就按第二个表达式的值进行排序, 接下来按第三个, 依此类推。

例 8.9 让我们找出迪斯尼电影的集合, 但让结果是按照长度排序的电影列表。如果为等序, 则让同等长度的电影按字母顺序排列。该查询是:

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = Disney
ORDER BY m.length, m.title
```

前三行与例 8.8 中的子查询相同。第四行规定 select-from-where 查询所产生的对象 m, 首先按照 m.length 的值(也就是电影的长度)排序, 然后, 如果为等序, 就按 m.title 的值(也就是电影名)排序。于是该查询所产生的值是 Movie 对象的列表。默认的顺序为升序, 但是选择升序还是降序可以在 ORDER BY 子句的末尾分别用关键字 ASC 或者 DESC 来指定, 这和 SQL 是一样的。

8.2.9 本节练习

练习 8.2.1: 利用练习 8.1.1 和图 8.2 中的 ODL 模式, 用 OQL 写出下列查询:

- * (a) 找出价格在 \$ 2 000 以下的所有 PC 产品的型号。
- (b) 找出 RAM 至少 32M 字节的所有 PC 产品的型号。
- * ! (c) 找出至少制造两种不同型号激光打印机的制造商。
- (d) 某个 PC 或者便携式电脑有 r M 字节 RAM 和 h G 字节硬盘, 找出这样的(r, h)对的集合。
- (e) 按照处理器速度的升序建立 PC(对象, 不是型号)的列表。
- ! (f) 按照屏幕尺寸的降序, 建立至少有 16M 字节 RAM 的便携式电脑的型号列表。

! 练习 8.2.2: 重做练习 8.2.1, 在每个查询中至少有一个子查询。

练习 8.2.3: 利用练习 8.1.2 和图 8.3 中的 ODL 模式, 用 OQL 写出下列查询:

- (a) 找出至少有 9 门火炮的舰艇的等级名。
- (b) 找出至少有 9 门火炮的舰艇(对象, 不是舰艇名)。
- (c) 找出排水量在 30 000 吨以下的舰艇名。结果首先按照最早的下水年份排成列表, 如果为等序, 再按照舰艇名的字母顺序排列。
- (d) 找出是姊妹舰艇(也就是, 同等级的舰艇)的对象对。注意, 要的是舰艇本身, 而不是舰艇的名字。

- !(e) 找出至少有两个不同国家的舰艇沉没的战役的名字。
- !!(f) 找出所列出的舰艇没有损坏的战役的名字。

8.3 OQL 表达式的附加格式

在这一节将会看到,OQL 提供给我们用来建造表达式的,除了 select-from-where 之外,还有其他一些运算符。这些运算符包括逻辑量词(全称(for-all)和存在(there-exists))、聚合运算符、分组(group-by)运算符和集合运算符(并、交和差)。

8.3.1 量词表达式

我们可以检测是否所有的集合成员或至少有一个集合成员满足某个条件。为了检测是否集合 S 的所有成员都满足条件 C(x)(其中 x 是变量),我们使用 OQL 表达式

FOR ALL x IN S C(x)

如果 S 中的每个 x 都满足 C(x),则该表达式的结果就为真(TRUE),否则为假(FALSE)。

类似地,如果 S 中至少有一个 x 使 C(x) 为真(TRUE),则表达式

EXISTS x IN S C(x)

为真(TRUE),否则为假(FALSE)。

例 8.10 表达“找出迪斯尼电影的所有影星”这一查询的另一种方法如图 8.5 所示。在这里,我们把注意力集中在影星 s 上,查询 s 是否是某个迪斯尼电影 m 中的影星。3)行考虑电影集合 s.starredIn(它是影星 s 出演的电影的集合)中的所有电影 m。然后,4)行查询电影 m 是否是迪斯尼电影。我们即使找到一部这样的电影,则 3)和 4)行中的 EXISTS 表达式的值就为 TRUE,否则为 FALSE。

例 8.11 让我们用全称(for-all)运算符写一个查询来查找只在迪斯尼电影中出现的影星。在技术上,该集合包括电影中根本没出现(仅就我们的数据库而言)的影星。可以把另一种情况加到我们的查询中,即要求影星至少在一部电影中出现,但是我们把该改进留作练习。图 8.6 显示了该查询。

```

1)  SELECT s
2)  FROM Stars s
3)  WHERE EXISTS m IN s.starredIn:
4)      m.ownedBy.name =  Disney
```

图 8.5 使用存在量词的子查询

```

1)  SELECT s
2)  FROM Stars s
3)  WHERE FOR ALL m IN s.starredIn :
4)      m.ownedBy.name =  Disney
```

图 8.6 使用全称量词的子查询

8.3.2 聚合表达式

OQL 使用与 SQL 相同的五种聚合运算符: AVG,COUNT,SUM,MIN 和 MAX。但是可以认为 SQL 中的这些运算符是应用于表的指定列,而 OQL 中的同样运算符则应用于其成员为合适类型的聚集。也就是,COUNT 可以应用于任何聚集;SUM 和 AVG 可以用于算术(例如整数)类型的聚集;MIN 和 MAX 可以用于任何可比较的类型(例如算术

值或者字符串)的聚集。

例 8.12 为了计算所有电影的平均长度, 我们需要为所有电影的长度建立一个包。注意, 我们不应要电影长度的集合, 否则具有相同长度的两部电影将作为一部统计。该查询为:

```
AVG(SELECT m. length FROM Movies m)
```

也就是, 我们利用子查询从电影中提取长度分量。它的结果是电影长度的包, 然后我们将 AVG 运算符应用于该包从而得到需要的结果。

8.3.3 分组表达式

SQL 中的 GROUP BY 子句也用在 OQL 中, 但是具有有趣的新意。OQL 中 GROUP BY 子句的格式是:

- 1. 关键字 GROUP BY。
- 2. 用逗号分开的一个或者多个分区属性的列表。每个分区属性组成如下:
 - (a) 域名 f ,
 - (b) 冒号,
 - (c) 表达式 e 。

也就是, GROUP BY 子句的格式为:

```
GROUP BY  $f_1 \quad e_1, f_2 \quad e_2, \dots, f_n \quad e_n$ 
```

GROUP BY 子句跟在 select-from-where 查询后面。表达式

```
 $e_1, e_2, \dots, e_n$ 
```

可以引用 FROM 子句中提到的变量。为了便于理解 GROUP BY 是如何工作的, 我们将 FROM 子句限制在只有一个变量 x 的一般情况。 x 的值覆盖某个聚集 C 。对于满足 WHERE 子句条件的 C 的每个成员(记为 i), 我们求跟在 GROUP BY 之后的所有表达式, 以获得 $e_1(i), e_2(i), \dots, e_n(i)$ 的值。这些值的列表就是值 i 所属的组。

GROUP BY 实际的返回值是结构的集合。该集合的成员具有如下形式:

```
Struct( $f_1 \quad v_1, f_2 \quad v_2, \dots, f_n \quad v_n, \text{partition} \quad P$ )
```

前 n 个域代表组。也就是, v_1, v_2, \dots, v_n 是值的列表, 该列表是根据聚集 C 中至少有一个满足 WHERE 子句条件的 i 值计算 $e_1(i), e_2(i), \dots, e_n(i)$ 得来的。

最后一个域有一个特殊的名字分区(partition)。直观看来, 分区的值 P 就是属于该组的 i 值。更确切地说, P 是如下格式的结构组成的包,

```
Struct( $x \quad i$ )
```

其中 x 是 FROM 子句中的变量。

具有 GROUP BY 子句的 select-from-where 表达式的 SELECT 子句可能只引用 GROUP BY 结果中的域, 称为 f_1, f_2, \dots, f_n 和 partition。通过 partition, 我们可以引用域 x, x 出现在作为结构的包 P (形成 partition 的值)的成员中。这样, 我们就可以引用出现在 FROM 子句中的变量 x , 但是也许只能在聚合运算符(聚合了包 P 的所有成员)中这么做。

例 8.13 让我们为每个制片公司和每年制作的电影建立电影总长度表。在 OQL 中, 我们实际建立的是结构的包, 每个包有三个分量——制片公司、年份和该制片公司这一年

制作的电影总长度。该查询如图 8.7 所示。

```
SELECT std, yr, sumLength  SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY std  m.ownedBy.name, yr  m.year
```

图 8.7 按制片公司和年份将电影分组

为了理解该查询,我们先从 FROM 子句开始。我们发现,变量 m 覆盖所有的 Movie 对象。在这里 m 起到在通常讨论中 x 的作用。GROUP BY 子句中有两个域 std 和 yr,它们分别对应于表达式 m.ownedBy.name 和 m.year。

例如,Pretty Woman 是迪斯尼在 1990 年制作的一部电影。当 m 是这部电影的对象时,m.ownedBy.name 的值是 Disney ,同时 m.year 的值是 1990。结果 GROUP BY 子句构造的集合的每个成员都有如下结构

