关系数据库设计

在本章中，我们考虑为关系数据库设计模式的问题。其中的许多问题和我们在第 7 章中使用 E-R 模型时所考虑的设计问题相似。

一般而言，关系数据库设计的目标是生成一组关系模式，使我们存储信息时避免不必要的冗余，并且让我们可以方便地获取信息。这是通过设计满足适当*范式*(normal form)的模式来实现的。要确定一个关系模式是否属于期望的范式，我们需要数据库所建模的真实企业的信息。其中的一部分信息存在于设计良好的 E-R 图中，但是可能还需要关于该企业的额外信息。

本章介绍基于函数依赖概念的关系数据库设计的规范方法。然后根据函数依赖及其他类型的数据依赖来定义范式。不过，我们首先从给定实体 - 联系设计转换得到的模式的角度，考察关系设计的问题。

8.1 好的关系设计的特点

第 7 章对实体 - 联系设计的研究为创建关系数据库设计提供了一个很好的起点。我们在 7.6 节中看到，可以直接从 E-R 设计生成一组关系模式。显然，生成的模式集的好(或差)首先取决于 E-R 设计的质量。本章后面将研究评价一组关系模式的满意度的精确方法。不过，通过利用我们已学过的概念，我们可以向好的设计迈一大步。

为了易于参照，我们在图 8-1 中重复大学数据库的模式。 323

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

图 8-1 大学数据库模式

8.1.1 设计选择：更大的模式

现在，让我们探究一下这个关系数据库设计的特点，以及一些其他的选择方案。假定我们用以下这个模式代替 *instructor* 模式和 *department* 模式：

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

这表示在 *instructor* 和 *department* 对应的关系上进行自然连接的结果。这似乎是个好主意，因为某些查询可以用更少的连接来表达，除非我们仔细考虑产生我们的 E-R 设计的大学的实际状况。

让我们考虑图 8-2 中 *inst_dept* 关系的实例。注意，我们对系里的每个教师都不得不重复一遍系信息(“*building*”和“*budget*”)。例如，关于计算机科学系的信息是(Taylor, 100000)，教师 Katz、Srinivasan 和 Brandt 的元组中均包含该信息。

所有这些元组的预算数额统一这一点很重要，否则我们的数据库将会不一致。在使用 *instructor* 和 *department* 的原始设计中，对每个预算数额存储一次。这说明使用 *inst_dept* 是一个坏主意，因为它重复存储预算数额，且要承担某些用户可能更新一条元组而不是所有元组中的预算数额，并因此产生不一致的风险。

即使我们决定容许冗余的问题，*inst_dept* 模式仍存在其他问题。假设我们在大学里创立一个新系。在上面的设计选择中，我们无法直接表示关于一个系的信息(*dept_name*，*building*，*budget*)，除非该系在学校中有至少一位教师。这是因为 *inst_dept* 表中的元组需要 *ID*、*name* 和 *salary* 的值。这意味着我们不能记录新成立的系的信息，直到该新系录用了第一位教师为止。在旧的设计中，模式 *department* 可以处理这个情况，但是在修改的设计中，我们将不得不创建一条 *building* 和 *budget* 为空值的元组。在一些情况下，空值会带来麻烦，正如我们在 SQL 的学习中所看到的。然而，如果我们认为这种情况对于我们不是问题的话，那么我们可以继续使用该修改的设计。

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

图 8-2 *inst_dept* 表

8.1.2 设计选择：更小的模式

再假定我们从 *inst_dept* 模式开始。我们将如何发现它需要信息重复，且应该分成 *instructor* 和 *department* 两个模式呢？

通过观察模式 *inst_dept* 上实际关系的内容，我们能够注意到为每位与系关联的教师都要列一遍办公楼和预算所导致的信息重复。但是，这是一个不可靠的处理方式。一个真实的数据库拥有大量模式以及数量甚至更多的属性。元组的数量可以为数百万或更多。探查重复将代价高昂。这个方法还存在一个甚至更根本的问题。它使我们无法确定没有重复是否仅仅是一个"幸运的"特别情况，或者它是否为一般规则的表现。在我们的例子中，我们将如何知道在我们的大学中每个系(由系名唯一标识)必须位于单个办公楼中并且必须具有单个预算数额呢？计算机科学系的预算数额出现三次且完全相同的情况仅是一个巧合吗？如果不回到企业本身并了解它的规则，我们就无法回答这些问题。特别是，我们将需要发现大学要求每个系(由系名唯一标识)必须只有一个办公楼和一个预算值。

324 ~ 325

在 *inst_dept* 的例子中，我们创建 E-R 设计的过程成功避免了这种模式的产生。但是，这个幸运的情况并不总出现。因此，我们需要让数据库设计者即使在 *dept_name* 不是所讨论模式的主码的情况下，也能定义如"*dept_name* 的每个特定的值对应至多一个 *budget*"这样的规则。换句话说，我们需要写这样一条规则"如果存在模式(*dept_name*，*budget*)，则 *dept_name* 可以作为主码"。这条规则被定义为**函数依赖**(functional dependency)

$$dept\_name \rightarrow budget$$

给定这样一条规则，我们现在就有足够的信息来发现 *inst_dept* 模式的问题了。由于 *dept_name* 不能是 *inst_dept* 的主码(因为一个系可能在模式 *inst_dept* 上的关系中需要多条元组)，因此预算数额可能会重复。

类似于这些的观察及由它们导出的规则(在此为函数依赖)让数据库设计者可以发现一个模式应拆分或分解成两个或多个模式的情况。不难发现，分解 *inst_dept* 的正确方法是同原始设计一样分解成模式 *instructor* 和 *department*。对于具有大量属性和多个函数依赖的模式，找到正确的分解要难得多。为了

处理这种情况，我们将依靠本章后面所介绍的规范方法。

并不是所有的模式分解都是有益的。考虑我们拥有的所有模式都由一个属性构成这样一个极端的情况。任何类型的有意义的联系都无法表示。现在考虑一个不那么极端的情况，我们把 *employee* 模式（见 7.8 节）：

employee (*ID*, *name*, *street*, *city*, *salary*)

分解为以下两个模式：

employee1 (*ID*, *name*)
employee2 (*name*, *street*, *city*, *salary*)

这个分解的缺陷在于企业可能拥有两个同名的雇员。在实际中这不是不可能的，因为许多文化中都有某些非常流行的名字。当然，每个人都要有一个唯一的雇员号，这就是 *ID* 能够作为主码的原因。举例来说，让我们假定两个雇员，均名为 Kim，在该大学工作，并且在原始设计中的 *employee* 模式上的关系中有以下元组：

(57766, Kim, Main, Perryridge, 75000)
(98776, Kim, North, Hampton, 67000)

图 8-3 所示为这些元组、利用分解产生的模式所生成的元组，以及我们试图用自然连接重新生成原始元组所得到的结果。如我们在图 8-3 中看到的，那两个原始元组伴随着两个新的元组出现于结果中，这两个新的元组将属于这两个名为 Kim 的职员的数据值错误地混合在一起。虽然我们拥有较多的元组，但是实际上从以下意义来看我们拥有较少的信息。我们能够指出某个街道、城市和工资信息属于一个名为 Kim 的人，但是我们无法区分是哪一个 Kim。因此，我们的分解无法表达关于大学雇员的某些重要的事实。显然，我们想要避免这样的分解。我们将这样的分解称为**有损分解**（lossy decomposition），反之则称为**无损分解**（lossless decomposition）。

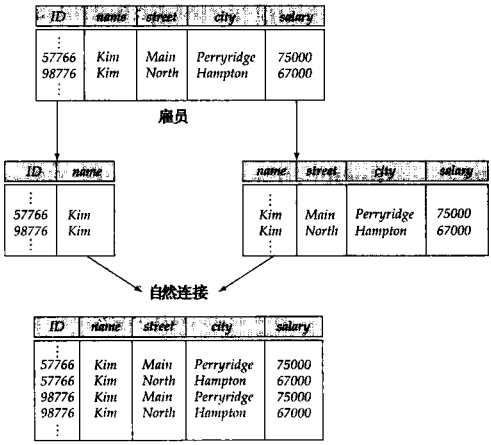

图 8-3　由不好的分解导致的信息丢失

8.2　原子域和第一范式

E-R 模型允许实体集和联系集的属性具有某些程度的子结构。特别地，它允许像图 7-11 中的 *phone_number* 这样的多值属性以及组合属性（诸如具有子属性 *street*、*city*、*state* 以及 *zip* 的属性 *address*）。当从包含这些类型的属性的 E-R 设计创建表时，要消除这种子结构。对于组合属性，让每个子属性本身成为一个属性。对于多值属性，为多值集合中的每个项创建一条元组。

326
327

在关系模型中，我们将属性不具有任何子结构这个思想形式化。一个域是**原子的**(atomic)，如果该域的元素被认为是不可分的单元。我们称一个关系模式 R 属于**第一范式**(First Normal Form，1NF)，如果 R 的所有属性的域都是原子的。

名字的集合是一个非原子值的例子。例如，如果关系 *employee* 的模式包含一个属性 *children*，它的域元素是名字的集合，该模式就不属于第一范式。

组合属性，比如包含子属性 *street*、*city*、*state* 和 *zip* 的属性 *address*，也具有非原子域。

假定整数是原子的，那么整数的集合是一个原子域；然而，所有整数集的集合是一个非原子域。区别在于，我们一般不认为整数具有子部分，但是我们认为整数的集合具有子部分——构成该集合的那些整数。不过，重要的问题不是域本身是什么，而是在数据库中如何使用域元素。如果我们认为每个整数是一列有序的数字，那么全部整数的域就是非原子的。

作为上述观点的实际例证，考虑一个机构，它给雇员分配下述样式的标识号：前两个字母表示系，剩下的四位数字是雇员在该系内的唯一号码。例如这样的号码可以是“CS0012”和“EE1127”。这样的标识号可以分成更小的单元，因此是非原子的。如果一个关系模式的一个属性的域是由如上编码的标识号所组成，则该模式不属于第一范式。

当采用这种标识号时，雇员所属的系可以通过编写解析标识号结构的代码得到。这么做需要额外的编程，而且信息是在应用程序中而不是在数据库中编码。如果这种标识号用作主码，还会产生进一步的问题：当一个雇员换了系时，该雇员的标识号在它所出现的每个地方都必须修改(这会是一个困难的任务)，否则解释该号码的代码将会给出错误的结果。

根据以上讨论，我们可能会使用形如“CS-101”的课程标识号，其中“CS”表示计算机科学系，这意味着课程标识号的域不是原子的。使用该系统的人认为这样的域不是原子的。然而，只要数据库应用没有将标识号拆开并将标识号的一部分解析为系的缩写，它仍然将该域视为原子的。*course* 模式将系名
328 作为一个单独的属性存储，数据库应用就可以根据这个属性值找到课程所属系，而不需要解析课程标识号中特定的字母。因此，我们的大学模式可以被认为属于第一范式。

使用以集合为值的属性会导致冗余存储数据的设计，进而会导致不一致。比如，数据库设计者可能不将教师和课程安排之间的联系表示为一个单独的关系 *teaches*，而是尝试为每一个教师存储一个课程段标识号的集合，并为每一个课程段存储一个教师标识号的集合。(*section* 和 *instructor* 的主码用作标识号。)每当关于哪位教师讲授哪个课程段的数据变化时，必须在两个地方执行更新：课程段的教师集合，以及教师的课程段集合。对两个更新的执行失败，会导致数据库处于不一致的状态。只保留其中一个集合，即要么课程段的教师集合，要么教师的课程段集合，将避免重复的信息；然而，只保留其中一个集合将使某些查询变得复杂，并且也不好确定保留哪一个。

有些类型的非原子值使用的时候要小心，但是它们可能很有用。例如，组合值属性常常很有用，而且很多情况下以集合为值的属性也是有用的，所以在 E-R 模型里这两种值都是支持的。在许多含有复杂结构的实体域中，强制使用第一范式会给应用程序员造成不必要的负担，他必须编写代码把数据转换成原子形式。从原子形态来回转换数据也会有运行时的额外开销。所以在这样的域里支持非原子的值是很有用的。事实上，如我们将在第 22 章看到的，现代数据库系统确实支持很多类型的非原子值。然而，本章我们限定只讨论属于第一范式的关系，因此，所有的域都是原子的。

8.3 使用函数依赖进行分解

在 8.1 节中，我们知道存在一个规范方法判断一个关系模式是否应该分解。这个方法基于码和函数依赖的概念。

在讨论关系数据库设计的算法时，我们需要针对任意的关系及其模式讨论，而不只是讨论例子。回想第 2 章对关系模型的介绍，我们在这里对我们的表示法进行概述。

- 一般情况下，我们用希腊字母表示属性集(例如 α)。我们用一个小写的罗马字母后面跟一个用一对圆括号括住的大写字母来指关系模式(例如 $r(R)$)。我们用表示法 $r(R)$ 表示该模式是关系 r 的，R 表示属性集，不过当我们不关心关系的名字时经常简化我们的表示法而只用 R。

当然，一个关系模式是一个属性集，但是并非所有的属性集都是模式。当使用一个小写的希腊字母时，我们是指一个有可能是模式也可能不是模式的属性集。当我们希望指明属性集一定为一个模式时，就使用罗马字母。 329

- 当属性集是一个超码时，我们用 K 表示它。超码属于特殊的关系模式，因此我们使用术语“K 是 $r(R)$ 的超码”。
- 我们对关系使用小写的名字。在我们的例子中，这些名字是企图有实际含义的（例如 *instructor*），而在我们的定义和算法中，我们使用单个字母，比如 r。
- 当然，一个关系在任意给定时间都有特定的值；我们将那看作一个实例并使用术语“r 的实例”。当我们明显在讨论一个实例时，我们可以仅用关系的名字（例如 r）。

