第1章 引 言

数据库管理系统（DBMS）由一个互相关联的数据的集合和一组用以访问这些数据的程序组成，这个数据集合通常称作数据库，其中包含了关于某个企业的信息。DBMS的基本目标是要提供一个可以方便地、有效地存取数据库信息的环境。

设计数据库系统的目的是为了管理大量信息。对数据的管理既涉及到信息存储结构的定义，又涉及信息操作机制的提供。另外，数据库系统还必须提供所存储信息的安全性保证，即使在系统崩溃或有人企图越权访问时也应保障信息的安全性。如果数据将被多用户共享，那么系统还必须设法避免可能产生的异常结果。

对大多数组织而言，信息都非常重要，这决定了数据库的价值，并使得大量的用于有效管理数据的概念、技术得到发展。本章将简要介绍数据库系统的基本原理。

1.1 数据库系统的目的

假设储蓄银行的某个部门需要保存所有客户及储蓄帐户的信息，在计算机上保存这些信息的一种方法是将它们存放在永久性系统文件中。为了使用户可以对信息进行操作，系统中应有一些对文件进行操作的应用程序，包括：

- 处理某帐户的借/贷程序。
- 创建新帐户的程序。
- 查询帐户余额的程序。
- 产生每月财务报告的程序。

这些应用程序是由系统程序员根据银行的需求编写的。

随着需求的增长，新的应用程序加入到系统中来。例如，如果政府颁布新条例允许储蓄银行开设支票帐户，这时银行就需要创建新的永久文件来存放银行所维护的所有支票帐户的信息，同时还可能需要编写新的应用程序来处理那些在储蓄帐户里不会出现的情况（例如：透支）。因此，随着时间的推移，越来越多的文件和应用程序加入到系统中。

以上所描述的典型的文件处理系统是传统的操作系统所能支持的。永久记录被存储在多个不同的文件中，人们编写不同的应用程序来将记录从适当的文件中取出或加入到适当的文件中。在DBMS出现以前，各个组织通常都采用这样的系统来存储信息。

在文件处理系统中存储信息的主要弊端包括：

- 数据的冗余和不一致。由于文件和程序是很长一段时间内由不同的程序员创建的，因此不同文件可能采用不同格式，不同程序可能采用不同语言。此外，相同的信息可能在几个地方（文件）重复存储。例如，某个客户的地址和电话号码可能既在由储蓄帐户记录组成的文件中出现，也可能又在由支票帐户记录组成的文件中出现。这种冗余除了导致存储和访问开销增大以外，还可能导致数据不一致，即同一数据的不同副本不一致。例如，某个客户地址的更改可能只在储蓄帐户记录中得到反映而在系统的其他地方却没有得到反映。
- 数据访问困难。假设银行的某个高级职员想要找出所有居住地邮编为78733的客户的姓名，这时他会要求数据处理部门生成这样的一个列表。由于在最初的系统设计时并未预料到

会有这样的需求，所以没有现成的应用程序去满足此需求。但是，系统中却有一个产生所有客户列表的应用程序，这时该高级职员有两种选择：一种是取得所有客户的列表并从中手工提取所需信息；另一种是要求数据处理部门让某个系统程序员编写相应的应用程序。这两种方案显然都不太令人满意，假设编写了相应的程序，几天后这个高级职员可能又需要将该列表减少到只列出帐户余额不少于 $10 000 的那些客户，可以预见，产生这样一个列表的程序又不存在，这个高级职员就又再一次面临着前面那两种不尽人意的选择。

这里要着重指出的是，传统的文件处理环境不能支持以一种方便而有效的方式去获得所需数据，需要开发通用的、能对变化的需求作出更快反应的数据检索系统。

• 数据孤立。由于数据分散在不同文件中，这些文件又可能具有不同的格式，因而编写新的且检索恰当数据的应用程序是很困难的。

• 完整性问题。数据库中所存储数据的值必须满足某种一致性约束。例如，银行帐户的余额永远不会低于某个预定的值（如：$25)。开发者通过在不同应用程序中加入适当的代码来体现系统中的这些约束。然而，当新的约束加入时，很难通过修改程序来体现这些新的约束。尤其是当约束涉及不同文件中的多个数据项时，问题就变得更加复杂了。

• 原子性问题。如同其他的机械或电子设备一样，计算机系统也常常会发生故障，一旦故障发生并被检测到，数据就应恢复到与故障发生前一致的状态。对许多应用来说，这样的功能是至关重要的。让我们看看把 A 帐户的 $50 转入 B 帐户这样一个例子。假设在程序的执行过程中发生了系统故障，很可能 A 帐户上减去的 $50 还没来得及存入 B 帐户，这就造成了数据库状态的不一致。显然，为了保证数据库状态的一致性，这里的借、贷两个操作必须是要么都发生要么都不发生，也就是说，转帐这个操作必须是原子的——它要么全部发生要么根本不发生。在传统的文件处理系统中，这样的性能难以得到保证。

• 并发访问异常。为了提高系统的总体性能以及加快响应速度，许多系统允许多个用户同时更新数据。在这样的环境中，并发更新操作相互影响，可能就会导致数据的不一致。设 A 帐户中有 $500，假如两个客户几乎同时从 A 帐户中取款，分别取出 $50 和 $100，这样的并发执行就可能使帐户处于一种错误的或者说不一致的状态。假设每个取款操作对应执行的程序是读取帐户余额，在其上减去取款的金额，然后将结果写回。如果两次取款的程序并发执行，可能它们读到的余额都是 $500，并将分别写回 $450 和 $400，帐户中到底剩下 $450 还是 $400 要视哪个程序后写回结果而定，而实际上这两种结果都是错的，正确的值应该是 $350。由此可见，为了消除这种情况发生的可能性，系统必须进行某种形式的管理。但是，由于数据可能被多个不同应用程序访问，这些程序相互间事先又没有协调管理，因而很难进行。

• 安全性问题。并非数据库系统的所有用户都可以访问所有数据。例如在银行系统中，工资发放人员只需要看到数据库中关于银行员工的那部分信息，他们不需要访问关于客户帐户的信息。由于应用程序总是即兴加入到系统中来，这样的安全性约束难以实现。

以上问题以及还未提到的一些其他问题，加速了 DBMS 的发展。接下来看一看数据库系统为了解决上述问题而提出的概念和算法。本书的大部分篇幅在讨论企业常见的数据处理应用时总以银行为实例。典型的数据处理应用中存储记录的数量总是很大，而每一条记录既小又简单。

在第 8、9 章中，我们将讨论一些其他类型的数据库应用，例如用于交互式设计的应用，这些应用常常需要处理更大更复杂的记录，例如一个完整的建筑设计，而其所需的记录数通常较少。目前许多研究和开发工作正致力于提供能管理这些应用且既足够强大又足够灵活的数据库系统。

1.2 数据视图

DBMS 是一些互相关联的文件以及一组使得用户可以访问和修改这些文件的程序集合。数据库系统的一个主要目的是给用户提供数据的抽象视图，也就是说，系统隐藏关于数据存储和维护的某些细节。

1.2.1 数据抽象

一个可用的系统必须能有效地检索数据。为了达到这样的要求，人们设计了复杂的数据结构，用来在数据库中表示数据。由于许多数据库系统的用户并未受过计算机专业训练，系统开发人员通过如下几个层次的抽象来向用户屏蔽复杂性，以简化系统的用户界面：

• 物理层。最低层次的抽象，描述数据实际上是怎样存储的。物理层详细描述复杂的低层数据结构。

• 逻辑层。比物理层层次稍高的抽象，描述数据库中存储什么数据以及这些数据间存在什么关系，因而整个数据库通过少量相对简单的结构来描述。虽然简单的逻辑层结构的实现涉及到复杂的物理层结构，但逻辑层的用户不必知道这种复杂性，逻辑层抽象是由数据库管理员所使用的，管理员必须确定数据库中应该保存哪些信息。

• 视图层。最高层次的抽象，但只描述整个数据库的某个部分。尽管在逻辑层使用了比较简单的结构，但由于数据库的规模巨大，所以仍存在一定程度的复杂性。数据库系统的多数用户并不需要关心所有的信息，而只需要访问数据库的一部分。视图抽象层的定义正是为了使用户与系统的交互更简单。系统可以为同一数据库提供多个视图。

这三层抽象的相互关系如图 1-1 所示。

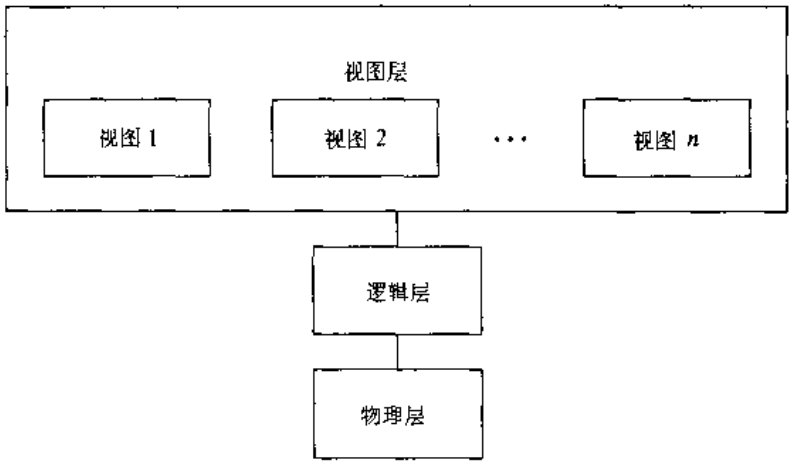


图 1-1 数据抽象的三个层次

通过与程序设计语言中数据类型的概念进行类比，可以弄清各层抽象间的区别。大多数高级程序设计语言支持记录类型的概念。例如，用一种类 Pascal 语言，可以定义如下记录：

```
type customer = record
                    customer-name : string;
                    social-security : string;
                    customer-street : string;
                    customer-city : string;
```

end;

以上代码定义了具有四个字段的新记录 *customer*。每个字段有一个字段名和所属类型。对一个银行来说，可能有几个这样的记录类型：

- *account*。包含字段 *account-number* 和 *balance*。
- *employee*。包含字段 *employee-name* 和 *salary*。

在物理层，*customer*、*account* 或 *employee* 记录可能被描述为由连续存储位置（如字或字节）组成的存储块。语言编译器为程序设计人员屏蔽了这一层的细节。与此类似，数据库系统为数据库程序设计人员屏蔽了许多低层的存储细节，然而数据库管理员可能需要了解数据物理组织的某些细节。

在逻辑层，正如前面的代码段所示，每个这样的记录通过类型定义进行描述。在逻辑层上同时还要定义这些记录类型的相互关系。程序设计人员正是在这个抽象层次上使用某种程序设计语言进行工作。与此类似，数据库管理员也常常在这个抽象层次上工作。

最后，在视图层，计算机用户看见的是为其屏蔽了数据类型细节的一组应用程序。与此类似，视图层上定义了数据库的多个视图，数据库用户看到的是这些视图。除了屏蔽数据库的逻辑层细节以外，视图还提供了防止用户访问数据库某些部分的安全性机制。例如，银行的出纳员只能看见数据库中关于客户帐户信息的部分，而不能访问涉及员工工资的信息。

1.2.2　实例和模式

随着时间的推移，信息会被插入或删除，数据库因而也就发生了改变。特定时刻存储在数据库中的信息的集合称作数据库的一个实例，而数据库的总体设计称作数据库模式。数据库模式即使发生变化，也不是很频繁的。

在此很有必要将这些概念同数据类型、变量以及值的概念做一个类比，再回过头来看一下 *customer* 记录类型的定义。注意，我们只是定义了 *customer* 类型而没有定义任何变量。以类 Pascal 语言为例，可以这样来定义变量：

var *customer*1 : *customer*;

变量 *customer*1 对应于一块包含 *customer* 类型记录的存储区域。

数据库模式对应于程序设计语言中的类型定义。给定类型的一个变量在某个给定的时刻有特定的值。因此，程序设计语言中变量的值对应于数据库模式的一个实例。

根据前面所讨论的抽象层次的不同，数据库系统可以分为不同模式。处于最低层的是物理模式，其次是逻辑模式，最高层是子模式。通常，数据库系统支持一个物理模式、一个逻辑模式和多个子模式。

1.2.3　数据独立性

在某个层次上修改模式定义而不影响位于其上一层模式的能力叫做**数据独立性**。有两个层次的数据独立性：

1）物理数据独立性是修改物理模式而不必重写应用程序的能力。为了提高性能，偶尔会在物理层做一些修改。

2）逻辑数据独立性是修改逻辑模式而不必重写应用程序的能力。只要数据库的逻辑结构发生了变化（例如，当金融市场帐户加入银行系统中时），逻辑层就需要做一些修改。

逻辑数据独立性比物理数据独立性更难做到，这是因为应用程序对于它们所访问的数据的逻辑结构依赖程度很大。

数据独立性的概念同现代程序设计语言中抽象数据类型的概念在许多方面是相似的。两者均为用户屏蔽具体实现的细节，使用户可以只考虑概括的结构，而不必考虑低层实现细节。

1.3 数据模型

数据库结构的基础是数据模型。数据模型是描述数据、数据联系、数据语义以及一致性约束的概念工具的集合。现有数据模型可分为三类：基于对象的逻辑模型、基于记录的逻辑模型和物理模型。

1.3.1 基于对象的逻辑模型

基于对象的逻辑模型用于在逻辑层和视图层上描述数据。其特点是，提供灵活的结构组织能力，允许显式定义数据约束。现在已经有多种属于这一类的模型，或许将来还会有更多。几种较著名的逻辑模型是：

- 实体-联系模型。
- 面向对象的模型。
- 语义数据模型。
- 功能数据模型。

本书把实体-联系模型和面向对象的模型作为基于对象的逻辑模型的代表，对它们进行讨论。实体-联系模型在数据库设计中被多数人所接受，在实践中也有广泛的应用，我们将在第2章讨论。面向对象的模型包含了实体-联系模型中的许多概念，但其在表示数据以外还能表示可执行代码，我们将在第8章讨论。面向对象的模型在实践中正获得越来越多的认同。接下来将对这两种模型进行简要介绍。

1. 实体-联系模型

实体-联系（E-R）数据模型基于对现实世界的这样一种认识：现实世界是由一组称作实体的基本对象以及这些对象间的联系构成的。实体是现实世界中可区别于其他对象的一个“事件”或一个“物体”，例如，每个人是一个实体，每个银行帐户也是一个实体。数据库中实体通过属性集合来描述，例如，帐户号（*account-number*）与余额（*balance*）属性描述了银行的某个特定帐户。联系是实体间的相互关联，例如，存款者联系将一个客户和他的帐户相关联。同一类型的所有实体的集合称作实体集，同一类型的所有联系的集合称作联系集。

除了实体和联系以外，E-R模型中还可以表示出数据库内容必须遵循的特定约束。一个重要的约束是映射的基数，它表示通过某个联系集能与另一实体进行关联的实体数目。

数据库的总体逻辑结构可以用E-R图进行图形表示。E-R图由以下元素构成：

- 矩形。代表实体集。
- 椭圆。代表属性。
- 菱形。代表实体集间的联系。
- 段。将属性与实体集相连或将实体集与联系相连。

每个成分都标上它所代表的实体或联系。

作为例子，我们来看一下银行系统数据库中由客户及其帐户组成的部分，对应的E-R图如图1-2所示。这个例子将在第2章进一步展开。

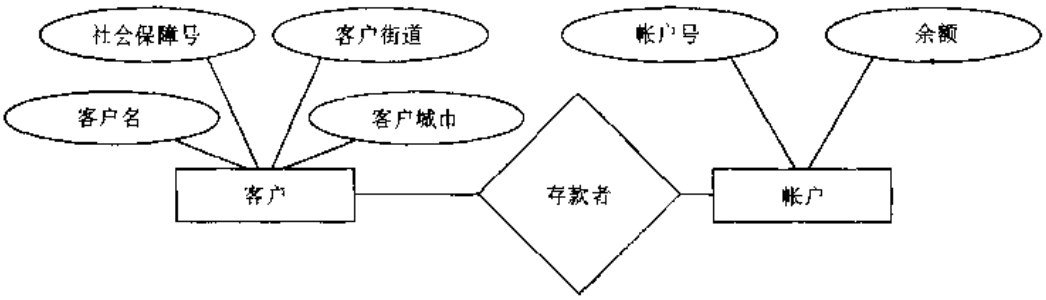


图 1-2　E-R 图示例

2. 面向对象的模型

和 E-R 模型一样，面向对象的模型基于对象的一个集合。一个对象既包括对象内存储在实例变量中的值，又包括对此对象进行操作的代码体，代码体称作方法。

对象划分为不同的类，含有相同类型的值和相同方法的对象属于同一个类。类可以看作是对象的类型定义。这种将数据和方法相结合的类型类似于程序设计语言中的抽象数据类型。

一个对象访问另一个对象数据的唯一途径是激发被访问对象的方法，这一行为称作向对象发送消息。因此，对象方法的调用接口定义了该对象的外部可见部分，对象的内部（实例变量和方法代码）对外部是不可见的，这样就可以产生两个层次的数据抽象。

为了阐明上述概念，来看一个表示银行帐户的对象，这样的一个对象包括实例变量（帐户号 *account-number* 和余额 *balance*）和方法（付息 *pay-interest*，即在帐户余额上加上利息）。假设银行原来对所有帐户均付 6% 的利息，现在要改为少于 $1000 的帐户只付给 5% 的利息，而不少于 $1000 的帐户才付给 6% 的利息。对大多数的数据模型而言，做这样的更改将涉及一个或多个应用程序代码的修改；而对面向对象的模型而言，只需要在 *pay-interest* 方法内部做修改，而对象的外部接口保持不变。

与 E-R 模型中的实体不同，面向对象模型中的每个对象都有一个与其所包含值无关的唯一标识，因此，包含相同值的两个对象仍然是有区别的。这种不同对象的区别是通过在物理层赋给对象唯一的对象标识符来实现的。

1.3.2　基于记录的逻辑模型

基于记录的逻辑模型用于在逻辑层和视图层描述数据。与基于对象的数据模型不同，基于记录的模型既用来定义数据库的全局逻辑结构，又用来提供关于实现的高层描述。

基于记录的模型的名称由来是由于它用一些固定格式的记录来描述数据库结构。每个记录类型定义了固定数目的字段（或属性），通常每个字段的长度也是固定的。在第 10 章我们将看到，使用定长记录可以简化数据库的物理层实现。这种简单性同许多基于对象的模型形成对照，后者丰富的结构常常会导致物理层的变长记录。

基于记录的模型中广为接受的是关系模型、网状模型和层次模型，近年来关系模型受欢迎的程度超过了其他两种，我们将在第 3～7 章对其进行详细讨论。网状模型和层次模型仍在大量早期开发的数据库中使用，在附录中我们将有所讨论。这里先对每个模型进行简要介绍。

1. 关系模型

关系模型用表的集合来表示数据和数据间的联系。每个表有多个列，每列有唯一列名。图 1-3 是一个关系型数据库示例，其中包括两个表：一个表中是银行的客户，另一个表中是属于这些客户的帐户。例如，从其中我们可以得到这样的信息：客户 Johnson 的社会保障号是 192-

83-7465，居住在 Palo Alto 市的 Alma 大街。他有两个帐户：A-101 帐户余额为 $500，A-201 帐户余额为 $900。注意，客户 Johnson 和 Smith 共有帐户 A-201(或许他们共同进行了商业投资)。

customer-name	*social-security*	*customer-street*	*customer-city*	*account-number*
Johnson	192-83-7465	Alma	Palo Alto	A-101
Smith	019-28-3746	North	Rye	A-215
Hayes	677-89-9011	Main	Harrison	A-102
Turner	182-73-6091	Putnam	Stamford	A-305
Johnson	192-83-7465	Alma	Palo Alto	A-201
Jones	321-12-3123	Main	Harrison	A-217
Lindsay	336-66-9999	Park	Pittsfield	A-222
Smith	019-28-3746	North	Rye	A-201

account-number	*balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

图 1-3 关系数据库示例

2. 网状模型

网状模型中的数据用记录（与 Pascal 语言中的记录含义相同）的集合来表示，数据间的联系用链接（可看作指针）来表示。数据库中的记录可被组织成任意图的集合。图 1-4 是一个网状数据库示例，其中信息与图 1-3 给出的相同。

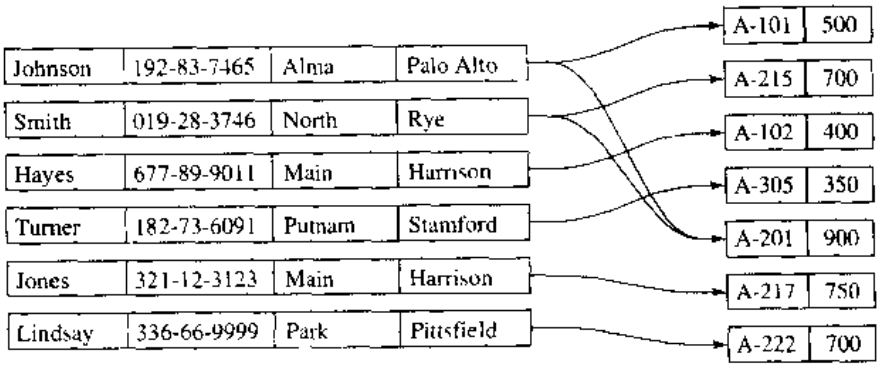


图 1-4 网状数据库示例

3. 层次模型

层次模型与网状模型类似，分别用记录和链接来表示数据和数据间的联系。层次模型不同于网状模型的地方是：层次模型中的记录只能组织成树的集合而不能是任意图的集合。图 1-5 是一个层次数据库示例，其中信息与图 1-4 给出的相同。

4. 三种模型的差别

关系模型与网状模型及层次模型不同的地方在于关系模型不使用指针或链接，而通过记录所包含的值把记录联系起来。使用这样的方式可以为关系模型定义规范的数学基础。

1.3.3 物理数据模型

物理数据模型用于在最低层次上描述数据。与逻辑数据模型不同，实际使用中的物理数

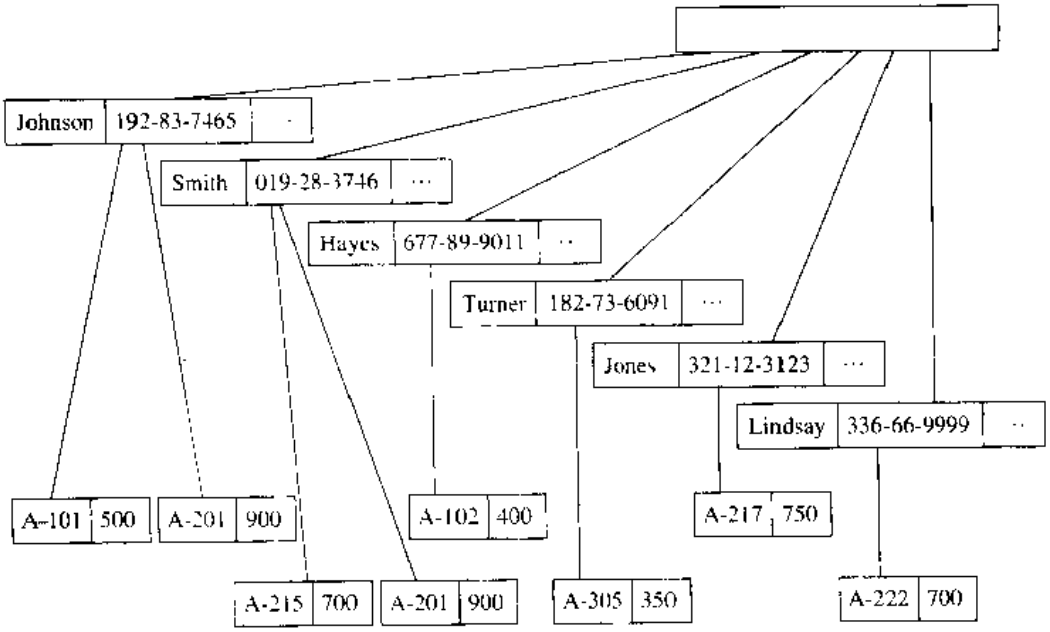


图 1-5　层次数据库示例

据模型较少。常用的两种物理数据模型是一致化模型和框架存储模型。

物理数据模型同数据库系统实现方面的内容有关，这不是本书所要讨论的内容。

1.4　数据库语言

数据库系统提供两种不同类型的语言：一种用于定义数据库模式；另一种用于表达数据库的查询和更新。

1.4.1　数据定义语言

数据库模式是通过一系列定义来说明的，这些定义由称作*数据定义语言*（DDL）的一种特殊语言来表达。DDL 语句的编译结果是产生了存储在一个特殊文件中的一系列表，称作*数据字典*或*数据目录*。

数据字典是一个包含*元数据*的文件，元数据是关于数据的数据。在数据库系统中，实际数据读取和修改前总要先查询该文件。

数据库系统所使用的存储结构和访问方式通过一系列特殊的 DDL 语句来定义，这种特殊的 DDL 语句称作*数据存储定义语言*。这些语句的编译结果是一系列用来描述数据库模式实现细节的指令，这些实现细节对用户来说通常是不可见的。

1.4.2　数据操纵语言

1.2 节讨论的数据抽象层次不仅适用于定义数据和组织数据，还适用于对数据的操纵。*数据操纵*是指：

- 对存储在数据库中的信息进行检索。
- 向数据库中插入新的信息。
- 从数据库中删除信息。
- 修改数据库中存储的信息。

在物理层，必须定义可以高效访问数据的算法。在较高的抽象层次上，我们强调数据的易用性，目的是要提供人与系统间的有效交互。

数据操纵语言（DML）使得用户可以访问和操纵由适当的数据模式组织起来的数据，通常有两类数据操纵语言：

- 过程化的 DML 要求用户指定需要什么数据以及如何获得这些数据。
- 非过程化的 DML 只要求用户指定需要什么数据，而不必指明如何获得这些数据。

通常非过程化的 DML 比过程化的 DML 易学易用。但是，由于非过程化的 DML 的用户不必指明如何获得数据，通常这种语言产生的代码不如过程化语言产生的代码的效率高。可以通过各种优化技术来克服这个问题。在第 12 章里我们将讨论一些优化技术。

查询是用来对信息进行检索的语句。DML 中涉及信息检索的部分称作查询语言。实践中常认为查询语言和数据操纵语言是同义的，虽然严格说来这并不正确。

1.5 事务管理

通常，对数据库的几个操作合起来形成一个逻辑执行单元。前面已经看见的一个例子是资金转帐，其中的一个帐户（A 帐户）发生借出操作而另一帐户（B 帐户）发生贷入操作。显然，这两个操作必须保证要么都发生要么都不发生，也就是说，资金转帐必须完成或根本不发生。这种要求要么完成要么不发生的特性称为原子性。除此以外，资金转帐还必须保持数据库的一致性，也就是说，$A + B$ 的和应该是保持不变的，这种正确性的要求称作一致性。最后一点，当资金转帐成功结束后，即使发生系统故障，帐户 A 和帐户 B 的值也应该保持不变，这种保持不变的要求称作持久性。

事务是数据库应用中完成单一逻辑功能的操作集合，是一个既具原子性又具一致性的单元。因此，我们要求事务不违反任何的数据库一致性约束，也就是说，如果事务启动时数据库是一致的，那么当这个事务成功完成时数据库也应该是一致的。但是，在事务执行过程中，必要时允许暂时的不一致，这种暂时的不一致尽管是必需的，但在故障发生时，很可能导致问题的产生。

正确定义不同事务是程序员的任务，事务的定义应使之能保持数据库的一致性。例如，资金从帐户 A 转到帐户 B 可以定义成由两个单独的程序组成：一个对帐户 A 执行借出操作，另一个对帐户 B 执行贷入操作。这两个程序的依次执行可以保持一致性，但是，这两个程序本身都不能把数据库从一个一致状态转入另一个新的一致状态，因此它们都不是事务。

原子性和持久性的保证则是数据库系统自身的任务，更确切一些，是事务管理器的任务。在没有故障发生的情况下，所有事务均成功完成，这时要保证原子性很容易。但是，由于各种各样的故障，事务并不总能成功执行完毕。为了保证原子性，失败的事务必须对数据库状态不产生任何影响，因此，数据库必须能恢复到该失败事务开始执行以前的状态。数据库系统应该能检测到系统故障并将数据库恢复到故障发生以前的状态。

最后，当多个事务同时对数据库进行更新时，即使每个单独的事务都是正确的，数据的一致性也可能被破坏。并发控制管理器控制并发事务间的相互影响，保证数据库的一致性。

为小型个人计算机所设计的数据库系统可能并不具备上述所有特征。例如，很多小系统限制在一个时刻只允许一个用户访问数据库，有的系统把备份和恢复的任务留给用户。这样的设定可以降低对物理资源（尤其是主存）的要求，使得数据管理相对简单。这种低成本低能力的方式对小型个人数据库来说已经足够，但不能满足大中型企业的需求。

1.6 存储管理

数据库常常需要大量存储空间。公司数据库的大小是用 *gigabyte*（10^9 字节，1GB）来计算的，最大的甚至需要用 *terabyte*（10^{12} 字节，1TB）来计算。一个 gigabyte 等于 1000 个 megabyte（10^6 字节，1MB），1 个 terabyte 等于 100 万个 megabyte。由于计算机主存不可能存储这么多信息，因而信息被存储在磁盘上，需要时信息在主存和磁盘间移动。由于同中央处理器的速度相比数据出入磁盘的速度很慢，数据库系统对数据的组织必须满足使磁盘和主存间数据移动的需求最小化。

数据库系统的目标是要简化和辅助数据访问，高层视图有助于实现这样的目标。系统用户可以不受系统实现的物理细节所带来的不必要的负担所累。但是，决定用户对数据库系统满意与否的一个主要因素是系统的性能。如果一个要求的响应速度太慢，系统的价值就会下降。系统性能决定于用来表示数据库中数据的数据结构的高效性，以及系统对这样的数据结构进行操作的高效性。正如计算机系统中其他地方也会出现一样，不仅要在时间与空间两者间进行权衡，还要在不同操作的效率间进行权衡。

*存储管理器*是在数据库中存储的低层数据与应用程序及向系统提交的查询之间提供接口的程序模块。存储管理器应负责与文件管理器的交互。原始数据通过文件系统存储在磁盘上，文件系统通常由传统的操作系统所提供。存储管理器将不同的 DML 语句翻译成低层文件系统命令，因此，存储管理器负责数据库中数据的存储、检索和更新。

1.7 数据库管理员

使用 DBMS 的一个主要原因是可以对数据和访问这些数据的程序进行集中控制。对系统进行集中控制的人称作*数据库管理员*（DBA）。DBA 的作用包括：

- 模式定义。DBA 通过书写一系列的定义来创建最初的数据库模式，这些定义被 DDL 编译器翻译成永久地存储在数据字典中的表集合。
- 存储结构及存取方式定义。DBA 通过书写一系列的定义来创建适当的存储结构和存取方式，这些定义由数据存储和数据定义语言编译器来翻译。
- 模式及物理组织的修改。程序设计人员偶尔也会对数据库模式或物理存储组织的描述进行修改，这是通过书写一系列的定义来实现的。DDL 编译器或数据存储和数据定义语言编译器使用这些定义，对适当的内部系统表（例如：数据字典）产生修改。
- 数据访问授权。通过授予不同的权限，数据库管理员可以规定不同的用户各自可以访问的数据库的部分。授权信息保存在一个特殊的系统结构中，一旦系统中有访问数据的要求，数据库系统就会去查阅这些信息。
- 完整性约束的定义。数据库中所存储的数据的值必须满足一定的一致性约束。例如，员工一周工作的总时间也许不能超过某个特定的限制（如：80 小时），这样的约束必须由数据库管理员显式定义。完整性约束保存在一个特殊的系统结构中，一旦系统中发生更新，数据库系统就去查阅这些约束。

1.8 数据库用户

数据库系统的一个基本目标是提供从数据库中检索信息和往数据库中存储新信息的环境。按照与系统交互方式的不同，数据库系统的用户可以分为四种不同类型：

- 应用程序设计人员是计算机专业人员，他们通过 DML 调用，同系统进行交互。

DML 调用嵌在用宿主语言（如 Cobol、PL/I、Pascal、C）书写的程序中，这些程序通常称为应用程序。银行系统中应用程序的例子包括工资支票产生程序、帐户借出程序、帐户贷入程序、以及帐户间的转帐程序。

由于 DML 调用语法通常与宿主语言的语法显著不同，为了产生正确代码，DML 调用常常以一个特殊的字符作为前导。一个称为 DML 预编译器的特殊编译器将 DML 语句转化为宿主语言中的普通过程调用语句，接着，预编译所产生的程序通过宿主语言的编译器产生正确的目标代码来运行。

有一些特殊的程序设计语言将类 Pascal 语言的控制结构与用于操纵数据库对象（如：关系）的控制结构结合起来，这样的语言有时称作第四代语言，通常包括帮助建立表格以及在屏幕上显示数据的工具。多数主要的商用数据库系统中含有第四代语言。

- 富有经验的用户并非通过编写程序来同系统交互，而是用数据库查询语言来表达他们的要求。每个这样的查询都被提交给查询处理器，其作用是将 DML 语句分解为能被存储管理器所能理解的指令。分析员通过提交查询来研究数据库中的数据，所以属于这一类用户。
- 专门的用户是编写专门的、不适合于传统数据处理模式的数据库应用程序的富有经验的用户。这样的应用包括：计算机辅助设计系统、知识库和专家系统、存储复杂结构数据（如图像数据和声音数据）的系统、以及环境建模系统。在第 8 章和第 9 章中将要讨论几个这样的应用。
- 新用户是没有经验的用户，他们通过激活以前已经写好的一个永久的应用程序同系统进行交互。例如，银行出纳员在将帐户 A 的 \$50 转入帐户 B 时激活一个叫做 *transfer* 的程序，该程序要求出纳员输入转帐金额、转出的帐户以及转入的帐户。

1.9 系统总体结构

数据库系统划分为不同的模块，每个模块完成整个系统的一个功能。数据库系统的部分功能由计算机的操作系统提供。通常，计算机操作系统提供最基本的服务，数据库系统就建立在这个基础上。因此，设计数据库系统必须考虑到数据库系统与操作系统的接口。

数据库系统的功能部件大致可分为查询处理器部件和存储管理器部件。查询处理器部件包括：

- DML 编译器。将查询语言中 DML 语句翻译成查询求值引擎能理解的低级指令。另外，DML 编译器力图将用户请求转换成一个等价的但效率更高的形式，以找到执行查询的更好策略。
- 嵌入式 DML 预编译器。将嵌在应用程序中的 DML 语句转化成宿主语言中普通的过程调用语句。预编译器必须同 DML 编译器共同发挥作用，以产生正确代码。
- DDL 解释器。解释 DDL 语句并将其记录到包含元数据的一系列表中。
- 查询求值引擎。执行由 DML 编译器产生的低级指令。

存储管理器部件提供数据库中存储的低层数据与应用程序及向系统提交的查询之间的接口，包括：

- 权限及完整性管理器。检测是否满足完整性约束，检查试图访问数据的用户的权限。
- 事务管理器。保证即使发生了故障，数据库也保持在一致的（正确的）状态；保证并发事务的执行不发生冲突。
- 文件管理器。管理磁盘空间的分配，管理用于表示数据库所存储信息的数据结构。
- 缓冲管理器。负责将数据从磁盘上取到内存中来，并决定哪些数据应被缓冲存储在内存中。

另外，还要求几个数据结构，以作为系统物理实现的一部分：

• 数据文件。存储数据库自身。

• 数据字典。存储关于数据库结构的元数据。由于频繁使用数据字典，因此，字典的良好设计和高效实现是非常重要的。

• 索引。提供对包含特定值的数据项的快速访问。

• 统计数据。存储关于数据库中数据的统计信息。这些信息被查询处理器用来选择高效地执行查询的方法。

图 1-6 所示为这些部件及其相互间的联系。

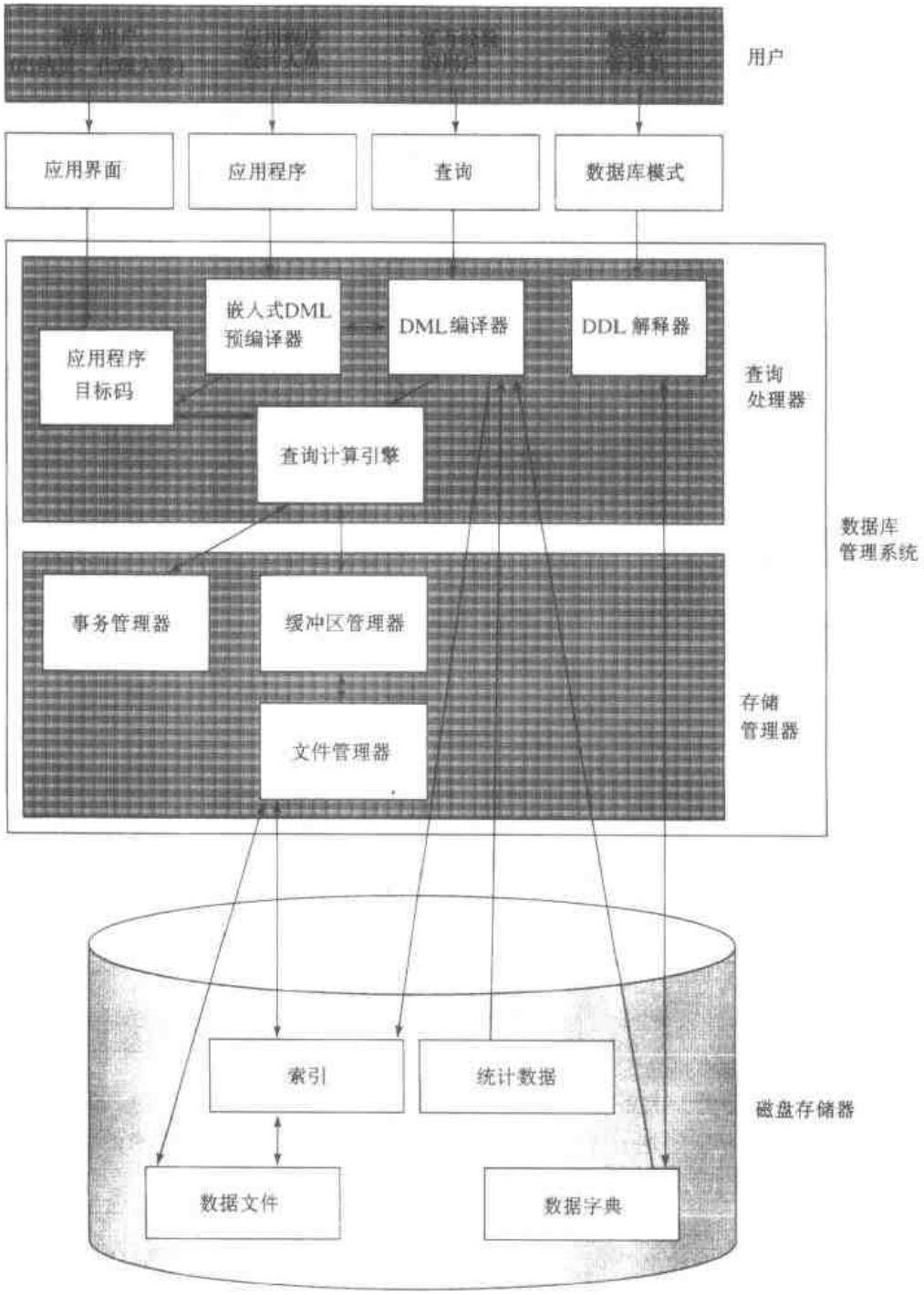


图 1-6　系统结构

1.10 总结

数据库管理系统由一个互相关联的数据的集合和一组用以访问这些数据的程序组成。这些数据用以描述某个特定的企业。DBMS的基本目标是要提供一个可以让人们方便地、高效地存取信息的环境。

数据库系统用于存储大量信息。对数据的管理既涉及到信息存储结构的定义，又涉及到信息操作机制的提供。另外，数据库系统还必须提供所存储数据的安全性保证，即使在系统崩溃或有人企图越权访问时也应如此。如果数据被多用户共享，那么系统还必须设法避免可能产生的异常结果。

数据库系统的一个主要目的是要提供给用户数据的抽象视图，也就是说，系统隐藏了数据存储和维护的细节。这是通过定义三个可对数据库系统进行观察的抽象层次来实现的。这三个层次是：物理层、逻辑层和视图层。

数据库结构的基础是数据模型，一个用于描述数据、数据间关系、数据语义和数据约束的概念工具的集合。现有的不同数据模型分为三类：基于对象的逻辑模型、基于记录的逻辑模型以及物理数据模型。

随着时间的推移，信息会被插入或删除，数据库随之也发生了改变。特定时刻存储在数据库中的信息的集合称作数据库的一个实例。数据库的总体设计称作数据库模式。

在某个层次上修改模式而不影响较高一层模式的能力叫做数据独立性。有两个层次的数据独立性：物理数据独立性和逻辑数据独立性。

数据库模式通过一系列用数据定义语言（DDL）表达的定义来描述。DDL语句经过编译，产生存储在一个特殊文件中的一系列表，这个文件称作数据字典，因此数据字典中存储的是元数据。

数据操纵语言（DML）是使得用户可以访问和操纵数据的语言，主要有两种：过程化的DML和非过程化的DML。过程化的DML要求用户指明需要什么数据以及如何获得这些数据，非过程化的DML只要求用户指明需要什么数据，而不必指明如何获得这些数据。

事务管理器负责保证无论是否有故障发生，数据库都要处于一致的（正确的）状态。事务管理器还保证并发事务的执行互不冲突。

存储管理器是在数据库中存储的低层数据与应用程序及向系统提交的查询之间提供接口的程序模块。存储管理器负责与磁盘上存储的数据进行交互。

习题

1.1 文件处理系统和DBMS的四个主要区别是什么？

1.2 本章讲述了数据库系统的一些主要优越性，它有哪两个不足之处？

1.3 说明物理数据独立性与逻辑数据独立性的区别。

1.4 列出数据库管理器的五个任务。对每个任务，说明当它不能被完成时会产生什么样的问题。

1.5 数据库管理员的五个主要作用是什么？

1.6 列举七种过程化的程序设计语言和两种非过程化程序设计语言，哪组更易学易用？解释你的答案。

1.7 列出为某个企业建立数据库的六个主要步骤。

1.8 考虑一个你最喜欢的程序设计语言中的一个 $n \times m$ 的二维整数数组，以此数组为例，说

明（a）数据三层抽象间的区别；（b）模式和实例的区别。

文献注解

Fry 和 Sibley［1976］以及 Sibley［1976］给出了关于 DBMS 的演变以及数据库技术发展的讨论。

CODASYL DBTG 报告中引入了数据抽象的三个层次。ANSI/SPARC 报告中提出了类似的建议，这个报告中抽象层次为内部层、概念层和外部层［ANSI 1975］。

实体－联系模型、面向对象模型、关系模型、网状模型和层次模型在本书的其他章节进行了讨论。本书后续章节中给出了更广泛的参考文献书目。

语义数据模型所基于的数据模型最初的发展同人工智能的研究关系密切。关于各种模型的讨论可以在 Roussopoulos 和 Mylopoulos［1975］、Wong 和 Mylopoulos［1977］、以及 Hammer 和 McLeod［1981］中找到。Hull 和 King［1987］以及 Peckham 和 Maryanski［1988］给出了关于语义数据模型的概述。

功能数据模型由 Sibley 和 Kerschberg［1977］提出，Shipman［1981］进行了扩充。人们提出了几种功能查询语言，包括 FQL［Buneman 和 Frankel 1979］和 DAPLEX［Shipman 1981］。

Batory 和 Gotlieb［1982］引入了一致化模型。March 等［1981］引入了框架存储结构。

Weldon［1981］中讨论了数据库管理问题。关于数据库系统的教科书还有 Abiteboul 等［1995］、Date［1995］、Elmasri 和 Navathe［1994］、O'Neil［1994］、以及 Ullman［1988］。

一些书中包含了数据库管理研究论文的汇集，其中包括 Bancilhon 和 Buneman［1990］，Date［1986，1990］、Kim 等［1995］、以及 Stonebraker［1994］。Silberschatz 等［1990，1996］给出了关于数据库管理已有成果和未来研究方向预测的一个综合评述。

第2章　实体-联系模型

实体-联系（E-R）数据模型基于对现实世界的这样一种认识：世界由一组称作实体的基本对象及这些对象间的联系组成。此模型通过允许对企业模式进行定义来帮助数据库的设计，企业模式代表了数据库的全局逻辑结构。E-R模型是一种语义模型，模型的语义方面主要体现在模型力图去表达数据的意义。E-R模型在将现实世界中事实的含义和相互关联映射到概念模式方面非常有用，因此，许多数据库设计工具都利用了E-R模型的概念。

2.1　基本概念

E-R数据模型所采用的三个主要概念是：实体集、联系集和属性。

2.1.1　实体集

实体是现实世界中可区别于其他对象的“事件”或“物体”，例如，企业中的每个人都是一个实体。每个实体有一组性质，其中一部分性质的取值可以唯一地标识实体，例如，社会保障号677-89-9011唯一地标识了企业中的某个人。与此类似，贷款也可以被看作实体，而分支机构名称Perryridge和贷款流水号L-15一起唯一地标识了某个贷款实体。实体可以是实实在在的，如人或书；也可以是抽象的，如贷款、假期或概念。本书中，为简单起见，假设处理的事务都发生在美国，这样每个个人实体都具有社会保障号。我们将在2.4.1节讨论关于选择唯一标识符的话题。

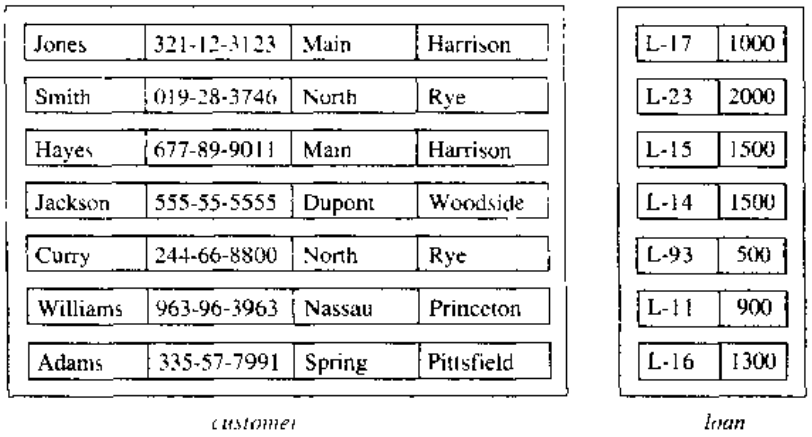
实体集是具有相同类型及相同性质（或属性）的实体集合，例如，某个银行的所有客户的集合可被定义为实体集 *customer*。同样，实体集 *loan* 表示某个银行所发放的所有贷款的集合。组成实体集的各实体称作实体集的外延，因此，所有银行客户是实体集 *customer* 的外延。

实体集可以相交。例如，可以定义银行所有员工的实体集 *employee* 和所有客户的实体集 *customer*，而一个 *person* 实体可以是 *employee* 实体，也可以是 *customer* 实体，可以既是 *employee* 实体又是 *customer* 实体，也可以都不是。

实体通过一组*属性*来表示，属性是实体集中每个成员具有的描述性性质。将一个属性赋予某实体集表明数据库为实体集中每个实体存储相似信息，但对每个属性来说，各实体有自己的属性值。实体集 *customer* 可能具有属性 *customer-name*、*social-security*、*customer-street* 和 *customer-city*；实体集 *loan* 可能具有属性 *loan-number* 和 *amount*。每个属性有其所允许的值的集合，称为该属性的域或值集。属性customer-name的域可能是某个长度的所有字符串的集合，同样，属性 *loan-number* 的域可以是所有正整数的集合。

由此，数据库包括一组实体集，每个实体集中包括一些相同类型的实体。图2-1所示为银行数据库的一部分，其中有两个实体集：*customer* 和 *loan*。

形式化地说，实体集的属性是将实体集映射到域的函数。由于一个实体集可能有多个属性，每个实体可以用（属性，数据值）对构成的集合来表示，对应实体集的每个属性有一个（属性，数据值）对。例如，某个 *customer* 实体可以用集合 {（*name*，Hayes），（*social-security*，677-89-9011），（*customer-street*，Main），（*customer-city*，Harrison）} 来描述，表示该实体描述了



Jones	321-12-3123	Main	Harrison
Smith	019-28-3746	North	Rye
Hayes	677-89-9011	Main	Harrison
Jackson	555-55-5555	Dupont	Woodside
Curry	244-66-8800	North	Rye
Williams	963-96-3963	Nassau	Princeton
Adams	335-57-7991	Spring	Pittsfield

customer

L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-93	500
L-11	900
L-16	1300

loan

图 2-1　实体集 *customer* 和实体集 *loan*

一个叫 Hayes 的人，他的社会保障号为 677-89-9011，居住在 Harrison 市的 Main 街。从这里可以看出抽象模式与作为建模对象的现实世界的事实间的一致性。描述实体的属性值是数据库中所存储数据的重要组成部分。

E-R 模型中的属性可以按照如下的属性类型来划分。

• 简单属性和复合属性。此例中，迄今为止出现的属性都是简单属性，也就是说，它们不能再划分为更小的部分。而复合属性可以再划分为更小的部分（即划分为别的属性），例如，*customer-name* 可被设计成包括 *first-name*、*middle-initial* 和 *last-name* 的成分属性。如果用户希望在某些时候访问整个属性，而在另一些时候访问属性的一个成分，那么在设计模式中使用复合属性是一个很好的选择，例如可用由属性 *street*、*city*、*state* 和 *zip-code* 构成的复合属性 *customer-address* 替换实体集 *customer* 中原属性 *customer-street* 和 *customer-city*。复合属性可将相关属性聚集起来，使模型更清晰。

注意，复合属性可以是层次的。再来看复合属性 *customer-address* 的例子，其成分属性 *street* 可以进一步划分为 *street-number*、*street-name* 和 *apt-number*。关于实体集 *customer* 的复合属性的这些例子如图 2-2 所示。

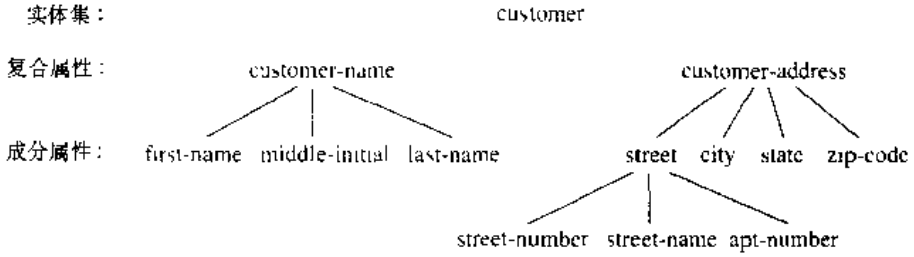


图 2-2　复合属性 *customer-name* 和 *customer-address*

• 单值属性和多值属性。我们的例子中所定义的属性对一个特定实体都只有单独的一个值。例如，对某个特定的贷款实体而言，属性 *loan-number* 只对应于一个贷款号码，这样的属性叫单值属性。而在某些情况下对某个特定实体而言，一个属性可能对应于一组值。例如假设实体集 *employee* 有一个属性 *dependent-name*，由于每个员工可以有 0 个、1 个或多个亲属，因此，该实体集中不同的员工实体在属性 *dependent-name* 上有不同数目的值，这样的属性称为多值属性。需要时，可以对某个多值属性的取值数目进行上、下界的限制。例如，银行可能将一个客户的地址限制在两个以内，这个限制表明实体集 *customer* 的属性 *customer-address* 的值可以是 0－2 个。

• NULL 属性。当实体在某个属性上没有值时使用 *null* 值。例如，如果某个员工没有亲属，那么该员工的 *dependent-name* 属性值将是 *null*，表示“无意义”。*null* 还可以用于当值未知时。未知的值可能是缺失的（即值存在，只不过我们没有该信息）或不知道的（我们并不知道该值是否真的存在）。例如，如果某个客户的 *social-security* 值为 *null*，则认为该值是缺失的，因为它对报税是必需的。而如果 *apt-number* 值为 *null*，则可能意味着地址中不含房间号，或者房间号存在但我们不知道，或者我们不知道房间号是否是客户地址的一部分。

• 派生属性。这类属性的值可以从别的相关属性或实体派生出来。例如，假设实体集 *customer* 有一个属性 *loans-held*，表示客户从银行获得了多少次贷款。可以通过计算与该客户相联系的所有贷款实体的数目来得到这个客户的属性 *loans-held* 的值。又如，假设实体集 *employee* 具有相关联的两个属性 *start-date* 和 *employment-length*，分别表示员工开始在银行工作的日期和员工在银行工作的总时间，那么 *employment-length* 的值可以由 *start-date* 和当前日期得到。在这里，*start-date* 可称为基属性，或存储属性。

银行的数据库可能包括多个不同的实体集。例如，除了对客户和贷款进行跟踪外，银行还保存帐户信息，帐户用实体集 *account* 来表示，它包括属性 *account-number* 和 *balance*。另外，如果银行有不同的分支机构，那么可能还要保存关于银行所有分支机构的信息，每个实体集 *branch* 可以通过属性 *branch-name*、*branch-city* 和 *assets* 来描述。

本章将处理不同类型的实体集。为了避免混淆，我们使用互不重复的属性名。在必要时将定义一些新的实体集。

2.1.2　联系集

联系是多个实体间的相互关联。例如，可以定义将客户 Hayes 和贷款 L-15 相关联，这一联系指明 Hayes 是拥有贷款 L-15 的客户。

联系集是同类联系的集合。规范地说，联系集是 n（$n \geqslant 2$）个实体集上的数学关系，这些实体集不必互异。如果 E_1，E_2，…，E_n 为 n 个实体集，那么联系集 R 是

$$\{(e_1, e_2, \cdots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \cdots, e_n \in E_n\}$$

的一个子集，而（e_1，e_2，…，e_n）是一个联系。

考虑一下图 2-1 中的两个实体集 *customer* 和 *loan*，我们定义联系集 *borrower* 来表示客户与其银行贷款之间的联系，这一联系如图 2-3 所示。

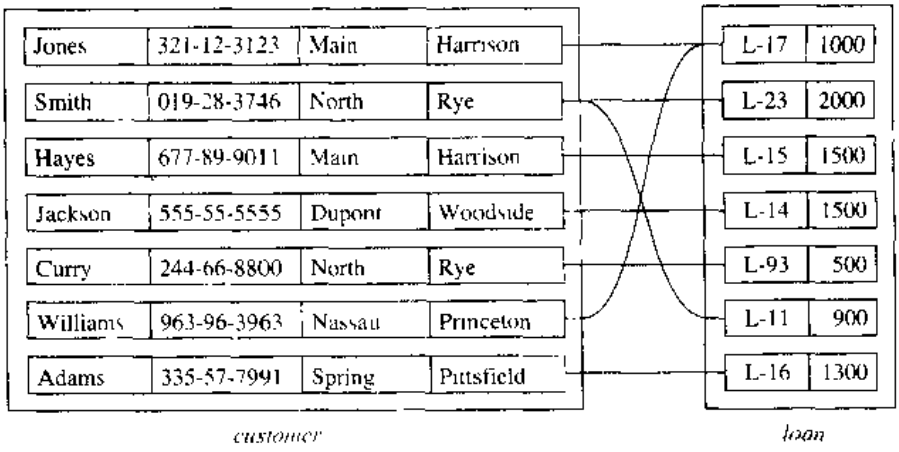


图 2-3　联系集 *borrower*

再看另一个例子。在实体集 *loan* 和 *branch* 间，可以定义联系集 *loan-branch* 来表示银行贷

款和维护此贷款的分支机构之间的联系。

实体集之间的关联称为参与，也就是说，实体集 E_1，E_2，…，E_n 参与联系集 R。E-R 模式中的一个联系实例表示所模拟的现实世界的命名实体间存在联系。例如，一个 *customer* 实体 Hayes（社会保障号 677-89-9011）和一个实体 *loan* L-15 参与到 *borrower* 的一个实例中。这一联系实例表示在现实世界中，社会保障号为 677-89-9011、名字叫 Hayes 的人具有编号为 L-15 的贷款。

实体在联系中的作用称为实体的角色。由于参与一个联系的实体集通常是互异的，因而角色是隐含的并且常常不声明。但是，当联系的含义需要解释时角色还是很有用的，这主要是当参与联系集的实体集并非互异的时候，也就是说，同一个实体集在一个联系集中参与的次数大于一次，且每次参与具有不同的角色。在这类联系集（有时称作自环的联系集）中，有必要用显式角色名来定义一个实体参与联系实例的方式。例如，假设实体集 *employee* 记录关于银行所有员工的信息，我们可能用联系集 *works-for* 来对有序的 *employee* 实体对进行建模。每对实体中的第一个员工具有经理的角色，而第二个员工具有职员的角色。按照这种方式，所有的 *works-for* 联系通过（经理，职员）对来描述，而不描述成（职员，经理）对。

联系也可能具有描述性属性。让我们看一看实体集 *customer* 和 *account* 之间的联系集 *depositor*。可以将属性 *access-date* 与该联系关联起来，以表示客户访问一个帐户的最近日期。客户 Jones 对应的实体和帐户 A-217 对应的实体之间的联系 *depositor* 可以通过 {（*access-date*，23 May 1996）} 来描述，表示 Jones 访问帐户 A-217 的最近日期为 1996 年 5 月 23 日。

联系集 *borrower* 和 *loan-branch* 是二元联系集的例子。二元联系集是涉及两个实体集的联系集，数据库系统中的大多数联系集都是二元的。但是，偶尔有联系集中涉及的实体集会多于两个。例如，可以把联系集 *borrower* 和 *loan-branch* 进行合并，形成三元联系集 CLB，其中涉及实体集 *customer*、*loan* 和 *branch*。这样，对应于客户 Hayes、贷款 L-15 和 Perryridge 分支机构的三个实体之间的三元联系表明客户 Hayes 在 Perryridge 分支机构有贷款 L-15。

参与联系集的实体集的数目也称为联系集的度，二元联系集的度为 2，三元联系集的度为 3。

2.2　设计问题

实体集和联系集的概念并不精确，而且定义一组实体及它们的相互联系可以有多种不同的方式。本节讨论数据库 E-R 模式设计中的一些基本问题。设计过程将在 2.7.4 节中更详细地讨论。

2.2.1　用实体集还是用属性

来看一看具有属性 *employee-name* 和 *telephone-number* 的实体集 *employee*。显然电话可以作为一个单独的实体，它具有属性 *telephone-number* 和 *location*（电话所处的办公室）。如果我们赞成上述观点，那么上面的实体集 *employee* 就必须重新定义如下：

- 实体集 *employee* 具有属性 *employee-name*。
- 实体集 *telephone* 具有属性 *telephone-number* 和 *location*。
- 联系集 *emp-telephone* 表示员工及其电话间的联系。

那么，员工的这两个定义主要差别是什么呢？前一种情况隐含说明了每个员工只有一个电话号码与之相联系；而第二种情况则表明了每个员工可以有若干个电话号码（包括 0 个）与之相联系。因此，第二个定义比第一个更通用，而且可能更精确地反映了现实世界的情况。

即使每个员工正好就同一个电话号码相联系，当有多个员工共用一个电话时，第二个定义仍比第一个定义更确切。

但是，此种方法并不能运用于属性 *employee-name*。与电话不同，将 *employee-name* 单独作为一个实体很不具有说服力。因此，正确的做法是将 *employee-name* 作为实体集 *employee* 的一个属性。

由此自然就产生两个问题：什么可作为属性？什么可作为实体集？很不幸，对这两个问题并不能简单地回答。它们的主要区别依赖于被建模的现实世界事实的结构，以及所讨论的属性的相关语义。

2.2.2　用实体集还是用联系集

一个对象最好表述为实体集还是联系集并不总是非常清楚的。在 2.1.1 节中，我们将银行贷款作为一个实体来建模。另一种方法是，不将贷款作为一个实体，而将其作为客户和银行分支机构之间的一个联系，这一联系具有描述性属性 *loan-number* 和 *amount*，每次贷款都用客户和银行分支机构间的一个联系来表示。

如果每笔贷款正好为一个客户所有，并且正好同一个分支机构相联系，那么可以发现用联系来表示贷款是能满足设计要求的。但是，采用这样的设计，就不能很方便地表示几个客户共有一笔贷款的情况，必须为共有贷款的每个持有人分别定义一个联系，于是，不得不在每个这样的联系中复制描述性属性 *loan-number* 和 *amount* 的值。当然，每个这样的联系必须有相同的 *loan-number* 和 *amount* 值。由复制产生的问题有两个：1）数据多次存储，浪费存储空间；2）更新很可能使数据处于不一致的状态，即两个联系中应该具有相同值的属性具有了不同的值。如何避免这种复制问题在规范化理论中进行了形式化处理，我们将在第 7 章讨论。属性 *loan-number* 和 *amount* 的复制问题在 2.1.1 节最初的设计中是不存在的，因为在那里 *loan* 是一个实体集。

在确定用实体集还是联系集时可采用的一个原则是，当描述发生在实体间的行为时采用联系集。这一方法在决定将某些属性表述为联系时是否会更确切时也很有用。

2.2.3　二元联系集与 n 元联系集

一个非二元的（n 元，$n>2$）联系集总可以用一组不同的二元联系集来替代。为简单起见，考虑一个抽象的三元（$n=3$）联系集 R，它将实体集 A、B 和 C 联系起来。用实体集 E 替代联系集 R，并建立三个联系集：

- R_A。联系 E 和 A。
- R_B。联系 E 和 B。
- R_C。联系 E 和 C。

如果联系集 R 有属性，那么将这些属性赋给实体集 E，否则，为 E 建立一个特殊的标识属性（因为每个实体集都应该至少有一个属性，以区别实体集中的各个成员）。针对联系集 R 中的每个联系 (a_i, b_i, c_i)，在实体集 E 中创建一个新的实体 e_i，然后，在三个新联系集中，分别插入新联系如下：

- 在 R_A 中插入 (e_i, a_i)。
- 在 R_B 中插入 (e_i, b_i)。
- 在 R_C 中插入 (e_i, c_i)。

可以将这一过程直接推广到 n 元联系集的情况。因此，概念上可以限制 E-R 模型中只包含二元联系集，然而，这种限制并不总是令人满意的。

• 对于为表示联系集而创建的实体集，可能不得不为其创建一个标识属性。该标识属性和所需附加的那些联系集增加了设计的复杂程度以及对总的存储空间的需求（正如在 2.9 节将看到的一样）。

• n 元联系集可以更清晰地表示出几个实体集参与到一个联系集中，而在对应的使用二元联系的设计中，难以体现这样的参与性约束。

2.3 映射约束

E-R 模式可以定义数据库中的内容必须满足的某些约束。本节将讨论映射的基数以及存在依赖这两类最重要的约束。

2.3.1 映射的基数

映射的基数，或基数比例，指明通过一个联系集能同另一实体相联系的实体数目。

映射的基数在描述二元联系集时非常有用，尽管有时也用于描述涉及多个实体集的联系集。本节只讨论二元联系集的情况。

对于实体集 A 和 B 之间的二元联系集 R 来说，映射的基数必然是以下情况之一：

• 一对一。A 中的一个实体至多同 B 中的一个实体相联系，B 中的一个实体也至多同 A 中的一个实体相联系，如图 2-4a 所示。

• 一对多。A 中的一个实体可以同 B 中的任意数目的实体相联系，而 B 中的一个实体至多同 A 中的一个实体相联系，如图 2-4b 所示。

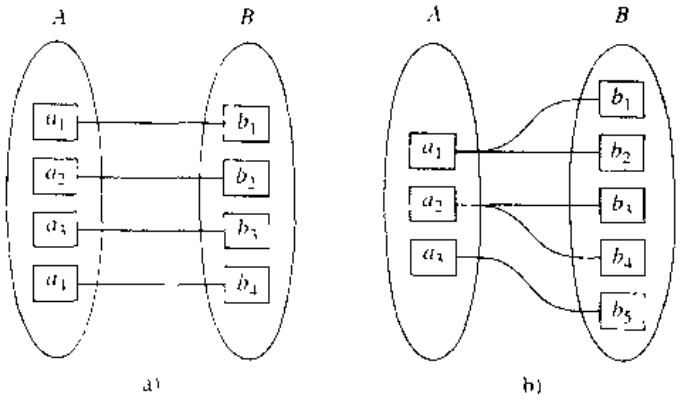


图 2-4　映射的基数

a) 一对一　b) 一对多

• 多对一。A 中的一个实体至多同 B 中的一个实体相联系，而 B 中的一个实体可以同 A 中任意数目的实体相联系，如图 2-5a 所示。

• 多对多。A 中的一个实体可以同 B 中任意数目的实体相联系，B 中的一个实体也可以同 A 中任意数目的实体相联系，如图 2-5b 所示。

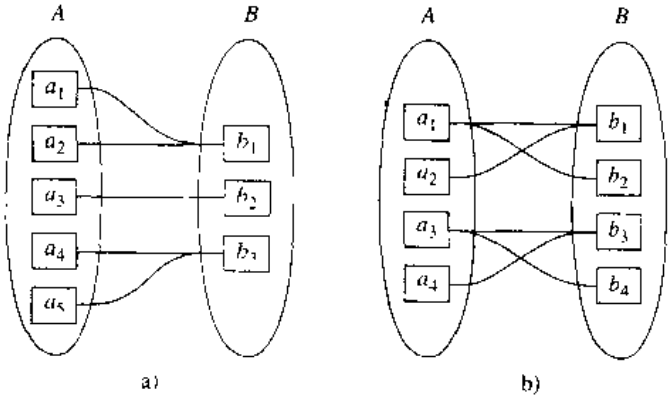


图 2-5　映射的基数

a) 多对一　b) 多对多

显然，某个联系集正确的映射基数应是什么依赖于该联系集用来作为建模对象的现实世界情况。

作为例子，我们来看一看联系集 *borrower*。如果某个银行中，一笔贷款只能属于一个客户，而一个客户可有多笔贷款，那么 *customer* 到 *loan* 的联系集就是一对多的，这类联系如图 2-3 所示。如果一笔贷款可属于多个客户（如贷款可以被多个商业伙伴共有），那么此联系集就是多对多的。

联系的基数比例可能影响联系属性所处的位置。一对一或一对多联系集的属性可以放到参

与的实体集之中，而不是放到联系集中。例如，将 *depositor* 定义为一对多的联系集，一个客户可以有多个帐户，而一个帐户只能为一个客户所有。在这种情况下，*access-date* 属性可以放到实体集 *account* 中，如图 2-6 所示。为简单起见，图中只列出了两个实体集的部分属性。由于每个 *account* 实体至多同 *customer* 的一个实例参与到联系中，因而将属性 *access-date* 放到实体集 *account* 中和将属性 *access-date* 放到联系集 *depositor* 中具有相同的含义。一对多联系集的属性可以放到联系中“多”的一方的实体集中，而对一对一的联系集而言，联系的属性可以放到参与联系的任何一个实体集中。

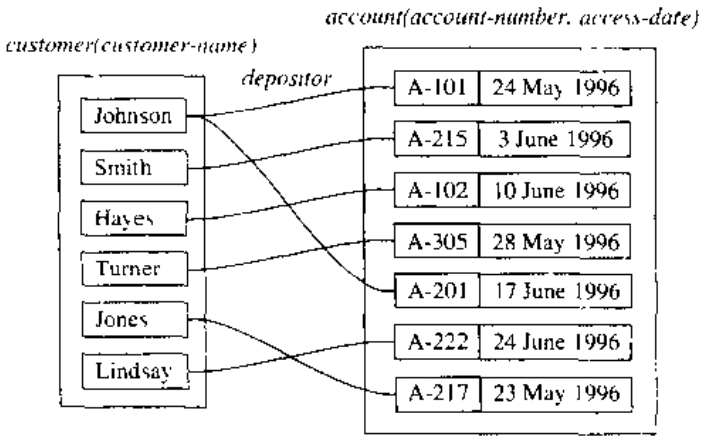


图 2-6　*Access-date* 作为实体集 *account* 的属性

设计时将描述性属性作为联系的属性还是实体的属性这一决定应该反映所模拟的事实的特点。设计者可以选择保留 *access-date* 作为 *depositor* 的属性，用以显式地表明访问发生在实体集 *customer* 和 *account* 的交互点上。

属性位置的选择在多对多联系集中体现得更清楚。仍是这个例子，将 *depositor* 定义为或许更符合现实情况的多对多联系集，表明一个客户可有一个或多个帐户，而一个帐户可以为一个或多个客户所拥有。如果想要表达某个客户最近一次访问某个帐户的日期，*access-date* 则必须作为联系集 *depositor* 的属性，而不是任何一个参与此联系集的实体集的属性。而如果将 *access-date* 作为 *account* 的属性，对一个共有的帐户，就不能确定是哪个客户进行了最近一次访问。当一个属性是由参与的实体集联合确定而不是由单独的某个实体集确定时，该属性就必须放到多对多联系集中。将 *access-date* 作为联系集的属性，如图 2-7 所示。为使图示简单，在图中再次只给出了两个实体集的部分属性。

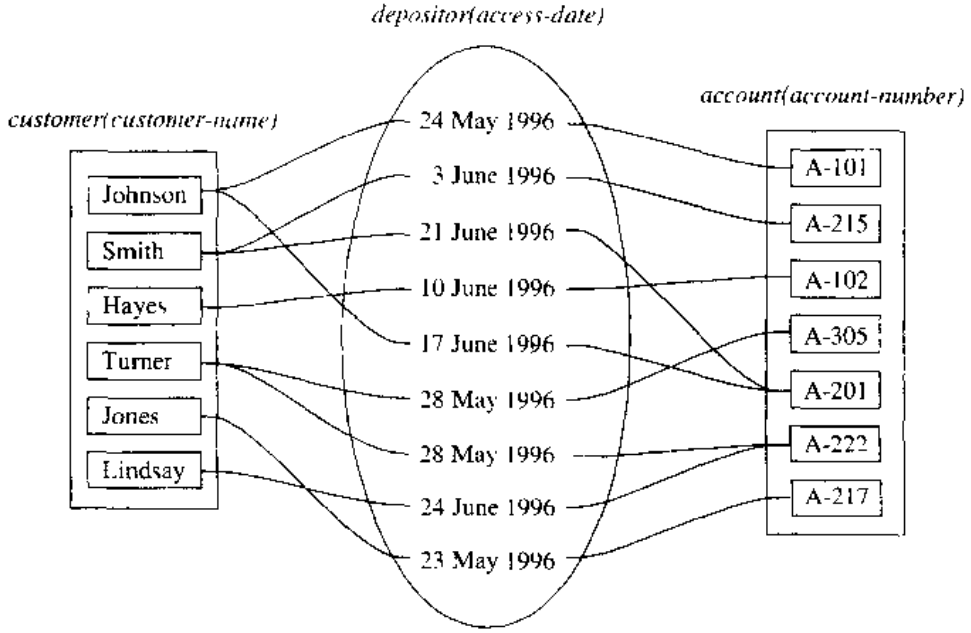


图 2-7　*Access-date* 作为 *depositor* 联系集的属性

2.3.2 存在依赖

另一类重要的约束是存在依赖。具体地说，如果实体 x 的存在依赖于实体 y 的存在，那么就说 x 存在依赖于 y。在操作上，如果 y 被删除，那么 x 也要被删除。实体 y 称作支配实体，实体 x 称作从属实体。

作为例子，让我们看一看实体集 *loan* 和实体集 *payment*，后者记载同某笔贷款相联系的所有付款的信息。实体集 *payment* 通过属性 *payment-number*、*payment-date* 和 *payment-amount* 来描述。在这两个实体集间建立一个联系集 *loan-payment*，该联系集从 *loan* 到 *payment* 是一对多的。每个 *payment* 实体必须同一个 *loan* 实体相联系，如果一个 loan 实体被删除，那么所有与之相联系的 *payment* 实体也必须被删除。相反地，*payment* 实体从数据库中删除却可以不影响任何一个 *loan* 实体。因此在联系集 *loan-payment* 中，实体集 *loan* 是支配的，而实体集 *payment* 是从属的。

如果实体集 E 中的每个实体都参与到联系集 R 的至少一个联系中，称实体集 E 全部参与联系集 R。如果实体集 E 中只有部分实体参与到联系集 R 的联系中，称实体集 E 部分参与联系集 R。全部参与同存在依赖紧密相关。例如，由于每个 *payment* 实体必须通过 *loan-payment* 联系同某个 *loan* 实体相联系，因而实体集 *payment* 对联系集 *loan-payment* 的参与是全部的。相反地，一个人不管是否从银行贷款，都可能成为银行的客户，因此，很可能只有 *customer* 实体的一个子集同实体集 *loan* 相联系，故 *customer* 对联系集 *borrower* 的参与是部分的。

2.4 码

关于给定实体集中的实体或给定联系集中的联系如何相互区别的声明是非常重要的。从概念上来说，各个实体或联系是互异的，但从数据库的观点来看，它们的区别必须用其属性来表明。码的概念使得我们可以进行这样的区别。

2.4.1 实体集

*超码*是一个或多个属性的集合，这些属性的组合可以使我们在一个实体集中唯一地标识一个实体。例如，实体集 *customer* 的 *social-security* 属性足以将不同客户区分开来，因此，*social-security* 是一个超码。同样，*customer-name* 和 *social-security* 的组合也是实体集 *customer* 的一个超码。*customer* 的 *customer-name* 不是超码，因为几个人很可能同名。

超码的概念并不足以帮助我们达到目的，正如我们看到的，超码中可能包含一些无关紧要的属性。如果 K 是一个超码，那么 K 的任意超集也是超码。我们通常只对这样的一些超码感兴趣：它们的任意真子集都不能成为超码。这样的最小超码称为*候选码*。

几个不同的属性集都可以做候选码的情况是存在的。假设 *customer-name* 和 *customer-street* 的组合足以区分实体集 *customer* 的各个成员，那么 {*social-security*} 和 {*customer-name*, *customer-street*} 都是候选码。虽然属性 *social-security* 和 *customer-name* 一起能区别 *customer* 实体，但它们的组合并不能成为候选码，因为单独的 *social-security* 属性已是候选码。

候选码的选择必须慎重，如我们所见，人名是不足以作为候选码的，因为可能有多个人重名。在美国，社会保障号可以作为候选码。由于非美国居民通常不具有社会保障号，企业可以产生自己的唯一标识符，如客户号、学号等；也可以使用某些属性的唯一组合来作为候选码。一种常用的方法是以姓名、生日及住址的组合来作为候选码，因为两个人在这些属性上的值都相等是几乎不可能的。

我们用主码来代表被数据库设计者选中的，用来在同一实体集中区分不同实体的候选码。码（主码、候选码和超码）是实体集的性质，而不是一个个实体的性质。实体集中的任意两个实体都不允许同时在码属性上具有相同的值。码的指定代表了被建模的现实世界中的约束。