```
Struct(std  Disney , yr  1990, partition  P)
```

其中 P 是结构的集合。它包括结构:

```
Struct(m  mpw)
```

其中 m_{pw} 是 Pretty Woman 的 Movie 对象。对于 1990 年迪斯尼的所有其他电影, P 中也都是域名为 m 的单分量结构。

现在,我们检查 SELECT 子句。作为 GROUP BY 子句的结果得到一个集合,而对于该集合中的每个结构,我们在查询结果的包中建立一个结构。第一个分量是 std。也就是域名为 std,它的值是从 GROUP BY 得到的结构的 std 域的值。与此类似,结果的第二个分量具有域名 yr 并且其值与 GROUP BYD 结果中的 yr 分量相同。

每个输出结构的第三个分量是:

```
SUM(SELECT p.m.length FROM partition p)
```

为了理解该 select-from-where 表达式,我们首先要认识到变量 p 覆盖 GROUP BY 结果结构的 partition 域的成员。回忆一下,p 的每个值都是 Struct(m o) 形式的结构,其中 o 是电影对象。因此,表达式 p.m 引用该对象 o。于是,p.m.length 引用该 Movie 对象的长度分量。

作为结果,该 select-from 表达式按特定组产生电影长度的包。例如,如果 std 具有值 “Disney ”并且 yr 具有值 1 990,那么 select-from 的结果就是 1990 年迪斯尼所制作的电影长度的包。当我们把 SUM 运算符用于该包时,就得到该组的电影长度的总和。因此,如果 1 234 是 1 990 年所有的迪斯尼电影的正确的总长度,那么输出包中的一个结构就会是:

```
Struct(std  Disney , yr  1 990, sumLength  1 234)
```

万一 FROM 子句不只有一个变量,对该查询的解释做一些改变是必要的,但是原则上和上面所描述的单变量情况是一样的。假定出现在 FROM 子句中的变量是 x₁, x₂, ..., x_k。那么:

- 1. 所有变量 x₁, x₂, ..., x_k 都可能用在 GROUP BY 子句的表达式 e₁, e₂, ..., e_n 中。
- 2. 作为 partition 域值的包,其中的结构具有名为 x₁, x₂, ..., x_k 的域。
- 3. 假定 i₁, i₂, ..., i_k 分别是变量 x₁, x₂, ..., x_k 的值,并使得 WHERE 子句为真,那么

作为 GROUP BY 的结果得到的集合中就有一个结构, 其格式如下:

```
Struct(f1 = e1(i1, ..., ik), ..., fn = en(i1, ..., ik), partition = P)
```

而在包 P 中的结构是

```
Struct(x1 = i1, x2 = i2, ..., xk = ik)
```

8.3.4 HAVING 子句

OQL 的 GROUP BY 子句后面可以跟着 HAVING 子句, 它的含义类似于 SQL 的 HAVING 子句。也就是形如

```
HAVING 条件
```

的子句用来删除由 GROUP BY 建立的某些组。该条件用于 GROUP BY 结果的每个结构的 partition 域值。如果为真, 那么该结构就像 8.3.3 节中那样传给输出来处理。如果为假, 那么该结构就不用来作为该查询的结果。

例 8.14 让我们重复例 8.13, 但是只查询在某年至少制作了一部超过 120 分钟电影的制片公司在该年制作的电影长度之和。图 8.8 中的查询完成这项工作。注意在 HAVING 子句中, 我们使用和 SELECT 子句中一样的查询以便得到给定的制片公司和年份的电影长度的包。在 HAVING 子句中, 我们拿这些长度中的最大值和 120 相比较。

```
SELECT std, yr, sumLength = SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY std = m.ownedBY.name, yr = m.year
HAVING MAX(SELECT p.m.length FROM partition p) > 120
```

图 8.8 限制所考虑的组

8.3.5 集合运算符

我们可以把并、交和差运算符用于类型为集合或者包的两个对象上。像 SQL 那样, 这三个运算符分别用关键字 UNION, INTERSECT 和 EXCEPT 来表示。

```
1)      (SELECT DISTINCT m
2)      FROM Movies m, m.stars s
3)      WHERE s.name = Harrison Ford )
4) EXCEPT
5)      (SELECT DISTINCT m
6)      FROM Movies m
7)      WHERE m.ownedBy.name = Disney )
```

图 8.9 使用两个集合的差集的查询

例 8.15 我们可以通过图 8.9 中的两个 select-from-where 查询之间的差集找出不是迪斯尼公司制作的但是哈里森·福特(Harrison Ford)主演的电影集合。1)到 3)行找出哈里森·福特(Harrison Ford)主演的电影集合, 而 5)到 7)行则找出迪斯尼公司制作的电影集合。4)行中的 EXCEPT 执行两者的差。

我们应当注意到图 8.9 中的 1)和 5)行中的关键字 DISTINCT。该关键字使这两个查询的结果成为集合类型;没有 DISTINCT,该结果将成为包(多集)类型。在 OQL 中,运算符 UNION, INTERSECT 和 EXCEPT 可以对集合或包进行运算。当两个参数都是集合时,这些运算符就具有它们通常的集合含义。

然而,当两个参数的类型都是包,或者一个是包而另一个是集合时,运算符就用包的含义。回顾一下 4.6.2 节,包允许一个对象在其中出现任意次。假定 B_1 和 B_2 是两个包,而 x 是在 B_1 中出现 n_1 次并在 B_2 中出现 n_2 次的对象。 n_1 或者 n_2 ,或者二者都可以是 0。包的运算规则如下:

- 在 $B_1 \cup B_2$ 中, x 出现 $n_1 + n_2$ 次。
- 在 $B_1 \cap B_2$ 中, x 出现 $\min(n_1, n_2)$ 次。
- 在 $B_1 - B_2$ 中,
 1. 如果 $n_1 \leq n_2$, 那么 x 出现 0 次。
 2. 如果 $n_1 > n_2$, 那么 x 出现 $n_1 - n_2$ 次。

对于图 8.9 中的特定查询,一部电影在每个子查询的结果中出现的次数是 0 或者 1,所以不管是否使用 DISTINCT 结果都一样。但是结果的类型不同。如果使用了 DISTINCT,那么结果的类型就是 $\text{Set} < \text{Movie} >$,但如果在一处或者两处省略了 DISTINCT,那么结果的类型就是 $\text{Bag} < \text{Movie} >$ 。

8.3.6 本节练习

练习 8.3.1: 利用练习 8.1.1 和图 8.2 中的 ODL 模式,用 OQL 写出下列查询:

- * (a) 找出同时制造 PC 和打印机的制造商。
- * (b) 找出只制造 PC 且硬盘容量至少为 2G 字节的 PC 制造商。
- (c) 找出制造 PC 但不制造便携式电脑的制造商。
- * (d) 找出 PC 的平均速度。
- * (e) 对于每个 CD 速度,找出 PC 中 RAM 的平均容量。
- ! (f) 找出这些制造商,他们制造的某个产品具有至少 16M 字节的 RAM,并且还制造一种价格在 \$ 1 000 之下的产品。
- !! (g) 对于那些所制造的 PC 的平均速度至少为 150 的制造商,找出他们提供的 PC 中 RAM 的最大容量。

练习 8.3.2: 利用练习 8.1.2 和图 8.3 中的 ODL 模式,用 OQL 写出下列查询:

- (a) 某些等级的所有舰艇都在 1919 年以前下水,找出这些舰艇的等级。
- (b) 找出任何等级的最大排水量。
- ! (c) 对于每种火炮的口径,找出装备了该火炮的舰艇下水的最早年份。
- * !! (d) 对于至少有一艘舰艇在 1919 年以前下水的每个舰艇等级,找出战役中沉没的该等级舰艇数目。
- ! (e) 找出一个等级舰艇的平均数目。
- ! (f) 找出舰艇的平均排水量。
- !! (g) 找出至少有一艘来自大不列颠(Great Britain)的舰艇参战,并且至少有两艘

舰艇沉没的战役(对象,不是名字)。

! 练习 8.3.3: 我们在例 8.11 中提到,图 8.6 中的 OQL 查询将返回这样的影星:完全没有主演过其他电影,因此,“只是专门出现在迪斯尼电影中”。重写该查询以返回至少主演一部电影的影星和他们出演的所有迪斯尼电影。

8.4 OQL 中对象的赋值和建立

在这一节我们将考虑如何把 OQL 和它的宿主语言相连,在我们的实例中将用 C++ 作为宿主语言,不过某些系统中可能用另一种面向对象的通用编程语言作为宿主语言。

8.4.1 对宿主语言变量赋值

SQL 需要在元组分量和宿主语言变量之间传递数据,而 OQL 与之不同,它很自然地适合于它的宿主语言。也就是,我们学过的 OQL 的表达式(比如 select -from -where)在产生对象的同时产生值。可以把这些 OQL 表达式的结果值赋给任何合适类型的宿主语言变量。

例 8.16 OQL 表达式