8.3.1　码和函数依赖

一个数据库对现实世界中的一组实体和联系建模。在现实世界中，数据上通常存在各种约束（规则）。例如，在一个大学数据库中预计要保证的一些约束有：

- 学生和教师通过他们的 ID 唯一标识。
- 每个学生和教师只有一个名字。
- 每个教师和学生只（主要）关联一个系。[⊖]
- 每个系只有一个预算值，且只有一个关联的办公楼。

一个关系的满足所有这种现实世界约束的实例，称为关系的**合法实例**（legal instance）；在一个数据库的合法实例中所有关系实例都是合法实例。

几种最常用的现实世界约束可以形式化地表示为码（超码、候选码以及主码），或者下面所定义的函数依赖。

在 2.3 节中，曾定义*超码*的概念为可以唯一标识关系中一条元组的一个或多个属性的集合。在这里重新表述该定义如下：令 $r(R)$ 是一个关系模式。R 的子集 K 是 $r(R)$ 的**超码**（superkey）的条件是：在关系 $r(R)$ 的任意合法实例中，对于 r 的实例中的所有元组对 t_1 和 t_2 总满足，若 $t_1 \neq t_2$，则 $t_1[K] \neq t_2[K]$。也就是说，在关系 $r(R)$ 的任意合法实例中没有两条元组在属性集 K 上可能具有相同的值。显然，如果 r 中没有两条元组在 K 上具有相同的值，那么在 r 中一个 K 值唯一标识一条元组。 330

鉴于超码是能够唯一标识整条元组的属性集，函数依赖让我们可以表达唯一标识某些属性的值的约束。考虑一个关系模式 $r(R)$，令 $\alpha \subseteq R$ 且 $\beta \subseteq R$。

- 给定 $r(R)$ 的一个实例，我们说这个实例**满足**（satisfy）**函数依赖** $\alpha \rightarrow \beta$ 的条件是：对实例中所有元组对 t_1 和 t_2，若 $t_1[\alpha] = t_2[\alpha]$，则 $t_1[\beta] = t_2[\beta]$。
- 如果在 $r(R)$ 的每个合法实例中都满足函数依赖 $\alpha \rightarrow \beta$，则我们说该函数依赖在模式 $r(R)$ 上**成立**（hold）。

使用函数依赖这一概念，我们说如果函数依赖 $K \rightarrow R$ 在 $r(R)$ 上成立，则 K 是 $r(R)$ 的一个超码。也就是说，如果对于 $r(R)$ 的每个合法实例，对于实例中每个元组对 t_1 和 t_2，凡是 $t_1[K] = t_2[K]$，总有 $t_1[R] = t_2[R]$（即 $t_1 = t_2$），则 K 是一个超码。[⊜]

函数依赖使我们可以表示不能用超码表示的约束。在 8.1.2 节中我们曾考虑模式：

$$inst\_dept\ (ID, name, salary, dept\_name, building, budget)$$

在该模式中函数依赖 $dept\_name \rightarrow budget$ 成立，因为对于每个系（由 $dept\_name$ 唯一标识）都存在唯一的预算数额。

属性对（ID，$dept\_name$）构成 $inst\_dept$ 的一个超码，我们将这一事实记做：

⊖ 一个教师或一个学生可以和多个系相关联，例如兼职教师或者辅修系。我们简化的大学模式只对每个教师或学生所关联的主系建模。一个实际的大学模式会在另外的关系中表示次要的关联。

⊜ 注意，我们在这里假设关系为集合。SQL 处理多集，并且 SQL 中声明一个属性集 K 为主码不仅需要当 $t_1[K] = t_2[K]$ 时 $t_1 = t_2$，还需要没有重复的元组。SQL 还要求 K 集中的属性不能赋予空值。

ID, $dept\_name \rightarrow name$, $salary$, $building$, $budget$

我们将以两种方式使用函数依赖：

- 判定关系的实例是否满足给定函数依赖集 F。
- 说明合法关系集上的约束。因此，我们将只关心满足给定函数依赖集的那些关系实例。如果我们希望只考虑模式 R 上满足函数依赖集 F 的关系，我们说 F 在 $r(R)$ 上**成立**。

让我们考虑图 8-4 中的关系 r 的实例，看看它满足什么函数依赖。注意到 $A \rightarrow C$ 是满足的。存在两条元组的 A 值为 a_1。这些元组具有相同的 C 值，c_1。类似地，A 值为 a_2 的两条元组具有相同的 C 值，c_2。没有其他元组对具有相同的 A 值。但是，函数依赖 $C \rightarrow A$ 是不满足的。为了说明这一点，考虑元组 $t_1 = (a_2, b_3, c_2, d_3)$ 和元组 $t_2 = (a_3, b_3, c_2, d_4)$。这两条元组具有相同的 C 值，c_2，但它们具有不同的 A 值，分别为 a_2 和 a_3。因此，我们找到一对元组 t_1 和 t_2，使得 $t_1[C] = t_2[C]$，但 $t_1[A] \neq t_2[A]$。

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

图 8-4 关系 r 的实例的例子

有些函数依赖称为**平凡的**(trivial)，因为它们在所有关系中都满足。例如，$A \rightarrow A$ 在所有包含属性 A 的关系中满足。从字面上看函数依赖的定义，我们知道，对所有满足 $t_1[A] = t_2[A]$ 的元组 t_1 和 t_2，有 $t_1[A] = t_2[A]$。同样，$AB \rightarrow A$ 也在所有包含属性 A 的关系中满足。一般地，如果 $\beta \subseteq \alpha$，则形如 $\alpha \rightarrow \beta$ 的函数依赖是**平凡的**。

重要的是，要认识到一个关系实例可能满足某些函数依赖，它们并不需要在关系的模式上成立。在图 8-5 的 *classroom* 关系的实例中，我们发现 $room\_number \rightarrow capacity$ 是满足的。但是，我们相信，在现实世界中，不同建筑里的两个教室可以具有相同的房间号，但具有不同的空间大小。因此，有时有可能存在一个 *classroom* 关系的实例，不满足 $room\_number \rightarrow capacity$。所以，我们不将 $room\_number \rightarrow capacity$ 包含于 *classroom* 关系的模式上成立的函数依赖集中。然而，我们会期望函数依赖 $building, room\_number \rightarrow capacity$ 在 *classroom* 模式上成立。

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

图 8-5 *classroom* 关系的实例

给定关系 $r(R)$ 上成立的函数依赖集 F，有可能会推断出某些其他的函数依赖也一定在该关系上成立。例如，给定模式 $r(A, B, C)$，如果函数依赖 $A \rightarrow B$ 和 $B \rightarrow C$ 在 r 上成立，可以推出函数依赖 $A \rightarrow C$ 也一定在 r 上成立。这是因为，给定 A 的任意值，仅存在 B 的一个对应值，且对于 B 的那个值，只能存在 C 的一个对应值。我们稍后将在 8.4.1 节学习如何进行这种推导。

331
{
332

我们将使用 F^+ 符号来表示 F 集合的**闭包**(closure)，也就是能够从给定 F 集合推导出的所有函数依赖的集合。显然，F^+ 包含 F 中所有的函数依赖。

8.3.2 Boyce-Codd 范式

我们能达到的较满意的范式之一是 **Boyce-Codd 范式**(Boyce-Codd Normal Form，BCNF)。它消除所有基于函数依赖能够发现的冗余，虽然，如我们将在 8.6 节中看到的，可能有其他类型的冗余还保留着。具有函数依赖集 F 的关系模式 R 属于 BCNF 的条件是，对 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖(其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$)，下面至少有一项成立：

- $\alpha \rightarrow \beta$ 是平凡的函数依赖(即，$\beta \subseteq \alpha$)。
- α 是模式 R 的一个超码。

一个数据库设计属于 BCNF 的条件是，构成该设计的关系模式集中的每个模式都属于 BCNF。

我们已经在 8.1 节中见过了不属于 BCNF 的关系模式的例子：

$inst\_dept$ (ID, $name$, $salary$, $dept\ name$, $building$, $budget$)

函数依赖 $dept\_name \rightarrow budget$ 在 $inst\_dept$ 上成立，但是 $dept\_name$ 并不是超码(因为一个系可以有多个不同的教师)。在 8.1.2 节中，我们看到把 $inst\_dept$ 分解为 *instructor* 和 *department* 是一个更好的设计。模式 *instructor* 属于 BCNF。所有成立的非平凡的函数依赖，例如：

$ID \rightarrow name$, $dept\_name$, $salary$

在箭头的左侧包含 *ID*，且 *ID* 是 *instructor* 的一个超码(事实上，在这个例子中，是主码)。(也就是说，在不包含 *ID* 的另一侧上，对于 *name*、*dept_name* 和 *salary* 的任意组合不存在非平凡的函数依赖。)因此，*instructor* 属于 BCNF。

类似地，*department* 模式属于 BCNF，因为所有成立的非平凡函数依赖，例如：

$$dept\_name \rightarrow building, budget$$

333

在箭头的左侧包含 *dept_name*，且 *dept_name* 是 *department* 的一个超码(和主码)。因此，*department* 属于 BCNF。

我们现在讲述分解不属于 BCNF 的模式的一般规则。设 R 为不属于 BCNF 的一个模式。则存在至少一个非平凡的函数依赖 $\alpha \rightarrow \beta$，其中 α 不是 R 的超码。我们在设计中用以下两个模式取代 R：

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

在上面的 *inst_dept* 例子中，α = *dept_name*，β = {*building*, *budget*}，且 *inst_dept* 被取代为：

- $(\alpha \cup \beta)$ = (*dept_name*, *building*, *budget*)
- $(R - (\beta - \alpha))$ = (*ID*, *name*, *dept_name*, *salary*)

在这个例子中，结果是 $\beta - \alpha = \beta$。我们需要像上述那样的表述规则，从而正确处理箭头两边都出现的属性的函数依赖。技术上的原因我们会在 8.5.1 节介绍。

当我们分解不属于 BCNF 的模式时，产生的模式中可能有一个或多个不属于 BCNF。在这种情况中，需要进一步分解，其最终结果是一个 BCNF 模式集合。

8.3.3　BCNF 和保持依赖

我们已经看到多种表达数据库一致性约束的方式：主码约束、函数依赖、check 约束、断言和触发器。在每次数据库更新时检查这些约束的开销很大，因此，将数据库设计成能够高效地检查约束是很有用的。特别地，如果函数依赖的检验仅需要考虑一个关系就可以完成，那么检查这种约束的开销就很低。我们将看到，在有些情况下，到 BCNF 的分解会妨碍对某些函数依赖的高效检查。

对此举例，假定我们对我们的大学机构做一个小的改动。在图 7-15 的设计中，一位学生只能有一位导师。这是由从 *student* 到 *advisor* 的联系集 *advisor* 为多对一而推断出的。我们要做的"小"改动是一个教师只能和单个系关联，且一个学生可以有多个导师，但是一个给定的系中至多一位。⊖

利用 E-R 设计实现这个改动的一种方法是，把联系集 *advisor* 替换为涉及实体集 *instructor*、*student* 和 *department* 的三元联系集 *dept_advisor*，它是从{*instructor*, *student*}对到 *department* 多对一的，如图8-6所示。该 E-R 图指明了"一个学生可以有多位导师，但是对应于一个给定的系最多只有一个"的约束。

图 8-6　联系集 *dept_advisor*

对应于新的 E-R 图，*instructor*、*department* 和 *student* 的模式没有变。然而，从 *dept_advisor* 导出的模式现在为：

$$dept\_advisor\ (s\_ID, i\_ID, dept\_name)$$

虽然没有在 E-R 图中指明，不过假定我们有附加的约束"一位教师只能在一个系担任导师。"

⊖　这样的安排对于双专业的学生是有意义的。

那么，下面的函数依赖在 *dept_advisor* 上成立：

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

第一个函数依赖产生于我们的需求“一位教师只能在一个系担任导师”。第二个函数依赖产生于我们的需求“对于一个给定的系，一个学生可以有至多一位导师”。

注意，在这种设计中，每次一名教师参与一个 *dept_advisor* 联系的时候，我们都不得不重复一次系的名称。我们看到 *dept_advisor* 不属于 BCNF，因为 *i_ID* 不是超码。根据 BCNF 分解规则，得到：

$$(s\_ID, i\_ID)$$
$$(i\_ID, dept\_name)$$

334 | 335 上面的两个模式都属于 BCNF。（事实上，你可以验证，按照定义任何只包含两个属性的模式都属于 BCNF。）然而注意，在 BCNF 设计中，没有一个模式包含函数依赖 $s\_ID, dept\_name \rightarrow i\_ID$ 中出现的所有属性。

由于我们的设计使得该函数依赖的强制实施在计算上很困难，因此我们称我们的设计不是**保持依赖的**（dependency preserving）。[⊖]由于常常希望保持依赖，因此我们考虑另外一种比 BCNF 弱的范式，它允许我们保持依赖。该范式称为第三范式。[⊜]

8.3.4 第三范式

BCNF 要求所有非平凡函数依赖都形如 $\alpha \rightarrow \beta$，其中 α 为一个超码。第三范式（3NF）稍微放宽了这个约束，它允许左侧不是超码的某些非平凡函数依赖。在定义 3NF 之前，我们回想到候选码是最小的超码——任何真子集都不是超码的超码。

具有函数依赖集 F 的关系模式 R 属于**第三范式**（third normal form）的条件是：对于 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖（其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$），以下至少一项成立：

- $\alpha \rightarrow \beta$ 是一个平凡的函数依赖。
- α 是 R 的一个超码。
- $\beta - \alpha$ 中的每个属性 A 都包含于 R 的一个候选码中。

注意上面的第三个条件并没有说单个候选码必须包含 $\beta - \alpha$ 中的所有属性；$\beta - \alpha$ 中的每个属性 A 可能包含于不同的候选码中。

前两个条件与 BCNF 定义中的两个条件相同。3NF 定义中的第三个条件看起来很不直观，并且它的用途也不是显而易见的。在某种意义上，它代表 BCNF 条件的最小放宽，以确保每一个模式都有保持依赖的 3NF 分解。它的用途在后面介绍 3NF 分解时会变得更清楚。