2.4.2　联系集

实体集的主码使得我们可以将实体集中的不同实体区别开来。我们需要一种类似的机制来区别一个联系集中不同的联系。

假设 R 是一个涉及实体集 E_1，E_2，…，E_n 的联系集，而 *primary-key*（E_i）代表构成实体集 E_i 主码的属性集合。这里假设所有主码的属性名是唯一的（如果不是这样，可以采用适当的重命名机制）。联系集主码的构成依赖于同联系集相联系的属性的结构。

如果没有属性同联系集 R 相联，那么属性集合

$$primary\text{-}key(E_1) \cup primary\text{-}key(E_2) \cup \cdots \cup primary\text{-}key(E_n)$$

描述了集合 R 中的一个联系。

如果属性 a_1，a_2，…，a_m 同联系集 R 相联系，那么属性集合

$$primary\text{-}key(E_1) \cup primary\text{-}key(E_2) \cup \cdots \cup primary\text{-}key(E_n) \cup \{a_1, a_2, \cdots, a_m\}$$

描述了集合 R 中的一个联系。

在以上两种情况下，属性集合

$$primary\text{-}key(E_1) \cup primary\text{-}key(E_2) \cup \cdots \cup primary\text{-}kay(E_n)$$

构成联系集的一个超码。

联系的主码结构依赖于联系集映射的基数。作为例子，来看实体集 *customer* 和 *employee*，以及表示客户及其银行帐户负责人（一个 *employee* 实体）间联系的联系集 *cust-banker*。假设联系集是多对多的，而且有表示联系特征的属性 *type*（如贷款负责人或个人银行帐户负责人）与之相联系，那么 *cust-banker* 的主码由 *customer* 和 *employee* 的主码共同组成。但是，如果一个客户只能有一个银行帐户负责人，即联系集 *cust-banker* 是多对一的，则 *cust-banker* 的主码就是 *customer* 的主码。在一对一的联系中，可以使用两个主码中的任何一个来作联系集的主码。

2.5　实体 - 联系图

1.3 节曾简要介绍过数据库的全局逻辑结构可以通过 E-R 图作图形化表示。这种图形化表示技术的简单性及其图示清晰性是 E-R 模型被广泛使用的重要原因。E-R 图中包括如下几个主要构件：

- 矩形。表示实体集。
- 椭圆。表示属性。
- 菱形。表示联系集。
- 线段。将属性连接到实体集或将实体集连接到联系集。
- 双椭圆。表示多值属性。
- 虚椭圆。表示派生属性。
- 双线。表示一个实体全部参与到联系集中。

如图 2-8 所示，实体集属性中那些作为主码的一部分的属性以下划线标明。

来看一下图 2-8 所示的实体 - 联系图，该图包括了两个实体集 *customer* 和 *loan*，它们通过二元联系集 *borrower* 联系起来。同 *customer* 相联系的属性有 *customer-name*、*social-security*、

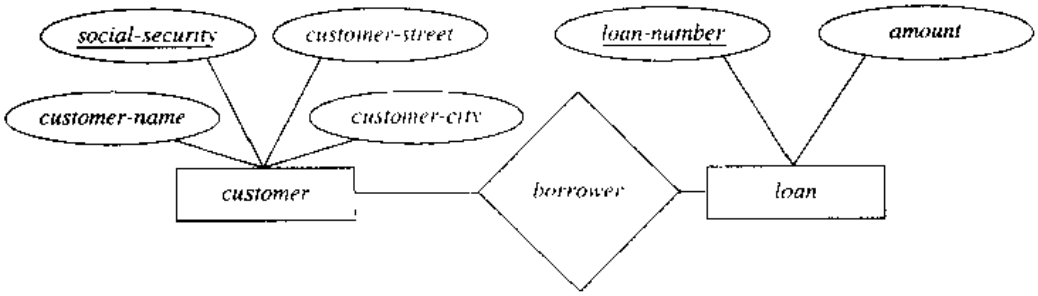
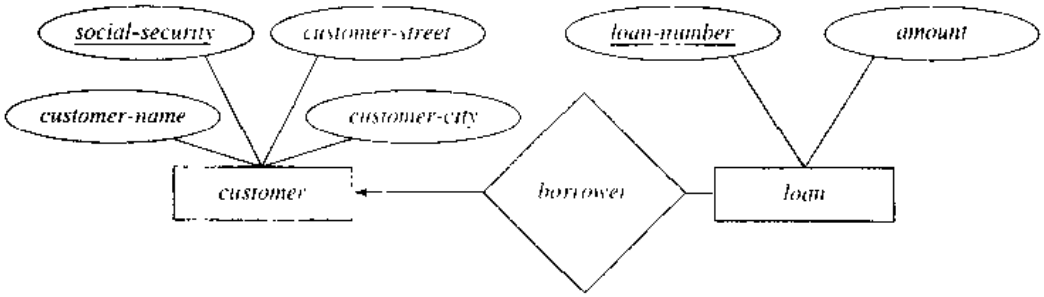


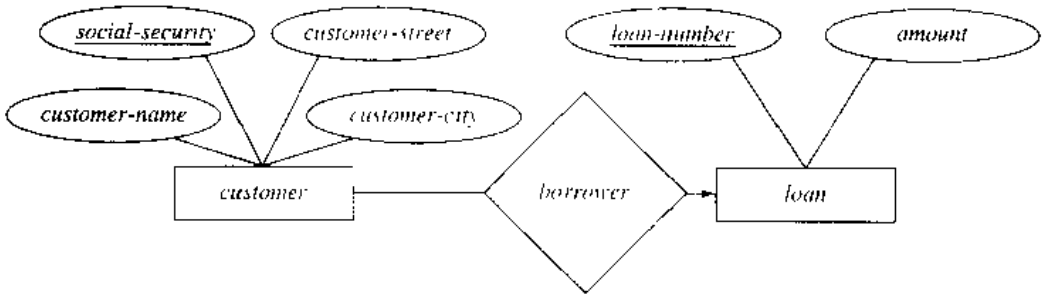
图 2-8　与客户和贷款相对应的 E-R 图

customer-street 和 *customer-city*，同 *loan* 相联系的属性有 *loan-number* 和 *amount*。

联系集 *borrower* 可以是多对多的、一对多的、多对一的或一对一的。为了将这些类型相互区别开来，在联系集和所讨论的实体集间或者用箭头（→），或者用线段（—），如图 2-9 所示。



a)



b)

图 2-9　联系

a) 一对多　b) 多对一

• 从联系集 *borrower* 到实体集 *loan* 的箭头表示 *borrower* 从 *customer* 到 *loan* 只能是一对一的或多对一的，而不能是多对多的或一对多的。

• 从联系集 *borrower* 到实体集 *loan* 的线段表示 *borrower* 从 *customer* 到 *loan* 是多对多的或一对多的。

回到图 2-8 中的 E-R 图，可以看到联系集 *borrower* 是多对多的。如果联系集 *borrower* 从 *customer* 到 *loan* 是一对多的，那么从 *borrower* 到 *customer* 的线就应该是箭头，箭头方向指向实体集 *customer*（图 2-9a）。同样，如果联系集 *borrower* 从 *customer* 到 *loan* 是多对一的，那么从 *borrower* 到 *loan* 的线就应该是箭头，箭头方向指向实体集 *loan*（图 2-9b）。最后，如果联系

集 *borrower* 是一对一的，那么从 *borrower* 出发的两条线都应该是箭头，一个箭头方向指向实体集 *loan*，另一个箭头方向指向实体集 *customer*（图 2-10）。

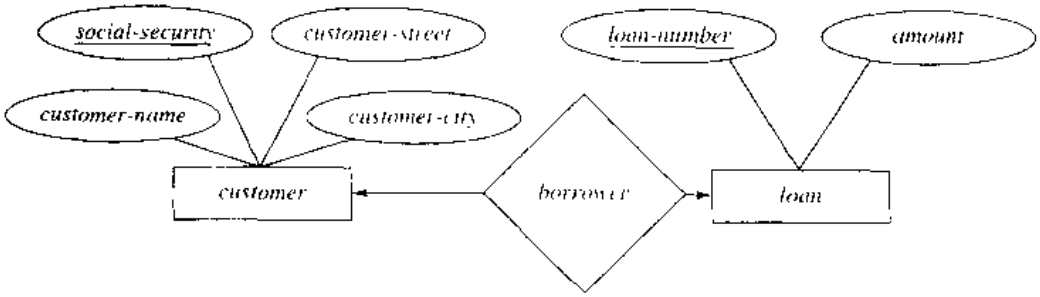


图 2-10　一对一联系

如果有某些属性同联系集相联系，那么也要将这些属性同该联系集连接起来。例如，在图 2-11 中，有一个描述性属性 *access-date* 同联系集 *depositor* 相联系，用以表示客户访问一个帐户的最近日期。

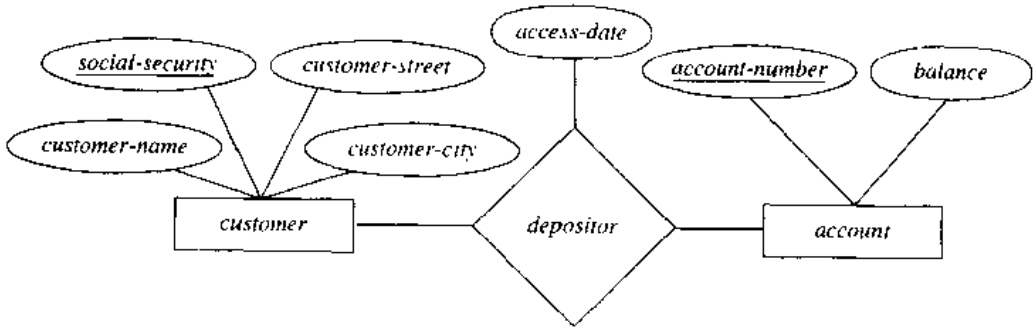


图 2-11　带有属性的联系集的 E-R 图

在 E-R 图中，通过在连接菱形和矩形的线上加标注来标识角色。图 2-12 给出了实体集 *employee* 和联系集 *works-for* 之间的角色标识 *manager* 和 *worker*。

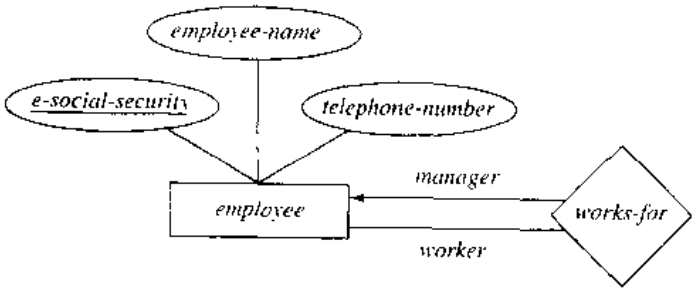


图 2-12　带有角色标识的 E-R 图

非二元的联系集在 E-R 图中也可以简单地表示。图 2-13 中实体集 *customer*、*loan* 和 *branch* 通过联系集 CLB 相互关联。该图表示，一个客户可以有多个贷款，一个贷款也可以属于多个不同的客户。该图中指向 *branch* 的箭头表明，每个客户-贷款对与一个确定的银行分支机构相联系。如果图中除了有指向 *branch* 的箭头外，还有箭头指向 *customer*，这时此图表明每一个贷款同一个确定的客户和一个确定的银行分支机构相联系。

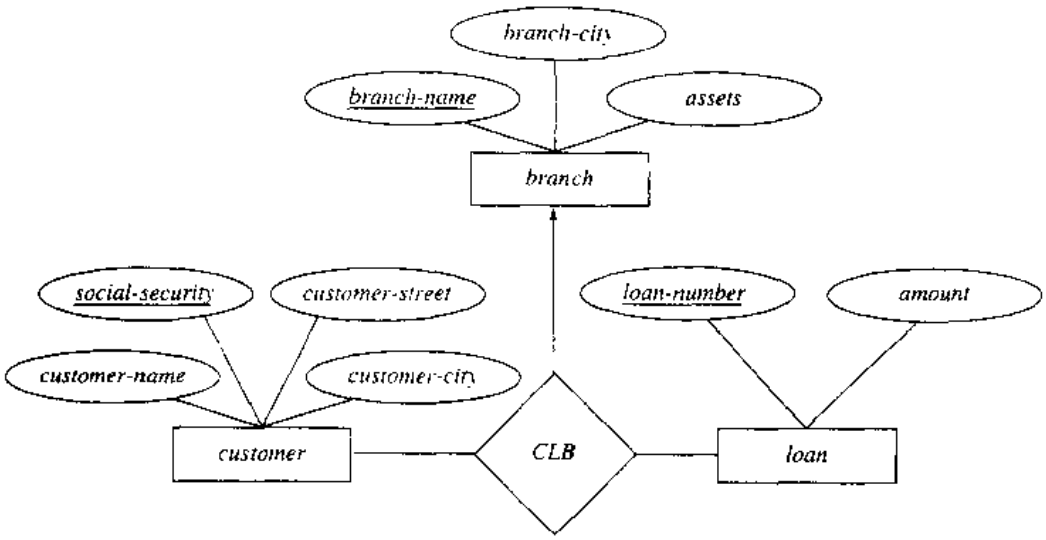


图 2-13　包含三元联系的 E-R 图

2.6　弱实体集

有些实体集的属性都不足以形成主码，这样的实体集称作*弱实体集*。与此相对，有主码的实体集称作*强实体集*。以实体集 *payment* 为例，该实体集具有属性 *payment-number*、*payment-date* 和 *payment-amount*。虽然各个 *payment* 实体互不相同，但不同贷款的付款却可能具有相同的付款号，因此，实体集 *payment* 没有主码，是一个弱实体集。弱实体集只有作为一对多联系的一部分才有意义，这时该联系集就应该不具有任何描述性属性，因为任何所需属性都可以同弱实体集相联系（参考 2.3.1 节关于将联系集属性移入参与实体集中的讨论）。

强实体集和弱实体集的概念是同 2.3.2 节引入的存在依赖相关的。强实体集的成员必然是支配实体，而弱实体集的成员是从属实体。

虽然弱实体集没有主码，仍需要用某种方法来区分该实体集中依赖于某个特定强实体的所有实体。弱实体集的*分辨符*是使得我们能进行这种区分的属性集合。例如，弱实体集 *payment* 的分辨符是属性 *payment-number*，因为对每笔贷款来说，付款号唯一标识了为该贷款而付的一笔款项。弱实体集的分辨符也称为该实体集的*部分码*。

弱实体集的主码由该弱实体集所存在依赖的强实体集的主码和该弱实体集的分辨符共同组成。例如弱实体集 *payment* 的主码是 {*loan-number*, *payment-number*}，其中 *loan-number* 标识了 *payment* 的支配实体，而 *payment-number* 区分同一个贷款的不同 *payment* 实体。

标识的支配实体集拥有它所标识的弱实体集。弱实体集与其拥有者之间的联系称为*标识性联系*。上例中，*loan-payment* 是 *payment* 的标识性联系。

E-R 图中弱实体集以双边框的矩形表示，而对应的标识性联系以双边框的菱形表示。图 2-14 中，弱实体集 *payment* 通过联系集 *loan-payment* 依赖于强实体集 *loan*。该图还示范了如何用双线表示全部参与——（弱）实体集 *payment* 全部参与联系 *loan-payment*，即每个付款都必须通过 *loan-payment* 同某个贷款相联系。从 *loan-payment* 指向 *loan* 的箭头表明，任何付款都针对单一的贷款。弱实体集的分辨符也以下划线标明，但是是虚线而不是实线。

弱实体集可以作为拥有者参与到与另一个弱实体集的标识性联系中。尽管一个弱实体集总

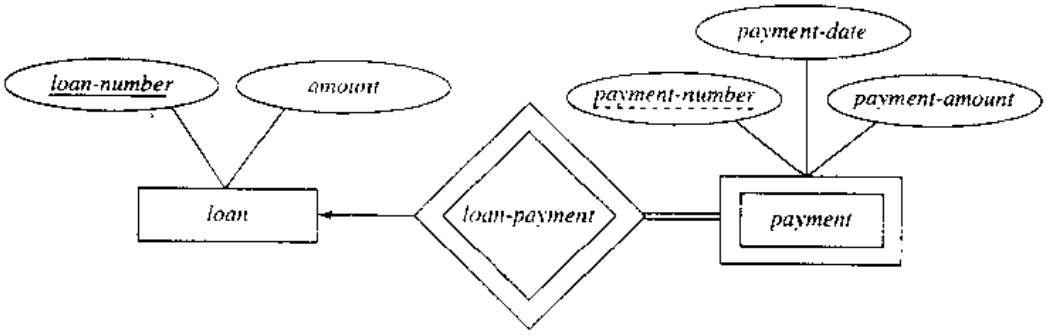


图 2-14　包含弱实体集的 E-R 图

是存在依赖于一个强实体集，但是一个存在依赖并不总会导致一个弱实体集——从属实体集也可以有主码。

在某些情况下，数据库设计者会选择用拥有者实体集的多值、复合属性来表示弱实体集。上例中，这样的表示需要实体集 *loan* 具有一个多值、复合属性 *payment*，它由 *payment-number*、*payment-date* 和 *payment-amount* 组成。如果弱实体集只参与标识性联系，而且其属性不多，那么在建模时将其表述为一个属性更恰当。相反地，如果弱实体集参与到标识性联系以外的联系中，或者其属性较多，则建模时将其表述为弱实体集更恰当。

2.7　扩展 E-R 特性

虽然基本的 E-R 概念已足以对大多数数据库特征建模，但可以通过对基本 E-R 模型作某些扩展来更恰当地表述数据库的某些方面。本节将讨论如下的扩展 E-R 特性：特殊化、概括、高层实体集、低层实体集、属性继承和聚集。

2.7.1　特殊化

实体集中可能包含一些子集，子集中的实体在某些方面区别于实体集中的其他实体。例如，实体集中的某个实体子集可能具有不被该实体集中所有实体所共享的一些属性。E-R 模型提供了表示这种与众不同的实体组的方法。

让我们来看一看实体集 *account*，它具有属性 *account-number* 和 *balance*。帐户可以进一步划分，一个帐户可能属于以下两类帐户之一：

- *savings-account*。
- *checking-account*。

每类帐户都可以通过由实体集 *account* 的所有属性和一些附加属性构成的属性集合来描述。例如，*savings-account* 实体通过属性 *interest-rate* 进一步描述，而 *checking-account* 实体通过属性 *overdraft-amount* 进一步描述。在实体集内部进行分组的过程称为特殊化。对 *account* 的特殊化使得我们可以基于帐户类型来区别不同帐户。

实体集可以根据多个差异特征来进行特殊化。比例中，帐户实体间的差异特征是帐户类型。另外，同时还可以根据帐户拥有者类型进行特殊化，其结果是产生实体集 *commercial-account* 和 *personal-account*。当一个实体集上进行了多次特殊化时，某个特定实体可能同时属于多个特殊化实体集。例如，某给定帐户可以既是个人帐户，又是支票帐户。

可以反复使用特殊化来精简设计模式。例如，银行可能提供以下三类支票帐户：

1）标准支票帐户：每月收取 $3.00 服务费并提供 25 张免费支票。对这些帐户，银行记录每个帐户的已填写支票数。

2）金支票帐户：要求至少有 $1000.00 的余额，付给 2% 的利息，提供无限制的免费支票。对这类帐户，银行监控最小余额和每月付给的利息。

3）高级支票帐户：针对 65 岁以上的客户，每月不收取任何服务费，并提供无限制的免费支票。对这类帐户，需要记录的是客户生日。

根据帐户类型对 *checking-account* 进行特殊化处理将产生如下实体集：

1）*standard*，具有属性 *num-checks*。

2）*gold*，具有属性 *min-balance* 和 *interest-payment*。

3）*senior*，具有属性 *date-of-birth*。

在 E-R 图中，特殊化通过标记为 ISA 的三角形构件来表示，如图 2-15 所示。标记 ISA 表示“is a（是一种）”，例如，储蓄帐户是一种（is a）帐户。ISA 联系也称父类-子类联系。高层实体和低层实体仍是普通实体的表示法，即以包含了实体集名称的矩形表示。

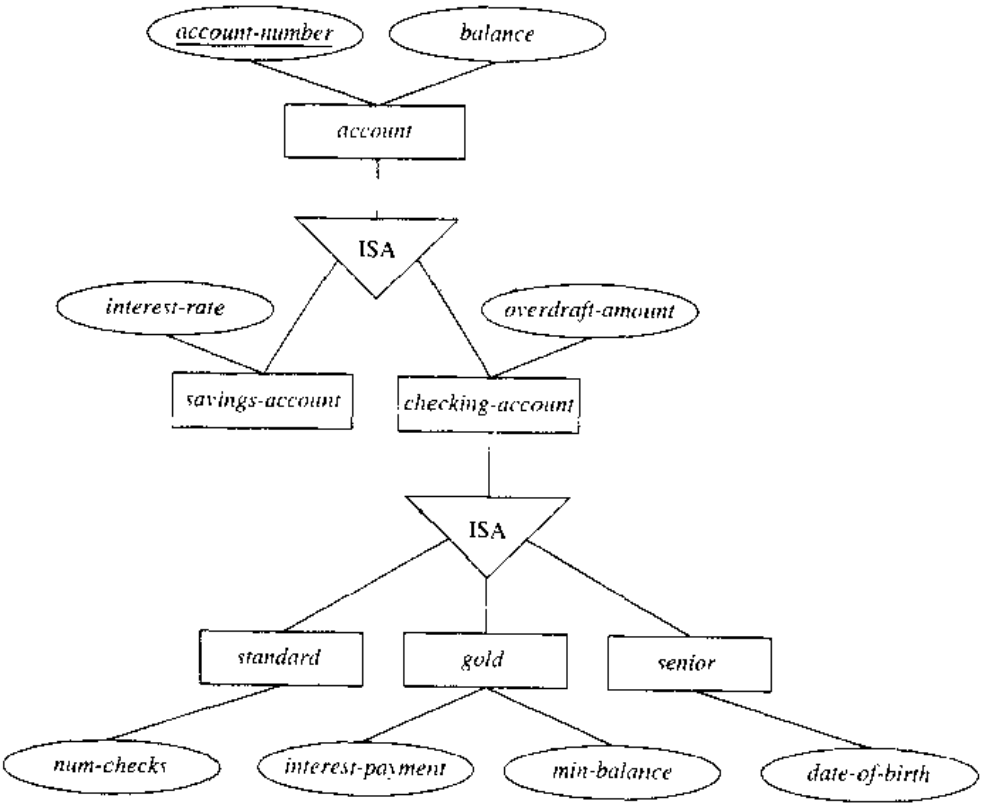


图 2-15　特殊化和概括

2.7.2　概括

对初始实体集求精，产生一系列不同层次的实体子集。在这个自顶向下的设计过程中，其区别被显式地表达出来。设计过程还可以是自底向上的，根据共同具有的特征，多个实体集综合成一个较高层的实体集。实际的设计中，数据库设计者可能一开始就定义了实体集 *checking-account* 和 *savings-account*，其中实体集 *checking-account* 具有属性 *account-number*、*balance* 和 *overdraft-amount*，而实体集 *savings-account* 具有属性 *account-number*、*balance* 和 *interest-rate*。

就所具有的属性而言，实体集 *checking-account* 和 *savings-account* 间存在共性，这种共性可以通过概括来表达，概括是高层实体集与一个或多个低层实体集间的包含关系。此例中，*account* 是高层实体集，而 *checking-account* 和 *savings-account* 是低层实体集。高层实体集与低层实体集也可以分别称作超类和子类，实体集 *account* 是子类 *checking-account* 和 *savings-account* 的超类。

对于所有实际应用来说，概括只不过是特殊化的逆过程。为企业设计 E-R 模型时，我们将配合使用这两个过程。在 E-R 图中，对概括和特殊化的表示不作区别。为了使设计模式充分体现数据库应用和数据库用户的要求，可能通过特殊化或概括产生新的实体层次。这两种方式的区别主要在于它们的出发点和总体目标。

特殊化从单一的实体集出发，通过创建不同的低层实体集来强调同一实体集中不同实体间的差异。低层实体集可以有不适用于高层实体集中所有实体的属性，也可以参与到不适用于高层实体集中所有实体的联系中。设计者采用特殊化的原因正是为了表达这些互不相同的特征。如果 *savings-account* 和 *checking-account* 没有各自特有的属性，就没有必要对实体集 *account* 进行特殊化了。

概括处理基于这样的认识：一些实体集具有共同的特征（即可以用相同的属性对它们进行描述，且它们都参与到相同的联系中）。概括是在这些实体集的共性的基础上将它们综合成一个高层实体集。概括用于强调低层实体集间隐藏于它们区别背后的相似性，同时由于去除了共同属性的重复出现，使用概括还可以做到经济的表示。

2.7.3　属性继承

特殊化和概括所产生的高层实体集和低层实体集的一个重要特性是属性继承。高层实体集的属性被低层实体集继承。例如，*savings-account* 和 *checking-account* 继承了 *account* 的属性，因此，*savings-account* 用属性 *account-number*、*balance* 和 *interest-rate* 来描述，而 *checking-account* 用属性 *account-number*、*balance* 和 *overdraft-amount* 来描述。低层实体集（或子类）同时还继承参与其高层实体集所参与的那些联系集。*Savings-account* 和 *checking-account* 都参与到联系集 *depositor* 中。属性继承作用于低层实体集的所有联系中。*Standard*、*gold* 和 *senior* 低层实体集继承了 *checking-account* 和 *account* 两者的属性以及对联系的参与。

对 E-R 图的某个给定部分来说，不管是通过特殊化还是通过概括得到的，其结果都是一样的：

- 同高层实体集相联系的所有属性和联系也适用于它的所有低层实体集。
- 低层实体集特有的性质仅仅适用于某个特定的低层实体集。

在后面的叙述中，虽然常常只说概括，但所讨论的性质也是特殊化过程所具有的。

图 2-15 所示为实体集的层次结构。图中，*checking-account* 是 *account* 的低层实体集，同时又是实体集 *standard*、*gold* 和 *senior* 的高层实体集。在层次结构中，给定实体集作为低层实体集只能参与到一个 ISA 联系中。如果一个实体集作为低层实体集能参与到多个 ISA 联系中，这时产生的结构就称为格。

2.7.4　约束设计

为了更准确地对企业建模，数据库设计者可能选择对特定的概括加上某些约束。一类约束

用来确定哪些实体能成为给定低层实体集的成员。成员资格可以是下列中的一种：

• 条件定义的。在条件定义的低层实体集中，成员资格的确定基于实体是否满足一个显式的条件或谓词。例如，假设高层实体集 *account* 具有属性 *account-type*，则所有实体都根据 *account-type* 属性进行评估。只有满足条件 *account-type* = "savingsaccount" 的实体才可以属于 *savings-account* 低层实体集，而所有满足条件 *account-type* = "checkingaccount" 的实体则都属于 *checking-account* 低层实体集。由于所有低层实体都基于同一属性（在这里是 *account-type*）进行评估，因而这种类型的概括称作是属性定义的。

• 用户定义的。用户定义的低层实体集不是通过成员资格条件来限制，而是由数据库用户将实体指派给某个实体集。例如，假设银行员工在三个月的雇佣期后将被分配到四个工作组中的一个，这些工作组用高层实体集 *employee* 的四个低层实体集来表示。一个员工被分配到哪个工作组并不能根据某个明确定义的条件自动得出，工作组的分配由做决定的用户的个人观点所确定，并通过将一个实体加入某个实体集的操作来实现。

第二类约束用来确定同一个概括中，一个实体是否可以属于多个低层实体集。低层实体集可以是下述情况之一：

• 不相交的。不相交约束要求一个实体至多属于一个低层实体集。上述例子中，一个 *account* 实体在属性 *account-type* 上只能满足一个条件；一个实体可以是储蓄帐户或支票帐户，但不能既是储蓄帐户又是支票帐户。

• 有重叠的。在有重叠的概括中，同一实体可以同时属于同一概括的多个低层实体集。再来看员工工作组的例子，并假设某些管理者参加到多个工作组中，在这种情况下给定实体就可以出现在多个工作组实体集中。

缺省情况下，低层实体是可以相互重叠的，而不相交约束必须显式地加到概括（或特殊化）中。

最后一类约束是对概括的全部性约束，用来确定高层实体集中的一个实体是否必须属于某个概括的至少一个低层实体集。这种约束可以是下述情况之一：

• 全部的。每个高层实体必须属于一个低层实体集。

• 部分的。允许一些高层实体不属于任何低层实体集。

对于 *account* 的概括是全部的。所有帐户实体都要么是一个储蓄帐户，要么是一个支票帐户。由于通过概括产生的高层实体集通常只包括低层实体集中的实体，因而对于概括产生的高层实体集来说，其全部性约束一般是全部的。如果全部性约束是部分的，高层实体就可以不出现在任何低层实体集中。工作组实体集即是部分特殊化的例子，由于员工在工作三月后才分配到某个工作组中，某些 *employee* 实体可以不属于任何低层工作组实体集。

工作组实体集是 *employee* 的部分的、可重叠的特殊化，而从 *checking-account* 和 *savings-account* 到 *account* 的概括是全部的、不相交的。尽管如此，全部性约束和不相交约束彼此并没有依赖关系，约束的模式也可以是部分-不相交的或全部-可重叠的。

可以看出，由于对概括或特殊化运用了某些约束，因而可能产生对某些插入和删除的需要。例如，当全部性约束是全部的时，一个实体被插入到高层实体集中，那么它至少同时还必须被插入到一个低层实体集中。在条件定义的约束中，所有满足条件的高层实体必须被插入到相应的低层实体集中。此外，从高层实体集删除的实体也必须从所有它所属于的相应低层实体集中删除。

2.7.5　聚集

E-R 模型的一个局限性在于它不能表达联系间的联系。为了说明这种结构的必要性，再来看描述客户及其贷款信息的数据库。假设每个客户-贷款对可能会有某个银行员工作为此对的贷款负责人。使用基本的 E-R 建模结构，得到图 2-16。看起来似乎联系集 *borrower* 和 *loan-officer* 可以合并成一个联系集，但是，我们不应这样做，因为这会模糊该模式的逻辑结构。如果将联系集 *borrower* 和 *loan-officer* 合并，则表明每个客户-贷款对都必须分配有贷款负责人，而实际情况并非如此。将 *borrower* 与 *loan-officer* 分别作为联系集则可以解决这样的问题。

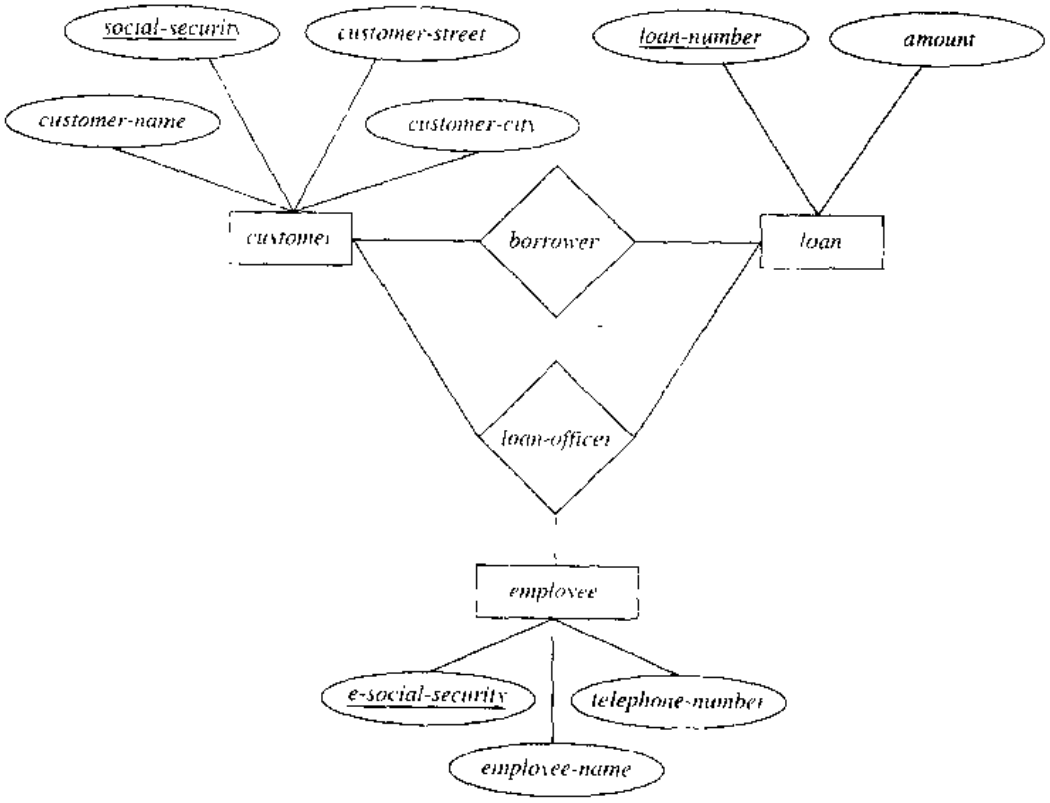


图 2-16　包含冗余联系的 E-R 图

然而，按照这种方法产生的图存在冗余信息，因为每个 *loan-officer* 中的客户-贷款对也在 *borrower* 中。如果贷款负责人是一个值而不是一个 *employee* 实体，可以将 *loan-officer* 作为联系集 *borrower* 的一个多值属性，但这样做使得某些信息更难获得（从逻辑上看和从执行开销上看），例如当要找某个员工负责处理的客户-贷款对时。由于贷款负责人是 *employee* 实体，这种做法就更无论如何也不可行了。

对类似上述情况建模的最好办法是使用聚集。聚集是一种抽象，通过这种抽象，联系被当作高层实体来看待。因此，在这个例子中，将联系集 *borrower* 以及实体集 *customer* 和 *loan* 看成一个称作 *borrower* 的高层实体集，并且可以像对待任何别的实体集那样对待这个实体集。图 2-17 所示为聚集的一种常用表示。

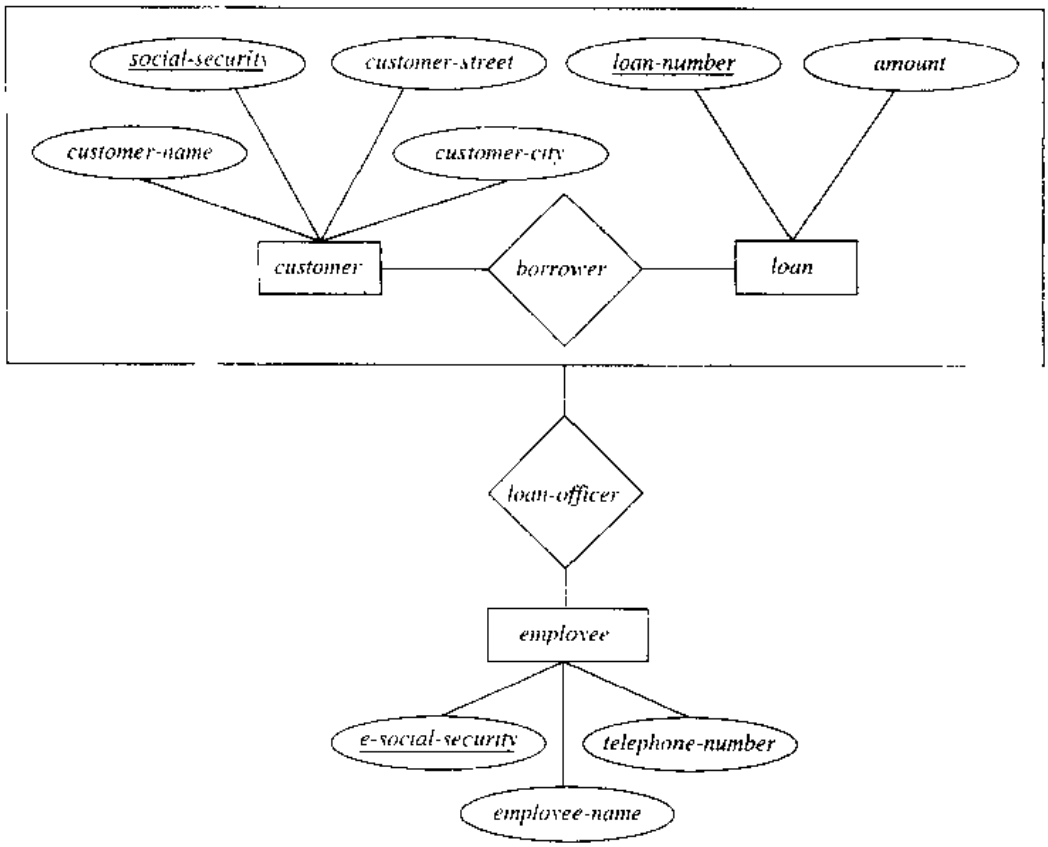


图 2-17　包含聚集的 E-R 图

2.8　设计数据库的 E-R 模式

E-R 数据模式使得我们通过设计数据库模式来给给定企业建模时有足够的灵活性。本节讨论数据库设计者如何从多个可选方案中进行选择。选择时需要做的决定包括：

- 用属性表示某个对象更恰当还是用实体集表示更恰当（参见 2.2.1 节）。
- 最准确描述现实世界中某个概念是用实体集还是联系集（参见 2.2.2 节）。
- 使用二元联系还是一对二元联系（参见 2.2.3 节）。
- 使用强实体集还是弱实体集（参见 2.6 节）。数据库中的一个强实体集和依附于它的弱实体集可被视为单一的“对象”，因为弱实体存在依赖于强实体。
- 使用概括是否合适（参见 2.7.2 节）。概括，或 ISA 联系的层次结构，通过允许在 E-R 图的某个地方表示相似实体集的共同属性，有助于进行模块化。
- 使用聚集是否合适（参见 2.7.5 节）。聚集将 E-R 图的某个部分集成为单一实体，使得我们可以将聚集实体看作一个单元，而不必关心其内部结构的细节。

可以看出，数据库设计者要做出正确的决定，就必须对被建模的企业有深刻的理解。

2.8.1　设计阶段

高层数据模型为数据库设计者提供概念框架，允许设计者以系统化的方式定义数据库用户的数据要求，以及为了满足这些要求应采用的数据库结构。因此，数据库设计的第一个阶段需

要刻画潜在数据库用户的数据需求。这一阶段的产品是用户需求规格说明。

接下来，设计者选择数据模型，并利用所选数据模型的概念将这些需求转化为数据库的概念模式。在此概念设计阶段所产生的模式提供企业的一个详细的综述。由于到目前为止我们只学习了 E-R 模型，因此将用它来设计概念模式。用 E-R 模型的术语来说，概念模式定义所有的实体集、联系集、属性和映射约束。设计者反复检查此模式，以确保所有数据需求都能满足，并且不相互冲突。设计者还可以通过对设计的检查来去除冗余特征。在这个阶段，设计者关注的是如何描述数据及其相互关系，而不是定义物理存储细节。

完善的概念模式还应指明企业的功能需求。在功能需求规格说明中，用户描述将在数据上进行的各类操作（或事务）。这些操作可以是对数据进行修改或更新，可以是查询或检索特定数据，也可以是删除数据等。在概念设计阶段，设计者需要再次检查模式，看是否满足功能需求。

最后两个设计阶段完成从抽象数据模型到数据库实现的过程。逻辑设计阶段将高层概念模式映射到将被使用的 DBMS 的实际数据模型。这一阶段产生的针对特定 DBMS 的数据库模式在后续的物理设计阶段将用到。物理设计阶段定义数据库的物理特征，包括文件组织格式和内部存储结构，这些将在第 10 章讨论。

本章只讨论了 E-R 模型用于概念模式设计阶段的相关概念。为了提供对 E-R 数据模型进行讨论的背景，我们还给出了数据库设计过程的简单概括。第 7 章将完整地讨论关于数据库设计的内容。

在 2.8.2 节和 2.8.3 节，我们将数据库设计的最初两个阶段应用于银行业务的例子。我们采用 E-R 数据模型将用户需求转化为 E-R 图所表示的概念设计模式。

2.8.2　银行业务的数据需求

用户需求的最初说明可以基于同数据库用户的交谈和设计者对企业的分析。这一设计阶段产生的描述是定义数据库概念结构的基础。下面列出了银行的主要需求：

- 银行有多个分支机构。各个分支机构位于某个城市，由唯一的名字标识。银行监控每个分支机构的资产。
- 银行的客户通过其社会保障号来标识。银行存储每个客户的姓名及其居住的街道和城市。客户可以有帐户，并且可以贷款。客户可能同某个银行员工发生联系，该员工是此客户的贷款负责人或银行帐户负责人。
- 银行员工也通过其社会保障号来标识。银行的管理机构存储每个员工的姓名、电话号码、亲属姓名及其经理的社会保障号。银行还需要知道员工开始工作的日期，由此日期可以推知员工的雇佣期。
- 银行提供两类帐户——储蓄帐户和支票帐户。帐户可以由两个或两个以上客户共有，一个客户也可以有两个或两个以上的帐户。每个帐户被赋以唯一的帐户号。银行记录每个帐户的余额以及每个帐户所有者访问该帐户的最近日期。另外，每个储蓄帐户有其利率，而每个支票帐户有其透支额。
- 每笔贷款由某个分支机构发放，能被一个或多个客户所共有。每笔贷款用唯一的贷款号标识。银行需要知道每笔贷款所贷金额以及逐次支付情况。虽然贷款的付款号并不能唯一标识银行所有为贷款所付的款项，但可以唯一标识为某贷款所付的款项。对每次的付款需要记载其日期和金额。

现实银行中，还应像记载贷款所付款项那样来记载每个储蓄帐户或支票帐户取出或存入的

金额。由于这些记载的建模过程类似，且为了保持作为示例的应用的简洁，在模型中不考虑对存款和取款的记录。

2.8.3　与银行相关的实体集

对数据需求的说明是建造数据库概念模式的出发点。根据上节所列举的需求，我们开始定义实体集及其属性。

- 实体集 *branch* 具有属性 *branch-name*、*branch-city* 和 *assets*。
- 实体集 *customer* 具有属性 *customer-name*、*social-security*、*customer-street* 和 *customer-city*，此外还可以考虑加上属性 *banker-name*。
- 实体集 *employee* 具有属性 *e-social-security*、*employee-name*、*telephone-number*、*salary* 和 *manager*；此外还包括多值属性 *dependent-nam*、基属性 *start-date* 及其派生属性 *employment-length*。
- 两个帐户实体集——*savings-account* 和 *checking-account*——共同的属性是 *account-number* 和 *balance*，此外 *savings-account* 还具有属性 *interest-rate*，*checking-account* 还具有属性 *overdraft-amount*。
- 实体集 *loan* 具有属性 *loan-number*、*amount* 和 *originating-branch*，此外还可以考虑加上复合属性 *loan-payment*，组成该属性的元素有 *payment-number*、*payment-date* 和 *payment-amount*。

2.8.4　与银行相关的联系集

现在回到上节的最初设计模式，定义如下联系集和映射的基数：

- *borrower* 是 *customer* 和 *loan* 间的一个多对多联系集
- *loan-branch* 指明贷款来自哪个银行分支机构的多对一联系集
- *loan-payment* 是从 *loan* 到 *payment* 的一对多联系集，表明一笔款项是为某笔贷款而付的。
- *depositor* 具有联系属性 *access-date*，是 *customer* 和 *account* 间的多对多联系集，表明某个客户具有某个帐户。
- *cust-banker* 具有联系属性 *type* 的一个多对一联系集，表明一个客户可以从一个银行员工处获得建议，而一个银行员工可以为一个或多个客户提供建议。
- *works-for* 具有 *manager* 和 *worker* 角色的 *employee* 实体之间的联系集，其映射的基数表明了一个员工仅为一个经理工作，而一个经理监督一个或多个员工。

注意我们用联系集 *cust-banker* 代替了实体集 *customer* 的属性 *banker-name*，用联系集 *works-for* 代替了实体集 *employee* 的属性 *manager*。我们保留把 *loan* 作为一个实体集，联系集 *loan-branch* 和 *loan-payment* 分别代替了实体集 *loan* 的属性 *originating-branch* 和 *loan-payment*。

2.8.5　银行企业 E-R 图

根据上节的讨论，现在给出作为例子的银行企业的 E-R 图。图 2-18 给出了银行概念模式采用 E-R 概念的完整表示。图中包括由 2.8.2 和 2.8.3 节的设计过程得到的，并在 2.8.4 节进行了精化了的实体集、属性、联系集和映射的基数。

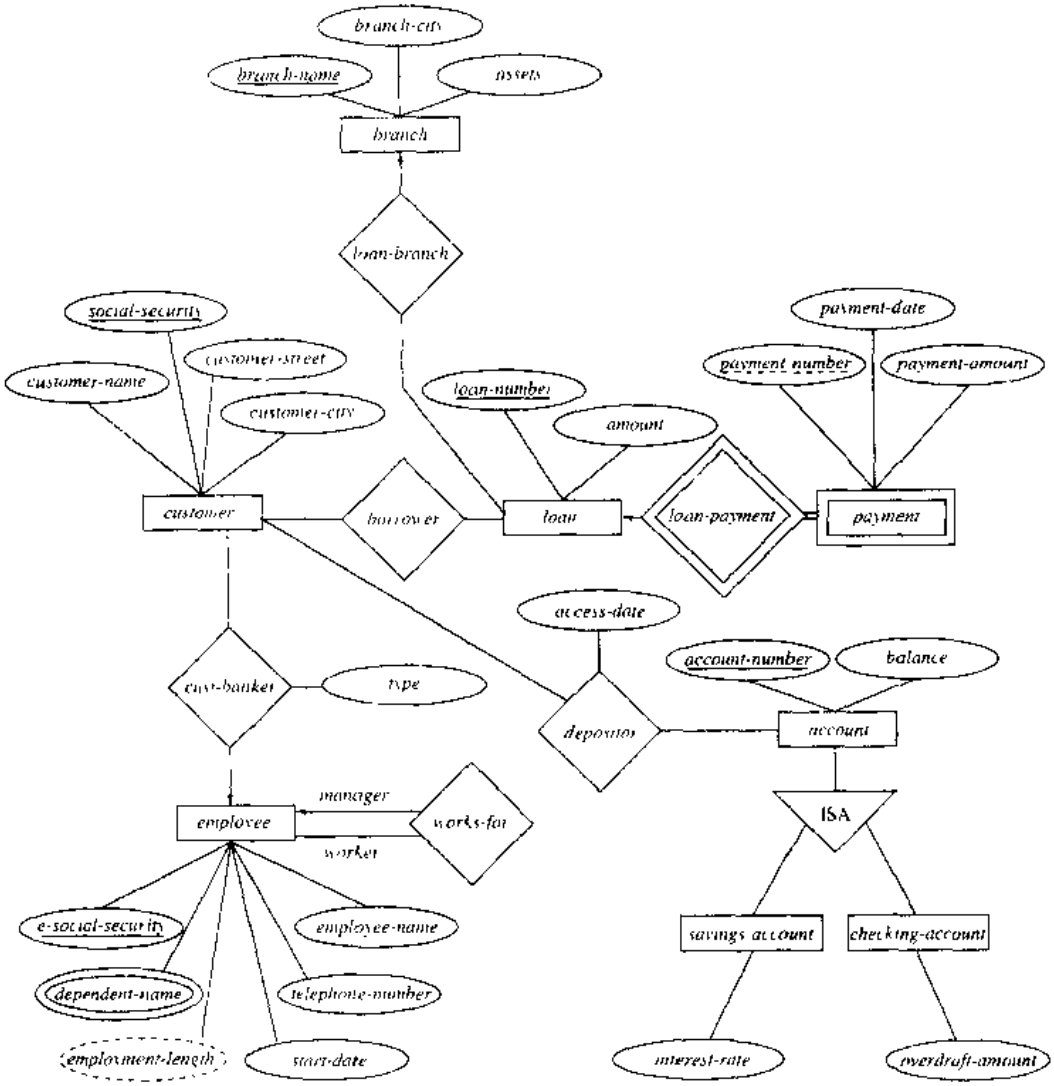


图 2-18　银行企业的 E-R 图

2.9　将 E-R 模式转换为表

符合 E-R 数据库模式的数据库可以表示为一些表的集合。数据库的每个实体集和联系集都有唯一的表与之对应，表名即为相应的实体集或联系集的名称。每个表有多个列，每列有唯一的列名。

E-R 模型和关系数据库模型都是现实世界抽象的逻辑表示。由于两种模型采用类似的设计原则，可以将 E-R 设计转换为关系设计。将数据库的表示从 E-R 图转为表的形式是由 E-R 图产生关系数据库设计的基础。虽然关系和表之间存在重大区别，但在不很严格的情况下，可以将关系看作是由某些值形成的一个表。本节描述如何用表来表示 E-R 模式，第 3 章讨论如何由一个 E-R 模式产生关系数据库模式。

2.9.1　用表表示强实体集

设 E 是具有描述性属性 a_1，a_2，…，a_n 的强实体集。用具有 n 个不同列的表 E 来表示

这个实体集，每列同实体集 E 的一个属性对应。表中各行对应于实体集中的各个实体。

以图 2-8 中所示的实体集 *loan* 为例说明。实体集 *loan* 有两个属性：*loan-number* 和 *amount*。用含两列的表 *loan* 来表示这个实体集，如图 2-19 所示。其中一行

(L-17, 1000)

表示贷款号为 L-17 的贷款金额为 $1000。通过在表中加入一行，可以达到在数据库中加入新实体的目的，此外还可以对行进行删除或修改。

loan-number	*amount*
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-93	500
L-11	900
L-16	1300

图 2-19　*loan* 表

设 D_1 代表所有贷款号的集合，D_2 代表所有金额的集合。表 *loan* 的任一行必须是一个二元组 (v_1, v_2)，其中 v_1 是贷款（即属于集合 D_1），v_2 是金额（即属于集合 D_2）。通常，*loan* 表中只会包含所有可能行的集合的一个子集。将 *loan* 的所有可能行的集合称为 D_1 和 D_2 的笛卡儿积，记作

$$D_1 \times D_2$$

一般说来，如果一个表的列数为 n，我们将 D_1，D_2，…，D_n 的笛卡儿积记为

$$D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$$

来看另一个例子。图 2-8 所示 E-R 图中的实体集 *customer* 具有属性 *customer-name*、*social-security*、*customer-street* 和 *customer-city*，对应的 *customer* 表有四列，如图 2-20 所示。

customer-name	*social-security*	*customer-street*	*customer-city*
Jones	321-12-3123	Main	Harrison
Smith	019-28-3746	North	Rye
Hayes	677-89-9011	Main	Harrison
Curry	244-66-8800	North	Rye
Lindsay	336-66-9999	Park	Pittsfield
Turner	182-73-6091	Putnam	Stamford
Williams	963-96-3963	Nassau	Princeton
Adams	335-57-7991	Spring	Pittsfield
Johnson	192-83-7465	Alma	Palo Alto

图 2-20　*customer* 表

2.9.2　用表表示弱实体集

设 A 是具有属性 a_1，a_2，…，a_m 的弱实体集。设 B 是 A 所依赖的强实体集，且其主码包括属性 b_1，b_2，…，b_n。用表 A 表示实体集 A，表中各列对应于以下属性集合中的各个属性：

$$\{a_1, a_2, \cdots, a_m\} \cup \{b_1, b_2, \cdots, b_n\}$$

以图 2-14 所示 E-R 图中的实体集 *payment* 为例说明。该实体集有三个属性：*payment-number*、*payment-date* 和 *payment-amount*；而实体集 *payment* 所依赖的实体集 *loan* 的主码是 *loan-number*，因此，用来表示 *payment* 的表应具有四个属性：*loan-number*、*payment-number*、*payment-date* 和 *payment-amount*，如图 2-21 所示。

2.9.3　用表表示联系集

设 R 是联系集，而所有参与 R 的实体集的主码属性集合为 a_1，a_2，…，a_m，如果 R 有

loan-number	*payment-number*	*payment-date*	*payment-amount*
L-17	5	10 May 1996	50
L-23	11	17 May 1996	75
L-15	22	23 May 1996	300
L-14	69	28 May 1996	500
L-93	103	3 June 1996	900
L-17	6	7 June 1996	50
L-11	53	7 June 1996	125
L-93	104	13 June 1996	200
L-17	7	17 June 1996	100
L-16	58	18 June 1996	135

图 2-21　*payment* 表

描述性属性则不妨设为 b_1，b_2，…，b_n。用表 R 表示该联系集，表中各列对应于以下属性集合中的各个属性：

$$\{a_1, a_2, \cdots, a_m\} \cup \{b_1, b_2, \cdots, b_n\}$$

以图 2-8 所示 E-R 图中的联系集 *borrower* 为例说明，此联系集涉及如下两个实体集：

- *customer*。主码为 *social-security*。
- *loan*。主码为 *loan-number*。

由于该联系集没有属性，*borrower* 表应有两列：*social-security* 和 *loan-number*，如图 2-22 所示。

social-security	*loan-number*
321-12-3123	L-17
019-28-3746	L-23
677-89-9011	L-15
555-55-5555	L-14
244-66-8800	L-93
019-28-3746	L-11
963-96-3963	L-17
335-57-7991	L-16

图 2-22　*borrower* 表

1. 表的冗余

将弱实体集和相应强实体集相关联的联系集比较特殊。正如前面提到的，这样的联系集是多对一的，且没有任何描述性属性。另外，弱实体集的主码包括了强实体集的主码。图 2-14 所示 E-R 图中，弱实体集 *payment* 通过联系集 *loan-payment* 依赖于强实体集 *loan*。弱实体集 *payment* 的主码是 {*loan-number*，*payment-number*}，实体集 *loan* 的主码是 {*loan-number*}。由于 *loan-payment* 没有任何描述性属性，表示 *loan-payment* 的表有两列：*loan-number* 和 *payment-number*，表示实体集 *payment* 的表有四列：*loan-number*、*payment-number*、*payment-date* 和 *payment-amount*，由此可见，*loan-payment* 表是多余的。一般情况下，将弱实体集与其所依赖的强实体集相关联的联系集的表是冗余的，在用表来表示 E-R 图时不必给出。

2. 表的合并

让我们看一下从实体集 A 到实体集 B 的多对一联系集 AB。用前面讲的建表方式将得到三个表：A、B 和 AB。然而，如果 A 对 B 有存在依赖（即对 A 中的任一实体 a 而言，其存在与否依赖于 B 中某实体 b 是否存在），那么可以将表 A 和 AB 合并成一个包含了两个表所有列的并集的表。

例如，对图 2-23 所示的 E-R 图来说，联系集 *account-branch* 从 *account* 到 *branch* 是多对一的。另外，E-R 图中的双线表明 *account* 对 *account-branch* 的参与是全部的，故一个帐户不可能不与任何银行分支机构相联系而存在。因此，只需以下两个表：

- *account*。具有属性 *account-number*、*balance* 和 *branch-name*。
- *branch*。具有属性 *branch-name*、*branch-city* 和 *assets*。

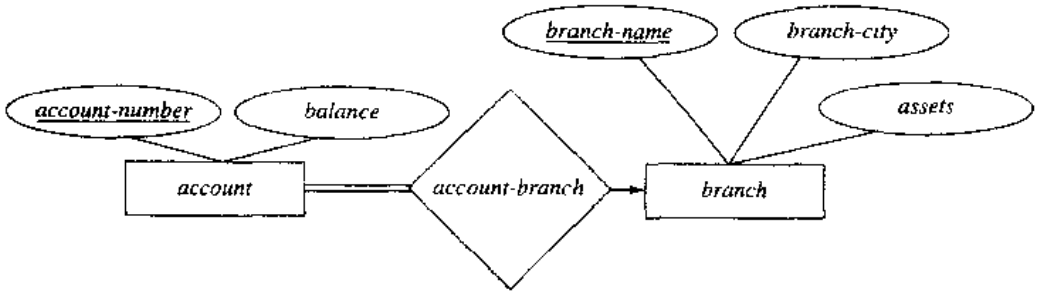


图 2-23　E-R 图

2.9.4　多值属性

我们已经看到，E-R 图中的属性通常都可以直接映射到相应表中的列。但是，多值属性并不如此简单，必须为其创建新表。

对一个多值属性 M，我们为其创建表 T，T 中有一列 C 与 M 对应，而 T 中其余列对应于以 M 为多值属性的实体集或联系集的主码。例如，图 2-18 所示 E-R 图中有多值属性 *dependent-name*。为该属性创建表 *dependent-name*，其中包括属性 *dname* 和 *e-social-security*，*dname* 与实体集 *employee* 的多值属性 *dependent-name* 对应，而 *e-social-security* 代表实体集 *employee* 的主码。员工的每个亲属在表中都以单独的一行出现。

2.9.5　用表表示概括

将包含概括的 E-R 图转换成表的表示有两种不同方法。尽管在图 2-15 中我们已提到概括，但为了简化讨论，我们谈到的内容实际上只包括第一层的低层实体集——即 *savings-account* 和 *checking-account*。

1）为高层实体集创建一个表。为每个低层实体集创建一个表，表中包括对应于低层实体集各属性的列，另外，对高层实体集主码的每个属性，表中也有相应的列。因此，将为图 2-15 所示的 E-R 图建立三个表：

- *account*。具有属性 *account-number* 和 *balance*。
- *savings-account*。具有属性 *account-number* 和 *interest-rate*。
- *checking-account*。具有属性 *account-number* 和 *overdraft-amount*。

2）如果概括是不相交且全部的——即如果两个低层实体集直接隶属于同一高层实体集，那么就不会有实体同时属于这两个低层实体集；同时，高层实体集的任何实体也必然会是某个低层实体集的成员——那么就可以采用另一种表示方法，这时，不为高层实体集创建任何表，只为每个低层实体集创建一个表，表中包括对应于低层实体集各属性的列和对应于高层实体集各属性的列。对图 2-15 所示的 E-R 图，将创建两个表：

- *savings-account*。具有属性 *account-number*、*balance* 和 *interest-rate*。
- *checking-account*。具有属性 *account-number*、*balance* 和 *overdraft-amount*。

对应于这两个表的关系 *savings-account* 和 *checking-account* 都以 *account-number* 为主码。

如果将第二种方法用于可重叠的概括，某些属性如 *balance* 就会不必要地被存储两次。同样，如果概括不是全部的，即有的帐户既不是储蓄帐户又不是支票帐户，这样的帐户用第二种方法就不能被表示出来。

2.9.6　用表表示聚集

将包含聚集的 E-R 图转换成表的表示是很直接的。考虑图 2-17 所示的 E-R 图，表示联系集 *loan-officer* 的表中，对应实体集 *employee* 和联系集 *borrower* 主码的每个属性都有相应的列。表中还应包括对应于联系集 *loan-officer* 各描述性属性的列（如果存在的话）。对 E-R 图中剩余部分重复前面的过程，可以得到如下的表：

- *customer*。具有属性 *customer-name*、*social-security*、*customer-street* 和 *customer-city*。
- *loan*。具有属性 *loan-number* 和 *amount*。
- *borrower*。具有属性 *social-security* 和 *loan-number*。
- *employee*。具有属性 *e-social-security*、*employee-name* 和 *telephone-number*。
- *loan-officer*。具有属性 *social-security*、*loan-number* 和 *e-social-security*。

2.10　总结

实体-联系（E-R）数据模型基于对现实世界的这样一种认识：世界由一组基本对象（称作实体）及这些对象间的联系组成。此模型的主要目的是服务于数据库设计过程，它的发展是为了帮助数据库设计，这是通过允许定义企业模式来实现的。企业模式代表了数据库的全局逻辑结构，这种全局结构可以用 E-R 图进行图形化表示。

实体是实际存在的可区别于其他对象的对象，我们通过把每个实体同描述该实体的一组属性相联系来将它与其他对象相区别。联系是多个实体间的相互关联。相同类型的所有实体的集合构成实体集，相同类型的所有联系的集合构成联系集。

映射的基数指明另一实体通过联系集可以和实体集中的多少个实体相联。有一种约束是存在依赖，它表明实体 x 的存在依赖于实体 y 的存在。

数据库建模的一个重要任务是要说明实体之间以及联系之间如何相互区别。概念上来说，各个实体或联系是互不相同的，但从数据库的角度来看，它们的差异必须用属性表示出来。为了进行这样的区别，为每个实体集指定一个主码。主码是一个或多个属性的集合，这些属性的整体可以使我们在实体集中唯一确定一个实体或在联系集中唯一确定一个联系。如果一个实体集没有足够形成主码的属性集合，就称其为弱实体集，而有主码的实体集称为强实体集。

特殊化和概括定义了一个高层实体集和一个或多个低层实体集之间内容上的联系。特殊化是取出高层实体集的一个子集来形成一个低层实体集。概括是用两个或多个不相交的（低层）实体集的并集来形成一个高层实体集。高层实体集的属性被低层实体集继承。

E-R 模型的一个局限是它不能表示联系间的联系，解决的办法是采用聚集。聚集是一种抽象，它将联系集看作高层实体集，这样，联系集及其相关实体集就可以像其他实体一样被看作高层实体集。

E-R 模型各种各样的特征提供给数据库设计者大量的选择机会，使得设计者可以最好地表现被建模的企业。在不同的场合，概念和对象可以用实体、联系或属性来表示。企业总体结构的某些方面可以用弱实体集、概括、特殊化或聚集来表示。设计者常常需要在简单的、紧凑的模型与更精确但也更复杂的模型之间做出恰当的选择。

符合 E-R 图的数据库可以用表的集合来表示。数据库的每个实体集和联系集都有唯一的表与之对应，表名即为相应的实体集或联系集的名称。每个表有多个列，每列有其唯一列名。将

数据库的表示从 E-R 图转向表的形式是由 E-R 图产生关系数据库设计的基础。

习题

2.1　解释主码、侯选码和超码这些术语之间的区别。

2.2　为车辆保险公司设计一个 E-R 图。该公司有很多客户，每个客户有一或多辆车。每辆车可能发生 0 次或任意多次事故。

2.3　为医院设计一个 E-R 图。医院有很多病人和医生，同每个病人相关的是一系列各种检查和测试的记录。

2.4　将习题 2.2、2.3 的每个 E-R 图转换成适当的表。

2.5　解释强实体集和弱实体集的差别。

2.6　我们只要通过加入恰当的属性就能将弱实体集转变为强实体集，那么，我们为什么要有弱实体集呢？

2.7　给出聚集的概念，举出两个例子说明此概念是有用的。

2.8　考虑有相同实体集重复多次出现的一个 E-R 图，为什么这样的冗余是应尽量避免的不良设计？

2.9　考虑在大学中为期末考试安排教室的这样一个数据库。此数据库可以仅仅用一个实体集 *exam* 来对其建模，此实体集 *exam* 包括属性 *course-name*、*section-number*、*room-number* 和 *time*。当然，也可以采用另一种方式，定义一个或多个实体集以及一些联系集来代替 *exam* 实体集的某些属性。这些实体集和联系集如下：

- *course*。具有属性 *name*、*department* 和 *c-number*。
- *section*。具有属性 *s-number* 和 *enrollment*，该实体集作为一个弱实体集依赖于 *course*。
- *room*。具有属性 *r-number*、*capacity* 和 *building*。

(a) 画一个 E-R 图来说明以上三个增加的实体集的作用。

(b) 说明哪些应用特性将会影响是否保留各个新增实体集的决定。

2.10　为某个企业设计 E-R 图时，通常需要从多个可选方案中选择一个。为了作出正确的选择，你应该考虑哪些准则？

2.11　E-R 图可以被视为一个图。当作为企业模式时，图的下述情况分别有何含义？

(a) 图是非连通的。

(b) 图中无环。

2.12　在 2.2.3 节中，我们曾用二元联系来表示一个三元联系（图 2-24a），如图 2-24b 所示。图 2-24c 为另一种可能的表示方法。讨论用二元联系表示三元联系的这两种不同方法相对而言各自的优点。

2.13　为机动车辆销售公司设计一个概括－特殊化层次结构。该公司出售摩托车、小客车、货车和公共汽车。论述你在层次结构的各层确定属性位置的合理性，说明为什么它们不被放在较高层次或较低的层次。

2.14　说明条件定义和用户定义设计约束的差别。系统能自动检查这些约束中的哪些？解释你的答案。

2.15　说明不相交约束和可重叠约束这两种设计约束之间的差别。

2.16　解释全部的和部分的这两种设计约束之间的差别。

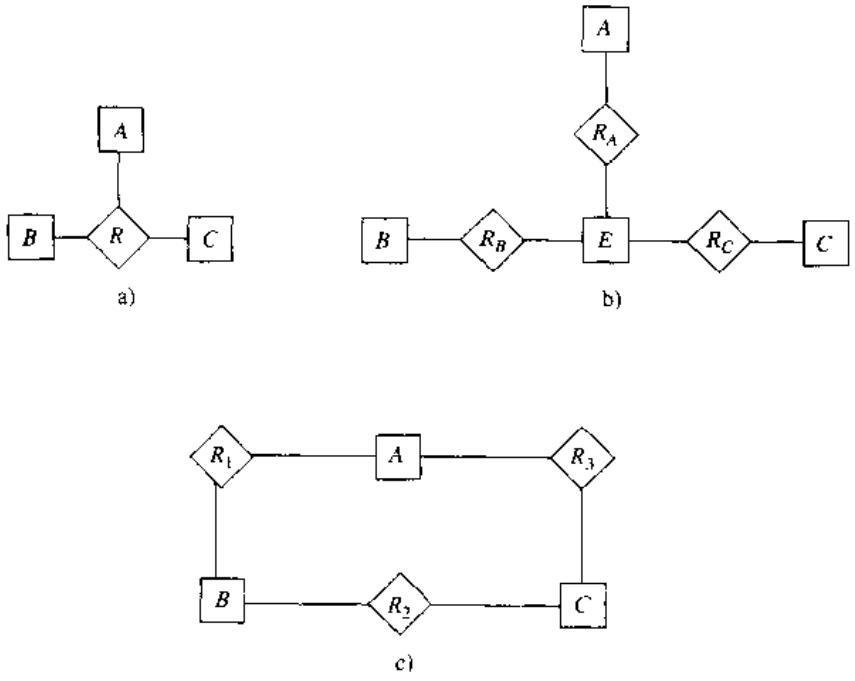


图 2-24 习题 2.12 的 E-R 图（属性未给出）

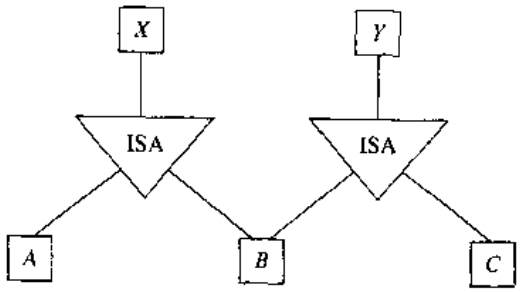


图 2-25 练习 2.17 的 E-R 图（属性未给出）

2.17 图 2-25 是概括和特殊化的一个格结构。说明实体集 A、B 和 C 如何从高层实体集 X 和 Y 继承属性。讨论 X 的某个属性和 Y 的某个属性同名时应怎样处理。

2.18 两个相互独立的银行将要合并，假设两个银行使用相同的 E-R 数据库模式——图 2-18 中的数据库模式（当然，这个假设很不实际，我们将在 18.9 节讨论更实际的情况）。如果合并后的银行只有一个数据库，那么可能存在如下问题：

- 合并前的两个银行的某些分支机构名称可能相同。
- 可能有的客户同时是两个银行的客户。
- 某些贷款号和帐户号可能在两个银行中都使用（却代表不同的贷款和帐户）。

对这些潜在的问题，说明为什么确实可能会存在困难。提出解决问题的一种办法，并说明采用这种办法需要做的修改以及这些修改对模式和数据的影响。

2.19 假设一个银行在美国，而另一个在加拿大，重新考虑上题所提出的问题。同样地，两个银行均采用图 2-18 的模式，但加拿大的银行使用由加拿大政府赋予的 *social-insurance* 号，而美国银行正如我们在本章一直假定的那样使用社会保障号。除了上题指出的问题外，在这种多个国家的情况下还会有什么样的问题？你将怎样解决？注意既要考虑模式又要考虑数据实际的值。

文献注解

E-R 数据模型由 Chen [1976] 引入。使用扩展 E-R 模型的关系数据库逻辑设计方法学由 Teorey 等 [1986] 给出。Lyngbaek 和 Vianu [1987] 以及 Markowitz 和 Shoshani [1992] 讨论了从扩展 E-R 模型到关系模型的映射。

E-R 模型的不同操纵语言已经被提出：GERM [Benneworth 等 1981]、GORDAS [ElMasri 和 Wiederhold 1981] 以及 ERROL [Markowitz 和 Raz 1983]。E-R 数据库的一种图形化查询语言由 Zhang 和 Mendelzon [1983] 以及 ElMasri 和 Larson [1985] 提出。一种一般 E-R 模型的代数由 Parent 和 Spaccapietra [1985] 给出。Campbell 等 [1985] 给出了 E-R 模型的一种与关系代数和演算（见第 3 章）具有同等表达能力的语言。Gogolla 和 Hohenstein [1991] 给出了具有形式化数学语义的 E-R 语言。

概括、特殊化和聚集的概念由 Smith 和 Smith [1977] 引入，在 Hammer 和 McLeod [1981] 中得到了扩充，Lenzerini 和 Santucci [1983] 在定义 E-R 模型中的基数约束时使用了这些概念。E-R 模型的一个变种即基于逻辑的语义由 Di Battista 和 Lenzerini [1989] 给出。

Batini 等 [1992] 以及 ElMasri 和 Navathe [1994] 是与此相关的基本教材。

第3章 关系模型

在商用数据处理应用中，关系模型现在已经成为主要的数据模型。早期的数据库系统建立在网状模型（如附录A）或层次模型（如附录B）的基础上。同关系模型相比，这两种较早的模型与底层实现的结合更加紧密。

关系数据库具有坚实的理论基础。这一理论有助于关系数据库的设计和用户对数据库信息需求的有效处理。在第6章和第7章我们将学习这一理论。

关系模型现在正广泛应用于传统数据处理领域以外的大量应用中。第9章将讨论关系模型为了适应这些新应用的需求而进行的必要扩展。

3.1 关系数据库的结构

关系数据库是表的集合，每个表有唯一的名字。表的结构如同第2章中用表表示E-R数据库时所给的那样。表中一行代表的是一系列值之间的联系。由于一个表就是这种联系的集合，表这个概念和数学上的关系这个概念是密切相关的，这也正是关系数据库名称的由来。在接下来的内容中将介绍关系的概念。

本章将使用许多不同的关系来说明作为关系数据模型基础的各种概念。这些关系代表银行企业的一部分。它们同第2章所用的表有轻微差别，这主要是为了简化表示。第7章将详细讨论有关判断什么是适当关系结构的准则。

3.1.1 基本结构

请看图3-1中的 *account* 表。该表有三个列首：*branch-name*、*account-number* 和 *balance*。用关系模型的术语来说，这些列首称作属性（正如第2章对E-R模型所做的一样）。每个属性有一个允许的值的集合，称为该属性的域。例如，属性 *branch-name* 的域是所有分支机构名称的集合，用 D_1 表示此集合，用 D_2 表示所有帐户号的集合，用 D_3 表示所有余额的集合。正如在第2章里看到的那样，*account* 的每一行都必须是一个三元组 (v_1, v_2, v_3)，其中 v_1 是分支机构名称（即 v_1 在域 D_1 中），v_2 是帐户号（即 v_2 在域 D_2 中），v_3 是余额（即 v_3 在域 D_3 中）。通常，*account* 所包含的只是所有可能的行的一个子集，因此，*account* 是

$$D_1 \times D_2 \times D_3$$

的一个子集。一般地说，有 n 个属性的表是

$$D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$$

的一个子集。

数学家将关系定义为一系列域上的笛卡儿积的子集。这一定义与我们对表的定义几乎是完全相符的，唯一的区别在于给属性赋予了名称，而数学家用数字化的“名称”，即整数1表示的属性对应的域是域列表中第一个，整数2表示的属性对应的域是域列表中第二个，依此类推。由于表实际上是关系，因而用数学名词关系和元组来代替表和行。

在图3-1所示关系 *account* 中共有七个元组。设元组变量 t 指向关系中的第一个元组。如果用 t [*branch-name*] 表示 t 在 *branch-name* 属性上的值，则 t [*branch-name*] = “Down-

town"，t［*balance*］=500。也可以用 t［1］来表示元组 t 在第一个属性（*branch-name*）上的值，t［2］表示 *account-number*，等等。由于关系是元组的一个集合，我们用数学上的表示法 $t \in r$ 表示元组 t 在关系 r 中。

branch-name	*account-number*	*balance*
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	750

图 3-1　*account* 关系

对所有关系 r 而言，r 的每个属性的域都应是原子的。如果域的元素被看作是不可再分的单元，则域是原子的。例如，整数的集合是一个原子的域，而所有整数集的集合不是原子的。这两个集合的区别在于，人们通常认为整数不能再划分为更小的部分，而整数集可以划分为更小的部分——即整数集由整数组成。最重要的问题不在于域自身是什么，而在于怎样在数据库中使用域中元素。如果将每个整数都看作是数字的一个有序列表，则所有整数构成的域也就不是原子的了。以下的例子中，总假设域是原子的。第 9 章将讨论嵌套关系数据模型，其中允许非原子的域。

有时几个属性会有相同的域。例如，如果关系 *customer* 具有三个属性 *customer-name*、*customer-street* 和 *customer-city*，关系 *employee* 具有属性 *employee-name*，很可能属性 *customer-name* 和 *employee-name* 有相同的域：所有人名的集合。与此相反，*balance* 和 *branch-name* 的域显然不同。而 *customer-name* 和 *branch-name* 的域是否相同可能不会如此一目了然。在物理层，客户姓名和分支机构名称都是字符串，但是，在逻辑层上，可能希望 *customer-name* 和 *branch-name* 有不同的域。

值 *null* 是所有可能域的成员，表明值未知或不存在。例如，如果在关系 *customer* 中包括属性 *telephone-number*，则可能有客户没有电话号码，或电话号码未提供，这时就借助于空值来强调该值未知或不存在。以后会看到，空值给数据库的访问和更新带来很多困难，因此应尽量避免使用空值。

3.1.2　数据库模式

当谈论数据库时，必须区分*数据库模式*和*数据库实例*。数据库模式是数据库的逻辑设计，而数据库实例是给定时刻数据库中数据的一个快照。

关系的概念对应于程序设计语言中变量的概念，而关系模式的概念对应于程序设计语言中类型定义的概念。

为了使用方便，通常给关系模式一个名字，正如在程序设计语言中给类型定义一个名字一样。在我们采用的表示法中，关系的名字由小写字母组成，关系模式的名字以大写字母开头。根据这样的表示法，用 *Account-schema* 表示关系 *account* 的关系模式，因此，

$$Account\text{-}schema = (branch\text{-}name,\ account\text{-}number,\ balance)$$

即用 *account*（*Account-schema*）表示 *account* 是 *Account-schema* 上的关系。

一般地说，关系模式由属性序列及各属性对应的域组成。第 4 章讨论 SQL 语言时我们才去关心每个属性域的精确定义。

关系实例的概念对应于程序设计语言中变量的值的概念。给定变量的值会随时间发生变化；同样，当关系被更新时，关系实例的内容也随时间发生了变化。尽管如此，常简单地用"关系"指代"关系实例"。