```
SELECT DISTINCT m
FROM Movies m
WHERE m.year < 1920
```

产生 1920 年以前制作的所有电影的集合。它的类型是 $\text{Set} < \text{Movie} >$ 。如果 oldMovies 是同类型的宿主语言变量,那么我们就可以写(用扩充了 OQL 的 C++):

```
oldMovies = SELECT DISTINCT m
              FROM Movies m
              WHERE m.year < 1920
```

并且 old Movies 的值将成为这些 Movie 对象的集合。

8.4.2 从聚集中提取元素

既然 select -from -where 和 group-by 表达式都产生聚集——集合或者包,如果我们想得到该聚集中的单个元素,就必须做一些额外的工作。即使可以确信某个聚集只包含一个元素,这句话也是正确的。OQL 提供了运算符 ELEMENT 用于把单独的集合或者包转换成它的单个成员。例如,该运算符可以用于已知返回单个值的查询结果。

例 8.17 假定我们想把代表电影《乱世佳人》(Gone With the Wind)的对象赋给 Movie 类型(也就是,Movie 类是它的类型)的变量 gwtw。查询

```
SELECT m
FROM Movies m
WHERE m.title = Gone With the Wind
```

的结果是仅仅包含这样一个对象的包。如果把该包直接赋给变量 gwtw,将得到一个类型错误,因此不能这样做。然而,如果我们首先应用 ELEMENT 运算符:


```

gwtw = ELEMENT(SELECT m
                FROM Movies m
                WHERE m.title =  Gone With the Wind
                );

```

那么变量和表达式的类型就匹配了,同时赋值也是合法的。

8.4.3 获取聚集的每个成员

获取集合或者包的每个成员是比较复杂的,但是仍然比 SQL 中需要基于游标的算法简单。首先,我们需要把集合或者包转换为列表。我们用带 ORDER BY 的 select-from-where 表达式来做这件事。回忆一下 8.2.8 节,这种表达式的结果是所选择的对象或值的列表。

例 8.18 假定我们想要类 Movie 的所有电影对象的列表。由于(title, year)是 Movie 的键码,因此我们可以利用电影的名称(title)和(为把等序分开)年份(year)。语句

```

movieList = SELECT m
            FROM Movies m
            ORDER BY m.title, m.year;

```

将把按名称和年份排序的所有 Movie 对象的列表赋给宿主语言变量 movieList。

我们一旦得到了一个列表,不管是排序的还是没有排序的,就可以用序号访问每个元素;列表 L 的第 i 个元素可以用 L[i- 1] 得到。注意,假定列表和数组的序号从 0 开始,这与 C 或 C++ 一样。

例 8.19 假定我们想写一个 C++ 函数来打印每部电影的名称、年份和长度。该函数的描述如图 8.10 所示。

```

1)  movieList = SELECT m
                FROM Movies m
                ORDER BY m.title, m.year;
2)  numberOfMovie = COUNT(Movies);
3)  for(i= 0; i< numberOfMovies; i++ ) {
4)      movie = movieList[i];
5)      cout << movie.title<< " " << movie.year<< " "
6)          << movie.length<< " \n ";
7)  }

```

图 8.10 检查并打印每部电影

1) 行对 Movie 类进行排序,把结果放到变量 movieList,它的类型是 List< Movie> 。
 2) 行用 OQL 运算符 COUNT 计算电影的数目。其次,3) 到 7) 行是 for 循环,在该循环中整数变量 i 覆盖了该列表的每个位置。为了方便,把列表的第 i 个元素赋给变量 movie。然后,5) 和 6) 行将打印电影的相关属性。

8.4.4 建立新对象

我们已经看到像 select-from-where 这样的 OQL 表达式允许我们建立新的对象。这些对象通过对已有对象的计算来建立。通过把常量和表达式显式地组合到结构和聚

集中来建立对象也是可能的。我们用一种显而易见的方式把类型构造符用于值来做到这一点。当我们建立值时,不用描述类型的尖括号,而是用圆括号。我们在例 8.7 就看到了这种约定的例子。在例 8.7 中,语句

```
SELECT DISTINCT Struct(star1 s1, star2 s2)
```

用来规定查询的结果是一个对象的集合,对象的类型是 Struct{star1 Star, star2 Star}。我们给出域名 star1 和 star2 来规定该结构,而这些域的类型则可以从变量 s1 和 s2 的类型推断出。

例 8.20 通过把任何一种聚集类型构造符 Set, Bag, List 或 Array 用于同种类型的对象,我们也可以建立聚集。例如,考虑下列赋值序列:

```
x = Struct(a 1, b 2);
y = Bag(x, x, Struct(a 3, b 4));
```

第一行赋给变量 x

```
Struct(a integer, b integer)
```

类型的值,该类型具有两个名为 a 和 b 的整型域。我们可以把这种类型的值看成元组,这些元组只是用这两个整数而不是域名 a 和 b 作为分量。于是, x 的值可以用(1, 2)来表示。第二行把 y 定义成一个包,它的成员是和上面的 x 类型相同的结构。(1, 2)组成的对在该包中出现了两次,而(3, 4)出现了一次。

如果我们有某种类型而某个查询将产生该类型的对象聚集,那么就可以用该类型名来代替显式的类型表达式。例如,我们在例 8.7 清楚地看到如何建立影星对的集合。我们在关键字 SELECT 之后跟着一个表达式,该表达式使用类型构造符 Struct 来建立包括两个域的对象,而这两个域的值均为影星对象。

现在假定我们已经定义了类型 StarPairs 为:

```
Struct{star1 Star, star2 Star}
```

然后可以重写例 8.7 中的查询以便在下面的 SELECT 子句中使用该类型。

```
SELECT DISTINCT StarPairs(star1 s1, star2 s2)
FROM Stars s1, Stars s2
WHERE s1.addr = s2.addr AND s1.name < s2.name
```

与例 8.7 的唯一的区别是该查询的结果具有类型 Set StarPair 。因此可以把该结果赋给已说明为该类型的宿主语言变量。

当类型名是类时将类型名作为参数特别有用。类一般有几种不同形式的构造函数,这取决于显式的初始化的特性和给定的某种默认值。例如,方法肯定不初始化,大多数属性将得到初始值,而联系可能初始化为空集以后再添充。每个构造函数名都是类名,它们通过参数中提到的域名来区别。如何定义这些构造函数其细节取决于宿主语言。

例 8.21 让我们考虑 Movie 对象的一个可能的构造函数。我们假定该函数利用属性 title, year, length 和 ownedBy 的值,产生在所列出的域具有这些值的对象以及影星的空集。那么,如果 mgm 是值为米高梅(MGM)Studio 对象的变量,我们就可以用如下语句来建立一个《乱世佳人》(Gone With the Wind)对象:

```
gwtw = Movie(title: Gone With the Wind ,
```

```
year: 1939,  
length: 239,  
ownedBy: mgm);
```

该语句有两个作用:

1. 建立一个新的 Movie 对象, 该对象将成为 Movies 范围的一部分。
2. 使该对象成为宿主语言变量 gwtw 的值。

8.4.5 本节练习

练习 8.4.1: 将下列常数赋给宿主语言变量 x:

- * (a) 集合{1, 2, 3}。
- (b) 包{1, 2, 3, 1}。
- (c) 列表(1, 2, 3, 1)。
- (d) 结构的第一个分量, 名为 a, 是集合{1, 2}, 而第二个分量名为 b, 是包{1, 1}。
- (e) 结构的包, 每个结构都有名为 a 和 b 的两个域。包中三个结构各自成对, 其值为(1, 2), (2, 1) 和(1, 2)。

练习 8.4.2: 利用练习 8.1.1 和图 8.2 中的 ODL 模式, 写出用 OQL 扩充了的 C++ (或者你选择的一种面向对象的宿主语言) 语句来完成下列操作:

- * (a) 把型号为 1 000 的 PC 对象赋给宿主语言变量 x。
- (b) 把至少有 16M 字节 RAM 的所有便携式电脑对象的集合赋给宿主语言变量 y。
- (c) 把售价低于 \$ 1 500 的 PC 的平均速度赋给宿主语言变量 z。
- ! (d) 找出所有的激光打印机, 打印它们的型号和价格的列表, 并随后给出信息指明价格最低的型号。
- !! (e) 打印一个表, 对于每个 PC 制造商, 给出 PC 的最低和最高价格。

练习 8.4.3: 在该练习中, 我们将利用练习 8.1.2 和图 8.3 的 ODL 模式。假定对于该模式的 4 个类中的每一个都有一个同名的构造函数, 它为每个属性和单值联系取值, 但是不为多值联系取值(它们将初始化为空)。对于其他类的单值联系, 你可以设定一个其当前值是相关对象的宿主语言变量。建立下列对象, 并且在每种情况下, 把该对象赋给宿主语言变量作为它的值。

- * (a) Maryland 等级的战列舰 Colorado, 1923 年下水。
- (b) L ü zow 等级的战列舰 Graf spee, 1936 年下水。
- (c) Malaya 战役的结局是战列舰 Prince of Wales 沉没。
- (d) Malaya 战役是 1941 年 10 月 10 日开战的。
- (e) Hood 等级的大不列颠巡洋舰具有 8 门 15 英寸的火炮和 41 000 吨的排水量。

8.5 SQL3 中的元组对象

OQL 中没有明确的关系概念; 它只是结构的集合(或者包)。然而, 在 SQL 中, 关系的概念是如此重要以致于 SQL3 中的对象仍把关系作为核心概念。SQL3 中的对象来自两

种风格:

- 1. 行对象(Row Object), 它们基本上是元组;
- 2. 抽象数据类型(Abstract Data Type 通常缩写为 ADT, 或在某些 SQL3 文档中缩写为值 ADT), 它们是只能用来作为元组分量的一般对象。

我们将在本节介绍行对象, 在 8.6 节介绍 ADT。

8.5.1 行类型

在 SQL3 中, 人们可以定义元组类型, 该类型大致相似于对象的类。行类型说明组成如下:

- 1. 关键字 CREATE ROW TYPE,
- 2. 类型的名字, 和
- 3. 用括号括起的属性及其类型的列表。

也就是, 行类型 T 定义的格式是:

```
CREATE ROW TYPE T( 分量说明 )
```

例 8.22 我们可以建立代表电影影星的行类型, 类似于图 8.4 的 OQL 例子中给出的 Star 类。然而, 我们不能直接把电影集合表示成 Star 元组的域。于是, 我们只能从 Star 元组的分量 name 和 address 开始。

首先, 注意图 8.4 中的地址类型本身是具有分量 street 和 city 的元组。这样我们需要两个类型定义, 一个为地址, 而另一个为影星。SQL3 中允许用行类型作为另一个行类型或关系的分量类型。图 8.11 给出了必要的定义。

类型 AddressType 的元组具有两个分量, 它们的属性是 street 和 city。这些分量的类型是长度分别为 50 和 20 的字符串。类型 StarType 的元组同样具有两个分量。第一个是属性 name, 它的类型是 30 个字符的字符串, 第二个是 address, 它的类型是 AddressType, 也就是, 它是具有分量 street 和 city 的元组。

```
CREATE ROW TYPE AddressType(  
    street CHAR( 50),  
    city CHAR(20)  
);  
  
CREATE ROW TYPE StarType(  
    name CHAR( 30),  
    address AddressType  
);
```

图 8.11 两个行类型定义

8.5.2 说明具有行类型的关系

说明了行类型之后, 我们可以说明其元组属于该类型的一个或者多个关系。关系说明的格式和 5.7.2 节的一样, 但是我们用

OF TYPE 行类型名

来代替标准 SQL 的表说明中属性及其类型的列表。

例 8.23 我们可以用

```
CREATE TABLE MovieStar OF TYPE StarType;
```

说明 MovieStar 是其元组类型为 StarType 的关系。结果, 表 MovieStar 具有两个属性, name 和 address。注意后者的类型本身是行类型, 这是 SQL3 以前的 SQL 标准通常所不允许的。

尽管每个行类型一般都对应一个关系, 并且把该关系看成对应于该元组类型的类的范围(按 8.1.3 节的含义), 但是对于给定的行类型, 允许有多个关系或者没有关系。

8.5.3 访问行类型的分量

由于 SQL3 中分量本身具有结构, 因此我们需要一种方法来访问分量中的分量。SQL3 使用双点表示法, 它严格地对应于 OQL 或者 C 中的单点表示法。

例 8.24 图 8.12 的查询找出住在 Beverly Hills 的每个影星的姓名和街道地址。我们选择了全属性名 MovieStar.name 和 MovieStar.address 来说明单点和双点之间的差别。然而, 既然 name 和 address 在这里是非二义性的属性, MovieStar 和单点就不需要了。

```
SELECT MovieStar.name, MovieStar.address..street
FROM MovieStar
WHERE MovieStar.address..city = Beverly Hills
```

图 8.12 范围分量的分量

8.5.4 引用

在 SQL3 中, 面向对象语言的对象标识的作用通过引用(reference)的概念来获得。行类型的分量可以把对另一个行类型的引用作为它的类型。如果 T 是行类型, 那么 REF(T) 就是对类型为 T 的元组的引用类型。如果我们把元组作为对象, 那么对该对象的引用是它的对象标识(ID)。

例 8.25 在 MovieStar 中我们还不能记录影星主演的所有电影的集合, 但是可以记录他们最好的电影。首先我们需要说明 Movie 关系, 如果愿意的话, 还可以同时为该关系说明行类型。不包括与影星、制片公司、制片人的重要联系的简单电影类型如下:

```
CREATE ROW TYPE MovieType(
    title    CHAR(30),
    year     INTEGER,
    inColor  BIT(1)
);
```

然后我们可以用

```
CREATE TABLE Movie OF TYPE MovieType;
```

来说明其为上述元组类型的关系。

接下来,我们必须更改 MovieStar 元组的类型以包括对该影星的最好电影的引用。StarType 的新定义如下:

```
CREATE ROW TYPE StarType(  
    name          CHAR( 30),  
    address       AddressType,  
    bestMovie     REF(MovieType)  
);
```

例 8.26 然而,假定我们要求在电影和影星之间有标准的多对多的联系:一个影星处于电影集合中,而一部电影有影星集合。虽然 ODL 允许影星集合作为电影的分量而且反之亦然,但是 SQL3 保持了贯穿全书所遵循的有关方法。 然而,多对多的联系可以用包含成对的相关项的独立关系来表示。

图 8.13 提示我们如何表示电影和影星之间的 star-in 联系。在 SQL3 以前的标准中,只能通过有关类成对的键码表示多对多的联系,SQL3 允许我们通过具有引用类型的属性直接引用对象(确切地说是元组)。我们先分别定义关系 Movie 和 MovieStar 的类型 MovieType 和 StarType。我们已经回到最初的没有最好电影的行类型 StarType。为关系 StarsIn 定义的行类型 StarsInType 含有成对的引用;每对引用一个影星和该影星出演的一部电影。

```
CREATE ROW TYPE MovieType(  
    title CHAR(30),  
    year  INTEGER,  
    inColor BIT(1)  
);  
  
CREATE ROW TYPE AddressType(  
    street  CHAR(50),  
    city    CHAR(20)  
);  
  
CREATE ROW TYPE StarType(  
    name      CHAR( 30),  
    address   AddressType  
);  
  
CREATE ROW TYPE StarsInType(  
    star      REF(StarType),  
    movie     REF(MovieType)  
);  
  
CREATE TABLE Movie OF TYPE MovieType;  
CREATE TABLE MovieStar OF TYPE StarType;  
CREATE TABLE StarsIn OF TYPE StarsInType;
```

图 8.13 影星、电影及其联系

SQL3 不包括 ALTER TYPE 或者类似语句,这类语句允许我们更改已有的类型定义。这样,如果我们希望更改以前定义的行类型,实际上将不得不撤消行类型和定义成具有该类型的任何表,然后重新定义类型和重新构造表。

尽管 SQL3 标准的某些草案确实允许聚集类型(例如,集合或者关系)作为属性的类型,但是很可能把这种聚集类型的使用推迟到以后的 SQL4 标准中。

域和行类型

在 5.7.6 节我们学习了域,它是一种类型说明。在域和行类型之间至少有两个重要的差别。首先,一个明显的差别是域定义分量的类型,而行类型是整个元组的类型。

但是还有一个比较细微的差别。域是简写形式(shorthands)。两个域可以表示同一类型,这些域的值将不予区分。然而,假设两种行类型 T_1 和 T_2 具有相同的定义。结果具有这两种类型的两个关系的元组并不能互换。例如,某属性的类型是对 T_1 的引用,而该属性并不能引用类型是 T_2 的元组。

行类型的定义后面跟着使用这些行类型的三个表 Movie、MovieStar 和 StarsIn 的说明。注意,行类型 AddressType 没有用来作为表的类型。而是把它作为行类型 StarType 的属性 address 的类型。

我们将比较在这里称为 StarsIn 的关系和 3.9 节数据库模式中的同名关系。后一个关系具有属性 movieTitle 和 movieYear 而不是对电影元组的引用,还具有属性 starName 以代替对影星元组的引用。

8.5.5 利用引用

一旦我们认为一个元组的分量可以是其他某个(或同一)关系的引用,那么通过提供引用运算符来扩展 SQL 就是很自然的。SQL3 中用 `->` 符号表示引用,它与 C 中的该运算符具有同样的含义。也就是,如果 x 是元组 t 的引用,并且 a 是 t 的属性,那么 $x->a$ 就是元组 t 中属性 a 的值。该运算符在许多 SQL3 查询中是很方便的,因为它可以代替 SQL2 中某些必要的连接。

例 8.27 让我们利用图 8.13 的模式,找出 Mel Gibson 主演的所有电影的名称。我们的策略是检查 StarsIn 中的每个对。如果所引用的影星是 Mel Gibson,那么我们就把该对的另一个分量所引用的电影名作为结果的一部分。该 SQL3 查询是:

```
SELECT movie-> title
FROM StarsIn
WHERE star-> name = 'Mel Gibson' ;
```

对该查询的解释如下。对于 SQL 的所有 select-from-where 查询,我们考虑 FROM 子句中提到的关系的每个元组,比如说 (s, m) 。其中 s 是对影星元组的引用, m 是对电影元组的引用。WHERE 子句在询问我们能否确定通过引用 s 所引用的 MovieStar 元组的 name 分量就是 Mel Gibson。如果是这样,那么我们就将获得通过 m 所引用的 Movie 元组的 title 分量的值,而且该值是该查询产生的元组之一。

8.5.6 引用的作用域

为了回答例 8.27 中的那种查询,SQL3 数据库系统必须把像 `star-> name` 这样的利用引用的表达式解释为对特定关系的 name 域的引用。一个简单的方法是查看 StarsIn 的每个元组,并利用它的 star 引用查看所引用的元组是否具有姓名“Mel Gibson”。然而,

间接引用和提取分量

SQL3 和 OQL 之间的差别之一可以在对 $->$ 和点运算符的解释上看出来。回忆一下 8.2.3 节, 在 OQL 中, 点和 $->$ 运算符含义相同。每个都应用于 OQL 元组对象, 而返回该对象的分量。在 SQL3 中和在 C 中一样, 这些运算符是有区别的。我们只能把 $->$ 用于对元组的引用, 并且只能够把点运算符 (SQL3 中写成两个点) 用于元组变量本身。和在 C 中一样, 如果 r 是对元组 t 的引用, 那么 $r -> a$ 产生和 $t.a$ 相同的值。

如果 StarsIn 很大, 那么用这种方法回答该问题将很费时间。

如果 DBMS 允许我们在某个关系 R 的 $name$ 属性上建立索引的话, 就可能有较好的方法把我们从某个特定的值 (例如, “Mel Gibson”) 引到某些 StarsIn 元组, 而这些元组引用 $name$ 等于 “Mel Gibson” 的关系 R 的元组。但是我们利用这样的索引在哪个 (或者哪些) 关系 R 中进行查找呢? 在这个例子中, 我们知道任何 StarsIn 元组的 $star$ 属性值都是对某个元组的引用, 而该元组必须在类型是 StarType 的关系中。由于只给出这样一个关系, MovieStar, 所以我们希望它是该引用所引用的关系。

然而, 有可能把其他关系也说明为类型 StarType, 如果这样, 就需要在每个关系的索引中查找姓名 “Mel Gibson”。如果由于模式设计者已知的原因, 所有的引用都针对一个特定关系的元组 (通常是这种情况), 那么这种查询可能会浪费时间。于是, SQL3 提供了一种机制来指定引用属性引用的是哪个关系。我们对其属性类型为引用的关系进行说明时可以增加一个子句, 表示为:

SCOPE FOR 属性 IS 关系

该语句的意思是所命名的属性 (必须是引用类型) 总是引用所命名的关系的元组。

例 8.28 为了保证在表 StarsIn 中, Star 的引用总是对 MovieStar 元组的引用, 同时 movie 的引用总是对 Movie 元组的引用, 我们可以写出如图 8.14 所示的关系 StarsIn 的说明。

```
CREATE ROW TYPE StarsInType(  
    star      REF (StarType),  
    movie     REF (MovieType)  
);  
  
CREATE TABLE StarsIn OF TYPE StarsInType  
SCOPE FOR star IS MovieStar,  
SCOPE FOR movie IS Movie;
```

图 8.14 说明引用属性的作用域

8.5.7 作为值的对象标识

面向对象语言通常遵循的原则是, 对象 ID 是内部系统值, 不能通过查询语言访问。例如 OQL 就做了这样的假定。然而, 我们不能显式地引用对象 ID 在原则上是没有理由

的,SQL3 给了我们这种能力。在关系或者它的行类型说明中,我们可以有一个属性,它的值是对同类型元组的引用。如果我们在行类型或者表的说明中增加如下子句:

VALUES FOR 属性 ARE SYSTEM GENERATED

那么命名属性的值将成为对该引用所在的同一元组的引用。于是,这样的属性既可以作为该关系的主键码,也可以作为它的元组的对象 ID。

例 8.29 让我们重写图 8.13 从而使 MovieStar 和 Movie 都具有对象 ID 属性,分别称为 star- id 和 movie- id。更改后的模式如图 8.15 所示。该图和图 8.13 之间的差别在于:

- 1. 把 movie- id 属性加到行类型 MovieType 中。
- 2. 把 star- id 属性加到行类型 StarType 中。
- 3. 把表 Movie 的 movie- id 值由系统产生这一语句加到该表的说明中。
- 4. 把表 MovieStar 的 star- id 值由系统产生这一语句加到该表的说明中。
- 5. 保留例 8.28 中的 SCOPE 说明。

```
CREATE ROW TYPE MovieType(  
    movie- id  REF(MovieType),  
    title      CHAR( 30),  
    year       INTEGER,  
    inColor    BIT(1)  
);  
  
CREATE ROW TYPE AddressType(  
    street  CHAR( 50),  
    city    CHAR( 20)  
);  
CREATE ROW TYPE StarType(  
    star- id  REF(StarType),  
    name     CHAR( 30),  
    address  AddressType  
);  
  
CREATE ROW TYPE StarsInType(  
    star      REF(StarType),  
    movie     REF(MovieType)  
);  
  
CREATE TABLE Movie OF TYPE MovieType  
VALUES FOR movie- id ARE SYSTEM GENERATED;  
  
CREATE TABLE MovieStar OF TYPE StarType  
VALUES FOR star- id ARE SYSTEM GENERATED;  
  
CREATE TABLE StarsIn OF TYPE StarsInType  
SCOPE FOR star IS MovieStar,  
SCOPE FOR movie IS Movie;
```

图 8.15 把对象 ID 加到关系中

现在我们有一种更方便的方法来写例 8.27 中讨论的查询,以找出 Mel Gibson 主演的

电影。在 WHERE 子句中我们可以使 StarsIn 中的两个引用等于关系 MovieStar 和 Movie 的两个对象 ID 属性,而这两个对象 ID 属性是对它们自己的元组的自引用。该查询如下:

```
SELECT Movie.title
FROM StarsIn, MovieStar, Movie
WHERE StarsIn.star = MovieStar.star_id AND
      StarsIn.movie = Movie.movie_id AND
      MovieStar.name = 'Mel Gibson';
```

也就是说, FROM 子句告诉我们,要考虑的三部分分别由关系 StarsIn、MovieStar 和 Movie 的元组组成。WHERE 子句的第一个条件是,来自 StarsIn 的元组一定引用(用它的 star 分量)来自 MovieStar 的元组。类似地, WHERE 子句的第二个条件表明, StarsIn 元组引用(用它的 movie 分量)来自表 Movie 的元组。这两个条件的作用是,要求来自 MovieStar 和 Movie 的元组代表 StarsIn 元组中成对的影星和电影。

然后, WHERE 子句的第三个条件要求所考虑的影星是 Mel Gibson,而 SELECT 子句将产生所考虑的电影名。注意,我们仍然可以像例 8.27 那样利用引用来写该查询。实际上,那种方法比该查询更简单,不过写该查询就像我们做过的那样,用来说明使用对象 ID 属性所特有的某些可能情况。

8.5.8 本节练习

练习 8.5.1: 为下列类型写行类型说明:

- (a) NameType, 具有分量教名(first name), 名(middle name), 姓(last name) 和学位(title)。
- * (b) PersonType, 具有某个人的姓名以及对其父母的引用。必须使用(a)中说明的行类型。
- (c) MarriageType, 具有结婚日期以及对丈夫和妻子的引用。

练习 8.5.2: 用行类型说明和适当的引用属性来重新设计练习 4.1.1 中不断滚动的产品数据库模式。特别是, 让关系 PC, Laptop 和 Printer 的 model 属性成为对该型号的 Product 元组的引用。

练习 8.5.3: 利用练习 8.5.2 中的模式写下列查询。在合适的时候尽量使用这些引用。

- (a) 找出其 PC 硬盘容量大于 2G 字节的 PC 制造商。
- (b) 找出激光打印机的制造商。
- ! (c) 产生一个表, 表中给出每个型号的便携式电脑, 而该型号便携式电脑的处理速度在同一制造商制造的所有便携式电脑中是最高的。

! 练习 8.5.4: 我们在练习 8.5.2 中建议: 表 PC, Laptop 和 Printer 中的型号可以是对表 Product 的元组的引用。Product 中的 model 属性也能成为对该类型产品的关系元组的引用吗? 为什么能, 或者为什么不能?

* 练习 8.5.5: 用行类型说明和适当的引用属性来重新设计练习 4.1.3 中不断滚动的战列舰数据库模式。练习 8.1.2 中的模式提示了这些引用属性可以用在哪里。查找多对一联系, 试着用具有引用类型的属性来表示它们。

练习 8.5.6: 利用练习 8.5.5 中的模式写出下列查询。在合适的时候尽量使用引用而避免使用连接(也就是, FROM 子句中避免使用子查询或者多个元组变量)。

- * (a) 找出排水量在 35 000 吨以上的舰艇。
- (b) 找出至少有一艘舰艇沉没的战役。
- ! (c) 找出有 1930 年以后下水的舰艇等级。
- !! (d) 找出至少有一艘美国舰艇受到损坏的战役。

8.6 SQL3 的抽象数据类型

SQL3 的行类型和对行类型的引用提供了 OQL 对象的许多功能。另外它们还允许我们使用 SQL 运算符(如插入和删除)来方便地修改“对象”。比较起来, OQL 倾向于在外层的面向对象编程语言(如 C++)中进行修改。

然而, 行类型不提供面向对象编程语言的有效封装。回忆一下 1.3.1 节, 我们“封装”了一个类来保证对象只能通过该类所定义的固定运算的集合来修改。封装的目的是防止在数据库的设计者不希望或未想到的一些情况下使用数据时通常会发生的编程错误。

行类型没有封装, 我们可以使用 SQL3 能表示的任何操作来操作行类型的元组。ODL 接口(类)没有完全封装, 所以我们可以用 OQL 查询的形式来访问对象的分量。另一方面, 查询对象的内部结构通常比以无计划的方式修改对象危险性小一些。在 OQL 中, 除了通过方法就不能更新对象(见 8.1.2 节)。可能用外层的传统语言(如 C++)编写的这些方法, 只能用于该类的对象。

SQL3 中还有一种的确支持封装的“类”的定义: 抽象数据类型 (Abstract Data Type), 即 ADT。ADT 对象可以用作元组的分量, 而不是作为元组本身。然而, 典型的对象本身都具有元组的结构, 就像 ODL 对象通常具有带分量的结构一样。

8.6.1 ADT 的定义

ADT 定义的格式如图 8.16 所示。1) 行是建立语句, 引入 ADT 名字。2) 行表示属性的名字及其类型的列表, 中间用逗号分开。

- 1) CREATE TYPE 类型名 (
- 2) 属性及其类型的列表
- 3) 该类型的 = 和 < 函数的可选说明
- 4) 该类型的函数(方法)的说明
- 5));

图 8.16 定义抽象数据类型

图 8.16 的 3) 行表示了比较运算符 = 和 < 的可选说明。相等函数的说明格式是
EQUALS 实现相等函数的名字
< 函数的定义类似, 只不过用关键字 LESS THAN 代替 EQUALS。注意, 其他四种比

随着 SQL3 标准的发展, 这些函数可能将不用专门处理, 但是还将不得不像这种类型的任何其他函数一样进行定义和使用

较运算符可以由此构成,不需要显式定义。例如, `=` 是“`=` 或者 `<`”,而 `<` 是“`不 <`”。如果定义了 `=` 和 `<`,那么在 WHERE 子句中就可以比较 ADT 的值,就像传统的 SQL 类型(例如整数和字符串)那样。

4) 行给出了该 ADT 的其他函数(也就是方法)的说明。SQL3 对每个 ADT 提供了一些不需要说明或者定义的“内置”函数。它们包括:

1. 构造函数(constructor function), 返回该类型的新对象。该对象的所有属性都初始化为 NULL, 如果 T 是 ADT 的名字, 那么 T()就是构造函数。

2. 观察函数(observer function), 对于每个属性都返回该属性的值。如果 A 是属性名, 并且 X 是一个变量(它的值是 ADT 对象), 那么 A(X)就是对象 X 中属性 A 的值。我们也可以使用更加传统的表示法 X.A 来表达同样的意思。

3. 变异函数(mutator function), 对于每个属性都设置该属性值为新值。它们通常用在赋值语句的左边, 其使用方式将在 8.6.2 节讨论。注意, 要实现封装, 就需要防止这些函数公用。SQL3 使用的方法是拥有函数的 EXECUTE 权限。这种权限可以像 7.4.1 节讨论的 SQL2 的 6 种权限那样授予和取消。

其他函数可以在 CREATE TYPE 语句内部或者外部定义。即使它们是外部定义的, 也可以只使用内部定义的函数, 包括上面所列的“内置”函数。

例 8.30 我们在例 8.22 定义了由 street 和 city 分量组成的地址为行类型。我们可以用另一种方式将地址说明为同样结构的 ADT。这种方法有封装地址的作用; 如果不把它们的观察和变异函数置成公用的, 就不能访问 street 和 city 分量。

```
1) CREATE TYPE AddressADT (  
2)     street      CHAR( 50),  
3)     city        CHAR( 20),  
4)     EQUALS      addrEq,  
5)     LESS THAN   addrLT  
           可在此说明的其他函数  
);
```

图 8.17 地址 ADT 的定义

图 8.17 给出了 ADT addressADT 的定义, 这里不包括与 ADT 相关的函数或者方法的实际定义。1) 行给出了 ADT 的名字。

2) 和 3) 行定义了表示方法, 一个元组具有名为 street 和 city 的两个分量。这些分量类型与例 8.22 一样: 字符串的长度分别为 50 和 20。

4) 和 5) 行告诉我们 AddressADT 的相等函数称为 addrEq, 而 `<` 比较函数则称为 addrLT。由于我们没有提供它们的定义, 因此还不知道这些函数做什么。我们将在例 8.32 给出我们选择的这些函数的定义。在那里, 我们将按字典顺序对地址进行排序, 首先按照城市名, 然后按照街道名。

下一个例子说明如何将 ADT 的功能引入 SQL 程序数据类型(这些数据类型不是数据库管理系统用于何处的传统假定可以预见的)。现在在数据库中存储非常大的对象(比如图像、音频剪辑或者电影)都是可行的。然而, 我们在这些对象上执行的操作不像“标

视频的 MPEG 编码

因为视频需要特别大的空间来存储,所以它通常用几种标准的压缩模式之一来进行编码。最常用的 MPEG(Motion Picture Experts Group, 运动图象专家组(全球影像/声音/系统压缩标准))模式利用了这样一个事实:动画的一帧与它前面的帧非常相似。这样,一帧的各个区域可以用前一帧类似区域的指针来表示。注意,前一帧的区域可能在同一位置(如果它是静止背景的一部分),也可能在不同的位置(如果它是运动物体的一部分)。

尽管 MPEG 压缩视频比标准的压缩文本模式好得多,但是用 MPEG 压缩一个小时的视频仍然需要 1G 字节。而且,由于允许相应区域有少许的不同,因此视频的质量通常稍微有些降低。为显示视频而进行的解压缩过程同样非常复杂。尽管有这些问题,MPEG 还是代表了图片质量、所用空间和所需要的计算能力之间良好的折衷。

准'的 SQL 操作(例如比较、打印、聚合等等)。相反,我们要显示这些对象通常需要使用复杂的解码算法来实现,将来甚至可能做复杂的图像比较或识别图像的重要特性。

例 8.31 假定我们想要一部电影的 MPEG 编码(MPEG 是视频压缩标准格式,详见主题框)的 ADT Mpeg。在技术上,MPEG 编码是字符串,所以可能认为 VARCHAR 类型比较合适。然而,通常 MPEG 编码视频的长度很大(G 字节),把视频作为字符串来处理是不现实的。

为了支持视频和其他非常大的数据项,现代数据库系统支持一种叫做 BLOB(Binary Large Object, 大容量二进制对象)的特殊数据类型,一种可能很长的特殊类型的位串,如果需要甚至可以达到 G 字节。在下例中我们将假定 BLOB 类型是数据库系统的内部类型。于是 ADT Mpeg 的合适定义如图 8.18 所示。