注意任何满足 BCNF 的模式也满足 3NF，因为它的每个函数依赖都将满足前两个条件中的一条。所以 BCNF 是比 3NF 更严格的范式。

3NF 的定义允许某些 BCNF 中不允许的函数依赖。只满足 3NF 定义中第三个条件的依赖 $\alpha \rightarrow \beta$ 在 BCNF 中是不允许的，但在 3NF 中是允许的。[⊝]

336 现在我们再次考虑联系集 *dept_advisor*，它具有以下函数依赖：

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

在 8.3.3 节中我们说函数依赖“$i\_ID \rightarrow dept\_name$”导致 *dept_advisor* 模式不属于 BCNF。注意这里 $\alpha = i\_ID$，$\beta = dept\_name$，并且 $\beta - \alpha = dept\_name$。由于函数依赖 $s\_ID, dept\_name \rightarrow i\_ID$ 在 *dept_advisor* 上成立，于是属性 *dept_name* 包含于一个候选码中，因此 *dept_advisor* 属于 3NF。

⊖ 从技术上来说，由于存在逻辑上蕴涵该依赖的其他依赖，因此属性在任何一个模式中都不完全出现的依赖也隐含地强制实施了。稍后我们将在 8.4.5 节讨论这个问题。

⊜ 你可能注意到我们跳过了第二范式。第二范式只有历史意义，已经不在实际中使用了。

⊝ 这些依赖是**传递依赖**（transitive dependency）的例子（参见实践习题 8.16）。3NF 的原始定义就是由传递依赖而来的，我们所使用的定义与之等价但更容易理解。

我们已经看到，当不存在保持依赖的 BCNF 设计时，必须在 BCNF 和 3NF 之间进行权衡。这些权衡将在 8.5.4 节详细讨论。

8.3.5 更高的范式

在某些情况中，使用函数依赖分解模式可能不足以避免不必要的信息重复。考虑在 *instructor* 实体集定义中的小变化，我们为每个教师记录一组孩子名字以及一组电话号码。电话号码可以被多个人共享。因此，*phone_number* 和 *child_name* 将是多值属性，并且根据我们从 E-R 设计生成模式的规则，我们会有两个模式，多值属性 *phone_number* 和 *child_name* 中每个属性对应一个模式：

$$(ID, child\_name)$$
$$(ID, phone\_number)$$

如果我们合并这些模式而得到

$$(ID, child\_name, phone\_number)$$

我们会发现该结果属于 BCNF，因为只有平凡的函数依赖成立。因此我们可能认为这样的合并是个好主意。然而，通过考虑有两个孩子和两个电话号码的教师的例子，我们会发现这样的合并是一个坏主意。例如，令 *ID* 为 99999 的教师有两个孩子，叫作“David”和“William”，以及有两个电话号码，512 - 555 - 1234 和 512 - 555 - 4321。在合并的模式中，我们必须为每个家属重复一次电话号码：

(99999, David, 512 - 555 - 1234)
(99999, David, 512 - 555 - 4321)
(99999, William, 512 - 555 - 1234)
(99999, William, 512 - 555 - 4321)

如果我们不重复电话号码，且只存储第一条和最后一条元组，我们就记录了家属名字和电话号码，但是结果元组将暗指 David 对应于 512 - 555 - 1234，而 William 对应于 512 - 555 - 4321。我们知道，这 337
是不正确的。

由于基于函数依赖的范式并不足以处理这样的情况，因此定义了另外的依赖和范式。我们在 8.6 和 8.7 节对此进行讲述。

8.4 函数依赖理论

在例子中我们已经看到，作为检查模式是否属于 BCNF 或 3NF 这一过程的一部分，能够对函数依赖进行系统地推理是很有用的。

8.4.1 函数依赖集的闭包

我们将会看到，给定模式上的函数依赖集 F，我们可以证明某些其他的函数依赖在模式上也成立。我们称这些函数依赖被 F **“逻辑蕴涵”**。当检验范式时，只考虑给定的函数依赖集是不够的；除此以外，还需要考虑模式上成立的所有函数依赖。

更正式地，给定关系模式 $r(R)$，如果 $r(R)$ 的每一个满足 F 的实例也满足 f，则 R 上的函数依赖 f 被 r 上的函数依赖集 F **逻辑蕴涵**(logically imply)。

假设我们给定关系模式 $r(A, B, C, G, H, I)$ 及函数依赖集：

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

那么函数依赖

$$A \rightarrow H$$

被逻辑蕴涵。也就是说，我们可以证明，一个关系只要满足给定的函数依赖集，这个关系也一定满足 $A \rightarrow H$。假设元组 t_1 及 t_2 满足

$$t_1[A] = t_2[A]$$

由于已知 $A \rightarrow B$，因此由函数依赖的定义推出

338

$$t_1[B] = t_2[B]$$

那么，又由于已知 $B \rightarrow H$，因此由函数依赖的定义推出

$$t_1[H] = t_2[H]$$

因此，我们证明了，对任意两个元组 t_1 及 t_2，只要 $t_1[A] = t_2[A]$，就一定有 $t_1[H] = t_2[H]$。而这正是 $A \rightarrow H$ 的定义。

令 F 为一个函数依赖集。F 的**闭包**是被 F 逻辑蕴涵的所有函数依赖的集合，记作 F^+。给定 F，可以由函数依赖的形式化定义直接计算出 F^+。如果 F 很大，则这个过程将会很长而且很难。这一 F^+ 运算需要论证，就像刚才用于证明 $A \rightarrow H$ 在依赖集例子的闭包中的那种论证。

公理(axiom)，或推理规则，提供了一种用于推理函数依赖的更为简单的技术。在下面的规则中，我们用希腊字母(α，β，γ，…)表示属性集，用字母表中从开头起的大写罗马字母表示单个属性。我们用 $\alpha\beta$ 表示 $\alpha \cup \beta$。

我们可以使用以下三条规则去寻找逻辑蕴涵的函数依赖。通过反复应用这些规则，可以找出给定 F 的全部 F^+。这组规则称为 **Armstrong 公理**(Armstrong's axiom)，以纪念首次提出这一公理的人。

- **自反律**(reflexivity rule)。若 α 为一属性集且 $\beta \subseteq \alpha$，则 $\alpha \rightarrow \beta$ 成立。
- **增补律**(augmentation rule)。若 $\alpha \rightarrow \beta$ 成立且 γ 为一属性集，则 $\gamma\alpha \rightarrow \gamma\beta$ 成立。
- **传递律**(transitivity rule)。若 $\alpha \rightarrow \beta$ 和 $\beta \rightarrow \gamma$ 成立，则 $\alpha \rightarrow \gamma$ 成立。

Armstrong 公理是**正确有效的**(sound)，因为它们不产生任何错误的函数依赖。这些规则是**完备的**(complete)，因为，对于给定函数依赖集 F，它们能产生全部 F^+。文献注解提供了对正确有效性和完备性的证明的参考。

虽然 Armstrong 公理是完备的，但是直接用它们计算 F^+ 会很麻烦。为进一步简化，我们列出另外的一些规则。可以用 Armstrong 公理证明这些规则是正确有效的(参见实践习题 8.4、8.5 及习题 8.26)。

- **合并律**(union rule)。若 $\alpha \rightarrow \beta$ 和 $\alpha \rightarrow \gamma$ 成立，则 $\alpha \rightarrow \beta\gamma$ 成立。
- **分解律**(decomposition)。若 $\alpha \rightarrow \beta\gamma$ 成立，则 $\alpha \rightarrow \beta$ 和 $\alpha \rightarrow \gamma$ 成立。
- **伪传递律**(pseudotransitivity rule)。若 $\alpha \rightarrow \beta$ 和 $\gamma\beta \rightarrow \delta$ 成立，则 $\alpha\gamma \rightarrow \delta$ 成立。

339

让我们将规则应用于模式 $R = (A, B, C, G, H, I)$ 及函数依赖集 $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$。这里列出 F^+ 中的几个依赖：

- $A \rightarrow H$。由于 $A \rightarrow B$ 和 $B \rightarrow H$ 成立，因此使用传递律。可以看到使用 Armstrong 公理证明 $A \rightarrow H$ 成立比本节前面直接使用定义论证要简单得多。
- $CG \rightarrow HI$。由于 $CG \rightarrow H$ 和 $CG \rightarrow I$ 成立，因此由合并律推出 $CG \rightarrow HI$。
- $AG \rightarrow I$。由于有 $A \rightarrow C$ 且 $CG \rightarrow I$，因此由伪传递律推出 $AG \rightarrow I$ 成立。

$AG \rightarrow I$ 的另一种推理方法如下：我们在 $A \rightarrow C$ 上使用增补律从而推出 $AG \rightarrow CG$。在这个依赖和 $CG \rightarrow I$ 上使用传递率，我们推出 $AG \rightarrow I$。

图 8-7 给出形式化地示范如何使用 Armstrong 公理计算 F^+ 的过程。在这个过程中，当一个函数依赖加入 F^+ 时，它可能已经存在了，这时 F^+ 没有变化。我们将在 8.4.2 节看到另一种计算 F^+ 的算法。

```
F⁺ = F
    repeat
        for each F⁺ 中的函数依赖 f
            在 f 上应用自反律和增补律
            将结果加入到 F⁺ 中
        for each F⁺ 中的一对函数依赖 f₁ 和 f₂
            if f₁ 和 f₂ 可以使用传递律结合起来
                将结果加入到 F⁺ 中
    until F⁺ 不再发生变化
```

图 8-7 计算 F^+ 的过程

函数依赖的左边和右边都是 R 的子集。由于包含 n 个元素的集合有 2^n 个子集，因此共有 $2^n \times 2^n = 2^{2n}$ 个可能的函数依赖，其中 n 是 R 中的属性个数。除最后一次迭代外，每一次执行 repeat 循环都至少往 F^+ 里加入一个函数依赖。因此，该过程保证可以终止。

8.4.2　属性集的闭包

如果 $\alpha \rightarrow B$，我们称属性 B 被 α **函数确定**(functionally determine)。要判断集合 α 是否为超码，我们必须设计一个算法，用于计算被 α 函数确定的属性集。一种方法是计算 F^+，找出所有左半部为 α 的函数依赖，并合并这些函数依赖的右半部。但是这么做开销很大，因为 F^+ 可能很大。

在本节后面我们将看到，一个用于计算被 α 函数确定的属性集的高效算法不仅可以用来判断 α 是否为超码，还可以用于其他的一些任务。

令 α 为一个属性集。我们将函数依赖集 F 下被 α 函数确定的所有属性的集合称为 F 下 α 的**闭包**，记为 α^+。图 8-8 是以伪码写的计算 α^+ 的算法。输入是函数依赖集 F 和属性集 α。输出存储在变量 *result* 中。

```
result := α;
    repeat
        for each 函数依赖 β→γ in F do
            begin
                if β⊆ result then result := result ∪ γ;
            end
    until (result 不变)
```

图 8-8　计算 F 下 α 的闭包 α^+ 的算法

为解释该算法如何进行，我们将用它计算 8.4.1 节中定义的函数依赖集下的 $(AG)^+$。开始时 *result* = AG。在第一次执行 **repeat** 循环测试各个函数依赖时，我们发现

- 由 $A \rightarrow B$，于是将 B 加入 *result*。这是因为，我们观察到 $A \rightarrow B$ 属于 F，$A \subseteq result$(即 AG)，所以 *result* := *result* ∪ B。
- 由 $A \rightarrow C$，*result* 变为 $ABCG$。
- 由 $CG \rightarrow H$，*result* 变为 $ABCGH$。
- 由 $CG \rightarrow I$，*result* 变为 $ABCGHI$。

我们第二次执行 **repeat** 循环时，没有新属性加入 *result*，算法终止。

让我们看看图 8-8 的算法为什么正确。第一步正确，因为 $\alpha \rightarrow \alpha$ 总是成立(由自反律)。我们说，对 *result* 的任意子集 β，有 $\alpha \rightarrow \beta$。由于开始 **repeat** 循环时 $\alpha \rightarrow$ *result* 为真，因此只要 $\beta \subseteq result$ 且 $\beta \rightarrow \gamma$，就可将 γ 加入 *result* 中。而由自反律得到 *result* $\rightarrow \beta$，故由传递律得到 $\alpha \rightarrow \beta$。再应用传递律就得到 $\alpha \rightarrow \gamma$(由 $\alpha \rightarrow \beta$ 及 $\beta \rightarrow \gamma$)。由合并律可推出 $\alpha \rightarrow$ *result* ∪ γ，所以 α 函数确定 **repeat** 循环中产生的任意新结果。也就是说，该算法所返回的任意属性都属于 α^+。

容易证明，该算法找出 α^+ 的全部属性。在执行当中，如果存在属性属于 α^+ 却还不属于 *result*，则必定存在函数依赖 $\beta \rightarrow \gamma$(其中 $\beta \subseteq result$)并且 γ 中至少有一个属性不在 *result* 中。当该算法结束时，所有的函数依赖都已经处理过，γ 中的属性都已经加到 *result* 中；因此我们可以确定 α^+ 中的所有属性都在 *result* 中。

340 / 341

经证明，在最坏情况下，该算法的执行时间为 F 集合规模的二次方。有一个更快的算法(但略微复杂一点儿)，执行时间与 F 的规模呈线性关系；这个算法作为实践习题 8.8 的一部分进行介绍。

属性闭包算法有多种用途：

- 为了判断 α 是否为超码，我们计算 α^+，检查 α^+ 是否包含 R 中的所有属性。
- 通过检查是否 $\beta \subseteq \alpha^+$，我们可以检查函数依赖 $\alpha \rightarrow \beta$ 是否成立(或换句话说，是否属于 F^+)。也就是说，我们用属性闭包计算 α^+，看它是否包含 β。本章后面我们将会看到这种检查非常有用。
- 该算法给了我们另一种计算 F^+ 的方法：对任意的 $\gamma \subseteq R$，我们找出闭包 γ^+；对任意的 $S \subseteq \gamma^+$，我们输出一个函数依赖 $\gamma \rightarrow S$。