作为关系实例的例子，来看图 3-2 中的 *branch* 关系。此关系的模式为

$$Branch\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets)$$

注意属性 *branch-name* 既出现在 *Branch-schema* 中又出现在 *Account-schema* 中，这样的重复并不是一个巧合。实际上，在关系模式中使用相同属性正是将不同关系的元组联系起来的一种方

法。例如，假设希望获得位于 Brooklyn 市的分支机构所维护的所有帐户信息。我们首先在关系 *branch* 中找出所有位于 Brooklyn 市的分支机构的名称，接着，对每一个分支机构，在关系 *account* 中找出它所维护的帐户信息。用 E-R 模型中的术语来说，属性 *branch-name* 在两个关系中代表相同的实体集。

继续看银行的例子，我们需要一个描述客户信息的关系，该关系模式为

Customer-schema = (*customer-name*, *customer-street*, *customer-city*)

图 3-3 中给出了关系 *customer* (*Customer-schema*) 的一个示例。

branch-name	*branch-city*	*assets*
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
Pownal	Bennington	300000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

图 3-2 *branch* 关系

customer-name	*customer-street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

图 3-3 *customer* 关系

此外还需要一个描述客户与帐户间联系的关系。该关系模式为

Depositor-schema = (*customer-name*, *account-number*)

图 3-4 中给出了关系 *depositor* (*Depositor-schema*) 的一个示例。

customer-name	*account-number*
Johnson	A-101
Smith	A-215
Hayes	A-102
Turner	A-305
Johnson	A-201
Jones	A-217
Lindsay	A-222

图 3-4 *depositor* 关系

在银行这个例子中，似乎可以只要一个关系模式而不需要多个。对用户来说，用一个关系模式而不是多个关系模式时考虑问题会更简单些。假设在此例中只用一个关系，其模式为

(*branch-name*, *branch-city*, *assets*, *customer-name*, *customer-street*, *customer-city*, *account-number*, *balance*)

不难发现，如果一个客户有几个帐户，则对他的每个帐户我们都要存储一遍它的地址，也就是说，必须重复某些信息。这种重复是一种浪费，而如前面所讲那样使用多个关系，就可以避免浪费。

另外，如果一个分支机构没有帐户（例如一个新建的还没有客户的分支机构），则使用前面的单一关系就不能构造完整的元组，因为同 *customer* 和 *account* 相关的数据还不存在。为了表示不完整元组，必须使用表示值未知或不存在的 *null* 值。因此，在本例中，*customer-name*、*customer-street* 等的值都为空。通过使用多个关系，可以不用空值就能表示没有客户的分支机构的信息，只需用 *Branch-schema* 上的一个元组来表示分支机构的信息，其他模式上的元组只有在所需信息可以得到时才去构造。

第 7 章将讲授一些准则，以确定什么样的关系模式集更恰当，这些准则的主要依据是信息的重复与空值的存在。目前，我们总假定关系模式已经给定。

再加入两个描述银行各分支机构所维护贷款信息的关系模式：

Loan-schema = (*branch-name*, *loan-number*, *amount*)

Borrower-schema = (*customer-name*, *loan-number*)

图 3-5 和图 3-6 分别给出了关系 *loan* (*Loan-schema*) 和 *borrower* (*Borrower-schema*) 的示例。

branch-name	*loan-number*	*amount*
Downtown	L-17	1000
Redwood	L-23	2000
Perryridge	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
Round Hill	L-11	900
Perryridge	L-16	1300

图 3-5　*loan* 关系

customer-name	*loan-number*
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

图 3-6　*borrower* 关系

前面描述的银行企业是从图 3-7 的 E-R 图产生的。各关系模式对应于使用 2.9 节所给方法所产生的各个表。假定实体集 *branch* 的主码是 *branch-name*，*Customer-schema* 的主码是 *customer-name*。我们并没有像第 2 章中那样使用属性 *social-security*，因为现在希望此银行数据库例子中的关系模式越小越好。事实上，在真实数据库中，我们却希望用属性 *social-security* 作主码。实体集 *account* 和 *loan* 的主码分别是 *account-number* 和 *loan-number*。

注意，对应于 *account-branch* 的表和对应于 *loan-branch* 的表已分别组合进 *account* 表和 *loan* 表了，因为如图 3-7 所示，*account* 和 *loan* 在相应的联系中都是完全参与的。

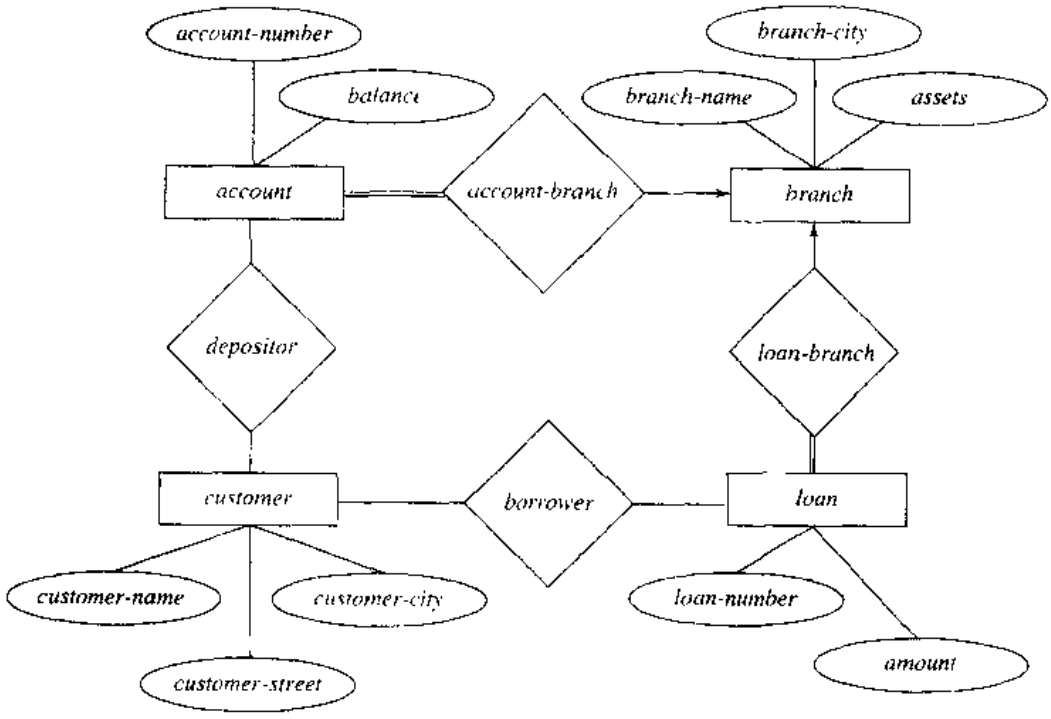


图 3-7　银行企业的 E-R 图

最后，还需要注意关系 *customer* 中可能包含在该银行中既无帐户又无贷款的客户的信息。

这里描述的银行情况将作为本章及后续章节的主要例子，偶尔为了说明一些特别地方我们还会另外引入一些关系模式。

3.1.3　码

第 2 章中讨论的超码、候选码和主码的概念也适用于关系模型。例如，在 *Branch-schema*

中，{*branch-name*} 和{*branch-name*，*branch-city*} 都是超码。{*branch-name*，*branch-city*} 不是候选码，因为{*branch-name*} 是{*branch-name*，*branch-city*} 的子集并且是超码。{*branch-name*} 是候选码，并且在此还将用它作主码。*branch-city* 不是超码，因为同一城市的两个分支机构可以名字不同（且资产不同）。

设 R 是一个关系模式。如果说 R 的子集 K 是 R 的超码，则限制了关系 $r(R)$，此关系中任意两个不同元组不会在 K 的所有属性上值完全相等，即，如果 t_1 和 t_2 在 r 中且 $t_1 \neq t_2$，则 $t_1[K] \neq t_2[K]$。

如果关系数据库模式是基于由 E-R 模式导出的表，那么就可以由导出关系数据库模式的实体集和联系集的主码确定关系模式的主码。

- 强实体集。实体集的主码成为关系的主码。
- 弱实体集。与弱实体集对应的表以及关系包括：
  - 弱实体集的属性。
  - 弱实体集所依赖的强实体集的主码。

关系的主码由强实体集的主码和弱实体集的分辨符共同组成。

- 联系集。相关实体集的主码共同构成关系的超码。如果联系是多对多的，则此超码也就是主码。2.4.2 节描述了如何确定其他情况下的主码，2.9.3 节中还曾讲到将弱实体集同相应强实体集连接起来的联系不产生表。
- 复合表。2.9.3 节中曾讲到，从 A 到 B 的二元一对多联系集可以表示为由 A 的属性和该联系集的属性（如果有的话）构成的一张表。“多”方实体集的主码成为关系的主码（即，如果联系集从 A 到 B 是多对一的，则 A 的主码是关系的主码）。对一对一的联系集来说，关系的构造如同多对一联系集那样，不同的是，任一实体集的主码都可以被选作关系的主码，因为它们都是候选码。
- 多值属性。2.9.4 节中曾经讲到，多值属性 M 可以表示为由以 M 作为属性的实体集或联系集的主码和保存单个 M 值的列 C 构成的一张表。实体集或联系集的主码与属性 C 共同构成关系的主码。

从上面所列可以看出，关系模式的属性中可以包含另一关系模式的主码，这样的码称为外码。例如，*Account-schema* 中的属性 *branch-name* 是外码，因为 *branch-name* 是 *Branch-schema* 的主码。

3.1.4 查询语言

查询语言是用户用来从数据库中请求获取信息的语言，这些语言通常比一般的程序设计语言层次更高。查询语言可以分为过程化的和非过程化的。在过程化语言中，用户指导系统对数据库执行一系列操作以计算所需结果。在非过程化语言中，用户只需描述所需信息，而不用给出获取该信息的具体过程。

大多数商用关系数据库系统提供的查询语言中既包含过程化方式的成分，又包含非过程化方式的成分。在第 4 和第 5 章将学习几种商用语言。本章讨论“纯”语言：关系代数是过程化的，而元组关系演算或域关系演算是非过程化的。这些语言简洁且形式化，缺少商用语言的“语法修饰”，但这些语言也说明了从数据库中提取数据的基本技术。

开始我们只关心查询。完整的数据库操纵语言不仅包括查询语言，还包括对数据库进行修改的语言。对数据库进行修改的语言包括插入或删除元组的命令以及对已有元组的部分进行修改的命令。在完成对“纯”查询语言的讨论后，将谈一谈对数据库的修改。

3.2 关系代数

关系代数是过程化的查询语言。它包括一个运算集合，这些运算以一个或两个关系为输入，产生一个新的关系作为结果。关系代数基本运算有选择、投影、并、集合差、笛卡儿积和更名。在基本运算以外，还有一些其他运算，即集合交、自然连接、除和赋值，这些运算将用基本运算来定义。

3.2.1 基本运算

选择、投影和更名运算称为一元运算，因为它们只对一个关系进行运算。另外三个运算对两个关系进行运算，因而称为二元运算。

1. 选择运算

选择运算选出满足给定谓词的元组。用小写希腊字母 σ 来表示选择，而将谓词写作 σ 的下标，并在 σ 后的括号中给出作为参数的关系。因此，为了选择 *loan* 关系中分支机构名称为“Perryridge”的那些元组，应写作

$$\sigma_{branch\text{-}name = \text{“Perryridge”}}(loan)$$

如果关系 *loan* 如图 3-5 所示，则从上述查询产生的关系如图 3-8 所示。

branch-name	*loan-number*	*amount*
Perryridge	L-15	1500
Perryridge	L-16	1300

图 3-8　$\sigma_{branch\text{-}name = \text{“Perryridge”}}(loan)$ 的结果

通过书写

$$\sigma_{amount > 1200}(loan)$$

可以找到贷款金额大于 $1200 的所有元组。

通常允许在选择谓词中进行比较，使用的是 =、≠、<、≤、>和≥。另外，可以用连词 *and*（∧）和 *or*（∨）将多个谓词合并为一个较大的谓词。因此，为了找到由 Perryridge 分支机构产生的贷款额大于 $1200 的贷款，需要书写

$$\sigma_{branch\text{-}name = \text{“Perryridge”} \wedge amount > 1200}(loan)$$

选择谓词中可以包括两个属性的比较，以关系 *loan-officer* 为例说明。关系 *loan-officer* 包含三个属性：*customer-name*、*banker-name* 和 *loan-number*，说明某银行员工是某客户某笔贷款的办理者。为了找出所有和其贷款办理者重名的客户，可以这样写

$$\sigma_{customer\text{-}name = banker\text{-}name}(loan\text{-}officer)$$

由于特殊值 *null* 表明“值未知或不存在”，因而所有涉及空值的比较均得到 false 值。

2. 投影运算

假设想要列出所有贷款的号码和金额，而不关心分支机构名称，投影运算使得可以产生这样的关系。投影运算是一元运算，返回作为参数的关系的某些属性。由于关系是一个集合，所以所有重复行均被去除。投影用希腊字母 Π 表示，所有希望在结果中出现的属性作为 Π 的下标，作为参数的关系在跟随 Π 后的括号中。因此，列出所有贷款号码及其金额的查询可以写作

$$\Pi_{loan\text{-}number, amount}(loan)$$

此查询产生的关系如图 3-9 所示。

loan-number	*amount*
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-93	500
L-11	900
L-16	1300

图 3-9　贷款号码和贷款金额

3. 关系运算的组合

关系运算的结果自身也是一个关系，这一事实非常重要。来

看一个更复杂的查询“找出居住在 Harrison 的所有客户”，我们写作

$$\Pi_{customer\text{-}name}\ (\sigma_{customer\text{-}city\ =\ \text{“Harrison”}}\ (customer))$$

注意，对投影运算而言，没有给出一个关系的名字来作为其参数，而是用一个对关系进行求值的表达式来作为参数。

一般地说，由于关系代数运算的结果同其输入的类型一样，仍为关系，所以可以把多个关系代数运算组合成一个关系代数表达式。将关系代数运算组合成关系代数表达式如同将算术运算（如+、-、×、÷）组合成算术表达式一样。将在 3.2.2 节给出关系代数表达式的形式化定义。

4. 并运算

假设有一个查询要找出所有有贷款或有帐户或二者兼具的银行客户。注意关系 *customer* 中并不能包含该信息，因为客户在该银行可以既无帐户又无贷款。为了回答这一查询，需要关系 *depositor*（图 3-4）和关系 *borrower*（图 3-6）中的信息。如下找出所有在此银行有贷款的客户：

$$\Pi_{customer\text{-}name}\ (borrower)$$

如下找出所有在此银行有帐户的客户：

$$\Pi_{customer\text{-}name}\ (depositor)$$

为了回答这个查询，需要将这两个集合并起来，即需要所有出现在这两个集合之一的或同时出现在这两个集合中的客户姓名。通过二元运算“并”来获得这些数据，“并”运算像集合论中一样用∪表示，于是，所需表达式为

$$\Pi_{customer\text{-}name}\ (borrower) \cup \Pi_{customer\ name}\ (depositor)$$

此查询产生的关系如图 3-10 所示。在结果中共有 10 个元组，尽管其中包含了 7 个不同的贷款者和 6 个不同的存款者。这种数量的明显减少是由于 Smith、Jones 和 Hayes 既是贷款者又是存款者。由于关系是集合，重复值被去除了。

customer-name
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Adams

图 3-10 有贷款或帐户的所有客户姓名

本例中做并运算的两个集合都由 *customer-name* 值构成。一般说来，必须保证做并运算的关系是相容的。例如，将关系 *loan* 和关系 *borrower* 做并运算就没有意义，前者是包含三个属性的关系，而后者是包含两个属性的关系。此外，来看客户姓名的集合和城市的集合之间的并运算，这样的并在大多数情况下也没有意义。因此，要使并运算 $r \cup s$ 有意义，要求以下两个条件同时成立：

1）关系 r 和 s 必须是同元的，即它们的属性数目必须相同。

2）对任意 i，r 的第 i 个属性的域必须和 s 的第 i 个属性的域相同。

注意 r 和 s 可以是作为关系代数表达式结果的临时关系。

5. 集合差运算

用 - 表示的集合差运算使得可以找出在一个关系中而不在另一个关系中的那些元组。表达式 $r - s$ 的结果即是一个包含所有在 r 中而不在 s 中的元组的关系。

可以通过书写如下表达式来找出所有有帐户而无贷款的客户

$$\Pi_{customer\text{-}name}\ (depositor) - \Pi_{customer\text{-}name}\ (borrower)$$

该查询产生的关系如图 3-11 所示。

customer-name
Johnson
Turner
Lindsay

图 3-11 有帐户而无贷款的客户姓名

正如并运算的情况一样，必须保证集合差运算在相容的关系间进行。因此，为使集合差运算 $r - s$ 有意义，要求关系 r 和 s 是同元的，

且 r 的第 i 个属性的域和 s 的第 i 个属性的域相同。

6. 笛卡儿积运算

用×表示的笛卡儿积运算使得可以将任意两个关系的信息组合在一起。将关系 r_1 和 r_2 的笛卡儿积写作 $r_1 \times r_2$。

我们曾将关系定义为一组域上的笛卡儿积的子集，从这个定义，应该已经对笛卡儿积运算的定义有了直观的认识。但是，由于相同的属性名可能同时出现在 r_1 和 r_2 中，需要提出一个命名机制来区别这些属性。这里采用的方式是在属性上附加该属性所来自的关系名称。例如，r = *borrower* × *loan* 的关系模式为

(*borrower*.*customer-name*, *borrower*.*loan-number*, *loan*.*branch-name*, *loan*.*loan-number*, *loan*.*amount*)

用这样的模式，可以区别 *borrower*.*loan-number* 和 *loan*.*loan-number*。对那些只在两个关系模式之一出现的属性，通常省略其关系名前缀，这样的简化不会导致任何歧义。这样，就可以将 r 的关系模式写作

(*customer-name*, *borrower*.*loan-number*, *branch-name*, *loan*.*loan-number*, *amount*)

上述命名规则规定，作为笛卡儿积运算参数的关系名必须不同，这一规定有时会带来一些问题，例如当某个关系需要与自身做笛卡儿积时。此外，在笛卡儿积中使用关系代数表达式的结果时也会产生类似问题，因为必须给关系一个名字以引用其属性。在下一部分中，将看一下如何通过更名运算避免这些问题。

我们已经知道 r = *borrower* × *loan* 的关系模式，那么到底那些元组会出现在 r 中呢？可能跟你所想的一样，如后面所述的每一个可能的元组对都形成 r 的一个元组：元组对中一个来自关系 *borrower* 而另一个来自关系 *loan*。因此，r 是一个很大的关系，正如从图 3-12 可看到的一样。图 3-12 中只不过是构成 r 的部分元组。

假设 *borrower* 中有 n_1 个元组，*loan* 中有 n_2 个元组，那么可以有 $n_1 \times n_2$ 种方式来选择元组对——每个关系中选出一个元组，因此 r 中有 $n_1 \times n_2$ 个元组。特别地，注意对 r 中的某些元组 t 来说，可能 t [*borrower*.*loan-number*] ≠ t [*loan*.*loan-number*]。

一般地，如果有关系 r_1 (R_1) 和 r_2 (R_2)，则关系 $r_1 \times r_2$ 的模式是 R_1 和 R_2 串接而成的。关系 R 中包含所有满足以下条件的元组 t：r_1 中存在元组 t_1，r_2 中存在元组 t_2，且 t [R_1] = t_1 [R_1]，t [R_2] = t_2 [R_2]。

假设希望找出所有在 Perryridge 分支机构中有贷款的客户姓名，为了实现这一要求，同时需要关系 *loan* 和关系 *borrower* 中的信息。如果如下所写

$$\sigma_{branch\text{-}name = \text{"Perryridge"}} (borrower \times loan)$$

其结果就是图 3-13 所示关系，这里得到的关系中的信息都属于 Perryridge 分支机构。然而，*customer-name* 列却可能包含在 Perryridge 分支机构中没有贷款的客户。(如果你不明白为什么会发生这种情况，不要忘了笛卡儿积中保留了所有可能的由一个来自 *borrower* 的元组和一个来自 *loan* 的元组构成的元组对。)

由于笛卡儿积运算将 *loan* 中的每个元组同 *borrower* 中的每个元组进行联系，而我们又知道如果客户在 Perryridge 分支机构中有贷款，则 *borrower* × *loan* 中必定存在某个元组，其中包含了该客户的姓名，并且 *borrower*.*loan-number* = *loan*.*loan-number*。因此，如果用

$$\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number} (\sigma_{branch\text{-}name = \text{"Perryridge"}} (borrower \times loan))$$

就可以只得到在 *borrower* × *loan* 中且与在 Perryridge 分支机构中有贷款的客户相关的所有元组。

最后，由于只需要 *customer-name*，进行投影运算：

customer-name	*borrower. loan-number*	*branch-name*	*loan. loan-number*	*amount*
Jones	L-17	Downtown	L-17	1000
Jones	L-17	Redwood	L-23	2000
Jones	L-17	Perryridge	L-15	1500
Jones	L-17	Downtown	L-14	1500
Jones	L-17	Mianus	L-93	500
Jones	L-17	Round Hill	L-11	900
Jones	L-17	Perryridge	L-16	1300
Smith	L-23	Downtown	L-17	1000
Smith	L-23	Redwood	L-23	2000
Smith	L-23	Perryridge	L-15	1500
Smith	L-23	Downtown	L-14	1500
Smith	L-23	Mianus	L-93	500
Smith	L-23	Round Hill	L-11	900
Smith	L-23	Perryridge	L-16	1300
Hayes	L-15	Downtown	L-17	1000
Hayes	L-15	Redwood	L-23	2000
Hayes	L-15	Perryridge	L-15	1500
Hayes	L-15	Downtown	L-14	1500
Hayes	L-15	Mianus	L-93	500
Hayes	L-15	Round Hill	L-11	900
Hayes	L-15	Perryridge	L-16	1300
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Williams	L-17	Downtown	L-17	1000
Williams	L-17	Redwood	L-23	2000
Williams	L-17	Perryridge	L-15	1500
Williams	L-17	Downtown	L-14	1500
Williams	L-17	Mianus	L-93	500
Williams	L-17	Round Hill	L-11	900
Williams	L-17	Perryridge	L-16	1300
Adams	L-16	Downtown	L-17	1000
Adams	L-16	Redwood	L-23	2000
Adams	L-16	Perryridge	L-15	1500
Adams	L-16	Downtown	L-14	1500
Adams	L-16	Mianus	L-93	500
Adams	L-16	Round Hill	L-11	900
Adams	L-16	Perryridge	L-16	1300

图 3-12 *borrower* × *loan* 的结果

customer-name	*loan-number*	*branch-name*	*loan-number*	*amount*
Jones	L-17	Perryridge	L-15	1500
Jones	L-17	Perryridge	L-16	1300
Smith	L-23	Perryridge	L-15	1500
Smith	L-23	Perryridge	L-16	1300
Hayes	L-15	Perryridge	L-15	1500
Hayes	L-15	Perryridge	L-16	1300
Jackson	L-14	Perryridge	L-15	1500
Jackson	L-14	Perryridge	L-16	1300
Curry	L-93	Perryridge	L-15	1500
Curry	L-93	Perryridge	L-16	1300
Smith	L-11	Perryridge	L-15	1500
Smith	L-11	Perryridge	L-16	1300
Williams	L-17	Perryridge	L-15	1500
Williams	L-17	Perryridge	L-16	1300
Adams	L-16	Perryridge	L-15	1500
Adams	L-16	Perryridge	L-16	1300

图 3-13 $\sigma_{branch\text{-}name = \text{"Perryridge"}}$ (*borrower* × *loan*) 的结果

$$\Pi_{customer\text{-}name}\ (\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}\ (\sigma_{branch\text{-}name = \text{"Perryridge"}}\ (borrower \times loan)))$$

此表达式的结果如图 3-14 所示，这就是我们查询的正确答案。

customer-name
Hayes
Adams

图 3-14　$\Pi_{customer\text{-}name}\ (\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}\ (\sigma_{branch\text{-}name = \text{"Perryridge"}}\ (borrower \times loan)))$ 的结果

7. 更名运算

关系代数表达式的结果不像数据库中的关系那样，关系代数表达式的结果没有可供引用的名字。具有可赋给它们名字的能力是很有用的。用小写希腊字母 ρ 表示的更名运算使得我们可以完成这一任务。对给定关系代数表达式 E，表达式

$$\rho_x\ (E)$$

返回表达式 E 的结果，并把名字 x 赋给了它。

关系自身被看作是一个（最小的）关系代数表达式。因此，也可以将更名运算运用于关系 r，这样可得到具有新名字的一个相同的关系。

更名运算的另一形式如下。假设关系代数表达式 E 是 n 元的，则表达式

$$\rho_{x(A_1, A_2, \cdots, A_n)}\ (E)$$

返回表达式 E 的结果，并赋给它名字 x，同时将各属性更名为 A_1，A_2，…，A_n。

作为关系更名的例子，来看查询“找出银行中最大的帐户余额”。我们的策略是首先计算出一个由非最大余额构成的临时关系，然后计算关系 $\Pi_{balance}\ (account)$ 和刚才算出的临时关系之间的集合差，以此来得到结果。为了计算该临时关系，需要比较所有帐户余额的值。要作这样的比较，可以通过计算笛卡儿积 $account \times account$ 并构造一个选择来比较任意两个出现在同一元组中的余额。首先，需要区别两个 *balance* 属性的机制，我们将使用更名运算来重新命名帐户关系的引用名字，这样就可以无歧义地两次引用这个关系。

现在，非最大余额构成的临时关系可以写作

$$\Pi_{account.balance}\ (\sigma_{account.balance < d.balance}\ (account \times \rho_d\ (account)))$$

这一表达式给出了关系 *account*（更名为 d）中总有余额比它大的那些余额，此结果中包含了除最大余额以外的所有余额。这个关系如图 3-15 所示。查找银行中最大余额的查询可写作：

$$\Pi_{balance}\ (account) - \Pi_{account.balance}\ (\sigma_{account.balance < d.balance}\ (account \times \rho_d\ (account)))$$

该查询的结果如图 3-16 所示。

balance
500
700
400
350
750

图 3-15　子表达式 $\Pi_{account.balance}\ (\sigma_{account.balance < d.balance}\ (account \times \rho_d\ (account)))$ 的结果

balance
900

图 3-16　银行中的最大帐户余额

再举一个关于更名运算的例子。查询要求是“找出所有与 Smith 居住在同一城市同一街道的客户”。可以通过如下表达式得到 Smith 居住的城市和街道

$$\Pi_{customer\text{-}street,\ customer\text{-}city}\ (\sigma_{customer\text{-}name = \text{"Smith"}}\ (customer))$$

可是，要找出居住在这个城市这条街道的其他客户，必须再次引用关系 *customer*。在下面的查询中，我们对上面的表达式做更名运算，给此表达式的结果以名字 smith-addr，并将其属性更名为 *street* 和 *city*，替代原有的 *customer-street* 和 *customer-city*：

$$\Pi_{customer.customer\text{-}name}$$

$$(\sigma_{customer.customer\text{-}street = smith\text{-}addr.street \wedge customer.customer\text{-}city = smith\text{-}addr.city}$$

$$(customer \times \rho_{smith\text{-}addr(street, city)}$$

$$(\Pi_{customer\text{-}street, customer\text{-}city} (\sigma_{customer\text{-}name = \text{“Smith”}} (customer)))))$$

如果对图 3-3 所示的关系 *customer* 运用此查询，则结果如图 3-17 所示。

customer-name
Smith
Curry

图 3-17 与 Smith 居住在同一城市同一街道的客户

更名运算不是必须的，因为可以对属性使用位置标记。可以用位置标记隐含地作为关系的属性名，用 $1，$2，…指代第一个属性，第二个属性，等等。位置标记也适用于关系代数表达式运算的结果。下面的关系代数表达式示例了一元运算 σ 中位置标记的用法：

$$\sigma_{\$2 = \$3} (R \times R)$$

如果一个二元运算需要在作为其操作数的两个关系之间进行区分，类似的位置标记也可以用来作为关系名称。例如，$R1 可以指代第一个操作数，$R2 可以指代第二个操作数。但是，对人而言位置标记不够方便，因为属性的位置是一个数字，不像属性名那样易于记忆。因此，本书里不采用位置标记。

3.2.2 关系代数的形式化定义

在 3.2.1 节中看到的运算使得可以给出关系代数表达式的完整定义。关系代数中基本表达式由如下之一构成：

- 数据库中的一个关系。
- 一个常量关系。

通常，关系代数中的表达式由更小的子表达式构成。设 E_1 和 E_2 是关系代数表达式，则下面的都是关系代数表达式：

- $E_1 \cup E_2$。
- $E_1 - E_2$。
- $E_1 \times E_2$。
- $\sigma_P (E_1)$。其中 P 是 E_1 中属性上的谓词。
- $\Pi_S (E_1)$。其中 S 是 E_1 中某些属性的列表。
- $\rho_x (E_1)$。其中 x 是 $E1$ 结果的新名字。

3.2.3 附加运算

关系代数的基本运算足以表达任何关系代数查询。但是，如果把自己只局限于基本运算，某些常用查询表达出来会显得过于冗长。因此，定义一些附加运算，它们不能增加关系代数的表达能力，但却可以简化一些常用的查询。对每个新运算，给出只采用基本运算的等价表达式。

1. 集合交运算

要定义的第一个附加关系代数运算是集合交（∩）。假设希望找出所有既有帐户又有贷款

的客户，使用集合交运算，可以写为

$$\Pi_{customer\text{-}name}(borrower) \cap \Pi_{customer\text{-}name}(depositor)$$

此查询产生的关系如图 3-18 所示。

customer-name
Hayes
Jones
Smith

图 3-18　银行中既有帐户又有贷款的客户

注意，任何使用了集合交的关系代数表达式都可以通过如下的一对集合差运算替代集合交运算来重写：

$$r \cap s = r - (r - s)$$

因此，集合交不是基本运算，不能增加关系代数的表达能力，这样做只不过因为写 $r \cap s$ 比写 $r - (r - s)$ 更方便。

2. 自然连接运算

对某些要用到笛卡儿积的查询进行简化常常是我们需要做的。通常情况下，涉及笛卡儿积的查询中会包含一个对笛卡儿积结果进行选择的运算。来看一下查询“找出所有在银行中有贷款的客户姓名及相应贷款金额”。首先，要产生关系 *borrower* 和 *loan* 的笛卡儿积。接着，只选出那些属于同一 *loan-number* 的元组，然后投影到 *customer-name*、*loan-number* 和 *amount*：

$$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan))$$

二元运算自然连接使得可以将某些选择和笛卡儿积运算合并为一个运算。用“连接”符号 $\bowtie$ 来表示它。自然连接运算首先形成它的两个参数的笛卡儿积，然后基于两个关系模式中都出现的属性上的相等性进行选择，最后还要去除重复属性。

尽管自然连接的定义很复杂，这种运算使用起来却很方便。举例来说，再看一下查询“找出所有在银行中有贷款的客户姓名及相应贷款金额”，用自然连接，可以将此查询表述如下：

$$\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}(borrower \bowtie loan)$$

由于 *borrower* 和 *loan* 的模式（即 *Borrower-schema* 和 *Loan-schema*）中具有相同属性 *loan-number*，自然连接运算只考虑在 *loan-number* 上值相同的元组对。它将每个这样的元组对合并为单一的元组，其模式为两个模式的并（即 *customer-name*、*branch-name*、*loan-number* 和 *amount*）。执行投影后，得到的关系如图 3-19 所示。

customer-name	loan-number	amount
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Curry	L-93	500
Smith	L-11	900
Williams	L-17	1000
Adams	L-16	1300

图 3-19　$\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}(borrower \bowtie loan)$ 的结果

考虑两个关系模式 R 和 S——当然，它们都是属性名的一个序列。如果认为模式是集合而不是序列，可以用 $R \cap S$ 表示同时出现在 R 和 S 中的那些属性名，用 $R \cup S$ 表示出现在 R 中或 S 中或在二者中都出现的那些属性名。同样，出现在 R 中而不出现在 S 中的属性名用 $R - S$ 表示，出现在 S 中而不出现在 R 中的属性名用 $S - R$ 表示。注意这里的并、交、差运算都是建立在属性的集合上的，而不是关系上的。

现在已经可以给出自然连接的形式化定义。设 r (R) 和 s (S) 是两个关系，r 和 s 的自然连接表示为 $r \bowtie s$，是模式 $R \cup S$ 上的一个关系，其形式化定义如下：

$$r \bowtie s = \Pi_{R \cup S}\ (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \cdots \wedge r.A_n = s.A_n} r \times s)$$

其中 $R \cap S = \{A_1, A_2, \cdots, A_n\}$。

由于自然连接在关系数据库理论和实践中的中心地位，在这里给出一些使用自然连接的例子。

• 找出含有居住在 Harrison 并在银行中有帐户的那些客户的所有分支机构的名称。

$$\Pi_{branch\text{-}name}\ (\sigma_{customer\text{-}city = \text{"Harrison"}}\ (customer \bowtie account \bowtie depositor))$$

此查询的结果关系如图 3-20 所示。

上式中我们在 *customer* ⋈ *account* ⋈ *depositor* 中没有加入括号来表明自然连接在这三个关系间执行的顺序。上述情况有两种可能：

• (*customer* ⋈ *account*) ⋈ *depositor*。

• *customer* ⋈ (*account* ⋈ *depositor*)。

branch-name
Brighton
Perryridge

图 3-20 $\Pi_{branch\text{-}name}\ (\sigma_{customer\ city = \text{"Harrison"}}\ (customer \bowtie account \bowtie depositor))$ 的结果

我们没有说明希望的是哪个表达式，因为二者是等价的。也就是说，自然连接是可结合的。

• 找出所有在银行中既有贷款又有帐户的客户。

$$\Pi_{customer\text{-}name}\ (borrower \bowtie depositor)$$

在第 1 部分中曾用集合交表达过此查询，在这里重复一下用集合交写出的表达式。

$$\Pi_{customer\text{-}name}\ (borrower) \cap \Pi_{customer\text{-}name}\ (depositor)$$

此查询结果在图 3-18 已经展示过。这一例子说明了关系代数中一个普遍的事实：写出彼此差别很大的几个等价关系代数表达式是可能的。

• 设 r (R) 和 s (S) 是没有任何公共属性的关系，即 $R \cap S = \varnothing$（$\varnothing$表示空集）。那么，$r \bowtie s = r \times s$。

theta 连接是自然连接的扩展，它使得可以把一个选择运算和一个笛卡儿积运算合并为单独一个运算。考虑关系 r (R) 和 s (S)，并设 θ 是模式 $R \cup S$ 属性上的谓词，则 theta 连接运算 $r \bowtie_\theta s$ 定义如下：

$$r \bowtie_\theta s = \sigma_\theta\ (r \times s)$$

3. 除运算

除运算用 ÷ 表示，适合于包含了“对所有的”此类短语的查询。假设希望找出在 Brooklyn 的所有分支机构都有帐户的客户。可用如下表达式得到位于 Brooklyn 的所有分支机构

$$r_1 = \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch))$$

此表达式产生的关系如图 3-21 所示。

可以通过下面的表达式找出客户在分支机构有贷款的所有 (*customer-name*, *branch-name*) 对

$$r_2 = \Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie account)$$

此表达式产生的结果如图 3-22 所示。

branch-name
Brighton
Downtown

图 3-21 $\Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch))$ 的结果

现在需要找出这样的客户，他与 r_1 中每个分支机构名称的结对都在 r_2 中出现。恰好给出所有这样客户的运算是除运算。

把此查询表述为：

$$\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie account)$$
$$\div \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch))$$

此表达式产生的关系模式为（*customer-name*），其中包含元组（Johnson）。

customer-name	*branch-name*
Johnson	Downtown
Smith	Mianus
Hayes	Perryridge
Turner	Round Hill
Lindsay	Redwood
Johnson	Brighton
Jones	Brighton

图 3-22　$\Pi_{customer\text{-}name,\ branch\text{-}name}$（*depositor* ⋈ *account*）的结果

形式化地说，设 r（R）和 s（S）是两个关系，并且 $S \subseteq R$ ——即模式 S 中的每个属性都在模式 R 中。关系 $r \div s$ 是模式 $R-S$ 上的关系，即此模式中包含所有在 R 中而不在 S 中的属性。元组 t 属于 $r \div s$ 当且仅当以下两个条件同时成立：

1）t 在 $\Pi_{R-S}(r)$ 中。

2）对 s 中的每一个元组 t_s，在 r 中都有元组 t_r 同时满足以下两个条件：

a）$t_r[S] = t_s[S]$

b）$t_r[R-S] = t$

实际上给定除运算和关系模式，可以用基本运算来定义除运算，当你发现这一事实时可能会感到惊讶。设 r（R）和 s（S）已知，且 $S \subseteq R$：

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

来看此表达式的正确性。不难发现，$\Pi_{R-S}(r)$ 给出了满足除运算定义中第一个条件的所有元组 t。而集合差运算符右侧的表达式

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

用来去掉那些不能满足除运算定义中第二个条件的元组。我们来看这是怎样实现的：先来看 $\Pi_{R-S}(r) \times s$，这是模式 R 上的关系，将 $\Pi_{R-S}(r)$ 中每个元组分别同 s 中每个元组配对。表达式 $\Pi_{R-S,S}(r)$ 只是重新排列 r 的属性顺序。因此，$(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ 从 $\Pi_{R-S}(r)$ 和 s 中得出不在 r 中出现的元组对。如果元组 t_j 在

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

中，则一定存在 s 中的某个元组 t_s 不能和 t_j 一起组成 r 中的元组。因此，t_j 中含有属性 $R-S$ 上的在 $r \div s$ 中不出现的一个值。我们恰恰需要从 $\Pi_{R-S}(r)$ 中去除这样的值。

4. 赋值运算

有时通过给临时关系变量赋值，可以将关系代数表达式分开一部分一部分地写，这样会比较方便。赋值运算用←表示，与程序设计中的赋值相类似。为了说明这个运算，来看上面第 3 部分所给的除的定义。可以把 $r \div s$ 写作

$$temp1 \leftarrow \Pi_{R-S}(r)$$

$$temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - r)$$

$$result = temp1 - temp2$$

赋值的执行不会导致把某个关系显示给用户看，而是将←右侧的表达式的结果赋给←左侧的关系变量。该关系变量可以在后续的表达式中使用。

使用赋值运算，可以把查询表达为一个顺序程序，该程序由一系列赋值加上一个其值被作为查询结果显示的表达式组成。对关系代数查询而言，赋值必须是赋给一个临时关系变量。对永久关系的赋值即是对数据库的修改，我们在 3.6 节讨论这个问题。注意，赋值运算不能增加关系代数的表达能力，但是可以使复杂查询的表达变得简单。

3.3　元组关系演算

当书写关系代数表达式时，我们提供了产生查询结果的过程序列。与之相反，元组关系演

算是非过程化查询语言。它只是描述所需信息，而不给出获得该信息的具体过程。

元组关系演算中的查询表达为：

$$\{t \mid P(t)\}$$

即，它是所有使谓词 P 为真的元组 t 的集合。和前面的记法一样，我们用 $t[A]$ 表示元组 t 在属性 A 上的值，并用 $t \in r$ 表示元组 t 在关系 r 中。

在给出元组关系演算的形式化定义之前，先回头看几个在3.2节中用关系代数表达式书写过的查询。

3.3.1 查询的例子

我们希望找出所有贷款额在$1200以上的贷款的 *branch-name*、*loan-number* 和 *amount*：

$$\{t \mid t \in loan \wedge t[amount] > 1200\}$$

假设只需要属性 *loan-number*，而不是需要关系 *loan* 的所有属性。为了用元组关系演算来书写这个查询，需要为模式（*loan-number*）上的关系写一个表达式，此外还需要（*loan-number*）上的、在关系 *loan* 中对应属性 *amount*>1200 的那些元组。为了表述这样的要求，需要引入数理逻辑中的“存在”这一结构。记法

$$\exists t \in r\ (Q(t))$$

表示“关系 r 中存在元组 t 使谓词 $Q(t)$ 为真”。

用这种记法，可以将查询“找出贷款额大于$1200的所有贷款的号码”表述为：

$$\{t \mid \exists s \in loan\ (t[loan\text{-}number] = s[loan\text{-}number] \wedge s[amount] > 1200)\}$$

可以这样来读上述表达式：“它是所有元组 t 的集合，元组 t 满足：在关系 *loan* 中存在元组 s 使 t 和 s 在属性 *loan-number* 上的值相等，且 s 在属性 *amount* 上的值大于1200。”

元组变量 t 只定义在属性 *loan-number* 上，因为这一属性是对 t 进行限制的条件涉及的唯一属性，因此，结果得到（*loan-number*）上的关系。

请看查询“找出从Perryridge分支机构贷款的所有客户姓名”。这个查询比前一个稍微复杂一些，因为它涉及 *borrower* 和 *loan* 两个关系。但是，如下所述，我们所做的只不过是在元组关系演算中使用两个用 *and*（∧）连接起来的“存在”子句。可将此查询表述如下：

$$\{t \mid \exists s \in borrower\ (t[customer\text{-}name] = s[customer\text{-}name] \wedge \exists u \in loan\ (u[loan\text{-}number] = s[loan\text{-}number] \wedge u[branch\text{-}name] = \text{“Perryridge”}))\}$$

可以这样来读上述表达式：“它是所有满足一定条件的（*customer-name*）元组的集合，条件是这些客户都在Perryridge分支机构有贷款”。元组变量 u 保证客户是Perryridge分支机构的一个贷款人。元组变量 s 被限制与 u 的贷款号码相同。查询产生的结果如图3-23所示。

customer-name
Hayes
Adams

图3-23 在Perryridge分支机构有贷款的所有客户姓名

为了找出在银行有贷款或有帐户或二者兼具的所有客户，在关系代数中使用了并运算。在元组关系演算中，将用到两个用 *or*（∨）连接的“存在”子句：

$$\{t \mid \exists s \in borrower\ (t[customer\text{-}name] = s[customer\text{-}name]) \vee \exists u \in depositor\ (t[customer\text{-}name] = u[customer\text{-}name])\}$$

此表达式给出所有 *customer-name* 元组的集合，它们至少满足下面两个条件中的一个：

- *customer-name* 作为贷款人出现在关系 *borrower* 的某个元组中。

• *customer-name* 作为存款人出现在关系 *depositor* 的某个元组中。

如果某客户在银行既有贷款又有帐户，在结果中也只出现一次，因为集合的数学定义不允许重复。此查询的结果已在图 3-10 中展示。

假设现在只需要在银行中既有贷款又有帐户的客户，所做的只不过是把上述表达式中的 *or*（∨）用 *and*（∧）替代。

$$\{t \mid \exists s \in borrower\ (t\ [customer\text{-}name] = s\ [customer\text{-}name])$$
$$\wedge\ \exists u \in depositor\ (t\ [customer\text{-}name] = u\ [customer\text{-}name])\}$$

此查询结果如图 3-18 所示。

现在考虑查询“找出在银行中有帐户而无贷款的所有客户”。这一查询的元组关系演算表达式同上面的表达式类似，只是使用了 *not*（¬）符号：

$$\{t \mid \exists u \in depositor\ (t\ [customer\text{-}name] = u\ [customer\text{-}name])$$
$$\wedge\ \neg \exists s \in borrower\ (t\ [customer\text{-}name] = s\ [customer\text{-}name])\}$$

这个元组关系演算表达式用 $\exists u \in depositor$ (…) 子句限制客户在银行中必须有帐户，用 $\neg \exists s \in borrower$ (…) 子句来去掉作为贷款人出现在 *borrower* 关系某个元组中的那些客户。此查询产生的结果如图 3-11 所示。

下面要考虑的查询将用到蕴含，用⇒表示。公式 $P \Rightarrow Q$ 表示“P 蕴含 Q”，即“如果 P 为真，则 Q 必然为真”。$P \Rightarrow Q$ 逻辑上等价于 $\neg P \vee Q$。用蕴含而不用 *not* 和 *or* 常常可以更直观地表达查询。

来看 3.2.3 节中除运算示例的查询：“找出在 Brooklyn 的每个分支机构都有帐户的所有客户”。为了用元组关系演算书写此查询，引入“对所有的……”结构，用 ∀ 表示。记法

$$\forall t \in r\ (Q\ (t))$$

表示“对关系 r 中的所有元组 t，Q 均为真”。

此查询的表达式如下：

$$\{t \mid \forall u \in branch\ (u\ [branch\text{-}city] = \text{“Brooklyn”} \Rightarrow$$
$$\exists s \in depositor\ (t\ [customer\text{-}name] = s\ [customer\text{-}name]$$
$$\wedge\ \exists w \in account\ (w\ [account\text{-}number] = s\ [account\text{-}number]$$
$$\wedge\ w\ [branch\text{-}name] = u\ [branch\text{-}name]))) \}$$

可以这样来读上述表达式：“它是所有满足一定条件的客户（即（*customer-name*）上的元组 t）的集合，这些客户满足：对关系 *branch* 中的所有元组 u，如果 u 在属性 *branch-city* 上的值是 Brooklyn，则此客户在 u 的属性 *branch-name* 所代表的分支机构中一定有帐户。”

3.3.2　形式化定义

现在可以给出元组关系演算的形式化定义。元组关系演算表达式具有如下形式：

$$\{t \mid P\ (t)\}$$

其中 P 是公式。公式中可以出现多个元组变量。如果元组变量不被 ∃ 或 ∀ 约束，则称为自由变量。因此，在

$$t \in loan \wedge \exists s \in customer\ (t\ [branch\text{-}name] = s\ [branch\text{-}name])$$

中，t 是自由变量，而 s 称为约束变量。

元组关系演算的公式由原子构成。原子可以是如下形式之一：

• $s \in r$。其中 s 是元组变量而 r 是关系（我们不允许使用∈运算符）。

• $s\ [x]\ \Theta u\ [y]$。其中 s 和 u 是元组变量，x 是 s 所基于的关系模式中的属性，y 是 u

所基于的关系模式中的属性，Θ 是比较运算符（$<$，$\leqslant$，$=$，$\neq$，$>$，$\geqslant$）。我们要求属性 x 和 y 属性域的成员能用 Θ 比较。

- $s[x]\ \Theta c$。其中 s 是元组变量，x 是 s 所基于的关系模式中的属性，Θ 是比较运算符，c 是属性 x 属性域中的常量。

根据如下规则，用原子构造公式：

- 原子是公式。
- 如果 P_1 是公式，那么 $\neg P_1$ 和 (P_1) 也都是公式。
- 如果 P_1 和 P_2 是公式，那么 $P_1 \vee P_2$，$P_1 \wedge P_2$ 和 $P_1 \Rightarrow P_2$ 也都是公式。
- 如果 $P_1(s)$ 是包含自由元组变量 s 的公式，且 r 是关系，则

$$\exists s \in r\ (P_1(s)) \text{ 和 } \forall s \in r\ (P_1(s))$$

也都是公式。

与关系代数一样，也可以写出一些形式上不一样的等价表达式。在元组关系演算中，这种等价性包括如下三条规则：

1) $P_1 \wedge P_2$ 等价于 $\neg(\neg P_1 \vee \neg P_2)$。

2) $\forall t \in r\ (P_1(t))$ 等价于 $\neg \exists t \in r\ (\neg P_1(t))$。

3) $P_1 \Rightarrow P_2$ 等价于 $\neg P_1 \vee P_2$。

3.3.3 表达式的安全性

最后还要讨论一个问题。元组关系演算表达式可能产生一个无限关系。例如如果写出表达式

$$\{t \mid \neg(t \in loan)\}$$

而不在 *loan* 中的元组无限多，大多数这些元组所包含的值甚至根本不在数据库中！显然，我们不希望有这样的表达式。

为了帮助我们对元组关系演算进行限制，引入了元组关系公式 P 的域这一概念。直观地说，P 的域用 $dom(P)$ 表示，是 P 所引用的所有值的集合。其中既包括 P 自身用到的值，又包括 P 用到的关系的元组中出现的所有值。因此，P 的域是 P 中显式出现的值及关系名称在 P 中出现的那些关系的所有值的集合。例如，$dom(t \in loan \wedge t[amount] > 1200)$ 是包括 1200 和出现在 *loan* 中的所有值的集合。另外，$dom(\neg(t \in loan))$ 是 *loan* 中出现的所有值的集合，因为关系 *loan* 在表达式中出现。

如果出现在表达式 $\{t \mid P(t)\}$ 结果中的所有值均来自 $dom(P)$，则说表达式 $\{t \mid P(t)\}$ 是安全的。表达式 $\{t \mid \neg(t \in loan)\}$ 不安全，因为 $dom(\neg(t \in loan))$ 是所有出现在 *loan* 中的值的集合，但是很可能有某个不在 *loan* 中的元组 t，它包含有不在 *loan* 中出现的值。本节所写的其他元组关系演算表达式都是安全的。

3.3.4 语言的表达能力

限制在安全表达式范围内的元组关系演算和关系代数具有相同的表达能力。因此，对于每个关系代数表达式，都有与之等价的元组关系演算表达式；对于每个元组关系演算表达式，也都有与之等价的关系代数表达式。我们在此不证明这个论断，文献注解中有关于此论断证明的信息，习题里也包含了部分证明。

3.4 域关系演算

关系演算的另一种形式称为域关系演算，其中采用的是域变量。域变量从属性的域中取

值，而不是取整个元组的值。尽管如此，域关系演算同元组关系演算联系紧密。

3.4.1 形式化定义

域关系演算的表达式形式如下：

$$\{<x_1, x_2, \cdots, x_n> | P(x_1, x_2, \cdots, x_n)\}$$

其中 x_1，x_2，…，x_n 代表域变量。和元组关系演算的情况一样，P 代表由原子构成的公式。域关系演算中的原子具有如下形式之一：

- $<x_1, x_2, \cdots, x_n> \in r$，其中 r 是 n 个属性上的关系，而 x_1，x_2，$\cdots x_n$ 是域变量或域常量。
- $x \Theta y$，其中 x 和 y 是域变量，Θ 是比较运算符（$<$，$\leqslant$，$=$，$\neq$，$>$，$\geqslant$）。我们要求属性 x 和 y 属性域可用 Θ 比较。
- $x \Theta c$，其中 x 是域变量，Θ 是比较运算符，c 是 x 作为域变量的那个属性域中的常量。

根据如下规则用原子构造公式：

- 原子是公式。
- 如果 P_1 是公式，那么 $\neg P_1$ 和 (P_1) 也都是公式。
- 如果 P_1 和 P_2 是公式，那么 $P_1 \vee P_2$，$P_1 \wedge P_2$ 和 $P_1 \Rightarrow P_2$ 也都是公式。
- 如果 $P_1(x)$ 是 x 的一个公式，其中 x 是域变量，则

$$\exists x (P_1(x)) \text{ 和 } \forall x (P_1(x))$$

也都是公式。

把 $\exists a, b, c (P(a, b, c))$ 记为 $\exists a (\exists b (\exists c (P(a, b, c))))$ 的简写。

3.4.2 查询的例子

现在对前面讲到的例子用域关系演算来给出查询表达。注意这些表达式和元组关系演算表达式的相似之处。

- 找出贷款额在 $1200 以上的贷款的分支机构、贷款号码和贷款金额：

$$\{<b, l, a> | <b, l, a> \in loan \wedge a > 1200\}$$

- 找出所有贷款额在 $1200 以上的贷款号码：

$$\{<l> | \exists b, a (<b, l, a> \in loan \wedge a > 1200)\}$$

虽然第二个查询看起来同在元组关系演算中书写的很相似，但实际上有重要区别。在元组关系演算中，当对某个元组变量 s 写出 $\exists s$ 时，立刻通过 $\exists s \in r$ 将它同某个关系束缚在一起。可是，当在域演算中写 $\exists b$ 时，b 不指代一个元组，而指的是某个域值。因此，在用子公式 $<b, 1, a> \in loan$ 将 b 约束给 $loan$ 关系中的分支机构名称以前，b 是不受约束的。

- 找出在 Perryridge 分支机构有贷款的所有客户及相应贷款金额：

$$\{<c, a> | \exists l (<c, l> \in borrower \wedge \exists b (<b, l, a> \in loan \wedge b = \text{“Perryridge”}))\}$$

- 找出在 Perryridge 分支机构有帐户或有贷款或二者兼具的所有客户姓名：

$$\{<c> | \exists l (<c, l> \in borrower \wedge \exists b, a (<b, l, a> \in loan \wedge b = \text{“Perryridge”})) \vee \exists a (<c, a> \in depositor \wedge \exists b, n (<b, a, n> \in account \wedge b = \text{“Perryridge”}))\}$$

- 找出在 Brooklyn 的每个分支机构均有帐户的所有客户姓名：

$$\{<c> | \forall x, y, z (<x, y, z> \in branch) \wedge y = \text{“Brooklyn”} \Rightarrow \exists a, b (<x, a, b> \in account \wedge <c, a> \in depositor)\}$$

这样来读这个表达式：它是所有（*customer-name*）元组 c 的集合，c 且满足对所有的（*branch-name*、*branch-city*、*assets*）元组 x，y，z，如果分支机构所在的城市是 Brooklyn，则下面两个条件同时成立：

- 在关系 *account* 中存在元组，其帐户号为 a 且分支机构名为 x。
- 在关系 *depositor* 中存在元组，其客户为 c 且帐户号为 a。

3.4.3 表达式的安全性

我们知道，在元组关系演算中可能会写出其结果为无限关系的表达式，这促使我们定义元组关系演算表达式的安全性。域关系演算中也有类似的情况，像

$$\{<b, l, a> \mid \neg(<b, l, a> \in loan)\}$$

这样的表达式就是不安全的，因为在它的结果中允许不在表达式域中的值出现。

在域关系演算中，还必须考虑“存在”和“对所有的”这类子句中公式的形式。来看表达式

$$\{<x> \mid \exists y\ (<x, y> \in r) \wedge \exists z\ (\neg(<x, z> \in r) \wedge P(x, z))\}$$

其中 P 是涉及 x 和 z 的公式。对于公式的第一部分，在测试时只需考虑 r 中的值。可是，对于公式的第二部分，在测试时必须考虑不在 r 中出现的值。由于所有关系都是有限的，不在 r 中出现的值无限多，因此，通常情况下如果不考虑无限多个可能的 z 值，就不能完成对第二部分的测试。事实上我们并不这样做，而是通过增加一些约束来限制像上面这样的表达式。

在元组关系演算中，将所有存在量词约束的变量限制在某个关系范围内。由于在域关系演算中没有这样做，我们增加用于定义安全性的规则，以处理类似上例中出现的情况。表达式

$$\{<x_1, x_2, \cdots, x_n> \mid P(x_1, x_2, \cdots, x_n)\}$$

是安全的是当下列条件同时成立时：

1）表达式的元组中出现的所有值均来自 $dom(P)$。

2）对每个形如 $\exists x(P_1(x))$ 的“存在”子公式而言，子公式为真当且仅当在 $dom(P_1)$ 中有某个值 x 使 $P_1(x)$ 为真。

3）对每个形如 $\forall x(P_1(x))$ 的“对所有的”子公式而言，子公式为真当且仅当对 $dom(P_1)$ 中的所有 x，都使 $P_1(x)$ 为真。

附加规则的目的是为了保证不需要测试无限多的可能性就可以完成对“存在”和“对所有的”子公式的测试。来看安全性定义中的第二个规则，要使 $\exists x(P_1(x))$ 为真，只需找到一个 x 使 $P_1(x)$ 为真。通常需要测试无数个值。但是，如果表达式是安全的，就知道可以只注意 $dom(P_1)$ 中的值。这种限制使考虑的元组减少到有限个。

形如 $\forall x(P_1(x))$ 的子公式情况类似。要判断 $\forall x(P_1(x))$ 为真，需要测试所有可能的值，因此必须检查无限多的值。跟前面一样，如果知道表达式是安全的，则只需要用 $dom(P_1)$ 中的值来测试 $P_1(x)$ 已经足够。

本节例子中所给的域关系演算表达式都是安全的。

3.4.4 语言的表达能力

限制在安全表达式范围内的域关系演算同限制在安全表达式范围内的元组关系演算等价。前面已经知道受限制的元组关系演算与关系代数等价，所以下述三者都是等价的：

- 关系代数。

- 限制在安全表达式范围内的元组关系演算。
- 限制在安全表达式范围内的域关系演算。

3.5　扩展关系代数运算

人们对基本关系代数运算在多个方面进行了扩展。一个简单扩展是允许将算术运算作为投影的一部分。一个重要的扩展是允许聚集运算，例如计算集合中元素的和或它们的均值。另一个重要扩展是外连接运算，它使得关系代数表达式可以对表示缺失信息的空值进行处理。

3.5.1　广义投影

广义投影运算通过允许在投影列表中使用算术函数来对投影进行扩展。广义投影运算形式为

$$\Pi_{F_1, F_2, \cdots, F_n}(E)$$

其中 E 是任意关系代数表达式，而 F_1，F_2，…，F_n 中的每一个都是涉及 E 模式中常量和属性的算术表达式。特别的，算术表达式可以仅仅是个属性或常量。

下面的例子说明了为什么要使用广义投影运算。假设我们有如图 3-24 所示的关系 *credit-info*，它列出了信贷额度和目前为止的花费（帐户上的 *credit-balance*）。如果希望找出每个客户还能花费多少，可以表述为：

$$\Pi_{customer\text{-}name,\ limit\ -\ credit\text{-}balance}(credit\text{-}info)$$

对图 3-24 所示关系应用上述表达式的结果如图 3-25 所示。

customer-name	*limit*	*credit-balance*
Jones	6000	700
Smith	2000	400
Hayes	1500	1500
Curry	2000	1750

图 3-24　关系 *credit-info*

customer-name	*limit* − *credit-balance*
Jones	5300
Smith	1600
Hayes	0
Curry	250

图 3-25　$\Pi_{customer\text{-}name,\ limit\ -\ credit\text{-}balance}(credit\text{-}info)$ 的结果

3.5.2　外连接

外连接运算是连接运算的扩展，可以处理缺失信息。假设包含全时工作的员工信息的关系模式如下：

employee (*employee-name*, *street*, *city*)

ft-works (*employee-name*, *branch-name*, *salary*)

考虑图 3-26 中的关系 *employee* 和 *ft-works*。假设想得到一个包含有关全时工作的员工的所有信息（街道、城市、分支机构名称和工资）的关系，一种可能的方式是采用自然连接：

$$employee \bowtie ft\text{-}works$$

此表达式结果如图 3-27 所示。不难发现，我们丢失了 Smith 的街道和城市信息，因为在关系 *ft-works* 中没有描述 Smith 的元组；同样，还丢失了 Gates 的分支机构名称和工资信息，因为关系 *employee* 中没有描述Gates 的元组。

employee-name	*street*	*city*
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

employee-name	*branch-name*	*salary*
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

图 3-26　关系 *employee* 和 *ft-works*

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500

图 3-27 *employee* ⋈ *ft-works* 的结果

使用外连接可以避免这样的信息丢失。外连接运算有三种形式：左外连接，用⟕表示；右外连接，用⟖表示；全外连接，用⟗表示。这三种形式的外连接都要计算连接，然后在连接结果上附加额外的元组。表达式 *employee* ⟕ *ft-works*，*employee* ⟖ *ft-works* 和 *employee* ⟗ *ft-works* 的结果分别如图 3-28、图 3-29 和图 3-30 所示。

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*

图 3-28 *employee* ⟕ *ft-works* 的结果

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	*null*	*null*	Redmond	5300

图 3-29 *employee* ⟖ *ft-works* 的结果

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*
Gates	*null*	*null*	Redmond	5300

图 3-30 *employee* ⟗ *ft-works* 的结果

左外连接取出左侧关系中所有与右侧关系的任一元组都不匹配的元组，用空值填充所有来自右侧关系的属性，再把产生的元组加到自然连接的结果上。图 3-28 中，元组（Smith，Revolver，Death Valley，*null*，*null*）即是这样的元组。所有来自左侧关系的信息在左外连接结果中都得到保留。

右外连接与左外连接相对称：用空值填充来自右侧关系的所有与左侧关系的任一元组都不匹配的元组，将产生的元组加到自然连接的结果上。图 3-24 中，元组（Gates，*null*，*null*，Redmond，5300）即是这样的元组。同时，所有来自右侧关系的信息在右外连接结果中都得到保留。

全外连接完成左外连接和右外连接的操作，既填充左侧关系中与右侧关系的任一元组都不匹配的元组，又填充右侧关系中与左侧关系的任一元组都不匹配的元组，并把产生的元组都加到自然连接的结果上。图 3-30 所示为全外连接的结果。

3.5.3 聚集函数

聚集函数输入一个值集合，然后返回单一值作为结果。例如，聚集函数 sum 输入的是一个值集合，返回的是这些值的和。因此，将函数 sum 用于集合

$$\{1, 1, 3, 4, 4, 11\}$$

上，返回值 24。聚集函数 avg 返回平均值，当用于上述集合时，返回值为 4。聚集函数 count 返回集合中元素的个数，对上述集合将返回 6。除此外，常用聚集函数还有 min 和 max，它们分别返回集合中的最小值和最大值，对上面的集合分别返回 1 和 11。

使用聚集函数的集合中，一个值可以出现多次，值出现的顺序是无关紧要的，这样的集合称为多重集。集合是多重集的特例，其中每个值都只出现一次。

为了说明聚集的概念，我们将使用图 3-31 中所示的关系 *pt-works*，该关系包含部分时间工作的员工的信息。假设希望找出银行中所有部分时间工作的员工工资的总和。这一查询的关系代数表达式为：

$$\mathbf{sum}_{salary}\ (pt\text{-}works)$$

此查询的结果是只有一个属性的关系，且只包含单独的一行，表示出银行中所有部分时间工作的员工的工资总和对应的数值。

有时，在计算聚集函数前，必须去除重复值。如果想去除重复，仍然使用以前的函数名，但用连接符将“distinct”附加在函数名后（如：count-distinct）。以查询“找出关系 *pt-works* 中出现的分支机构数”为例。在这里，每个分支机构名只应计算一次，而不管该分支机构中有多少员工。我们把此查询表达如下：

$$\mathbf{count\text{-}distinct}_{branch\text{-}name}\ (pt\text{-}works)$$

对图 3-31 所示关系而言，上述查询的结果是 3。

某些情况下不仅要对一个元组集合使用聚集函数，还要对多个组使用，其中每个组是元组的一个集合。这通过分组运算来完成。例如，我们希望找出各个分支机构的所有部分时间工作的员工工资总和。为了完成此查询，需要根据分支机构将关系 *pt-works* 分组，并对每个组使用聚集函数。

下面的表达式用聚集运算符 $\mathcal{G}$ 完成所需查询：

$$_{branch\text{-}name}\mathcal{G}\ \mathbf{sum}_{salary}\ (pt\text{-}works)$$

此表达式含义如下：$\mathcal{G}$ 左侧的属性 *branch-name* 表明输入关系 *pt-works* 必须按照 *branch-name* 的值进行分组。产生的分组如图 3-32 所示。$\mathcal{G}$ 右侧的表达式 sum_{salary} (*pt-works*) 表明对每组元组（即每个分支机构），聚集函数 sum 必须对属性 salary 的值起作用。输出关系的元组由分支机构名称和该分支机构的工资总和构成，如图 3-33 所示。

employee-name	*branch-name*	*salary*
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Sato	Austin	1600
Rao	Austin	1500
Gopal	Perryridge	5300
Adams	Perryridge	1500
Brown	Perryridge	1300

图 3-31　关系 *pt-works*

employee-name	*branch-name*	*salary*
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Sato	Austin	1600
Rao	Austin	1500
Gopal	Perryridge	5300
Adams	Perryridge	1500
Brown	Perryridge	1300

图 3-32　分组后的关系 *pt-works*

聚集运算 $\mathcal{G}$ 通常的形式如下：

$$_{G_1, G_2, \cdots, G_n}\mathcal{G}_{F_1, A_1, F_2, A_2, \cdots, F_m, A_m}\ (E)$$

其中 E 是任意关系代数表达式，G_1，G_2，…，G_n 是用于分组的一系列属性，F_i 是一个聚集函数，A_i 是一个属性名。运算的含义定义如下，表达式 E 的结果中元组被分成若干组，以使得：

1）同一组中所有元组在 G_1，G_2，…，G_n 上的值相同。

2）不同组中元组在 G_1，G_2，…，G_n 上的值不同。

因此，各组可以用属性 G_1，G_2，…，G_n 上的值来唯一标识。对每个组（g_1，g_2，…，g_n）来说，结果中有一个元组（g_1，g_2，…，g_n，a_1，a_2，…，a_m），其中对每个 i，a_i 是将聚集函数 F_i 作用于该组属性 A_i 上的多重值集所得到的结果。

回到前面的例子，如果在工资总和以外，还希望找出每个分支机构中部分时间工作的员工的最高工资，则表达式为

$$_{branch\text{-}name}\mathcal{G}\ \mathbf{sum}_{salary},\ \mathbf{max}_{salary}\ (pt\text{-}works)$$

此查询的结果如图 3-34 所示。

branch-name	sum of salary
Downtown	5300
Austin	3100
Perryridge	8100

图 3-33 $_{branch\text{-}name}\mathcal{G}\ \mathbf{Sum}_{salary}$（$pt\text{-}works$）的结果

branch-name	sum of salary	max of salary
Downtown	5300	2500
Austin	3100	1600
Perryridge	8100	5300

图 3-34 $_{branch\ name}\mathcal{G}\ \mathbf{sum}_{salary},\ \mathbf{max}_{salary}$（$pt\text{-}works$）的结果

3.6 数据库的修改

目前为止都只是考虑如何从数据库中提取信息。本节讨论如何增加、删除和修改数据库中的信息。

我们用赋值操作来表达数据库的修改。我们采用与 3.2.3 节所描述的赋值相同的记法来给数据库中真正的关系赋值。

3.6.1 删除

删除请求的表达和查询的表达非常相似。不同的是，前者不是要将找出的元组显示给用户，而是要将它们从数据库中去除。这样只能将元组整个地删除，而不能仅删除某些属性上的值。使用关系代数，删除可表达为

$$r \leftarrow r - E$$

其中 r 是关系，E 是关系代数查询。

下面是关系代数中删除请求的几个例子：

- 删除 Smith 的所有帐户记录

$$depositor \leftarrow depositor - \sigma_{customer\text{-}name = \text{“Smith”}}\ (depositor)$$

- 删除贷款额在 0～50 之间的所有贷款

$$loan \leftarrow loan - \sigma_{amount \geqslant 0\ and\ amount \leqslant 50}\ (loan)$$

- 删除位于 Needham 的分支机构的所有帐户

$$r_1 \leftarrow \sigma_{branch\text{-}city = \text{“Needhan”}}\ (account \bowtie branch)$$

$$r_2 \leftarrow \Pi_{branch\text{-}name,\ account\text{-}number,\ balance}\ (r_1)$$

$$account \leftarrow account - r_2$$

在最后一个例子中，通过赋值给临时关系（r_1 和 r_2）来简化表达式。

3.6.2 插入

为了将数据插入关系中，或者指明一个要插入的元组；或者写出一个查询，其结果是要插

入的元组集合。显然，插入元组的属性值必须是属性域中成员。另外，插入的元组必须是正确的。使用关系代数，插入被表达为

$$r \leftarrow r \cup E$$

其中 r 是关系，E 是关系代数表达式。如果让 E 是一个只包含元组的常量关系，就可以表达向关系中插入单一元组。

如果想插入这样的信息：Smith 在 Perryridge 分支机构的帐户 A-973 上有 $1200，则写作：

$$account \leftarrow account \cup \{(\text{“Perryridge”}, \text{A-973}, 1200)\}$$

$$depositor \leftarrow depositor \cup \{(\text{“Smith”}, \text{A-973})\}$$

更通常的情况是基于查询的结果来进行元组插入。假设想对 Perryridge 分支机构的每一个贷款客户赠送一个新的 $200 的存款帐户，并将其贷款号码作为此帐户的号码，则书写

$$r_1 \leftarrow (\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \bowtie loan))$$

$$r_2 \leftarrow \Pi_{branch\text{-}name,\ loan\text{-}number}(r_1)$$

$$account \leftarrow account \cup (r_2 \times \{(200)\})$$

$$depositor \leftarrow depositor \cup \Pi_{customer\text{-}name,\ loan\ number}(r_1)$$

我们不像前面那样来说明一个要插入的元组，而是说明插入关系 *account* 和 *depositor* 的元组集合。插入关系 *account* 的元组具有属性 *branch-name*（Perryridge）、*account-number*（与贷款号码相同）和新帐户的最初余额（$200）。插入关系 *depositor* 的元组具有属性 *customer-name*（其值为得到新帐户的贷款客户的名字）和与对应的 *account* 元组相同的帐户号。

3.6.3 更新

某些情况下，可能希望只改变元组中的某个值，而不希望改变元组中的所有值。可以用广义投影运算来完成此任务：

$$r \leftarrow \Pi_{F_1, F_2, \cdots, F_n}(r)$$

其中 F_i 当第 i 个属性不被修改时是 r 的第 i 个属性，当第 i 个属性将被修改时是一个只涉及常量和 r 的属性的表达式，表达式给出了此属性的新值。

如果希望选出一些元组并只对这些元组进行修改，可以用下述表达式，其中，P 代表用来选择需要修改的元组的条件：

$$r \leftarrow \Pi_{F_1, F_2, \cdots, F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

为了说明更新运算的使用，假设要付给所有帐户 5% 的利息。写出：

$$account \leftarrow \Pi_{branch\text{-}name,\ account\text{-}number,\ balance \leftarrow balance * 1.05}(account)$$

现在假设余额 $10 000 以上的帐户得到 6% 的利息，而其他帐户得到 5% 的利息。写出：

$$account \leftarrow \Pi_{BN, AN, balance \leftarrow balance * 1.06}(\sigma_{balance > 10000}(account))$$
$$\cup \Pi_{BN, AN, balance \leftarrow balance * 1.05}(\sigma_{balance \leqslant 10000}(account))$$

其中 *BN* 和 *AN* 分别是 *branch-name* 和 *account-number* 的缩写。

3.7 视图

目前为止所给的例子中都是在逻辑模型层上操作的。也就是说，假定所给的关系集合是实际存储在数据库中的关系。

我们并不希望让所有用户都看到整个逻辑模型。出于安全性的考虑，可能需要对用户隐藏某些数据。假设有一个人需要知道客户的贷款号码，但不需要知道贷款金额，此人应看到的关

系用关系代数描述如下：

$$\Pi_{customer\text{-}name,\ loan\text{-}number}\ (borrower \bowtie loan)$$

除安全性考虑之外，可能还希望产生比逻辑模型更符合特定用户直觉的个性化的关系集合。例如，广告部门的员工可能希望看见的是由在银行中有帐户或贷款的客户以及与他们有业务往来的分支机构构成的关系。应为该员工建立的关系如下：

$$\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie account)$$

$$\cup\Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie loan)$$

视图是用户可以看见的虚关系，它不是逻辑模型的一部分。任何给定的实际的关系集上都可以支持大量视图。

3.7.1 视图定义

用 create view 语句定义视图。要定义一个视图，必须给出视图名和计算该视图的查询。create view 语句的形式为

create view v **as** <查询表达式>

其中<查询表达式>是任意合法的关系代数表达式。视图名用 v 代表。

以包含分支机构及其客户的视图为例。假设希望此视图名为 *all-customer*，定义此视图如下：

create view *all-customer* **as**

$$\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie account)$$

$$\cup\Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie loan)$$

一旦定义了视图，就可以用视图名指代视图产生的虚关系。利用视图 *all-customer*，可以通过如下表达式找出 Perryridge 分支机构的所有客户：

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name = \text{"Perryridge"}}\ (all\text{-}customer))$$

在 3.2.1 节我们曾经在不用视图的情况下书写过此查询。

只要不是在视图上执行更新运算，视图名就可以出现在关系名可以出现的任何地方。在 3.7.2 节将讨论视图上的更新运算。

视图定义不同于关系代数赋值运算。假设定义关系 r_1 如下：

$$r_1 \leftarrow \Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie account)$$

$$\cup\Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie loan)$$

赋值运算只求一次值，当关系 *depositor*、*account*、*loan* 或 *borrower* 发生变化时，r_1 不再更新。与此相反，视图 *all-customer* 中元组集合在这些关系发生变化时也随之发生变化。直观地说，任意时刻视图关系中的元组集被定义为该时刻定义视图的查询表达式的求值结果。

因此，如果视图被计算并存储，当用来定义视图的关系修改时它就会过时。所以视图通常采取如下实现形式：视图定义时，数据库系统存储视图定义本身，而不是存储定义视图的关系代数表达式的求值结果。查询中只要用到视图的地方，就用存储的查询表达式来替代。因此，只要对查询求值，视图关系就会被重新计算。

某些数据库系统中允许存储视图关系，但它们能够确保定义视图的实际关系改变时，视图始终是最新的，这样的视图称为实体化视图。在某些应用中——如频繁使用某个视图以及在这些应用中，响应时间对某些基于视图的查询至关重要——通过使用实体化视图可以得到好处。从实体化视图中查询可获得的好处与存储代价及更新开销的增加之间必须进行权衡。

3.7.2　通过视图进行更新与空值

尽管视图对查询来说是有用的工具，但用视图表达更新、插入和删除却会带来重大问题，即用视图表达的对数据库的修改必须转换为数据库逻辑模式中实际关系的修改。

为了说明这个问题，以一个想看到关系 *loan* 中除 *loan-number* 以外所有数据的员工为例。假设 *branch-loan* 是给这个员工的视图，将此视图定义为

create view *branch-loan* **as**

$$\Pi_{branch\text{-}name,\ loan\text{-}number}\ (loan)$$

由于允许视图名出现在任何关系名可出现的地方，此员工还可以书写：

$$branch\text{-}loan \leftarrow branch\text{-}loan \cup \{(\text{“Perryridge”},\ \text{L-37})\}$$

此插入必须被表示为对关系 *loan* 的插入，因为视图 *branch-loan* 是在实际关系 *loan* 上构造起来的。然而，要在 *loan* 中插入元组，必须有某个 *amount* 值。处理这样的插入，有两种合理的方式：

- 拒绝插入，返回错误信息给用户。
- 往关系 *loan* 中插入（“Perryridge”，L-37，*null*）。

再看另外一个通过视图进行更新而产生的问题：

create view *loan-info* **as**

$$\Pi_{customer\text{-}name,\ amount}\ (borrower \bowtie loan)$$

此视图列出银行的所有客户及每笔贷款金额。如果有如下通过此视图进行的插入：

$$loan\text{-}info \leftarrow loan\text{-}info \cup \{(\text{“Johnson”},\ 1900)\}$$

往关系 *borrower* 和关系 *loan* 插入元组的唯一可能的方法是将（“Johnson”，*null*）插入 *borrower* 而将（*null*，*null*，1900）插入 *loan*，这样得到图 3-35 所示关系。可是，这样的更新并不能达到预期效果，因为视图关系 *loan-info* 中仍然没有包括元组（“Johnson”，1900）。（不要忘了任何涉及 *null* 的比较都被定义为 false，因而 *borrower* 和 *loan* 的自然连接不包含所需元组。）因此，不能通过使用空值对关系 *borrower* 和 *loan* 进行更新来实现 *loan-info* 所需的更新。

branch-name	*loan-number*	*amount*
Downtown	L-17	1000
Redwood	L-23	2000
Perryridge	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
Round Hill	L-11	900
Perryridge	L-16	1300
null	*null*	1900

customer-name	*loan-number*
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16
Johnson	*null*