```
1) CREATE TYPE Mpeg (  
2)     video          BLOB,  
3)     length         INTEGER,  
4)     copyright      VARCHAR(255),  
5)     EQUALS         DEFAULT,  
6)     LESS THAN NONE  
                                这里放函数的定义  
);
```

图 8.18 MPEG ADT 的定义

2)行定义 BLOB 类型的 video 属性。该属性拥有非常大的 MPEG 编码视频。3)和 4)行是另外两个“普通”的属性:视频的长度(运行时间)和版权说明。5)行说明类型 Mpeg 值的相等比较是默认的:全等。也就是说,当且仅当两个 Mpeg 对象在相应属性上逐位相同时这两个对象才是相等的。我们可以设想为“相等”写一个更复杂的定义,以反映这样的想法:如果解码并且显示在合适分辨率的屏幕上的两个 Mpeg 对象看起来相同,就认为两个 Mpeg 值“相等”,但是在这里我们将不做这件事。6)行说明 Mpeg 值之间没有 < 的定

大容量二进制对象

对用户来说, BLOB 看起来像巨大的位串, 但是在现象背后, 它们的实现比长度限制为像 255 字节这样小的字符串来说要远为复杂。例如, 作为元组的分量来存储巨大的字符串是没有意义的, 所以必须由所处的文件系统把它们分开存储。

对于另一个例子, 7.3.4 节中讨论的客户程序-服务程序模型假定值和元组都容量适中, 而且服务程序将回答查询的整个元组集返回给客户。而立即把整个 BLOB 传给客户是没有意义的。例如, 如果客户向服务程序查找一个视频的片段, 服务程序应该在一段时间只传送过来一小段, 或许是几秒钟的有效视频信息。这样, 客户就可以开始放映这部电影, 而不必在本地存储几 G 字节的视频, 也不必一直等到把整个视频都接收完才开始播放。

义。也就是, 如果 A 和 B 是类型 Mpeg 的值, 那么 $A < B$ 的写法是非法的。

8.6.2 ADT 方法的定义

在 ADT 的属性表之后, 可以增加任何函数说明的列表。函数说明的格式如下:

FUNCTION 名字 (自变量) RETURNS 类型 ;

每个自变量都由变量名及其变量类型组成。自变量之间用逗号分开。

函数有两种类型: 内部的和外部的。外部函数用宿主语言编写, 只有它们的署名出现在 ADT 的定义中。我们将在 8.6.3 节讨论外部函数。内部函数用扩充的 SQL 编写。下面是一些选项, 其中包括对 SQL2 以及对 SQL3 的查询语言部分的扩充。

- `=` 用来作为赋值运算符。
- 要说明函数的局部变量可以给出它的名字, 并在前面加上冒号, 后面加上它的类型。
- 点运算符用来访问结构的分量。
- 可以在 WHERE 子句中表达布尔值。
- 可以用 BEGIN 和 END 把若干语句聚集到函数体中。

例 8.32 让我们继续研究例 8.30, 在该例中, 我们定义了一个地址 ADT。图 8.19 给出了一些函数, 我们可以把这些函数和图 8.17 中的类型建立语句合在一起。

1) 到 6) 行为 ADT AddressADT 定义了构造函数。回忆一下, SQL3 提供了 0 自变量的内部构造函数, 名为 AddressADT(ADT 本身的名字)。然而, 我们希望有另一个构造函数, 它以 stree 和 city 的值为参数。我们可以在需要的时候调用它, 而使用同样的名字作为类是合法的和适宜的。

在 1) 行, 我们看到新的构造函数说明。它有两个自变量 s 和 c, 分别表示 street 和 city。它们的类型分别是长度为 50 和 20 的字符串。函数返回类型为 AddressADT 的值。

2) 行说明 a 为 AddressADT 类型的局部变量。

3) 到 6) 行是函数体。在 3) 行我们用内部构造函数 AddressADT() 来建立新的对象, 并同时使它成为变量 a 的值。注意, 不能把内部构造函数和我们写的函数相混淆, 因为

两个函数的自变量不同。也就是, 3) 行不能误解为递归调用。4) 行把第一个自变量复制到 a 的 street 分量中, 而 5) 行把第二个自变量复制到 a 的 city 分量中。最后, 6) 行返回构造值 a。

7) 和 8) 行定义了 ADT AddressADT 的相等函数。回忆一下图 8.17 的 4) 行, 把 AddressADT 的相等函数说明为 addrEq, 所以我们必须使用该名字。该函数很简单, 当且仅当两个值的 street 和 city 分量都匹配时才返回 TRUE。该函数实际上是默认相等——值相同——也可能已经像例 8.31 那样默认了。

```
1)  FUNCTION AddressADT(  s CHAR( 50),   c CHAR(20) )
        RETURNS AddressADT;
2)      a AddressADT;
        BEGIN
3)          a  = AddressADT();
4)          a.street  =  s;
5)          a.city   =  c;
6)          RETURN  a;
        END;

7)  FUNCTION addrEq(   a1 AddressADT,   a2 AddressADT )
        RETURNS BOOLEAN;
8)      RETURN (   a1.street =   a2.street AND
                  a1.city =   a2.city );

9)  FUNCTION addrLT(   a1 AddressADT,   a2 AddressADT )
        RETURNS BOOLEAN;
10)     RETURN (   a1.city <   a2.city OR
                  a1.city =   a2.city AND a1.street <   a2.street );

11) FUNCTION fullAddr(   a AddressADT ) RETURNS CHAR(82);
12)     z CHAR( 10 );
        BEGIN
13)         z = findZip(   a.street,   a.city );
14)         RETURN(   a.street @@' ' @@| a.city @@' ' @@| z );
        END;
```

图 8.19 地址 ADT 的某些函数

9) 和 10) 行是 < 函数, addrLT。在这里, 如果第一个城市按字典顺序(按字母顺序)比第二个城市排在前面, 我们就可以说第一个地址在第二个之前。如果城市相同, 我们就比较街道的名字。

11) 到 14) 行定义了函数 fullAddr, 它处理类型为 AddressADT 的对象并返回整个地址, 也就是街道地址、城市和 9 位数的(加上连字符)邮政编码。12) 行说明局部变量 z 临时保存邮政编码。在 13) 行调用函数 findZip。该函数是外部定义的, 并有两个字符串自变量分别表示街道地址和城市。我们将在 8.6.3 节讨论外部函数说明的格式。

通过某种复杂的处理, 可能在另一个数据库或者复杂的判定序列中进行查找, findZip 将返回该街道和城市的正确邮政编码。在这里我们将不试图写出 findZip。最后, 在

14)行,我们把从对象 a 中得到的街道和城市与保存在 z 中的邮政编码连接起来。我们在该地址的三个分量之间加入单个空格隔开它们。

8.6.3 外部函数

ADT 可能还有用某种宿主语言而不是用 SQL3 编写的方法。但只有函数的署名出现在 ADT 定义中,同时指出编写函数所用的语言,我们才能使用这种函数。外部说明的格式如下:

```
DECLARE EXTERNAL 函数名 署名
LANGUAGE 语言名
```

例 8.33 为了使用例 8.32 中的外部函数 findZip,我们需要在 ADT AddressADT 的定义中说明它。因为该函数有两个自变量,分别是长度为 50 和 20 的字符串,并且返回长度为 10 的字符串,所以合适的说明为:

```
DECLARE EXTERNAL findZip
CHAR(50) CHAR(20) RETURNS CHAR(10)
LANGUAGE C;
```

把语言说明为 C 意味着,应把地址自变量以适合于 C 语言程序的格式传给 findZip。

8.6.4 本节练习

* 练习 8.6.1: 定义“PC”抽象数据类型,其对象表示个人计算机,包括处理器速度、内存容量、硬盘容量、CD 的速度和价格。

练习 8.6.2: 使用由内部函数扩展的 SQL,为练习 8.6.1 中你定义的 ADT 编写下列函数:

- * (a) 名为 newPC 的构造函数,它具有 PC ADT 的 5 个属性值,并且返回该类型的新对象。回忆一下,你可以(必须)用内部构造函数 PC()来定义该函数。
- * (b) 函数 value,用 PC 对象作为自变量并返回对 PC 的“评价”,它是表明该 PC 如何“好”的实数。求 PC“值”的公式是处理器速度加上 5 倍的 RAM(用 M 字节表示),再加上 50 倍的硬盘(用 G 字节表示),和 10 倍的 CD 速度。
- (c) 函数 better,用 PC 对象作为自变量,并返回另一 PC 对象,后者具有两倍的处理器速度、内存容量、硬盘和 CD 速度,并有同样的价格。你可以利用(a)中的构造函数 newPC。
- (d) 作为 PC 对象相等的函数 equalPC。如果两台 PC 具有同样的速度和硬盘容量,该函数就报告它们是“相等的”而不管其他分量的值。
- (e) 作为 ADT PC 的小于函数的函数 ltPC。如果 PC p₁的“值”小于 PC p₂,那么该函数就认为 p₁< p₂,其中“值”就像在(b)中定义的那样,你可以使用(b)中定义的函数 value。

练习 8.6.3: 为舰艇定义抽象数据类型 Ship,包括舰艇的名字、下水日期、火炮的数目、火炮的口径、排水量、该舰艇在战斗中的一段 MPEG 编码视频片段,以及关于该舰艇历史的附录文献。

练习 8.6.4: 为练习 8.6.3 中的 Ship ADT 写出下列函数的说明和定义。

- (a) 函数 firePower, 用 Ship 对象作为自变量, 返回“火力”, 它是火炮数目乘以口径的立方。
- (b) 函数 playVideo, 用 Ship 对象作为自变量, 使用外部定义的播放 MPEG 文件的 playMpeg 函数(你必须说明它)放映该舰艇的视频。
- (c) 构造函数 newShip, 它接受作为名字的值(而没有其他分量), 并返回具有该名字的新 Ship 对象。
- (d) 作为 Ship 对象相等的函数 equalShips。如果两艘舰艇名字相同并在同一年下水, 那么该函数就报告这两艘舰艇相等, 而不管其他分量的值。
- (e) 作为 ADT Ship 的小于函数的 ltShip 函数。如果舰艇 s_1 的名字按字母顺序比舰艇 s_2 的名字排在前面, 或者如果名字相同但是 s_1 比 s_2 下水早, 那么该函数就认为 $s_1 < s_2$ 。

8.7 ODL/OQL 和 SQL3 方法的比较

在列举了为面向对象的数据库管理而提出的两种主要的标准——ODL/OQL 和 SQL3 之后, 我们应该看到这两种方法在许多方面是不同的。相似之处多于不同之处, 这也是事实, 而且即使这两种方法源于非常不同的模型——面向对象的编程语言和关系数据库语言——但是它们都有效地采用了另一种核心模型的许多基础部分。

在本节, 我们将列出这两种方法的原则区别以及对于折衷范围选择的不同点。同时, 我们将指出 SQL3 行类型和 ADT 彼此之间以及和 ODL 的接口(类)之间的区别。这样, 事实上, 我们比较的是面向对象的三种不同方法: ODL/OQL、SQL3 行类型和 SQL 值类型。

1. 编程环境。OQL 假定它的语句将嵌入到编程语言中, 二者共享同样的编程和数据模型。假定这种语言是面向对象的; 例如, C++ , SmallTalk 或者 Java。另一方面, SQL3 假定它的对象不是外层的宿主语言的对象。在所有的 SQL 标准和实现中, 都有一个限定的接口允许在 SQL 存储的数据和宿主语言变量之间传递值。在 SQL3 ADT 中使用外部函数是附加的通信机制, 它补充了通常的 SQL/ 宿主语言接口, 就像我们在 7.1 节看到的那样。

2. 关系的作用。对于 SQL3 的数据视图来说, 关系仍然处于中心位置。实际上, 行类型描述关系, 而 ADT 描述属性的新类型。另一方面, 由于对象或者结构的集合和包在 select -from -where 语句中的作用, 它们对于 OQL 来说是基本的。ODL/OQL 的结构聚集与 SQL3 的关系非常类似。

3. 封装性。行类型是没有封装的。可以用 SQL 允许的所有方式对给定行类型的关系、元组和分量进行查询和更新。SQL3 抽象数据类型按通常的意义封装。在封装的方法上, ODL 的类非常类似于 SQL3 的 ADT。

4. 类的范围。OQL 假定每个类维护着一个单独的范围。引用(也就是 OQL 术语中的联系)总是访问该范围中的某个或者某些成员。在 SQL3 中, 我们可以维护一个行类型的

范围,也就是,包含该类型的每个已有元组的关系,但是我们没有义务这样做。如果我们没有行类型的范围,那么查找由给定引用所引用的元组所在的关系时就可能出现问題,就像 8.5.6 节中关于引用的作用域所讨论的那样。

5. 对象的可变性。对象一旦建立了就是不可变的,它的值没有一部分可以改变。像整数和字符串这种基本类型的对象在该意义上是不可变的。如果对象的分量可以改变,而对象保持它的对象标识不变,那么该对象就称为是可变的。尽管在 ODL/OQL 中假定对象更新是发生在其外层的宿主语言中,而不是通过 OQL,但是 ODL 类和 SQL3 行类型都定义了可变对象的类。SQL ADT 对象并不是完全不可变的。然而,把它们的值用变异函数处理将产生新的值,它可以代替旧的值,这很像整数值属性上的 SQL UPDATE 语句产生新整数,它可以代替该元组中的旧整数。

6. 对象标识。ODL 和 SQL3 ADT 都遵循对对象标识的传统解释:它是系统产生的量,用户不能存储或操作它。然而,SQL3 行类型的引用不遵循该原则。用户可以在关系中建立一列,而元组的对象标识就存放在该元组本身的这一列中,好像它就是普通的值。结果元组的对象标识可以作为关系的键码。尽管它确实由于修改或者删除把悬挂的引用带到了数据库,但是这种能力是有一定意义的。由于不允许对象标识成为属性,典型的关系都具有两个键码:对象标识和像社会保险号或者“证书号”(我们用在不断滚动的电影实例中)这样的替代值。

8.8 本章总结

ODL 中的方法:除了在第 2 章学过的属性和联系之外,ODL 允许我们说明方法作为接口规定的一部分。我们只定义方法的署名,也就是输入和输出参数的类型。方法本身在外层的程序中定义并用面向对象的宿主语言来编写。

OQL 类型系统:OQL 中的类型是由类名和原子类型(例如,整数)构成的。类型构造符 Struct 用来构造结构以及集合、包、列表和数组的聚集类型。这样,除了在 OQL 中类型构造符的嵌套深度没有限制以外,其类型系统与 ODL 一样。

OQL 中的 select-from-where 语句:OQL 提供了与 SQL 类似的 select-from-where 表达式。在 FROM 子句中,我们可以说明覆盖所有聚集的变量,包括类的范围(类似于关系)和对象属性值的聚集。

OQL 的通用运算符:OQL 提供了实质上与 SQL 类似的全称(for-all)、存在(exists)、IN、并(union)、交(intersection)、差(difference)和聚合(agggregation)运算符。然而,聚合总是在聚集上进行的,而不是在关系的一列上进行的。

OQL 的分组:OQL 也在 select-from-where 语句中提供 GROUP BY 子句,这和 SQL 相类似。然而,在 OQL 中,每个组的对象聚集可以通过称为 partition 的域名来显式地访问。

从 OQL 聚集中提取元素:我们可以用 ELEMENT 运算符来获得单个元素聚集中的单个成员。我们可以用以下方法来访问具有多个成员的聚集的元素:首先用 select-from-where 语句中的 ORDER BY 子句把聚集转换成列表,然后用外层的

宿主语言程序的循环按顺序访问该列表的每个元素。

SQL3 中的对象: SQL3 提供了两种类型的对象: 行类型和抽象数据类型。行类型是元组的类型, 而抽象数据类型则是元组分量的类型。

行类型的对象标识: 每个行类型都有一个引用类型, 而且该引用类型的值是元组的对象 ID。SQL3 允许属性的类型为对它所在关系的行类型的引用, 而且该属性的值为它所在元组的对象 ID, 这样就允许对象 ID 也作为所在关系的键码属性。

SQL3 中的抽象数据类型: 在 SQL3 中可以用 CREATE TYPE 语句说明 ADT。ADT 的值是具有一个或者多个分量的记录结构, 可能还有相关的方法。

SQL3 ADT 的方法: 可以为 ADT 说明函数(方法)。它们可以用类似于 SQL 的编程语言编写, 也可以说明为用宿主语言编写的外部函数。

8.9 本章参考文献

OQL 的引用和 ODL 相同: [1]。有关 SQL3 的资料可以按照第 5 章文献注释中的描述来获得。此外, [3] 是行类型的来源, [2] 是 SQL3 抽象数据类型的早期说明, 从那时起, 该标准已在许多方面得到了发展。

- [1] Cattell, R. G. G. (ed.), The Object Database Standard: ODMG-93 Release 1. 2, Morgan-Kaufmann, San Francisco, 1996.
- [2] Melton, J., J. Bauer, and K. Kulkarni, Object ADT's (with improvements for value ADT's), ISO WG3 report X3H2-91-083, April, 1991.
- [3] Kulkarni, K., M. Carey, L. DeMichiel, N. Mattos, W. Hong, and M. Ubell, Introducing reference types and cleaning up SQL3' s object model, ISO WG3 report X3H2-95-456, Nov., 1995.