8.4.3　正则覆盖

假设我们在一个关系模式上有一个函数依赖集 F。每当用户在该关系上执行更新时，数据库系统必须确保此更新不破坏任何函数依赖，也就是说，F 中的所有函数依赖在新数据库状态下仍然满足。

如果更新操作破坏了 F 上的任一个函数依赖，系统必须回滚该更新操作。

我们可以通过测试与给定函数依赖集具有相同闭包的简化集的方式来减小检测冲突的开销。因为简化集和原集具有相同的闭包，所以满足函数依赖简化集的数据库也一定满足原集，反之亦然。但是，简化集更便于检测。稍后我们将看到简化集是如何构造的。首先，我们需要一些定义。

如果去除函数依赖中的一个属性不改变该函数依赖集的闭包，则称该属性是**无关的**(extraneous)。**无关属性**(extraneous attribute)的形式化定义如下：考虑函数依赖集 F 及 F 中的函数依赖 $\alpha \rightarrow \beta$。

- 如果 $A \in \alpha$ 并且 F 逻辑蕴含 $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$，则属性 A 在 α 中是无关的。
- 如果 $A \in \beta$ 并且函数依赖集 $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ 逻辑蕴含 F，则属性 A 在 β 中是无关的。

例如，假定我们在 F 中有函数依赖 $AB \rightarrow C$ 和 $A \rightarrow C$，那么 B 在 $AB \rightarrow C$ 中是无关的。再比如，假定我们在 F 中有函数依赖 $AB \rightarrow CD$ 和 $A \rightarrow C$，那么 C 在 $AB \rightarrow CD$ 的右半部中是无关的。

使用无关属性的定义时注意蕴涵的方向：如果将左半部与右半部交换，蕴含关系将总是成立的。也就是说，$(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ 总是逻辑蕴涵 F，同时 F 也总是逻辑蕴涵 $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$。

下面介绍如何有效检验一个属性是否无关。令 R 为一关系模式，且 F 是在 R 上成立的给定函数依赖集。考虑依赖 $\alpha \rightarrow \beta$ 中的一个属性 A。

- 如果 $A \in \beta$，为了检验 A 是否是无关的，考虑集合
  $$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
  并检验 $\alpha \rightarrow A$ 是否能够由 F' 推出。为此，计算 F' 下的 α^+ (α 的闭包)；如果 α^+ 包含 A，则 A 在 β 中是无关的。
- 如果 $A \in \alpha$，为了检验 A 是否是无关的，令 $\gamma = \alpha - \{A\}$，并且检查 $\gamma \rightarrow \beta$ 是否可以由 F 推出。为此，计算在 F 下的 γ^+ (γ 的闭包)；如果 γ^+ 包含 β 中的所有属性，则 A 在 α 中是无关的。

例如，假定 F 包含 $AB \rightarrow CD$、$A \rightarrow E$ 和 $E \rightarrow C$。为检验 C 在 $AB \rightarrow CD$ 中是否是无关的，我们计算 $F' = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$ 下 AB 的属性闭包。闭包为 $ABCDE$，包含 CD，所以我们推断出 C 是无关的。

F 的**正则覆盖**(canonical cover) F_c 是一个依赖集，使得 F 逻辑蕴含 F_c 中的所有依赖，并且 F_c 逻辑蕴涵 F 中的所有依赖。此外，F_c 必须具有如下性质：

- F_c 中任何函数依赖都不含无关属性。
- F_c 中函数依赖的左半部都是唯一的。即，F_c 中不存在两个依赖 $\alpha_1 \rightarrow \beta_1$ 和 $\alpha_2 \rightarrow \beta_2$，满足 $\alpha_1 = \alpha_2$。

函数依赖集 F 的正则覆盖可以按图 8-9 中描述的那样计算。需要重视的一点是，当检验一个属性是否无关时，检验时用的是 F_c 当前值中的函数依赖，而不是 F 中的依赖。如果一个函数依赖的右半部只包含一个属性，例如 $A \rightarrow C$，并且这个属性是无关的，那么我们将得到一个右半部为空的函数依赖。这样的函数依赖应该删除。

```
F_c = F
    repeat
        使用合并律将 F_c 中所有形如 α1→β1 和 α1→β2 的依赖
            替换为 α1→β1β2
        在 F_c 中寻找一个函数依赖 α→β，它在 α 或在 β 中
            具有一个无关属性
            /* 注意，使用 F_c 而非 F 检验无关属性 */
        如果找到一个无关属性，则将它从 F_c 中的 α→β 中删除
    until (F_c 不变)
```

图 8-9 计算正则覆盖

可以证明 F 的正则覆盖 F_c 与 F 具有相同的闭包；因此，验证是否满足 F_c 等价于验证是否满足 F。但是，从某种意义上说，F_c 是最小的——它不含无关属性，并且它合并了具有相同左半部的函数依赖。所以验证 F_c 比验证 F 本身更容易。

考虑模式(A, B, C)上的如下函数依赖集 F：

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$

让我们来计算 F 的正则覆盖。

- 存在两个函数依赖在箭头左边具有相同的属性集：

$$A \rightarrow BC$$
$$A \rightarrow B$$

我们将这些函数依赖合并成 $A \rightarrow BC$。

- A 在 $AB \rightarrow C$ 中是无关的，因为 F 逻辑蕴涵$(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$。这个断言为真，因为 $B \rightarrow C$ 已经在函数依赖集中。
- C 在 $A \rightarrow BC$ 中是无关的，因为 $A \rightarrow BC$ 被 $A \rightarrow B$ 和 $B \rightarrow C$ 逻辑蕴涵。

于是，正则覆盖为：

$$A \rightarrow B$$
$$B \rightarrow C$$

给定函数依赖集 F，可能集合中有一整条函数依赖都是无关的，也就是说删掉它不改变 F 的闭包。我们可以证明 F 的正则覆盖 F_c 不包含这种无关的函数依赖。利用反证法，假设 F_c 中存在这种无关函数依赖，则依赖的右半部分属性将是无关的，这与正则覆盖的定义矛盾。

正则覆盖未必是唯一的。例如，考虑函数依赖集 $F = \{A \rightarrow BC, B \rightarrow AC, C \rightarrow AB\}$。如果我们对 $A \rightarrow BC$进行无关性检验，会发现在 F 下 B 和 C 都是无关的。然而，两个都删掉是不对的！计算正则覆盖的算法选择其中一个删除。那么，

342 ↓ 344

1. 如果删除 C，我们得到集合 $F' = \{A \rightarrow B, B \rightarrow AC, C \rightarrow AB\}$。现在，在 F'下，在 $A \rightarrow B$ 右半部 B 就不是无关的了。继续执行算法，我们发现 $C \rightarrow AB$ 右半部的 A 和 B 都是无关的，这导致两种正则覆盖

$$F_c = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}$$

2. 如果删除 B，我们得到集合$\{A \rightarrow C, B \rightarrow AC, C \rightarrow AB\}$。这种情况与前面的情况是对称的，将导致两种正则覆盖

$$F_c = \{A \rightarrow C, C \rightarrow B, B \rightarrow A\}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, C \rightarrow AB\}$$

作为练习，你能再找到一个 F 上的正则覆盖吗？

8.4.4　无损分解

令 $r(R)$为一个关系模式，F 为 $r(R)$上的函数依赖集。令 R_1 和 R_2 为 R 的分解。如果用两个关系模式 $r_1(R_1)$和 $r_2(R_2)$替代 $r(R)$时没有信息损失，则我们称该分解是**无损分解**(lossless decomposition)。更精确地说，我们称分解是无损的，如果对于所有的合法数据库实例(即满足指定的函数依赖和其他约束的数据库实例)，关系 r 包含与下述 SQL 查询结果相同的元组集：

```
select *
from (select R1 from r)
    natural join
    (select R2 from r)
```

这用关系代数可以更简洁地表示为：

$$\prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$$

换句话说，如果我们把 r 投影至 R_1 和 R_2 上，然后计算投影结果的自然连接，我们仍然得到一模一样的 r。不是无损分解的分解称为**有损分解**(lossy decomposition)。**无损连接分解**(lossless-join decomposition)和**有损连接分解**(lossy-join decomposition)这两个术语有时用来代替无损分解和有损分解。

345 作为有损连接的一个例子，回想 8.1.2 节中 *employee* 模式的分解：

$$employee1\ (ID,\ name)$$
$$employee2\ (name,\ street,\ city,\ salary)$$

如图 8-3 所示，*employee1* ⋈ *employee2* 的结果是原关系 *employee* 的一个超集，但是分解是有损的，因为当有两个或多个雇员具有相同的名字时，连接的结果丢失了关于哪个雇员标识和哪个地址及工资相关联的信息。

我们可以用函数依赖来说明什么情况下分解是无损的。令 R、R_1、R_2 和 F 如上。R_1 和 R_2 是 R 的无损分解，如果以下函数依赖中至少有一个属于 F^+：

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

换句话说，如果 $R_1 \cap R_2$ 是 R_1 或 R_2 的超码，R 上的分解就是无损分解。正如我们前面看到的，我们可以用属性闭包的方法来高效地检验超码。

为举例说明，考虑模式

$$inst\_dept\ (ID,\ name,\ salary,\ dept\_name,\ building,\ budget)$$

我们在 8.1.2 节中将其分解为 *instructor* 和 *department* 模式：

$$instructor\ (ID,\ name,\ dept\_name,\ salary)$$
$$department\ (dept\_name,\ building,\ budget)$$

考虑这两个模式的交集，即 *dept_name*。我们发现，由于 $dept\_name \rightarrow dept\_name,\ building,\ budget$，因此满足无损分解条件。

对于一个模式一次性分解成多个模式的情况，判定无损分解就更复杂了。参看文献注解的相关主题。

虽然对二元分解的测试显然是无损连接的一个充分条件，但只有当所有约束都是函数依赖时它才是必要条件。后面我们将看到几种其他类型的约束(特别是 8.6.1 节讨论的称为多值依赖的一种约束类型)，它们即使在不存在函数依赖的情况下仍可保证一个分解是无损连接的。

8.4.5 保持依赖

346 使用函数依赖理论描述保持依赖，要比我们在 8.3.3 节中采用的即席方法容易得多。

令 F 为模式 R 上的一个函数依赖集，R_1，R_2，…，R_n 为 R 的一个分解。F 在 R_i 上的**限定**(restriction)是 F^+ 中所有只包含 R_i 中属性的函数依赖的集合 F_i。由于一个限定中的所有函数依赖只涉及一个关系模式的属性，因此判定这种依赖是否满足可以只检查一个关系。

注意定义限定时用到的是 F^+ 中的所有函数依赖，而不仅仅是 F 中的。例如，假设 $F = \{A \rightarrow B,\ B \rightarrow C\}$，且我们有一个到 AC 和 AB 的分解。F 在 AC 上的限定则包含 $A \rightarrow C$，因为 $A \rightarrow C$ 属于 F^+，尽管它并没有在 F 中。

限定 F_1，F_2，…，F_n 的集合是能高效检查的依赖集。我们现在必须要问是否只检查这些限定就够了。令 $F' = F_1 \cup F_2 \cup \cdots \cup F_n$。$F'$是模式 R 上的一个函数依赖集，不过通常 $F' \neq F$。但是，即使 $F' \neq F$，也有可能 $F'^+ = F^+$。如果后者为真，则 F 中的所有依赖都被 F' 逻辑蕴涵，并且，如果我们证明 F' 是满足的，则我们就证明了 F 也满足。我们称具有性质 $F'^+ = F^+$ 的分解为**保持依赖的分解**(dependency-preserving decomposition)。

图 8-10 给出了判定保持依赖性的算法。输入是分解的关系模式集 $D = \{R_1, R_2, \cdots, R_n\}$ 和函数依赖集 F。因为要计算 F^+，所以这个算法的开销很大。我们将考虑另外两个方法来替代图 8-10 中的算法。

首先，请注意如果 F 中的每一个函数依赖都可以在分解得到的某一个关系上验证，那么这个分解就是保持依赖的。这是一种简单的验证保持依赖性的方法；但是，它并不总是有效。有些情况下，尽管这个分解是保持依赖的，但是在 F 中存在一个依赖，它无法在分解后的任意一个关系上验证。所以这个验证方法只能用作一个易于检查的充分条件，如果验证失败我们也不能断定该分解就不是保持依赖的，而是将不得不采用一般化的验证方法。

```
计算 F+ ;
    for each D 中的模式 Ri do
        begin
            Fi := F+ 在 Ri 中的限定;
        end
    F' := ∅
    for each 限定 Fi do
        begin
            F' = F' ∪ Fi
        end
    计算 F'+ ;
    if (F'+ = F+) then return (true)
                  else return (false);
```

图 8-10　保持依赖性的验证

我们现在给出第二种避免计算 F^+ 的验证保持依赖的方法。我们会在列出验证方法之后解释该方法的思想。该验证方法对 F 中的每一个 $\alpha \rightarrow \beta$ 使用下面的过程：

```
result = α
repeat
    for each 分解后的 Ri
        t = (result ∩ Ri)+ ∩ Ri
        result = result ∪ t
until (result 没有变化)
```

这里的属性闭包是函数依赖集 F 下的。如果 *result* 包含 β 中的所有属性，则函数依赖 $\alpha \rightarrow \beta$ 保持。分解是保持依赖的当且仅当上述过程中 F 的所有依赖都保持。