图 3-35　插入 *loan* 和 *borrower* 的元组

由于上述问题，除了有限的一些情况以外，对视图关系的修改通常是不允许的。通过视图修改数据库这个普遍的问题一直以来都是研究的题目，文献注解中提到了关于此题目的近期工作。

3.7.3　用视图定义视图

在 3.7.1 节，提到除了在更新表达式中不能使用视图以外，视图关系可以出现在关系可以出现的任何地方。因此，视图可以出现在定义另一视图的表达式中。例如，可以定义视图 *perryridge-customer* 如下：

create view *perryridge-customer* **as**

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name = \text{"Perryride"}}\ (all\text{-}customer))$$

其中 *all-customer* 自身也是视图关系。

如果视图关系 v_2 用在定义视图关系 v_1 的表达式中，则称 v_1 直接依赖于 v_2。图 3-36 中形化地给出了视图间依赖的一个例子。图中表明，视图 *all-customer* 直接依赖于关系 *borrower* 和 *loan*，因为定义 *all-customer* 的表达式中用到这两个视图。同样，视图 *perryridge-customer* 依赖于视图 *all-customer*。图中所示的这样一个图称为依赖图，图中每个视图是一个结点，从 v_2 到 v_1 的有向边表明 v_1 直接依赖于 v_2。

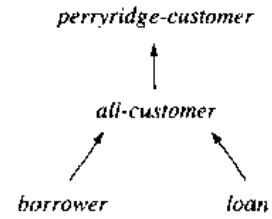


图 3-36 视图的依赖

如果在依赖图中有从 v_2 到 v_1 的路径，则称 v_1 依赖于 v_2。换言之，如果 v_1 直接依赖于 v_2，或者存在视图序列 r_1，r_2，…，r_{n-1}，r_n，满足 v_1 直接依赖于 r_1，r_1 直接依赖于 r_2，依此类推，直到 r_{n-1}直接依赖于 r_n，r_n直接依赖于 v_2，那么就说 v_1 依赖于 v_2。

如果视图依赖于自身，则称此视图关系是*递归*的。形象地说，v 是自环的当且仅当依赖图中有涉及 v 的环。第 5 章将讨论递归视图的含义，目前只是要求视图定义是非递归的。

视图展开是定义视图含义的一种方法，该视图已由其他视图定义。设视图 v_1 由表达式 e_1 定义，e_1 中又可能使用了视图关系。视图关系代表的是定义视图的表达式，因此视图关系可以用定义它的表达式来替换。当用视图定义替换表达式中的视图关系时，得到的结果表达式中可能还含有其他视图关系。因此，表达式的视图展开要多次重复替换步骤，如下：

repeat

找出 e_1 中的任意一个视图关系 v_i

用定义 v_i 的表达式替换视图关系 v_i

until e_1 中不存在视图关系

只要视图定义不是递归的，此循环就会终止。因此，包含视图关系的表达式 e 可以理解为 e 是通过视图展开产生的，不含任何视图关系的表达式。

用下面表达式为例说明视图展开：

$$\sigma_{customer\text{-}name = \text{"John"}}\ (Perryridge\text{-}customer)$$

视图展开过程第一步产生

$$\sigma_{customer\text{-}name = \text{"John"}}\ (\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name = \text{"Perryridge"}}\ (all\text{-}customer)))$$

接着产生

$$\sigma_{customer\text{-}name = \text{"John"}}(\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name = \text{"Perryridge"}}\ (\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie account) \cup \Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie loan))))$$

其中当不再用到视图关系时，视图展开也就结束了。

3.8 总结

关系数据模型建立在表的集合的基础上。数据库系统的用户可以对这些表进行查询，可以插入新元组、删除元组以及更新（修改）元组。表达这些操作的语言有几种。元组关系演算和域关系演算是非过程化语言，代表了关系查询语言所需的基本能力。关系代数是一种过程化语言，在能力上它等价于限制在安全表达式范围内的关系演算的两种形式。关系代数定义了关系查询语言中使用的基本运算。

关系代数和关系演算是简洁的形式化语言，不适合于那些偶尔使用数据库系统的用户。因此，商用数据库系统采用有更多“语法修饰”的语言。第 4 章和第 5 章我们将讨论三种最有影响力的商用语言：SQL、QBE 和 Quel，除此之外还会讲到用于研究的语言 Datalog。

数据库可以通过插入、删除或更新元组来修改。用包含赋值运算符的关系代数来表达这些修改。

共享数据库的不同用户可以得益于数据库的个性化的视图。视图是通过查询表达式定义的“虚关系”。以关系代数为例说明了视图如何定义和使用。通过用定义视图的表达式替换视图来对涉及视图的查询求值。视图是简化数据库查询的有用机制，但是通过视图对数据库进行修改却可能产生不利的后果。因此，数据库系统严格限制通过视图进行更新。出于查询处理效率的考虑，视图可以被实体化——即物理上存储视图。影响到实体化视图的更新会带来额外开销。

习题

3.1　给大学注册办公室设计一个关系数据库，此机构保存各门课的数据，包括讲课教师，选课学生数，上课时间地点。对于每个学生-课程对，还需要记录一个成绩。

3.2　解释术语关系和关系模式在意义上的区别。参照你对习题 3.1 的解，解释你的回答。

3.3　为图 3-37 所示的 E-R 图设计一个关系数据库。

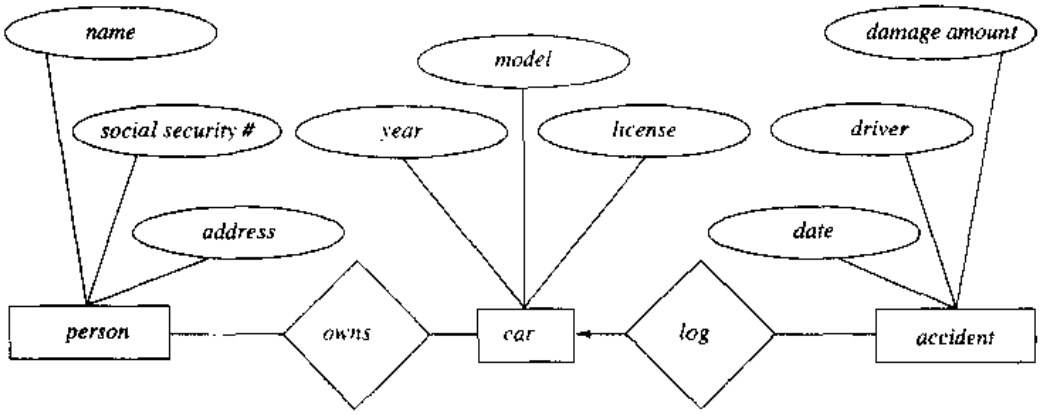


图 3-37　E-R 图

3.4　第 2 章说明了如何表示多对多，多对一，一对多和一对一的联系集。请说明主码怎样帮助我们在关系模型中表示这样的联系集。

3.5　考虑图 3-38 所示关系数据库。对于下述查询中的每一个，给出一个关系代数表达式、一个元组关系演算表达式和一个域关系演算表达式：

(a) 找出 First Bank Corporation 的所有员工姓名。

(b) 找出 First Bank Corporation 所有员工的姓名和居住城市。

(c) 找出 First Bank Corporation 所有年收入在 $10 000 以上的员工姓名和居住的街道、城市。

(d) 找出所有居住地与工作的公司在同一城市的员工姓名。

(e) 找出与其经理居住在同一城市同一街道的所有员工姓名。

(f) 找出此数据库中不在 First Bank Corporation 工作的所有员工姓名。

(g) 找出比 Small Bank Corporation 的所有员工收入都高的所有员工姓名。

(h) 假设公司可以位于几个城市中。找出 Small Bank Corporation 所在的每一个城市中的所有公司。

employee (*employee-name*, *street*, *city*)
works (*employee-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*employee-name*, *manager-name*)

图 3-38 习题 3.5 和 3.12 的关系数据库

3.6 给定如下关系模式：

$$R = (A, B, C)$$
$$S = (D, E, F)$$

设关系 r (R) 和 s (S) 已知。分别给出与下列表达式等价的元组关系演算表达式：

(a) $\Pi_A (r)$

(b) $\sigma_{B=17} (r)$

(c) $r \times s$

(d) $\Pi_{A,F} (\sigma_{C=D} (r \times s))$

3.7 设 $R = (A, B, C)$，r_1 和 r_2 都是模式 R 上的关系。分别给出与下列表达式等价的域关系演算表达式：

(a) $\Pi_A (r_1)$

(b) $\sigma_{B=17} (r_1)$

(c) $r_1 \cup r_2$

(d) $r_1 \cap r_2$

(e) $r_1 - r_2$

(f) $\Pi_{A,B} (r_1) \bowtie \Pi_{B,C} (r_2)$

3.8 设 $R = (A, B)$ 且 $S = (A, C)$，r (R) 和 s (S) 是关系。分别给出与下列域关系演算表达式等价的关系代数表达式：

(a) $\{<a> \mid \exists b\ (<a, b> \in r \wedge b = 17)\}$

(b) $\{<a, b, c> \mid <a, b> \in r \wedge <a, c> \in s\}$

(c) $\{<a> \mid \exists b\ (<a, b> \in r) \vee \forall c\ (\exists d\ (<d, c> \exists s) \Rightarrow <a, c> \exists s)\}$

(d) $\{<a> \mid \exists c\ (<a, c> \in s \wedge \exists b_1, b_2\ (<a, b_1> \in r \wedge <c, b_2> \in r \wedge b_1 > b_2))\}$

3.9 设 $R = (A, B)$ 且 $S = (A, C)$，r (R) 和 s (S) 是关系。试用特殊常量 *null*，分别写出等价于下列表达式的元组关系演算表达式：

(a) $r \bowtie s$

(b) $r \overrightarrow{\bowtie} s$

(c) $r \overrightarrow{\bowtie} s$

3.10 图 3-19 中的关系是查询“找出在银行中有贷款的所有客户姓名”的结果。重写查询，使结果中不仅包含姓名，还包含每个客户的居住城市。请注意现在客户 Jackson 不再出现在结果中，尽管 Jackson 实际上从银行贷了款。

(a) 说明为什么 Jackson 不出现在结果中。

(b) 假设你希望 Jackson 出现在结果中，你将如何修改数据库来达到这样的效果。

(c) 仍假设你希望 Jackson 出现在结果中，用外连接来书写查询，使得你可以在不修改数

据库的前提下也能实现所需。

3.11 外连接是自然连接的一种扩展，它使得参与运算的关系中的元组在连接结果中不丢失。描述 theta 连接运算可以怎样扩展，使得在 theta 连接结果中来自左侧、右侧及两侧关系的元组不被丢失。

3.12 考虑图 3-38 的关系数据库。对于下列每个要求，给出一个关系代数表达式：

(a) 修改数据库，使 Jones 现在居住在 Newton。

(b) 为 First Bank Corporation 的所有员工都提工资 10%。

(c) 为数据库中所有经理都提工资 10%。

(d) 为数据库中所有工资不高于 $100 000 的经理提工资 10%，而高于 $100 000 的经理提工资 3%。

(e) 删除 *works* 关系中属于 Small Bank Corporation 的员工的所有元组

3.13 列出在数据库中引入空值的两个原因。

3.14 某些系统允许带标注的空值。带标注的空值 $\perp_i$ 与自身相等，但 $i \neq j$ 时，$\perp_i \neq \perp_j$。带标注的空值的一个应用是允许通过视图进行某些更新。考虑视图 *loan-info*（3.7 节），指出如何用带标注的空值来通过视图 *loan-info* 插入元组（"Johnson", 1900）。

3.15 用银行的例子，以如下方式书写关系代数查询以找出两个以上客户共有的帐户：

(a) 用聚集函数。

(b) 不用任何聚集函数。

3.16 考虑图 3-38 的关系数据库，分别给出下列查询的关系代数表达式：

(a) 找出员工最多的公司。

(b) 找出工资最少的员工所在的公司。

(c) 找出人均工资比 First Bank Corporation 人均工资高的公司。

3.17 列举两个定义视图的原因。

3.18 列出用视图表达的更新操作的两个主要问题。

文献注解

关系模型是 IBM 研究实验室的 E. F. Codd 于 60 年代末在［Codd 1970］里提出的。这一工作使 Codd 在 1981 年获得了声望很高的 ACM 图灵奖。在 Codd 最初的论文之后，几个研究项目开始进行，它们的目标是构造实际的关系数据库系统，其中包括 IBM San Jose 研究实验室的 System R、加州大学伯克利分校的 Ingres、IBM T. J. Watson 研究实验中心的 Query-by-Example，以及位于英国 Peterlee IBM 科学中心的 Peterlee Relational Test Vehicle（PRTV）。System R 在 Astrahan 等［1976，1979］和 Chamberlin 等［1981］中进行了讨论。Ingres 在 Stonebraker［1980，1986b］和 Stonebraker 等［1976］中进行了讨论。Query-by-Example 在 Zloof［1977］中作了描述。PRTV 在 Todd［1976］中作了描述。

大量关系数据库产品现在可以从市场上购得。其中包括 IBM 的 DB2、Ingres、Oracle、Sybase、Informix、Microsoft SQL Server。个人计算机上的数据库产品包括 Microsoft Access、dBase 和 FoxPro。关于这些产品的信息可以在它们各自的手册中找到。专门介绍特定商用系统的教科书包括 Malamud［1989］和 Date［1987］，里面包括有 Ingres 的内容 Martin 等［1989］以及 Date 和 White［1993］，介绍的是 IBM 的 DB2 产品；Date 和 White［1989］介绍的是 SQL/DS；Corrigan 和 Gurry［1993］以及 Koch 和 Loney［1995］介绍的是 Oracle。

大多数数据库文献都对关系数据模型进行了一般性讨论。Gardarin 和 Valduriez［1989］、

Valduriez 和 Gardarin [1989]、Atzeni 和 De Antonellis [1993]，以及 Maier [1983] 是专门讨论关系数据模型的文献。

关系代数最初的定义在 Codd [1970] 中给出；元组关系演算最初的定义在 Codd [1972b] 中给出。元组关系演算和关系代数等价性的形式化定义在 Codd [1972b] 中给出。关系演算的一些扩展后来也被提出来。Klug [1982] 和 Escobar-Molano 等 [1993] 描述了通过引入标量聚集函数进行的扩展。关系模型的扩展，关系代数中空值的论述（RM/T 模型），以及外连接，这些内容都出现在 Codd [1979] 中。Codd [1990] 是 E. F. Codd 关于关系模型的所有文章的一个概括。外连接在 Date [1983b] 中也进行了讨论。

谈到通过视图更新关系数据库的问题的文献中包括 Barsalou 等 [1991]、Bancilhon 和 Spyratos [1981]、Cosmadakis 和 Papadimitriou [1984]、Dayal 和 Bernstein [1978, 1982]、Keller [1985]、以及 Langerak [1990]。

Blakeley 等 [1986, 1989] 以及 Griffin 和 Libkin [1995] 描述了实体化视图的维护技术。Gupta 和 Mumick [1995] 提供了关于实体化视图维护近来工作的一个综述。

第4章 SQL

在第3章中描述的形式语言提供了用来表示查询的一个简洁记法。然而，商品化的数据库系统需要对用户更加友好的查询语言。本章将研究最具影响的语言：SQL。SQL使用了关系代数和关系演算结构的组合。

尽管说SQL是一个“查询语言”，但它除了查询数据库以外还有许多别的功能。SQL具有定义数据结构、修改数据库中的数据、以及说明安全性约束条件等特性。

我们的目的并不是想提供一个完整的SQL用户手册，而是要介绍SQL的基本结构和概念。实现SQL的各种方法可能在一些细节上有所不同，或只是支持整个语言的一个子集。

第5章介绍另外两种有影响力的商业语言——QBE和Quel，以及一种在系统研究中使用的语言——Datalog。这些语言代表着不同的风格。QBE基于域关系演算，Quel基于元组关系演算，Datalog基于逻辑编程语言Prolog。这三种语言在研究数据库系统方面都有其影响，前两种在市场化的商业系统中也是如此。

4.1 背景

SQL已经确立起自己作为标准关系数据库语言的地位。SQL有许多种版本，最早的版本是由IBM的San Jose研究室（现在的Almaden研究中心）提出的。该语言最初叫做Sequel，作为System R项目的一部分于70年代初付诸实施，发展到现在，它的名字已变为SQL（结构化的查询语言）。现在有许多产品支持SQL语言。

1986年，美国国家标准协会（ANSI）和国际标准化组织（ISO）发布了SQL标准SQL-86。1987年IBM发布了自己的SQL企业标准——系统应用程序结构数据库界面（SAA-SQL）。扩充的标准SQL-89于1989年发表，现在的数据库系统通常至少要支持SQL-89。ANSI/ISO的最新SQL标准是SQL-92。文献注解提供了关于这些标准和正在进行的SQL-3标准化工作的参考文献。

本节将在SQL-89和SQL-92的基础之上对SQL进行一个总体介绍。应该注意的是，有的SQL实现仅支持SQL-89，而不支持SQL-92。为有益于那些使用的数据库系统不完全支持SQL-92标准的用户，那些不被SQL-89支持的SQL-92特性将会明确指出。

SQL语言有以下几个部分：

- 数据定义语言（DDL）。SQL DDL提供定义关系模式、删除关系、建立索引以及修改关系模式的命令。
- 数据操纵语言（DML）。SQL DML不仅包括基于关系代数和元组关系演算的查询语言，还包括在数据库中插入、删除、修改元组的命令。
- 嵌入式DML语言。嵌入式SQL用于某种通用编程语言中，如PL/1、Cobol、Pascal、Fortran和C。
- 视图定义。SQL DDL包括定义视图的命令。
- 权限管理。SQL DDL中包括指定对关系和视图的访问权的命令。
- 完整性。SQL DDL包括定义数据库中的数据必须满足的完整性约束条件的命令。破坏

完整性约束条件的更新将被禁止。

• 事务控制。SQL 提供定义事务开始和结束的命令。有些实现还允许显式地封锁数据以实现并发控制。

本章的内容将覆盖 SQL 的交互式 DML 语言和 SQL 基本的 DDL 特性。

本章和第 5 章将使用银行的例子，其关系模式定义如下：

Branch-schema = (*branch-name*, *branch-city*, *assets*)
Customer-schema = (*customer-name*, *customer-street*, *customer-city*)
Loan-schema = (*branch-name*, *loan-number*, *amount*)
Borrower-schema = (*customer-name*, *loan-number*)
Account-schema = (*branch-name*, *account-number*, *balance*)
Depositor-schema = (*customer-name*, *account-number*)

在第 6 章里，我们讨论与完整性有关的 SQL 特性。13、14 章将讨论事务，19 章将讨论安全性和权限管理。为了便于阅读，在本章和其他章节的模式名、关系名和属性名中使用连接符，但实际的 SQL 系统中注意应使用下划线 "_"。

4.2 基本结构

关系数据库是关系的集合，每个关系有一个唯一的名字。每个关系具有的结构类似第 3 章中所给出的那样。SQL 允许使用 null（空值）表示该值未知或不存在。为使用户能定义哪些属性不允许赋空值，SQL 做出了相应规定，这将在 4.11 节讨论。

SQL 表达式基本结构包括三个子句：select 子句，from 子句和 where 子句。

• select 子句对应关系代数中的投影运算，用来列出查询结果中的属性。

• from 子句对应关系代数中的笛卡儿积，它列出表达式求值中需扫描的关系。

• where 子句对应关系代数中的选择谓词。它由谓词构成，这些谓词涉及 from 子句中出现的那些关系的属性。

术语 select 在 SQL 和关系代数中有不同的含义，这是一个不幸的历史事实。在这里强调含义的不同是为了减少可能造成的混淆。

一个典型的 SQL 查询具有如下形式：

select A_1, A_2, $\cdots$, A_n
from r_1, r_2, $\cdots$, r_m
where P

每个 A_i 表示一个属性，每个 r_i 表示一个关系，P 是个谓词。此查询等价于关系代数表达式

$$\Pi_{A_1,A_2,\cdots,A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

如果省略 where 子句，则谓词 P 为 true。但与关系代数表达式不同，在 SQL 的查询结果中可以包含很多重复元组，我们将在 4.2.8 节中来讨论这个问题。

SQL 先构造 from 子句中关系的笛卡儿积，根据 where 子句中的谓词进行关系代数的选择运算，然后将结果投影到 select 子句中的属性上。实践中，SQL 也许会转换表达式至等价但效率更高的形式，我们将在第 12 章探讨效率问题。

4.2.1 Select 子句

SQL 查询语句的结果当然是一个关系。用银行的例子来看一个简单的查询“找出关系 *loan* 中所有分支机构的名字”：

select *branch-name*
from *loan*

结果是一个只有属性 *branch-name* 的关系。

形式化的查询语言基于关系是一个集合这样的数学概念，因此，重复的元组不会出现在关系中。实践中，重复元组的删除是相当费时的，所以 SQL 像大多数其他商业查询语言一样，允许在关系和 SQL 表达式结果中出现重复。因此，在上述查询中，*branch-name* 在关系 *loan* 的元组中每出现一次，都会在查询结果中列出一次。

如果想要强迫删除重复，可在 select 后加入关键词 distinct。如果想要删除重复元组，可以重写上述查询如下：

select distinct *branch-name*
from *loan*

SQL 允许使用关键词 all 来显式指明不删除重复：

select all *branch-name*
from *loan*

既然保留重复元组是缺省的，我们也不使用 all。为了保证查询结果中删除重复元组，我们将在任何有必要的地方使用 distinct。在大多数没有使用 distinct 的查询中，查询结果中每个元组重复出现的精确次数并不重要，但在某些应用中这个数目是很重要的，我们将在 4.2.8 节讨论这个问题。

星号“*”可以用来表示“所有的属性”，因而，前例在 select 子句中使用 *loan*.* 可指代 *loan* 中的所有属性。形式为 select * 的 select 子句表示出现在 from 子句中的所有关系的所有属性都应出现在结果中。

select 子句还可包含带 +、−、*、/的算术表达式，运算对象可以是常数或元组的属性。例如，查询

select *branch-name*, *loan-number*, *amount* * 100
from *loan*

将返回一个与 *loan* 一样的关系，只是属性 *amount* 的值是原来的 100 倍。

SQL-92 还提供了一些特殊数据类型，如各种形式的日期类型，并允许一些作用于这些类型的算术函数。

4.2.2 Where 子句

让我们举例说明 SQL 中 where 子句的用法。考虑查询“找出所有在 Perryridge 分支机构贷

款且贷款额超过 $1200 的贷款的贷款号”，该查询用 SQL 可书写如下：

select *loan-number*
from *loan*
where *branch-name* = “Perryridge” **and** *amount* > 1200

SQL 在 where 子句中使用逻辑运算符 and、or 和 not，而没有使用符号 ∧、∨、¬。逻辑运算符的运算对象可以是包含比较运算符<、<=、>、>=、=和<>的表达式。SQL 允许使用比较运算符比较字符串和算术表达式以及特殊类型，如日期类型。

为了简化 where 子句，SQL 还提供 between 比较运算符，说明一个值小于或等于某个值同时大于或等于另一个值。如果想找出贷款额在 90 000～100 000 美元之间的贷款的贷款号，可以使用 between 比较，书写如下：

select *loan-number*
from *loan*
where *amount* **between** 90000 **and** 100000

它可以取代

select *loan-number*
from *loan*
where *amount* <= 100000 **and** *amount* >= 90000

同样，还可以使用 not between 比较运算符。

4.2.3 from 子句

最后，来讨论 from 子句的用法。from 子句本身定义了子句中关系的笛卡儿积。由于自然连接是用笛卡儿积、选择和投影来定义的，为自然连接写 SQL 表达式还是相对简单的。

为查询“找出从银行贷款的所有用户的名字和贷款号”写出的关系代数表达式为：

$$\Pi_{customer\text{-}name,\ loan\text{-}number}\ (borrower \bowtie loan)$$

该查询用 SQL 可书写如下：

select distinct *customer-name*, *borrower*.*loan-number*
from *borrower*, *loan*
where *borrower*.*loan-number* = *loan*.*loan-number*

注意，SQL 使用 *relation-name*.*attribute-name* 的写法和关系代数的写法一样，这是为了避免一个属性名出现在多个关系中所引起的混乱。本可以在 select 子句中使用 *borrower*.*customer-name* 代替 *customer-name*，但既然 from 子句中出现的关系中只有一个含有属性 *customer-name*，因此书写 *customer-name* 并不会造成混淆。

让我们对前面的查询进行扩展，考虑一个更加复杂的例子，这个例子是要求客户在 Perryridge 银行有贷款：“找出在 Perryridge 银行中有贷款的客户姓名及贷款的贷款号”。书写该查

询需在 where 子句中列出两个约束条件，并用逻辑运算符 and 将它们连接起来：

select distinct *customer-name*, *borrower*.*loan-number*
from *borrower*, *loan*
where *borrower*.*loan-number* = *loan*.*loan-number* **and**
　　branch-name = "Perryridge"

SQL-92 的扩展中包括在 from 子句中使用自然连接和外连接，我们将在 4.10 节讨论这些扩展。

4.2.4 更名运算

SQL 提供可为关系和属性重新命名的机制，这是通过使用具有如下形式的 as 子句来进行的：

old-name **as** *new-name*

as 子句既可出现在 select 子句中，也可出现在 from 子句中。

考虑刚刚使用过的查询语句：

select distinct *customer-name*, *borrower*.*loan-number*
from *borrower*, *loan*
where *borrower*.*loan-number* = *loan*.*loan-number* **and**
　　branch-name = "Perryridge"

该查询的结果是含有以下两个属性的关系：

customer-name, *loan-number*

结果中的属性名是从 from 子句中关系的属性名得来。

但是不能总是用这个方法派生名字。首先，在 from 子句中的两个关系可能含有相同名字的属性，在这种情况下，结果中就会有一个属性名重复出现；其次，如果在 select 子句中使用了算术表达式，结果属性就没有名字了；第三，就算像上例中那样属性名可以从基本关系中得到，我们可能还想要更改结果中的属性名。因此，SQL 提供了更改结果关系中属性名的方法。

例如，如果想要把属性名 *loan-number* 用 *loan-id* 代替，重写上面查询为：

select distinct *customer-name*, *borrower*.*loan-number* **as** *loan-id*
from *borrower*, *loan*
where *borrower*.*loan-number* = *loan*.*loan-number* **and**
　　branch-name = "Perryridge"

4.2.5 元组变量

正如在元组关系演算中那样，as 子句在定义元组变量时特别有用。SQL 中的元组变量必须和特定的关系相联系。元组变量是通过在 from 子句中使用 as 子句来定义的。为了说明这一用

法，再次重写查询“找出在银行中贷款的所有客户的名字和贷款号”：

select distinct *customer-name*，*T*.*loan-number*
from *borrower* **as** *T*，*loan* **as** *S*
where *T*.*loan-number* = *S*.*loan-number*

注意在 from 子句中定义元组变量是通过将其放在与之相联系的关系名之后，而关键词 as 放在中间（as 是可选的）。事实上，当用形式 *relation-name*.*attribute-name* 书写表达式时，关系名即是隐含定义的元组变量。

元组变量在比较同一关系中的两个元组时非常有用。要记住在这种情况下，可以使用关系代数中的更名运算。假设需查询“找出资产至少比位于 Brooklyn 的某一家分支机构高的分支机构名”，写成 SQL 表达式如下：

select distinct *T*.*branch-name*
from *branch* **as** *T*，*branch* **as** *S*
where *T*.*assets* > *S*.*assets* **and** *S*.*branch-city* = “Brooklyn”

注意不能使用 *branch*.*asset* 这样的写法，因为我们并不清楚到底对 *branch* 的哪个引用是所需要的。

SQL-92 允许用记号（v_1，v_2，…，v_n）表示一个有 n 个分量的元组，各分量值分别为 v_1，v_2，…，v_n。在元组上可以按字典顺序进行比较运算。例如，（a_1，a_2）<=（b_1，b_2）在 $a_1 < b_1$ 或（$a_1 = b_1$）∧（$a_2 <= b_2$）时为真。同样，这两个元组相等当且仅当其各对应分量全都相等。

4.2.6 字符串操作

对字符串进行的最通常的操作是使用操作符 like 的模式匹配。使用两个特殊的字符来描述模式：

- 百分号（%）。匹配任意子串。
- 下划线（_）。匹配任意一个字符。

模式是大小写敏感的，也就是说，大写字符与小写字符不匹配，反之亦然。为了说明模式匹配，考虑下列例子：

- “Perry%”匹配任何以“Perry”开头的字符串。
- “%idge%”匹配任何包含 idge 为子串的字符串，例如“Perryridge”、“Rock Ridge”、“Mianus Bridge”和“Ridgeway”。
- “- - -”匹配只含三个字符的字符串。
- “- - - %”匹配至少含三个字符的字符串。

模式在 SQL 中用比较运算符 like 表达。考虑查询“找出街道地址中包含子串‘Main’的所有客户名”，该查询可书写如下：

select *customer-name*
from *customer*
where *customer-street* **like** “%Main%”

为使模式中能够包含特殊模式字符（即%和_），SQL 允许定义转义字符。转义字符紧靠特殊字符并放在它的前面，表示该特殊字符将被当成普通字符。在 like 比较中使用 escape 关键词来定义转义符。为了说明这一用法，考虑以下模式，它使用反斜线（\）作为转义符：

- like "ab\%cd%" escape "\" 匹配所有以 "ab%cd" 开头的字符串。
- like "ab\\cd%" escape "\" 匹配所有以 "ab\cd" 开头的字符串。

SQL 允许使用 not like 比较运算符搜寻不匹配项。

SQL 还允许在字符串上有多种函数，例如连接（使用 "‖"）、提取子串、计算字符串长度、大小写转换，等等。

4.2.7　排列元组的显示次序

SQL 为用户提供了对关系中元组显示次序的控制。order by 子句就是让查询结果中的元组按排列的顺序显示。为了按字母顺序列出在 Perryridge 分支机构中有贷款的客户，可写成：

```
select distinct customer-name
from borrower, loan
where borrower.loan-number = loan.loan-number and
      branch-name = "Perryridge"
order by customer-name
```

order by 子句缺省使用升序。要说明排序顺序，可以用 desc 表示降序，用 asc 表示升序。此外，排序可在多个属性上进行，假设希望关系 *loan* 按 *amount* 降序排列，但如果有相同贷款额，我们再将它们按贷款号升序排列。在 SQL 中表达该查询如下：

```
select *
from loan
order by amount desc, loan-number asc
```

要完成一个 order by 请求，SQL 必须执行排序。由于执行大数量元组的排序操作代价是很大的，不到万不得已不要排序。

4.2.8　重复

在有些情况下，包含重复元组的关系是有用的。SQL 不仅明确定义了查询结果中有哪些元组，而且还定义了结果中的每个元组各有几个复本。可以用关系运算符的多重集版来定义 SQL 查询的复本语义，在此定义几个关系代数运算符的多重集版本。已知多重集关系 r_1 和 r_2：

1）如果在 r_1 中有元组 t_1 的 c_1 个复本，而且 t_1 满足选择 σ_θ，那么有 c_1 个 t_1 的复本在 $\sigma_\theta(r_1)$ 中。

2）对于 r_1 中 t_1 的每个复本，在 $\Pi_A(r_1)$ 中都有一个 $\Pi_A(t_1)$ 的复本与其对应，其中 $\Pi_A(t_1)$ 表示单个元组 t_1 的投影。

3）如果有 c_1 个 t_1 的复本在 r_1 中且有 c_2 个 t_2 的复本在 r_2 中，那么有 $c_1 * c_2$ 个 $t_1 \cdot t_2$ 元组的复本在 $r_1 \times r_2$ 中。

例如，假设模式为 (A, B) 的关系 r_1 和模式为 (C) 的关系 r_2 是如下多重集：

$$r_1 = \{(1, a), (2, a)\} \qquad r_2 = \{(2), (3), (3)\}$$

那么，$\Pi_B\ (r_1)$ 将是$\{(a),\ (a)\}$，而 $\Pi_B\ (r_1) \times r_2$ 将是

$$\{(a,\ 2),\ (a,\ 2),\ (a,\ 3),\ (a,\ 3),\ (a,\ 3),\ (a,\ 3)\}$$

现在可以定义 SQL 查询结果中的各个元组有多少复本。形如：

select A_1, A_2, $\cdots$, A_n
from r_1, r_2, $\cdots$, r_m
where P

的 SQL 查询等价于关系代数表达式

$$\Pi_{A_1, A_2, \cdots, A_n}\ (\sigma_P\ (r_1 \times r_2 \times \cdots \times r_m))$$

其中关系运算符 σ、Π 和 $\times$ 使用其多重集版本。

4.3 集合操作

SQL-92 在关系上的 union、intersect 和 except 操作对应于关系代数中的运算 $\cup$、$\cap$ 和 $-$。如同关系代数中的并、交、差一样，SQL 中参加操作的关系必须是相容的，也就是说，它们必须含有相同的属性集。SQL-89 对 union、intersect 和 except 的用法有一些限制，甚至有的标准并不支持这些操作。

现在举例说明怎样将第 3 章中考虑的一些查询例子写成 SQL 语句。我们将构造在两个集合间使用 union、intersect 和 except 操作的查询，这两个集合分别是在银行有帐户的客户集合和在银行有贷款的客户集合。在银行有帐户的客户集合能够从以下得到

select *customer-name*
from *depositor*

在银行有贷款的客户集合能够从以下得到

select *customer-name*
from *borrower*

我们将用 d 和 b 分别指代包含以上查询结果的两个关系。

4.3.1 并操作

为了找出在银行有帐户、贷款或两者兼具的所有客户，可写出：

(**select** *customer-name*
from *depositor*)
union
(**select** *customer-name*
from *borrower*)

与 select 子句不同，union 操作自动去除重复。因此在前面查询中，如果一个叫 Jones 的客户在银行中有几个帐户或贷款（或两者都有），Jones 在结果中也只出现一次。

如果想保留所有重复，必须用 union all 代替 union：

```
(select customer-name
 from depositor)
union all
(select customer-name
 from borrower)
```

在结果中出现的重复元组数等于在 d 和 b 中出现的重复元组数的和。因此，如果 Jones 在银行中有三个帐户和两笔贷款，那么结果中将有五个元组含有 Jones 这个名字。

4.3.2 交操作

为了找出在银行同时有帐户和贷款的客户，可写出：

```
(select distinct customer-name
 from depositor)
intersect
(select distinct customer-name
 from borrower)
```

intersect 操作自动去除重复。因此在前面查询中，如果一个叫 Jones 的客户在银行中有几个帐户和贷款，Jones 在结果中也只出现一次。

如果想保留所有重复，必须用 intersect all 代替 intersect：

```
(select customer-name
 from depositor)
intersect all
(select customer-name
 from borrower)
```

在结果中出现的重复元组数等于在 d 和 b 中出现的重复元组数较少的那个。因此，如果 Jones 在银行中有三个帐户和两笔贷款，那么结果中将有两个元组含有 Jones 这个名字。

4.3.3 差操作

为了找出在银行中有帐户但无贷款的客户，可写出：

```
(select distinct customer-name
 from depositor)
except
(select customer-name
 from borrower)
```

except 操作自动去除重复。因此在前面查询中，当 Jones 在银行中有帐户但无贷款时，一个含有客户名 Jones 的元组在结果中出现且只出现一次。

如果想保留所有重复，必须用 except all 代替 except：

```
(select customer-name
 from depositor)
except all
(select customer-name
 from borrower)
```

在结果中出现的重复元组数，只要差是正的，就等于在 d 中出现的重复元组数减去 b 中出现的重复元组数。因此，如果 Jones 在银行中有三个帐户和一笔贷款，那么结果中将有两个元组含有 Jones 这个名字。但若客户有两个帐户和三笔贷款，那么结果中将没有含有 Jones 这个名字的元组。

4.4 聚集函数

聚集函数是以一个值集合（非多重集或多重集）为输入，返回单个值的函数。SQL 提供了五个预定义聚集函数：

- 平均值：avg。
- 最小值：min。
- 最大值：max。
- 总和：sum。
- 计数：count。

sum 和 avg 的输入必须是数字，而其他函数还可作用在非数字数据类型如字符串上。

作为示例，考虑查询“找出 Perryridge 分支机构帐户余额的平均值”。书写该查询如下：

```
select avg (balance)
from account
where branch-name = "Perryridge"
```

该查询的结果是只有一个属性的关系，其中只包含一行，这一行的数值对应 Perryridge 分支机构的平均余额。可以有选择地给结果关系中的属性用 as 子句赋个名字。

有时候不仅希望将聚集函数作用在单个元组集上，而且也希望将其作用在一组元组集上。在 SQL 中可用 group by 子句实现这个愿望，在 group by 子句中的一个或多个属性是用来构造分组的。group by 子句中所有属性有相同值的元组将放在一个组中。

作为示例，考虑查询“找出每个分支机构的帐户结算平均额”，该查询书写如下：

```
select branch-name, avg (balance)
from account
group by branch-name
```

在计算平均值时保留重复元组是很重要的。假设在（小）Brighton 分支机构中的帐户余额分别为 $1000、$3000、$2000 和 $1000，平均额为 $7000/4 = $1750.00，如果重复被删除了，将得到错误答案（$6000/3 − $2000）。

有些情况下在计算聚集函数前需先删掉重复元组。如果确实想删除重复元组，可在聚集表达式中使用关键词 distinct。比如这样一个例子“找出每个分支机构储户数”，在该例中不论一

个客户有几个帐户，作为一名储户只计算一次，书写该查询如下：

select *branch-name*，**count**（**distinct** *customer-name*）
from *depositor*，*account*
where *depositor*.*account-number* = *account*.*account-number*
group by *branch-name*

有时候，对分组限定条件比对元组限定有用。例如，我们也许只对帐户平均结算大于$1200的分支机构感兴趣。该条件并不是针对单个元组，而是针对group by子句形成的分组。为表达这样的查询，使用SQL的having子句。having子句中的谓词在形成分组后才起作用，因此可以使用聚集函数。用SQL表达该查询如下：

select *branch-name*，**avg**（*balance*）
from *account*
group by *branch-name*
having avg（*balance*）＞1200

有时，希望将整个关系看成一个分组，这时，不必使用group by子句。考虑查询“找出所有帐户的存款平均额”，该查询可书写如下：

select avg（*balance*）
from *account*

我们经常使用聚集函数count计算一个关系中元组的个数。SQL中该函数的写法是count（*）。因此，要找出关系*customer*中的元组数，可写成：

select count（*）
from *customer*

SQL不允许在用count（*）时使用distinct。在用max和min时使用distinct是合法的，尽管结果并无差别。可使用关键词all代替distinct来说明保留重复元组，但是，既然all是缺省的，就没必要这么做了。

如果在同一个查询中同时存在where子句和having子句，那么首先应该用where子句中的谓词。满足where谓词的元组通过group by子句形成分组。having子句若存在，就将作用于每一分组。不符合having子句谓词的分组将被抛弃，剩余的组被select子句用来产生查询结果元组。

为了说明having子句和where子句在同一个查询中的用法，考虑查询“找出住在Harrison且在银行中至少有二个帐户的客户的平均余额”：

select *depositor*.*customer-name*，**avg**（*balance*）
from *depositor*，*account*，*customer*
where *depositor*.*account-number* = *account*.*account-number* **and**
　　depositor.*customer-name* = *customer*.*customer-name* **and**
　　customer-city = “Harrison”

group by *depositor . customer-name*
having count (**distinct** *depositor . account-number*) > = 3

4.5 空值

先前提到过，SQL 允许使用 *null* 值表示某属性值信息缺失。

我们在谓词中使用特殊的关键词 *null* 测试空值。因而为找出关系 *loan* 中 *amount* 为空值的贷款号，可写成：

select *loan-number*
from *loan*
where *amount* **is null**

谓词 is not null 用来检测非空值。

null 的使用给算术运算和比较运算带来一些麻烦。如果算术运算的输入有一个是空，则该算术表达式（如执行 + 、 - 、 * 或 /）的结果是空。如果比较运算有空值作为比较对象，则可以把结果视为 false。更准确些，SQL-92 将这种比较运算结果看成 unknown，即既不是 true，也不是 false。SQL-92 允许测试比较结果是否是不知道，而不是去测试是否是真或假。但在其他几乎所有情况下，unknown 只被当作 false。文献注解引用的参考书提供了详细的内容。

空值的存在也给聚集运算符处理带来了麻烦。假设关系 *loan* 中有一些元组的 *amount* 是空值，考虑以下计算所有贷款额总和的查询：

select sum (*amount*)
from *loan*

由于一些元组的 *amount* 值是空值，上述查询待求和的值中包含了空值。SQL 标准并不认为总和也为 *null*，而是认为 sum 运算符应忽略输入的空值。

总而言之，聚集函数对待空值有以下原则：除了 count (*) 外所有聚集函数都忽略输入中的空值，空值被忽略有可能造成无输入参加函数运算。规定 count 在无输入运算情况下值为 0，其他所有聚集函数在无输入情况下返回一个空值。在一些更复杂的 SQL 结构中空值的影响是很微妙的。

4.6 嵌套子查询

SQL 提供嵌套子查询机制。子查询是嵌套在另一个查询中的 Select-from-where 表达式。一般子查询的使用是为了对集合的成员资格、集合的比较以及集合的基数进行测试。我们将在下面的小节中研究这些用途。

4.6.1 集合成员资格

SQL 引入测试元组是否是一个关系中成员的关系演算。连接词 in 测试集合中的成员，该集合是由 select 子句产生的一组值的集合。连接词 not in 则是测试非集合中的成员。作为示例，再考虑查询“找出在银行中同时有帐户和贷款的客户。”先前，写该查询是使用两个集合的交：银行的储户集合和银行的贷款用户集合。可以采用另一种方法，在贷款用户的集合中寻找拥有

帐户的用户。很明显，这种方法得出的结果与前面相同，但它使我们用 SQL 中的 in 连接词写这个查询。从找出所有拥有帐户者开始，写出以下子查询

```
(select customer-name
from depositor)
```

然后从子查询形成的拥有帐户者名单中找出有银行贷款者。完成此项任务可将子查询嵌套在外部 select 中。最后的查询语句是：

```
select distinct customer-name
from borrower
where customer-name in (select customer-name
                        from depositor)
```

该例说明了在 SQL 中书写同一查询可以有多种方法。这种灵活性是有好处的，它允许用户用最接近自然的方法去思考查询。我们将看到在 SQL 中有许多这样的冗余。

前面例子中，我们在单属性关系中测试成员资格。在 SQL-92 中测试任意关系的成员资格也是可能的。因此前面提到过的例子“找出在 Perryridge 分支机构同时有帐户和贷款的客户”还可以这样表达：

```
select distinct customer-name
from borrower, loan
where borrower.loan-number = loan.loan-number and
      branch-name = "Perryridge" and
      (branch-name, customer-name) in
          (select branch-name, customer-name
          from depositor, account
          where depositor.account-number = account.account-number)
```

现在举例说明 not in 结构的用法。为了找出所有在银行中有贷款但无帐户的客户，可写出：

```
select distinct customer-name
from borrower
where customer-name not in (select customer-name
                            from depositor)
```

in 和 not in 操作符也能用于枚举集合。下面的查询找出在银行中有贷款的用户，他的名字既不是“Smith”，也不是“Jones”。

```
select distinct customer-name
from borrower
where customer-name not in ("Smith", "Jones")
```

4.6.2 集合的比较

考虑查询“找出那些总资产至少比位于 Brooklyn 的某一家分支机构要多的分支机构名字”，在 4.2.5 节，我们将此查询写作：

```
select distinct T.branch-name
from branch as T, branch as S
where T.assets > S.assets and S.branch-city = "Brooklyn"
```

SQL 提供另外一种方法重写上面的查询。短语“至少比一个要大”在 SQL 中用 >some 表示。这一结构允许我们重新组织上面的查询，用一种更贴近在英语中描述此查询的形式书写：

```
select branch-name
from branch
where assets > some (select assets
                     from branch
                     where branch-city = "Brooklyn")
```

子查询

```
(select assets
 from branch
 where branch-city = "Brooklyn")
```

得出在 Brooklyn 的各分支机构的资产集合。当元组的资产值至少比 Brooklyn 各分支机构的资产集合中某一成员多时，外层 select 的 where 子句中 >some 的比较为真。

SQL 也允许 <some、<=some、>=some、=some 和 <>some 的比较。作为练习，验证 =some 等价于 in，不过 <>some 并不等价于 not in。SQL 中，关键词 any 同义于 some。早期 SQL 版本中仅允许使用 any，后来的版本为了避免和英语中 any 一词在语言上的混淆，又添加了另一个可选择的关键词 some。

现在让我们稍微修改一下查询，找出那些总资产比 Brooklyn 任意一家分支机构都多的分支机构名。结构 >all 对应词组“比所有…大”，使用该结构，写出查询如下：

```
select branch-name
from branch
where assets > all (select assets
                    from branch
                    where branch-city = "Brooklyn")
```

类似于 some，SQL 也允许 <all、<=all、>=all、=all 和 <>all 的比较。作为练习，证明 <>all 等价于 not in。

作为集合比较的另一个例子，考虑查询“找出平均余额最高的分支机构”。SQL 中聚集函数不能进行复合，因而 max (avg (…)) 是不允许的。我们的策略如下：首先写出一个查询找

出每个银行的平均余额，然后把它作为子查询嵌套在一个大的查询中，以找出那些平均余额大于等于所有平均余额的分支机构。

```
select branch-name
from account
group by branch-name
having avg (balance) >= all (select avg (balance)
                             from account
                             group by branch-name)
```

4.6.3 测试是否为空关系

SQL 还有一个特性可测试一个子查询的结果是否有元组。exists 结构在作为参数的子查询非空时返回 true。使用 exists 结构，可用另外一种方法表示“找出在银行既有帐户又有贷款的客户”。

```
select customer-name
from borrower
where exists (select *
              from depositor
              where depositor.customer-name = borrower.customer-name)
```

测试子查询结果元组集为空可用 not exists。使用 not exists 还可以模拟集合包含（即超集）操作：可将“关系 A 包含 B”写成“not exists（B except A）”。尽管 contains 运算符并不是 SQL-92 标准的一部分，一些早期关系系统中却曾提供了这一运算符。为了说明 not exists 操作符，让我们再次考虑“找出在 Brooklyn 所有分支机构都有帐户的客户”。对每一位客户，要判断他拥有帐户的银行集合中是否包含有 Brooklyn 的银行。使用 except，可以书写此查询如下：

```
select distinct S.cusotmer-name
from depositor as S
where not exists ((select branch-name
                   from branch
                   where branch-city = "Brooklyn")
                  except
                   (select R.branch-name
                    from depositor as T, account as R
                    where T.account-number = R.account-number and
                          S.customer-name = T.customer-name))
```

这里，子查询

```
(select branch-name
 from branch
```

where *branch-city* = "Brooklyn")

找出 Brooklyn 的所有银行，子查询

(**select** R.*branch-name*
from *depositor* **as** T, *account* **as** R
where T.*account-number* = R.*account-number* **and**
S.*customer-name* = T.*customer-name*

找出所有 S.*customer-name* 拥有帐户的银行。这样，外层 select 对每位客户测试其拥有帐户的分支机构集合是否包含 Brooklyn 的所有分支机构集合。

在包含子查询的查询中，元组变量有其作用域规则。在一个子查询中，合法的做法是只使用在该子查询中或包含该子查询的查询中定义的元组变量。如果一个元组变量既在子查询中定义，又在包含该子查询的查询中定义，则子查询中的定义有效。这条原则和编程语言中变量的作用域规则是类似的。

4.6.4 测试是否存在重复元组

SQL 能够测试一个子查询的结果中是否有重复元组。如果作为参数的了查询的结果中没有重复元组，使用 unique 结构测试将返回 true。可以用 unique 结构书写查询语句"找出所有在 Perryridge 分支机构中只有一个帐户的客户"如下：

select T.*customer-name*
from *depositor* **as** T
where unique (**select** R.*customer-name*
from *account*, *depositor* **as** R
where T.*customer-name* = R.*customer-name* **and**
R.*account-number* = *account*.*account-number* **and**
account.*branch-name* = "Perryridge")

可以用 not unique 测试在一个子查询的结果中存在重复的元组。为了说明这一结构，考虑查询"找出所有在 Perryridge 分支机构中有两个以上帐户的客户。"

select distinct T.*customer-name*
from *depositor* T
where not unique (**select** R.*customer-name*
from *account*.*depositor* **as** R
where T.*customer-name* = R.*customer-name* **and**
R.*account-number* = *account*.*account-number* **and**
account.*branch-name* = "Perryridge")

形式化地说，对一个关系的 unique 测试被定义为失败的当且仅当关系中包含两个元组 t_1 和 t_2 且 $t_1 = t_2$。因为在 t_1 或 t_2 的某个域为空时判断 $t_1 = t_2$ 为假，所以尽管一个元组有多个复本，但只要该元组有一个域为空，unique 测试就有可能为真。

4.7 派生关系

SQL-92 允许 from 子句中使用子查询表达式。如果使用这样的表达式，那么必须给表达式产生的关系命名，而其属性也可以重新命名。可以用 as 子句完成重命名，例如，考虑子查询：

(**select** *brance-name*, **avg** (*balance*)
 from *account*
 group by *branch-name*)
as *result* (*branch-name*, *avg-balance*)

该子查询产生的关系包含各分支机构的名字和相应的平均帐户余额。这个临时的关系名为 *result*，属性名为 *branch-name* 和 *avg-balance*。

以查询“找出那些平均帐户结算大于 $1200 的分支机构的平均帐户结算”为例来说明 from 子句中子查询表达式的用法。在 4.4 节我们用 having 子句写过这个查询，现在不用 having 子句来重写这个查询如下：

select *branch-name*, *avg-balance*
from (**select** *branch-name*, **avg** (*balance*)
　　from *account*
　　group by *branch-name*)
　　as *result* (*branch-name*, *avg-balance*)
where *avg-balance* > 1200

注意，我们确实不需要使用 having 子句，这是因为可以在 from 子句中计算临时关系 *result*，而 *result* 的属性在 where 子句中可以直接使用。

4.8 视图

在 SQL 中用 create view 命令可以定义视图。定义视图时需要给视图一个名字，并提供计算视图的查询。create view 的命令格式如下：

create view *v* **as** <query expression>

其中<query expression>可以是任何有效的查询表达式，*v* 表示视图名。注意，关系代数中用于定义视图的写法（见第 3 章）是基于 SQL 写法的。

例如，考虑这样的一个视图，它包含各分支机构的名字以及在分支机构中有帐户或贷款的客户的名字。假设希望这个视图被命名为 *all-customer*，定义这个视图如下：

create view *all-customer* **as**
　(**select** *branch-name*, *customer-name*
　 from *depositor*, *account*
　 where *depositor*.*account-number* = *account*.*account-number*)
　union
　(**select** *branch-name*, *customer-name*

from *borrower*, *loan*
where *borrower*.*loan-number* = *loan*.*loan-number*)

一个视图的属性名可以显式指定如下：

create view *branch-total-loan* (*branch-name*, *total-loan*) **as**
select *branch-name*, **sum** (*amount*)
from *loan*
group by *branch-name*

这个视图给出了各分支机构的贷款总数。因为表达式 sum (*amount*) 没有名字，因而其属性名在视图定义中显式指定。

视图名可以出现在任何关系名可以出现的地方。使用 *all-customer* 视图，可以找出 Perryridge 分支机构的所有客户，语句如下：

select *customer-name*
from *all-customer*
where *branch-name* = "Perryridge"

正如在结构化编程时把任务分解成过程一样，如果把复杂的查询分成更小的视图，然后再组合起来，复杂的查询就可以更容易地理解和写出。然而，和过程定义不一样的是，一条 create view 语句在数据库中产生一个视图的定义，在执行 drop view 语句之前将一直在数据库中存在。制定中的 SQL-3 标准包含一条提议，即可以定义临时的视图，这样的视图不会在数据库中存储。

4.9 数据库的修改

目前为止我们的注意力集中在从数据库中抽取信息上。下面将展示如何用 SQL 语句增加，删除和修改信息。

4.9.1 删除

删除请求的表达与查询非常类似。可以删除整个元组，但不能只删除某些属性上的值。在 SQL 中，一条删除语句格式如下：

delete from r
where p

其中 p 代表一个谓词，r 代表一个关系名。删除时首先从 r 中找出所有使 p (t) 为真的元组 t，然后把它们从 r 中删除。如果省略 where 语句，则 r 中所有元组被删除。

注意到 delete 命令只对一个关系起作用。如果想从多个关系中删除元组，必须为每个关系写一条 delete 命令。where 子句中的谓词可以和 select 语句的 where 子句中的谓词一样复杂。另一个相反的情况是可以有空的 where 子句，语句

delete from *loan*

将把 *loan* 关系中的所有元组删除。(设计良好的系统会在执行这样一条有严重后果的语句之前取得用户的确认。)

下面是 SQL 删除语句的一些例子:

- 删除 Perryridge 分支机构的所有帐户

```
delete from account
where branch-name = "Perryridge"
```

- 删除数额在 $1300 ~ $1500 之间的所有贷款

```
delete from loan
where amount between 1300 and 1500
```

- 删除位于 Needham 的每一个分支机构的所有帐户

```
delete from account
where branch-name in (select branch-name
                      from branch
                      where branch-city = "Needham")
```

上例的 delete 语句首先找出位于 Needham 的所有分支机构，然后将各分支机构的帐户全部删除。

注意，虽然一次只能从一个关系中删除元组，但在 delete 子句中的 where 子句中嵌套 select-from-where 语句，可以访问任意数目的关系。Delete 语句可以包含一个嵌套的 select 语句，指出哪些关系中的元组将被删除。例如，如果想删除余额低于银行平均余额的帐户记录，可以写出如下语句:

```
delete from account
where balance < (select avg (balance)
                 from account)
```

这条 delete 语句首先测试关系 *account* 中的每一个元组，检查其余额是否小于银行平均余额，然后删除所有符合条件的帐户。在执行任何删除之前检查所有元组是至关重要的，因为若有些元组在其余元组未被测试前先被删除，则平均余额将会改变，于是 delete 子句的最后结果将依赖于元组被处理的顺序。

4.9.2 插入

要在关系中插入数据，可以指定被插入的元组，或者是用一条查询语句，该语句的查询结果是希望被插入的元组集合。显然，插入元组的属性值必须在属性域中。同样，插入元组的分量数也必须是正确的。

最简单的插入语句是插入一个元组的语句。假设想要插入的信息是 Perryridge 分支机构中余额为 $1200 的帐户 A-9732，可写成:

```
insert into account
    values ( "Perryridge", "A-9732", 1200)
```

在上面的例子中，元组的值排列的顺序必须和关系模式中列出的属性相对应。考虑到用户可能不记得关系属性的排列顺序，SQL 允许在 insert 语句中指定属性。以下两个 insert 语句实现了与前面语句相同的功能。

```
insert into account (branch-name, account-number, balance)
    values ( "Perryridge", "A-9732", 1200)
insert into account (account-number, branch-name, balance)
    values ( "A-9732", "Perryridge", 1200)
```

更通常的情况是也许想在查询结果的基础上执行插入。假设 Perryridge 分支机构想给每个在该行贷款的客户一个礼物，对应客户的每笔贷款赠送给用户一个 $ 200 的新存款帐户，并以贷款号作为新存款帐户的帐户号，可写作：

```
insert into account
    select branch-name, loan-number, 200
    from loan
    where branch-name = "Perryridge"
```

和本节中前面例子不一样的是，我们没有指定一个元组，而是用 select 子句选取若干元组。这条 select 语句先被执行，求出将要插入关系 *account* 中的元组集合。每个元组包含 *branch-name*（Perryridge）、*loan-number*（作为新帐户的帐户号）和新帐户的初始余额（$ 200）。

若还需要向关系 *depositor* 中添加元组，可以写为：

```
insert into depositor
    select customer-name, loan-number
    from borrower, loan
    where borrower.loan-number = loan.loan-number and
        branch-name = "Perryridge"
```

这一查询将 Perryridge 分支机构中有贷款的元组（*customer-name*，*loan-number*）插入到关系 *depositor* 中，属性 *loan-number* 为贷款号 *loan-number*。

在执行插入之前执行 select 子句是非常重要的。如果在执行 select 子句的同时执行插入动作，像

```
insert into account
    select *
    from account
```

这样的请求就会插入无数的元组。关系 *account* 中的第一个元组将被再次插入关系 *account* 中，产生自身的一份拷贝。由于这个复本已是关系 *account* 中的一部分，select 语句将发现它，于是

第三份拷贝被插入关系 *account* 中。第三份拷贝又被 select 语句发现，于是又有第四份拷贝……形成死循环。在执行插入之前先完成 select 语句的执行可以避免这样的问题。

在讨论 insert 语句时我们只考虑了这样的例子：待插入元组的每个属性都被赋值。正如在第 3 章所见到的，有可能待插入元组只有模式的部分属性被赋值，其余的属性将缺省被赋为 *null*。考虑以下的语句：

insert into *account*
　　values (*null*, "A-401", 1200)

我们知道帐户 A-401 有 $1200，但分支机构的名字是未知的。考虑查询：

select *account-number*
from *account*
where *branch-name* = "Perryridge"

既然拥有帐户 A-401 的分支机构名字未知，我们不能确定它是否等于 "Perryridge"。

我们将在 4.11 节讨论使用 SQL DDL 来禁止插入空值的问题。

4.9.3 更新

有些情况可能希望在不改变整个元组的情况下改变其部分属性的值。为达到这一目的，我们使用 update 语句。与使用 insert、delete 语句类似，待更新的元组可以用查询语句找到。

假设现在银行要计算年息，同时所有存款余额增加 5%，可以写出如下的语句：

update *account*
set *balance* = *balance* * 1.05

上面的语句将在 *account* 的每个元组上执行一次。

现在假设超过 $10 000 的存款利息为 6%，其余的为 5%，需要写两个 update 子句：

update *account*
set *balance* = *balance* * 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* * 1.05
where *balance* < = 10000

注意，和在第 3 章所见到的一样，书写这两条 update 语句的顺序十分重要。假如改变这两条语句的顺序，那么少于 $10 000 的存款将获得 11.3% 的利息。SQL-92 提供 case 结构，可以使用它在一条 update 语句中执行前面两条的更新，详细请见习题 4.11。

一般而言，update 语句中的 where 子句可以是任何 select 语句中合法的 where 子句（包括嵌套的 select 语句）。和 insert、delete 语句类似，update 语句中嵌套的 select 语句可以指向待更新的关系。像以前一样，首先检查关系中的所有元组是否应该被更新，然后才执行更新。例如，要求 "为所有存款大于平均数的帐户增加 5% 的利息" 可以写为：

```
update account
set balance = balance * 1.05
where balance > select avg (balance)
                from account
```

4.9.4 视图的更新

在 SQL 中也存在我们曾在第 3 章讨论过的视图更新异常情况。作为示例，考虑如下的视图定义：

```
create view branch-loan as
    select branch-name, loan-number
    from loan
```

既然 SQL 允许在任何关系名可以出现的地方能够出现视图的名字，因而可以写出：

```
insert into branch-loan
    values ("Perryridge", "L-307")
```

由于视图 *branch-loan* 构造于事实关系 *loan* 基础之上，这条插入语句将把一个元组插入关系 *loan* 中，因此必须为 *amount* 赋值。现在这个值缺省为 null，所以，这条语句执行结果将把如下一个元组：("Perryridge", "L-307", *null*) 插入关系 *loan* 中。

正如在第 3 章所见到的，若一个视图是定义在多个关系基础之上的，这样的视图更新异常将更难以处理。因此，许多 SQL 数据库系统对视图允许的更新作了如下限制：

• 视图更新被允许当且仅当该视图定义在实际关系数据库（即逻辑层数据库）的单个关系上。

在这个限制之下，update、insert 和 delete 操作在前面作为例子的视图 *all-customer* 上是禁止的。

4.10 关系的连接

在连接关系方面，SQL-92 除了提供早期 SQL 版本所提供的基本的笛卡儿积机制外，还有多种其他机制，包括条件连接、自然连接和各种形式的外连接。增加的这些操作通常在 from 子句的子查询表达式中使用。

4.10.1 举例

我们用图 4-1 所示的关系 *loan* 与 *borrower* 来举例说明各种连接操作。

branch-name	*loan-number*	*amount*
Downtown	L-170	3000
Redwood	L-230	4000
Perryridge	L-260	1700

loan

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

图 4-1 关系 *loan* 和 *borrower*

先由简单的内连接开始，图 4-2 给出了以下表达式的结果。

loan **inner join** *borrower* **on** *loan . loan-number* = *borrower . loan-number*

branch-name	*loan-number*	*amount*	*customer-name*	*loan-number*
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230

图 4-2　在 *loan . loan-number* = *borrower . loan-number* 上，*loan* 与 *borrower* 做内连接的结果

这个表达式计算了 *loan* 与 *borrower* 之间的 theta 连接，连接条件是 *loan . loan-number* = *borrower . loan-number*。所得结果的属性由左边关系的属性加上右边关系的属性构成。注意属性 *loan-number* 在图中出现了两次，一次是从关系 *loan* 中得到，另一次是从关系 *borrower* 中得到。

可以用 as 子句重新命名结果关系的名字和其属性的名字，如下所示：

loan **inner join** *borrower* **on** *loan . loan-number* = *borrower . loan-number*
as *lb* (*branch*, *loan-number*, *amount*, *cust*, *cust-loan-num*)

第二次出现的 *loan-number* 已经被重新命名为 *cust-loan-num*。在连接结果中属性的排列次序对重命名是非常重要的。

接下来考虑 left outer join 操作用法的例子：

loan **left outer join** *borrower* **on** *loan . loan-number* = *borrower . loan-number*

左外连接的计算如下。首先像前面那样先计算内连接的结果，然后，对左边关系 *loan* 中的每个与右边关系 *borrower* 中的任何元组在内连接时都不匹配的元组 t，在结果中都加入一个元组 r。r 的从左边关系中得到的属性值被赋为 t 中的值，而 r 的其他属性值被赋为空。结果关系如图 4-3 所示。元组（Downtown，L-170，3000）和（Redwood，L-230，4000）能和 *borrower* 中的元组连接，同时出现在内连接的结果中，因而也出现在左外连接的结果中。另一方面，元组（Perryridge，L-260，1700）与 *borrower* 中的任何元组都不匹配，因此元组（Perryridge，L-260，1700，null，null）出现在左外连接的结果关系中。

branch-name	*loan-number*	*amount*	*customer-name*	*loan-number*
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230
Perryridge	L-260	1700	*null*	*null*

图 4-3　在 *loan . loan-number* = *borrower . loan-number* 上，*loan* 与 *borrower* 做左外连接的结果

最后，让我们考虑 natural join 操作用法的例子：

loan **natural inner join** *borrower*

这个表达式计算两个关系的自然连接。关系 *loan* 与 *borrower* 中唯一相同的属性名是 *loan-number*。此表达式计算的结果如图 4-4。这个结果和图 4-2 所示用 on 条件进行内连接的结果相似，

branch-name	*loan-number*	*amount*	*customer-name*
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith

图 4-4　*loan* 与 *borrower* 自然内连接的结果

因为它们实际上有相同的连接条件。然而，属性 *loan-number* 在自然连接的结果中只出现一次，而用 on 条件连接的结果中它出现两次。

4.10.2 连接类型和条件

在上节中我们见过了 SQL-92 中允许的连接操作的例子。连接操作以两个关系为输入，将另一个关系作为结果返回。虽然外连接表达式通常用在 from 子句中，但是它们也可以使用在任何可以使用关系的地方。

SQL-92 中每一种连接操作都包含一个连接类型和一个连接条件。连接条件决定了两个关系中哪些元组相互匹配，以及连接结果中出现哪些属性。连接类型决定了如何处理连接条件不匹配的元组。图 4-5 给出了一些允许的连接类型和连接条件。第一种连接类型是内连接，其余三种是外连接。三个连接条件分别是自然连接、已在前面见过的使用 on 条件的连接、以及使用 using 条件的连接，最后这种连接将在以后讨论。

连接类型
inner join
left outer join
right outer join
full outer join

连接条件
natural
on <谓词>
using $(A_1, A_2, \ldots, A_n)$

图 4-5 连接类型与连接条件

对外连接而言，连接条件是必须的；但对内连接而言，连接条件则是可选的（如果省略，将产生笛卡儿积）。按照语法，关键字 natural 如前所示出现在连接类型前，而 on 和 using 出现在连接表达式的末尾。关键字 inner 和 outer 是可选的，因为根据连接类型的其余内容可以推断出连接是内连接还是外连接。

连接条件 natural 的含义就两个关系中哪些元组相互匹配而言是显而易见的。自然连接的结果关系中属性的顺序如下：连接属性（即两个关系公共的属性）按照他们在左边关系中的顺序首先出现，然后是左边关系中所有非连接属性，最后是右边关系中所有非连接属性。

右外连接这一连接类型和左外连接是对称的。右边关系中与左边关系中任何元组都不匹配的元组被补上空值，然后加入到右外连接的结果中去。

下面的表达式是自然连接条件与右外连接类型相结合的例子：

loan **natural right outer join** *borrower*

表达式的结果如图 4-6 所示。自然连接这一连接条件决定了结果关系中的属性，因此 *loan-number* 只出现了一次。结果中前两个元组是从关系 *loan* 与 *borrower* 中自然连接得来。右边关系的元组（Hayes, L-155）与左边关系 *loan* 中任何元组都不匹配，因此元组（*null*, Hayes, L-155）出现在结果关系中。

branch-name	*loan-number*	*amount*	*customer-name*
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
null	L-155	null	Hayes

图 4-6 *loan* 与 *borrower* 自然右外连接的结果

连接条件 using $(A_1, A_2, \cdots, A_n)$ 类似于自然连接的条件，只是连接属性不再是两个关系中所有的公共属性，而是属性 $A_1, A_2, \cdots, A_n$。$A_1, A_2, \cdots, A_n$ 都必须是两边关系中的公共属性，而且在结果关系中只出现一次。

全外连接类型是左外连接与右外连接的组合。在内连接的结果被计算出来之后，左边关系中与右边关系中任何元组都不匹配的元组被添上空值并加到结果关系中。同样，右边关系中与左边关系中任何元组都不匹配的元组也被添上空值并加到结果关系中。

例如，表达式

loan **full outer join** *borrower* **using** (*loan-number*)

的结果如图 4-7 所示。

branch-name	*loan-number*	*amount*	*customer-name*
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perryridge	L-260	1700	*null*
null	L-155	null	Hayes

图 4-7　*loan* 与 *borrower* 使用 (*loan-number*) 全外连接的结果

作为使用外连接操作的另一个例子，查询“找出所有在银行有存款而无贷款的客户”可以这样写出：

select *d-CN*
from (*depositor* **left outer join** *borrower*
　　on *depositor* . *customer-name* = *borrower* . *customer-name*)
　　as *db*1 (*d-CN*, *account-number*, *b-CN*, *loan-number*)
where *b-CN* **is** *null*

同样，用自然全外连接可以书写查询“找出所有在银行或者有贷款，或者有存款（但不能二者都有）的客户”如下：

select *customer-name*
from (*depositor* **natural full outer join** *borrower*)
where *account-number* **is** *null* **or** *loan-number* **is** *null*

SQL-92 还提供另外两种连接类型 cross join 和 union join。前者等价于没有连接条件的内连接，而后者等价于使用 false 条件的全外连接，即内连接结果为空。

4.11　数据定义语言 DDL

在大部分与 SQL 和数据库有关的讨论中，我们使用了一些给定的关系。当然，数据库中的关系集合必须用数据定义语言 DDL 来向系统说明。

SQL DDL 不仅允许定义一组关系，也可以说明关于各关系的信息，包括：

- 各关系的模式。
- 各属性的值域。
- 完整性约束。
- 各关系要维护的索引集合。
- 各关系的安全性和权限信息。
- 各关系在磁盘上的物理存储结构。

下面将讨论模式定义和值域，其他有关 SQL DDL 的特性将在第 6 章讨论。

4.11.1 SQL 中的域类型

SQL-92 标准支持很多种预定义的域类型，其中包括下列类型：

- char（n）为固定长度的字符串，用户指定长度 n。也可以用全称 character。
- varchar（n）为可变长度的字符串，用户指定最大长度 n。等价于全称 character varying。
- int 为整数类型（和机器有关的整数的有限子集），等价于全称 integer。
- smallint 为小整数类型（和机器有关的 int 域类型的子集）。
- numeric（p，d）为一个定点数，精度由用户指定。这个数有 p 位数字（还有一个符号位），其中 d 位数字在小数点右边。所以在这样一种类型的字段上，numeric（3，1）可以精确储存 44.5，而 444.5 或 0.32 这样的数不能精确存储。
- real、double precision 为浮点数与双精度浮点数，精度与机器有关。
- float（n）为一个浮点数，其中用户指定至少 n 位数字的精度。
- date 为日期，包括年（四位）、月和日。
- time 为时间，包括小时、分和秒。

变长字符串、日期和时间在 SQL-89 标准中不支持。

SQL 允许在各种数据类型的域上进行算术运算和比较运算，比较运算在上面列出的类型上均可进行。SQL 还支持一种数据类型 interval，允许对 date、time 和 interval 进行计算。例如，假设 x，y 都是 date 类型，那么 $x-y$ 就是 interval 类型，其值为 x 到 y 的天数。同样，在 date 或 time 上加减 interval 类型将分别得到新的 date 或 time 类型的值。

兼容域类型之间进行比较常常是有用的。例如，既然每一个 smallint 数都是 int，那么若 x 是 small int 类型，y 是 int 类型（或相反），则 $x<y$ 的比较是有意义的。通过将小整数 x 转换为整数来进行这样的比较，这种转换称为强制类型转换。在通常的编程语言和数据库系统中经常使用强制类型转换。

例如，假设 *customer-name* 的域是长为 20 的字符串，而 *branch-name* 的域是长为 15 的字符串，虽然这两个字符串长度不一致，SQL 标准还是认为这两个域兼容。

在第 3 章已经讨论到，*null* 是所有域都可以取的值，但有些属性取空值是不恰当的。假设关系 *customer* 的某个元组的属性 *customer-name* 为空值，那么这样一个元组可能包含一个没有名字的客户的街道和城市信息，因而该元组没有真正有用的信息。在这种情况下，我们希望通过将空值从 *customer-name* 的域中排除出去来禁止空值出现。

SQL 允许对属性域的声明中包含对 not null 的说明，这样就禁止了在该属性上插入空值。任何一个试图将空值插入 not null 域的数据库修改将产生一个错误诊断信息。许多情况下不允许空值出现是必要的，特别指出的是关系模式中的主码不能为空值。因此，在关系 *customer* 中，不允许 *customer-name* 为空值，因为 *customer-name* 是关系 *customer* 的主码。

SQL-92 允许用 create domain 子句定义域，如下例：

create domain *person-name* **char** (20)

这样我们就可以像使用预定义域一样，用域名 *person-name* 来定义属性类型。

4.11.2 SQL 的模式定义

用 create table 命令定义 SQL 关系：

create table r $(A_1 D_1,\ A_2 D_2,\ \cdots,\ A_n D_n,$
$<\text{integrity-constraint}_1>,$
$\cdots,$
$<\text{integrity-constraint}_k>)$

其中 r 是关系名，A_i 是关系模式 r 的一个属性名，D_i 是属性 A_i 域值的域类型。允许的完整性约束包括

primary key $(A_{j1},\ A_{j2},\ \cdots,\ A_{jm})$

和

check (P)

primary key 声明表示属性 A_{j1}，A_{j2}，…，A_{jm} 构成该关系的主码。虽然主码声明是可选的，但指定每个关系的主码通常会更好一些。check 子句说明关系中每个元组必须满足谓词 P，这是 SQL-92 标准提出的。create table 语句还包括其他一些完整性约束，我们将在第 6 章讨论。

图 4-8 是银行数据库的部分 SQL DDL 定义。记住 SQL 中 null 是每个类型的合法值，除非该类型像图中那样特别指明为 not null。

```
create table customer
    (customer-name    char(20) not null,
     customer-street  char(30),
     customer-city    char(30),
     primary key (customer-name))

create table branch
    (branch-name      char(15) not null,
     branch-city      char(30),
     assets           integer,
     primary key (branch-name),
     check (assets >= 0))

create table account
    (account-number   char(10) not null,
     branch-name      char(15),
     balance          integer,
     primary key (account-number),
     check (balance >= 0))

create table depositor
    (customer-name    char(20) not null,
     account-number   char(10) not null,
     primary key (customer-name, account-number))
```

图 4-8　银行数据库的部分 SQL 数据定义

被指定为关系主码的属性必须非空且唯一，即该关系没有两个元组在所有主码属性上取值相等。如果一个新插入的或修改后的元组在主码属性上为空值，或其主码值和关系中其他元组取值相等，更新将被禁止并设置错误标识。同样，如果 check 条件不满足，更新也将被禁止并设置错误标识。注意，图 4-8 中的 not null 声明在 SQL-92 中是多余的，因为它们定义在作为主码的一部分的属性上。然而，在 SQL-89 中主码属性必须像图中那样以显式的方式声明为 not null。

通常使用 check 子句是为了保证属性值满足某些前提条件，相当于建立一个强有力的类型系统。例如，建立关系 *branch* 的 create table 命令中 check 子句中检查 assets 值非负。另举一例，考虑：

```
create table student
  (name          char (15) not null,
   student-id    char (10) not null,
   degree-level  char (15) not null,
   primary key (student-id),
   check (degree-level in ("Bachelors", "Masters", "Doctorate")))
```

在这里用 check 语句模拟枚举类型，声明 *degree-level* 必须为 "Bachelors"、"Masters"、"Doctorate" 之一。

当插入或修改元组时，前面所讲的 check 条件检查是很容易的。然而，一般 check 语句会更复杂（也更难以检查），因为 check 语句可以包含指向其他关系的子查询。例如，在关系 *depositor* 上可以指定如下 check 语句：

```
check (branch-name in (select branch-name from branch))
```

上述 check 语句保证在关系 *depositor* 中每个元组的 *branch-name* 值必须是关系 *branch* 中一个分支机构的名字。因此，上述 check 语句不仅在插入、修改关系 *depositor* 中的元组时要检查，在修改关系 *branch* 时（例如，删除或修改关系 *branch* 中元组）也要检查。这样的条件是外码约束这类条件的一个例子，我们将在第 6 章中详细讨论外码约束。如果想保证数据完整性，像上述例子中那样复杂的 check 条件是非常有用的，但使用时必须小心，因为它们可能要花费很多时间来检查。

一个新建的关系初始是空的，可以用 insert 命令把数据装入关系中。许多关系数据库产品都有特殊的批量装入工具，可以把一组初始元组装入关系中。

要从 SQL 数据库中删除关系可以用 drop table 命令。drop table 命令从数据库中删除被删关系的所有信息。命令

```
drop table r
```