上述验证方法背后的两个关键的思想如下：

- 第一个思想是验证 F 中的每个函数依赖 $\alpha \rightarrow \beta$，看它是否在 F' 中保持（F' 的定义如图8-10所示）。为此，我们计算 F' 下 α 的闭包；当该闭包包含 β 时，该依赖得以保持。该分解是保持依赖的当（且仅当）F 中的所有函数依赖都保持。
- 第二个思想是使用修改后的属性闭包算法计算 F' 下的闭包，不用先真正计算出 F'。我们希望避免 F' 的计算，因为计算开销很大。注意到 F' 是 F_i 的并集，而 F_i 是 F 在 R_i 上的限定。该算法计算出 F 上（*result* $\cap$ R_i）的属性闭包，并与 R_i 的闭包求交，然后将结果属性集加入 *result*；这一系列步骤等价于计算 F_i 下的 *result* 闭包。在 while 循环中对每个 i 重复该步骤就得到了 F' 下的 *result* 闭包。

为了理解这个修改后的属性闭包算法运算正确的原因，我们注意到对于每个 $\gamma \subseteq R_i$，$\gamma \rightarrow \gamma^+$ 是 F^+ 中的一个函数依赖，并且 $\gamma \rightarrow \gamma^+ \cap R_i$ 是 F^+ 在 R_i 上的限定 F_i 中的函数依赖。反之，如果 $\gamma \rightarrow \delta$ 出现在 F_i 中，则 δ 是 $\gamma^+ \cap R_i$ 的子集。

该判定方法的代价是多项式时间，而不是计算 F^+ 所需的指数时间的代价。

8.5　分解算法

现实世界的数据库模式要比能装在书页中的例子大得多。因此，我们需要能生成属于适当范式的设计的算法。本节给出 BCNF 和 3NF 的相应算法。

347
348

8.5.1　BCNF 分解

BCNF 的定义可以直接用于检查一个关系是否属于 BCNF。但是，计算 F^+ 是一个繁重的任务。下面首先描述判定一个关系是否属于 BCNF 的简化检验方法。如果一个关系不属于 BCNF，它可以被分解以创建属于 BCNF 的关系。本节后面将描述一种创建关系的无损分解的算法，使得该分解属于 BCNF。

8.5.1.1　BCNF 的判定方法

在某些情况下，判定一个关系是否属于 BCNF 可以作如下简化：

- 为了检查非平凡的函数依赖 $\alpha \rightarrow \beta$ 是否违反 BCNF，计算 α^+（α 的属性闭包），并且验证它是否包含 R 中的所有属性，即验证它是否是 R 的超码。
- 检查关系模式 R 是否属于 BCNF，仅须检查给定集合 F 中的函数依赖是否违反 BCNF 就足够了，不用检查 F^+ 中的所有函数依赖。

我们可以证明如果 F 中没有函数依赖违反 BCNF，那么 F^+ 中也不会有函数依赖违反 BCNF。

遗憾的是，当一个关系分解后，后一步过程就不再适用。也就是说，当我们判定 R 上的一个分解 R_i 是否违反 BCNF 时，只用 F 就不够了。例如，考虑关系模式 $R(A, B, C, D, E)$，其函数依赖集 F 包括 $A \rightarrow B$ 和 $BC \rightarrow D$。假设 R 分解成 $R_1(A, B)$ 和 $R_2(A, C, D, E)$。现在，因为 F 中没有一个函数依赖只包含来自 (A, C, D, E) 的属性，所以我们也许会误认为 R_2 满足 BCNF。实际上，F^+ 中有一个函数依赖 $AC \rightarrow D$（可以由伪传递律从 F 中的两个依赖推出），这表明 R_2 不属于 BCNF。所以，我们也许需要一个属于 F^+ 但不属于 F 的依赖，来证明一个分解后的关系不属于 BCNF。

BCNF 的另一种判定方法有时会比计算 F^+ 上的所有函数依赖简单。为了检查 R 分解后的关系 R_i 是否属于 BCNF，我们应用如下判定：

- 对于 R_i 中属性的每个子集 α，确保 α^+（F 下 α 的属性闭包）要么不包含 $R_i - \alpha$ 的任何属性，要么包含 R_i 的所有属性。

如果 R_i 上有某个属性集 α 违反该条件，考虑如下的函数依赖，可以证明它出现在 F^+ 中：

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$

上面这个函数依赖说明 R_i 违反 BCNF。

8.5.1.2 BCNF 分解算法

我们现在能给出一个关系模式分解的一般方法以满足 BCNF。图 8-11 所示为实现该过程的一个算法。若 R 不属于 BCNF，则可用这个算法将 R 分解成一组 BCNF 模式 R_1，R_2，…，R_n。这个算法利用违反 BCNF 的依赖进行分解。

```
result := {R};
    done := false;
    计算 F+;
    while (not done) do
        if (result 中存在模式 Ri 不属于 BCNF)
        then begin
            令 α → β 为一个在 Ri 上成立的非平凡函数依赖，满足 α → Ri 不属于 F+，并且 α ∩ β = ∅;
            result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
    end
    else done := true;
```

图 8-11 BCNF 分解算法

用这个算法产生的分解不仅是一个 BCNF 分解，而且是一个无损分解。来看一下为什么该算法只产生无损分解，我们注意到，当我们用 $(R_i - \beta)$ 和 (α, β) 取代模式 R_i 时，依赖 $\alpha \rightarrow \beta$ 成立，而且 $(R_i - \beta) \cap (\alpha, \beta) = \alpha$。

如果我们没有要求 $\alpha \cap \beta = \emptyset$，那么 $\alpha \cap \beta$ 中的那些属性就不会出现在模式 $(R_i - \beta)$ 中，而依赖 $\alpha \rightarrow \beta$ 也不再成立。

容易看出在 8.3.2 节中对 *inst_dept* 的分解结果可以通过应用该算法获得。函数依赖 *dept_name* → *building*，*budget* 满足 $\alpha \cap \beta = \emptyset$ 条件，因此被选择用来分解该模式。

BCNF 分解算法所需时间为原始模式规模的指数级别，因为该算法检查分解中的一个关系是否满足 BCNF 的代价可以达到指数级。文献注解提供了一个算法的参考文献，该算法能够在多项式时间内计算 BCNF 分解。但是，这个算法可能“过于规范化”，即，分解不必要分解的关系。

349
|
350

举一个更长的使用 BCNF 分解算法的例子，假定我们有一个数据库设计使用了以下的 *class* 模式：

class (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

我们要求在 *class* 上成立的函数依赖集合为

$$course\_id \rightarrow title, dept\_name, credits$$
$$building, room\_number \rightarrow capacity$$
$$course\_id, sec\_id, semester, year \rightarrow building, room\_number, time\_slot\_id$$

该模式的一个候选码是 $\{course\_id, sec\_id, semester, year\}$。

我们可以按如下方式将图 8-11 的算法用于 *class* 例子：

- 函数依赖：

$$course\_id \rightarrow title, dept\_name, credits$$

成立，但 *course_id* 不是超码。因此，*class* 不属于 BCNF。我们将 *class* 替换为：

$$course\ (course\_id, title, dept\_name, credits)$$
$$class-1\ (course\_id, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)$$

course 上成立的唯一一个非平凡函数依赖的箭头左侧包含 *course_id*。由于 *course_id* 是 *course* 的码，因此关系 *course* 属于 BCNF。

- *class* - 1 的一个候选码为 $\{course\_id, sec\_id, semester, year\}$，函数依赖：

$$building, room\_number \rightarrow capacity$$

在 *class* - 1 上成立，但 $\{building, room\_number\}$ 不是 *class* - 1 的超码。我们将 *class* - 1 替换为：

$$classroom\ (building, room\_number, capacity)$$
$$section\ (course\_id, sec\_id, semester, year, building, room\_number, time\_slot\_id)$$

classroom 和 *section* 属于 BCNF。

于是，*class* 分解为三个关系模式 *course*、*classroom* 和 *section*，它们都属于 BCNF。这些模式对应于我们在本章和前面章节使用的模式。你可以验证该分解既是无损的，又是保持依赖的。

8.5.2　3NF 分解

图 8-12 给出了将模式转化为 3NF 的保持依赖且无损的分解的算法。该算法中使用的依赖集 F_c 是 F 的正则覆盖。注意，该算法考虑的是模式 R_j 的集合（$j = 1, 2, \cdots, i$）；初始 $i = 0$ 且该集合为空。

让我们将该算法用于 8.3.4 节中的例子，我们曾证明

$$dept\_advisor\ (s\_ID, i\_ID, dept\_name)$$

是属于3NF的，虽然它不属于 BCNF。该算法使用 F 中以下的函数依赖：

$$f_1: i\_ID \rightarrow dept\_name$$
$$f_2: s\_ID, dept\_name \rightarrow i\_ID$$

在 F 的任意一个函数依赖中都不存在无关属性，因此 F_c 包含 f_1 和 f_2。该算法生成模式（*i_ID*, *dept_name*）作为 R_1，以及模式（*s_ID*, *dept_name*, *i_ID*）作为 R_2。该算法随即发现 R_2 包含一个候选码，因此不再继续创建关系模式。

```
令 F_c 为 F 的正则覆盖；
i := 0；
for each F_c 中的函数依赖 α → β
    i := i + 1；
    R_i := αβ；
if 模式 R_j, j = 1, 2, . . . , i 都不包含 R 的候选码
then
    i := i + 1；
    R_i := R 的任意候选码；
/*（可选）移除冗余关系 */
repeat
    if 模式 R_j 包含于另一个模式 R_k 中
    then
    /* 删除 R_j */
    R_j := R_i；
    i := i - 1；
until 不再有可以删除的 R_j
return(R_1, R_2, ···, R_i)
```

图 8-12　到 3NF 的保持依赖且无损的分解

生成的模式集可能会包含冗余的模式，即一个模式 R_k 包含另一个模式 R_j 所有的属性。例如，上面的 R_2 包含 R_1 所有的属性。这个算法会删除所有这种包含于其他模式的模式。在删除的 R_j 上验证的任意一个依赖在相应的关系 R_k 上也验证，即使删除了 R_j，分解仍旧是无损的。

351 ↓ 352

现在我们再次考虑 8.5.1.2 节中的 *class* 模式，并运用 3NF 分解算法。我们所列出的函数依赖集恰

好是正则覆盖。因此，该算法给出同样的三个模式 *course*、*classroom* 和 *section*。

上例说明了3NF算法的一个有趣的特性。有时，结果不仅属于3NF，而且也属于BCNF。这暗示了生成BCNF设计的另一个方法。首先使用3NF算法，然后对于3NF设计中任何一个不属于BCNF的模式，用BCNF算法分解。如果结果不是保持依赖的，则还原到3NF设计。

8.5.3 3NF算法的正确性

3NF算法通过为正则覆盖中的每个依赖显式地构造一个模式确保依赖的保持。该算法通过保证至少有一个模式包含被分解模式的候选码，确保该分解是一个无损分解。实践习题8.14深入审视这种算法足以保证无损分解的证据。

这个算法也称为**3NF合成算法**(3NF synthesis algorithm)，因为它接受一个依赖集合，每次添加一个模式，而不是对初始的模式反复地分解。该算法的结果不是唯一确定的，因为一个函数依赖集有不止一个正则覆盖，而且，某些情况下该算法的结果依赖于该算法考虑 F_c 中的依赖的顺序。该算法有可能分解一个已经属于3NF的关系；不过，仍然保证分解是属于3NF的。

如果一个关系 R_i 在由该合成算法产生的分解中，则 R_i 属于3NF。回想当我们判定3NF时，考虑右半部是单个属性的函数依赖就足够了。因此，要看 R_i 是否属于3NF，你必须确认 R_i 上的任意函数依赖 $\gamma \rightarrow B$ 满足3NF的定义。假定该合成算法中产生 R_i 的函数依赖是 $\alpha \rightarrow \beta$。现在，因为 B 属于 R_i，而且 $\alpha \rightarrow \beta$ 产生 R_i，所以 B 一定属于 α 或 β。让我们考虑以下三种可能情况：

- B 既属于 α 又属于 β。在这种情况下，依赖 $\alpha \rightarrow \beta$ 将不可能属于 F_c，否则 B 在 β 中将是无关的。所以这种情况不成立。
- B 属于 β 但不属于 α。考虑两种情况：
  - □ γ 是一个超码。满足3NF的第二个条件。
  - □ γ 不是超码。则 α 必定包含某些不在 γ 中的属性。现在，由于 $\gamma \rightarrow B$ 在 F^+ 中，因此它一定是通过使用 γ 上的属性闭包算法从 F_c 推导出来的。该推导不可能用到 $\alpha \rightarrow \beta$，否则 α 一定包含在 γ 的属性闭包里，而这是不可能的，因为我们假定 γ 不是超码。现在，使用 $\alpha \rightarrow (\beta - \{B\})$ 和 $\gamma \rightarrow B$，我们可以推导出 $\alpha \rightarrow B$(由于 $\gamma \subseteq \alpha\beta$；并且因为 $\gamma \rightarrow B$ 是非平凡的，因此 γ 不包含 B)。这表明 B 在 $\alpha \rightarrow \beta$ 的右半部是无关的，这也是不可能的，因为 $\alpha \rightarrow \beta$ 属于正则覆盖 F_c。所以，如果 B 属于 β，那么 γ 一定是超码并且满足3NF的第二个条件。
- B 属于 α 但不属于 β。

因为 α 是候选码，所以满足3NF定义中的第三个条件。

有趣的是，我们描述的3NF分解算法可以在多项式时间内实现，尽管判定给定关系是否属于3NF是NP-hard的(这意味着设计一个多项式时间的算法来解决该问题是不太可能的)。

8.5.4 BCNF和3NF的比较

对于关系数据库模式的两种范式——3NF和BCNF，3NF的一个优点是我们总可以在满足无损并保持依赖的前提下得到3NF设计。然而3NF也有一个缺点：我们可能不得不用空值表示数据项间的某些可能有意义的联系，并且存在信息重复的问题。

我们对应用函数依赖进行数据库设计的目标是：

1. BCNF。
2. 无损。
3. 保持依赖。

由于不是总能达到所有这三个目标，因此我们也许不得不在BCNF和保持依赖的3NF中做出选择。

值得注意的是，除了可以通过用主码或者唯一约束来声明超码的特殊情况外，SQL不提供指定函数依赖的途径。通过写断言来保证函数依赖(参见实践习题8.9)虽然有点麻烦，但也是有可能的。遗憾的是，目前没有数据库系统支持强制实施函数依赖所需的复杂断言，而且检查这些断言的开销很大。所以即使我们有一个保持依赖的分解，但如果我们使用标准SQL，我们只能对那些左半部是码的函数依赖进行高效的检查。

353

虽然在分解不能保持依赖时，对函数依赖的检查可能会用到连接，但倘若数据库系统支持物化视图上的主码约束，我们就可以通过使用物化视图的方法来降低开销，这是许多数据库系统都支持的。给定一个非保持依赖的 BCNF 分解，我们考虑正则覆盖 F_c 里每一个在分解中未保持的依赖。对于每一个这样的依赖 $\alpha \rightarrow \beta$，我们定义一个物化视图对分解中所有的关系计算连接，并且将结果投影到 $\alpha\beta$ 354 上。利用一个约束 **unique**(α)或者 **primary key**(α)，就可以在物化视图上很容易地检查函数依赖。

从负面来看，物化视图会带来一些时间和空间的额外开销；但是从正面来看，应用程序员不需要写代码来保持冗余数据在更新上的一致性。维护物化视图是数据库系统的工作，即在数据库更新时做出相应的更新。(本书后面的 13.5 节对数据库系统如何高效地进行物化视图的维护进行了概述。)

遗憾的是，目前绝大多数数据库系统不支持物化视图上的约束。虽然 Oracle 数据库支持物化视图上的约束，但它默认当访问视图时进行视图维护，而不是更新底层关系时；[⊖]结果是，约束冲突有可能在更新已经执行之后才检测出来，这使得该检测没有用。

因此，在不能得到保持依赖的 BCNF 分解的情况下，通常我们倾向于选择 BCNF，因为在 SQL 中检查非主码约束的函数依赖很困难。

8.6　使用多值依赖的分解

有些关系模式虽然属于 BCNF，但从某种意义上说仍存在信息重复的问题，所以看起来没有充分规范化。考虑大学的例子，一个教师可以和多个系相关联。

$$inst\ (ID,\ dept\_name,\ name,\ street,\ city)$$

敏锐的读者将发现这个模式是一个非 BCNF 模式，因为有函数依赖

$$ID \rightarrow name,\ street,\ city$$

并且 *ID* 不是 *inst* 的码。

进一步假设一个教师可以有多个地址(比如说，一个冬天的家和一个夏天的家)。那么，我们不再想强制实施函数依赖"$ID \rightarrow street,\ city$"，不过当然，我们仍想强制实施"$ID \rightarrow name$"(即大学不处理有多个别名的教师的情况)。根据 BCNF 分解算法，得到两个模式： 355

$$r_1(ID,\ name)$$
$$r_2(ID,\ dept\_name,\ street,\ city)$$

这两个模式都属于 BCNF(回想到一个教师可以和多个系关联，并且一个系可以有多名教师，因此"$ID \rightarrow dept\_name$"和"$dept\_name \rightarrow ID$"都不成立)。

尽管 r_2 属于 BCNF，但是冗余仍然存在。对于一名教师所关联的每个系都要将该教师的每一个居住地址信息重复一次。为了解决这个问题，可以把 r_2 进一步分解为：

$$r_{21}(dept\_name,\ ID)$$
$$r_{22}(ID,\ street,\ city)$$

但是，并不存在任何约束来引导我们进行这个分解。

为处理这个问题，我们必须定义一种新的约束形式，称为**多值依赖**。正如我们对函数依赖所做的一样，我们将利用多值依赖定义关系模式的一种范式。这种范式称为**第四范式**(Fourth Normal Form，4NF)，它比 BCNF 的约束更严格。我们将看到每个 4NF 模式也都属于 BCNF，但有些 BCNF 模式不属于 4NF。

8.6.1　多值依赖

函数依赖规定了某些元组不能出现在关系中。如果 $A \rightarrow B$ 成立，我们就不能有两个元组在 A 上的值相同而在 B 上的值不同。从另一个角度出发，多值依赖并不排除某些元组的存在，而是要求某种形式的其他元组存在于关系中。由于这个原因，函数依赖有时称为**相等产生依赖**(equality-generating dependency)，而多值依赖称为**元组产生依赖**(tuple-generating dependency)。

⊖ 至少到 Oracle 10g 版本如此。

令 $r(R)$ 为一关系模式，并令 $\alpha \subseteq R$ 且 $\beta \subseteq R$。**多值依赖**(multivalued dependency)

$$\alpha \twoheadrightarrow \beta$$

在 R 上成立的条件是，在关系 $r(R)$ 的任意合法实例中，对于 r 中任意一对满足 $t_1[\alpha] = t_2[\alpha]$ 的元组对 t_1 和 t_2，r 中都存在元组 t_3 和 t_4，使得

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

356

这个定义看似复杂，实则不然。图 8-13 所示为 t_1、t_2、t_3 和 t_4 的表格图。直观地，多值依赖 $\alpha \twoheadrightarrow \beta$ 是说 α 和 β 之间的联系独立于 α 和 $R - \beta$ 之间的联系。若模式 R 上的所有关系都满足多值依赖 $\alpha \twoheadrightarrow \beta$，则 $\alpha \twoheadrightarrow \beta$ 是模式 R 上平凡的多值依赖。因此，如果 $\beta \subseteq \alpha$ 或 $\beta \cup \alpha = R$，则 $\alpha \twoheadrightarrow \beta$ 是平凡的。

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

图 8-13 $\alpha \twoheadrightarrow \beta$ 的表格表示

为说明函数依赖和多值依赖的区别，我们再次考虑模式 r_2 及图 8-14 中所示的该模式上的一个关系的例子。我们必须为教师的每个地址重复一遍系名，并且必须为教师关联的每个系重复地址。这种重复是不必要的，因为教师与其地址间的联系独立于教师与系间的联系。如果一个 *ID* 为 22222 的教师与物理系关联，我们希望这个系与该教师的所有地址都关联。因此，图 8-15 的关系是非法的。要使该关系合法化，需要向图 8-15 的关系中加入元组(Physics，22222，Main，Manchester)及(Math，22222，North，Rye)。

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck

图 8-14 一个 BCNF 模式上的关系中有冗余的例子

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Math	Main	Manchester

图 8-15 一个非法的 r_2 关系

将前面这个例子与多值依赖的定义相比较，我们发现我们需要多值依赖

$$ID \twoheadrightarrow street, city$$

成立。(多值依赖 $ID \twoheadrightarrow dept\_name$ 也可以。我们不久将会看到它们是等价的。)

与函数依赖相同，我们将以两种方式使用多值依赖：

1. 检验关系以确定它们在给定的函数依赖集和多值依赖集下是否合法。
2. 在合法关系集上指定约束；因此我们将只考虑满足给定函数依赖集和多值依赖集的关系。

请注意，若关系 r 不满足给定多值依赖，我们可以通过向 r 中增加元组构造一个确实满足多值依赖的关系 r'。

令 D 表示函数依赖和多值依赖的集合。D 的**闭包** D^+ 是由 D 逻辑蕴涵的所有函数依赖和多值依赖的集合。与函数依赖相同，我们可以用函数依赖和多值依赖的规范定义根据 D 计算出 D^+。我们可以用这样的推理处理非常简单的多值依赖。幸运的是，实际问题中遇到的多值依赖都很简单。对于复杂的依赖，用推理规则系统来推导出依赖集会更好。

由多值依赖的定义，我们可以得出以下规则，对于 α，$\beta \subseteq R$：

- 若 $\alpha \rightarrow \beta$，则 $\alpha \twoheadrightarrow \beta$。换句话说，每一个函数依赖也是一个多值依赖。
- 若 $\alpha \twoheadrightarrow \beta$，则 $\alpha \twoheadrightarrow R - \alpha - \beta$。

附录 C. 1. 1 列出了多值依赖的一个推理规则系统。

8. 6. 2　第四范式

再次考虑 BCNF 模式的例子

$$r_2(ID, dept\_name, street, city)$$

其中多值依赖"$ID \twoheadrightarrow street, city$"成立。我们在 8. 6 节的开头看到，尽管该模式属于 BCNF，但是这个设计并不理想，因为我们必须为每个系重复教师的地址信息。我们将看到，我们可以用给定的多值依赖改进数据库的设计，将该模式分解为第四范式。

函数依赖和多值依赖集为 D 的关系模式 $r(R)$ 属于**第四范式**(4NF)的条件是，对 D^+ 中所有形如 $\alpha \twoheadrightarrow \beta$的多值依赖(其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$)，至少有以下之一成立： 357 / 358

- $\alpha \twoheadrightarrow \beta$ 是一个平凡的多值依赖。
- α 是 R 的一个超码。

数据库设计属于 4NF 的条件是构成该设计的关系模式集中的每个模式都属于 4NF。

请注意，4NF 定义与 BCNF 定义的唯一不同在于多值依赖的使用。4NF 模式一定属于 BCNF。为了了解这点，我们注意到，如果模式 $r(R)$不属于 BCNF，则 R 上存在非平凡的函数依赖 $\alpha \rightarrow \beta$，其中 α 不是超码。由于 $\alpha \rightarrow \beta$ 就意味着 $\alpha \twoheadrightarrow \beta$，因此 $r(R)$不属于 4NF。

令 $r(R)$为关系模式，$r_1(R_1)$，$r_2(R_2)$，…，$r_n(R_n)$为 $r(R)$的分解。要检验分解中的每一个关系模式 r_i 是否属于 4NF，我们需要找到每一个 r_i 上成立的多值依赖。回想到对于函数依赖集 F，F 在 R_i 上的限定 F_i 是 F^+ 中所有只含 R_i 中属性的函数依赖。现在考虑函数依赖和多值依赖的集合 D。D 在 R_i 上的限定是集合 D_i，它包含

1. D^+ 中所有只含 R_i 中属性的函数依赖。
2. 所有形如：

$$\alpha \twoheadrightarrow \beta \cap R_i$$

的多值依赖，其中 $\alpha \subseteq R_i$ 且 $\alpha \twoheadrightarrow \beta$ 属于 D^+。

8. 6. 3　4NF 分解

我们可以将 4NF 与 BCNF 之间的相似之处应用到 4NF 模式分解中。图 8-16 给出了 4NF 分解算法。它与图 8-11 的 BCNF 分解算法相同，除了它使用多值依赖以及 D^+ 在 R_i 上的限定。

```
result := {R};
done := false;
计算 D+；给定模式 Ri，令 Di 表示 D+ 在 Ri 上的限定
while (not done) do
    if (result 中存在 Ri 不属于 4NF)
    then begin
        令 α↠β 为 Ri 上成立的非平凡多值依赖
        使得 α → Ri 不属于 Di，并且 α ∩ β = ∅;
        result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
end
else done := true;
```

图 8-16　4NF 分解算法

如果我们将图 8-16 的算法应用于$(ID, dept_name, street, city)$，我们发现 $ID \twoheadrightarrow dept\_name$ 是非平凡的多值依赖，并且 ID 不是该模式的超码。按照该算法，我们将它替换为两个模式：

$$r_{21}(ID, dept\_name)$$
$$r_{22}(ID, street, city)$$

这对模式属于 4NF，它消除了我们前面所遇到的冗余。

与单独考虑函数依赖时一样，我们希望得到无损的和保持依赖的分解。从下面关于多值依赖和无损性的性质可以看出图 8-16 的算法只产生无损分解： 359

- 令 $r(R)$为一关系模式，D 为 R 上的函数依赖和多值依赖的集合。令 $r_1(R_1)$和 $r_2(R_2)$为 R 的一个分解。该分解是 R 的无损分解，当且仅当下面的多值依赖中至少有一个属于 D^+：

$$R_1 \cap R_2 \rightarrow\!\!\!\rightarrow R_1$$
$$R_1 \cap R_2 \rightarrow\!\!\!\rightarrow R_2$$

回想我们在 8.4.4 节提到的，若 $R_1 \cap R_2 \rightarrow R_1$ 或 $R_1 \cap R_2 \rightarrow R_2$，则 $r_1(R_1)$和 $r_2(R_2)$为 $r(R)$的无损分解。上面这个关于多值依赖的性质是对无损性更一般化的表述。它指明，对于每个将 $r(R)$分解为两个模式 $r_1(R_1)$和 $r_2(R_2)$的无损分解，两个依赖 $R_1 \cap R_2 \rightarrow\!\!\!\rightarrow R_1$ 或 $R_1 \cap R_2 \rightarrow\!\!\!\rightarrow R_2$ 中至少有一个成立。

当我们对存在多值依赖的关系模式进行分解时，保持依赖的问题变得更加复杂。附录 C.1.2 将继续讨论这个问题。

8.7 更多的范式

第四范绝不是“最终”的范式。正如我们前面看到的，多值依赖有助于我们理解并消除某些形式的信息重复，而这种信息重复用函数依赖是无法理解的。还有一些类型的约束称作**连接依赖**(join dependency)，它概化了多值依赖，并引出了另一种范式称作**投影 - 连接范式**(Project-Join Normal Form，PJNF)(PJNF 在某些书中称为**第五范式**，fifth normal form)。还有一类更一般化的约束，它引
360 出一种称作**域 - 码范式**(Domain-Key Normal Form，DKNF)的范式。

使用这些一般化的约束的一个实际的问题是，它们不仅难以推导，而且也还没有形成一套具有正确有效性和完备性的推理规则用于约束的推导。因此 PJNF 和 DKNF 很少使用。附录 C 将更详细地介绍这些范式。

很明显，我们的讨论中没有**第二范式**(Second Normal Form，2NF)。因为它只具有历史的意义，所以我们没有对它进行讨论。在实践习题 8.17 中，我们对它简单地定义，并留做练习。