是比命令

```
delete from r
```

更危险的行为。后者保留关系 r，但 r 中所有元组被删除。前者不仅删除 r 中所有元组，而且 r 的关系模式定义也被删除。在 r 被删除后，没有元组能被插入到 r 中，除非用 create table 命令重建 r。

在 SQL-92 中可以用 alter table 命令为已有关系添加属性，关系中所有元组在这个新属性上都将赋值 null。alter table 命令格式如下：

```
alter table r add A D
```

其中 r 是已有关系的名字，A 是将加到关系 r 中的一个属性，D 是添加属性的域。也可以用如下命令从关系中删除属性

alter table r **drop** A

其中 r 是已有关系的名字，A 是关系中的一个属性。

4.12 嵌入式 SQL

SQL 提供了一种强有力的声明性查询语言。用 SQL 写查询语句比用通用编程语言编码实现相同的查询要简单的多。然而，使用通用编程语言访问数据库还是有必要的，原因至少有下面两条：

1）用 SQL 不能表达所有查询要求，因为 SQL 并没提供通用编程语言的所有表达能力。也就是说，有可能存在这样的查询语句，可以用 Pascal、C、Cobol 或 Fortran 写出，而用 SQL 却不能做到。要写出这样的查询，可以将 SQL 嵌入到一种更强大的语言中。SQL 的设计目标是用其自身语言写出的查询语句可以被自动优化和高效执行，而提供编程语言的所有能力将使得自动优化极难进行。

2）非声明性动作（例如打印一份报告、和用户交互，或者把一次查询的结果送到一个图形用户界面中）都不能用 SQL 实现。一个应用程序通常包括好几个部件，查询或更新数据只是其中一个部件，而其他部件则用通用编程语言实现。对于一个集成的应用来说，用编程语言写出的程序必须能够访问数据库。

SQL 标准定义了许多语言的嵌入式 SQL，例如 Pascal、PL/I、Fortran、C 和 Cobol。SQL 查询所嵌入的语言称为宿主语言，宿主语言中使用的 SQL 结构称为嵌入式 SQL。

使用宿主语言写出的程序可以通过嵌入式 SQL 的语法访问和修改数据库中的数据。嵌入式 SQL 使编程人员操作数据库的能力更强。在嵌入式 SQL 中，所有的查询都由数据库系统处理，然后程序可以一次一个元组（记录）地获取查询结果。

一个使用嵌入式 SQL 的程序在编译前必须先由一个特殊的预处理器进行处理。嵌入的 SQL 请求被宿主语言的声明以及允许运行时刻执行数据库访问的过程调用所代替。然后，产生的程序由宿主语言编译器编译。为使预处理器识别嵌入式 SQL 请求，我们使用 EXEC SQL 语句，格式如下：

EXEC SQL ＜embedded SQL statement＞ END-EXEC

嵌入式 SQL 的确切语法依赖于宿主语言。例如，当宿主语言是 C 或 Pascal 时，使用分号而不是 END-EXEC。

在应用程序的合适地方插入 SQL INCLUDE 语句，标志在此预处理器应插入特殊变量以用于程序和数据库系统间通信。在嵌入的 SQL 语句中可以使用宿主语言的变量，不过变量前面要加上冒号（:）以区别 SQL 变量。

嵌入式 SQL 语句的格式和本章描述的 SQL 语句类似，但这儿要指出一些重要的不同之处。

为了书写关系查询，我们使用 declare cursor 语句，查询的结果并不被计算，而程序必须用 open 和 fetch 语句（本章后面将讨论）得到结果元组。

考虑本章所使用的银行模式。假设有一个宿主变量 *amount*，我们想找出所有在银行有余额大于 *amount* 这样帐户的客户的名字和居住城市，可写出查询语句如下：

```
EXEC SQL
    declare c cursor for
    select customer-name, customer-city
    from depositor, customer, account
    where depositor.customer-name = customer.customer-name and
          account.account-number = depositor.account-number and
          account.balance > :amount
END-EXEC
```

上述表达式中的变量 *c* 被称为查询的游标。我们在 open 语句和 fetch 语句中使用这个变量标识该查询，open 语句是对查询求值，而 fetch 语句则把一个元组的值放到宿主变量中。

在这个例子中 open 语句如下：

EXEC SQL **open** *c* END-EXEC

该语句使得数据库系统执行这条查询并把执行结果存于一个临时关系中。如果 SQL 查询出错，数据库系统将在 SQL 通信区域（SQLCA）变量中存储一个错误诊断信息，该变量的声明是由 SQL INCLUDE 语句插入的。

程序可利用一系列的 fetch 语句获得结果元组。fetch 语句要求对应结果关系的每一个属性有一个宿主变量。在查询例子中，需要一个变量存储 *customer-name* 的值，另一个变量存储 *customer-city* 的值。假设这两个变量分别是 *cn* 和 *cc*，结果关系的一个元组可以通过这样的语句取得：

EXEC SQL **fetch** *c* **into** :*cn*, :*cc* END-EXEC

接下来应用程序就可以利用宿主编程语言的特性对 *cn* 和 *cc* 进行操作了。

一条单一的 fetch 请求只能得到一个元组。如果想得到所有的结果元组，程序中必须包含对所有元组执行的一个循环。嵌入式 SQL 在程序员对这种循环进行管理方面提供了支持，虽然关系在概念上是一个集合，查询结果中的元组还是有一定的物理顺序的。执行 SQL 的 open 语句后，游标指向结果中的第一个元组。执行一条 fetch 语句后，游标指向结果中的下一个元组。SQLCA 中有一个变量用于指明后面不再有待处理的元组，因此，可以用一个 while 循环（或类似语句，视宿主语言而定）来处理结果中的每一个元组。

必须使用 close 语句来告诉数据库系统删除用于保存查询结果的临时关系。对于上例，该语句格式如下：

EXEC SQL **close** *c* END-EXEC

用于数据库修改（update、insert 和 delete）的嵌入式 SQL 表达式不返回结果，因此，这种语句表达起来相对简单。数据库修改请求格式如下：

EXEC SQL ＜任何有效的 **update**、**insert** 或 **delete** 语句＞ END-EXEC

前面带冒号的宿主语言的变量可以出现在数据库修改语句的表达式中。如果在语句执行过程中出错，SQLCA 中将设置错误诊断信息。

SQL-92 提供动态 SQL 语句，允许程序在运行时刻动态构造、提交 SQL 查询。与此相反，嵌入式 SQL 语句必须在编译时刻全部确定，并交由嵌入式 SQL 预处理器编译。使用动态 SQL，程序能够在运行时将 SQL 查询生成为字符串（也许基于用户输入），同时可以立即执行该查询或使其为后续使用做好准备。准备动态 SQL 语句是对其进行编译，以后使用准备好的语句就使用其编译过的版本。下面是在 C 程序中使用动态查询语句的例子：

```
char * sqlprog = "update account set balance = balance * 1.05
        where account-number = ?"
EXEC SQL prepare dynprog from : sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using : account;
```

该动态 SQL 程序包含一个?，这是一个占位符，其所在位置的值在 SQL 程序执行时提供。SQL-92 还包含一种模块语言，允许在 SQL 中定义过程。一个模块通常包含多个 SQL 过程。每个过程包含一个名字、可选参数和一个 SQL 语句。这些过程可以由外部语言直接调用而无须特别语法。

4.13　其他 SQL 特性

嵌入式 SQL 允许宿主语言程序访问数据库，但对把结果提交给用户或产生报表毫无帮助。大多数商业数据库产品提供一种特殊语言，帮助应用程序员创建用户界面的屏幕模板，并为报表的生成格式化数据。这种特殊的语言称为第四代语言。

有些第四代语言甚至提供高级结构，可直接表达关系上的循环，程序员无须处理有关游标管理的细节问题。然而，不像 SQL 和嵌入式 SQL 那样，第四代语言至今没有一个普遍接受的标准，每种产品都提供各自的第四代语言。

SQL 92 标准定义了 SQL 会话和 SQL 环境的概念。SQL 会话提供了客户与服务器（可以是远端的）的抽象。客户连接到 SQL 服务器上，建立一个会话，执行一系列语句并最终断开连接。在普通 SQL 命令基础上，会话中还可以包括将会话中执行的工作提交或回滚的命令。

SQL 环境由几部分组成，包括用户标识和模式。多种模式的存在允许不同应用程序和不同用户独立工作。所有通常的 SQL 语句，包括 DDL 和 DML 语句，都工作在模式的上下文中，可以用 create schema 和 drop schema 语句创建或撤消模式。如果不显式设置模式，和用户标识有关的缺省模式将被使用，因此不同用户将看到他们自己的模式。模式的另一用途是允许一个应用程序有多个版本，产品化版本和其他测试版将在同一个数据库系统上运行。

4.14　总结

商业数据库系统并不使用第 3 章的简洁的、形式化查询语言，而在形式化查询语言基础上提供许多便利的方法。商业数据库系统不仅提供对数据库的查询结构，还提供用于更新、插入、删除信息的结构。在这里讨论的 SQL 语言已被商品化产品广泛接受。

我们学习了用 SQL 语言表达查询和修改数据库（对数据库的修改可能在元组中产生空值）；讨论了空值的产生以及 SQL 查询语言如何处理在含有空值的关系上的查询；展示了如何定义关系和视图。更多的有关 SQL DDL 的细节见第 6 章和第 19 章。

习题

4.1 考虑图 4-9 定义的保险公司数据库，其中加下划线的是主码。对这个关系数据库写出如下 SQL 查询语句：

person (*ss#*, *name*, *address*)
car (*license*, *year*, *model*)
accident (*date*, *driver*, *damage-amount*)
owns (*ss#*, *license*)
log (*license*, *date*, *driver*)

图 4-9 保险公司数据库

(a) 找出在 1989 年其车辆出过车祸的人员总数。
(b) 找出和 John Smith 的车有关的车祸数量。
(c) 为数据库添加一个新客户。
(d) 删除 John Smith 的马自达（Mazda）。
(e) 为 Jones 的丰田车（Toyota）加一条车祸记录。

4.2 考虑图 4-10 的雇员数据库，为下面每个查询语句写出 SQL 表达式：
(a) 找出所有为 First Bank Corporation 工作的雇员名字。
(b) 找出所有为 First Bank Corporation 工作的雇员名字和居住城市。
(c) 找出所有为 First Bank Corporation 工作且薪金超过 $ 10 000 的雇员名字、居住街道和城市。
(d) 找出数据库中所有居住城市和公司所在城市相同的雇员。
(e) 找出数据库中所有居住街道和城市与其经理相同的雇员。
(f) 找出所有不为 Frist Bank Corporation 工作的雇员。
(g) 找出数据库中所有工资高于 Small Bank Corporation 每一个雇员的雇员。
(h) 假设一个公司可以在好几个城市有分部。找出与 Small Bank Corporation 在同一城市的所有分公司。
(i) 找出工资高于其所在公司雇员平均工资的所有雇员。
(j) 找出雇员最多的公司。
(k) 找出工资总额最小的公司。
(l) 找出平均工资高于 First Bank Corporation 平均工资的所有公司。

4.3 考虑图 4-10 的关系数据库，为下列查询写出 SQL 表达式。

employee (*employee-name*, *street*, *city*)
works (*employee-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*employee-name*, *manager-name*)

图 4-10 雇员数据库

(a) 修改数据库使 Jones 居住在 Newtown 市。

(b) 为 First Bank Corporation 所有雇员增加 10% 的薪水。

(c) 为 First Bank Corporation 所有经理增加 10% 的薪水。

(d) 为 First Bank Corporation 所有雇员增加 10% 的薪水，若薪水超过 $100 000，只增加 3%。

(e) 删除 Small Bank Corporation 雇员的关系 *works* 中的所有元组。

4.4 给出如下关系模式：

$$R = (A, B, C)$$
$$S = (D, E, F)$$

设关系 r (R) 和 s (S) 已知，给出下列查询等价的 SQL 表达式。

(a) $\Pi_A (r)$

(b) $\sigma_{B<17} (r)$

(c) $r \times s$

(d) $\Pi_{A,F} (\sigma_{C=D} (r \times s))$

4.5 设 $R = (A, B, C)$，r_1 和 r_2 是模式 R 上的关系。给出与下列查询等价的 SQL 表达式。

(a) $r_1 \cup r_2$

(b) $r_1 \cap r_2$

(c) $r_1 - r_2$

(d) $\Pi_{AB} (r_1) \bowtie \Pi_{BC} (r_2)$

4.6 设 $R = (A, B)$，$S = (A, C)$，r (R) 和 s (S) 为关系。为下列查询写出 SQL 表达式。

(a) $\{<a> \mid \exists b\ (<a, b> \in r \wedge b = 17)\}$

(b) $\{<a, b, c> \mid <a, b> \in r \wedge <a, c> \in s)\}$

(c) $\{<a> \mid \exists c(<a, c> \in s \wedge \exists b_1, b_2(<a, b_1> \in r \wedge <c, b_2> \in r \wedge b_1 > b_2))\}$

4.7 证明在 SQL 中，<>all 等价于 not in。

4.8 考虑图 4-10 的关系数据库。用 SQL 定义一个视图，属性有经理名字 *manager-name* 和其下属员工的平均工资。解释为何数据库系统不允许对这个视图进行更新。

4.9 考虑 SQL 查询：

```
select p.a1
from p, r1, r2
where p.a1 = r1.a1 or p.a1 = r2.a1
```

在何种情况下该查询选择的 $p.a1$ 的值或在 $r1$ 或 $r2$ 中？仔细考虑 $r1$ 或 $r2$ 为空的情况。

4.10 SQL-92 提供 case 语句，如下定义：

```
case
        when pred1 then result1
        when pred2 then result2
        ...
        when predn then resultn
```

else $result_0$

end

若 $pred_1$，$pred_2$，…，$pred_n$ 中第一个满足的谓词是 $pred_i$，则此操作返回 $result_i$；若没有一个谓词被满足，操作返回 $result_0$。

假设有关系 *marks*（*student-id*，*score*），我们希望基于如下标准为学生评定等级：*score*<40 得 *F*，40≤*score*<60 得 *C*，60≤*score*<80 得 *B*，80≤*score* 得 *A*。写出下列查询：

（*a*）在关系 *marks* 基础上求出每个学生的等级。

（*b*）找出各等级的学生数。

4.11 用习题 4.10 给出的 case 操作在一个更新语句中完成关系 *account* 中每个元组的更新："若 *balance*>10 000 则增加 6%，若 *balance*≤10 000 则增加 5%"。

4.12 SQL-92 提供 *n* 维数组操作叫 coalesce，定义如下：coalesce（A_1，A_2，…，A_n）。该操作返回 A_1，A_2，…，A_n 中第一个非空值 A_i，否则返回空值。试用 case 操作实现 coalesce 操作。

4.13 设模式 *A*（*name*，*address*，*title*）和 *B*（*name*，*address*，*salary*），关系 *a*，*b* 分别是这两个模式上的关系。写出用 on 条件的全外连接操作和 coalesce 操作如何表达 *a* natural full outer join *b*。要求结果中没有重复的属性 *name* 和 *address*，即使 *a* 或 *b* 在 *name* 和 *address* 上有空值结果也正确。

4.14 给出图 4-10 中雇员数据库的 SQL 模式定义。为每个属性选择合适的域，并为每个关系模式选择合适的主码。

4.15 为在习题 4.14 中定义的模式书写 check 条件以保证：

(a) 每个雇员工作的公司所在城市和其居住城市相同。

(b) 没有雇员的工资比他的经理高。

4.16 描述何种情况下你会选择使用嵌入式 SQL，而不是单独使用 SQL 或某种通用编程语言。

文献注解

SQL 的最早版本 Sequel 2 由 Chamberlin 等［1976］描述。Sequel 2 是从语言 Square［Boyce 等 1975］和 Sequel［Chamberlin and Boyce 1974］派生出来的。美国国家标准 SQL-86 在 ANSI［1986］中描述。IBM 系统应用体系结构对 SQL 的定义由 IBM［1987］给出。SQL-89 和 SQL-92 官方标准可分别可从 ANSI［1989］和 ANSI［1992］获得。SQL-92 在 U.S.Dept.of Commerce［1992］与 X/Open［1992，1993］中也有描述。SQL 标准的下一版本现正在制订中，暂时叫做 SQL-3。

Cannan and Otten［1993］是专门讲述 SQL 的书。描述 SQL-92 语言的教科书包括 Date and Darwen［1993］与 Melton and Simon［1993］。Date and Darwen［1993］和 Date［1993］包括了对 SQL-92 的评价。Melton［1993］和 Melton［1996］讨论了 SQL-3。

许多数据库产品支持标准说明以外的一些特性，关于这些特性的更多信息可在各产品的 SQL 用户手册中找到。Date and White［1993］与 Martin 等［1989］介绍了 DB2 中实现的所有 SQL。Valduriez and Gardarin［1989］是多种关系数据库系统的一个总览。

SQl 查询的处理，包括算法和性能等问题，在本书第 12 章讨论。关于这些问题的参考文献也在第 12 章中。

第5章　其他关系语言

在第4章，我们已经描述了市场上最具影响的商品化关系数据库语言——SQL。本章将研究另外三种语言：QBE、Quel和Datalog。挑选这些语言是因为它们代表不同风格。QBE建立在域关系演算基础上；Quel建立在元组关系演算基础上。这两种语言不仅在研究性的数据库系统中影响广泛，在市场上也是如此。Datalog的文法以Prolog为模型。尽管Datalog现在并没有在商业上应用，但它已经用于一些研究性数据库系统中。

像SQL一样，QBE和Quel除了查询数据库外，还有一些其他功能。其中包括定义数据结构、修改数据库中的数据、说明安全性约束条件等特性。

在这里我们并不想提供这些语言的完整的用户手册，而是只介绍基本结构和概念。实现某种语言的各种方法也许在一些细节上会不同，也许只是支持整个语言的一个子集。

5.1　Query-by-Example

Query-by-Example（QBE）既指一种数据操纵语言，也指包含这种语言的数据库系统。QBE数据库系统是70年代初由IBM的T.J.Watson研究中心研制的。QBE数据操纵语言后来被用在IBM的Query Management Facility（QMF）中。现在一些个人电脑的数据库系统支持不同种类的QBE语言。本节只考虑数据操纵语言，这种语言有两个显著的特点：

1）不同于大多数查询语言和编程语言，QBE的文法是二维的，查询语句看起来像表格。一维查询语言（如SQL）的一个查询语句可以在一行（可能很长）中写完，而二维查询语言则至少需要两行才能表达一个查询。（QBE也有一维的版本，但我们不打算讨论。）

2）QBE的查询是用“例子”表达的。用户不是写一个过程获得所需答案，而是举出所需答案的一个例子，系统会把这个例子一般化并计算出查询的答案。

尽管有这些独特的特性，QBE和域关系演算仍是密切相关的。

在QBE中用框架表表达查询，这些表表明了关系模式，如图5-1所示。用户不是把所有的框架表都显示出来以致使屏幕混乱不堪，而是根据所给查询选择所需的框架表，并在其中填入示例行。示例行可以包含常量和示例元素（即域变量）。为了避免混淆，QBE中域变量前加下划线（_），例如_x，而常量无需任何标志。这个习惯和大多数其他语言相反，其他语言通常做法是常量用引号标出，而变量无需标志。

5.1.1　在一个关系上的查询

回到银行的例子，假设要找出所有在Perryridge分支机构贷款的贷款号，取出关系*loan*的框架表，并按以下填写

loan	*branch-name*	*loan-number*	*amount*
	Perryridge	P._x	

上面的查询使系统去查找关系*loan*中所有*branch-name*属性值为“Perryridge”的元组。对

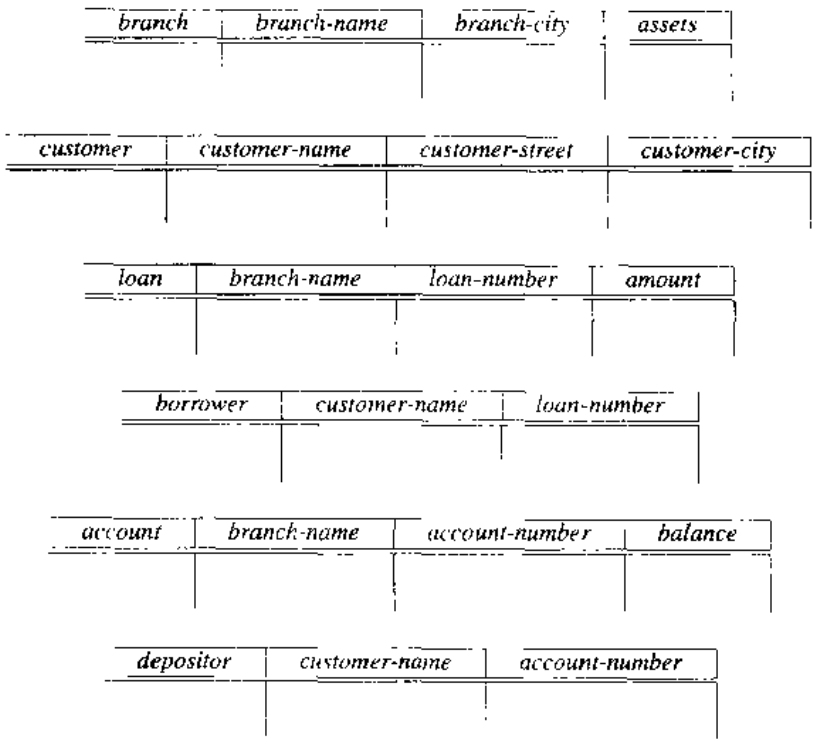
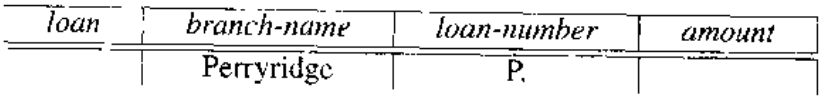


图 5-1　银行例子的 QBE 框架表

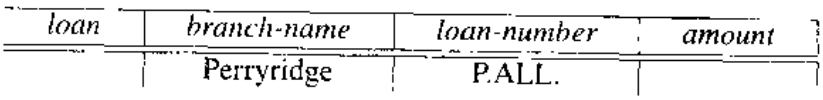
每一个符合条件的元组，它的 *loan-number* 属性值将赋给变量 x。变量 x 的值将被“打印”(就是显示出来)，因为在 *loan-number* 列中变量 x 前有命令 P.。注意这个结果和如下域关系演算表达的查询结果类似：

$$\{\langle x\rangle \mid \exists b, a\ (\langle b, x, a\rangle \in loan \wedge b = \text{“Perryridge”})\}$$

QBE 假设行中的每个空位包含唯一的一个变量。因此，如果变量名在一个查询中只出现一次，那么就可以省略该变量名。前面的查询可以重写如下：



QBE（与 SQL 不同）将自动删除重复元组．如果不想删除重复元组，可在 P. 命令后加上 ALL. 命令



如果要显示整个关系 *loan*，可以建立起每个域都包含 P. 的一行。另一种省事的做法是只在关系名所起头的列中填入一个 P.

loan	*branch-name*	*loan-number*	*amount*
P.			

除了等于比较外，QBE 还允许查询中出现算术比较（如>），例如在“找出所有贷款额大于 $ 700 的贷款的贷款号”中

loan	*branch-name*	*loan-number*	*amount*
		P.	>700

比较中只能有一个算术表达式出现在比较运算符的右边（例如，> (_x + _y 20)）。表达式中既可以有变量，也可以有常量。比较运算符左侧必须为空。QBE 支持的算术运算包括 =、<、≤、>、≥和 ¬。

注意，限制比较运算符右侧只能出现一个算术表达式意味着不能比较两个名字不同的变量。我们不久将处理这个难题。

再看一个例子，考虑查询“找出所有不在 Brooklyn 的分支机构的名字”，可以如下写出

branch	*branch-name*	*branch-city*	*assets*
	P.	¬ Brooklyn	

QBE 设置变量的主要目的是为了让某些特定元组在特定属性上有相同值。考虑查询“找出所有由 Smith 和 Jones 联合贷款的贷款号”

borrower	*customer-name*	*loan-number*
	“Smith”	P._x
	“Jones”	_x

要执行以上查询，系统会找出关系 *borrower* 中所有 *loan-number* 相等的元组对，其中一个元组的 *customer-name* 属性为 Smith，而另一个元组为 Jones，然后，*loan-number* 属性的值被显示出来。

将此查询与查询“找出所有贷款者为 Smith 或 Jones 或二人联合贷款的贷款号”进行比较。后者可写为

borrower	*customer-name*	*loan-number*
	“Smith”	P._x
	“Jones”	P._y

上述两个查询语句的不同之处关键在于：前面一个查询在两行中使用了相同的域变量

(x)，而后者使用了不相同的域变量（x 和 y)。注意，在域关系演算中，前面的查询可写作

$$\{\langle l\rangle \mid \exists x\ (\langle x, l\rangle \in borrower \wedge x = \text{“Smith”}) \wedge \exists x\ (\langle x, l\rangle \in borrower \wedge x = \text{“Jones”})\}$$

而后一个查询可写为

$$\{\langle l\rangle \mid \exists x\ (\langle x, l\rangle \in borrower \wedge x = \text{“Smith”}) \vee \exists x\ (\langle x, l\rangle \in borrower \wedge x = \text{“Jones”})\}$$

作为另一个例子，考虑查询“找出所有与 Jones 居住城市相同的客户”

customer	*customer-name*	*customer-street*	*customer-city*
	P._x		_y
	Jones		y

5.1.2　在多个关系上的查询

QBE 允许跨越多个不同关系进行查询（类似于关系代数中的笛卡儿积或自然连接)。不同关系的连接是通过使用变量令某些元组在某些属性上取值相等来实现的。来看一个例子，假设想找出所有在 Perryridge 分支机构有贷款的客户姓名，可以写为

loan	*branch-name*	*loan-number*	*amount*
	Perryridge	_x	

borrower	*customer-name*	*loan-number*
	P._y	_x

要执行上面的查询，系统在关系 *loan* 中找出所有 *branch-name* 属性为“Perryridge”的元组。对每个元组，系统在关系 *borrower* 中找出和它有相同 *loan-number* 属性的元组。这些元组的 *customer-name* 属性值将被显示。

采用类似上例的方法，可以写出查询“找出所有在银行既有贷款又有存款的客户的名字”

depositor	*customer-name*	*account-number*
	P._x	

borrower	*customer-name*	*loan-number*
	_x	

现在考虑查询“找出所有在银行有存款但没有贷款的客户名字”。在 QBE 中表示否定是通过在关系名下与示例行相邻的位置写一个 not 符号（¬)

depositor	*customer-name*	*account-number*
	P._x	

borrower	*customer-name*	*loan-number*
¬	_x	

将这个查询与先前的查询“找出所有在银行既有贷款又有存款的客户的名字”对比，唯一的不同是在 *borrower* 框架表中示例行前有一个 ¬。然而，这个区别对查询处理有重要影响。QBE 将找出所有符合以下条件的值 x：

1）关系 *depositor* 中有一个元组的 *customer-name* 为域变量 x。

2）关系 *borrower* 中不存在 *customer-name* 属性值与域变量 x 相同的元组。

“¬”可以读作“不存在”。

把“¬”放在关系名下而不是一个属性名下，这一点很重要。在属性名下的“¬”表示“≠”。因此，可以如下表达“找出至少有两个帐户的所有客户”

depositor	*customer-name*	*account-number*
	P._x	_y
	_x	¬ _y

在英语中，上面的查询可读作“显示所有这样的客户名，它出现在两个以上元组中，第二个元组中的 *account-number* 与前一个不同”。

5.1.3 条件框

有时候，在框架表中表达对域变量的所有约束很不方便甚至无法实现。为了克服这些困难，QBE 提供了条件框，允许在其中填入对任何域变量的一般约束表达式。假设把上节中的最后一个查询修改为“找出所有在银行中有两个以上帐户，且名字不叫‘Jones’的客户名”，我们想在这个查询中包含“x≠Jones”的约束条件，为了做到这点，可以通过添加条件框并在其中填入“x ¬ = Jones”来实现

conditions
x ¬ = Jones

另举一例。要找出所有余额在 $1300～$1500 之间的帐户的帐户号，可写为

account	*branch-name*	*account-number*	*balance*
		P.	_x

conditions
_x ≥ 1300
_x ≤ 1500

再看另外一个例子，考虑查询“找出所有这样的分支机构：它的资产至少比 Brooklyn 的一家分支机构要多”。这个查询可以这样写

branch	*branch-name*	*branch-city*	*assets*
	P._x		_y
		Brooklyn	_z

conditions
_y > _z

QBE 允许在条件框中出现复杂的算术表达式。可以这样表达查询“找出所有这样的分支机构：它的资产至少比 Brooklyn 的某家分支机构的总资产的两倍还要多”。和前例十分相似，只需把条件框改成

conditions
_y ≥ 2 * _z

QBE 还允许逻辑表达式出现在条件框中。逻辑运算符有单词 and 和 or 或者符号“&”和“|”。要找出所有帐户余额在 $1300～$2000 之间且不等于 $1500 的帐户号码，写为

account	*branch-name*	*account-number*	*balance*
		P.	_x

conditions
_x = (≥ 1300 **and** ≤ 2000 **and** ¬1500)

为了与常量集合进行比较，QBE 中的 or 结构还有一种非传统的用法。要找出那些或在 Brooklyn 或在 Queens 的分支机构，写为

branch	*branch-name*	*branch-city*	*assets*
	P.	_x	

conditions
_x = (Brooklyn **or** Queens)

5.1.4　结果关系

目前为止我们写的所有查询有一个共同之处：显示的结果出现在一个关系模式中。如果查询结果中包含来自几个关系模式的属性，需要一种机制来将想要的结果显示在一张表中。为了

这个目的，可以声明一个临时关系 *result*，使之包含查询结果的全部属性。在打印结果时只需要在 *result* 框架表中填入命令 P.。

作为示例，考虑查询“找出所有 Perryridge 分支机构帐户的 *customer-name*、*account-number* 和 *balance*”。在关系代数中将这样来完成该查询：

1）将关系 *depositor* 和 *account* 连接起来。

2）投影到 *customer-name*、*account-number* 和 *balance* 上。

在 *QBE* 中完成这个查询的做法如下：

1）建立一个框架表，名为 *result*，属性有 *customer-name*，*account-number* 和 *balance*。新建框架表名（即 *result*）不能和数据库中任何已存在的关系名相同。

2）写出查询。

这样得到的查询是：

account	*branch-name*	*account-number*	*balance*
	Perryridge	*y*	*z*

depositor	*customer-name*	*account-number*
	_*x*	*y*

result	*customer-name*	*account-number*	*balance*
P.	_*x*	_*y*	_*z*

5.1.5　元组的显示次序

QBE 可以让用户对显示关系中元组的次序加以一定控制。通过在适当列中添加命令 AO.（上升序）或 DO.（下降序）来进行这样的控制。因此，按字母升序列出银行中有帐户的客户名，可以写为

depositor	*customer-name*	*account-number*
	P.AO.	

QBE 提供将数据按多列进行排序和显示的机制。通过在每个排序操作符（A0 或 D0）后加用括号括起的数字来指明排序的顺序。因此，如果要按字母升序列出 Perryridge 分支机构的帐户号码，同时相应的帐户余额按降序排列，写为

account	*branch-name*	*account-number*	*balance*
	Perryridge	P.AO(1).	P.DO(2).

命令 P.AO（1）．说明首先对帐号排序，命令 P.DO（2）．说明接下来对每个帐户排序。

5.1.6　聚集操作

QBE 包含聚集操作符 AVG、MAX、MIN、SUM 和 CNT。为了保证不去除重复元组，必

须在这些操作符后面添加后缀 ALL。因此，要找出 Perryridge 分支机构所有帐户的结算，写为

account	*branch-name*	*account-number*	*balance*
	Perryridge		P.SUM.ALL.

假设在使用聚集操作符时希望删除重复元组。由于所有聚集操作符后面都必须有后缀 ALL，必须加入一个新操作符 UNQ 来指明想删除重复元组。因此，若想找出在银行有帐户的客户总数，写为

depositor	*customer-name*	*account-number*
	P.CNT.UNQ.ALL.	

通过使用 G. 操作符，QBE 还提供在元组的分组上计算函数的功能，这和 SQL 的 group by 结构类似。因此，要找出各银行平均余额，写为

account	*branch-name*	*account-number*	*balance*
	P.G.		P.AVG.ALL._x

平均余额是逐个银行计算的。*balance* 列的 P.AVG.ALL. 中的 ALL. 确保所有余额都考虑了进去。如果想按银行名字升序排列，可以用 P.AO.G. 代替 P.G.。

若只想找出那些平均帐户余额大于 $ 1200 的分支机构的平均帐户余额，可以加上这样的条件盒：

conditions
AVG.ALL._x > 1200

要找出那些在银行中有多于一个帐户的客户的名字、街道和城市，写为

customer	*customer-name*	*customer-street*	*customer-city*
P.	_x		

depositor	*customer-name*	*account-number*
	G._x	CNT.ALL._y

conditions
CNT.ALL._y > 1

其方法是先计算出关系 *depositor* 中银行里各个客户的存款数目。条件框则选择符合条件的客户名，然后通过与 *customer* 连接获得所需信息。

最后一个例子，考虑查询“找出那些在 Brooklyn 各分支机构均有存款的客户”

depositor	*customer-name*	*account-number*
	P.G._x	_y

account	*branch-name*	*account-number*	*balance*
	CNT.UNQ.ALL._z	_y	

branch	*branch-name*	*branch-city*	*assets*
	_z	Brooklyn	
	_w	Brooklyn	

conditions
CNT.UNQ.ALL._z = CNT.UNQ.ALL._w

域变量 w 中包含 Brooklyn 各分支机构的名字。因此，CNT.UNQ.ALL._w 就是在 Brooklyn 的分支机构的总数。域变量 z 中包含满足以下条件的分支机构的名字：

- 该分支机构位于 Brooklyn。
- 名为 x 的客户在该分支机构有帐户。

因此，CNT.UNQ.ALL._z 就是客户 x 在 Brooklyn 的分支机构拥有帐户的不同分支机构的数目。如果 CNT.UNQ.ALL._z = CNT.UNQ.ALL._w，那么客户 x 在 Brooklyn 的各个分支机构一定都有帐户。在这种情况下，x 将出现在显示的结果中（因为有命令 P.）。

5.1.7 数据库的修改

本节将展示如何用 QBE 添加、删除和修改信息。

1. 删除

从一个关系中删除元组的表达方式和查询类似，主要的区别是用 D. 取代 P.。在 QBE 中（不同于 SQL），既可以删除整个元组，也可以删除元组中指定列的值。如果是删除指定列的值，用“-”表示的空值将被插入。

注意命令 D. 只对一个关系起作用。如果想从多个关系中删除元组，必须对每个关系都使用命令 D.。

下面是 QBE 删除请求的一些例子：

- 删除客户 Smith

customer	*customer-name*	*customer-street*	*customer-city*
D.	Smith		

- 删除 Perryridge 分支机构的 *branch-city* 值

branch	*branch-name*	*branch-city*	*assets*
	Perryridge	D.	

这样，如果在删除操作前关系 *branch* 中有这样的元组（Perryridge，Brooklyn，50000），删除之后取代它的将是元组（Perryridge，－，50000）。

• 删除所有余额在 $1300～$1500 之间的贷款

loan	*branch-name*	*loan-number*	*amount*
D.		y	x

borrower	*customer-name*	*loan-number*
D.		y

conditions
$x = (\geqslant 1300$ **and** $\leqslant 1500)$

注意，要删除贷款，必须同时从关系 *loan* 和关系 *borrower* 中删除元组。

• 删除 Brooklyn 所有分支机构的所有帐户

account	*branch-name*	*account-number*	*balance*
D.	x	y	

depositor	*customer-name*	*account-number*
D.		y

branch	*branch-name*	*branch-city*	*assets*
	x	Brooklyn	

注意，在表达一个删除操作时，可以引用不进行删除操作的关系的信息。

2. 插入

要在关系中插入数据，可以指明待插入的元组，也可以写出一个查询，该查询的结果即为待插入的元组集合。要执行插入，需要查询表达式中写入 I. 操作符。显然，待插入元组属性值必须在其关系属性域中。

最简单的插入是插入一个元组的情形。假设想插入 Perryridge 分支机构一个帐号为‘A-9732’的帐户，其帐户余额为 $700，可以写为

account	*branch-name*	*account-number*	*balance*
I.	Perryridge	A-9732	700

也可以插入只含有部分信息的元组。若想在关系 *branch* 中插入一个名为“Capital”、城市为“Queens”，而总资产为空值的元组，可写为

branch	*branch-name*	*branch-city*	*assets*
I.	Capital	Queens	

更一般的情况是可能想在查询结果基础上插入元组。再次考虑这种情形：Perryridge 分支机构想为所有从该分支机构贷款的用户提供礼物，对应每笔贷款赠送一个余额为 200 美元的新存款帐户，并将贷款号作为存款帐户号，可写为

account	*branch-name*	*account-number*	*balance*
I.	Perryridge	_x	200

depositor	*customer-name*	*account-number*
I.	_y	_x

loan	*branch-name*	*loan-number*	*amount*
	Perryridge	_x	

borrower	*customer-name*	*loan-number*
	_y	_x

要执行以上插入操作，系统必须先从关系 *borrower* 中获取恰当信息，然后根据这些信息往关系 *depositor* 和 *account* 中插入正确的新元组。

3. 更新

有时候希望改变某个元组的某一属性值而不是该元组的全部值。为了这个目的，我们使用 U. 操作符。正如插入和删除那样，可以使用查询来确定待更新元组，但 QBE 不允许用户更新元组的主码。

假设想把 Perryridge 分支机构总资产值更新为 $ 10 000 000，这个更新操作可写为

branch	*branch-name*	*branch-city*	*assets*
	Perryridge		U.10000000

属性 *branch-city* 的位置为空暗示该属性值无需更新。

上述查询只是将 Perryridge 分支机构的总资产更新为 $ 10 000 000，而不管旧值是什么。有时候更新操作可能会用到更新前的值。必须用两行来表达这种请求：一行指明需要更新的旧元组，另一行说明更新后的待插入新元组。

假设要计算利息，且所有余额都增加 5%，可写为

account	*branch-name*	*account-number*	*balance*
U.			_x * 1.05
			_x

这个查询表明一次从关系 *account* 中检索出一个元组，取得其余额 x，然后把余额更新为 $x * 1.05$。

5.2　Quel

Quel 是作为 Ingres 数据库系统的查询语言引入的，由加利福尼亚大学伯克利分校研制。Quel 的基本结构十分类似于元组关系演算。表达大多数 Quel 查询只用三种子句：range of、retrieve 和 where。

- 每个元组变量都在一个 range of 子句中声明，写为：

$$\textbf{range of } t \textbf{ is } r$$

表明 t 是取自关系 r 中元组值的一个元组变量。

- retrieve 子句的功能类似于 SQL 中的 select 子句。
- where 子句包含选择谓词。

一个典型的 Quel 查询形式如下：

$$\begin{aligned}
&\textbf{range of } t_1 \textbf{ is } r_1\\
&\textbf{range of } t_2 \textbf{ is } r_2\\
&\quad\vdots\\
&\textbf{range of } t_m \textbf{ is } r_m\\
&\textbf{retrieve } (t_{i_1}.A_{j_1},\ t_{i_2}.A_{j_2},\ \cdots,\ t_{i_n}.A_{j_n})\\
&\textbf{where } P
\end{aligned}$$

其中 t_i 是一个元组变量，r_i 是一个关系，而 A_{j_k} 是一个属性。Quel 和 SQL 一样，用

$$t.A$$

表示元组变量 t 在属性 A 上的值。这个表达式的含义和元组关系演算中 $t[A]$ 一样。

Quel 不支持 intersect、union 和 minus 这样的关系代数运算，而且 Quel（与 SQL 不同）不支持嵌套子查询，即不能在一个 where 子句中嵌套 retrieve-where 子句。这些限制并不会减弱 Quel 的表达能力，尽管有时写一个查询会复杂一些。

5.2.1　简单查询

回到银行的例子，用 Quel 写一些以前的查询。首先，找出所有在银行贷款的客户的名字：

$$\begin{aligned}
&\textbf{range of } t \textbf{ is } borrower\\
&\textbf{retrieve } (t.customer\text{-}name)
\end{aligned}$$

上述查询并不会删除重复元组，因此，那些在银行有数笔贷款的客户的名字会出现多次。为了删除重复元组，需要在 retrieve 子句中加上关键词 unique：

$$\begin{aligned}
&\textbf{range of } t \textbf{ is } borrower\\
&\textbf{retrieve unique } (t.customer\text{-}name)
\end{aligned}$$

虽然删除重复元组会增加查询处理的开销，但必要时还是要指明删除重复元组。

让我们考虑一个更复杂的例子，找出 Perryridge 分支机构中贷款数额超过 $700 的所有贷款号。在 Quel 中查询如下写出：

range of t **is** *loan*
retrieve (t.*loan-number*)
where t.*branch-name* = "Perryridge" **and** t.*amount* > 700

考虑用 Quel 对两个以上关系进行查询的例子："找出所有在 Perryridge 分支机构贷款的客户名字"：

range of t **is** *borrower*
range of s **is** *loan*
retrieve unique (t.*customer-name*)
where s.*branch-name* = "Perryridge" **and**
　　t.*loan-number* = s.*loan-number*

注意，Quel 像 SQL 一样使用逻辑连接符 and、or 和 not，而不是像元组关系演算那样使用符号 ∧、∨ 和 ¬。我们需要显式地表达连接谓词。对自然连接，Quel 没有特殊的记号。像 SQL 一样，它包括模式匹配的比较，但在这里我们不打算讨论。

再看另外一个涉及两个关系的例子，考虑查询"找出银行中既有帐户又有贷款的所有客户的名字"：

range of s **is** *borrower*
range of t **is** *depositor*
retrieve unique (s.*customer-name*)
where t.*customer-name* = s.*customer-name*

在 SQL 中，可以选择使用关系代数运算符 intersect 来表达这个查询，正如以前提到的，Quel 不包括这种操作。

5.2.2　元组变量

在某些查询中，需要声明同一个关系的两个不同的元组变量。考虑查询"找出所有和 Jones 住在同一个城市的客户的名字"。如下写为：

range of s **is** *customer*
range of t **is** *customer*
retrieve unique (s.*customer-name*)
where t.*customer-name* = "Jones" **and**
　　s.*customer-city* = t.*customer-city*

由于上述查询要求把每个客户元组和客户名是 Jones 的元组进行比较，因此需要关系 *customer* 的两个不同元组变量。然而，一个查询在一个关系上通常只声明一个元组变量。在这种情况

下，可以省略 range of 语句，并把关系名本身当作已隐含声明的元组变量来使用。遵照这一惯例，重写查询“找出所有在银行既有贷款又有帐户的客户的名字”如下：

retrieve unique (*borrower . customer-name*)
where *depositor . customer-name* = *borrower . customer-name*

最初学术上的 Quel 并不允许使用隐含声明的元组变量。

5.2.3 聚集函数

Quel 中的聚集函数在元组的分组上计算。然而，它们的形式与 SQL 并不相同。在 SQL 中，group by 子句是查询语句的一部分，因此同一查询中的所有聚集函数采用相同的元组划分。在 Quel 中，元组划分是作为聚集表达式的一部分来说明的。

Quel 的聚集表达式可以是如下形式：

aggregate function (*t . A*)
aggregate function (*t . A* **where** *P*)
aggregate function (*t . A* **by** $s.B_1, s.B_2, \cdots, s.B_n$ **where** *P*)

其中

- 聚集函数可以为 count、sum、avg、max、min、countu、sumu、avgu 或 any。
- *t* 和 *s* 是元组变量。
- *A*，B_1，B_2，…，B_n 是属性。
- *P* 是和 retrieve 中 where 子句类似的谓词。

聚集表达式可以出现在常量可以出现的任何地方。count、sum、avg、max 和 min 这些函数的含义很直观，我们将在本节的后面部分解释 any 的作用。函数 countu、sumu 和 avgu 分别等同于 count、sum 和 avg，不同的是前者将从运算对象中删除重复元组。

要找出 Perryridge 分支机构所有帐户的平均帐户余额，写为

range of *t* **is** *account*
retrieve avg (*t . balance*
　　where *t . branch-name* = “Perryridge”)

聚集函数可以出现在 where 子句中。如果想找出结算高于银行平均结算的所有帐户，可写为：

range of *u* **is** *account*
range of *t* **is** *account*
retrieve (*t . account-number*)
where *t . balance* > **avg** (*u . balance*)

其中 avg (…) 表达式计算出银行所有帐户的平均余额。

让我们对前面的例子进行修改。我们将只考虑 Perryridge 分支机构，而不是所有分支机构的平均余额。这样，如果要找出的是余额比 Perryridge 分支机构平均余额高的所有帐户，这个查询可写为如下：

```
range of u is account
range of t is account
retrieve (t.account-number)
where t.balance > avg (u.balance
                       where u.branch-name = "Perryridge")
```

再做进一步的修改，现在想找出帐户余额比其所在银行平均余额高的所有帐户号码，这时，需要对关系 *account* 中的每个元组 t 计算 $t.branch\text{-}name$ 中银行的平均余额。为了形成这样的元组分组，需要在聚集表达式中使用 by 结构：

```
range of u is account
range of t is account
retrieve (t.account-number)
where t.balance > avg (u.balance by t.branch-name
                       where u.branch-name = t.branch-name)
```

Quel 中 by 结构的效果与 SQL 中 group by 子句不同，区别主要源于元组变量的作用。by 中使用的元组变量 t 和该查询其余地方出现的 t 是同一个，而聚集表达式中出现的其他所有元组变量都是聚集表达式的局部变量，即使它们与查询的其他语句中出现的变量同名。因此在上述 Quel 查询语句中，如果从表达式中去掉“by $t.branch\text{-}name$”，元组变量 t 将变成聚集表达式的局部变量，不再和外层查询中的元组变量 t 相对应。

考虑查询“找出在银行中有帐户而没有贷款的所有客户的名字”。在关系代数中，用集合差运算写这个查询，在 Quel 中可以用 count 聚集函数来写这个查询，因为这个查询也可以理解为“找出在银行中有帐户，而从银行贷款数量为 0 的所有客户的名字”。

```
range of t is depositor
range of u is borrower
retrieve unique (t.customer-name)
where count (u.loan-number by t.customer-name
             where u.customer-name = t.customer-name) = 0
```

Quel 还提供了一个叫 any 的聚集函数，同样可以运用到这个例子上。如果用 any 代替 count，那么当总数比 0 大时结果为 1，否则结果为 0。函数 any 可以加快这个查询的执行，因为只要发现一个符合查询条件的元组，处理就可以停止。使用 any 作比较类似于关系演算中的“存在”量词，这样，可以重写上例为：

```
range of t is depositor
range of u is borrower
retrieve unique (t.customer-name)
where any (u.loan-number by t.customer-name
           where u.customer-name = t.customer-name) = 0
```

举一个更复杂的例子，考虑查询“找出在 Brooklyn 各分支机构都有帐户的所有客户的名

字"。用 Quel 表达这个查询的策略如下：首先找出 Brooklyn 共有多少分支机构，然后将这个数字分别与每个客户有帐户的、位于 Brooklyn 的分支机构数目相比较。前面所使用的 count 函数会计算重复元组，因此，在这里将使用 countu 函数，它不计算重复元组。

```
range of t is depositor
range of u is account
range of s is branch
range of w is branch
retrieve unique (t.customer-name)
where countu (s.branch-name by t.customer-name
              where u.account-number = t.account-number
              and u.branch-name = s.branch-name
              and s.branch-city = "Brooklyn") =
      countu (w.branch-name where w.branch-city = "Brooklyn")
```

因为一次只能考虑一个客户，所以在第一个 countu 表达式中使用了 by。然而，在后面的 countu 表达式中没有使用 by，因为我们关心的是在 Brooklyn 共有多少分支机构，不必使 countu 表达式内外的元组变量相对应。

5.2.4　数据库的修改

在 Quel 中对数据库的修改与在 SQL 中类似，尽管在语法上略有不同。

1. 删除

在 Quel 中删除的格式如下：

```
range of t is r
delete t
where P
```

元组变量 t 可以隐含定义。谓词 P 可以是任何有效的 Quel 谓词。如果 where 子句省略，那么关系中的所有元组都将被删除。

下面是一些 Quel 删除请求的例子：

- 删除关系 *loan* 中的所有元组

```
range of t is loan
delete t
```

- 删除 Smith 的所有帐户记录

```
range of t is depositor
delete t
where t.customer-name = "Smith"
```

- 删除位于 Needham 的那些分支机构的所有帐户

```
range of t is account
range of u is branch
delete t
where t.branch-name = u.branch-name
      and u.branch-city = "Needham"
```

2. 插入

在 Quel 中的插入有两种一般的形式：插入一个单一的元组和插入元组的集合。Quel 使用关键词 append 表示插入，下面是一些 Quel 插入请求的例子：

• 插入具有 Perryridge 分支机构帐号为 A-9732、结算为 $ 700 这些信息的元组

```
append to account (branch-name = "Perryridge",
                   account-number = A-9732,
                   balance = 700)
```

• Perryridge 分支机构想为所有从该分支机构贷款的用户提供礼物，对应每笔贷款赠送一个余额为 $ 200 的新存款帐户，并将贷款号作为新存款帐户的帐户号

```
range of t is loan
range of s is borrower
append to account (branch-name = t.branch-name,
                   account-number = t.loan-number,
                   balance = 200)
where t.branch-name = "Perryridge"
append to depositor (customer-name = s.customer-name,
                     account-number = s.loan-number)
where t.branch-name = "Perryridge" and t.loan-number = s.loan-number
```

注意，需要在关系 *account* 和 *depositor* 中记录新帐户信息，因此要写两个 append 语句。

3. 更新

在 Quel 中用 replace 命令表达更新操作。下面是一些 Quel 更新请求的例子：

• 支付 5% 利息（所有帐户余额增加 5%）

```
range of t is account
replace t (balance = 1.05 * t.balance)
```

• 为那些存款额超过 $ 10 000 的帐户支付 6% 的利息，其余支付 5% 的利息

```
range of t is account
replace t (balance = 1.06 * balance)
where t.balance > 10000
replace t (balance = 1.05 * balance)
where t.balance ≤ 10000
```

要提醒的是，上述两个语句的顺序会影响到余额略低于 $10 000 的那些帐户所获得的利息。

5.2.5　集合操作

考虑在 SQL 中用 union 操作表示的一个例子“找出所有在银行有帐户或贷款或二者兼有的客户的名字”。由于 Quel 中并没有 union 操作，并且我们知道 Quel 是基于元组关系演算的，因而也许能从该查询的下列元组关系演算表达式得到启发：

$$\{t \mid \exists\ s \in borrower\ (t\ [customer\text{-}name] = s\ [customer\text{-}name])$$
$$\vee \exists\ u \in depositor\ (t\ [customer\text{-}name] = u\ [customer\text{-}name])\}$$

然而我们并不能根据上述表达式来写 Quel 查询。问题在于，在元组关系演算查询中，我们既从元组变量 s（范围为关系 *borrower*），又从元组变量 u（范围为关系 *depositor*）得到客户。在 Quel 中，retrieve 子句必须为以下形式之一：

- retrieve s . *customer-name*。
- retrieve u . *customer-name*。

如果用第一种形式，就排除了没有贷款的存款者；如果采用后者，就排除了没有存款的贷款者。

要在 Quel 中写出这个查询，必须创建一个新的关系，并且往这个新的关系中插入元组。让我们把这个新关系叫作 *temp*。可以这样得到银行的所有存款者：

range of u **is** *depositor*
retrieve into *temp* **unique** (u . *customer-name*)

这个 into *temp* 子句将产生一个新的关系 *temp* 来存放查询的结果。现在可以找出银行的所有贷款者，并把它们插入到新建的关系 *temp* 中。用 append 命令可以做到这点：

range of s **is** *borrower*
append to *temp* **unique** (s . *customer-name*)

现在已有了一个关系 *temp*，其中存放的是有存款或有贷款或二者兼有的所有客户。在 retrieve 和 append 请求中使用关键词 unique 可以保证重复的元组已被删除。

关系 *temp* 中有想要的客户名单，现在可以用下面的语句来完成查询：

range of t **is** *temp*
retrieve unique (t . *customer-name*)

使用 append 策略允许在 Quel 中实现并操作。要实现集合的差操作 $r - s$（SQL 中的 minus），先创建一个代表 r 的临时关系，然后从这个关系中删除那些也属于 s 的元组。为了说明这种策略，来考虑查询“找出所有那些在银行有存款但没有贷款的客户的名字”。首先创建一个临时关系：

range of u **is** *depositor*
retrieve into *temp* (u . *customer-name*)

这时，*temp* 中包含那些在银行中有存款的客户，其中也可能包括那些从银行贷款的客户。现在删除那些在银行贷款的客户。

range of s **is** *borrower*
range of t **is** *temp*
delete t
where t.*customer-name* = s.*customer-name*

关系 *temp* 中现在包含我们想要的客户名单，因而可以用下面的语句来完成查询：

range of t **is** *temp*
retrieve unique (t.*customer-name*)

5.2.6 Quel 和元组关系演算

为了更清楚地理解 Quel 和元组关系演算的联系，考虑下面的 Quel 查询：

range of t_1 **is** r_1
range of t_2 **is** r_2
.
.
.
range of t_m **is** r_m
retrieve uniqne ($t_{i_1}.A_{j_1}$, $t_{i_2}.A_{j_2}$, …, $t_{i_n}.A_{j_n}$)
where P

上面的 Quel 查询用元组关系演算可以表达为：

$$\{t \mid \exists\ t_1 \in r_1, t_2 \in r_2, \cdots, t_m \in r_m\ (t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}] \wedge t[r_{i_2}.A_{j_2}] = t_{i_2}[A_{j_2}] \wedge \cdots \wedge t[r_{i_n}.A_{j_n}] = t_{i_n}[A_{j_n}] \wedge P(t_1, t_2, \cdots, t_m))\}$$

为理解这个表达式，可以把“存在”后面的公式看作三个部分：

- $t_1 \in r_1 \wedge t_2 \in r_2 \wedge \cdots \wedge t_m \in r_m$。这个部分限制 t_1, t_2, …, t_m 只能在各自的关系范围内取值。
- $t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}] \wedge t_{i_2}[A_{j_2}] = t[r_{i_2}.A_{j_2}] \wedge \cdots \wedge t[r_{i_n}.A_{j_n}] = t_{i_n}[A_{j_n}]$。

这一部分对应于 Quel 查询中的 retrieve 子句。要保证元组 t 的第 k 个属性对应于 retrieve 子句中的第 k 项。考虑第一项：$t_{i_1}.A_{j_1}$。这一项是 r_{i_1} 的某个元组（因为 t_{i_1} 的范围是 r_{i_1}）在属性 A_{j_1} 上的值。因此，需要 $t[A_{j_1}] = t_{i_1}[A_{j_1}]$。考虑到有些属性名可能在几个关系中都出现，我们使用了麻烦一点的写法 $t[r_{i_1}.A_{j_1}] = t_{i_1}[A_{j_1}]$。

- $P(t_1, t_2, \cdots, t_m)$。这个部分表述了 t_1, t_2, …, t_m 可接受的值的限制，在 Quel 查

询中由 where 子句给出。

上述 Quel 查询中，对元组关系演算中的“对所有的”和“不存在”没有相应的表示。Quel 通过使用 any 聚集函数和在临时关系中插入、删除来获得关系代数的表达能力。

5.3 Datalog

Datalog 是基于逻辑编程语言 Prolog 上的非过程化的查询语言。与在关系演算中一样，用户描述所需信息，但无需给出获取信息的具体过程。Datalog 语法类似于 Prolog 语法。然而，Datalog 程序的含义用纯声明性的方式定义，不像 Prolog 那样有更多的过程化语义，所以 Datalog 简化了简单查询的书写，使查询优化更容易进行。

5.3.1 基本结构

Datalog 程序由一组规则构成。在给出 Datalog 规则的形式化定义和含义以前，先让我们看一些例子。下面的例子是用 Datalog 规则定义视图关系 *vl*，该视图包含了 Perryridge 分支机构中余额大于 $ 700 的所有帐户的帐户号和余额：

$$vl\ (A,\ B)\ :-\ account\ (\text{“Perryridge”},\ A,\ B),\ B>700$$

Datalog 规则用来定义视图。前面的规则使用了关系 *account*，定义了视图关系 *vl*。符号 :- 读作“如果”，把“*account* (“Perridge”, *A*, *B*)”和“$B>700$”分开的逗号读作“并且”。这个规则可以直观地理解为：

for all A, B
if (“Perryridge”, A, B) $\in$ *account* **and** $B > 700$
then $(A, B) \in vl$

假设关系 *account* 如图 5-2，那么，视图关系 *vl* 包含的元组如图 5-3 所示。

branch-name	*account-number*	*balance*
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Perryridge	A-201	900
Redwood	A-222	700
Perryridge	A-217	750

图 5-2　关系 *account*

account-number	*balance*
A-201	900
A-217	750

图 5-3　关系 *vl*

要检索视图关系 *vl* 中帐户 A-217 的余额，可以写出以下查询

$$?\ vl\ (\text{“A-217”},\ B)$$

查询结果是

$$(\text{A-217},\ 750)$$

要找出关系 *vl* 中余额大于 $ 800 的所有帐户的帐户号与余额，可以写为

$$? vl\ (A,\ B),\ B>800$$

查询结果是

$$(\text{A-201},\ 900)$$

一般情况下，定义一个视图关系可能需要不止一条规则，每一条规则定义视图关系必须包含的一组元组。视图关系中的元组集合将是所有规则所定义的元组集合的并。下面的 Datalog 程序定义了存款的利率，这个程序用两条规则定义了一个视图关系 *interest-rate*，属性为帐号和利率。规则表明，如果余额低于 $2000，那么利率为 0；如果余额大于或等于 $2000，利率为 5%。

$$\begin{aligned} &interest\text{-}rate\ (A,\ 0)\ :-\ account\ (N,\ A,\ B),\ B < 2000\\ &interest\text{-}rate\ (A,\ 5)\ :-\ account\ (N,\ A,\ B),\ B >= 2000 \end{aligned}$$

Datalog 规则也可以使用否定。下面的规则定义了一个视图关系 c，其中包含在银行有存款但没有贷款的所有客户的姓名：

$$\begin{aligned} &c\ (N)\ :-\ depositor\ (N,\ A),\ \textbf{not}\ is\text{-}borrower\ (N)\\ &is\text{-}borrower\ (N)\ :-\ borrower\ (N,\ L), \end{aligned}$$

在 Prolog 和大多数 Datalog 的实现中，关系的属性通过位置来识别，而省略了属性名。所以和 SQL 查询相比，Datalog 的规则更紧凑。但是，如果关系有很多属性，或者关系属性的顺序或个数可能改变，用位置来识别属性的记法会很麻烦，且易出错。创造 Datalog 语法的另一种形式，即用名字属性来代替位置属性并不困难。在这种系统中，定义 *vl* 的 Datalog 规则可以写为：

$$\begin{aligned} &vl\ (account\text{-}number\ A,\ balance\ B)\ :-\\ &\qquad account\ (branch\text{-}name\ \text{“Perryridge”},\ account\text{-}number\ A,\ balance\ B),\\ &\qquad B > 700 \end{aligned}$$

如果给出关系模式，在上述两种格式之间转换无需太多周折。

5.3.2 Datalog 规则语法

我们已经非形式化地介绍了规则和查询，现在来形式化地定义其语法，此外在下节讨论规则的含义。我们将使用和关系代数中一样的习惯记法来表示关系名、属性名、常量（如数字和字符串）和变量名。常量如数字 4、字符串 “John”；X 和 *Name* 是变量。一个正的文字量格式如下：

$$p\ (t_1,\ t_2,\ \cdots,\ t_n)$$

p 是关系名，有 n 个属性，t_1，t_2，…，t_n 或者是常数，或者是变量。一个负的文字量格式如下：

$$\textbf{not}\ p\ (\ t_1,\ t_2,\ \cdots,\ t_n)$$

关系 P 有 n 个属性。下面是文字量的一个例子：

$$account\ (\ \text{“Perryridge”},\ A,\ B)$$

包含算术运算的文字量将被特殊对待。例如，文字量 $B>700$，虽然不符合前面描述的语法，但可以从概念上认为其代表 $>\ (B,\ 700)$，这种形式符合语法，且其中 $>$ 是一个关系。

但这种写法对“$>$”这样的算术运算意味着什么呢？关系 $>$（概念上）包含形如 $(x,\ y)$ 的元组，只要 $x>y$，任何 $(x,\ y)$ 都可作为一个元组。因此，(2，1) 和 (5，−33) 都是 $>$ 中的元组。显然，(概念上的) 关系 $>$ 是无穷的。其他的算术运算（例如 $>$、$=$、$+$ 或 $-$）在概念上都被认为是关系。例如，$A=B+C$ 概念上代表 $+\ (B,\ C,\ A)$，其中 $+$ 关系包含所有使 $z=x+y$ 的元组 $(x,\ y,\ z)$。

一个事实形式如下：

$$p\ (\ v_1,\ v_2,\ \cdots,\ v_n)$$

表示元组 $(v_1,\ v_2,\ \cdots,\ v_n)$ 在关系 P 中。关系的一组事实也可以写成通常的表格形式。数据库模式中所有关系的事实集合和数据库模式的一个实例等价。规则建立在文字量基础上，形式为

$$p\ (\ t_1,\ t_2,\ \cdots,\ t_n):-\ L_1,\ L_2,\ \cdots,\ L_n$$

其中 L_i 是一个（正的或负的）文字量。文字量 $p\ (\ t_1,\ t_2,\ \cdots,\ t_n)$ 称作规则的头部，规则中其余的文字量构成规则的主体。

一个 Datalog 程序由一组规则构成，规则之间的顺序无关紧要。正如前面所提到的，可以有好几个规则定义一个关系。

图 5-4 中的例子是一个相对较复杂的 Datalog 程序，它定义了 Perryridge 分支机构各帐户的利息。程序的第一条规则定义了一个视图关系 *interest*，其属性为帐号和该帐户的利息。此规则中使用了视图关系 *interest-rate* 和 *perryridge-account*。第二条规则定义了视图关系 *perryridge-account*，其中使用了数据库关系 *account*。程序的最后两条规则前面已经见过。

```
interest(A, I) :- perryridge-account(A, B),
                  interest-rate(A, R), I = B * R/100.
perryridge-account(A, B) :- account("Perryridge", A, B).
interest-rate(A, 0) :- account(N, A, B). B < 2000.
interest-rate(A, 5) :- account(N, A, B). B >= 2000.
```

图 5-4　定义 Perryridge 帐户利息的 Datalog 程序

3.7.3 节定义了什么时候一个视图是依赖于另一个视图的，什么时候一个视图是递归的。虽然 Datalog 中视图定义的语法与 3.7.3 节中视图定义的语法不同，依赖视图和递归视图的定义仍可用于 Datalog 视图中。例如，在图 5-4 所示程序中，关系 *interest*（直接）依赖于关系 *interest-rate* 和 *perryridge-account*。关系 *interest-rate* 和 *perryridge-account* 又直接依赖于 *account*。因为从 *interest* 到 *interest-rate* 到 *account* 之间有一条依赖链，所以关系 *interest*（间接）依赖于

account。

图 5-4 的程序是非递归的。然而，图 5-5 中的程序是递归的，因为（由于第二条规则）*empl* 依赖于自身。

```
empl(X, Y) :- manager(X, Y).
empl(X, Y) :- manager(X, Z), empl(Z, Y).
```

图 5-5　递归的 Datalog 程序

5.3.3　非递归 Datalog 语义

让我们来考虑 Datalog 程序的形式语义。目前只考虑非递归的程序，递归程序的语义相对较复杂，将在 5.3.6 节讨论。我们总是从一个简单规则开始定义一个程序的语义。

1. 规则的语义

规则在某一场合的实例化是指用一些常量代替规则中出现的各个变量。如果一个变量在一个规则中出现多次，那么该变量出现的所有地方都应用同一个常量取代。某一场合的实例化通常可以简称为实例化。

作为例子定义 *vl* 的规则以及对该规则的一个实例化如下所示：

vl（*A*，*B*）：－*account*（“Perryridge”，*A*，*B*），*B* > 700

vl（“A-217”，750）：－*account*（“Perryridge”，“A-217”，750），750 > 700

在这里，变量 *A* 被“A-217”代替，变量 *B* 用 750 代替。

规则通常有很多可能的实例，这些实例对应着给规则中各变量赋值的不同方法。

假设有规则 *R*：

$$p\ (t_1,\ t_2,\ \cdots,\ t_n):-\ L_1,\ L_2,\ \cdots,\ L_n$$

和该规则中所用关系的事实集合 *I*（*I* 也可以视为是一个数据库实例）。考虑规则 *R* 的任何一个实例 *R′*：

$$p\ (v_1,\ v_2,\ \cdots,\ v_n):-\ l_1,\ l_2,\ \cdots,\ l_n$$

其中文字量 l_i 或者形如 $q_i\ (v_{i,1},\ v_{i,2},\ \cdots,\ v_{i,n_i})$，或者形如 not $q_i\ (v_{i,1},\ v_{i,2},\ \cdots,\ v_{i,n_i})$，且每个 v_i 和 $v_{i,j}$ 是一个常量。

如果下面条件成立，就说规则实例 *R′* 的主体在 *I* 中被满足：

1）对 *R′* 中每一个正的文字量 $q_i\ (v_{i,1},\ v_{i,2},\ \cdots,\ v_{i,n_i})$，集合 *I* 包含事实 $q\ (v_{i,1},\ v_{i,2},\ \cdots,\ v_{i,n_i})$。

2）对 *R′* 中每一个负的文字量 not $q_j\ (v_{j,1},\ v_{j,2},\ \cdots,\ v_{j,n_j})$，集合 *I* 不包含事实 $q_j\ (v_{j,1},\ v_{j,2},\ \cdots,\ v_{j,n_j})$。

定义根据规则 *R* 从给定事实集合 *I* 能推导出的事实集合为：

$$\begin{aligned} infer\ (R,\ I) = \{&p\ (t_1,\ \cdots,\ t_{n_i})\ |\text{存在 } R \text{ 的一个实例 } R',\\ &\text{使 } p\ (t_1,\ \cdots,\ t_{n_i}) \text{ 是 } R' \text{的头部，而 } R'\\ &\text{的主体在 } I \text{ 中被满足}\}\ . \end{aligned}$$

对于给定的规则的集合$\mathcal{R}=\{R_1, R_2, \cdots, R_n\}$，定义

$$infer(\mathcal{R}, I) = infer(R_1, I) \cup infer(R_2, I) \cup \cdots \cup infer(R_n, I)$$

假设已知事实的集合 I，其中包含图 5-2 所示的关系 *account* 中的元组。我们用来作为例子的规则 R 的一个可能的实例如下：

$$vl(\text{“A-217”}, 750) : -account(\text{“Perryridge”}, \text{“A-217”}, 750), 750 > 700.$$

事实 *account*（“Perryridge”，“A-217”，750）属于事实集合 I。而且，750 大于 700，故概念上（750，700）属于关系“>”，因此，这个规则实例的主体在 I 中被满足。可能还有 R 的其他实例，使用它们可以发现 *infer*（R，I）包含的正好是 *vl* 的事实集合，如图 5-6 所示。

account-number	*balance*
A-201	900
A-217	750

图 5-6 *infer*（R，I）的结果

2. 程序的语义

当一个视图关系用另一视图关系来定义时，前者的事实集合依赖于后者的事实集合。已经假设本节中的定义都是非递归的，即没有视图关系（直接或间接）依赖于自己。因此，可以对视图关系分层如下，并用这种层次来定义程序的语义：

- 一个关系是第一层的，如果定义它的规则的主体使用的关系都贮存在数据库中。
- 一个关系是第二层的，如果定义它的规则的主体使用的关系或是贮存在数据库中，或是第一层的。
- 一般情况下，一个关系 p 是第 $i+1$ 层的，如果 1）它不是第 1，2，…，i 层的；2）定义它的规则的主体使用的关系或是贮存在数据库中，或是在第 1，2，…，i 层的。

考虑图 5-4 的程序。这个程序中视图关系的层次如图 5-7 所示。关系 *account* 是存在数据库中的。关系 *interest-rate* 是第一层的，因为定义它的两条规则使用的关系都是在数据库中的。关系 *perryridge-account* 也是第一层的。最后，关系 *interest* 是第二层的，因为它不在第一层，并且定义它的规则使用的关系或是在数据库中或是在第二层以下。

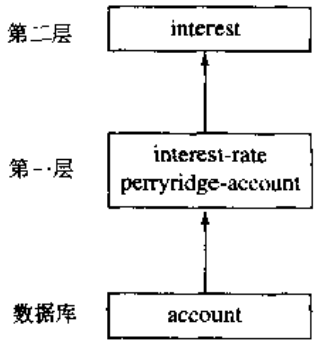


图 5-7 视图关系的分层

现在可以用视图关系的层次来定义 Datalog 程序的语义。假设一个给定程序的层次为 1，2，…，n，设 $\mathcal{R}_i$ 代表定义第 i 层上的视图关系的所有规则的集合。

- 定义 I_0 是数据库中存储的事实集合，定义 I_1 为

$$I_1 = I_0 \cup infer(\mathcal{R}_1, I_0)$$

- 使用如下定义，可以按类似的方式继续进行下去，根据 I_1 和 $\mathcal{R}_2$ 定义 I_2 等等

$$I_{i+1} = I_i \cup infer(\mathcal{R}_{i+1}, I_i)$$

- 最后，由程序定义的视图关系的事实集合（也称为程序的语义）由与最高的第 n 层对应的事实集合 I_n 给出。

考虑图 5-4 的程序，I_0 是存于数据库中的事实集合，I_1 除了存于数据库中的事实集合外，还包含用定义 *interest-rate* 和 *perryridge-account* 的规则从 I_0 中可导出的事实。最后，I_2 包含 I_1 中事实，还包含关系 *interest* 的事实，这些事实是用定义 *interest* 的规则从 I_1 中事实导出的。该程序的语义，即所有视图关系的事实所组成的集合，被定义为事实集合 I_2。

3.7.3 节曾提到利用一种叫视图扩展的技术来定义非递归关系代数视图的含义。视图扩展也可用于非递归的 Datalog 视图中；反过来，这里描述的分层技术也可以用于关系代数视图。然而，分层技术可以用于扩展的 Datalog，而视图扩展却不能。

5.3.4 安全性

有可能写出的规则会产生无穷多的解。考虑如下规则：

$$gt\ (X,\ Y)\ :-\ X > Y$$

因为关系定义 > 是无穷的，这条规则会为关系 *gt* 产生无穷的事实，相应地，这一计算会耗费无穷的时间与空间。

否定的使用也可能导致类似的问题。考虑如下规则：

$$not\text{-}in\text{-}loan\ (B,\ L)\ :-\ \textbf{not}\ loan\ (B,\ L)$$

这个想法是如果元组（*branch*，*loan-number*）不是在关系 *loan* 中出现，则应属于视图关系 *not-in-loan* 中。然而，如果可能的分支机构名字或帐户号是无穷的，那么关系 *not-in-loan* 也可能是无穷的。

最后，如果在头部有一个变量不在主体出现，那也可能得到一个无穷的事实，其中该变量被实例化为不同的值。

为了避免上述可能性，Datalog 的规则必须满足以下安全性条件：

1）规则头部出现的每一个变量必须出现在规则主体中的一个非算术的正的文字量中。

2）规则主体中负文字量中出现的每个变量必须在主体的某个正的文字量中也出现过。

如果一个 Datalog 非递归程序的全部规则都满足前面的安全性条件，那么只要数据库关系是有限的，则该程序所定义的全部视图关系也可以证明是有限的。这些条件可弱化为某些情况下允许头部的变量只出现在主体的一个算术的文字量中。例如，在规则

$$p\ (A)\ :-\ q\ (B),\ A\ =\ B+1$$

中可以看出，如果关系 q 是有限的，根据加法的性质，那么 p 也是有限的，虽然变量 A 只是在一个算术的文字量中出现。

5.3.5 Datalog 中的关系运算

不使用算术运算的非递归的 Datalog 表达式在表达能力上和使用基本运算的关系代数等价（除了修改操作外）。我们不打算在这里给出形式化的证明，而是要通过例子来说明不同的关系代数运算如何用 Datalog 表达。在所有例子中，通过定义视图关系 *query* 来说明这些运算。

前面已经见过用 Datalog 规则如何进行选择。要实现投影，只需要在规则头部写出所需属性。要从 *account* 中投影出属性 *account-name*，使用：

$$query\ (A)\ :-\ account\ (N,\ A,\ B)$$

在 Datalog 中可以用下面方法获得关系 r_1 与 r_2 的笛卡儿积：

$$query(X_1, X_2, \cdots, X_n, Y_1, Y_2, \cdots, Y_m) :- r_1(X_1, X_2, \cdots, X_n), r_2(Y_1, Y_2, \cdots, Y_m)$$

其中 r_1 有 n 个分量，r_2 有 m 个分量，X_1、X_2、…、X_n、Y_1、Y_2、…、Y_m 都是不重复的变量名。

关系 r_1 与 r_2（都是 n 个分量）集合的并形式如下：

$$query\ (X_1,\ X_2,\ \cdots,\ X_n):-\ r_1\ (X_1,\ X_2,\ \cdots,\ X_n)$$
$$query\ (X_1,\ X_2,\ \cdots,\ X_n):-\ r_2\ (X_1,\ X_2,\ \cdots,\ X_n)$$

最后，关系 r_1 与 r_2 集合的差形式如下：

$$query(X_1, X_2, \cdots, X_n):- r_1(X_1, X_2, \cdots, X_n), \textbf{not}\ r_2(X_1, X_2, \cdots, X_n)$$

可以证明，能用关系代数运算表达任何不含算术运算的非递归 Datalog 查询。我们把这一证明作为练习留给读者，读者由此可以建立起关系代数的基本运算与不含算术运算的非递归 Datalog 的等价性。

Datalog 的某些扩展支持插入、删除和更新这些扩展关系更新操作。不同的实现中这些操作的语法各不相同。有些系统允许在规则头部使用 + 或 - 来表示插入或删除。例如：

$$+\ account\ (\text{“Johnstown”},\ A,\ B):-\ account\ (\text{“Perryridge”},\ A,\ B)$$

把 Perryridge 的所有帐户都插入到 Johnstown 的帐户集合中。在 Datalog 的某些实现中支持扩展的关系代数的聚集操作。同样，这种操作也没有标准语法。

5.3.6 Datalog 中的递归

有些数据库应用程序需要处理类似树型数据结构的结构。例如，考虑某一机构中的雇员，有些雇员是经理，每个经理管理一批人，这些人向他或她汇报。但这批人中又有人可能是经理，因此他们又有向他们汇报的人。因而这些雇员可以组织成类似于树的结构。

假设有关系模式：

$$Manager\text{-}schema\ =\ (employee\text{-}name,\ manager\text{-}name)$$

设 *manager* 是在该模式上的一个关系。

假设现在想找出直接或间接被某个经理（例如 Jones）管理的雇员。因此，假如 Alon 的经理是 Barinsky，Barinsky 的经理是 Estovar，Estovar 的经理是 Jones，那么 Alon、Barinsky 和 Estovar 都是受 Jones 控制的雇员。人们经常用递归程序来处理树型数据结构。使用递归的技术，可以定义受 Jones 控制的雇员集合如下：受 Jones 监管的雇员 1）他们的经理是 Jones；2）他们的经理受 Jones 监管。注意，第二个条件是递归的。