8.8 数据库设计过程

迄今为止我们详细讨论了范式和规范化的问题，本节将研究规范化是如何糅合在整体数据库设计过程中的。

在本章的前面，从 8.3 节开始，我们假定给定关系模式 $r(R)$，并对之进行规范化。我们可以采用以下几种方法得到模式 $r(R)$：

1. $r(R)$可以是由 E-R 图向关系模式集进行转换时生成的。

2. $r(R)$可以是一个包含所有有意义的属性的单个关系。然后规范化过程中将 R 分解成一些更小的模式。

3. $r(R)$可以是关系的即席设计的结果，然后我们检验它们是否满足一个期望的范式。

在本节的剩余的部分，我们将考察这些方法之间的潜在关系。我们也将考察数据库设计中一些实际的问题，包括为了保证性能的去规范化，以及规范化时没检测到的不良设计的例子。

8.8.1 E-R 模型和规范化

当我们小心地定义 E-R 图，并正确地识别所有的实体时，由 E-R 图生成的关系模式应该就不需要太多进一步的规范化。然而，一个实体的属性之间有可能存在函数依赖。例如，假设 *instructor* 实体集包含属性 *dept_name* 和 *dept_address*，并且存在函数依赖 *dept_name* → *dept_address*。那么我们就要规范化从 *instructor* 生成的关系。

大多数这种依赖的例子都是由不好的 E-R 图设计引起的。在上面的例子中，如果我们正确设计 E-R 图，我们将创建一个包含属性 *dept_address* 的 *department* 实体集以及 *instructor* 与 *department* 之间的
361 一个联系集。同样，包含两个实体集以上的联系集有可能会使产生的模式不属于所期望的范式。由于大部分的联系集都是二元的，因此这样的情况相对稀少。(事实上，某些变种形式的 E-R 图实际上很难或者不可能指定非二元的联系集。)

函数依赖有助于我们检测到不好的 E-R 设计。如果生成的关系模式不属于想要的范式，该问题可以在 E-R 图中解决。也就是说，规范化可以作为数据建模的一部分规范地进行。规范化既可以留给数据库设计者在 E-R 建模时靠直觉实现，也可以从 E-R 模型生成的关系模式上规范地进行，二者可任选其一。

细心的读者可能注意到，为了解释多值依赖和第四范式的必要性，我们不得不从一个不是由E-R设计导出的模式出发。事实上，创建 E-R 设计的过程倾向于产生 4NF 设计。如果一个多值依赖成立且不是被相应的函数依赖所隐含的，那么它通常由以下情况引起：

- 一个多对多的联系集。
- 实体集的一个多值属性。

对于多对多的联系集，每个相关的实体集都有自己的模式，并且存在一个额外的模式用于表示联系集。对于多值属性，会创建一个单独的模式，包含该属性以及实体集的主码（正如在实体集 *instructor* 中的 *phone_number* 属性的例子一样）。

关系数据库设计的泛关系方法从一个假设开始，它假定存在单个关系模式包含所有有意义的属性。这单个模式定义了用户和应用程序如何与数据库交互。

8.8.2　属性和联系的命名

数据库设计的一个期望的特性是**唯一角色假设**（unique-role assumption），这意味着每个属性名在数据库中只有唯一的含义。这使得我们不能使用同一个属性在不同的模式中表示不同的东西。例如，相反地，我们可能考虑使用属性 *number* 在模式 *instructor* 中表示电话号码，而在模式 *classroom* 中表示房间号。对模式 *instructor* 上的一个关系和模式 *classroom* 上的一个关系进行连接是毫无意义的。用户和应用程序开发人员必须小心操作以保证在每个场合使用了正确的 *number*，而对电话号码和房间号分别使用不同的属性名则能减少用户的错误。

虽然对不兼容的属性使用不同的名字是个好主意，但是如果不同关系中的属性有相同的含义，使用相同的名字可能就是个好主意了。因此我们对实体集 *instructor* 和 *student* 使用相同的属性名 "*name*"。如果不这样（即我们对教师和学生的名字用不同的命名规范），那么如果我们想通过创建一个实体集 *person* 来概化这两个实体集，我们就不得不重命名属性。因此，即使我们当前没有对 *instructor* 和 *student* 的概化，然而如果我们预见到这种可能性，最好还是在这两个实体集（和关系）中使用同样的名字。 362

尽管从技术上看，一个模式中的属性名的顺序无关紧要，然而习惯上把主码属性列在前面。这会使得查看默认输出（例如 **select** * 的输出）更加容易。

在大的数据库模式中，联系集（及其导出的模式）常常以相关实体集名称的拼接来命名，可能使用连字符或下划线连接。我们已经使用了几个这样的名字，例如 *inst_sec* 和 *student_sec*。我们使用名字 *teaches* 和 *takes* 而不使用更长的拼接名字。由于对于少数几个联系集，想起它们的相关实体集并不难，因而这是可以接受的。我们无法一直通过简单的拼接创建联系集的名字；例如，雇员之间的管理者或被管理联系，如果我们称之为 *employee_employee* 就没有什么意义。相似地，如果一对实体集间可能有多个联系集，联系集的名字就必须包含额外的部分以区别这些联系集。

不同的组织对于命名实体集有不同的习惯。举例来说，我们可能称一个学生的实体集为 *student* 或者 *students*。在数据库设计中我们采用了单数形式。使用单数或者复数形式都是可以接受的，只要所有实体集都一致地使用该习惯就可以了。

随着模式变得更大，联系集的数量不断增加，使用对属性、联系和实体一致的命名方式会使得数据库设计者和应用程序员更加轻松。

8.8.3　为了性能去规范化

有时候数据库设计者会选择包含冗余信息的模式，也就是说，没有规范化。对一些特定的应用，他们使用冗余来提高性能。不使用规范化模式的代价是用来保持冗余数据一致性的额外工作（以编码时间和执行时间计算）。

例如，假定每次访问一门课程时，所有的先修课都必须和课程信息一起显示。在规范化的模式中，需要连接 *course* 和 *prereq*。

一个不计算连接的方法是保存一个包含 *course* 和 *prereq* 的所有属性的关系。这使得显示"全部"课程信息更快。然而，对于每门先修课都要重复课程的信息，而且每当添加或删除先修课时应用程序就必须更新所有的副本。把一个规范化的模式变成非规范化的过程称为**去规范化**(denormalization)，设计者用它调整系统的性能以支持响应时间苛刻的操作。

现今许多数据库系统支持一种更好的方法，即使用规范化的模式，同时将 *course* 和 *prereq* 的连接作为物化视图额外存储。(回忆一下，物化视图是将结果存储在数据库中的视图，当它用到的关系更
363 新时也相应更新。)像去规范化一样，使用物化视图确实会有空间和时间上的开销；不过它也有优点，就是保持视图更新的工作是由数据库系统而不是应用程序员完成的。

8.8.4 其他设计问题

数据库设计中有一些方面不能用规范化描述，而它们也会导致不好的数据库设计。与时间或时间段有关的数据就存在这样的问题。这里我们给出一些例子；显然，这样的设计应该避免。

考虑一个大学数据库，我们想存储不同年份中每个系的教师总数。可以用关系 *total_inst*(*dept_name*, *year*, *size*)来存储所需要的信息。这个关系上的唯一的函数依赖是 *dept_name*, *year* → *size*，这个关系属于 BCNF。

另一个设计是使用多个关系，每个关系存储一个年份的系规模信息。假设我们感兴趣的年份为 2007、2008 和 2009；我们将得到形如 *total_inst_2007*, *total_inst_2008*, *total_inst_2009* 这样的关系，它们都建立在模式(*dept_name*, *size*)之上。因为每一个关系上的唯一的函数依赖是 *dept_name* → *size*，所以这些关系也属于 BCNF。

但是，这个设计显然是一个不好的主意——我们必须每年都创建一个新的关系，每年还不得不写新的查询从而把新的关系考虑进去。由于可能涉及很多关系，因此查询也将更加复杂。

然而还有一种方法可以表示同样的数据，即建立一个单独的关系 *dept_year*(*dept_name*, *total_inst_2007*, *total_inst_2008*, *total_inst_2009*)。这里唯一的函数依赖是从 *dept_name* 指向其他属性的依赖，该关系也属于 BCNF。这种设计也是一个不好的想法，因为它存在与前面类似的问题，就是每年我们都不得不修改关系模式并书写新的查询。由于可能涉及很多属性，因此查询也会变得更加复杂。

像 *dept_year* 关系中那样的表示方法，属性的每一个值一列，叫作**交叉表**(crosstab)。它们在电子数据表、报告和数据分析工具中广泛使用。虽然这些表示方法显示给用户看是很有用的，但是由于上面给出的原因，它们在数据库的设计中并不可取。如 5.6.1 节所述，为了显示方便，SQL 扩展已经提出了将数据从规范化的关系表示转换到交叉表的方法。

8.9 时态数据建模

假定我们在大学中保存的数据不仅包括每个教师的地址，还包括该大学所知道的所有以前的地址。我们可能提出诸如"找出在 1981 年居住在普林斯顿的所有教师"的查询。在该情况下，我们可能对于每个教师有多个地址。每个地址有一个关联的起始日期和终止日期，表示该教师居住在该地址
364 的时间。对于终止日期，一个特殊的值，比如空值或者像 9999 - 12 - 31 这样的相当远的未来的值，可以用来表示教师仍然居住在该地址。

一般来说，**时态数据**(temporal data)是具有关联的时间段的数据，在时间段之间数据**有效**(valid)[⊖]。我们使用数据的**快照**(snapshot)这个术语来表示一个特定时间点上该数据的值。这样，*course* 数据的一张快照给出了在一个特定时间点上的所有课程的(诸如名称和系等)所有属性的值。

由于某些原因，对时态数据建模是一个相当有挑战性的问题。例如，假定我们有一个 *instructor* 实体，我们希望它关联着一个随时间变化的地址。为了给一个地址加上时态信息，我们不得不建立一个

⊖ 有些其他时态数据模型会区分**有效时间**(valid time)和**事务时间**(transaction time)，后者记录该事实记录到数据库中的时间。为简单起见，我们忽略这样的细节。

多值属性，其中的每个值是一个组合值(包含一个地址和一个时间段)。除了随时间变化的属性值，实体本身可能也要关联一个有效时间。例如，一个学生实体可能有一个从入学日期到毕业(或者离校)日期的有效时间。联系也可能有关联的有效时间。例如，联系 *prereq* 可以记录该课程成为另一门课程的先修课的时间。这样我们就需要在属性值、实体集和联系集上加上有效时间段。在 E-R 图上增加这些细节会使其非常难于创建和理解。当前已经有多个扩展 E-R 表示法的提案，希望用简单的方式指明属性值或联系是随时间变化的，但是至今仍没有采用的标准。

当我们随时间监测数据值时，我们假定成立的函数依赖，如：

$$ID \rightarrow street, city$$

可能不再成立。而以下约束(用自然语言表述)则可能成立："对任意给定的时间 t，一个教师 *ID* 仅有一个 *street* 和 *city* 的值。"

$X \xrightarrow{\tau} Y$ 在某个特定时间点上成立的函数依赖称为时态函数依赖。形式化地说，**时态函数依赖**(temporal functional dependency) $X \xrightarrow{\tau} Y$ 在关系模式 $r(R)$ 上成立的条件是，对于 $r(R)$ 的所有合法实例，r 的所有快照都满足函数依赖 $X \rightarrow Y$。

我们可以扩展关系数据库设计的理论，把时态函数依赖考虑进去。但是，使用常规的函数依赖已经很难于推理了，而且几乎没有设计者做好了处理时态函数依赖的准备。

在实践中，数据库设计者退回到更简单的方法来设计时态数据库。一个常用的方法是设计整个数据库(包括 E-R 设计和关系设计)而忽略时态改变(等价于仅考虑一张快照)。在这之后，设计者研究多 365
个关系并决定哪些关系需要跟踪时态变化。

接下来的步骤是通过把起始和终止时间作为属性，为每个这样的关系添加有效时间信息。例如，考虑 *course* 关系，课程的名称可能随时间变化，这可以通过添加一个有效时间范围来处理；得到的模式为

$$course\ (course\_id, title, dept\_name, start, end)$$

该关系的一个实例可能有两条记录(CS－101，"Introduction to Programming"，1985－01－01，2000－12－31)和(CS－101，"Introduction to C"，2001－01－01，9999－12－31)。如果更新该关系，将该课程名称改为"Introduction to Java"，时间"9999－12－31"就应该更新为原来的值("Introduction to C")有效的终止时间；然后加入包含新名称"Introduction to Java"和相应的起始时间的一条新元组。

如果另一个关系有一个参照时态关系的外码，数据库设计者必须决定参照是针对当前版本的数据还是一个特定时间点的数据。例如，我们可以通过扩展 *department* 关系来记录系的办公楼和预算随时间的变化，但是一个来自 *instructor* 或 *student* 关系的参照可能并不关心以往的办公楼和预算，而是可能隐含地参照对应 *dept_name* 的时态当前记录。另一方面，一个学生的成绩单记录应当参照该学生修习该课程期间的课程名称。在后一种情况下，参照关系必须也记录时间信息，以便于从 *course* 关系中确定一条特定的记录。在我们的例子中，选课时的 *year* 和 *semester* 可以映射到一个代表性的时间/日期值，比如该学期开学日的午夜；该时间/日期值用于从时态 *course* 关系中识别一条特定的记录，从中查询 *title*。

一个时态关系中原始的主码无法再唯一地标识一条元组。为了解决这个问题，我们可以在主码中加入起始和终止时间属性。但是，仍然留下一些问题：

- 有可能存储时间段重叠的数据，该问题主码约束无法捕获。如果系统支持自带的有效时间类型，它就能够检测并避免这种重叠的时间段。
- 为了指明参照这种关系的外码，参照元组必须包含起始和终止时间属性为它们的外码的一部分，其值也必须与被参照元组相匹配。进一步说，如果被参照元组更新(而且原本在未来的终止时间也更新)，则该更新必须传播到所有参照元组。

  如果系统以一种更好的方式支持时态数据，我们就能够允许参照元组指定一个时间点而不是一个时间范围，并且依靠系统保证在被参照关系中存在一条元组，该元组的有效时间段包含该时间点。例如，一条成绩单记录可以指定一个 *course_id* 和一个时间(比如一个学期的开学日 366
  期)，这样就足够在 *course* 关系中识别正确的记录了。

作为一个常见的特例，如果所有对时态数据的参照都仅仅指向当前数据，更简单的解决办法是不在关系中添加时间信息，而是为过去的值创建一个相应的有时态信息的 *history* 关系。例如，在我们的大学数据库中，我们可以使用已经创建的设计，忽略时态变化，仅仅存储当前信息。所有的历史信息都转移到历史关系中。这样，*instructor* 关系可以只存储当前地址，而关系 *instructor_history* 可以包含 *instructor* 所有的属性，以及额外的 *start_time* 和 *end_time* 属性。

尽管我们没有提供任何处理时态数据的形式化方法，我们所讨论的问题和我们所给出的例子会帮助你设计记录时态数据的数据库。处理时态数据中的进一步的问题，包括时态查询，稍后将在 25.2 节讲述。

8.10 总结

- 我们给出了数据库设计中易犯的错误，和怎样系统地设计数据库模式来避免这些错误。这些错误包括信息重复和不能表示某些信息。
- 我们给出了从 E-R 设计到关系数据库设计的发展过程，什么时候可以安全地合并模式，什么时候应该分解模式。所有有效的分解都必须是无损的。
- 我们描述了原子域和第一范式的假设。
- 我们介绍了函数依赖的概念，并且使用它介绍两种范式，Boyce-Codd 范式(BCNF)和第三范式(3NF)。
- 如果分解是保持依赖的，则给定一个数据库更新，所有的函数依赖都可以由单独的关系进行验证，无须计算分解后的关系的连接。
- 我们展示了如何用函数依赖进行推导。我们着重讲了什么依赖是一个依赖集逻辑蕴涵的。我们还定义了正则覆盖的概念，它是与给定函数依赖集等价的最小的函数依赖集。
- 我们概述了一个将关系分解成 BCNF 的算法，有一些关系不存在保持依赖的 BCNF 分解。
- 我们用正则覆盖将关系分解成 3NF，它比 BCNF 的条件弱一些。属于 3NF 的关系也许会含有冗余，但是总存在保持依赖的 3NF 分解。

367

- 我们介绍了多值依赖的概念，它指明仅用函数依赖无法指明的约束。我们用多值依赖定义了第四范式(4NF)。附录 C.1.1 给出了有关多值依赖推理的细节。
- 其他的范式，例如 PJNF 和 DKNF，消除了更多细微形式的冗余。但是，它们难以操作而且很少使用。附录 C 给出了这些范式的详细信息。
- 回顾本章中的要点可以发现，我们之所以可以对关系数据库设计定义严格的方法，是因为关系数据模型建立在严谨的数学基础之上。这是关系模型与我们已经学过的其他数据模型相比的主要优势之一。

术语回顾

- E-R 模型和规范化
- 分解
- 函数依赖
- 无损分解
- 原子域
- 第一范式(1NF)
- 合法关系
- 超码
- R 满足 F
- F 在 R 上成立
- Boyce-Codd 范式(BCNF)
- 保持依赖
- 第三范式(3NF)
- 平凡的函数依赖
- 函数依赖集的闭包
- Armstrong 公理
- 属性集闭包
- F 在 R_i 上的限定
- 正则覆盖
- 无关属性
- BCNF 分解算法
- 3NF 分解算法
- 多值依赖
- 第四范式(4NF)
- 多值依赖的限定
- 投影 - 连接范式(PJNF)
- 域 - 码范式(DKNF)
- 泛关系
- 唯一角色假设
- 去规范化

实践习题

8.1 假设我们将模式 $r(A, B, C, D, E)$ 分解为

368

$$r_1(A, B, C)$$
$$r_2(A, D, E)$$

证明该分解是无损分解，如果如下函数依赖集 F 成立：

$$A \rightarrow BC \qquad CD \rightarrow E \qquad B \rightarrow D \qquad E \rightarrow A$$

8.2　列出图 8-17 所示关系满足的所有函数依赖。

8.3　解释如何用函数依赖表明：

- 实体集 *student* 和 *instructor* 间存在的一对一联系集。
- 实体集 *student* 和 *instructor* 间存在的多对一联系集。

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

图 8-17　实践习题 8.2 的关系

8.4　用 Armstrong 公理证明合并律的正确有效性。（提示：使用增补律可证，若 $\alpha \rightarrow \beta$，则 $\alpha \rightarrow \alpha\beta$。再次使用增补律，利用 $\alpha \rightarrow \gamma$，然后使用传递律。）

8.5　用 Armstrong 公理证明伪传递律的正确有效性。

8.6　计算关于关系模式 $r(A, B, C, D, E)$ 的如下函数依赖集 F 的闭包。

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

列出 R 的候选码。

8.7　用习题 8.6 中的函数依赖计算正则覆盖 F_c。 369

8.8　考虑图 8-18 中计算 α^+ 的算法。证明该算法比图 8-8(8.4.2 节)中提出的算法更高效，并且能正确计算 α^+。

```
result: = ∅;
/* fdcount 是一个数组，它的第 i 个元素包含即将属于 α + 的第 i 个函数依赖左半部的属性个数 */
for i := 1 to | F | do
    begin
        令 β → γ 表示第 i 个函数依赖;
        fdcount [i] : = | β | ;
    end
/* appears 是一个数组，每个属性对应数组的一个入口。属性 A 的入口是一列整数，该列中每个整数 i 指明 A 出现在
   第 i 个函数依赖的左半部 */
for each 属性 A do
    begin
        appears [A] := NIL;
        for i := 1 to | F | do
            begin
                令 β → γ 表示第 i 个函数依赖;
                if A ∈ β then 把 i 加到 appears [A] 中;
            end
    end
addin (α);
return (result);

procedure addin (α);
for each α 中的属性 A do
    begin
        if A ∉ result then
            begin
                result := result ∪ {A};
                for each appears[A] 的元素 i do
                    begin
                        fdcount [i] := fdcount [i] - 1;
                        if fdcount [i]: = 0 then
                            begin
                                令 β → γ 表示第 i 个函数依赖;
                                addin (γ);
                            end
                    end
            end
    end
```

370 ~ 371

图 8-18　一个计算 α^+ 的算法

8.9 给定数据库模式 $R(a, b, c)$ 及模式 R 上的关系 r，写出检验函数依赖 $b \rightarrow c$ 是否在关系 r 上成立的 SQL 查询。并写出保证函数依赖的 SQL 断言。假设不存在空值。（虽然 SQL 标准中的某些部分，诸如断言等目前还没有在任何数据库实现中得到支持。）

8.10 我们对无损连接分解的讨论隐含假设了函数依赖左部的属性不能为空值。如果违反了这个性质，在分解中可能会出现什么错误？

8.11 在 BCNF 分解算法中，假设你用一个函数依赖 $\alpha \rightarrow \beta$ 将关系模式 $r(\alpha, \beta, \gamma)$ 分解为 $r_1(\alpha, \beta)$ 和 $r_2(\alpha, \gamma)$。

a. 你预期在分解后的关系上有什么样的主码和外码约束成立？

b. 如果在上述分解后的关系上没有强制实施外码约束，给出一个由于错误更新而导致不一致的例子。

c. 当用 8.5.2 节中的算法将一个关系分解为 3NF 时，你预期将有什么样的主码和外码依赖在分解后的模式上成立？

8.12 令 R_1，R_2，…，R_n 为模式 U 的分解。令 $u(U)$ 为一个关系，并且令 $r_i = \prod_{R_i}(u)$。

证明

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

8.13 证明实践习题 8.1 中的分解不是保持依赖的分解。

8.14 证明有可能通过保证至少有一个模式包含被分解模式的候选码来确保一个保持依赖的 3NF 模式分解是一个无损分解。（提示：证明到分解后模式上的所有的投影的连接不会具有比原关系多的元组。）

8.15 给出一个关系模式 R' 及函数依赖集 F' 的例子，使得至少有三种不同的无损分解可将 R' 转化为 BCNF。

8.16 令**主**(prime)**属性**为至少在一个候选码中出现的属性。令 α 和 β 为属性集，使得 $\alpha \rightarrow \beta$ 成立但 $\beta \rightarrow \alpha$ 不成立。令 A 为一属性，它不属于 α 或 β，并且 $\beta \rightarrow A$ 成立。我们称 A **传递依赖**(transitively dependent)于 α。我们可以重新如下定义 3NF：*具有函数依赖集 F 的关系模式 R 属于 3NF，如果 R 中没有非主属性 A 传递依赖于 R 的一个码*。证明这个新定义等价于原定义。

8.17 函数依赖 $\alpha \rightarrow \beta$ 称为**部分依赖**(partial dependency)的条件是，存在 α 的真子集 γ 使得 $\gamma \rightarrow \beta$。我们称 β *部分依赖*于 α。关系模式 R 属于**第二范式**(Second Normal Form，2NF)，如果 R 中的每个属性 A 都满足如下准则之一：

- 它出现在一个候选码中。
- 它没有部分依赖于一个候选码。

证明每个 3NF 模式都属于 2NF。（*提示*：证明每个部分依赖都是传递依赖。）

8.18 给出一个关系模式 R 和依赖集的例子，使得 R 属于 BCNF，但不属于 4NF。

习题

8.19 给出实践习题 8.1 中模式 R 的一个无损连接的 BCNF 分解。

8.20 给出实践习题 8.1 中模式 R 的一个无损连接并保持依赖的 3NF 分解。

8.21 将具有如下约束的下列模式规范化为 4NF。

books (*accessionno*, *isbn*, *title*, *author*, *publisher*)
users (*userid*, *name*, *deptid*, *deptname*)
accessionno → *isbn*
isbn → *title*
isbn → *publisher*
isbn →→ *author*
userid → *name*
userid → *deptid*
deptid → *deptname*

372 8.22 解释什么是*信息重复*和*无法表示信息*。解释为什么这些性质可能意味着不好的关系数据库设计。

8.23 为什么某些函数依赖称为*平凡*的函数依赖？

8.24 使用函数依赖的定义论证 Armstrong 公理(自反律、增补律、传递律)是正确有效的。

8.25 考虑下面提出的函数依赖规则：若 $\alpha \rightarrow \beta$ 且 $\gamma \rightarrow \beta$，则 $\alpha \rightarrow \gamma$。通过给出一个关系 r，满足 $\alpha \rightarrow \beta$ 且 $\gamma \rightarrow \beta$ 但不满足 $\alpha \rightarrow \gamma$，证明该规则不是正确有效的。

8.26　用 Armstrong 公理证明分解律的正确有效性。

8.27　用实践习题 8.6 中的函数依赖计算 B^+。

8.28　证明实践习题 8.1 中的模式 R 的如下分解不是无损分解：

$$(A, B, C)$$
$$(C, D, E)$$

提示：给出模式 R 上一个关系 r 的例子，使得

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

8.29　考虑如下关系模式 $r(A, B, C, D, E, F)$ 上的函数依赖集 F：

$$A \rightarrow BCD$$
$$BC \rightarrow DE$$
$$B \rightarrow D$$
$$D \rightarrow A$$

a. 计算 B^+。

b. (使用 Armstrong 公理)证明 AF 是超码。

c. 计算上述函数依赖集 F 的正则覆盖；给出你的推导的步骤并解释。

d. 基于正则覆盖，给出 r 的一个 3NF 分解。

e. 利用原始的函数依赖集，给出 r 的一个 BCNF 分解。

f. 你能否利用正则覆盖得到与上面的 r 相同的 BCNF 分解？

8.30　列出关系数据库设计的三个目标，并解释为什么要达到每个目标。 373

8.31　在关系数据库设计中，为什么我们有可能会选择非 BCNF 设计？

8.32　给定关系数据库设计的三个目标，有没有理由设计一个属于 2NF 但不属于任何一个更高范式的数据库模式？(2NF 的定义参见实践习题 8.17。)

8.33　给定一个关系模式 $r(A, B, C, D)$，$A \rightarrow\!\!\!\!\rightarrow BC$ 是否逻辑蕴涵 $A \rightarrow\!\!\!\!\rightarrow B$ 和 $A \rightarrow\!\!\!\!\rightarrow C$？如果是请证明之，否则给出一个反例。

8.34　解释为什么 4NF 是一个比 BCNF 更好的范式。

文献注解

对于关系数据库设计理论最初的讨论是在 Codd[1970]这篇早期论文中。在那篇论文中，Codd 引入了函数依赖，以及第一、第二和第三范式。

Armstrong 公理是由 Armstrong[1974]引入的。在 20 世纪 70 年代后期关系数据库理论有了显著的进展。这些结果收集在一些数据库理论的课本中，包括 Maier[1983]、Atzeni 和 Antonellis[1993]以及 Abiteboul 等[1995]。

BCNF 是在 Codd[1972]中引入的。Biskup 等[1979]给出了用于找到无损并保持依赖的 3NF 分解的算法。有关无损分解特性的基础结论在 Aho 等[1979a]中给出。

Beeri 等[1977]给出了一组多值依赖的公理，并证明这些公理是正确有效且完备的。4NF、PJNF 和 DKNF 的概念分别来自 Fagin[1977]、Fagin[1979]和 Fagin[1981]。关于规范化的进一步的参考文献，请参看附录 C 中的文献注解。

Jensen 等[1994]给出了一个时态数据库概念的术语表。Gregersen 和 Jensen[1999]给出了一个对 E-R 建模进行扩展以处理时态数据的综述。Tansel 等[1993]探讨了时态数据库的理论、设计和实现。Jensen 等[1996]描述了将依赖理论扩展至时态数据的方法。 374