可以根据上面的递归定义写出递归 Datalog 视图 *empl-jones*，如下所示：

$$empl\text{-}jones\ (X):-\ manager\ (X,\ \text{“Jones”})$$
$$empl\text{-}jones\ (X):-\ manager\ (X,\ Y),\ empl\text{-}jones\ (Y)$$

第一条规则对应于1)；第二条规则对应于2)。视图 *empl-jones* 由于第二条规则依赖于自己，因

此，上述 Datalog 程序是递归的。我们假设递归的 Datalog 程序不包含有负文字量的规则，后面会解释原因。文献注解给出了关于什么情况下可以在递归的 Datalog 程序中使用负文字量的文献。

规则集合$\mathcal{R}$组成的递归程序中的视图关系恰好包含图 5-8 所示的由迭代过程 Datalog-Fixpoint 所计算出的事实集合 I。Datalog 程序中的递归在过程中被转换成了迭代。在过程的结尾，$infer\ (\mathcal{R},\ I) = I$，$I$ 被称为程序的一个不动点。

```
procedure Datalog-Fixpoint
    I = 数据库中的事实集
    repeat
        Old_I = I
        I = I ∪ infer(R, I)
    until I = Old_I
```

图 5-8　过程 Datalog-Fixpoint

考虑图 5-9 的程序，其中用关系 *manager* 定义了 *empl-jones*。图 5-10 是用迭代计算视图关系 *empl-jones* 后的事实集合。每次迭代都多计算一级 Jones 的下级雇员，并将其添加到集合 *empl-jones* 中。这个过程在集合 *empl-jones* 不再变化时终止，系统通过检查 $I = Old\_I$ 来确定集合 *empl-jones* 是否变化。这样的终点肯定会达到，因为经理与雇员的集合都是有限的。在给出的关系 *manager* 上，过程 Datalog-Fixpoint 在第四次迭代时停止，因为这时它发现不能再推导出新的事实。

应该验证一下，在迭代的终点，视图关系 *empl-jones* 包含的是否正好是那些在 Jones 领导下的雇员。要根据视图打印出 Jones 领导的雇员的名单，可以使用下面的查询：

employee-name	*manager-name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

图 5-9　关系 *manager*

迭代次数	*empl-jones* 中的元组
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

图 5-10　过程 Datalog-Fixpoint 迭代后 Jones 的雇员

$$?\ empl\text{-}jones\ (N)$$

为过程 Datalog-Fixpoint，我们应该记得规则可用来从一个给定的事实集合推导出新的事实。迭代一开始使用的事实集合 I_0 设为数据库中的事实。这些事实已知是真的，但还有其他

事实也可能为真[⊖]。下一步，假若 I_0 中的事实为真，根据 Datalog 程序中的规则集$\mathcal{R}$就能推导哪些事实为真。可以证明，只要程序的规则中不包含负文字量，任何在 $infer\ (\mathcal{R},\ I_0)$ 中出现的事实也会属于 $infer\ (\mathcal{R},\ J)$，其中 J 是任意的包含 I_0 的事实集合。根据这个论断，可以推出 $infer\ (\mathcal{R},\ I_0)$ 中每个事实为真，而不管以后还可以推出哪些事实也为真。

令 $I_1 = I_0 \cup infer\ (\mathcal{R},\ I_0)$，根据前面的观点，所有 I_1 中的事实已知为真。现在，I_1 对应于过程 *fixed-point* 第一次迭代结束时 I 的值。因此，过程 *fixed-point* 第一次迭代结束时所推导出的所有事实都为真。

既然 I_1 中的事实都为真，可以推导出 $I_2 = I_1 \cup infer\ (\mathcal{R},\ I_1)$ 中的所有事实也都为真。通常，设

$$I_{i+1} = I_i \cup infer\ (\mathcal{R},\ I_i)$$

I_i 的事实集合对应于过程 fixed-point 第 i 次迭代结束时 I 的值。每次迭代后都能推断出 I_i 中所有事实为真。而且，显然 $I_{i+1} \supseteq I_i$，那即是说已知为真的事实的集合在每次迭代后都在增加。

对于安全的 Datalog 程序，可以证明总有某个时候（或某个点）不能再推导出新的事实，即有某个 k，$I_{k+1} = I_k$。在这一点，我们将得到最后的为真的事实的集合。进一步说，给定一个 Datalog 程序和一个数据库，过程 fixed-point 能够推导出所有可以推导出的为真的事实的集合。

因为过程 Datalog-Fixpoint 只计算真的事实，只要程序中没有规则包含负的文字量，就可以认为此过程对这样的程序是保真的。进一步可以证明这个过程能推导出所有对给定数据库和 Datalog 程序为真的事实，所以，过程 Datalog-Fixpoint 可以认为是完备的。

不仅可为一个特定经理 Jones 创建其雇员视图，而可以使用以下程序（在图 5-5 中曾出现过）创建一个更一般的视图关系 *empl*，其中包含每一个表示 X 直接或间接受 Y 的管理的元组 $(X,\ Y)$：

$$empl\ (X,\ Y) :- manager\ (X,\ Y)$$
$$empl\ (X,\ Y) :- manager\ (X,\ Z),\ empl\ (Z,\ Y)$$

要找出 Jones 的直接或间接下属，只需使用查询

$$?\ empl\ (X,\ \text{“Jones”})$$

结果的元组集合将和视图 *empl-jones* 一样。大多数的 Datalog 实现都有很成熟的查询优化器和求值引擎，在运行上述查询时速度与执行视图 *empl-jones* 查询几乎一样。

前面定义的视图 *empl* 称为关系 *manager* 的传递闭包。如果关系 *manager* 被任何其他二元关系 R 代替，前面的程序将定义 R 的传递闭包。

5.3.7 递归的能力

允许递归的 Datalog 比不允许递归的 Datalog 有更强的表达能力。换句话说，有些对数据库的查询用递归可以做到，而不用递归则无法实现。例如，在Datalog中不用递归就无法表达传

⊖ “事实” 一词是技术意义上的用法，用来表示元组在关系中的成员资格。因此，根据 Datalog 中“事实”的意义，事实可能为真（即元组确实在关系中）或为假（即元组不在关系中）。

递闭包（不带递归的 SQL、QBE、Quel 也是如此）。考虑关系 *manager* 的传递闭包，直观的，固定数目的连接对任一经理而言只能找出另外一个固定层次数的下级级别中的雇员（我们在此并不打算予以证明）。由于任何给定的非递归的查询连接数目固定，因此该查询能够找出的雇员的级别数有一个限制。如果关系 *manager* 中雇员的级别数比查询的限制要大，这个查询就会丢掉一些级别的雇员。这样，一个非递归的 Datalog 程序不能表达传递闭包。

递归实现的另一种方法是使用一种外部的机制（例如嵌入式的 SQL）来对一个非递归的查询进行迭代。迭代事实上实现了图 5-8 中的不动点循环。实际上，这就是那些不支持递归的数据库系统实现这种查询的方法。然而，用迭代写这样的查询比用递归写要更复杂，而且可以对用递归写的程序进行优化，使其比用迭代写的程序运行更快。

递归使程序能力更强，但要小心使用。相对来说，写递归程序更易产生无穷多的事实，如下面程序所示：

$$number\ (0)$$
$$number\ (A)\ :-\ number\ (B),\ A\ =\ B\ +\ 1$$

这个程序对所有正整数 n 产生一个 $number\ (n)$，而这显然是无穷的，并且不会停止。程序的第二个规则不满足 5.3.4 节所述的安全性条件。在数据库关系是有限的前提下，满足安全条件的程序一定会停止，即使是递归的也不例外。对于这样的程序，视图关系中的元组只能包含数据库中的常量，因此视图关系一定也必定是有限的。反过来就不正确了，即存在不满足安全性条件但会停止的程序。

过程 Datalog-Fixpoint 迭代地使用函数 $infer\ (\mathcal{R},\ I)$ 来计算哪些事实为真。虽然只考虑不含负文字量的 Datalog 程序，但这个过程也可用于其他语言定义的视图上，如关系代数，只要视图满足下面描述的条件：不论使用何种语言定义视图 V，该视图都可以看成是由一个表达式 E_V 定义的，对一个事实集合 I，这一表达式返回事实集合 $E_V\ (I)$ 作为视图关系 V 的元组。给定一个视图定义集合 $\mathcal{R}$（不论何种语言），可以定义一个函数 $infer\ (\mathcal{R},\ I)$，返回值为 $I \cup \cup_{V \in \mathcal{R}} E_V\ (I)$。此函数形式上和 Datalog 中的 infer 函数形式是一样的。

视图 V 被称作单调的，如果对任何两个已知的事实集合 I_1 和 I_2，若 $I_1 \subseteq I_2$，则有 $E_V\ (I_1) \subseteq E_V\ (I_2)$，其中 E_V 是用于定义 V 的表达式。同样，函数 *infer* 被称为单调的，如果

$$I_1 \subseteq I_2 \Rightarrow infer\ (\mathcal{R},\ I_1) \subseteq infer\ (\mathcal{R},\ I_2)$$

因此，如果 *infer* 是单调的，给定一个全为真的事实的集合子集 I_0，可以确定 $infer\ (\mathcal{R},\ I_0)$ 中的所有事实也为真。使用 5.3.6 节同样的推理，可以证明只要 *infer* 函数是单调的，过程 Datalog-Fixpoint 就是保真的（即它只计算真的事实）。

只使用 Π、σ、$\times$、$\bowtie$、$\cup$、$\cap$ 或 ρ 这些运算符的关系代数表达式是单调的，递归视图可以用这些表达式定义。

然而，使用运算符 $-$ 的关系表达式不是单调的。例如，设 $manager_1$ 和 $manager_2$ 是关系模式同为 *manager* 的关系，设

$$I_1 = \{manager_1\ (\text{“Alon”}, \text{“Barinsky”}),\ manager_1\ (\text{“Barinsky”}, \text{“Estovar”}),\ manager_2\ (\text{“Alon”}, \text{“Barinsky”})\}$$
$$I_2 = \{manager_1\ (\text{“Alon”}, \text{“Barinsky”}),\ manager_1\ (\text{“Barinsky”}, \text{“Estovar”}),$$

$manager_2$（"Alon", "Barinsky"), $manager_2$（"Barinsky", "Estovar"）}

考虑表达式 $manager_1 \quad manager_2$。这一表达式作用在 I_1 上的结果是（"Barinsky", "Estovar"），而作用在 I_2 上得出一个空关系。但由于 $I_1 \subseteq I_2$，因此这个表达式不是单调的。使用扩展关系代数的分组运算的表达式也不是单调的。

不动点技术在用非单调表达式定义的递归视图上不能正确工作。然而，有实例可以证明这样的视图是有用的，尤其是在部分-子部分联系上定义聚集时。这样的联系定义其各部分由哪些子部分组成。子部分又可能由更小的部分组成，如此下去等等。因此，这样的联系像经理联系一样，有一个自然递归结构。文献注解提供了关于定义这种视图的研究的参考书。

定义某些递归查询而不使用视图是可能的。例如，已有人建议用扩展关系运算来定义传递闭包，同时用 SQL 语法的扩展版来定义广义传递闭包的建议也已被提出。然而，递归视图定义提供了比其他形式的递归查询更强的表达能力。

5.4 总结

我们已经学习了三种查询语言：QBE、Quel 和 Datalog。QBE 基于一种可视化范例，查询看起来很像表格。由于可视化范例因符合直觉而具有简单性，因此 QBE 和其变形在非专业的数据库用户中十分流行。Quel 基于元组关系演算，在许多方面与 SQL 类似，但避免了后者的许多复杂性。Datalog 是从 Prolog 派生出来的，但不像 Prolog，它有声明性的语义，使简单查询易于书写，查询求值易于优化。在 Datalog 中定义视图特别简单，并且 Datalog 所支持的递归视图使我们可以写出像传递闭包查询这样的不用递归或迭代就无法写出的查询。然而，Datalog 中很多重要特性如分组或聚集并没有广泛接受的统一标准，Datalog 仍主要作为一种研究性语言。

习题

5.1 考虑图 5-11 定义的保险公司数据库（首字母大写的是主码）。对这个关系数据库写出如下的 QBE 查询：

(a) 找出在 1989 年其车辆出过车祸的人员总数。

(b) 找出车祸中至少涉及 John Smith 的一辆车的车祸数量。

(c) 为数据库添加一个新的保单持有者。

(d) 删除 John Smith 的马自达车（Mazda）。

(e) 为 Williams 的丰田车加一条车祸记录。

person (*SS#*, *name*, *address*)
car (*License*, *year*, *model*)
accident (*Date*, *Driver*, *damage-amount*)
owns (*SS#*, *License*)
log (*License*, *Date*, *Driver*)

图 5-11 保险公司数据库

5.2 考虑图 5-12 的雇员数据库。为下面每个查询语句分别写出 QBE、Quel 和 Datalog 表达式：

(a) 找出所有为 First Bank Corporation 工作的雇员名字。

(b) 找出所有为 First Bank Corporation 工作的雇员名字和居住城市。

(c) 找出所有为 First Bank Corporation 工作且年薪超过 $ 10 000 的雇员名字、居住街道和城市。

(d) 找出数据库中居住城市和公司所在城市相同的所有雇员。

(e) 找出居住街道和城市与其经理相同的所有雇员。

(f) 找出数据库中所有不为 first Bank Corporation 工作的雇员。

(g) 找出所有工资高于 Small Bank Corporation 每一个雇员的雇员。

(h) 假设一个公司可以在好几个城市有分部。找出与 Small Bank Corporation 在同一城市的所有公司。

employee (*person-name*, *street*, *city*)
works (*person-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*person-name*, *manager-name*)

图 5-12　雇员数据库

5.3　考虑图 5-12 的关系数据库。为下列查询写出 QBE 和 Quel 表达式。

(a) 找出工资高于其所在公司雇员平均工资的所有雇员。

(b) 找出有最多雇员的公司。

(c) 找出工资总额最小的公司。

(d) 找出平均工资高于 First Bank Corporation 平均工资的所有公司。

5.4　考虑图 5-12 的关系数据库。为下列查询写出 QBE 和 Quel 表达式。

(a) 修改数据库使 Jones 居住在 Newtown 市。

(b) 为 First Bank Corporation 所有雇员增加 10% 的薪水。

(c) 为数据库中的所有经理增加 10% 的薪水。

(d) 为数据库中的所有经理增加 10% 的薪水，若薪水超过 $ 100 000，只增加 3%。

(e) 删除关于 Small Bank Corporation 雇员的关系 *works* 中的所有元组。

5.5　已知如下关系模式：

$$R = (A, B, C)$$
$$S = (D, E, F)$$

关系 r (R) 和 s (S) 已给出。给出与下列查询等价的 QBE、Quel 和 Datalog 表达式。

(a) Π_A (r)

(b) $\sigma_{B=17}$ (r)

(c) $r \times s$

(d) $\Pi_{A,F}$ ($\sigma_{C=D}$ ($r \times s$))

5.6　设 $R = (A, B, C)$，r_1、r_2 都是模式 R 上的关系。给出与下列查询等价的 QBE、Quel 和 Datalog 表达式。

(a) $r_1 \cup r_2$

(b) $r_1 \cap r_2$

(c) $r_1 - r_2$

(d) $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

5.7 设 $R=(A, B)$，$S=(A, C)$，$r(R)$ 和 $s(S)$ 为关系。为下列查询写出 QBE、Quel 和 Datalog 表达式。

(a) $\{<a> \mid \exists b\ (<a, b> \in r \wedge b=17)\}$

(b) $\{<a, b, c> \mid <a, b> \in r \wedge <a, c> \in s)\}$

(c) $\{<a> \mid \exists c\ (<a, c> \in s \wedge \exists b_1, b_2\ (<a, b_1> \in r \wedge <c, b_2> \in r \wedge b_1 > b_2))\}$

5.8 在5.2.3节，我们曾写过一个Quel表达式，找出余额比其所在银行平均余额高的所有帐户。这个查询使用了by子句。说一说如果查询语句中不包含“by *t.branch-name*”，查询将检索的是什么。

5.9 考虑图5-12的关系数据库。为下面的查询写出Datalog程序。

(a) 找出所有（直接或间接）在经理“Jones”下工作的雇员。

(b) 找出（直接或间接）在经理“Jones”下工作的雇员居住的所有城市。

(c) 找出所有的雇员对，他们（直接或间接）为同一个经理工作。

(d) 找出所有的雇员对，他们（直接或间接）为同一个经理工作，并且是位于该经理以下的同一个级别中。

5.10 写出与下面Datalog规则等价的扩展关系代数视图。

$$p(A, C, D) :- q1(A, B), q2(B, C), q3(4, B), D=B+1$$

5.11 描述一个任意的Datalog规则怎样表达为一个扩展的关系代数视图。

文献注解

Query-by-Example的实验版在Zloof [1997]中描述；商业版在IBM [1978b]中描述。许多数据库系统（特别是运行在个人计算机上的数据库系统）实现了QBE或其变体。Quel由Stonebraker等[1976]定义。Stonebraker [1986b]提供了一组关于Ingres系统的研究和概述文章。Ingres现在属于计算机协会。

对Datalog的实现包括LDL系统（在Tsur and Zaniolo [1986]和Naqvi and Tsur [1988]中描述）、Nail!（在Derr等[1993]中描述），以及Coral（在Ramakrishnan等[1992a, 1993]中描述）。关于逻辑库的早期讨论出现在Gallaire and Minker [1978]和Gallaire等[1984]中。这一问题的文章选编由Minker [1988]给出。Ullman [1988, 1989]提供的教科书对逻辑查询语言和实现技术做了全面讨论，这正是我们所用Datalog语法的来源。Ramakrishnan and Ullman [1995]对推理数据库给出了一个更近期的概览。对Datalog查询处理技术做进一步讨论的有Bancilhon and Ramakrishnan [1986]、Beeri and Ramakrishnan [1991]、Ramakrishnan等[1992b]、Srivastava等[1995]，以及Mumick等[1996]。有递归和否定的Datalog程序在否定被“分层”（即不存在通过否定进行的递归）的条件下可被赋予简单的语义。分层否定由Chandra and Harel [1982]和Apt and Pugin [1987]讨论。模块分层语义是一个重要的扩展，处理一类有否定文字的递归程序，在Ross [1990]中讨论。对这种程序求值的一种技术在Ramakrishnan等[1992c]中进行了描述。

第6章　完整性约束

完整性约束提供了一种手段来保证当授权用户对数据库做修改时不会破坏数据的一致性。因此，完整性约束防止的是对数据的意外破坏。

第2章已讲过一种E-R模型的完整性约束。它有以下几种形式：

• 码定义。规定一给定实体集上的某些属性构成一个候选码。合法的插入、更新操作必须保证不会创建两个在候选码上有相同值的实体。

• 联系的形态。多对多、一对多和一对一。一对一或一对多的联系限定了实体集内实体间的合法联系集。

一般来说，一个完整性约束可以是与数据库有关的任意谓词。但检测任意谓词的代价可能太高，因此，通常只局限于那些只需极小开销就可检测的完整性约束。

6.1　域约束

我们都知道，每个属性都必须对应于一个所有可能的取值构成的域。在第4章中，我们知道了如何在SQL的DDL中指明这种约束。域约束是最基本的完整性约束。每当有新数据项插入到数据库中，系统可方便地进行域约束检测。

多个属性有相同的域是可能的。例如，*customer-name* 和 *employee-name* 可能就有相同的域：即所有人名的集合。但 *balance* 和 *branch-name* 这两个属性的域显然不同，*customer-name* 和 *branch-name* 是否应有相同域则可能不太清楚，因为在实现时，*customer-name* 和 *branch-name* 均为字符串。但是，通常认为“找出所有与分支机构同名的客户”并不是个有意义的查询。因此，如果仅从概念上而非物理实现上看数据库，*customer-name* 和 *branch-name* 就应该有不同的域。

由上面的讨论可以看出，域约束的恰当定义不仅可以对插入数据库的值进行检测，而且还可以对查询进行检测，以保证比较是有意义的。

属性域在原理上非常类似于编程语言中变量的类型。强类型语言使得编译器能更细致地检查程序。但是，强类型语言禁止“clever hacks”，而这对编写系统程序常常是必要的。由于数据库系统用于支持非计算机专家的使用，所以强类型检查往往利大于弊。然而，现有的许多系统都只允许少数几种域。一些新系统，特别是面向对象数据库系统，提供了丰富的、易扩展的域类型集合。面向对象数据库将在第8～9章讨论。

SQL-92中的check子句允许对域做强有力的限制，而大多数编程语言的类型系统都不支持如此强大的功能。具体来说，check子句允许数据模式的设计者指定一个谓词，对类型属于该域的变量所赋的任意值都必须满足该谓词。例如，用check子句可以保证小时工资域的值必须大于某一指定值（如最低工资），如下所示：

```
create domain hourly-wage numeric (5, 2)
        constraint wage-value-test check (value >= 4.00)
```

域 *hourly-wage* 定义为一个五位十进制数，其中小数点后有两位，且该域上有一个约束以

保证小时工资数大于或等于 4.00。子句 constraint *wage-value-test* 是可选的，它用来将该约束命名为 *wage-value-test*。这个名字可用于指明哪个约束发生了更新违例的情况。

check 子句也可用于对域做不包含空值的限制，例如：

create domain *account-number* **char** (10)
　　constraint *account-number-null-test* **check** (**value not null**)

另外，使用 in 子句可以将域限制为只包含指定的一组值：

create domain *account-type* **char** (10)
　　constraint *account-type-test*
　　check (**value in** ("Checking", "Saving"))

6.2　参照完整性

我们常常希望，一个关系中给定属性集上的取值也在另一关系的某一属性集的取值中出现，这一条件称为参照完整性。

6.2.1　基本概念

考虑关系 r (R) 和 s (S) 及其自然连接 $r \bowtie s$。可能存在 r 中元组 t_r，它无法与 s 中的任何元组连接。也就是说，不存在 s 中元组 t_s，使得 $t_r[R \cap S] = t_s[R \cap S]$。这样的元组称为悬挂元组。根据被建模的实体集或联系集的不同，可能允许悬挂元组出现，也可能不允许。3.5.2 节曾讨论过一种变形的连接——外连接，它可作用于含悬挂元组的关系。这里，我们主要关心的不是查询，而是考虑希望什么时候允许数据库中有悬挂元组。

假设在关系 *account* 中有元组 t_1 且 t_1「*branch-name*」= "Lunartown"，但关系 *branch* 中没有对应于 Lunartown 分支机构的元组。我们当然不希望出现这种情况，我们希望关系 *branch* 列出了所有分支机构。因此，元组 t_1 可能指的是一个不存在的分支机构中的帐户。显然，需要一种完整性约束来禁止这种悬挂元组。

然而，并非所有悬挂元组的情况都不允许。假设在关系 *branch* 中有元组 t_2 且 t_2[*branch-name*] = "Mokan"，但关系 *account* 中没有对应于 Mokan 分支机构的元组。在这种情况下，分支机构存在但没有帐户　尽管这种情况很少发生，但在分支机构刚刚成立或即将倒闭的时候也是有可能的。因此，我们倒不禁止这种情况。

以上两个例子的区别在于两点：

- *Account-schema* 中的属性 *branch-name* 是一个外码，它参照 *Branch-schema* 的主码。
- *Branch-schema* 的属性 *branch-name* 不是外码。

在 3.1.3 节，我们讲过外码是关系模式中的一个属性集，它同时是另一个模式的主码。

在 Lunartown 的例子中，*account* 中元组 t_1 在外码 *branch-name* 上的值没有出现在 *branch* 中。在 Mokan-branch 的例子中，*branch* 中元组 t_2 在 *branch-name* 上的值没有出现在 *account* 中，但这里 *branch-name* 不是外码。也就是说，这两个悬挂元组的例子的区别在于外码是否存在。

令关系 r_1 (R_1) 和 r_2 (R_2) 的主码分别为 K_1 和 K_2。称 R_2 的子集 α 为参照关系 r_1 中 K_1 的外码，是要求对 r_2 中任意元组 t_2，均存在 r_1 中元组 t_1 使得 $t_1[K_1] = t_2[\alpha]$。这种要

求称为参照完整性约束，或子集依赖。后面一种称法是由于上述参照完整性可写成 $\Pi_{\alpha}(r_2) \subseteq \Pi_{K_1}(r_1)$。注意，为使参照完整性约束有意义，要么 α 等于 K_1，要么 α 和 K_1 为相容的属性集。

6.2.2 E-R 模型中的参照完整性

参照完整性约束是十分常见的。如果像第 2 章那样，通过由 E-R 图创建表来导出关系数据库模式，那么由联系集得到的每一个关系都有参照完整性约束。图 6-1 描述了一个 N 元联系集 R，及相关的实体集 E_1，E_2，…，E_n。令 K_i 表示 E_i 的主码。联系集 R 的关系模式的属性包括 $K_1 \cup K_2 \cup \cdots \cup K_n$。$R$ 模式中的每个 K_i 都是导致参照完整性约束的外码。

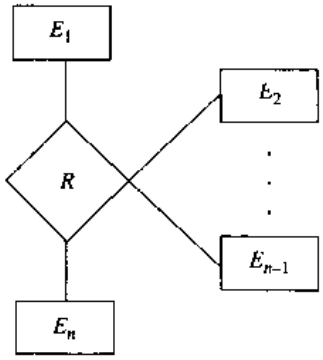


图 6-1　N 元联系集

另一种参照完整性约束的来源是弱实体集。在第 2 章中我们曾提到，弱实体集的关系模式必须包含它所依赖的实体集的主码。因此，每个弱实体集的关系模式包含一个导致参照完整性约束的外码。

6.2.3 数据库的修改

数据库的修改会导致参照完整性的破坏。这里列出对各种类型的数据库修改应做的测试，以保持如下的参照完整性约束：

$$\Pi_{\alpha}(r_2) \subseteq \Pi_{K}(r_1)$$

- 插入。如果向 r_2 中插入元组 t_2，则系统必须保证 r_1 中存在元组 t_1 使得 $t_1[K] = t_2[\alpha]$。即

$$t_2[\alpha] \in \Pi_{K}(r_1)$$

- 删除。如果从 r_1 中删除元组 t_1，则系统必须计算 r_2 中参照 t_1 的元组集合

$$\sigma_{\alpha = t_1[K]}(r_2)$$

如果该集合非空，要么删除命令报错并撤消，要么参照 t_1 的元组本身必须被删除。后一种方案可能导致级联删除，因为对于参照 t_1 的元组，可能又有元组参照它，依此类推。

- 更新。必须考虑两种更新：对参照关系（r_2）做更新，以及对被参照关系（r_1）做更新
  - 如果关系 r_2 中元组 t_2 被更新，并且该更新修改外码（α）上的值，则进行类似插入情况的测试。令 t'_2 表示元组 t_2 的新值，则系统必须保证

$$t'_2[\alpha] \in \Pi_{K}(r_1)$$

  - 如果关系 r_1 中元组 t_1 被更新，并且该更新修改主码（K）上的值，则进行类似删除情况的测试。系统必须用旧的 t_1 值（更新前的值）计算

$$\sigma_{\alpha = t_1[K]}(r_2)$$

如果该集合非空，则更新失败，或者以类似删除的方式做级联更新。

6.2.4 SQL 中的参照完整性

主码、候选码和外码可在 SQL 的 create table 语句中指明：

- create table 语句的 primary key 子句中包含一组构成主码的属性。
- create table 语句的 unique 子句中包含一组构成候选码的属性。
- create table 语句的 foreign key 子句中包含一组构成外码的属性以及被该外码所参照的关系名。

我们使用银行数据库的部分 SQL DDL 定义来说明如何定义主码和外码，如图 6-2 所示。注意，和前几章一样，在银行数据库这个例子中，我们并不要求精确地对现实世界建模。在现实世界中，可能有多个人具有相同的名字，因此，*customer-name* 不应该是 customer 的主码。而另一些属性如社会保障号，或姓名和地址的属性的组合，可以被用作主码。为使库模式简短，用 *customer-name* 作为主码。

```
create table customer
    (customer-name    char(20) not null,
     customer-street  char(30),
     customer-city    char(30),
     primary key (customer-name))

create table branch
    (branch-name      char(15) not null,
     branch-city      char(30),
     assets           integer,
     primary key (branch-name),
     check (assets >= 0))

create table account
    (account-number   char(10) not null,
     branch-name      char(15),
     balance          integer,
     primary key (account-number),
     foreign key (branch-name) references branch,
     check (balance >= 0))

create table depositor
    (customer-name    char(20) not null,
     account-number   char(10) not null,
     primary key (customer-name, account-number),
     foreign key (customer-name) references customer,
     foreign key (account-number) references account)
```

图 6-2　银行数据库的部分 SQL 数据定义

可以使用如下的简写形式定义单个列为外码：

branch-name **char** (15) **references** *branch*

SQL 还支持在外码子句中显式指明被参照关系的一组属性，表明使用这组属性而不用主码。这一组指定的属性必须定义为被参照关系的候选码。

当参照完整性约束被违犯时，通常的处理是拒绝执行导致完整性破坏的操作。但是，在 SQL-92 的 foreign key 子句中可以规定：如果被参照关系的一个删除或更新动作违犯了约束，则可以通过采取一些步骤修改参照关系中的元组来恢复完整性约束，而不是拒绝这一动作。例如关系 *account* 上的如下完整性约束定义：

```
create table account
    (…
    foreign key (branch-name) references branch
                                    on delete cascade
                                    on update cascade,
    …)
```

由于有了与外码定义相关联的 on delete cascade 子句，如果删除 *branch* 中元组导致上面的参照完整性约束被破坏，则删除并不被拒绝，而是对关系 *account* 做“级联”删除，即删除参照了 branch 中被删除元组的那些元组。同样，如果更新一个被参照字段时破坏了约束，则更新操作也不被拒绝，而是将 *account* 中参照元组的 *branch-name* 字段也改为新值。SQL-92 还允许 foreign key 子句指明 cascade 以外的其他动作，例如当约束被破坏时，将参照域（这里是 *branch-name*）置为空或置为缺省值。

如果有涉及多个关系的外码依赖链，则在链一端做的删除或更新操作可能传至整个链。习题 6.4 中有一个有趣的情况，一个关系上定义的 foreign key 约束，其所参照的关系就是它本身。如果一个级联更新或删除导致的对约束的破坏不能通过进一步的级联操作解决，则系统中止该事务。于是，该事务所做的所有改变及级联动作将被撤消。

SQL 中关于码的语义很复杂，这是由于 SQL 中允许空值存在。下面的一些规则用于处理空值，尽管其中有一些并没有什么道理：

- 主码中的所有属性隐式地定义为 not null。
- unique 定义的属性（即候选码中的属性）允许为空，除非它们被显式地定义为非空。关系上的唯一性约束被破坏是指关系中有两个元组在 unique 约束的所有属性上具有相同值，并且所有值均为非空。因此，只要至少有一个列为空值，就可以有任意多个元组在定义为唯一的所有列上均相同且不破坏约束性。这里对空值的处理类似于 4.6.4 节中定义的 unique 结构中对空值的处理。
- 外码中的属性允许为空值，除非它们被显式地定义为非空。如果某一元组中外码的所有列均为非空，则对该元组采用外码约束的通常定义。如果任何一个外码列均为空，则该元组自动被认为满足约束。这样规定没有什么道理，有时还不一定是正确的，因此 SQL 也提供一些结构使你可以修改对空值的处理。这里我们不讨论这样的结构。

由于 SQL 中对空值约束的复杂性和随意性，因而最好保证 unique 和 foreign key 的所有列都定义为非空。

6.3　断言

一个断言就是一个谓词，它表达了我们希望数据库总能满足的一个条件。域约束和参照完整性约束是断言的特殊形式。前面之所以用大量篇幅介绍了这几种断言，是因为它们容易检测并且适用于很多数据库的应用。但是，还有许多约束不能用这几种特殊形式表达。这种约束的例子有：

- 每个分支机构的贷款金额总和必须少于该支行帐户余额的总和。
- 每笔贷款的客户中至少有一人帐户余额不小于 $1000.00。

SQL-92 中的断言为如下形式：

create assertion ＜assertion-name＞ **check** ＜predicate＞

上面两个约束可以写成下面的形式。由于 SQL 不提供“for all X，P（X）”结构（其中 P 是一个谓词），只好使用等价的“not exists X such that not P（X）”结构，该结构可以用 SQL 书写。

```
create assertion sum-constraint check
    (not exists (select * from branch
        where (select sum (amount) from loan
                where loan.branch-name = branch.branch-name)
            >= (select sum (amount) from account
                where account.branch-name = branch.branch-name)))

create assertion balance-constraint check
    (not exists (select * from loan
        where not exists (select *
            from borrower, depositor, account
            where loan.loan-number = borrower.loan-number
                and borrower.customer-name = depositor.customer-name
                and depositor.account-number = account.account-number
                and account.balance >= 1000)))
```

断言创建以后，系统会检测其有效性。如果断言有效，则以后只有不破坏断言的数据库修改才被允许。如果断言较复杂，则检测会带来相当大的开销。因此，使用断言应该特别小心。由于检测和维护断言的开销较高，一些系统开发员省去了对一般性断言的支持，或只提供易于检测的特殊形式的断言。

6.4　触发器

触发器是一条语句，当对数据库做修改时，它自动被系统执行。要设置触发器机制，必须满足两个要求：

1）指明什么条件下触发器被执行。

2）指明触发器执行的动作是什么。

对于示警或满足特定条件时自动执行某项任务来说，触发器是非常有用的机制。例如，银行处理透支时，不是将帐户余额设成负值，而是将帐户余额设成零，并且建一笔贷款，其金额为透支额。这笔贷款的贷款号等于该透支帐户的帐户号。此例中，执行触发器的条件是对关系 *account* 更新时导致了负余额。比如假设 Jones 从帐户支出一些钱从面导致帐户余额成为负值。令 t 代表 *account* 中 *balance* 为负值的元组，则将执行的动作如下：

- 在 *loan* 关系中插入一条新元组 s 如下

$$s\ [branch\text{-}name] = t\ [branch\text{-}name]$$
$$s\ [loan\text{-}number] = t\ [account\text{-}number]$$
$$s\ [amount] = -t\ [balance]$$

注意，由于 t [*balance*] 是负值，因而取它的相反数使得贷款金额为一个正数。

- 在 *borrower* 关系中插入一条新元组 u 如下

$$u\ [customer\text{-}name] = \text{“Jones”}$$
$$u\ [loan\text{-}number] = t\ [account\text{-}number]$$

- 将 t [*balance*] 设为零。

SQL-92 标准不包括触发器，尽管原先的 System R SQL 建议中包含了少量的触发器特性。一些现有系统具有自己的非标准触发器特性。这里给出如何用原先的 SQL 书写帐户透支触发器：

```
define trigger overdraft on update of account T
  (if new T.balance < 0
  then (insert into loan values
            (T.branch-name, T.account-number, - new T.balance)
        insert into borrower
            (select customer-name, account-number
            from depositor
            where T.account-number = depositor.account-number)
        update account S
        set S.balance = 0
        where S.account-number = T.account-number))
```

T.balance 前的关键字 new 表明应使用更新后的 *T.balance* 值；如果不写，则将使用更新前的值。

触发器有时被称作规则或活跃规则，但不要把它和目录规则（参见 5.3 节）混淆起来，目录规则实际上是视图定义。

6.5　函数依赖

本节重点讲述一种称为函数依赖的特殊约束。函数依赖这一概念是第 2、3 章讨论的码的概念的推广。函数依赖在数据库设计中具有重要作用，我们将在第 7 章中探讨。

6.5.1　基本概念

函数依赖是合法关系集上的约束。函数依赖使我们可以表达通过数据库来建模的现实世界中的某些事实。

第 2 章曾定义超码的概念如下。令 R 是关系模式，R 的子集 K 是 R 的超码是指对任意合法关系 r（R）及 r 中任意两个元组 t_1 和 t_2，它们总满足若 $t_1 \neq t_2$，则 t_1 [K] $\neq t_2$ [K]。也就是说，任意合法关系 r（R）中不能有两个元组在属性集 K 上有相同值。

函数依赖的概念是超码概念的推广。令 $\alpha \subseteq R$ 且 $\beta \subseteq R$，R 上存在函数依赖

$$\alpha \rightarrow \beta$$

是指对任意合法关系 r（R）及 r 中任意两个元组 t_1 和 t_2，若 t_1 [α] $= t_2$ [α]，则 t_1 [β] $= t_2$ [β]。

使用函数依赖这一概念，我们可以说如果 $K \rightarrow R$，则 K 是 R 的超码。也就是说，K 是超

码意只要 $t_1[K]=t_2[K]$，均有 $t_1[R]=t_2[R]$（即 $t_1=t_2$）。

函数依赖使我们可以表示不能用超码表示的约束。考虑如下模式

$$Loan\text{-}info\text{-}schema=(branch\text{-}name,\ loan\text{-}number,\ customer\text{-}name,\ amount)$$

在这个关系模式上我们希望有函数依赖集

$$loan\text{-}number \rightarrow amount$$
$$loan\text{-}number \rightarrow branch\text{-}name$$

但是，我们不希望有函数依赖

$$loan\text{-}number \rightarrow customer\text{-}name$$

因为一般来说，一笔贷款可对应于多个客户（例如，贷给夫妻双方）。

可以两种方式使用函数依赖：

1）用于指明合法关系集上的约束。这样就可以只考虑满足给定函数依赖集的那些关系。如果希望只局限于模式 R 上满足函数依赖集 F 的关系，我们说 R 上 F 成立。

2）用于检测关系是否在给定函数依赖集上合法。如果关系 r 在函数依赖集 F 上合法，则称 r 满足函数依赖集 F。

让我们考虑图 6-3 的关系 r，看看它满足什么函数依赖。显然函数依赖 $A\rightarrow C$ 是满足的，因为有两个元组在 A 上的值都为 a_1，且它们在 C 上的值也相等，均为 c_1。同样，在 A 上值为 a_2 的两个元组在 C 上也有相同值 c_2。此外再没有其他不同元组在 A 上有相同值。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 $t_2=(a_3,\ b_3,\ c_2,\ d_4)$，这两个元组在 C 上具有相同的 c_2，但在 A 上的值却不同，分别为 a_2 和 a_3。于是，我们找到两个元组 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 6-3　示例关系 r

r 上还满足很多其他的函数依赖，例如函数依赖 $AB\rightarrow D$。注意，我们用 AB 作为 $\{A,\ B\}$ 的简写，以保持和习惯表示的一致。由于没有两个不同元组 t_1 和 t_2 能使 $t_1[AB]=t_2[AB]$，因此，若 $t_1[AB]=t_2[AB]$，必有 $t_1=t_2$，也就有 $t_1[D]=t_2[D]$。所以，r 满足 $AB\rightarrow D$。

有些函数依赖被称为平凡的，因为它们在所有关系中都是满足的。例如，$A\rightarrow A$ 在所有包含属性 A 的关系中都是满足的。完全按定义解释，该函数依赖是指对所有元组 t_1 及 t_2，若 $t_1[A]=t_2[A]$，则 $t_1[A]=t_2[A]$。同样，$AB\rightarrow A$ 也在所有包含属性 A 的关系中都是满足的。一般来说，如果 $\beta\subseteq\alpha$，则形如 $\alpha\rightarrow\beta$ 的函数依赖是平凡的。

为区分概念“满足依赖的关系”以及“模式上成立的依赖”，再看银行那个例子。考虑如图 6-4 所示的关系 *customer*（在 *Customer-schema* 上），*customer-street*→*customer-city* 是满足的。但是，在现实世界中，有可能有两个城市具有相同名字的街道。因此，就有可能在某些时候，某 *customer* 关系实例上不满足 *customer-street*→*customer-city*。所以，不能将 *customer-street*→*customer-city* 包含进 *Customer-schema* 上成立的函数依赖集中。

在图 6-5 的关系 *loan* 中（在 *Loan-schema* 上），*loan-number*→*amount* 是满足的。与 *Customer-schema* 中的 *customer-city* 与 *customer-street* 的例子不同，这里所描述的现实世界的情况的

customer-name	*customer-street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

图 6-4　关系 *customer*

确要求一笔贷款只有一个金额数。因此，在任何时候关系 *loan* 上都有 *loan-number*→*amount* 满足。换句话说，约束 *loan-number*→*amount* 在 *Loan-schema* 上成立。

branch-name	*loan-number*	*amount*
Downtown	L-17	1000
Redwood	L-23	2000
Perryridge	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
Round Hill	L-11	900
Pownal	L-29	1200
North Town	L-16	1300
Downtown	L-18	2000
Perryridge	L-25	2500
Brighton	L-10	2200

图 6-5　关系 *loan*

在图 6-6 的关系 *branch* 中，*branch-name*→*assets* 和 *assets*→*branch-name* 都是满足的。但是，我们要求 *Branch-schema* 上 *branch-name*→*assets* 成立，却不要求 *assets*→*branch-name* 成立，因为很可能有几个分支机构具有相同的资产额。

branch-name	*branch-city*	*assets*
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
Pownal	Bennington	300000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

图 6-6　关系 *branch*

尽管 SQL 不提供定义函数依赖的简便方法，但可以书写查询语句来检测函数依赖以及创建断言来体现函数依赖，正如习题 6.19 将讨论的那样。

以下假设在设计关系数据库时，首先列出那些总是必须成立的函数依赖。在银行的例子

中，包含如下一些依赖：

- 在 *Branch-schema* 上

$$branch\text{-}name \rightarrow branch\text{-}city$$
$$branch\text{-}name \rightarrow assets$$

- 在 *Customer-schema* 上

$$customer\text{-}name \rightarrow customer\text{-}city$$
$$customer\text{-}name \rightarrow customer\text{-}street$$

- 在 *Loan-schema* 上

$$loan\text{-}number \rightarrow amount$$
$$loan\text{-}number \rightarrow branch\ name$$

- 在 *Borrower-schema* 上无函数依赖
- 在 *Account-schema* 上

$$account\text{-}number \rightarrow branch\text{-}name$$
$$account\text{-}number \rightarrow balance$$

- 在 *Depositor-schema* 上无函数依赖

6.5.2　函数依赖集的闭包

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其他所有函数依赖。给定函数依赖集 F，可以证明其他某些函数依赖也成立，我们称这些函数依赖被 F 逻辑蕴涵。

假设给定关系模式 $R=(A, B, C, G, H, I)$ 及函数依赖集

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

则函数依赖

$$A \rightarrow H$$

被逻辑蕴涵。也就是说，我们能够证明，只要如上给定的函数依赖集成立，$A \rightarrow H$ 也一定成立。假设有元组 t_1 及 t_2，满足

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

根据已知 $A \rightarrow B$，由函数依赖的定义可以推出

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

又根据已知 $B \rightarrow H$，由函数依赖的定义可以推出

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

因此，已经证明，对任意两个元组 t_1 及 t_2，只要 $t_1[A] = t_2[A]$，均有 $t_1[H] = t_2[H]$。这正是 $A \rightarrow H$ 的定义。

令 F 为一个函数依赖集。F 的闭包是指 F 逻辑蕴涵的所有函数依赖的集合。将 F 的闭包记为 F^+。可以由函数依赖的形式化定义直接计算 F^+。如果 F 很大，则这个过程会很长而且很难。F^+ 的这种算法需要像前面例子中证明 $A \rightarrow H$ 属于给定依赖集的闭包那样进行论证。推导函数依赖还有一些更简单的技术。

第一种技术基于三个公理或称函数依赖推理规则。通过反复使用这些规则，可以找出给定 F 的 F^+。在下面的规则中，按惯例用希腊字母（α，β，γ，…）表示属性集，用字母表中的大写罗马字母表示单个属性，用 $\alpha\beta$ 表示 $\alpha \cup \beta$。

- 自反律。若 α 为一属性集且 $\beta \subseteq \alpha$，则有 $\alpha \rightarrow \beta$。
- 增补律。若有 $\alpha \rightarrow \beta$ 且 γ 为一属性集，则有 $\gamma\alpha \rightarrow \gamma\beta$。
- 传递律。若有 $\alpha \rightarrow \beta$ 及 $\beta \rightarrow \gamma$，则有 $\alpha \rightarrow \gamma$。

这些规则是保真的，因为它们不会产生错误的函数依赖。这些规则也是完备的，因为对一给定的函数依赖集 F，它们能产生整个 F^+。这组规则称为 *Armstrong* 公理，以纪念首次提出这一公理的人。

尽管 *Armstrong* 公理是完备的，但直接用它们计算 F^+ 会很麻烦。为进一步简化，我们列出其他一些规则。可以用 *Armstrong* 公理证明这些规则也是正确的（参见习题 6.12、6.13 及 6.14）。

- 合并律。若有 $\alpha \rightarrow \beta$ 及 $\alpha \rightarrow \gamma$，则有 $\alpha \rightarrow \beta\gamma$。
- 分解律。若有 $\alpha \rightarrow \beta\gamma$，则有 $\alpha \rightarrow \beta$ 及 $\alpha \rightarrow \gamma$。
- 伪传递律。若有 $\alpha \rightarrow \beta$ 及 $\gamma\beta \rightarrow \delta$，则有 $\alpha\gamma \rightarrow \delta$。

让我们对前面作为例子的模式 $R = (A, B, C, G, H, I)$ 及函数依赖集 $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$ 使用这些规则。在这里列出 F^+ 中的几个依赖：

- $A \rightarrow H$。由于有 $A \rightarrow B$ 及 $B \rightarrow H$，使用传递律。可以看到使用 *Armstrong* 公理证明 $A \rightarrow H$ 比前面直接用定义论证要简单得多。
- $CG \rightarrow HI$。由于有 $CG \rightarrow H$ 及 $CG \rightarrow I$，由合并律可推出 $CG \rightarrow HI$。
- $AG \rightarrow I$。由于有 $A \rightarrow C$ 及 $CG \rightarrow I$，由伪传递律可推出 $AG \rightarrow I$。

6.5.3 属性集的闭包

要检验属性集 α 是否为超码，必须设计算法来计算被 α 函数确定的属性集。这个算法也可以作为计算函数依赖集 F 的闭包的一部分。

令 α 为一属性集。我们称在函数依赖集 F 下被 α 函数确定的所有属性为 F^+ 下 α 的闭包，记为 α^+。图 6-7 是用伪 Pascal 书写的计算 α^+ 的算法。输入为函数依赖集 F 和属性集 α，输出存储在变量 *result* 中。

为解释图 6-7 中的算法如何进行，我们将使用该算法计算前面定义的函数依赖集下的 $(AG)^+$。开始时 $result = AG$。在第一次执行检验各个函数依赖的 while 循环时：

- 由 $A \rightarrow B$ 得出 B 属于 *result*。这是因为 $A \rightarrow B$ 属于 F，$A \subseteq result$（即 AG），于是 $result := result \cup B$。

```
result := α;
while (result发生变化) do
    for each 函数依赖 β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ;
        end
```

图 6-7　计算 F 下 α 的闭包 α^+ 的算法

- 由 $A \rightarrow C$，*result* 变为 $ABCG$。
- 由 $CG \rightarrow H$，*result* 变为 $ABCGH$。
- 由 $CG \rightarrow I$，*result* 变为 $ABCGHI$。

在第二次执行 while 循环时，*result* 中未加入新属性，算法终止。

来看看图 6-7 的算法为什么正确。第一步正确是因为总有 $\alpha \rightarrow \alpha$（由自反律）。对 *result* 的任意子集 β，有 $\alpha \rightarrow \beta$。因为既然 while 循环时 $\alpha \rightarrow result$ 为真，只要 $\beta \subseteq result$ 且 $\beta \rightarrow \gamma$，就可将 γ 加入 *result*。其中 $result \rightarrow \beta$ 由自反律得到，$\alpha \rightarrow \beta$ 由传递律得到。再使用传递律就得到 $\alpha \rightarrow \gamma$（由 $\alpha \rightarrow \beta$ 及 $\beta \rightarrow \gamma$）。由合并律可推出 $\alpha \rightarrow result \cup \gamma$，所以 α 函数确定 while 循环中产生的所有新结果。也就是说，算法所返回的属性一定属于 α^+。

容易证明算法可找出所有 α^+ 中的属性。如果存在属性属于 α^+ 却还未包含到 *result* 中，则必有函数依赖 $\beta \rightarrow \gamma$（$\beta \subseteq result$）且 γ 中至少有一个属性还未包含到 *result* 中。

在最坏情况下，该算法的执行时间为 F 集合规模的二次方。另外还有一个更快但略微复杂一点儿的算法，它的执行时间与 F 的规模呈线性关系。该算法作为习题 6.18 的一部分已列出。

6.5.4　正则覆盖

假设在关系模式上有函数依赖集 F，那么在该关系上做任何更新时，数据库系统都要保证 F 中所有函数依赖在新数据库状态下仍然满足，否则就回滚该更新操作。可以简化给定的函数依赖集，但不改变它的闭包，以降低检测的开销。由于闭包相同，所以满足简化后的函数依赖集的数据库也一定满足原依赖集，反之亦然。然而，简化后的集合更便于检测。简化后的集合可用后面所描述的方法构造。首先，需要给出一些定义。

如果可以划掉一个函数依赖集中的属性而不改变函数依赖集的闭包，我们称该属性是无关的。无关属性可形式化地定义如下。考虑函数依赖集 F 及 F 中函数依赖 $\alpha \rightarrow \beta$。

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

F 的**正则覆盖** F_C 是一个依赖集意指 F 逻辑蕴涵 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 $\alpha_2 \rightarrow \beta_2$ 且 $\alpha_1 = \alpha_2$。

函数依赖集 F 的正则覆盖可用如下算法计算：

repeat

使用合并律将 F 中形如 $\alpha_1 \rightarrow \beta_1$ 和 $\alpha_1 \rightarrow \beta_2$ 的依赖替换为 $\alpha_1 \rightarrow \beta_1\beta_2$.

找出在 α 或 β 中含无关属性的函数依赖 $\alpha \rightarrow \beta$.

若发现无关属性，则将它从 $\alpha \rightarrow \beta$ 中删除.

until F 不再改变.

可以证明 F 的正则覆盖 F_C 具有与 F 相同的闭包。因此，检测 F_C 是否满足等价于检测 F 是否满足。但是，从某种意义上说，F_C 是最小的——它不含无关属性，并且具有相同左半部的函数依赖都已被合并。所以检测 F_C 比检测 F 本身要容易。

考虑模式（A，B，C）上的如下函数依赖集 F：

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

现在来计算 F 的正则覆盖。

- 有两个函数依赖的箭头左边有相同的属性集

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

将它们合并成 $A \rightarrow BC$。

- A 在 $AB \rightarrow C$ 中是无关的，因为 F 逻辑蕴涵 $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$。这个断言为真是由于 $B \rightarrow C$ 在我们的函数依赖集中已经存在。
- 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，可能集合中整个函数依赖都是无关的，也就是说删掉它不改变 F 的闭包。可以证明 F 的正则覆盖 F_C 不会包含这种无关函数依赖。利用反证法，假设 F_C 中存在无关函数依赖，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。

6.6　总结

完整性约束保证了授权用户对数据库的修改不会导致数据一致性的破坏。前面几章考虑了几种约束，包括码定义及联系的形态的定义（多对多、多对一、一对一）。本章讨论了其他几种约束以及保证这些约束的机制。

域约束指出了与属性相关联的可取值的集合。它也可以禁止某属性上取空值。

参照完整性约束保证一个关系的给定属性集上的取值也出现在另一关系的某个属性集上。

函数依赖是码依赖的扩展。它要求某属性集的值唯一确定另一属性集的值。使用函数依赖的形式化定义，可以确定给定函数依赖集 F 逻辑蕴涵的所有函数依赖的集合，该集合称为 F 的闭包。我们还知道了如何在不改变闭包的前提下简化函数依赖集 F。此简化后的集合称为 F

的正则覆盖 F_C。

域约束、参照完整性约束及函数依赖比较易于检测。更复杂的约束的使用可能导致很大的开销。有两种方式表示更一般的约束。断言是声明性的表达式，它可以定义总要求其为真的谓词。触发器是当某事件发生时要执行的过程。

习题

6.1　完成图 6-2 中银行数据库的 SQL DDL 定义，使其包含关系 *loan* 和 *borrower*.

6.2　考虑如下关系数据库：

employee (*persom-name*, *street*, *city*)
works (*person-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*person-name*, *manager-name*)

给出该数据库的 SQL DDL 定义。指出其应具有的参照完整性约束，并将它们在 DDL 中表示出来。

6.3　本章中所定义的参照完整性约束总是涉及两个关系。考虑包含如下关系的数据库：

salaried-worker (*name*, *office*, *phone*, *salary*)
hourly-worker (*name*, *hourly-wage*)
address (*name*, *street*, *city*)

假设要求 *address* 中出现的每个名字要么在 *salaried-worker* 中出现，要么在 *hourly-worker* 中出现，但是不必在两个中都出现。

(a) 设计表达该约束的语法。

(b) 讨论系统为保证这种约束必须采取的动作。

6.4　SQL-92 标准中允许外码依赖参照同一个关系，如下面的例子：

create table *manager*
(*employee-name* **char** (20) **not null**
manager-name **char** (20) **not null**,
primary key *employee-name*,
foreign key (*manager-name*) **references** *manager*
on delete cascade)

这里，*employee-name* 是表 *manager* 的码，每个雇员至多有一个经理。外码子句要求每个经理也是雇员。解释当关系 *manager* 中的元组被删除时会发生什么。

6.5　假设有两个关系 r 和 s，r 的外码 B 参照 s 的主码 A。考虑如何用触发器实现元组从 s 中删除时的 on delete cascade 选项。

6.6　为银行数据库写一个断言，保证 Perryridge 分支机构的资产额等于 Perryridge 分支机构借出的金额总和。

6.7　为什么某些函数依赖被称作平凡的函数依赖？

6.8　列出图 6-8 中的关系所满足的所有函数依赖。

6.9　使用函数依赖的定义论证 Armstrong 公理（自反律，增补律，传递律）是保真的。

6.10　解释如何用函数依赖表明：

- 实体集 *student* 和 *advisor* 间存在一对一的联系集。
- 实体集 *student* 和 *advisor* 间存在多对一的联系集。

6.11　假设函数依赖有如下规则：若 $\alpha \rightarrow \beta$ 且 $\gamma \rightarrow \beta$，则 $\alpha \rightarrow \gamma$。证明该规则不保真，即证明存在关系 r 满足 $\alpha \rightarrow \beta$ 且 $\gamma \rightarrow \beta$，但不满足 $\alpha \rightarrow \gamma$。

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 6-8　习题 6.8 的关系

6.12　用 Armstrong 公理证明合并律的保真性。
（提示：使用增补律可证，若 $\alpha \rightarrow \beta$，则 $\alpha \rightarrow \alpha\beta$。再对 $\alpha \rightarrow \gamma$ 也使用增补律，最后使用传递律。）

6.13　用 Armstrong 公理证明分解律的保真性。

6.14　用 Armstrong 公理证明伪传递律的保真性。

6.15　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。

6.16　用习题 6.15 中的函数依赖计算 B^+。

6.17　用习题 6.15 中的函数依赖计算正则覆盖 F_C。

6.18　考虑图 6-9 中计算 α^+ 的算法。证明该算法比图 6-7（6.5.3 节）中的算法更有效，并且是正确的。

6.19　给定数据库模式 $R(a, b, c)$ 及模式 R 上的关系 r，写出检测关系 r 上是否具有函数依赖 $b \rightarrow c$ 的 SQL 查询，并写出保证函数依赖的 SQL 断言，假设不存在空值。

文献注解

关系模型上完整性约束的讨论参考了 Hammer 和 McLeod [1975]、Eswaran 和 Chamberlin [1975]、Schmid 和 Swenson [1975]，及 Codd [1979]。

函数依赖最先由 Codd [1970] 定义。Armstrong 公理是由 Armstrong [1974] 引入的。函数依赖理论在 Maier [1983] 中作了讨论。合法关系概念的形式描述在 Graham 等 [1986] 中作了讨论。将函数依赖应用于语义数据模型在 Weddell [1992] 中作了讨论。

有关断言和触发器的 SQL 建议最初在 Astrahan 等 [1976]、Chamberlin 等 [1976]，及 Chamberlin 等 [1981] 中作了讨论。SQL 标准可参见第 4 章的文献注解。

如何有效地维护和检测语义完整性断言在 Hammer 和 Sarin [1978]、Badal 和 Popek [1979]、Bernstein 等 [1980b]、Hsu 和 Imielinski [1985]、McCune 和 Henschen [1989]，及 Chomicki [1992a, 1992b] 中作了讨论。主动数据库在 McCarthy 和 Dayal [1989]、Gehani 和 Jagadish [1991]，及 Gehani 等 [1992] 中作了讨论。

运行时刻完整性检测的另一个用途是保证访问数据库的程序的正确性。这在 Sheard 和

Stemple［1989］中作了讨论。

```
result := ∅;
/* fdcount 是一个数组，其第 i 个元素包含第 i 个 FD 左端且尚不在 α+
   中的属性数目 */
for i := 1 to |F| do
   begin
      令 β → γ 表示第 i 个 FD;
      fdcount [i] := |β|;
   end
/* appears 是一个数组，对应于每一属性有一项，对应属性 A 的项
是一个整数列表。列表中每个整数 i 代表 A 出现在第 i 个 FD 左端。*/
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 FD;
            if A ∈ β then 将 i 加入到 appears [A];
         end
   end
addin (α);
return (result);

procedure addin (α);
for each α 中属性 A do
   begin
      if A ∉ result then
         begin
            result := result ∪ {A};
            for each appears[A] 中元素 i do
               begin
                  fdcount [i] := fdcount [i] − 1;
                  if fdcount [i] := 0 then
                     begin
                        令 β → γ 表示第 i 个 FD;
                        addin (γ);
                     end
               end
         end
   end
```

图 6-9　计算 α^+ 的一种算法

第 7 章　关系数据库设计

本章继续讨论关系数据库的设计问题。总的来说，关系数据库设计的目标是生成一组关系模式，使我们既不必存储不必要的冗余信息，又可以方便地获取信息。关系数据库设计的方法之一就是设计满足适当范式的模式。要确定关系模式是否属于某一范式，还需要有关作为数据库建模对象的现实世界活动的额外信息。我们已经知道如何用函数依赖表示有关数据的一些事实，本章继续用函数依赖及其他数据依赖定义范式。

7.1　关系数据库设计中易犯的错误

在讨论范式及数据依赖之前，先来看看什么是不好的数据库设计。不好的设计可能有如下性质：

- 信息重复。
- 不能表示某些信息。

我们将通过对银行的例子使用一个修改了的数据库设计来说明这些问题。与第 3～6 章的关系模式不同，这里关于贷款的信息存在一个单独的关系 *lending* 中，它的关系模式是

$$Lending\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name,\ loan\text{-}number,\ amount)$$

图 7-1 表示了关系 *lending*（*Lending-schema*）的一个实例。关系 *lending* 中元组 t 的直观意义是：

- t [*assets*] 是名为 t [*branch-name*] 的分支机构的资产额。
- t [*branch-city*] 是名为 t [*branch-name*] 的分支机构的所在城市。
- t [*loan-number*] 是名为 t [*branch-name*] 的分支机构给名为 t [*customer-name*] 的客户贷款的贷款号。
- t [*amount*] 是贷款号为 t [*loan- number*] 的贷款额。

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-23	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

图 7-1　示例关系 *lending*

假设希望向数据库中加入一笔新贷款，比如 Perryridge 分支机构要向 Adams 贷款 $ 1500，*loan-number* 是 L-31。由于元组在 *Lending-schema* 的所有属性上都得有值，这样，就不得不重复 Perryridge 分支机构的资产和所在城市这两个信息。本例向关系 *lending* 中所插入的元组是

(Perryridge，Horseneck，1700000，Adams，L-31，1500)

一般来说，分支机构的资产和所在城市信息在该分支机构的每笔贷款中都要出现一次。

使用修改的设计所造成的信息重复并不是我们想要的。重复信息会浪费空间，此外，它还会使对数据库的更新复杂化。例如，假设 Perryridge 分支机构从 Horseneck 搬到了 Newtown。在原来的设计中，只需修改关系 *branch* 中的一个元组；而在修改后的设计中，则要修改关系 *lending* 中的多个元组。也就是说，修改设计后的更新比原设计中的更新开销要大。当对修改设计后的数据库做更新时，必须保证有关 Perryridge 分支机构的所有元组都被更新，否则 Perryridge 分支机构将对应于两个所在城市。

以上对于理解为什么修改后的设计不好是很重要的。我们知道，一个分支机构总是位于一个城市，而另一方面，一个分支机构可以有多笔贷款。换句话说，*Lending-schema* 上有函数依赖

$$branch\text{-}name \rightarrow branch\text{-}city$$

但没有函数依赖 $branch\text{-}name \rightarrow loan\text{-}number$。一个分支机构位于一个城市与一个分支机构贷了一笔款是两个独立的事实，所以如上所述，最好用单独的两个关系来表示。以下我们会看到用函数依赖可形式地指明什么样的数据库设计是好的设计。

Lending-schema 设计中的另一个问题是不能直接表示有关一个分支机构的信息（*branch-name*，*branch-city*，*assets*），除非该分支机构已至少有一笔贷款。此问题的原因是由于关系 *lending* 中的元组在 *loan-number*、*amount*、*customer-name* 上必须有值。

对于这个问题的解决方法是引入空值通过视图处理更新。但是前面提到过，空值处理起来会很麻烦。如果不想引入空值，就只能在分支机构贷第一笔款时才将它的信息放入关系中。更糟的是，当所有贷款都偿还后，不得不删掉这一信息。显然这样做是不好的，因为在原数据库设计中，不管分支机构当前有无贷款，分支机构信息都能得到，而且不用引入空值。

7.2　模式分解

7.1 节中的设计启示我们，应将一个有许多属性的关系模式分解成只有较少属性的几个模式。但是分解不当可能会导致另一种不好的设计。

例如，*Lending-schema* 被分解成如下两个模式：

$$Branch\text{-}customer\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name)$$
$$Customer\text{-}loan\text{-}schema = (customer\text{-}name,\ loan\text{-}number,\ amount)$$

使用图 7-1 的关系 *lending*，构造如下新关系 *branch-customer*（*Branch-customer-schema*）和 *customer-loan*（*Customer-loan-schema*）：

$$branch\text{-}customer = \Pi_{branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name}\ (lending)$$
$$customer\text{-}loan = \Pi_{customer\ name,\ loan\text{-}number,\ amount}\ (lending)$$

图 7-2 和图 7-3 中分别给出了这两个新关系 *branch-customer* 和 *customer-loan*。

branch-name	*branch-city*	*assets*	*customer-name*
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

图 7-2　关系 *branch-customer*

customer-name	*loan-number*	*amount*
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-23	2000
Glenn	L-25	2500
Brooks	L-10	2200

图 7-3　关系 *customer-loan*

当然，有时需要重新构造关系 *loan*。例如，假设希望找出金额少于 $1000 的贷款的所有分支机构。在修改后的数据库中没有关系含这种数据，需要重新构造关系 *lending*。乍一看，可以用

$$branch\text{-}customer \bowtie customer\text{-}loan$$

来构造。图 7-4 给出了 *branch-customer* ⋈ *customer-loan* 的计算结果。将它与原来的关系 *lending*（图 7-1）比较，我们发现了一些不同。尽管 *lending* 中的每个元组都出现在 *branch-customer* ⋈ *customer-loan* 中，但 *branch-customer* ⋈ *customer-loan* 中却有元组不在 *lending* 中。本例 *branch-customer* ⋈ *customer-loan* 中有如下额外元组：

(Downtown, Brooklyn, 9000000, Jones, L-93, 500)
(Perryridge, Horseneck, 1700000, Hayes, L-16, 1300)
(Mianus, Horseneck, 400000, Jones, L-17, 1000)
(North Town, Rye, 3700000, Hayes, L-15, 1500)

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-23	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

图 7-4　关系 branch-customer ⋈ customer-loan

考虑查询“找出有金额少于 $1000 的贷款的所有分支机构”。如果从图 7-1 中找，只有 Mianus 和 Round Hill 分支机构有金额少于 $1000 的贷款。但是，如果用表达式

$$\Pi_{branch\text{-}name}\ (\sigma_{amount<1000}\ (branch\text{-}customer \bowtie customer\text{-}loan))$$

就会得到三个分支机构：Mianus、Round Hill 和 Downtown。

让我们更深入地看看这个例子。如果一个客户恰好从不同的分支机构贷款，将分不清哪个分支机构贷了哪笔款。于是，当连接 *branch-customer* 和 *customer-loan* 时，不仅会得到原先在 *lending* 中的元组，还有一些额外元组。尽管 *branch-customer* ⋈ *customer-loan* 中有更多的元组，实际上信息却少了。一般来说已无法再在数据库中表示哪个客户从哪个分支机构贷款这样的信息。由于丢失了信息，我们称 *Lending-schema* 至 *Branch-customer-schema* 和 *Customer-loan-schema* 的分解为有损分解，或有损连接分解。不是有损连接分解的分解称为无损连接分解。从这个例子不难看出有损连接分解通常是不好的数据库设计。

让我们再更深入地看看分解为什么会是有损的。*Branch-customer-schema* 和 *Customer-loan-schema* 有一个公共属性：

$$Branch\text{-}customer\text{-}schema \cap Customer\text{-}loan\text{-}schema = \{customer\text{-}name\}$$

表示诸如 *loan-number* 和 *branch-name* 间的联系的唯一方式是通过 *customer-name*。而这样表示是不够的，因为一个客户可能有多笔贷款，而这些贷款不一定来自同一个分支机构。

再看另一个设计，*Lending-schema* 被分解为如下两个模式：

$$Branch\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets)$$
$$Loan\text{-}info\text{-}schema = (branch\text{-}name,\ customer\text{-}name,\ loan\text{-}number,\ amount)$$

这两个模式有一个公共属性：

$$Branch\text{-}loan\text{-}schema \cap Customer\text{-}loan\text{-}schema = \{branch\text{-}name\}$$

因此，表示诸如 *customer-name* 和 *assets* 间的联系的唯一方式是通过 *branch-name*。本例和前一个例子的区别在于不论客户是谁，一个分支机构的资产额都是一样的；而分支机构与贷款的联系则依赖于客户。对于给定的 *branch-name*，只有一个 *assets* 值和一个 *branch-city* 值，而对于 *customer-name* 则不然。也就是说，存在函数依赖

$$branch\text{-}name \rightarrow assets\ branch\text{-}city$$

但 *customer-name* 却不能函数确定 *loan-number*。

无损连接的概念对关系数据库设计至关重要。因此，我们更简洁和更形式化地讨论一下前面几个例子。令 R 为关系模式，关系模式集 $\{R_1, R_2, \cdots, R_n\}$ 为 R 的一个分解：

$$R = R_1 \cup R_2 \cup \cdots \cup R_n$$

即对 $i = 1, 2, \cdots, n$，R_i 为 R 的子集，且 R 中的每个属性至少出现在一个 R_i 中。

令 r 为模式 R 上的关系，$r_i = \Pi_{R_i}(r)$ $(i = 1, 2, \cdots, n)$。即 $\{r_1, r_2, \cdots, r_n\}$ 是将

R 分解为 $\{R_1, R_2, \cdots, R_n\}$ 产生的数据库。于是总有

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

下面证明这一断言为真。考虑关系 r 中一个元组 t。当计算关系 $r_1, r_2, \cdots, r_n$ 时，元组 t 在每个关系 r_i（$i=1, 2, \cdots, n$）中增加一个元组 t_i。当计算 $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$ 时，这 n 个元组连接可形成 t。证明的细节作为习题由读者完成。因此，r 中的每个元组都出现在 $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$ 中。

通常，$r \neq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$。为说明这一点，考虑前面的例子，其中

- $n=2$。
- $R = Lending\text{-}schema$。
- $R_1 = Branch\text{-}customer\text{-}schema$。
- $R_2 = Customer\text{-}loan\text{-}schema$。
- r = 图 7-1 中的关系。
- r_1 = 图 7-2 中的关系。
- r_2 = 图 7-3 中的关系。
- $r_1 \bowtie r_2$ = 图 7-4 中的关系。

注意，图 7-1 和图 7-4 中的关系不一样。

为得到无损连接分解，需要在可能的关系集合上加约束。将 *Lending-schema* 分解为 *Branch-schema* 和 *Loan-info-schema* 是无损的，因为 *Branch-schema* 上有函数依赖

$$branch\text{-}name \rightarrow branch\text{-}city\ assets$$

本章后面将介绍除了函数依赖以外的其他约束。一个关系如果满足定义在数据库上的所有规则或约束，则我们称它为合法的。

令 C 表示数据库上的约束集。如果对模式 R 上满足 C 的所有合法关系 r，均有

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \cdots \bowtie \Pi_{R_n}(r)$$

则我们说关系模式 R 的分解 $\{R_1, R_2, \cdots, R_n\}$ 是关于 R 的无损连接分解。

我们将在下面几节中看到如何检测一个分解是否为无损连接分解。本章主要是讲述如何定义数据库中的约束，以及如何得到无损连接分解，以避免本节中那些不好的数据库设计的例子所犯的错误。

7.3　利用函数依赖作规范化

可以在关系数据库设计中利用给定的函数依赖集，以避免 7.1 节讨论的大多数不良性质。在设计系统时，可能需要将一个关系分解为几个更小的关系。利用函数依赖，可以定义几种范式来代表“好”的数据库设计。范式有很多种，在这里我们要讨论的是 BCNF（7.3.2 节）和 3NF（7.3.3 节）。

7.3.1　分解应具有的特性

本节用 7.1 节的 *Lending-schema* 解释有关概念：

$$Lending\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name,\ loan\text{-}number,\ amount)$$

Lending-schema 上有函数依赖集 F

$$branch\text{-}name \rightarrow assets\ branch\text{-}city$$
$$loan\text{-}number \rightarrow amount\ branch\text{-}name$$

正如 7.1 节所讨论的，*Lending-schema* 是不好的数据库设计。假设将它分解成如下三个关系：

$$Branch\text{-}schema = (branch\text{-}name,\ assets,\ branch\text{-}city)$$
$$Loan\text{-}schema = (branch\text{-}name,\ loan\text{-}number,\ amount)$$
$$Borrower\text{-}schema = (customer\text{-}name,\ loan\text{-}number)$$

则该分解就具备了几个良好性质，下面还会讨论。注意，这三个关系模式正是第 3～5 章使用的模式。

1. 无损连接分解

我们在 7.1 节中提出，将一个关系分解成若干较小的关系时，保证分解无损是很重要的。我们还指出上面的分解就是无损的。为证明该判断正确，首先必须提出判定分解是否无损的标准。

令 R 为一关系模式，F 为 R 上函数依赖集。R_1 和 R_2 为 R 的分解。该分解为 R 的无损连接分解只要 F^+ 中至少有如下函数依赖中的一个：

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

可以通过给出产生分解的步骤，证明对 *Lending-schema* 的分解是无损连接分解。先将 *Lending-schema* 分成两个模式：

$$Branch\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets)$$
$$Loan\text{-}info\text{-}schema = (branch\text{-}name,\ customer\text{-}name,\ loan\text{-}number,\ amount)$$

由 *Branch-name→branch-city assets* 和函数依赖的增补律（6.5.2 节）可得

$$branch\text{-}name \rightarrow branch\text{-}name\ branch\text{-}city\ assets$$

由 $Branch\text{-}schema \cap Loan\text{-}info\text{-}schema = \{branch\text{-}name\}$ 可知第一步分解为无损连接分解。

下一步，将 *Loan-info-schema* 分解成

$$Loan\text{-}schema = (branch\text{-}name,\ loan\text{-}number,\ amount)$$
$$Borrower\text{-}schema = (customer\text{-}name,\ loan\text{-}number)$$

这一步也是无损连接分解，因为 *Loan-number* 是公共属性并且 *loan-number→amount branch-name*。

2. 保持依赖

关系数据库设计的另一个目标是保持依赖。当对数据库做更新时，系统应能检查该更新操作，保证不会产生非法关系，即不能满足所有给定函数依赖的关系。如果想要有效地检查更新，设计的关系数据库模式应允许不用做连接就可确认更新正确。

为决定是否必须计算连接，需要确定通过逐一检查每个关系，可验证哪些函数依赖。令 F 为模式 R 上的函数依赖集，R_1，R_2，…，R_n 为 R 的分解。F 在 R_i 上的投影是 F^+ 中所有只包含 R_i 中属性的函数依赖的集合 F_i。由于一个投影中的所有函数依赖只涉及一个关系模式的属性，因而判定这种依赖是否满足可以只检查一个关系。

投影集 F_1，F_2，…，F_n 是能被有效检查的依赖集。现在必须要问是否只验证投影就够了。令 $F' = F_1 \cup F_2 \cup \cdots \cup F_n$。$F'$是模式 R 上的函数依赖集，但通常 $F' \neq F$。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，如果证明 F'是满足的，则 F 也是满足的。称具有性质 $F'^+ = F^+$的分解为保持依赖的分解。图 7-5 给出了判定保持依赖性的算法。输入是分解了的关系模式集 $D = \{R_1, R_2, \cdots, R_n\}$ 和函数依赖集 F。

```
计算F+;
for each D中模式Ri do
    begin
        Fi := F+ 在Ri上的投影;
    end
F' := ∅
for each 投影Fi do
    begin
        F' = F' ∪ Fi
    end
计算F'+;
if (F'+ = F+) then return (true)
              else return (false);
```

图 7-5　判定保持依赖性的一种算法

可以证明对 *Lending-schema* 的分解是保持依赖的。对 *Lending-schema* 上的函数依赖集 F 中的每个依赖，证明它可以在分解后的至少一个关系中被验证。

- 可以用 *Branch-schema* = (*branch-name*, *branch-city*, *assets*) 验证函数依赖：*branch-name* → *branch-city assets*。
- 可以用 *Loan-schema* = (*branch-name*, *loan-number*, *amount*) 验证函数依赖：*loan-number* → *amount branch-name*。

如上所述，不采用图 7-5 中的算法判定保持依赖性往往会更简单，因为算法中第一步计算 F^+是指数时间的。

3. 信息重复

Lending-schema 的分解不存在 7.1 节所说的信息重复的问题。在 *Lending-schema* 中，必须为每笔贷款重复存放分支机构所在城市及资产额。而分解将分支机构和贷款数据分到不同的关系中，于是消除了冗余。同样，如果一笔贷款贷给了几个客户，必须为每个客户重复存放贷款额（类似于分支机构的所在城市及资产额）。在分解中，*Borrower-schema* 上的关系包含 *loan-number* 和 *customer-name* 的联系，而其他模式则不含该联系。因此，对应于贷款的每个客户，

在 *Borrower-schema* 上的关系中都有一个元组。而在涉及 *loan-number* 的其他关系（*Loan-schema* 和 *Borrower-schema* 上的关系）中，每笔贷款只需有一个元组。

显然，没有冗余的分解是我们所希望的。消除冗余的程度可由几种范式表示，这将在本章其余部分讨论。

7.3.2 Boyce-Codd 范式

我们所知的较令人满意的范式之一是 *Boyce-Codd* 范式（BCNF）。如果对 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖，其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$，下面至少有一个成立：

- $\alpha \rightarrow \beta$ 是平凡函数依赖（即，$\beta \subset \alpha$）。
- α 是模式 R 的超码。

则具有函数依赖集 F 的关系模式 R 属于 BCNF。此外，如果关系模式集中的每个模式都属于 BCNF，则这个数据库设计属于 BCNF。

例如，考虑如下关系模式及其相应的函数依赖：

- *Customer-schema* = (*customer-name*, *customer-street*, *customer-city*)
  customer-name → *customer-street customer-city*
- *Branch-schema* = (*branch-name*, *assets*, *branch-city*)
  branch-name → *assets branch-city*
- *Loan-info-schema* = (*branch-name*, *customer-name*, *loan-number*, *amount*)
  loan-number → *amount branch-name*

Customer-schema 属于 BCNF。*customer-name* 是该模式的一个候选码。*Customer-schema* 上仅有的非平凡函数依赖在箭头左侧是 *customer-name*。由于 *customer-name* 是候选码，因而左半部为 *customer-name* 的函数依赖不破坏 BCNF 的定义。同理，易证明关系模式 *Branch-schema* 属于 BCNF。

但是，模式 *Loan-info-schema* 不属于 BCNF。首先，注意 *loan-number* 不是 *Loan-info-schema* 的超码，因为可以有两个元组表示贷给两个人的一笔贷款，如，

(Downtown, Mr.Bell, L-44, 1000)
(Downtown, Ms.Bell, L-44, 1000)

因为没有函数依赖可以排除上述情况，所以 *loan-number* 不是候选码。但是，函数依赖 *loan-number* → *amount* 是非平凡的，因此，*Loan-info-schema* 不满足 BCNF 的定义。

Loan info-schema 不是一个良好的设计，因为它有 7.1 节所讲的信息重复的问题。如果一笔贷款对应几个客户名，那么在 *Loan-info-schema* 的关系中就必须为这几个客户重复分支机构名和金额。为消除冗余，可以重新设计数据库，使所有模式都属于 BCNF。办法之一是以当前的非 BCNF 设计为起点，分解不属于 BCNF 的模式。例如将 *Loan-info-schema* 分解成两个模式：

Loan-schema = (*branch-name*, *loan-number*, *amount*)
Borrower-schema = (*customer-name*, *loan-number*)

该分解是无损连接分解。

要判定模式是否属于 BCNF，需要确定其上有哪些函数依赖。本例容易看到 *Loan-schema* 上有

$$loan\text{-}number \rightarrow amount\ branch\text{-}name$$

Borrower-schema 上只有平凡的函数依赖。尽管 *loan-number* 不是 *Loan-info-schema* 的超码，它却是 *Loan-schema* 的候选码。所以，分解后的两个模式都属于 BCNF。

现在可以避免在一笔贷款上有几个客户时的冗余。对应于每笔贷款，在 *Loan-schema* 上的关系中恰有一个元组，对应于每笔贷款的每个客户，在 *Borrower-schema* 上的关系中恰有一个元组。所以，不必为每笔贷款的每个客户都重复分支机构名和金额。

我们现在能给出生成一个 BCNF 模式集的一个通用方法。若 R 不属于 BCNF，可用图 7-6 中算法将 R 分解成一组 BCNF 模式 R_1，R_2，…，R_n，该算法不仅产生一个 BCNF 分解，而且还是无损连接分解。之所以算法只产生无损连接分解，是因为当用 $(R_i - \beta)$ 和 (α, β) 替换模式 R_i 时，有依赖 $\alpha \rightarrow \beta$ 成立且 $(R_i - \beta) \cap (\alpha, \beta) = \alpha$。

```
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 是 Ri 上的一个非平凡的函数依赖，满足 α→Ri 不在 F+
            中且 α ∩ β =∅;
            result := (result − Ri) ∪ (Ri − β) ∪ (α, β);
          end
     else done := true;
```

图 7-6　BCNF 分解算法

7.1 节中的 *Lending-schema* 是不好的数据库设计的例子，让我们将 BCNF 分解算法用于该模式：

$$Lending\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name,\ loan\text{-}number,\ amount)$$

Lending-schema 上要求满足的函数依赖集是

$$branch\text{-}name \rightarrow assets\ branch\text{-}city$$
$$loan\text{-}number \rightarrow amount\ branch\text{-}name$$

该模式的一个候选码是 $\{loan\text{-}number,\ customer\text{-}name\}$。

可以按如下方式将图 7-6 的算法用于 *Lending-schema*：

- *Lending-schema* 上有函数依赖

$$branch\text{-}name \rightarrow assets\ branch\text{-}city$$

但 *branch-name* 不是超码，因此，*Lending-schema* 不属于 BCNF。将 *Lending-schema* 替换为

$$Branch\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets)$$
$$Loan\text{-}info\text{-}schema = (branch\text{-}name,\ customer\text{-}name,\ loan\text{-}number,\ amount)$$

• *Branch-schema* 上的非平凡函数依赖的左边都是 *branch-name*。由于 *branch-name* 是 *Branch-schema* 的码，因而关系模式 *Branch-schema* 属于 BCNF。

• *Loan-info-schema* 上有函数依赖

$$loan\text{-}number \rightarrow amount\ branch\text{-}name$$

但 *loan-number* 不是 *Loan-info-schema* 的码。用模式

$$Loan\text{-}schema = (branch\text{-}name,\ loan\text{-}number,\ amount)$$
$$Borrower\text{-}schema = (customer\text{-}name,\ loan\text{-}number)$$

代替 *Loan-info-schema*。

• *Loan-schema* 和 *Borrower-schema* 属于 BCNF。

这样，*Lending-schema* 分解成三个关系模式 *Branch-schema*、*Loan-schema*、*Borrower-schema*，它们都属于 BCNF。这些关系模式与 7.3.1 节中的相同，该节指出该分解既是无损连接分解，又是保持依赖的分解。

不是每个 BCNF 分解都是保持函数依赖的。例如，考虑关系模式

$$Banker\text{-}schema = (branch\text{-}name,\ customer\text{-}name,\ banker\text{-}name)$$

它表示的是一个客户在某一分支机构有一个“银行帐户负责人”。*Banker-schema* 上要求满足的函数依赖集 F 为

$$banker\text{-}name \rightarrow branch\text{-}name$$
$$branch\text{-}name\ customer\text{-}name \rightarrow banker\text{-}name$$

显然，*Banker-schema* 不属于 BCNF，因为 *banker-name* 不是超码。

如果运用图 7-6 的算法，会得到如下 BCNF 分解：

$$Banker\text{-}branch\text{-}schema = (banker\text{-}name,\ branch\text{-}name)$$
$$Customer\text{-}banker\text{-}schema = (customer\text{-}name,\ banker\text{-}name)$$

分解后的模式只保持依赖 *banker-name* → *branch-name*（及平凡依赖），但 $\{banker\text{-}name \rightarrow branch\text{-}name\}$ 的闭包不包含 *customer-name branch-name* → *banker-name*。只有计算连接后才能检测该依赖是否被破坏。

可以用图 7-5 的算法证明将 *Banker-schema* 分解成 *Banker-branch-schema* 和 *Customer-banker-schema* 不是保持依赖的。F 对两个模式的投影 F_1 和 F_2 如下（为了简洁起见，我们只给出正则覆盖）：

$$F_1 = \{banker\text{-}name \rightarrow branch\text{-}name\}$$
$$F_2 = \varnothing\ (Customer\text{-}banker\text{-}schema\ 上只有平凡依赖)$$

可见，集合 F' 的正则覆盖是 F_1。

易证函数依赖 *customer-name branch-name* → *banker-name* 属于 F^+，但不属于 F'^+。因此，

$F_1^+ \neq F^+$，分解不是保持依赖的。

上面的例子说明，不是每个 BCNF 分解都是保持依赖的，同时它也说明我们不是总能满足所有以下三个设计目标：

1）BCNF。

2）无损连接。

3）保持函数依赖。

本例中不能满足的原因是 *Banker-schema* 的每个 BCNF 分解都不能保持 *customer-name branch-name*→*banker-name*。

7.3.3　第三范式

当不能满足所有三个设计目标时，可以放弃 BCNF 而接受相对较弱的、称为第三范式（3NF）的范式。3NF 中总能找到无损连接并保持依赖的分解。

BCNF 要求所有形如 $\alpha \rightarrow \beta$ 的非平凡函数依赖中，α 为超码。3NF 稍微放松了这个约束，允许非平凡函数依赖的左边不是超码。

具有函数依赖集 F 的关系模式 R 属于 3NF，只要 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖，其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$，至少有以下之一成立：

- $\alpha \rightarrow \beta$ 是平凡函数依赖。
- α 是 R 的超码。
- $\beta - \alpha$ 中的每个属性 A 都包含在 R 的候选码中。

3NF 的定义允许某些 BCNF 中不允许的函数依赖出现。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，这些依赖是传递依赖的例子（参见习题 7.13）。

注意，如果关系模式属于 BCNF，则所有函数依赖要么形如“超码确定属性集”，要么是平凡的。因此，BCNF 模式不能有任何传递依赖，所以，每个 BCNF 模式一定属于 3NF，BCNF 也就自然比 3NF 约束更严格。

回到 *Banker-schema* 的例子（7.3.2 节）。我们已经看到没有能将该关系模式转化成 BCNF 而又保持依赖和无损连接的分解，但该模式属于 3NF。要说明这一点，首先应注意到{*customer-name*，*branch-name*} 是 *Banker-schema* 的候选码，所以 *Banker-schema* 上不包含在候选码中的属性只有 *banker-name*。而形如

$$\alpha \rightarrow banker\text{-}name$$

的非平凡函数依赖都以{*customer-name*，*branch-name*} 作为 α 的一部分。由于{*customer-name*，*branch-name*} 是候选码，这些依赖不会破坏 3NF 的定义。

图 7-7 给出了找出将模式转化为 3NF 且保持依赖和无损连接的分解的算法。该算法中使用的依赖集 F_C 是 F 的正则覆盖，也即正则形式（6.5.4 节），因此，可证明每个关系模式 R_i 都属于 3NF。文献注解中给出了关于证明的参考文献。通过显式地为每一个给定依赖构造一个模式，该算法确保保持依赖。通过使至少有一个模式含被分解模式的候选码，该算法还保证分解是无损连接的。习题 7.6 对为什么这样就能保证无损连接性的证明给出了提示。

为说明图 7-7 的算法，将 7.3.2 节介绍的 *Banker-schema* 做如下扩展：

$$Banker\text{-}info\text{-}schema = (branch\text{-}name,\ customer\text{-}name,\ banker\text{-}name,$$

office-number)

扩展前后的主要区别在于这里将银行帐户负责人的办公室号码也作为信息的一部分。该关系模式的函数依赖是：

$$banker\text{-}name \rightarrow branch\text{-}name\ office\text{-}number$$
$$customer\text{-}name\ branch\text{-}name \rightarrow banker\text{-}name$$

图 7-7 算法中的 for 循环使我们在分解中包含如下模式：

$$Banker\text{-}office\text{-}schema = (banker\text{-}name,\ branch\text{-}name,\ office\text{-}number)$$
$$Banker\text{-}schema = (customer\text{-}name,\ branch\text{-}name,\ banker\text{-}name)$$

由于 *Banker-schema* 包含了 *Banker-info-schema* 的一个候选码，这样就完成了分解过程。

```
令 F_c 是 F 的一个正则覆盖;
i := 0;
for each F_c 中的函数依赖 α → β do
   if R_j (j = 1, 2, ..., i)中没有一个包含 αβ
     then begin
            i := i + 1;
            R_i := αβ;
          end
if R_j (j = 1, 2, ..., i)中没有一个包含R的候选码
   then begin
          i := i + 1;
          R_i := R的任一候选码;
        end
return (R_1, R_2, ..., R_i)
```

图 7-7　将模式转化为 3NF 的保持依赖且无损连接的分解

7.3.4　BCNF 和 3NF 的比较

我们讨论了两种关系数据库模式的范式：3NF 和 BCNF。3NF 有一个优点，即总可以在满足无损连接并保持依赖的前提下得到 3NF 设计。但 3NF 也有一个缺点，即如果没有消除所有的传递依赖，必须用空值表示数据项间的某些可能有意义的联系。此外，3NF 还存在信息重复的问题。

例如，考虑 *Banker-schema* 及其相关的函数依赖。因为有 *banker-name* → *branch-name*，可能需要表示 *banker-name* 和 *branch-name* 之间的联系。但如果需要这样做，则要么必须有相应的 *customer-name* 值，要么 *customer-name* 为空值。

Banker-schema 的另一个问题是信息重复。例如，考虑图 7-8 中 *Banker-schema* 的实例。注意表示 Johnson 在 Perryridge 分支机构工作的信息是重复的。

如果必须在 BCNF 和保持依赖的 3NF 间选择的话，通常倾向于选择 3NF。如果不能有效地检验依赖的保持情况，我们或者要牺牲系统性能，或者会破坏数据库中数据的完整性，这当然都不好。相形之下，3NF 中允许传递依赖造成少量冗余反倒可以容忍了。所以，我们常选择保持依赖，而放弃 BCNF。

总之，下面再重申一下关系数据库设计的三个目标：

customer-name	*banker-name*	*branch-name*
Jones	Johnson	Perryridge
Smith	Johnson	Perryridge
Hayes	Johnson	Perryridge
Jackson	Johnson	Perryridge
Curry	Johnson	Perryridge
Turner	Johnson	Perryridge

图 7-8　*Banker-schema* 的一个实例

1）BCNF。

2）无损连接。

3）保持依赖。

如果不能达到所有这三个目标，可以接受

1）3NF。

2）无损连接。

3）保持依赖。

7.4　利用多值依赖作规范化

有的关系模式属于 BCNF，但仍存在信息重复的问题，从这一意义上看它们还没有被充分规范化。再看银行的例子。假设在另一个银行数据库模式设计中，有模式

$$BC\text{-}schema = (loan\text{-}number,\ customer\text{-}name,\ customer\text{-}street,\ customer\text{-}city)$$

读者一定发现这个模式不是 BCNF 模式，因为在函数依赖

$$customer\text{-}name \rightarrow customer\text{-}street\ customer\text{-}city$$

中，*customer-name* 不是 *BC-schema* 的码。但是，假设银行吸引了一些有钱的客户，他们有多个地址（比如，有一幢冬天住的房子和一幢夏天住的房子）。这样，就不再需要函数依赖 *customer-name*→*customer-street customer-city*。如果去掉这个函数依赖，*BC-schema* 则在修改后的函数依赖集下属于 BCNF，尽管其中仍有前面提到的信息重复问题。

为解决这个问题，必须定义一种新的约束，称为多值依赖。与利用函数依赖定义范式一样，我们将利用多值依赖定义关系模式的范式。这一范式称为*第四范式*（4NF），它比 BCNF 的约束更严格。下面会看到每个 4NF 模式都是 BCNF，但 BCNF 模式不一定是 4NF。

7.4.1　多值依赖

函数依赖规定了某些元组不能出现在关系中。例如，如果 $A \rightarrow B$，就不能有两个元组在 A 上的值相同而在 B 上的值不同。多值依赖则不是排除某些元组的存在，而是要求某种形式的其他元组必须在关系中。因此，有时函数依赖称为相等产生依赖，而多值依赖称为元组产生依赖。

令 R 为一关系模式，$\alpha \subseteq R$ 且 $\beta \subseteq R$。R 上有多值依赖

$$\alpha \twoheadrightarrow \beta$$

是指在任意合法关系 $r(R)$ 中，对 r 中任一元组对 t_1 和 t_2，若 $t_1[\alpha] = t_2[\alpha]$，则 r 中都

存在元组 t_3 和 t_4，使得

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

这个定义看似复杂，实则不然。图 7-9 以图表的形式给出了 t_1、t_2、t_3 和 t_4。直观地，多值依赖 $\alpha \rightarrow\rightarrow \beta$ 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 $\alpha \rightarrow\rightarrow \beta$，则 $\alpha \rightarrow\rightarrow \beta$ 是模式 R 上平凡的多值依赖，即，如果 $\beta \subseteq \alpha$ 或 $\beta \cup \alpha = R$，则 $\alpha \rightarrow\rightarrow \beta$ 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 7-9　用表格表示 $\alpha \rightarrow\rightarrow \beta$

为说明函数依赖和多值依赖的区别，再考虑 *BC-schema* 及图 7-10 中的关系 *bc*（*BC-schema*）。我们必须为客户的每个地址重复贷款号，并且必须为客户的每笔贷款重复地址。这种重复是不必要的，因为客户与其地址间的联系独立于客户与其贷款间的联系。如果一个客户（如 Smith）有一笔贷款（贷款号为 L-23），我们需要将这笔贷款与 Smith 的所有地址关联起来。因此，图 7-11 的关系是非法的。要使该关系合法化，需要向图 7-11 的关系 *bc* 中加入元组（L-23，Smith，Main，Manchester）及（L-27，Smith，North，Rye）。

loan-number	*customer-name*	*customer-street*	*customer-city*
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

图 7-10　关系 *bc*：BCNF 关系中有冗余的例子

loan-number	*customer-name*	*customer-street*	*customer-city*
L-23	Smith	North	Rye
L-27	Smith	Main	Manchester

图 7-11　一个非法的 *bc* 关系

将这个例子与多值依赖的定义相比较，这里需要多值依赖

customer-name $\rightarrow\rightarrow$ *customer-street customer-city*

(多值依赖 *customer-name*→→*loan-number* 也可以，下面会看到它们是等价的。)

与函数依赖相同，可以用两种方式使用多值依赖：

1) 验证关系是否满足给定的函数依赖集或多值依赖集。

2) 定义合法关系集上的约束。我们将只考虑满足给定函数依赖集和多值依赖集的关系。注意，若关系 r 不满足给定的多值依赖，可以通过向 r 中增加元组来构造满足多值依赖的关系 r'。

7.4.2 多值依赖的理论

与函数依赖以及 3NF 和 BCNF 一样，需要确定给定多值依赖集逻辑蕴涵的所有多值依赖。

我们用与前面处理函数依赖相同的方法。令 D 表示函数依赖和多值依赖的集合。D 的闭包 D^+ 是 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，可以用函数依赖和多值依赖的形式化定义由 D 计算 D^+。但是，用推理规则系统常常更易用来推理依赖集。

下面列出的函数依赖和多值依赖的推理规则是保真的和完备的。前面提到过，保真的规则不会产生不被 D 逻辑蕴涵的依赖；完备的规则可以产生 D 中的所有依赖。下面前三个规则是第 6 章中曾见过的 Armstrong 公理。

1) 自反律。若 α 是属性集且 $\beta \subseteq \alpha$，则有 $\alpha \rightarrow \beta$。

2) 增补律。若有 $\alpha \rightarrow \beta$ 且 γ 为一属性集，则有 $\gamma\alpha \rightarrow \gamma\beta$。

3) 传递律。若有 $\alpha \rightarrow \beta$ 且 $\beta \rightarrow \gamma$，则有 $\alpha \rightarrow \gamma$。

4) 补充律。若有 $\alpha \rightarrow\!\!\!\rightarrow \beta$，则有 $\alpha \rightarrow\!\!\!\rightarrow R - \beta - \alpha$。

5) 多值增补律。若有 $\alpha \rightarrow\!\!\!\rightarrow \beta$，$\gamma \subseteq R$ 且 $\delta \subseteq \gamma$，则有 $\gamma\alpha \rightarrow\!\!\!\rightarrow \delta\beta$。

6) 多值传递律。若有 $\alpha \rightarrow\!\!\!\rightarrow \beta$ 且 $\beta \rightarrow\!\!\!\rightarrow \gamma$，则 $\alpha \rightarrow\!\!\!\rightarrow \gamma - \beta$。

7) 复制律。若有 $\alpha \rightarrow \beta$，则有 $\alpha \rightarrow\!\!\!\rightarrow \beta$。

8) 联合律。若有 $\alpha \rightarrow\!\!\!\rightarrow \beta$，$\gamma \subseteq \beta$，且存在 δ 使得 $\delta \subseteq R$，$\delta \cap \beta = \varnothing$ 且 $\delta \rightarrow \gamma$，则有 $\alpha \rightarrow \gamma$。

文献注解中给出了有关这些规则的保真性和完备性证明的参考文献。下面的例子对如何进行形式化证明作了解释。

令 $R = (A, B, C, G, H, I)$ 为一关系模式，假设有 $A \rightarrow\!\!\!\rightarrow BC$。多值依赖的定义指出，若 $t_1[A] = t_2[A]$，则存在元组 t_3 和 t_4，使得

$$
\begin{aligned}
t_1[A] &= t_2[A] = t_3[A] = t_4[A] \\
t_3[BC] &= t_1[BC] \\
t_3[GHI] &= t_2[GHI] \\
t_4[GHI] &= t_1[GHI] \\
t_4[BC] &= t_2[BC]
\end{aligned}
$$

补充律指出，若 $A \rightarrow\!\!\!\rightarrow BC$，则 $A \rightarrow\!\!\!\rightarrow GHI$。如果修改一下下标就可以看出，$t_3$ 和 t_4 满足 $A \rightarrow\!\!\!\rightarrow GHI$ 的定义。

通过使用多值依赖的定义可对规则 5) 和规则 6) 给出类似的理由 (参见习题 7.17)。

规则 7)，即复制律，涉及了函数依赖和多值依赖。假设 R 上有 $A \rightarrow BC$。若 $t_1[A] = t_2[A]$ 且 $t_1[BC] = t_2[BC]$，则 t_1 和 t_2 本身就可以是多值依赖 $A \rightarrow\!\!\!\rightarrow BC$ 定义中的元组 t_3 和 t_4。

规则 8)，即联合律，是八个规则中最难证明的 (参见习题 7.19)。

可以用如下规则简化 D 的闭包运算，这些规则可由规则 1）～8）证明得到（参见习题 7.20）。

- 多值合并律：若有 $\alpha \twoheadrightarrow \beta$ 且 $\alpha \twoheadrightarrow \gamma$，则有 $\alpha \twoheadrightarrow \beta\gamma$。
- 取交律：若有 $\alpha \twoheadrightarrow \beta$ 且 $\alpha \twoheadrightarrow \gamma$，则有 $\alpha \twoheadrightarrow \beta \cap \gamma$。
- 取差律：若有 $\alpha \twoheadrightarrow \beta$ 且 $\alpha \twoheadrightarrow \gamma$，则有 $\alpha \twoheadrightarrow \beta - \gamma$ 和 $\alpha \twoheadrightarrow \gamma - \beta$。

让我们对下面的例子使用这些规则。令 $R = (A, B, C, G, H, I)$，给定如下依赖集 D：

$$A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \rightarrow H$$

我们列出 D^+ 中的几个依赖：

- $A \twoheadrightarrow CGHI$。因为 $A \twoheadrightarrow B$，由补充律（规则 4））可推出 $A \twoheadrightarrow R - B - A$，$R\text{-}B\text{-}A = CGHI$，所以 $A \twoheadrightarrow CGHI$。
- $A \twoheadrightarrow HI$。因为 $A \twoheadrightarrow B$ 且 $B \twoheadrightarrow HI$，由多值传递律（规则 6））可推出 $A \twoheadrightarrow HI - B$。因为 $HI - B = HI$，$A \twoheadrightarrow HI$。
- $B \rightarrow H$。证明它成立，可使用联合律（规则 8））。我们有 $B \twoheadrightarrow HI$。又因为 $H \subseteq HI$，$CG \rightarrow H$ 且 $CG \cap HI = \varnothing$ 可见满足联合律的前提，其中 α 是 B，β 是 HI，δ 是 CG，γ 是 H，由此可得出 $B \rightarrow H$。
- $A \twoheadrightarrow CG$。我们已知 $A \twoheadrightarrow CGHI$ 且 $A \twoheadrightarrow HI$。由取差律，有 $A \twoheadrightarrow CGHI - HI$；由于 $CGHI - HI = CG$，所以 $A \twoheadrightarrow CG$。

7.4.3　第四范式

回到 *BC-schema* 的例子，其上有多值依赖 *customer-name* $\twoheadrightarrow$ *customer-street customer-city*，但没有非平凡的函数依赖。前面可知，尽管 *BC-schema* 属于 BCNF，但这个设计并不是完美的，因为必须为每笔贷款重复客户的地址信息。通过将 *BC-schema* 分解为第四范式（4NF），可以用给定的多值依赖改进数据库的设计。

如果对 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

- $\alpha \twoheadrightarrow \beta$ 是平凡的多值依赖。
- α 是模式 R 的超码。

则函数和多值依赖集为 D 的关系模式 R 属于 4NF。如果构成设计的关系模式集中的每个模式都属于 4NF，则数据库设计属于 4NF。

注意，4NF 定义与 BCNF 定义的唯一不同是用多值依赖替代了函数依赖。4NF 模式一定属于 BCNF。因为如果模式 R 不属于 BCNF，则 R 上有非平凡的函数依赖 $\alpha \rightarrow \beta$ 且 α 不是超码。由于 $\alpha \rightarrow \beta$ 蕴涵 $\alpha \twoheadrightarrow \beta$（由复制律），因而 R 不属于 4NF。

令 R 为关系模式，R_1，R_2，…，R_n 为 R 的分解。前面曾讲到对于函数依赖集 F，F 在 R_i 上的投影 F_i 是 F^+ 中所有只含 R_i 中属性的函数依赖。现在考虑函数和多值依赖集 D。D 在 R_i 上的投影是集合 D_i，它包含

- D^+ 中所有只含 R_i 中属性的函数依赖。
- 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^+。

利用 4NF 与 BCNF 的相似之处，我们可以将模式分解为 4NF 的设计算法。图 7-12 给出了 4NF 分解算法。它与图 7-6 的 BCNF 分解算法相似，只不过用多值依赖代替了函数依赖，并使用了 D^+ 在 R_i 上的投影。

```
result := {R};
done := false;
    计算D+;已知模式 Ri,令Di表示 D+ 在 Ri 上的投影
while (not done) do
  if (result中存在模式 Ri,它相对于Di不属于4NF)
    then begin
          令α →→β 是Ri上的一个非平凡的多值依赖，满足α → Ri不在Di
          中，且α ∩ β = ∅;
           result := (result − Ri) ∪ (Ri − β) ∪ (α, β);
        end
    else done := true;
```

图 7-12　4NF 分解算法

如果将图 7-12 的算法用于 *BC-schema*，*customer-name*→→*loan-number* 是非平凡的多值依赖，并且 *customer-name* 不是 *BC-schema* 的超码。按照该算法，将 *BC-schema* 替换为两个模式：

$$Borrower\text{-}schema = (customer\text{-}name,\ loan\text{-}number)$$
$$Customer\text{-}schema = (customer\text{-}name,\ customer\text{-}street,\ customer\text{-}city)$$

这两个模式属于 4NF，它们消除了前面遇到的 *BC-schema* 的冗余问题。

与只考虑函数依赖时一样，我们关心分解的无损连接性和保持依赖性。从下面关于多值依赖和无损连接的性质可以看出图 7-12 的算法只产生无损连接分解：

令 R 为一关系模式，D 为 R 上的函数和多值依赖集。令 R_1 和 R_2 为 R 的一个分解，该分解是 R 的无损连接分解，当且仅当下面的多值依赖中至少有一个属于 D^+：

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

前面曾提到，若 $R_1 \cap R_2 \rightarrow R_1$ 或 $R_1 \cap R_2 \rightarrow R_2$，则 R_1 和 R_2 为 R 的无损连接分解。上面关于多值依赖的性质是有关无损连接的更一般化陈述。它指明，将 R 分解为两个模式 R_1 和 R_2 的任何一个无损连接分解，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。

多值依赖中保持依赖的问题不像只有函数依赖时那么简单。如果对于每一个关系集 r_1 (R_1)，r_2 (R_2)，…，r_n (R_n)，其中对任意 i，r_i 满足 D_i（D 在 R_i 上的投影），都存在关系 r (R) 满足 D 且对所有 i，有 $r_i = \Pi_{R_i}(r)$，则将模式 R 分解为模式 R_1，R_2，…，R_n 对于函数和多值依赖集 D 是保持依赖的分解。

让我们对例子 $R = (A, B, C, G, H, I)$ 使用图 7-12 的 4NF 分解算法，此模式上 $D = \{A \rightarrow\rightarrow B,\ B \rightarrow\rightarrow HI,\ CG \rightarrow H\}$，然后看看分解的结果是否保持依赖。

R 不属于 4NF。因为 $A \rightarrow\rightarrow B$ 非平凡，而 A 不是超码。在 while 循环的第一轮利用 $A \rightarrow\rightarrow B$，我们用两个模式 (A, B) 和 (A, C, G, H, I) 替换了 R。容易看到 (A, B) 属于 4NF，因为 (A, B) 上的所有多值依赖都是平凡的。但是，模式 (A, C, G, H, I) 不属于 4NF。利用多值依赖 $CG \rightarrow\rightarrow H$（由给定的函数依赖 $CG \rightarrow H$ 通过复制律得到），我们用两个

模式 (C, G, H) 和 (A, C, G, I) 替换 (A, C, G, H, I)。模式 (C, G, H) 属于 4NF，但模式 (A, C, G, I) 又不属于 4NF。要明白 (A, C, G, I) 为什么不属于 4NF，别忘了前面曾提到过 $A \twoheadrightarrow HI$ 属于 D^+。因此，$A \twoheadrightarrow I$ 是 D 在 (A, C, G, I) 上的投影。所以，在 while 循环的第三轮，我们用两个模式 (A, I) 和 (A, C, G) 替换 (A, C, G, I)。这时算法终止，4NF 分解的结果是 $\{(A, B), (C, G, H), (A, I), (A, C, G)\}$。

该 4NF 分解不是保持依赖的，因为它不能保持多值依赖 $B \twoheadrightarrow HI$。考虑图 7-13，它给出了由 (A, B, C, G, H, I) 上关系投影到分解后的四个模式得到的四个关系。D 在 (A, B) 上的投影是 $A \twoheadrightarrow B$ 和一些平凡依赖。容易看到 r_1 满足 $A \twoheadrightarrow B$，因为没有两个元组在 A 上有相同值。r_2 满足所有的函数和多值依赖，因为在任一属性上 r_2 中没有两个元组具有相同值。r_3 和 r_4 的情况也是这样。因此，这样分解数据库能够满足 D 的投影中的所有依赖。但是，没有 (A, B, C, G, H, I) 上的关系 r 能满足 D 并且分解为 r_1、r_2、r_3 和 r_4。图 7-14 给出了关系 $r = r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$。关系 r 不满足 $B \twoheadrightarrow HI$。任何包含 r 且满足 $B \twoheadrightarrow HI$ 的关系 s 必然包含元组 $(a_2, b_1, c_2, g_2, h_1, i_1)$。但是，$\Pi_{CGH}(s)$ 包含元组 (c_2, g_2, h_1)，而 r_2 中不包含。因此，分解无法检测是否破坏了 $B \twoheadrightarrow HI$。

r_1:

A	B
a_1	b_1
a_2	b_1

r_2:

C	G	H
c_1	g_1	h_1
c_2	g_2	h_2

r_3:

A	I
a_1	i_1
a_2	i_2

r_4:

A	C	G
a_1	c_1	g_1
a_2	c_2	g_2

图 7-13　将关系 r 投影到 R 的一个 4NF 分解上

A	B	C	G	H	I
a_1	b_1	c_1	g_1	h_1	i_1
a_2	b_1	c_2	g_2	h_2	i_2

图 7-14　不满足 $B \twoheadrightarrow HI$ 的关系 r (R)

我们已经看到，如果给定一个多值和函数依赖集，数据库设计最好能满足以下三个准则

1）4NF。

2）保持依赖。

3）无损连接。

如果只有函数依赖，那么第一条准则就是 BCNF。

我们也已经看到这三条准则并不是总能满足的。我们成功地为银行的例子找到了这样的分解，但在模式 $R = (A, B, C, G, H, I)$ 的例子中却失败了。

当不能同时达到这三个目标时，只好在必要时放弃 4NF，而接受 BCNF 甚至 3NF 以保证保持依赖性。

7.5　利用连接依赖作规范化

我们已经看到，无损连接性是良好的数据库设计的几个性质之一。这个性质的确很重要，没有它就会丢失信息。当我们将合法关系集限定为满足函数和多值依赖集的关系时，可以用这些依赖指明某些分解是无损连接分解。

由于无损连接概念的重要性，有时需要将模式 R 上的合法关系集限制为满足下面条件的关系：对于给定分解，若将该分解用于该关系，则该分解是无损连接分解。本节将定义这种称

为连接依赖的约束。正如各种依赖引出了各种范式，连接依赖也引出了一种范式，称为投影连接范式（*PJNF*）。

7.5.1 连接依赖

令 R 为一关系模式，R_1，R_2，…，R_n 为 R 的分解。连接依赖 $*(R_1, R_2, \cdots, R_n)$ 用于限制合法关系集以使得 R_1，R_2，…，R_n 为 R 上的无损连接分解。形式化地说，若 $R = R_1 \cup R_2 \cup \cdots \cup R_n$，且 $r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \cdots \bowtie \Pi_{R_n}(r)$，则我们称关系 $r(R)$ 满足连接依赖 $*(R_1, R_2, \cdots, R_n)$。如果某个 R_i 就是 R 本身，则连接依赖是平凡的。

考虑模式 R 上的连接依赖 $*(R_1, R_2)$。该依赖要求对于所有合法的 $r(R)$，都有

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

令 r 包含如下两个元组 t_1 和 t_2

$$\begin{array}{llll} t_1[R_1 - R_2] & (a_1, a_2, \cdots, a_i) & t_2[R_1 - R_2] = (b_1, b_2, \cdots, b_i) \\ t_1[R_1 \cap R_2] & (a_{i+1}, \cdots, a_j) & t_2[R_1 \cap R_2] = (a_{i+1}, \cdots, a_j) \\ t_1[R_2 - R_1] = (a_{j+1}, \cdots, a_n) & & t_2[R_2 - R_1] = (b_{j+1}, \cdots, b_n) \end{array}$$

于是，$t_1[R_1 \cap R_2] = t_2[R_1 \cap R_2]$，但 t_1 和 t_2 在其他所有属性上的值是不同的。让我们计算 $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$。图 7-15 给出了 $\Pi_{R1}(r)$ 和 $\Pi_{R2}(r)$。计算连接后我们除 t_1 和 t_2 外还得到两个元组 t_3 和 t_4，参见图 7-16。

	$R_1 - R_2$	$R_1 \cap R_2$
$\Pi_{R_1}(t_1)$	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$
$\Pi_{R_1}(t_2)$	$b_1 \ldots b_i$	$a_{i+1} \ldots a_j$

	$R_1 \cap R_2$	$R_2 - R_1$
$\Pi_{R_2}(t_1)$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
$\Pi_{R_2}(t_2)$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$

图 7-15　$\Pi_{R_1}(r)$ 和 $\Pi_{R_2}(r)$

	$R_1 - R_2$	$R_1 \cap R_2$	$R_2 - R_1$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$b_1 \ldots b_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$b_1 \ldots b_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$

图 7-16　$*(R_1, R_2)$ 的表格表示

若 $*(R_1, R_2)$ 成立，则只要有元组 t_1 和 t_2，就一定也有 t_3 和 t_4。因此，图 7-16 以表格的形式给出了连接依赖 $*(R_1, R_2)$。比较图 7-16 和图 7-9，在图 7-9 中以表格的形式给出了 $\alpha \rightarrow\rightarrow \beta$。若令 $\alpha = R_1 \cap R_2$ 且 $\beta = R_1$，则这两个图中的表格就是一样的。实际上，$*(R_1, R_2)$ 就是 $R_1 \cap R_2 \rightarrow\rightarrow R_1$ 的另一种表示方式。利用多值依赖的补充律和增补律，可以证明 $R_1 \cap R_2 \rightarrow\rightarrow R_1$ 蕴涵 $R_1 \cap R_2 \rightarrow\rightarrow R_2$，因此，$*(R_1, R_2)$ 等价于 $R_1 \cap R_2 \rightarrow\rightarrow R_2$。这并不奇怪，因为前面我们已经注意到，$R_1$ 和 R_2 是 R 的无损连接分解，当且仅当 $R_1 \cap R_2 \rightarrow\rightarrow R_2$ 或 $R_1 \cap R_2 \rightarrow\rightarrow R_1$。

因此任何形如 $*(R_1, R_2)$ 的连接依赖都等价于一个多值依赖。但是，也存在与任何多值依赖都不等价的连接依赖。这种依赖最简单的例子是在模式 $R = (A, B, C)$ 上。连接依赖

$$*((A, B), (B, C), (A, C))$$

不等价于任何一组多值依赖。图 7-17 给出了这种连接依赖的表格表示。为说明没有多值依赖集逻辑蕴涵 $*((A, B), (B, C), (A, C))$，将图 7-17 看作图 7-18 中的关系 $r(A, B,$

C)。关系 r 满足连接依赖 $*((A, B), (B, C), (A, C))$，因为计算

$$\Pi_{AB}(r) \bowtie \Pi_{BC}(r) \bowtie \Pi_{AC}(r)$$

得到的结果正好就是 r。但是，r 不满足任何非平凡的多值依赖。因为通过检验发现，r 不满足 $A \twoheadrightarrow B$、$A \twoheadrightarrow C$、$B \twoheadrightarrow A$、$B \twoheadrightarrow C$、$C \twoheadrightarrow A$ 或 $C \twoheadrightarrow B$ 中的任何一个。

A	B	C
a_1	b_1	c_2
a_2	b_1	c_1
a_1	b_2	c_1
a_1	b_1	c_1

图 7-17　$*((A, B), (B, C), (A, C))$ 的表格表示

A	B	C
a_1	b_1	c_2
a_2	b_1	c_1
a_1	b_2	c_1
a_1	b_1	c_1

图 7-18　关系 r (A, B, C)

正如多值依赖表达了两个联系间的独立性，连接依赖则表达了一个联系集中的各个成员都是独立的。联系的独立性这一概念是我们通常定义关系的方式的自然结果。考虑银行例子中的

$$Loan\text{-}info\text{-}schema = (branch\text{-}name, customer\text{-}name, loan\text{-}number, amount)$$

可以定义关系 *loan-info*（*Loan-info-schema*）为 *Loan-info-schema* 上所有元组的集合，其中

- 贷款号为 *loan-number* 的贷款是由名为 *branch-name* 的分支机构贷出的。
- 贷款号为 *loan-number* 的贷款是贷给名为 *customer-name* 的客户。
- 贷款号为 *loan-number* 的贷款的金额为 *amount*。

关系 *loan-info* 的上述定义是三个谓词的与：一个是关于 *loan-number* 和 *branch-name*，一个是关于 *loan-number* 和 *customer-name*，一个是关于 *loan-number* 和 *amount*。我们惊讶地发现，上面关于 *loan-info* 的直观定义逻辑蕴涵连接依赖 $*((loan\text{-}number, branch\text{-}name), (loan\text{-}number, customer\text{-}name), (loan\text{-}number, amount))$。因此，连接依赖很直观，符合良好的数据库设计的三个准则之一。

对于函数依赖和多值依赖，我们能给出保真且完备的推理规则系统。然而，对于连接依赖则没有这样的规则集。看起来必须考虑比连接依赖更一般的依赖类型，以构造保真且完备的推理规则集合。文献注解中列出了这方面的研究。

7.5.2　投影 - 连接范式

投影 - 连接范式（PJNF）的定义与 BCNF 和 4NF 类似，只不过用的是连接依赖。如果对于 D^+ 中所有形如 $*(R_1, R_2, \cdots, R_n)$ 的连接依赖，其中 $R_i \subseteq R$ 且 $R = R_1 \cup R_2 \cup \cdots \cup R_n$，下面至少有一个成立：

- $*(R_1, R_2, \cdots, R_n)$ 是平凡的连接依赖。
- 每个 R_i 均为 R 的超码。

则具有函数、多值和连接依赖集 D 的关系模式 R 属于 PJNF。如果设计中的每个关系模式都属于 PJNF，则数据库设计属于 PJNF。在有些关于数据库规范化的文献中，PJNF 称为第五范式（5NF）。

回到银行的例子。我们已经知道连接依赖 $*((loan\text{-}number, branch\text{-}name), (loan\text{-}number, customer\text{-}name), (loan\text{-}number, amount))$，但 *Loan-info-schema* 不属于 PJNF。要使 *Loan-info-*

schema 变成 PJNF，必须将它分解成连接依赖中指出的那三个模式：（*loan-number*，*branch-name*），（*loan-number*，*customer-name*），（*loan-number*，*amount*）。

由于多值依赖也一定是连接依赖，容易看出 PJNF 模式也一定属于 4NF。所以一般来说，我们不一定能找到将给定模式分解为 PJNF 的保持依赖分解。

7.6　域-码范式

前面用到的规范化方法是先定义一种约束（函数、多值或连接依赖），然后用这种约束定义范式。域-码范式（DKNF）是基于以下三个概念：

1）域定义。令 A 为一个属性，dom 为值的集合。域定义 $A \subseteq \text{dom}$ 要求所有元组在 A 上的值均为 dom 中的值。

2）码定义。令 R 为一关系模式，且有 $K \subseteq R$。码定义 *key*（K）要求 K 为模式 R 的超码——即 $K \rightarrow R$。注意，所有的码定义都是函数依赖，但不是所有的函数依赖都是码定义。

3）一般性约束。一般性约束是关于给定模式上所有关系的集合的一个谓词。本章讨论的依赖都是一般性约束的例子。通常，一般性约束是以某种约定形式表示的谓词，如一阶逻辑。

下面给出一个一般性约束的例子，它不是函数、多值或连接依赖。假设所有 *account-number* 中以数字 9 打头的帐户都是特殊的高息帐户，并且其中最小余额为 $2500。将它写成一个一般性约束就是“如果 t［*account-number*］的第一个数字为 9，那么 t［*balance*］$\geqslant$2500”。

域定义和码定义在实际的数据库系统中易于检测。但是，一般性约束检测起来（在时间上和空间上）可能代价非常高。DKNF 数据库设计的目的就是允许只用域约束和码约束来检测一般性约束。

形式化地说，令 D 为关系模式 R 上的域约束集合，K 为码约束集合。令 G 表示 R 的一般性约束。如果 $D \cup K$ 逻辑蕴涵 G，模式 R 属于 DKNF。

让我们再看前面那个关于帐户的一般性约束。该约束意味着数据库设计不属于 DKNF。为构造一个 DKNF 设计，需要用两个模式代替 *Account-schema*：

$$\textit{Regular-acct-schema} = (\textit{branch-name}, \textit{account-number}, \textit{balance})$$
$$\textit{Special-acct-schema} = (\textit{branch-name}, \textit{account-number}, \textit{balance})$$

我们保持了 *Account-schema* 上的所有一般性约束。*Special-acct-schema* 上的域约束要求每个帐户的

- 帐户号以 9 打头。
- 余额大于 2500。

Regular-acct-schema 上的域约束要求帐户号都不以 9 打头。这个设计属于 DKNF，其证明不属于本书范围。

让我们将 DKNF 与已经学过的其他范式比较一下。其他范式中不考虑域约束。我们（隐式地）假设每个属性域为某个无限域，如所有整数的集合或所有字符串的集合。我们允许码定义（因为我们有函数依赖）。对于每个范式，只允许限定形式的一般性约束（函数、多值、或连接依赖的集合）。也就是说，我们可以另一种方式重新定义 PJNF、4NF、BCNF 和 3NF，从而说明它们都是 DKNF 的特例。

下面以 DKNF 的角度重新定义 PJNF。令 $R = (A_1, A_2, \cdots, A_n)$ 为一个关系模式，$dom(A_i)$ 为属性 A_i 的域，所有这些域都是无限的。所有域约束 D 的形式为 $A_i \subseteq dom(A_i)$。

令一般性约束为函数、多值或连接依赖的集合 G。若 F 为 G 中的函数依赖集，令码约束集合 K 为 F^+ 中所有形如 $\alpha \rightarrow R$ 的那些非平凡函数依赖。模式 R 属于 PJNF 当且仅当它关于 D，K 和 G 属于 DKNF。

DKNF 消除了所有插入和删除异常。

DKNF 代表了“最终”的范式，因为它允许任意约束，而不只是依赖，而且能够有效地检测这些约束。当然，如果模式不属于 DKNF，可以通过分解将它转化为 DKNF，但是我们已经知道这种分解并不总是保持依赖的分解。因此，尽管 DKNF 是数据库设计者的目标，但在实际设计中有时不得不放弃。

7.7　数据库设计的其他方法

本节从规范化对实际数据库系统设计的影响这一角度重新考察关系模式的规范化。

我们前面采取的方法是由一个关系模式开始逐步分解。我们的目标之一是选择无损连接分解。为达到无损性，我们假设分解后的数据库中所有关系的连接是有效的。

考虑图 7-19 中的数据库，图中给出了关系 *loan-info* 的 PJNF 分解。在图 7-19 中，假设有这样一种情况，即还没有确定贷款 L-58 的金额，但希望记录该贷款的其他信息。如果计算这些关系的自然连接，那么所有关于 L-58 的元组都消失了。换句话说，没有关系 *loan-info* 符合图 7-19 的关系。计算连接时消失的元组称为悬挂元组（参见 6.2.1 节）。形式化地，令 r_1（R_1），r_2（R_2），…，r_n（R_n）为关系集。关系 r_i 中的元组 t 是悬挂元组，当且 t 不属于关系 Π_{R_i}（$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$）。

branch-name	*loan-number*
Round Hill	L-58

loan-number	*amount*

loan-number	*customer-name*
L-58	Johnson

图 7-19　PJNF 的分解

悬挂元组可能出现在实际数据库应用中。它们代表不完整信息，正如上例中我们希望存储有关仍在协商过程中的贷款信息。关系 $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$ 称为泛关系，因为它包含了 $R_1 \cup R_2 \cup \cdots \cup R_n$ 中的所有属性。

要为图 7-19 中的例子写一个泛关系的唯一方法是在泛关系中引入空值。我们在第 3 章中知道空值是很难处理的。有关空值和泛关系的研究在文献注解中有进一步讨论。由于难以处理空值，我们用分解后的关系表示数据库，而不是用规范化过程中分解的泛关系模式表示。

注意，如果不使用空值，就不能将不完整的信息存入图 7-19 的数据库中。例如，无法存入贷款号，除非我们至少知道如下之一：

- 客户名。
- 分支机构名。
- 贷款额。

因此，特定的分解限定了数据库中可以接受的不完整信息。

前面定义的各种范式从表示不完整信息的角度来说，能够产生良好的数据库设计。再回到图 7-19 的例子，我们不希望存储如下事实：“有一笔贷款（贷款号未知）贷给 Jones，金额为 $100。”由于有

$$loan\text{-}number \rightarrow customer\text{-}name\ amount$$

所以将 *customer-name* 和 *amount* 关联起来的唯一方式就是通过 *loan-number*。如果不知道贷款号，就不能区分这笔贷款和其他贷款。

换句话说，我们不希望存储码属性未知的数据。注意前面定义的各种范式不允许我们存储这种信息，除非使用空值。也就是说，范式允许通过悬挂元组来表示可以接受的不完整信息，但同时又禁止存储不希望的不完整信息。

如果在数据库中允许悬挂元组，可以换一种角度来看数据库设计过程。我们不是分解一个泛关系，而是由给定属性集合成一组范式模式。我们对得到同样的范式感兴趣，不管是使用分解还是使用合成。分解法更易于理解，使用也更广泛。文献注解中列出了有关合成法研究的参考文献。

数据库设计方法另一个问题是泛关系中的属性名必须是唯一的。不能用 *name* 来既指 *customer-name*，又指 *branch-name*，通常最好是像前面那样使用唯一属性名。但是，如果直接定义关系模式，而不是从泛关系的角度，那么对于银行的例子，可能得到如下模式上的关系：

branch-loan（*name*，*number*）
loan-customer（*number*，*name*）
amt（*number*，*amount*）

注意，如果使用上面几个关系，表达式 *branch-loan* ⋈ *loan-customer* 是无意义的。实际上，表达式 *branch-loan* ⋈ *loan-customer* 会找出所有客户名与分支机构名相同的贷款。

但是，在 SQL 这样的语言中，没有自然连接操作。所以，在一个涉及 *branch-loan* 和 *loan-customer* 的查询中，必须用关系名作前缀以消除引用的二义性。在这种情况下，*name* 具有多个角色（分支机构名和客户名）不仅不麻烦而且用起来可能反倒更简单。

通常认为，使用唯一角色假设（即每个属性名在数据库中只有唯一的含义）比在多个角色中反复使用同一个名字更好。当唯一角色假设不成立时，数据库设计者在构造规范化的关系数据库设计时必须格外小心。

7.8　总结

本章给出了良好数据库设计的三个准则：

1）PJNF、BCNF、4NF、或 3NF。

2）无损连接。

3）保持依赖。

我们已看到如何达到这些目标，以及如何在不能满足所有这些准则时做权衡。

为描述这些准则，我们定义了几种数据依赖：

- 函数依赖（在第 5 章中定义）。
- 多值依赖。
- 连接依赖。

我们研究了这些依赖的性质，特别是在一个依赖集能逻辑蕴涵什么依赖这一方面。

DKNF 是完美的范式，但在现实中可能难以达到。DKNF 有一些良好的性质，在好的数据库设计中应该尽可能保持。

我们讨论了 3NF、4NF 等，但没有讨论第一和第二范式。没有讨论第二范式（2NF）的原因是它只在历史上有价值。习题 7.14 只给出它的简单定义，并让读者用它做个练习。尽管我们没有提第一范式，但从第 3 章引入关系模型时我们就假设它成立。如果 R 的所有属性域都是原子的，我们称关系模式 R 属于第一范式（1NF）。如果域的元素是不可分的单元，那么一

个域就是原子的。我们将在第 8 章中进一步讨论非原子的域。

回顾在本章中讨论的问题可以发现，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格的数学基础之上，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。

习题

7.1　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。

7.2　假设我们将模式 $R=(A, B, C, D, E)$ 分解为

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

如果有如下函数依赖集 F：

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

证明该分解是无损连接分解。

7.3　证明习题 7.2 中的模式 R 的如下分解不是无损连接分解：

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

（提示：给出模式 R 上一个关系 r 的例子，使得 $\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$。）

7.4　令 $R_1, R_2, \cdots, R_n$ 为模式 U 的分解。令 $u(U)$ 为一个关系，$r_i = \Pi_{R_i}(u)$。证明

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

7.5　证明习题 7.2 中的分解不是保持依赖的分解。

7.6　如果能保证至少有一个模式包含被分解模式的候选码，证明保持函数依赖的将模式分解为 3NF 中模式的分解是无损连接分解。

（提示：证明原关系在分解后模式上的所有投影的连接不可能含原关系以外的元组。）

7.7　列出关系数据库设计的三个目标，并解释为什么想达到这些目标。

7.8　给出将习题 7.2 的模式 R 转化为 BCNF 的无损连接分解。

7.9　给出一个关系模式 R' 及函数依赖集 F'，使得至少有两个不同的无损连接分解可将 R' 转化为 BCNF。

7.10　在关系数据库设计中，为什么我们有时会选择非 BCNF 设计？

7.11　给出将习题 7.2 的模式 R 转化为 3NF 的无损连接并保持依赖的分解。

7.12　证明如果关系模式属于 BCNF，那么它也属于 3NF。

7.13　令主属性至少为一个在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，并且有 $\beta \rightarrow A$。我们称 A 传递依赖于

α_c 可以重新如下定义 3NF：如果 R 中没有非主属性 A 传递依赖于 R 的一个码，具有函数依赖集 F 的关系模式 R 属于 3NF。证明这个新定义等价于原定义。

7.14　如果存在 α 的真子集 γ 使得 $\gamma \rightarrow \beta$，函数依赖 $\alpha \rightarrow \beta$ 称为部分依赖，同时称 β 部分依赖于 α。关系模式 R 属于第二范式（2NF），如果 R 中的每个属性 A 都满足如下准则之一：

- 它属于某个候选码。
- 它不部分依赖于候选码。

证明 3NF 模式一定属于 2NF。

（提示：证明每个部分依赖都是传递依赖。）

7.15　我们已经知道关系数据库设计的三个目标，有没有可能设计一个属于 2NF 的数据库模式，而不是更高的范式？（2NF 的定义参见习题 7.14。）

7.16　列出图 7-20 的关系满足的所有非平凡的多值依赖。

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

图 7-20　习题 7.16 的关系

7.17　利用多值依赖的定义（7.4 节）证明下列公理是保真的：

(a) 补充律

(b) 多值增补律

(c) 多值传递律

7.18　利用函数依赖和多值依赖的定义（6.5 节和 7.4 节）证明复制律的保真性。

7.19　证明联合律是保真的。

（提示：对两个元组 t_1 和 t_2 其中 $t_1[\alpha] = t_2[\alpha]$ 使用定义 $\alpha \rightarrow\rightarrow \beta$。注意，既然 $\delta \cap \beta - \varnothing$，那么如果两个元组在 $R - \beta$ 上有相同值，则它们在 δ 上有相同值。）

7.20　利用函数依赖和多值依赖的公理证明下列规则是保真的：

(a) 多值合并律

(b) 取交律

(c) 取差律

7.21　令 $R = (A, B, C, D, E)$，M 为如下多值依赖集

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

列出 M^+ 中的非平凡依赖。

7.22　给出将习题 7.21 中的模式 R 转化为 4NF 的无损连接分解。

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

7.24　解释为什么 4NF 比 BCNF 更好。

7.25　给出一个关系模式 R 和依赖集，使得 R 属于 4NF，但不属于 PJNF。

7.26　解释为什么 PJNF 比 4NF 更好。

7.27　用域约束和一般性约束的概念重新书写 4NF 和 BCNF 的定义。

7.28　解释为什么 DKNF 是一个良好的范式，但在实际中却难以达到。

7.29　解释悬挂元组是怎么出现的，以及它们会导致什么问题。

文献注解

关系数据库理论最初是在 Codd [1970] 早期的一篇论文中讨论的。在那篇论文中，Codd 引入了第一、第二和第三范式。

BCNF 是在 Codd [1972a] 中引入的。BCNF 的好处在 Bernstein 和 Goodman [1980b] 中做了讨论。BCNF 分解的多项式时间算法由 Tsou 和 Fischer [1982] 提出。Biskup 等 [1979] 给出了我们所使用的找出转化为 3NF 的无损连接并保持依赖的分解的算法。该算法产生的分解属于 3NF 的证明参见 Ullman [1988]。在 Aho 等 [1979b] 中有无损连接性研究的主要成果。

Zaniolo [1976] 讨论了多值依赖。Beeri 等 [1977] 给出了一组多值依赖的公理，并证明了这些公理是保真的和完备的。我们的公理系统就是基于这些公理。

4NF、PJNF 和 DKNF 等概念分别来自 Fagin [1977]、Fagin [1979] 和 Fagin [1981]。Bernstein [1976] 中讨论了数据库设计的合成法。

连接依赖是由 Rissanen [1979] 引入的。Sciore [1982] 给出了关于一些依赖的公理集，其中也包含了连接依赖。连接依赖除了在 PJNF 中使用外，对泛关系数据库的定义也很重要。Fagin 等 [1982] 介绍了连接依赖和将关系定义成“谓词的与”（参见 7.5.1 节）之间的联系。连接依赖的使用导致了大量有关无环数据库模式的研究。直观地，如果每两个属性之间通过唯一的方式关联，模式是无环的。无环模式的形式定义由 Fagin [1983] 和 Beeri 等 [1983] 中给出。

Maier [1983] 中有对其他依赖的详细讨论。Casanova 等 [1984] 和 Cosmadakis 等 [1990] 中讨论了包含依赖。Sadri 和 Ullman [1982] 中涉及了模板依赖。Furtado [1978]、Mendelzon 和 Maier [1979] 中考察了相互依赖。

Maier [1983] 详细阐述了关系数据库的设计理论。Ullman [1988] 和 Abiteboul 等 [1995] 中对本章的各种依赖和范式做了更理论化的讨论。

第 8 章　面向对象数据库

近年来，一些数据库新应用领域的出现暴露出关系数据模型的局限性，针对这些新的应用领域，人们提出了几个新的数据模型。本章及第 9 章将介绍这些新数据模型的一些基础概念。本章的重点是面向对象模型，它的基础是面向对象程序设计范型，这种程序设计方法最初出现在 Simula 67 语言（一种用来对模拟进行编程的语言）中，以后又出现的 C++ 和 Smalltalk 语言则很快成为广为人知的面向对象程序设计语言。本章我们将介绍面向对象程序设计的一些基本概念，并探讨这些概念在数据库系统中的使用。

8.1　新的数据库应用

数据库系统的宗旨是管理大量信息。数据库早期是从文件管理系统发展而来的，这些系统最初演变成网状数据库（参见附录 A）或层次数据库（参见附录 B），后来发展到关系数据库。这些“老”应用的共同特征如下：

- 结构统一。有大量结构相似的数据项，每个数据项都具有相同的字节数。
- 面向记录。基本的数据项由固定长度的记录组成。
- 数据项小。每条记录都很短，很少超过几百个字节。
- 原子字段。一个记录内的各个字段都很短并且是定长的，字段内部是无结构的，换句话说，符合第一范式（见第 7 章）。

近年来，数据库技术已经被应用到数据处理范围之外的领域，新领域中的应用至少在某一方面不具备上面所列举的特征。这些新的应用包括：

- 计算机辅助设计（CAD）。CAD 数据库存储了与一个工程设计相关的数据，包括所设计物品的各个组件、这些组件之间的相互关系，以及设计的各个先前版本。
- 计算机辅助软件工程（CASE）。CASE 数据库存储了用于辅助软件开发者的一些数据。这些数据包括源代码、软件模块间的依赖关系、变量的定义和使用，以及软件系统的开发历史等。
- 多媒体数据库。多媒体数据库包含图像、空间数据、音频数据、视频数据，以及其他类似的数据。这种类型数据库的出现是由于存储照片和地理数据，以及语音邮件系统、图形系统和视频点播系统的需求。
- 办公信息系统（OIS）。办公自动化包括基于工作台的文档生成和检索工具、维护日程安排的工具，等等。OIS 数据库必须允许对日程、文档和文档内容进行查询。
- 超文本数据库。超文本是经过增强的文本，它带有指向其他文档的链。万维网（WWW）是超文本的一个实例（更准确地说是超媒体的一个实例，因为万维网上的文档可能是多媒体文档）。为了对它进行索引，超文本也可以是有结构的。超文本数据库必须支持基于链的文档检索，以及根据文档结构对它们进行查询的功能。

这些新的数据库应用在 70 年代是没有人曾考虑过的，那时候大多数当今的商业数据库才刚刚开始设计。近年来内存和磁盘容量的增加、中央处理器速度的加快、硬件成本的下降，以及对数据库管理认识的提高，使得这些新的应用成为现实。

关系模型和实体-联系模型并不足以对这些新应用所需要的数据进行建模。此外即便是传统商业应用的建模需求也变得更加复杂，有些要求甚至用关系模型也是难以表达的，例如，今天的商业应用经常必须处理图像数据以及超文本数据库。

本章及第 9 章考虑几种为满足这些应用需求而开发的数据建模概念，我们并不对前述应用的所有特征进行一一探讨，而只使用其中一些作为例子，来说明在银行示例中没有很好演示的那些概念。这里从面向对象范型相关的角度来考虑这些概念，第 9 章将考虑对象-关系范型。

8.2　面向对象数据模型

面向对象数据模型包括几个方面，我们将在下面几节中加以研究。

8.2.1　对象结构

粗略地讲，一个对象对应着 E-R 模型中的一个实体。面向对象范型的基础是将一个对象的相关数据和代码封装为一个单元。在概念上一个对象和系统其余部分的所有交互都要通过消息，因此对象和系统其余部分的接口定义为一个所允许的消息的集合。

一般来讲，一个对象有如下相关内容：

- 一个包含对象数据的变量集合；变量对应于 E R 模型中的属性。
- 一个对象所响应的消息集合；每个消息可能有零个、一个或多个参数。
- 一个方法集合，每个方法是实现一个消息的代码段；一个方法返回一个值作为对消息的响应。

在面向对象背景下，消息一词并不是指计算机网络中一个物理消息的使用，它指的是不考虑特定实现细节情况下对象间请求的传递。引发一个消息有时用来表示发送一个消息给一个对象的动作和相应方法的执行。

可以通过一个银行数据库中的实体 *employee* 来阐明使用这种途径的动机。假定雇员的年薪对于不同的雇员有不同的计算方法，例如经理可能依据银行的业绩来获得奖金，而出纳员则根据他们工作了多少小时得到奖金。可以（概念上）将计算每个雇员薪金的代码封装为一个方法，当响应一个 *annual-salary* 消息时就执行这个方法。

每一个 *employee* 对象都响应 *annual-salary* 消息，但是他们响应的途径不同。通过将如何计算年薪的信息封装到对象内部，所有的雇员对象都提供了相同的接口。由于一个对象所提供的唯一外部接口是对象所响应的消息集合，因此可以做到修改方法和变量的定义时而不影响系统的其余部分。这种修改一个对象的定义而不影响系统其余部分的能力被认为是面向对象程序设计范型的主要优点之一。

对象的方法可以分为只读与更新，只读方法不影响对象中变量的值，反之更新方法则可能改变变量的值。同样，对象所响应的消息也可以根据实现这些消息的方法分成只读与更新。

E-R 模型中一个实体的派生属性在面向对象模型中可以表示为只读消息，例如，实体 *employee* 的派生属性 *employment-length* 可以表示为一个 *employee* 对象的 *employment-length* 消息。实现这个消息的方法可以用当前日期减去雇员的 *start-date* 从而得到雇佣时间。

严格来讲，在面向对象模型中，实体的任何属性都必须表示为相应对象中的一个变量和一对消息，变量用来保存属性的值，一对消息中其中一个用来读取属性值，另外一个则用来更新这个值。例如，实体 *employee* 的属性 *address* 可以表示为：

- 一个 *address* 变量。
- 一个 *get-address* 消息，回答地址是什么。
- 一个 *set-address* 消息，接受一个参数 *new-address*，更新地址内容。

不过为了简单起见，很多面向对象数据模型允许对变量直接进行读取和更新，而不用定义消息去读写它们。

8.2.2 对象类

在数据库中通常有很多相似的对象，我们所说的相似，是指它们响应相同的消息、使用相同的方法、并且有相同名称和类型的变量。对每个这样的对象单独进行定义是很浪费的，因此我们将相似的对象分组形成一个类，每个这样的对象称为类的一个实例，一个类中的所有对象共享一个公共的定义，尽管它们对变量所赋予的值不同。

面向对象数据模型中类的概念对应于 E-R 模型中实体集概念，银行数据库中类的例子有雇员、客户、帐户和贷款等。

下面给出了用伪码写的类 *employee* 的定义，这个定义显示了类的变量和类的对象所响应的消息，处理这些消息的方法在这里并没有给出。

```
class employee {
    /* 变量 */
    string   name;
    string   address;
    date     start-date;
    int      salary;
/* 消息 */
    int      annual-salary ();
    string   get-name ();
    string   get-address ();
    int      set-address (string new-address);
    int      employment-length ();
};
```

以上的定义中，每个 *employee* 类对象都包含如下变量：字符串类型的 *name* 和 *address*、日期类型的 *start-date* 和整数类型的 *salary*。每个对象都响应所列出的五个消息：*annual-salary*、*get-name*、*get-address*、*set-address* 和 *employment-length*。每个消息名前面的类型名表示对消息的应答的类型。同时我们也看到消息 *set-address* 接受一个参数 *new-address*，用来指定地址的新取值。

类的概念类似于抽象数据类型，不过除了具有抽象数据类型的特征之外，类的概念还有另外一些特性。为了表示这些特性，我们将类本身看作一个对象，一个类对象包括：

- 一个集合变量，它的值是该类的所有实例对象所组成的集合。
- 对消息 new 实施一个方法，用以创建类的一个新实例。

我们将在 8.5.2 节继续讨论这些特征。

8.2.3　继承

一个面向对象数据库的模式通常需要很多类，然而经常有些类是相似的。举例来说，假定对于银行应用有一个面向对象数据库，可以认为银行客户类与银行雇员类相似，因为它们都定义了 *name*、*address* 等等变量。不过，也有些变量是雇员类所特有的（如 *salary*）或客户类所特有的（如 *credit-rating*）。我们希望将那些共同的变量定义在同一个地方，而这只有将雇员和客户合并到一个类中才能做到。

为了能直接表示类之间的相似性，需要将类放入一个特殊化层次（ISA 联系）中，就像在第 2 章中为实体 - 联系模型定义的那样。例如，我们说 *employee* 是 *person* 的一个特殊化，因为所有雇员的集合是所有人的集合的子集，换句话说，每一个雇员都是一个人。同样，*customer* 也是 *person* 的一个特殊化。类层次的概念类似于实体 - 联系模型中特殊化的概念，雇员和客户可以表示成由类 *person* 特殊化形成的类，雇员特有的变量和方法在 *employee* 中，客户特有的变量和方法在 *customer* 中，雇员和客户都具有的变量和方法在类 *person* 中。

图 8-1 显示了一个带有特殊化层次的 E-R 图，它表示了在银行例子的运行中所涉及的人员。图 8-2 显示了对应的类层次，层次中的类可以用伪码定义，如图 8-3 。为了简化，我们在其中并没有给出这些类中的方法的定义，一个完整的银行数据库中应该包括这些定义。

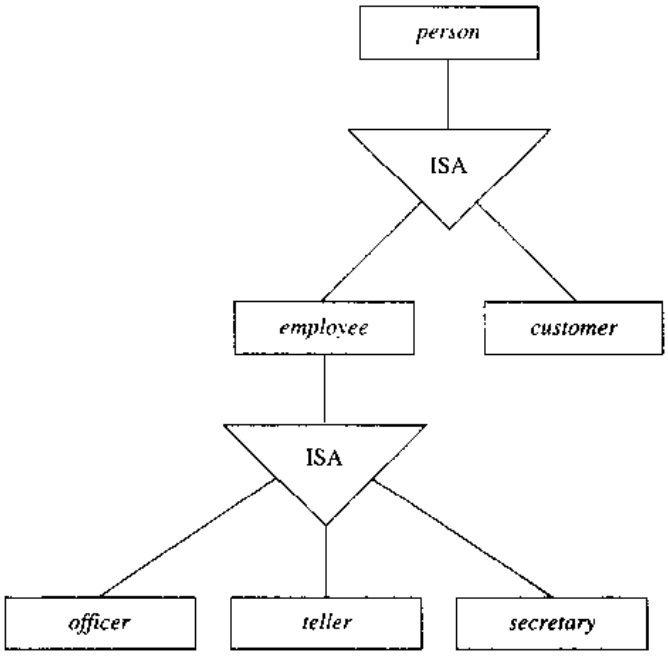


图 8-1　银行例子的特殊化层次

关键词 isa 用来指出一个类是另一个类的特殊化，类的特殊化称作子类。这样，*employee* 是 *person* 的一个子类，*teller* 是 *employee* 的一个子类。从相反的方向说，*employee* 是 *teller* 的超类，*person* 是 *employee* 的超类。

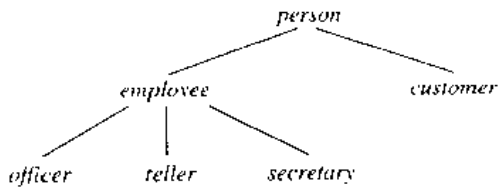


图 8-2　图 8-1 对应的类层次

代表一位主管的对象包含 *officer* 类的所有变量，此外该代表主管的对象同时也包含类 *employee* 和类 *person* 的所有变量，这是由于每位主管都被定义为一名雇员，同时每名雇员又都被定义为一个人，这样可以推出每位主管也都是一

个人。这种一个类的对象包含在其超类中定义的变量的特性称为变量的继承性。

```
class person {
    string name;
    string address;
};
class customer isa person {
    int    credit-rating;
};
class employee isa person {
    date   start-date;
    int    salary;
};
class officer isa employee {
    int    office-number;
    int    expense-account-number;
};
class teller isa employee {
    int    hours-per-week;
    int    station-number;
};
class secretary isa employee {
    int    hours-per-week;
    string manager;
};
```

图 8-3　用伪码书写的类层次定义

方法的继承方式与变量的继承方式相同。在面向对象系统中继承性的一个重要好处是所谓可置换性：一个类的方法，比如 A（或者是接受 A 类的对象作为参数的函数），可以等价地被属于 A 的任何子类 B 的任何对象来引发。这种特性带来了代码重用的好处，因为对于类 B 不必再重写这些方法和函数。例如，如果为类 *person* 定义了 *get-name*（）方法，它就可以被一个 *person* 对象引发，或者也可以被 *person* 的任何子类的任何对象所引发，如 *customer* 或 *officer* 对象。

前面提到每个类本身也是一个对象，在这个对象中有一个变量，它包含该类所有实例的集合。对于位于类继承层次叶结点的类来说，判断哪些对象与之相关联是容易的，比如将银行所有客户的集合与类 *customer* 相关联。然而对于非叶结点来说，这个问题就相对复杂一些，在图 8-2 的层次结构中，有两种看起来较合理的方法能将对象与类关联在一起：

1）将所有雇员对象，包括类 *officer*、*teller* 和 *secretary* 的实例，与类 *employee* 相关联。

2）只将那些不是类 *officer*、*teller* 或 *secretary* 的实例的雇员对象与类 *employee* 相关联。

在面向对象系统中一般选择后一种方法，在这种情况下要找出所有雇员对象的集合也是可以做到的，只需将属于类 *employee* 的所有子类的对象合并起来。绝大多数面向对象系统都允许部分的特殊化，也就是说，允许存在属于一个类（如类 *employee*）而并不属于这个类的任何子类的对象。

8.2.4　多重继承

在多数情况下，类的树形结构组织已足以用来描述应用，在这些树形结构中，一个类的所有超类都在层次结构中互相作为祖先或后代（也就是说，任何两个超类之间都是祖先和后代的关系）。然而，有些情况用树形结构类层次并不能很好地表达。

假定在图 8-2 的例子中想区分全时工作与部分时间工作的出纳和秘书，再假定对于全时工作与部分时间工作的雇员需要不同的变量和方法来表达，这样，每个雇员都可以用两种不

同的方法来分类。可以创建以下子类：*part-time-teller*、*full-time-teller*、*part-time-secretary* 和 *full-time-secretary*。图 8-4 所示的类层次有如下一些缺陷：

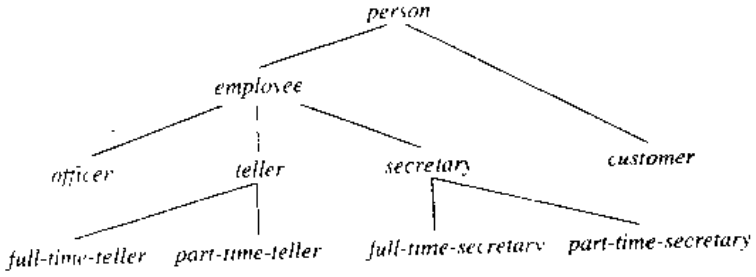


图 8-4　全时/部分时间工作的雇员的类层次

- 正如我们所提到过的，有些变量和方法是特定于全时工作雇员的而另外有些是特定于部分时间工作雇员的。在图 8-4 中，那些全时工作雇员的变量和方法必须要定义两次：一次是为 *full-time-secretary*，另一次是为 *full-time-teller*，这些冗余在部分时间工作的雇员中也同样存在。这种形式的冗余是不良的，因为若要对全时工作或部分时间工作的雇员的特性进行任何改变，必须在两个地方都进行更改，从而容易导致潜在的不一致性。此外，这种层次不能利用可能潜在的代码重用。
- 这个层次无法表示那些既不是主管、也不是出纳或秘书的雇员，除非将层次进一步扩展使之包括 *full-time-employee* 类和 *part-time-employee* 类。

如果不是像这个简单例子中这样仅有两种工作分类，而是有好几种，那么这种模型的局限性就会变得更加明显。

我们刚才看到的这种困难在面向对象模型中用多重继承的概念来解决，多重继承表示一个类具有从多个超类中继承变量和方法的能力。在多重继承的情况下，类-子类的关系可以用一个有向无环图（DAG）来表示，其中一个类可以有多于一个的超类。再看银行的例子。使用 DAG，我们可以在一个地方定义全时工作和部分时间工作的雇员的特性。如图 8-5 所示，我们定义一个类 *part-time*，其中定义了那些特定于部分时间工作的雇员的变量和方法；同时还有一个类 *full-time*，定义了特定于全时工作雇员的变量和方法。*part-time-teller* 类既是类 *teller* 的子类也是类 *part-time* 的子类，它从类 *teller* 继承了属于出纳的那些变量和方法，同时又从类 *part-time* 继承了属于部分时间工作的雇员的那些变量和方法。如此一来消除了图 8-4 中层次的冗余。

使用图 8-5 所示的 DAG 可以表示出那些既不是出纳也不是秘书的全时/部分时雇员，它们分别作为类 *full-time* 和类 *part-time* 的实例。

使用了多重继承，当同一个变量或方法可以从多个超类中继承时就会产生含义模糊的问题，在银行例子中，现假定我们不为 *employee* 定义 *salary* 变量，而是为类 *full-time*、*part-time*、*teller* 和 *secretary* 都定义一个变量 *pay*，如下：

- *full-time*。*pay* 是一个范围从 0～100 000 的整数，表示年薪。
- *part-time*。*pay* 是一个范围从 0～20 的整数，表示每小时付的薪金比例。
- *teller*。*pay* 是一个范围从 0～20 000 的整数，表示年薪。
- *secretary*。*pay* 是一个范围从 0～25 000 的整数，表示年薪。

下面考虑 *part-time-secretary* 类，它既可以从类 *part-time* 也可以从类 *secretary* 中继承 *pay* 变量，选择从哪个类继承所得到的结果是不同的。以下是各个面向对象模型实现所选用的几种方案：

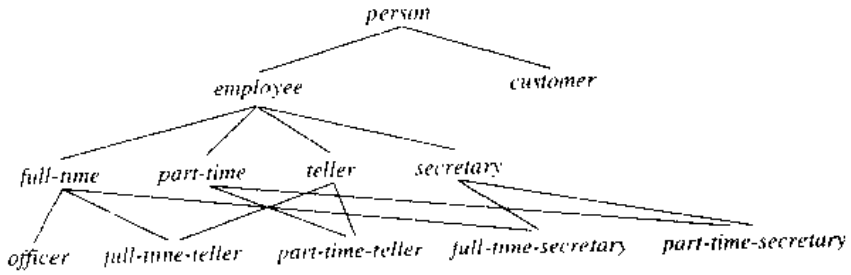


图 8.5　银行例子的类 DAG

- 同时包含两个变量，分别重新命名为 *part-time.pay* 和 *secretary.pay*。
- 根据类 *part-time* 和类 *secretary* 的创建时间顺序选择其中之一。
- 强制用户在 *part-time-secretary* 类的定义中显式地作出选择。
- 将这种情况作为一种错误来处理。

没有哪种解决方案被普遍认为最好的，不同的系统会作出不同的选择。

并不是所有的多重继承情况都会导致含义模糊。如果不定义 *pay* 变量，而是在类 *employee* 中保留 *salary* 的定义，并且在其他地方不再去定义这个变量，那么类 *part-time*、*full-time*、*secretary* 和 *teller* 都从类 *employee* 中继承变量 *salary*。由于这四个类共享相同的 *salary* 定义，从而 *part-time-secretary*、*full-time-secretary* 等类对 *salary* 变量的继承就不会再有二义性。

可以使用多重继承来对“角色”的概念进行建模。为了理解这个概念，考虑一个大学的数据库，该数据库中可能有类 *person* 的多个子类，如 *student*、*teacher* 和 *footballPlayer* 等。一个对象可以同时属于这些类别中的多个，每一个这样的类别被称为一个角色。可以使用多重继承来创建诸如 *student-teacher*、*student-footballPlayer* 等子类，用以表达一个对象可能同时具有多种角色。我们将在第 9 章中讨论多重继承的数据定义语言问题，在那里我们将给出角色建模的另外一种方法。

8.2.5　对象标识

在面向对象数据库中一个对象通常对应着数据库所描述的企业中的一个实体。一个实体保持自身的标识不变，即使它的有些特性多次改变；同样，一个对象也保持自身的标识不变，即使它的一些或者全部变量的值或方法的定义多次改变。这种标识的概念对于关系数据库中的一个元组来说并不适用。在关系数据库系统中，关系的元组只能从它们所包含的值来进行区分。

对于程序设计语言或非面向对象数据模型中通常使用的标识来说，对象标识是一种更强的标识方法，下面列出了几种形式的标识：

- 值。使用一个数据值来进行标识，在关系系统中通常使用这种形式的标识。例如，一个元组的主码标识了这个元组。
- 名称。用用户提供的名称作为标识，这种形式的标识通常用于文件系统中的文件。不管文件的内容是什么，每个文件都被赋予一个名称来唯一标识。
- 内置。在数据模型或程序设计语言中内置的一种标识方法，不需要用户提供标识符。面向对象系统中使用这种形式的标识，每个对象在创建时被系统自动赋予一个标识符。

对象的标识是一种概念上的东西；实际的系统需要一种物理机制来唯一标识对象。对于人来说，姓名再加上其他一些信息，如出生日期和地点，就可经常被用来作为标识符。面向

对象系统提供了一种对象标识符的观念来标识对象。对象标识符是唯一的，也就是说，每个对象具有单一的标识符，并且没有两个对象具有相同的标识符。对象标识符的形式不必一定是要人所容易理解的，例如，它可以是一长串的数字，能够像存储对象的一个字段那样存储一个对象的标识符比有一个容易记忆的名称更为重要。

来看对象标识符的一个示例，*person* 对象的一个属性可以是 *spouse*，这个属性实际上是对应这个人的配偶的另外一个 *person* 对象的标识符，这样，*person* 对象就可以存储代表这个人配偶的对象的引用。

标识符通常是由系统自动生成的。与此相反，在关系数据库中，一个 *person* 关系的 *spouse* 字段通常会是由数据库系统外部生成的此人配偶的一个唯一标识（也可能是一个社会保障号）。很多情况下由系统自动生成标识符是有好处的，因为它不再需要人来完成这项工作。然而，在使用这种功能时也应该有所注意，系统生成的标识符通常是特定于这个系统的，如果要将数据转移到另外一个不同的数据库系统中，则标识符必须进行转化。如果一个实体在建模时已经有一个系统之外的唯一标识，则系统生成的标识符就可能是多余的，例如，在美国经常用社会保障号作为人的唯一标识。

8.2.6　对象包含

对象之间的引用可以用来描述不同的现实世界概念，对象包含即是其中之一。为了说明包含，考虑图 8-6 所示的简化了的自行车结构数据库。每一个自行车结构包括车轮、车架、车闸和齿轮。车轮又包括一个轮框、一套辐条和一个轮胎。结构的每一个构件都描述为一个对象，同时构件间的包含也可以用对象间的包含来描述。

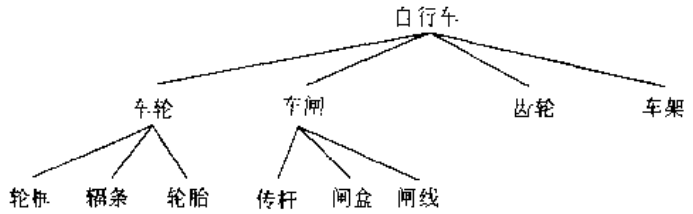


图 8-6　自行车结构数据库的包含层次

包含其他对象的对象称为复杂对象或复合对象。可以存在多层包含，正如图 8-6 中所示，这种情形就产生了对象间的包含层次。

通过列出类名而不是列出单个对象，图 8-6 中的层次用模式的方法显示了对象间的包含关系。类之间的这种关系必须解释为 *is-part-of*，而不是像继承层次中的关系那样解释为 *is-a*。

在图中并没有将各个类的变量显示出来。例如自行车类，它可能包括的一些变量如商标名，其中就包含了关于自行车类的一个实例的描述数据。此外，自行车类还包括一些变量，它们分别包含了对车轮、车闸、齿轮和车架类的引用，或者引用集合。

在面向对象系统中，包含是一个重要的概念，因为它允许不同的用户从不同的角度来观察数据。一个车轮设计师可以只专注于车轮类的实例，而不用太去或者根本不用去关心齿轮类或车闸类的对象。而对于一个做市场的职员来说，他要尝试为整个自行车定一个价格，因而他就可以通过引用自行车类的相应实例来引用属于这个自行车的所有数据。包含层次可以用来寻找在一个自行车对象中所包含的所有对象。

在某些应用中，一个对象可能要包含在多个对象中，这时包含关系要用一个 DAG 而不

是用层次来表示。

8.3　面向对象的语言

到目前为止，我们所讲的内容已经在一个非常抽象的层次上覆盖了面向对象的基本概念。要在一个数据库系统中实际地使用这些概念，它们必须要表达为某种语言，这种表达可以用以下两条途径之一来完成：

1）将面向对象的这些概念纯粹作为一种设计工具，并编码到如一个关系数据库中。当使用实体-联系图对数据进行建模时可以按照这种方法，然后再手工转化为一系列的关系。

2）将面向对象的概念结合到一种用来操纵数据库的语言中。

在这条途径中，与这些概念集成的语言可能有多种：

- 一种选择是通过增加复杂类型和面向对象特征来扩展某种数据操纵语言，例如SQL。在关系系统上提供面向对象扩展的系统称为对象-关系系统，我们将在第9章中描述对象-关系系统，特别是SQL的面向对象扩展。
- 另一种选择是找一种现存的面向对象程序设计语言，将它扩展以便能对数据库进行处理，这种语言称为持久化程序设计语言。

我们将在本章的随后几节研究后一种方法，哪一种方法更好要针对不同的情况。在第9章描述完对象-关系系统之后，我们将讨论它们之间的关系。

8.4　持久化程序设计语言

数据库语言与传统程序设计语言的区别在于前者直接操纵持久的数据——即使创建数据的程序已经终止，这些数据仍然继续存在。数据库中的关系和关系中的元组都是持久数据的例子。相反地，传统程序设计语言所直接操纵的唯一持久数据是文件。

访问数据库只是现实世界应用中的一个组成部分，即便一种数据操纵语言（比如SQL）存取数据是相当高效的，仍然需要一种程序设计语言来实现应用中的其他组成部分，如用户界面或与其他计算机的通信。要将数据库访问结合到程序设计语言中，传统的方法是在程序设计语言中嵌入SQL。

持久化程序设计语言是一种用以处理持久数据而扩充了结构的程序设计语言，持久化程序设计语言与嵌入SQL的语言至少可以从以下两个方面进行区分：

1）在嵌入式语言中，宿主语言的类型系统通常与数据操纵语言的类型系统有所不同，任何宿主语言和数据操纵语言之间的类型转化都由程序设计者负责。要求程序设计者来实现这项工作有如下几点缺陷：

- 对象和元组之间进行转换的代码是在面向对象类型系统之外进行操作的，因此存在没有检测到的错误的机率很高。
- 数据库中面向对象格式和关系格式元组之间的转换占据相当数量的代码。进行格式转换的代码，以及从数据库装入/卸出的代码，它们占了应用所需要的所有代码的相当大的比例。

与此相反，在持久化程序设计语言中，查询语言已经完全集成到宿主语言中，它们共用相同的类型系统。对象在数据库中的创建和存储可以不需要任何外在类型或格式转换，任何所需要的格式转换都透明地完成。

2）使用嵌入查询语言的程序设计者要负责编写外在的代码来把数据从数据库取到内存中，如果数据做了任何更新，程序设计者必须显式地编写代码将更新后的数据写回到数据库

中。与此相反，在持久化程序设计语言中，程序设计者要操纵持久数据，并不需要显式地编写代码去将它们取到内存或写回到磁盘中。

一些程序设计语言（如 Pascal）的持久化版本很早就已提出，近年来，一些面向对象程序设计语言如 C++ 和 Smalltalk 的持久化版本受到了人们普遍的关注，它们允许程序设计者直接从程序设计语言中操纵数据，而不必再通过一个像 SQL 那样的数据操纵语言。这样，它们能够比其他方式，如嵌入 SQL，更紧密地将程序设计语言与数据库集成在一起。

不过持久化程序设计语言也有一定的不足之处，这在我们决定是否使用它时应该记住的一点。由于程序设计语言本身能力通常很强，因此出现编程错误而破坏数据库的机会相对较大。此外，语言的复杂性也使得自动的高级优化（例如减少磁盘 I/O）比较困难。对许多应用来说支持说明性查询是很重要的，但当前的持久化程序设计语言对说明性查询的支持并不理想。

8.4.1　对象的持久性

面向对象程序设计语言中已经有了对象的概念，同时有一个用于定义对象类型的类型系统以及创建对象的一些结构，不过，这些对象是瞬态的——在程序结束后它们就随之消失，正如 Pascal 或 C 程序中的变量在程序结束后会消失一样。如果想要将这样的一种语言变成数据库程序设计语言，第一步就是要提供一种方法使得对象变成持久的，对此已经提出了如下一些途径：

- 按类持久。一种最简单、但最不方便的途径是声明一个类为持久的，于是该类的所有对象在缺省情况下都是持久对象，非持久类的对象都是瞬态的。由于经常需要在单个类中同时有瞬态和持久对象，因此这种方法是不灵活的。在许多面向对象系统中，声明一个类是持久的常解释为该类中的对象可能潜在会变成持久的，而不是类中所有对象就是持久的。这些类被称作“可持久的”类或许会更合适一些。
- 按创建持久。这条途径中，引入新的语法，即通过扩展创建瞬态对象的语句来创建持久对象，这样，一个对象是持久的还是瞬态的，取决于它们是被如何创建的。有些面向对象数据库系统就采用了这种方法。
- 按标志持久。前述途径的一个变种为在对象创建后将它们标志为持久的。所有的对象都被创建为瞬态对象，但是，如果一个对象要在程序结束后持久存在，那么必须在程序终止前显式地将这个对象标志为持久的。与前一条途径不同，决定对象是持久的还是瞬态的被延迟到对象创建以后。
- 按引用持久。一个或多个对象被显式地声明为（根）持久对象。对于所有其他对象来说，当（且仅当）从一个根持久对象中直接或间接引用到它们时，它们才是持久的。这样，所有从根持久对象引用到的对象都是持久的，同时所有从这些对象引用到的对象也都是持久的，之后它们再引用的对象也是持久的，依此类推。这种方案有一个好处是很容易将整个数据结构变成持久的，因为只需声明数据结构的根为持久的就可以了。然而，这样一来数据库系统就得沿着引用链查找以确定对象是否是持久的，而这样的代价可能是很高的。

8.4.2　对象标识与指针

当创建一个持久对象时，对象被赋予一个持久对象标识符。在一个尚未被扩展以处理持久化的程序设计语言中，当创建一个对象时，返回的是一个瞬态对象标识符。两种对象标识符之间的唯一差别在于，瞬态对象标识符只有当创建它的程序正在执行时才是合法的，一旦

程序结束，对象就被删除，同时其标识符也就没有任何意义了。

对象标识的概念与程序设计语言中的指针之间存在有趣的关系。要做到内置标识，最简单的方法是通过指向内存的物理位置的指针，特别是在许多面向对象语言中（例如C++），对象标识符实际上就是一个内存指针。

然而，一个对象与存储器中的物理位置之间的关联可能经常发生改变，标识的持久程度有以下几种：

- 过程内部。只有在单个过程的执行期间标识才是持久的。过程内的局部变量就是过程内部标识的例子。
- 程序内部。只有在单个程序或查询的执行期间标识才是持久的。程序内部标识的例子如程序设计语言中的全局变量，内存或虚存指针只提供程序内部的标识。
- 程序之间。从一个程序执行到另一个程序执行之间的标识都是持久的。指向磁盘文件系统数据的指针提供了程序之间的标识，但是当数据存储在文件系统中的方式发生变化时，它们仍可能被改变。
- 持久的。标识的持久性不仅仅跨越了各个程序的执行，还跨越了数据结构的重新组织。这种持久性正是面向对象系统所要求的。

在诸如C++这样的语言的持久化扩展中，持久对象的对象标识符用“持久化指针”来实现，持久化指针与内存指针不同，持久化类型的指针即使在程序结束后仍然是有效的，并且跨越了某些形式的数据重组，程序设计者在程序设计语言中可以像使用内存指针一样使用持久化指针。我们可以概念上地将持久化指针看作指向数据库中某个对象的指针。

8.4.3　持久对象的存储和访问

在数据库中存储一个对象的含义是什么？显然，对象的数据部分必须要对每个对象单独存储。在逻辑上，实现类的方法的代码应该作为数据库模式的一部分存储在数据库中，与类的类型定义放在一起。然而，许多实现简单地将代码存储在数据库之外的文件中，这样做是为了避免必须将如编译器这样的系统软件集成到数据库系统中。

在数据库中寻找对象的方法有好几种。一条途径是给出对象的名字，正如我们指定文件名一样。这种方法适用于对象数目相对较少的情况下，而不适用于上百万个对象的情形。第二条途径是将对象标识符或指向对象的持久化指针提供出来，使它们可以在外部存储。与名称不同，这些指针不一定是人所容易记忆的，它们甚至可能就是指向数据库内部的物理指针。

第三种途径是存储对象的一个集合体，然后允许程序在集合体上进行迭代，寻找出所要的对象。对象的集合体本身可以被描述成一个集合体类型的对象，集合体类型包括集合、多重集合、列表等等。集合体的一种特殊情形是类区间，它是属于一个类的所有对象的集合体。当一个类存在类区间时，则不论何时，只要创建了该类的一个对象都会被自动插入到类区间中，并且一旦对象被删除也会自动从类区间中移去。类区间允许类能像关系一样对待，可以检查类的所有对象，正如可以检查一个关系中的所有元组一样。

多数面向对象数据库系统支持上面所有三种访问持久对象的方法。所有的对象都有对象标识，且一般只对类区间和其他集合体对象赋予名字，或许其他所选择的对象也有名字，但对绝大多数的对象并没有命名。通常对所有可以有持久对象的类都维护类区间，但是在很多实现中，类区间只包括持久对象。

8.5　持久化 C++ 系统

近几年中出现了一些基于 C++ 的持久化扩展的面向对象数据库（参见文献注解），它们从系统结构上看有一定的差异，然而从程序设计语言的角度来看它们又有很多共同的特征。

C++ 语言的一些面向对象特征有助于支持持久化而无需改变语言本身。例如，我们可以声明一个名为Persistent_Object 的类，它具有一些属性和方法来支持持久化，其他任何持久的类都可以作为这个类的子类，从此继承对持久化的支持。C++ 语言（像其他现代程序设计语言一样）也提供了根据运算对象的类型对标准函数名和操作符，如 +、−、指针内容操作符 −>，等等，进行重定义的能力。这种能力被称为重载，用于重定义操作符，使得当这些操作符在持久对象上操作时能够按所需要的方式工作，使用这种特征的例子将在下一节给出。

通过类库来提供持久化支持具有对 C++ 修改最小的优点，且相对比较容易实现。然而，它也有一些缺陷，程序设计者必须要花大量的时间书写处理持久对象的程序，此外，要在模式中指定完整性约束也是不容易的。同样不容易的还有提供说明性查询。因此，大多数持久化 C++ 的实现都对 C++ 的语法进行了扩展，至少在一定程度上是这样。

8.5.1　ODMG C++ 对象定义语言

对象数据库管理小组（ODMG）一直在从事对 C++ 和 Smalltalk 语言的持久化扩展进行标准化的工作，他们的工作同时也包括定义类库以支持持久化。ODMG 标准力图尽可能小的扩展 C++，而通过类库提供绝大多数的功能。我们通过例子来描述 ODMG 标准，但必须首先具有 C++ 程序设计语言的知识才能完全理解这些例子。（关于 C++ 请参见文献注解。）

ODMG 的 C++ 扩展有两个部分：1）C++ 对象定义语言（C++ ODL）；2）C++ 对象操纵语言（C++ OML）。C++ ODL 扩展了 C++ 的类型定义语法。

图 8-7 显示了一个用 ODMG C++ ODL 书写的代码示例，在代码中定义了四个类模式，每个类都被定义为Persistent_Object 的一个子类，这样类中的对象就变为持久的。Person、Branch 和Account 类是Persistent_Object 的直接子类，Customer 类是Person 类的子类，因此间接地也是Persistent_Object 的一个子类。

C++ 并不直接支持消息的概念，但是可以直接引发方法，功能类似于过程调用。关键词private 表示随后的属性或方法仅对类中的方法可见，public 表示属性或方法同时也对其他代码可见。关键词public 用在一个超类名字的前面，表示从超类继承来的公用属性或方法在子类中仍然是公用的。这些特征都是标准 C++ 的一部分。

将属性分别定义为int、String 和Date 类型使用的是标准的 C++ 语法形式，String 和Date 类型属于 ODMG 所定义的标准类型。Ref<Branch>类型是一个引用，亦即持久化指针，指向类型为Branch 的一个对象，Set<Ref<Account>>类型是一个持久化指针的集合，这些指针指向类型为Account 的对象。我们使用符号 Inverse 来指定参照完整性约束，例如任一Customer 对象，对accounts 属性指定 Inverse Account::owners 表示，对于该Customer 对象的accounts 集合中所引用的每个account 对象来说，其owners 属性必须包含一个引用指回这个Customer 对象。

类Ref<T>和Set<T>为 ODMG 标准中定义的模板类，它们是用类型无关的方法定义的，同时需要实例化为所要求的类型（例如Ref<Account>）。模板类Set<T>提供了一些诸如insert_element 和delete_element 之类的函数。

```
class Person : public Persistent_Object {
public:
    String name;
    String address;
};

class Customer : public Person {
public:
    Date member_from;
    int customer_id;
    Ref<Branch> home_branch;
    Set<Ref<Account>> accounts inverse Account::owners;
};

class Branch : public Persistent_Object {
public:
    String name;
    String address;
    int assets;
};

class Account : public Persistent_Object {
private:
    int balance;
public:
    int number;
    Set<Ref<Customer>> owners inverse Customer::accounts;
    int find_balance();
    int update_balance(int delta);
};
```

图 8-7　ODMG C++ 对象定义语言的示例

Account 类的声明同时也说明了 C++ 的“封装”特性。balance 和 Account 属性被声明为私有的，这意味着除类本身的方法以外没有别的函数能够读写它。这个类同时包括两个方法：find_balance() 和update-balance (int delta)，这些方法可以读写balance 属性，我们应该使用它们来读取和更新帐户的余额（我们并没有给出它们的代码）。

前面的示例中对于 C++ 语法的唯一的扩展是指定reverse 关系以保证参照完整性，其他所有 ODMG 添加的功能都是通过类库的方法提供的。

计划中对标准的扩展还包括将Ref< T >替换为 C++ 的符号T *，这样做是为了提高可读性。对于程序设计者来说，指向持久对象的指针看起来与指向常规内存对象的指针是相同的，这样就使得编写程序容易了许多。这种功能是通过一种称为指针混写的转换技术来实现的，指针混写是在持久化指针和内存指针间进行转换。我们将在第 10 章中描述指针混写。

8.5.2　ODMG C++ 对象操纵语言

图 8-8 显示了一个用 ODMG C++ OML 书写的代码示例。首先打开一个数据库，然后开始一个事务。

事务的概念将在第 13 章中详细介绍，简而言之，事务是一些操作步骤的序列，以一个开始事务的调用及一个提交或中止事务的调用作为分界。事务的这些步骤被看作是一个原子的单元。数据库系统保证要么所有这些步骤都成功执行，要么如果任何步骤由于某种原因未能执行，则所有已执行步骤所得的结果都在数据库中撤消。第一种情形称为该事务已提交，第二种情形称为该事务被中止。

下一步，创建一个帐户和一个所有者对象并进行初始化，这些都作为事务的一部分。最

终，这个事务被提交。Persistent_Object 类实现了一些方法，包括示例代码中用到的 C++ 内存分配操作符new 的持久化版本，new 操作符的这个版本是在指定的数据库中分配对象的空间，而不是在内存中。我们使用模板类Set<>的insert_element 方法来在创建客户和帐户对象后将客户和帐户引用插入到相应的集合中，如果这两个插入中只有一个执行，当事务提交时就会检测到这种对参照完整性的违背，整个事务将被中止。

```
int create_account_owner(String name, String address) {
   Database bank_db_obj;
   Database * bank_db = &bank_db_obj;
   bank_db->open("Bank-DB");
   Transaction Trans;
   Trans.begin();

   Ref<Account> account = new(bank_db) Account;
   Ref<Customer> cust = new(bank_db) Customer;
   cust->name = name;
   cust->address = address;
   cust->accounts.insert_element(account);
   account->owners.insert_element(cust);
   初始化客户号、帐户号等等的代码
   Trans.commit();
}
```

图 8-8　ODMG C++ 对象操纵语言的示例

图 8-8 中的示例代码并不完整，这体现在当事务结束以后，客户和帐户对象的标识符就丢失掉，虽然对象仍然在数据库中。使用持久集合来存储这些持久Customer 对象的标识符，将该集合的标识符存储在一个与类Customer 关联的全局变量中是比较方便的，声明这个全局变量可以通过添加下面一行代码作为Customer 类声明的一部分：

```
static Ref<Set<Ref<Customer>>> all_customers;
```

下面的代码是将Customer 的持久集合在数据库中初始分配，同时将其标识符存储在全局变量中：

```
Customer::all_customers
        = new (bank-db) Set<Ref<Customer>>;
```

当创建一个客户对象后，通过下面形式的语句将它的标识符插入到客户集合中：

```
Customer::all_customers->insert_element (cust);
```

同样，可以创建一个存储Account 对象标识符的持久集合，帐户对象的标识符在其中进行存储。

但是，当事务结束后，变量Customers::all_customers 的值就会丢失，这样持久集合的标识符也就随之失去。为了在以后能找到这个集合，当集合初始分配时我们在数据库中给它的对象标识符赋予一个名字，如下所示：

```
bank_db->set_object_name (Customer::all_customers,
                          "All-Customers");
```

当后面的事务启动时，首先打开数据库，然后就可以找到先前创建的那个集合并初始化Cus

tomer:: all _ customers 如下：

```
Customer:: all _ customers =
    bank _ db- > lookup _ object ("All-Customers");
```

类的构造函数是一个特殊的方法，用于在创建对象时对对象进行初始化。当执行new操作符时自动地调用构造函数。同样，类的析构函数也是一个特殊的方法，当类中的对象删除时自动调用它。通过将insert _ element 语句加入到类的构造函数中，同时在类的析构函数中加入相应的delete _ element 语句，程序设计者可以确保能够正确地维护Customers:: all _ customer 集合体。在某些持久化C++扩展中，对于每个可以存在持久对象的类，类区间（即包含类的所有持久对象的集合体）是自动创建并维护的，这样程序设计者既不用编写代码为类区间创建持久化集合，也不用编写代码从类区间中插入或删除对象。

可以使用一个迭代算子在引用的集合体中进行迭代，如图8-9所示。使用由Collection类及其子类（如Set类）提供的create _ iterator 方法创建一个迭代算子，再使用算子所提供的

```
int print_customers() {
   Database bank_db_obj;
   Database * bank_db = & bank_db_obj;
   bank_db->open("Bank-DB");
   Transaction Trans;
   Trans.begin();
   Customer::all_customers =
      bank_db->lookup_object("All_Customers");

   Iterator<Ref<Customer>> iter =
      Customer::all_customers.create_iterator();
   Ref<Customer> p;
   while(iter.next(p)) {
      print_cust(p);
   }
   Trans.commit();
}
```

图8-9　使用ODMG C++迭代算子的示例

next（）方法，以顺序遍历客户集合体中的成员。对于每个客户，调用print _ cust 方法（假定在别处已经定义）来打印出这个客户。

8.6　总结

对于诸如计算机辅助设计、软件工程及办公信息系统这样的数据库应用来说，为早期的数据处理类型的应用所做的那些假设并不适合它们，人们提出面向对象数据模型来处理这些新类型的应用。

面向对象数据模型是面向对象程序设计范型在数据库系统中的改造，它的基础是将一个对象的数据及对这些数据操作的代码封装在对象中这样一个概念，相类似的对象被分组形成类。基于实体-联系模型中的ISA概念的一个扩展，类的集合被结构化为子/超类。由于对象中的一个数据项的值也是一个对象，因此表示对象包含是可能的，结果就形成了复合对象。

有两种途径可以创建一个面向对象数据库：我们可以将面向对象的概念加入到现存的数据库语言中，或者通过加入诸如持久化和集合体之类的概念来扩展现存的面向对象语言使之能够处理数据库。扩展关系数据库采用的是前一种方法，持久化程序设计语言则是按照后一条途径。C++的持久化扩展在几年中取得了重要的进步。将持久化与现存语言结构无缝地、

正交地集成在一起对于易用性是非常重要的。

习题

8.1　对于下列每一个应用领域，解释为什么关系数据库系统是不适合的，并列出所有需要修改的特定系统成分。

(a) 计算机辅助设计

(b) 计算机辅助软件工程

(c) 多媒体数据库

(d) 办公信息系统

8.2　面向对象模型中的对象概念与实体 - 联系模型中的实体概念有什么不同？

8.3　一个汽车租赁公司为其当前车队中的所有车辆维护着一个车辆数据库。对于所有的车辆，数据库中包括的信息有车辆识别号、牌照号、制造商、型号、购买日期以及颜色，对于某些类型的车辆还包括特殊的数据：

- 卡车：载货容量。
- 跑车：马力、对租用者的年龄限制。
- 厢式货车：乘客数目。
- 越野车：离地间隙、驱动类型（四轮还是两轮驱动）。

为这个数据库构造一个面向对象数据库模式定义，适当的时候使用继承。

8.4　解释为什么对于多重继承存在潜在的模糊含义，请举例说明。

8.5　解释面向对象模型中对象标识的概念与关系模型中元组相等的概念有什么不同。

8.6　解释在表示继承的 DAG 和表示对象包含的 DAG 中边的含义有什么区别。

8.7　对一个提供持久对象的系统，这样的系统一定要是一个数据库系统吗？请解释你的答案。

8.8　假定 O_1 和 O_2 是包含相同对象集合的复合对象，O_1 一定要等同于 O_2 吗？请解释你的答案。

8.9　为什么持久化程序设计语言中允许有瞬态对象？只使用持久对象，并在结束时删除掉不需要的对象，这样做会更简单一些吗？解释你的答案。

8.10　解释如何实现持久化指针，将这种实现与通用程序设计语言（如 C 或 Pascal）中指针的实现做一个比较。

8.11　如果一个对象创建时没有任何引用指向它，那么如何删除这个对象？

文献注解

关于数据库概念在 CAD 中的应用在以下文献中讨论：Haskin 和 Lorie [1982]，Kim 等 [1984]、Lorie 等 [1985]、Ranft 等 [1990]、Bancilhon 等 [1985a，1985b]。

面向对象的程序设计在文献 Goldberg 和 Robson [1983]、Cox [1986]、Stefik 和 Bobrow [1986]、及 Stroustrup [1988] 中讨论。Stroustrup [1992] 描述了 C++ 程序设计语言。有许多已实现的面向对象数据库系统，包括 Cactis（Hudson 和 King [1989]）、威斯康星大学开发的 E/Exodus（Carey 等 [1990]）、Gemstone（Maier 等 [1986]）、惠普公司开发的 Iris（Fishman 等 [1990]）、富士通实验室开发的 Jasmine（Ishikawa 等 [1993]））、O2（Lecluse 等 [1988]）、ObjectStore（Lamb 等 [1991]）、贝尔实验室开发的 Ode（Agrawal 和 Gehani [1989]）、Ontos、Open-OODB、Orion（Banerjee 等 [1987a]、Kim 等 [1988] 和 Kim

[1990a])、Versant 等等。

面向对象数据库研究的概述包括 Kim 和 Lochovsky [1989]、Kim [1990b]、Zdonik 和 Maier [1990]、Dittrich [1988]。对象标识在 Khoshafian 和 Copeland [1990] 中有详细地描述，关于对象标识的深入讨论有 Abiteboul 和 Kanellakis [1989]。对象数据库标准 ODMG-93 在 Cattell [1994] 中有详细描述。面向对象数据库的最新发展在文献 Dogac 等 [1994] 中讨论。

面向对象数据库中的模式修改比起关系数据库中的模式修改要更为复杂，因为面向对象数据库有着复杂的类型系统并带有继承。模式修改在以下文献中讨论：Banerjee 等 [1987b]、Penney 和 Stein [1987]。其他关于模式修改和语义的讨论见 Skarra 和 Zdonik [1986]。

Goodman [1995] 描述了在一个基因数据库应用中使用面向对象数据库的好处及缺点。

第 9 章　对象-关系数据库

持久化程序设计语言是将持久性及其他数据库特征加入到一个现存的、已经具有一个面向对象的类型系统的程序设计语言中，与此不同，对象-关系数据模型扩展关系模型的方式是通过提供一个具有面向对象的更加丰富的类型系统，同时将一些成分加入到关系查询语言（如 SQL）中以处理这些新增的数据类型。经过扩展的类型系统允许元组的属性具有复杂类型。这种扩展力图在扩展建模能力的同时保留关系的基础——特别是说明性查询。对象　关系数据库系统（即以对象-关系模型为基础的数据库系统）为那些想要使用面向对象特征的关系数据库用户提供了一个方便的移植途径。

我们首先给出嵌套关系模型的概念形成，这个模型可不符合第一范式的关系，且允许直接表达层次结构。然后展示如何通过加入丰富的类型系统和面向对象来扩展 SQL 的 DDL，接着将展示如何扩展 SQL 查询语言来处理复杂类型（包括嵌套关系）和对象。最后讨论持久化程序设计语言与对象-关系系统的差别，并论述在两者之间进行选择的标准。

9.1　嵌套关系

第 7 章定义了第一范式（1NF），它要求所有的属性都具有原子的域，当一个域的成员都被认为是不可再分的单元时该域就被称为是原子的。

例如，整数的集合是一个原子的域，但整数的集合所组成的集合是一个非原子的域，区别在于通常认为整数不可以再分，而整数的集合还可以分成更小的部分——即组成集合的整数。如果认为每个整数是一串数字的有序列表，那么所有整数组成的域就会被认为是非原子的。可见，1NF 中的要点并不是域本身，而是在数据库中使用域成员的方法。

我们到这里才强调 1NF 假定，是因为在所考虑的银行例子中 1NF 假定是自然的。然而并不是所有的应用都符合 1NF 假定。某些应用的用户将数据库视为一个对象的集合，而不是一个记录的集合，这些对象可能需要数条记录来表示它们。我们可以看到一个简单、易用的用户界面是要求用户直观上的对象概念与数据库系统中数据项的概念是一一对应的。

嵌套关系模型是关系模型的一个扩展，它的扩展之处在于域可以是原子的也可以是关系值。这样元组在一个属性上的取值可以是一个关系，于是关系可以存储在关系中，从而一个复杂对象就可以用嵌套关系的单个元组来表示。如果将嵌套关系的一个元组视为一个数据项，那么在数据项和用户观念上的对象之间就有了一个一对一的对应。

我们通过一个办公信息系统的例子来说明嵌套关系。考虑一个文档检索系统，其中对每个文档存储如下信息：

- 文档标题。
- 作者列表。
- 获得日期。
- 关键词列表。

可以看到，如果为上面这些信息定义一个关系，下列这些域将是非原子的：

- 作者。一个文档可能有一组作者，然而，当我们想要寻找其中一名作者是 Jones 的所

有文档，我们就对域元素“作者集合”的那个部分感兴趣。

• 关键词。如果为一个文档存储了一组关键词，希望能够检索出关键词集合中包括一个或多个关键词的所有文档，那么，我们就将关键词列表域视为非原子的。

• 日期。与关键词和作者不同，日期没有一个集合值的域。但是，我们可能将日期视为由子域日、月、年组成，这种观念使得日期域成为非原子的。

图 9-1 展示了一个示例的文档关系 *doc*，*doc* 关系可以在 1NF 中表示，如图 9-2 所示。由于在 1NF 中只能有原子的域，但我们又想要访问单个作者和单个关键词，因而我们对每个（关键词，作者）对都需要一个元组。日期属性在 1NF 版本中被三个属性所取代，它们分别对应于日期的三个子域。

title	*author-list*	*date*	*keyword-list*
		(*day*, *month*, *year*)	
salesplan	{Smith, Jones}	(1, April, 89)	{profit, strategy}
status report	{Jones, Frick}	(17, June, 94)	{profit, personnel}

图 9-1 非 1NF 文档关系 *doc*

title	*author*	*day*	*month*	*year*	*keyword*
salesplan	Smith	1	April	89	profit
salesplan	Jones	1	April	89	profit
salesplan	Smith	1	April	89	strategy
salesplan	Jones	1	April	89	strategy
status report	Jones	17	June	94	profit
status report	Frick	17	June	94	profit
status report	Jones	17	June	94	personnel
status report	Frick	17	June	94	personnel

图 9-2 非 1NF 文档关系 doc 的一个 1NF 版本 *flat-doc*

如果能保持下列多值依赖，图 9-2 中关系 *flat-doc* 的很多不足之处就可以去除：

• $title \twoheadrightarrow author$。

• $title \twoheadrightarrow keyword$。

• $title \rightarrow day\ month\ year$。

那么，就可以使用下面的模式将这个关系分解成 4NF：

• (*title*, *author*)。

• (*title*, *keyword*)。

• (*title*, *day*, *month*, *year*)。

图 9-3 展示了图 9-2 中的关系 flat-doc 到上述分解的映射。

尽管 1NF 足够表达我们的示例文档数据库，但由于文档管理系统的典型用户是以非 1NF 设计的观点来考虑数据库，因此用非 1NF 表示可能是一个更易于理解的模型。4NF 设计会要求用户在他们的查询中包含连接操作，这样也就带来了与系统的复杂交互。可以设计一个无嵌套的关系视图来消除用户在他们的查询中编写连接操作的需要，然而在这样的视图中，我们却失去了元组与文档间的一对一对应。

title	*author*
salesplan	Smith
salesplan	Jones
status report	Jones
status report	Frick

title	*keyword*
salesplan	profit
salesplan	strategy
status report	profit
status report	personnel

title	*day*	*month*	*year*
salesplan	1	April	89
status report	17	June	94

图 9-3　图 9-2 中关系 *flat-doc* 的 4NF 版本

9.2　复杂类型和面向对象

嵌套关系只是对基本关系模型扩展的一个实例，其他非原子数据类型，如嵌套记录，也已被证明是有用的。面向对象数据模型已经导致了对于诸如对象的继承和引用之类特征的需求。复杂对象系统和面向对象使得 E-R 模型的一些概念，如实体标识、多值属性及概括化和特殊化，能够直接表达，而不再需要至关系模型的复杂转化。

本节描述对 SQL 的扩展以允许复杂类型，包括嵌套关系以及面向对象特征。一些这样的扩展已经被提出，同时向标准化的努力也正在进行之中。我们的介绍基本上是基于 SQL-3 标准的一个初步草案，该标准可望在几年内最终确定。我们的介绍并不是严格按照当前的草案，况且标准也在不断地发展和变化。我们的介绍同时也结合了其他已提出的扩展，如 XSQL，以及 Illustra 数据库系统的查询语言。XSQL 是一个用强大的面向对象特征来扩展 SQL 的研究性语言，Illustra 是 Postgres 数据库系统的商业化版本。Postgres 由加州大学伯克利分校开发，是使用面向对象特征扩展关系模型的早期系统之一。关于这些语言请参见文献注解。

9.2.1　有结构的类型和集合体类型

考虑以下的语句，它们定义了一个具有上节所描述的复杂类型的关系 *doc*：

```
create type MyString char varying.
create type MyDate
        (day integer,
      month char (10),
      year integer)
create type Document
        (name MyString,
author-list setof (MyString),
date MyDate,
```

keyword-list **setof** (*MyString*))
create table *doc* **of type** *Document*

第一个语句定义了一个类型 *MyString*，它是一个变长的字符串。第二个语句定义了一个类型 *MyDate*，它有三个组成部分：day、month 和 year。第三个语句定义了一个类型 *Document*，它包含一个 *name*、一个作者的集合 *author-list*、一个类型为 *MyDate* 的日期以及一个关键词集合。最后创建表 *doc*，它包含了类型为 *Document* 的元组。上述表的定义与普通关系数据库中的表定义是有区别的，因为前者允许属性为集合或者如 *MyDate* 那样的属性具有结构类型，这些特征使得 E-R 图中的复合属性及多值属性能够直接表达。

需要注意的是，使用上面语句所创建的类型是被记录于在数据库中存储的模式中的，这样，访问数据库的其他语句就可以使用这些类型定义。程序设计语言（包括持久化程序设计语言）中的类型定义通常是不存储在数据库中的，这些类型定义只对那些包括文本文件的程序才是可见的，其中文本文件中包含了这些类型定义。

也可以不为表创建一个中间类型而直接创建表，例如，表 *doc* 也可以定义如下：

create table *doc*
(*name MyString*,
author-list **setof** (*MyString*),
date MyDate,
keyword-list **setof** (*MyString*))

复杂类型系统通常还支持其他一些集合体类型，如数组和多重集合（即无序的集合，一个元素可以出现多次），下面的属性定义演示了这些集合体类型的使用：

author-array MyString [10]
print-runs **multiset** (**integer**)

这里，*author-array* 是一个作者姓名的数组。从一个作者集合中不能够确定谁是第一作者、谁是第二作者等等，但从一个作者数组中却可以确定。第二个属性 *print-runs* 是一个整数的多重集合，它用来记录每一次文档打印操作所打印出的文档拷贝数目。由于两次打印可能有相同的拷贝数，因此应使用一个多重集合而不是一个集合。

9.2.2 继承

继承可以在类型级别进行，也可以在表级别进行。首先考虑类型的继承，假定有如下的人的类型定义：

create type *Person*
(*name MyString*,
social-security **integer**)

我们可能希望在数据库中对那些是学生或教师的人分别存储一些额外的信息，由于学生和教师同样是人，因而可以使用继承来定义学生和教师类型如下：

```
create type Student
        (degree MyString,
        department MyString)
        under Person
create type Teacher
        (Salary integer,
        department MyString)
        under Person
```

Student 和 *Teacher* 都继承了 *Person* 的属性，即 *name* 和 *social-security*。*Student* 和 *Teacher* 被称为 *Person* 的子类型，*Person* 是 *Student* 的父类型，同时也是 *Teacher* 的父类型。

现在假定要存储关于助教的信息，这些助教既是学生又是教师，甚至可能是在不同的系里。回忆一下我们在第 8 章中学习过的多重继承的思想，如果类型系统支持多重继承，可以试着为助教定义一个类型如下：

```
create type TeachingAssistant
        under Student, Teacher
```

TeachingAssistant 应该继承了 *Student* 和 *Teacher* 的所有属性，但是却有一个问题，即属性 *social-security*、*name* 和 *department* 同时存在于 *Student* 和 *Teacher* 中。

social-security 和 *name* 属性实际上是从同一个来源——*Person* 继承来的，因此同时从 *Student* 和 *Teacher* 中继承这两个属性不会引起冲突。然而 department 属性在 Student 和 *Teacher* 中分别都有定义，因为，一个助教可能是某个系的学生同时又是另一个系的教师。为了避免两次出现的 *department* 之间的冲突，可以使用 as 子句将它们重新命名，如下面的 *TeachingAssistant* 类型定义所示：

```
create type TeachingAssistant
        under Student with (department as student-dept)
              Teacher with (department as teacher-dept)
```

类型继承应该小心使用。一个大学数据库可能有 *Person* 的许多子类，例如 *Student*、*FootballPlayer*、*ForeignCitizen* 等等。*Student* 自己可能也有一些子类型，如 *UndergraduateStudent*、*GraduateStudent* 和 *PartTimeStudent*。显然，一个人可以同时属于这些种类中的好几个。正如在第 8 章中所提到的，有时这些种类中的每一个被称为一个角色。

在大多数程序设计语言中，一个实体必须只有一个最明确类型，即如果一个实体有一个以上的类型，则必须有单一一个类型是实体所属于的，同时这个类型又是实体所属的所有类型的子类型。举例来说，假定一个实体具有类型 *Person*，同时又具有类型 *Student*，那么这个实体的最明确类型为 *Student*，因为 *Student* 是 *Person* 的子类型。然而一个实体不能同时既具有类型 *Student* 又具有类型 *Teacher*，除非这个实体具有一个如 TeachingAssistant 那样既是 Student 子类型又是 Teacher 子类型的类型。

为了使每个实体都恰好只有一个最明确类型，我们将不得不为每一种子类型的可能组合创建一个子类型。在前面的例子中，我们将有诸如 *ForeignUndergraduateStudent*、*Foreig-*

nUndergraduateStudentFootballPlayer 等等之类的子类型。然而，我们将由于 *Person* 子类型的巨大数目而止步。

就数据库系统来说一个更好的途径是允许对象具有多重类型，而不必有一个最明确的类型。对象-关系系统可以通过如下途径来描述这种特征：使用表级的继承而不是类型级的继承，且允许一个实体同时存在于多于一个的表中。可以改写前面关于学生和教师的例子如下，首先定义 *People* 表

create table *People*
(*name MyString*,
social-security **integer**)

然后，定义 *students* 和 *teachers* 表：

create table *Students*
(*degree MyString*,
department MyString)
under *People*
create table *Teachers*
(*Salary* **integer**,
department MyString)
under *People*

子表 *students* 和 *teachers* 继承了 *people* 表的所有属性。没有必要再去创建同时从 *students* 和 *teachers* 继承的更深一层的子表（如 *teaching-assistants*），除非对于既是 *students* 又是 *teachers* 的实体存在特殊的属性。

在父/子表间存在着一些完整性要求，举例说明如下：

- 父表 *people* 的每个元组可以对应（指对于所有继承属性具有相同值）*students* 表和 *teachers* 表中至多一个元组。
- *students* 表和 *teachers* 表中的每个元组在 *people* 中必须恰好只有一个对应元组（指对于所有继承属性具有相同值）。

如果没有第一个条件，我们可能在 *students*（或 *teachers*）表中有两个元组对应于同一个人；如果没有第二个条件，我们可能在 *students*（或 *teachers*）表中有某个元组不对应 *people* 中的任何元组，或者对应着 people 中的好几个元组。这些情形无法与现实世界相对应，因此是错误的。

子表可以用不复制所有继承字段的高效方式进行存储，父表除主码以外的其他继承属性也不必存储，它们可以通过与父表进行基于主码的连接的方法派生得到。

多重继承对于表来说也是可能的，正如对于类型是可能的一样。一个助教实体可以简单地属于 *students* 表或 *teachers* 表。然而，如果我们愿意，我们可以为助教实体创建一个表如下：

create table *teaching-assistants*
under *students* **with** (*department* **as** *student-dept*)
teachers **with** (*department* **as** *teacher-dept*)

子表的完整性要求保证了如果一个实体存在于 *teaching-assistants* 表中，那么它也同时存在于 *teachers* 和 *students* 表中。

继承使得模式定义变得更自然。如果没有表的继承，模式设计者不得不显式地通过主码来将对应于子表的那些表与对应于父表的那些表链接，并且不得不在这些表间定义约束以确保参照和基数约束。使用继承使得为父类型定义的函数可以用于属于子类型的对象，并且允许对数据库系统进行有序扩展以包括新的类型。

9.2.3　引用类型

面向对象的程序设计语言提供了引用对象的能力，类型的一个属性可以是对属于指定类型的对象的引用。例如，对人的引用属于类型 ref（*Person*），*Document* 类型的作者列表字段可以被重新定义为：

author-list **setof**（**ref**（*Person*））

即对 *Person* 对象的引用的集合。

对于表中的元组也可以有引用，对 *people* 表中元组的引用具有类型 ref（*people*）。可以用表的主码来实现对表中元组的引用。另一种方法是，表中的每一个元组可以有一个元组标识符作为隐含属性，对元组的引用简单地就是这个元组的标识符。子表隐含地继承这个元组标识符属性，就像它从父表中继承其他属性一样。

符号 ref 是 Illustra 数据库系统中所使用的。在 SQL-3 的初步草稿中“ref（*Person*）”被写做“*Person* identity”。元组和对象标识符分别可以通过特殊属性 identity 和 oid 来访问。如果我们允许对属于某种类型的对象进行引用，SQL-3 标准可能要求这个数据类型必须被申明为“with oid”。同样，如果我们允许对属于某个表的元组进行引用，这个表也可能要求被申明为“with identity”。

9.3　与复杂类型有关的查询

本节介绍 SQL 查询语言的一个扩展，它的目的是为了处理复杂类型。从一个简单的例子开始：找出每个文档的名称和发行年份。下面的查询实现了这项任务：

select *name*, *date*.*year*
from *doc*

注意我们使用点号引用了复合属性 *date* 的 *year* 字段（SQL-3 建议使用双点号“..”而不是点号）。

9.3.1　以关系为值的属性

扩展 SQL 允许用于计算关系的表达式出现在任何关系名可以出现的地方，比如 from 子句。这种可自由使用子表达式的能力使得充分利用嵌套关系结构成为可能，如下例所示。

假定有一个如下模式的关系 *pdoc*：

create table *pdoc*
（*name MyString*,
author-list **setof**（**ref**（*people*））,

date MyDate,
keyword-list **setof** (*MyString*))

如果想要找出关键词中有“database”的所有文档，可以使用以下查询：

select *name*
from *pdoc*
where “*database*” **in** *keyword-list*

注意在语句中我们使用了以关系为值的属性 *keyword-list*，该属性所处的位置在无嵌套关系的 SQL 中是要求一个 select-from-where 的子表达式。

现在，假定我们想得到一个关系，它包含形为“文档名，作者名”的对，对应每个文档和文档的每个作者。可以使用下面的查询：

select *B*.*name*, *Y*.*name*
from *pdoc* **as** *B*, *B*.*author-list* **as** *Y*

由于 *pdoc* 的 *author-list* 属性是一个以集合为值的字段，因此可以用在本来需要一个关系的 from 子句中。

聚集函数（如 min、max 和 count）以一个值的集合体作为参数并返回单个值作为结果，它们可以应用于任何以关系为值的表达式。查询“找出每个文档的名称以及作者人数”可以写成如下形式：

select *name*, **count** (*author-list*)
from *pdoc*

由于 *author-list* 是一个以集合为值的属性，对于每个作者包含了一个元组，因此对集合进行 count 聚集就得到作者人数。

9.3.2　路径表达式

用来引用复合属性的点号也可以用于引用类型。假定已有早些时候定义过的表 *people*，同时还有一个表 *phd-students* 定义如下：

create table *phd-students*
(*advisor* **ref** (*people*))
under *people*

那么就可以使用下面的查询来找出所有博士生的导师姓名：

select *phd-students*.*advisor*.*name*
from *phd-students*

由于 *phd-students* 是一个对 *people* 表中元组的引用，上述查询中的属性 *name* 就是 people 表中元组的属性 *name*。引用可以用来隐藏连接操作。在上面的例子中，如果没有使用引用，

则 *phd-students* 的 *advisor* 字段就会作为 *people* 表的一个外码。这样，要找出一个博士生的导师姓名，就需要将关系 *phd-students* 与 *people* 基于这个外码显式地做一个连接。使用引用在很大程度上简化了查询。

形如“*students.advisor.name*”这样的表达式称为路径表达式。在上面的例子中，路径表达式中的每个属性都只有单个值（在 *advisor* 的情况下为一个引用）。通常情况下，路径表达式中的属性可以是一个集合体，例如一个集合或多重集合。如果想要得到关系 *pdoc* 中文档的所有作者名，可以书写如下查询：

select $Y.name$
from $pdoc.author\text{-}list$ **as** Y

变量 Y 包括关系 *pdoc* 中每个文档的每位作者。

9.3.3　嵌套与解除嵌套

将一个嵌套关系转换成为 1NF 的过程称为解除嵌套。关系 *doc* 有 *author-list* 和 *keyword-list* 两个属性，这两者都是嵌套关系，同时关系 *doc* 另外还有 *name* 和 *date* 两个属性，它们都不是嵌套关系。假定想要将该关系转化为单个平面关系，使其不包含嵌套关系或者结构类型作为属性，可以使用以下查询来完成这个任务：

select *name*, A **as** *author*, *date.day*, *date.month*, *date.year*, K **as** *keyword*
from *doc* **as** B, $B.author\text{-}list$ **as** A, $B.keyword\text{-}list$ **as** K

from 子句中的变量 B 被声明以 *doc* 为取值范围，变量 A 被声明以该文档的 *author-list* 中的作者为取值范围，同时 K 被声明以该文档的 *keyword-list* 中的关键词为取值范围。图 9-1（见 9.1 节）显示了关系 *doc* 的一个实例，图 9-2 显示了上述查询结果形成的 1NF 关系。

反向过程即将一个 1NF 关系转化为嵌套关系称为嵌套。嵌套可以用对 SQL 分组的一个扩展来完成。在 SQL 分组的常规使用中，需要对每个组（逻辑上）创建一个临时的多重集合关系，然后在这个临时关系上应用一个聚集函数。如果不应用聚集函数而只返回这个多重集合，我们就可以创建一个嵌套关系。假定有一个 1NF 关系 *flat-doc*，如图 9-2 所示，下面的查询在属性 *keyword* 上对关系进行了嵌套：

select *title*, *author*, (*day*, *month*, *year*) **as** *date*, **set** (*keyword*) **as** *keyword-list*
from *flat-doc*
groupby *title*, *author*, *date*

在图 9-2 中关系 *flat-doc* 上执行这个查询的结果如图 9-4 所示。如果我们同时要对作者属性进行嵌套，从而使图 9-2 中的 1NF 表 *flat-doc* 转化回图 9-1 中所示的嵌套表 *doc*，可以使用如下查询：

select *title*, **set** (*author*) **as** *author-list*, (*day*, *month*, *year*) **as** *date*,
　　set (*keyword*) **as** *keyword-list*
from *flat-doc*
groupby *title*, *date*

title	author	date	keyword-list
		(*day*, *month*, *year*)	
salesplan	*Smith*	(1, *April*, 89)	{*profit*, *strategy*}
salesplan	*Jones*	(1, *April*, 89)	{*profit*, *strategy*}
status report	*Jones*	(17, *June*, 94)	{*profit*, *personnel*}
status report	*Frick*	(17, *June*, 94)	{*profit*, *personnel*}

图 9-4　*flat-doc* 关系的一个部分嵌套版本

9.3.4　函数

对象-关系系统允许由用户定义函数，这些函数既可以用一种程序设计语言如 C 或 C++ 来定义，也可以用某种数据操纵语言如 SQL 来定义。首先考虑使用扩展 SQL 中的函数定义。

假定想定义这样一个函数：给定一个文档，返回其作者的人数。可以定义这个函数如下：

create function *author-count* (*one-doc Document*)
returns integer as
select count (*author-list*)
from *one-doc*

这里 *Document* 是一个类型名。这个函数用单个文档对象来调用，select 语句同关系 *one-doc* 一起执行，这个关系仅包括单个元组，即函数的参数。这个 select 语句的结果是单个值，严格来讲，它是一个只有单个属性的元组，其类型被转化为一个值。

上面的函数可以使用在如下查询中，该查询返回具有多于一个作者的所有文档的名称：

select *name*
from *doc*
where *author-count* (*doc*) > 1

注意，上面的 SQL 表达式中，尽管在 from 子句中 *doc* 是指一个关系，但在 where 子句中它隐含地被视为一个元组变量，因此它可以用来作为 *author-count* 函数的一个参数。

一般来说，一个 select 语句可以返回一个值集合。如果函数的返回类型是一个集合体类型，那么这个函数的结果就是整个集合体；如果返回类型不是集合体类型，正如上面例子中的情形，则 select 语句所生成的集合体应当仅包含一个元组，并将其作为结果返回。如果在 select 语句的结果中有多于一个的元组，系统有两种选择：将这种情况作为一个错误处理，或者从集合体中任意选择一个元组作为结果返回。

有些数据库系统允许我们使用如 C 或 C++ 这样的程序设计语言来定义函数。用这种方式定义的函数比用 SQL 定义的函数效率更高，并且有些无法用 SQL 完成的计算能够用这些函数来执行。使用这些函数的例子如在一个元组的数据上做一个复杂的算法。

用程序设计语言定义的函数在数据库系统的外部编译，它们需要被装入并与数据库系统代码一起执行。这个过程要冒一定的风险，因为程序中的错误可能会破坏数据库的内部结构，并且可能绕过数据库系统的存取控制功能。某些被设计为可扩展的系统，如 Illustra，通常允许这种函数，然而那些强调数据安全性的系统则不允许。

使用用程序设计语言定义的函数看起来与使用嵌入 SQL 没什么不一样，使用嵌入 SQL 时数据库查询包含在一个通用程序中。但是它们之间还是有一个重要差别，在嵌入 SQL 中，用户程序将查询传送给数据库系统执行，结果返回给该程序（一次一个元组），因此，用户书写的代码永远不会需要访问数据库本身，于是操作系统就可以保护数据库不被任何用户进程所存取。

当在查询中使用用户编码的函数时，要么这些代码必须由数据库系统本身运行，要么该函数所操作的数据必须被拷贝到一个分离的数据空间中。第二种方法增加了系统开销，第一种方法则诱发了潜在的脆弱性，这同时表现在完整性方面（用户代码中存在的错误）和安全性方面（恶意插入的操作作为用户代码的一部分）。

9.4 复杂值和复杂对象的创建

迄今为止，我们已经看到了如何创建复杂类型定义以及使用复杂类型定义的查询关系。我们现在来看如何创建和更新具有复杂类型的关系中的元组。

可以创建 *doc* 关系所定义类型的一个元组如下：

("salesplan", **set**("Smith", "Jones"), (1, "April", 89), **set**("profit", "strategy"))

上面的元组展示了创建复杂类型的几个方面。为复合属性 *date* 创建值的方法是：将其各个属性——日、月和年——在圆括号中列出；创建集合值属性 *author-list* 和 *keyword-list* 是通过在圆括号中列举它们的元素并在前面加上关键词 set。

可以用许多方法来使用所构造的复杂值，如下所示。如果想将前面的元组插入到关系 *doc* 中，可以执行以下语句：

insert into doc
values ("salesplan", set("Smith", "Jones"), (1,"April", 89),**set**("profit", "strategy"))

我们也可以在查询中使用复杂值。例如，在查询中任何需要集合的地方就可以列举出一个集合，如下面的查询所示：

select *name*, *date*
from *doc*
where *name* **in set** ("salesplan", "opportunities", "risks")

上述查询找出所有那些名称为 "salesplan"，"opportunities" 或 "risks" 其中之一的文档的名称和日期。

多重集合值的创建与集合值类似，将关键词 set 换成 multiset 即可。值的列表或数组的创建类似于元组值。

要创建新的对象，我们可以使用构造函数。对一个类型为 T 的对象其构造函数是 $T()$。当调用它时它创建一个新的、未初始化的 T 类型对象，填写 oid 字段，然后返回这个对象，该对象的其他字段随后也必须被初始化。

可以使用通常的 SQL update 语句来完成复杂关系的更新，这种更新与 1NF 关系的更新非常类似。

9.5　面向对象数据库与对象-关系数据库的比较

我们已经研究了建立在持久化程序设计语言上的面向对象数据库，也研究了建立在关系模型之上的面向对象的对象-关系数据库。这两种类型的数据库系统在市场上都存在，数据库设计者要选择那种适合应用需求的系统。

程序设计语言的持久化扩展和对象-关系系统有着不同的市场目标。SQL 语言的声明性特征和有限的能力（与程序设计语言相比）为防止程序设计错误对数据造成破坏提供了很好的保护，同时使得一些高级优化，例如减少 I/O，变得相对简单。（我们将在第 12 章讨论关系表达式的优化。）对象　关系系统的目标在于通过使用复杂数据类型而使得数据建模和查询更加容易。典型的应用有复杂数据（包括多媒体数据）的存储和查询等。

然而，对于某些类型的应用（包括那些主要在内存中运行以及那些对数据库进行大批量访问的应用）来说，一个说明性语言（如 SQL）会带来显著的性能损失。持久化程序设计语言的目标就是具有很高性能要求的这些形式的应用。持久化程序设计语言提供了对持久数据的低开销存取，并且取消了数据转换的要求（如果这些数据要用程序设计语言来进行操纵的话）。但是，持久化程序设计语言对由于程序错误而引起的数据破坏更为敏感，且通常没有强大的查询能力。它们典型的应用包括 CAD 数据库。

这些不同种类的数据库系统的能力可以总结如下：

- 关系系统。简单数据类型、功能强大的查询语言、高保护性。
- 以持久化程序设计语言为基础的 OODB。复杂数据类型、与程序设计语言集成、高性能。
- 对象-关系系统。复杂数据类型、功能强大的查询语言、高保护性。

这些描述具有普遍性，但是请记住对有些数据库系统来说它们的分界线是模糊的。例如，有些以持久化程序设计语言为基础的面向对象数据库系统是在一个关系数据库系统之上实现的，这些系统的性能可能比不上那些直接建立在存储系统之上的面向对象数据库系统，但这些系统却提供了关系系统所具有的较强的保护能力。

9.6　总结

复杂数据类型（如嵌套关系）对于许多应用中的复杂数据的建模很有好处。这种关系模型的扩展超出了 1NF 的范畴，使之允许非原子类型。对象-关系系统将基于扩展关系模型的复杂数据与面向对象的一些概念（如对象标识和继承）结合在一起。SQL 数据定义和查询语言也通过扩展来处理复杂类型和面向对象特性。我们已经看到了这种扩展的数据定义语言及查询语言的一些特征，特别如支持集合值属性、继承、对象及元组引用。

习题

9.1　考虑下面的数据库模式

$$Emp = (ename, \textbf{setof}(Children), \textbf{setof}(Skills))$$
$$Children = (name, Birthday)$$
$$Birthday = (day, month, year)$$
$$Skills = (type, \textbf{setof}(Exams))$$
$$Exams = (year, city)$$

假定类型为 setof（*Children*）、setof（*Skills*）和 setof（*Exams*）的属性分别具有属性名 *ChildrenSet*、*SkillsSet* 和 *ExamsSet*。用扩展 SQL 为关系 *emp*（*Emp*）分别书写以下查询：

(a) 找出所有那些有一个孩子且孩子的生日在三月的雇员的名字。

(b) 找出那些在城市“Dayton”中做过技能种类为“typing”的测验的所有雇员。

(c) 列出在关系 *emp* 中的所有技能种类。

9.2 重新设计习题 9.1 中的数据库使之满足第一范式和第四范式。列出你所设想的任何函数依赖和多值依赖，同时列出在第一和第四范式的模式中应该存在的所有参照完整性约束。

9.3 对 9.2.2 节中的表 *people* 以及在其下创建的表 *students* 和 *teachers* 的模式，给出一个表示相同信息的第三范式的关系模式。回忆在子表上的那些约束，给出必须施加在该关系模式上的所有约束，以使得关系模式的每个数据库实例也能够被带有继承的此模式的一个实例所表示

9.4 一个汽车租赁公司为其当前车队中的所有车辆维护着一个车辆数据库。对于所有的车辆，数据库中包括的信息有车辆识别号、牌照号、制造商、型号、购买日期以及颜色，对于一些类型的车辆还包括有特殊的数据：

- 卡车：载货容量。
- 跑车：马力、对租用者的年龄限制。
- 厢式货车：乘客数目。
- 越野车：离地间隙、驱动类型（四轮还是两轮驱动）。

为这个数据库构造一个扩展 SQL 的模式定义，适当的时候使用继承。

9.5 解释类型 x 与引用类型 ref（x）之间的区别，什么情况下你会选择使用引用类型？

9.6 比较嵌入式 SQL 的使用与 SQL 中通用程序设计语言定义的函数的使用，对于这两种方式分别说明在什么情况下你将选择使用它们？

9.7 假定你受聘为客户的应用选择一个数据库系统。对于下面这些应用中的每一种，说明你将推荐哪种类型的数据库（关系数据库、基于持久化程序设计语言的 OODB 还是对象-关系数据库，不用指定某个商业产品），并且说明你的推荐是正确的。

(a) 为飞机制造商开发的计算机辅助设计系统。

(b) 为政府机关设计的对候选人捐款进行追踪的系统。

(c) 为高速公路建设项目负责人设计的辅助系统。

(d) 支持电影制作的信息系统。

文献注解

嵌套关系模型的介绍在文献 Makinouchi［1977］与 Jaeschke 和 Schek［1982］中，文献 Fischer 和 Thomas［1983］、Zaniolo［1983］、Ozsoyoglu 等［1987］、Van Gucht［1987］和 Roth 等［1988］中介绍了各种代数查询语言。关于嵌套关系中空值的管理在文献 Roth 等［1989］中讨论，设计和规范化问题的讨论见文献 Ozsoyoglu 和 Yuan［1987］、Roth 和 Korth［1987］、Mok 等［1996］。在文献 Abiteboul 等［1989］中有一系列关于嵌套关系的论文，文献 Sacks-Davis 等［1995］中展示了嵌套关系模型在包含文档的数据库中的一个应用。

若干种 SQL 的面向对象扩展已经被提出。POSTGRES（Stonebraker 和 Rowe［1986］、Stonebraker［1986a，1987］、Stonebraker 等［1987a，1987b］）是对象-关系系统的一个早期

实现。作为 POSTGRES 的后续，Illustra 是一个商业对象-关系系统。惠普公司的 Iris 数据库系统（Fishman 等［1990］和 Wilkinson 等［1990］）在关系数据库系统之上提供了面向对象扩展，Iris 支持一种被称为 Object SQL（OSQL）的语言，它是 SQL 的一个面向对象扩展。文献 Bancilhon 等［1989］和 Deux 等［1991］中描述了 O_2 查询语言，它是 O_2 面向对象数据库中实现的一个 SQL 的面向对象扩展。UniSQL 在文献［UniSQL 1991］中描述。面向对象数据库系统的触发器在以下文献中讨论：Gehani 和 Jagadish［1991］、Gehani 等［1992］、Jagadish 和 Qian［1992］。

XSQL 是 Kifer 等［1992］中提出的一个 SQL 的面向对象扩展。文献 Kim［1995］包含了关于现代数据库系统的一系列论文，其中包括关于 OSQL 和 ZQL［C++］的描述，后者将说明性查询与 C++ 集成在一起。SQL 的面向对象扩展的标准化项目正作为正在研制的 SQL-3 标准的一部分进行着，查看本书的万维网主页（http://www.bell-labs.com/topic/books/db-book）可了解 SQL-3 的更新情况，包括 SQL-3 与本章中所使用的语法之间的差别。

第 10 章　存储结构和文件结构

前面几章着重讲述了数据库的较高层模型。在概念层或逻辑层上，我们认为关系模型中的数据库是表的集合。数据库的逻辑模型是数据库用户所应关注的层次。数据库系统的目标是简化和协助对数据的存取。数据库系统的用户不必被系统实现的物理细节所困扰。

本章以及第 11、12 章将描述实现前几章所提出的数据模型和语言的不同方法。我们将从基本存储介质（如磁盘和磁带系统）的特性开始，然后定义几种不同的数据结构，这些数据结构使我们能够快速地存取数据。我们还将考虑几种可供选择的数据结构，各种数据结构适合于不同类型的数据存取。数据结构的最终选择依赖于系统的使用和特定机器的物理特性。

10.1　物理存储介质概览

大多数计算机系统中存在多种数据存储类型。存储介质可以根据数据存取的速度、购买介质时每单位数据的成本和介质的可靠性进行分类。通常可获得的介质有这样一些：

- 高速缓冲存储器。高速缓冲存储器是最快最昂贵的存储介质。高速缓冲存储器一般很小，它的使用由操作系统来管理。在数据库系统中，我们将不考虑高速缓冲存储器的存储管理。
- 主存储器。用于存放可被处理的数据的存储介质是主存储器。通用机器指令在主存储器上执行。尽管主存储器可以包含若干 MB 的数据，但通常还是因为它太小（或太昂贵）而不能存储整个数据库。如果发生电源故障或者系统崩溃，主存储器中的内容一般会丢失。
- 快闪存储器。也称为电可擦可编程只读存储器（*EEPROM*）。快闪存储器不同于主存储器的地方是在电源故障发生时数据可被保存下来。从快闪存储器读数据的时间小于 100 纳秒（1 纳秒等于 0.001 微秒），大致等于从主存储器中读数据的时间。然而，向快闪存储器写数据是非常复杂的——数据写入一次，大约需要 4～10 微秒，但数据不能被直接覆盖，要想覆盖已经被写过的快闪存储器，必须一次性擦除整个快闪存储器，然后它才可以再被写入一次。快闪存储器的一个缺点是它只支持有限的擦除次数，其范围从 10 000～1 百万。在低成本计算机系统中，例如在嵌入至其他设备的计算机系统中，快闪存储器作为磁盘的替代物来存储少量数据（5MB～10MB）已经非常流行。
- 磁盘存储器。用于长期联机数据存储的主要介质是磁盘。通常整个数据库都存储在磁盘上。为了能够访问到数据，必须将数据从磁盘移到主存储器。完成操作后，被修改过的数据则必须写回磁盘。磁盘存储器称为直接存取存储器，因为在磁盘上可以按任意顺序读取数据（与顺序存取的存储器不同）。在发生电源故障或系统崩溃时，磁盘存储器不会丢失数据。磁盘存储设备自己有时会发生故障而毁坏数据，但这种故障发生的频率远远低于系统崩溃发生的频率。
- 光盘存储器。光盘存储器最流行的形式是只读光盘（*CD-ROM*）。数据通过光学方法存储在光盘上，并且可以被激光器读取。用于 CD-ROM 存储器的光盘是不可写的，但是可以提供预先记录的数据，并且可以装入驱动器或从驱动器中移走。另一种光盘存储器的形式是“一次写，多次读”（*WORM*）光盘，它允许写入数据一次，但是不允许擦除和重写这些数据。这种介质用于数据的归档存储。此外还有磁光结合的存储设备，它们使用光学方法读取以磁方法编码的数据，并且允许对旧数据的覆盖。

多数光盘存储类型允许从驱动器中移走一张光盘，并且用其他光盘来替换。自动光盘机系统含有少量驱动器和大量可按要求（通过机械手臂）自动装入某一驱动器的光盘。

• 磁带存储。磁带存储主要用于备份数据和归档数据。尽管磁带比磁盘便宜得多，但是存取数据也比磁盘慢得多，这是因为磁带必须从头顺序存取。正因为如此，磁带存储被称为顺序存取存储。磁带具有很大的容量（5GB 的磁带是很常见的），并且可以从磁带设备中移出，这些都有利于进行便宜的归档存储。自动磁带机用于容纳大量数据，比如卫星传回的遥感数据，这些数据在不久的将来将达到几百 TB（$1TB = 10^{12}B$）。

根据不同存储介质的速度和成本，可以把它们按层次结构组织起来（图 10-1）。层次越高，这种存储介质的成本就越高，但速度也越快。当沿着层次结构向下，存储介质每比特的成本也在下降，但存取时间会增加。这种权衡是合理的：如果某一存储系统比另一种存储系统快而且便宜（其他特性相同），那么就没有理由使用那种又慢又昂贵的存储系统。实际上，当磁带和半导体存储器变得更快更便宜时，很多早期存储设备，包括纸带和磁心存储器，都进了博物馆。当磁盘还很昂贵并且只有很小的存储容量时，我们用磁带存储正在使用的数据。然而到了今天，几乎所有正在使用的数据都存储在磁盘上，只在极少数情况下存储在磁带或自动光盘机中。

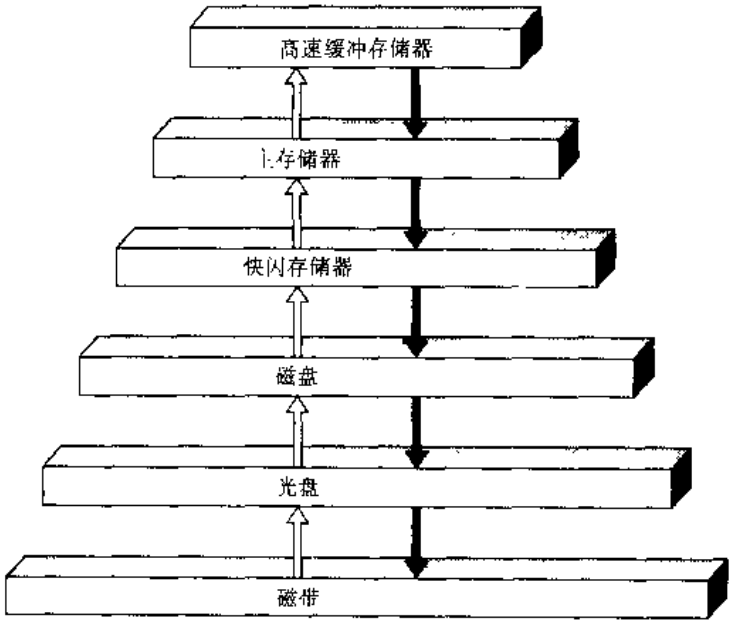


图 10-1　存储设备层次结构

最快的存储介质称为基本存储，例如高速缓冲存储器和主存储器。层次结构中基本存储的下一层介质称为辅助存储，或联机存储，例如磁盘。层次结构中最底层的介质称为第三级存储，或脱机存储，如磁带和自动光盘机。

不同存储系统除了速度和成本不同之外，还存在一个存储易失性的问题。易失性存储是指设备失去能源后将丢失所有内容。在缺少由昂贵的电池和发电机组成的后备能源系统时，为了保存数据，必须将数据写到非易失性存储中去。在图 10-1 所示的层次结构中，从主存储器向上的存储系统都是易失性存储，而主存储器之下的存储系统都是非易失性存储。我们将在第 15 章讲到这个问题。

10.2　磁盘

磁盘为现代计算机系统提供了大容量的辅助存储。一个磁盘的存储容量从 10MB 到 10GB

不等。一个典型的大型商业数据库需要数百个磁盘。

10.2.1　磁盘的物理特性

从物理特性讲，磁盘是相对简单的（图 10-2）。磁盘的每一个盘片是扁平的圆盘。它的两个表面都覆盖着磁性物质，信息就记录在表面上。盘片由硬金属或玻璃制成，被磁性物质所覆盖（通常是两面）。我们称这样的磁盘为硬盘，以区别于用柔性物质制成的软盘。

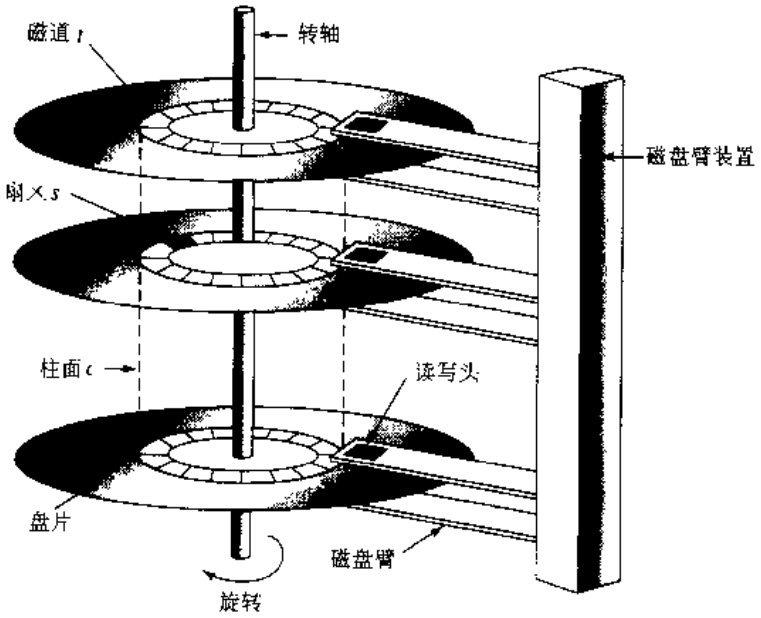


图 10 2　可移动磁头磁盘装置

当使用磁盘时，驱动马达使磁盘以很高的恒定速度旋转（通常为每秒 60、90 或 120 转）。一个读写头恰好位于盘片表面的上方。盘片的表面被逻辑地划分为磁道，磁道又被划分为扇区。扇区是从磁盘读出和写入信息的最小单位。根据磁盘的不同类型，一个扇区的大小可从 32～4096 字节不等，但通常是 512 字节。每个磁道有 4～32 个扇区，每个盘片表面有 20～1500 个磁道。通过朝磁性物质磁化的方向反转，读写头将信息存储到扇区上。盘片的一个表面可能包含数百个构成同心圆的磁道和数千个扇区。

磁盘的每个盘片的每一面都有一个读写头，读写头通过在盘片上移动来访问不同的磁道。一个磁盘通常包括很多个盘片，所有磁道的读写头安装于一个称为磁盘臂的装置上，并且一起移动。安装在一个转轴上的所有磁盘盘片和安装在磁盘臂装置上的所有读写头总称为磁头　磁盘装置。因为所有盘片上的读写头一起移动，所以当一个盘片的读写头在第 i 个磁道上时，那么所有其他盘片的读写头也都在各自盘片的第 i 个磁道上。因此，所有盘片的第 i 个磁道合在一起称为第 i 个柱面。

磁盘盘片的直径可从 1.8～14 英寸不等。现在占主导地位的是 5.25 英寸和 3.5 英寸的磁盘，因为尽管更大的磁盘能提供更大的存储容量，但也更昂贵，且寻道时间也更长（源于寻道距离变长）。

磁盘控制器作为计算机系统和实际磁盘驱动器硬件之间的接口，接受来自高层次的读写扇区的命令，然后产生动作，如移动磁盘臂到正确的磁道去实际读写数据。磁盘控制器为它所写的每个扇区附加校验和，校验和是由写到扇区中的数据计算得到的。当读取一个扇区时，用读到的数据再次计算校验和，并且与存储的校验和比较。如果数据被破坏，则新计算出的校验和

与存储的校验和不一致的可能性就很高。一旦这样的错误发生了，磁盘控制器就会重读几次，如果错误还继续发生，磁盘控制器就会发一个读操作失败的信号。

磁盘控制器执行的另一个有趣的任务是坏扇区的重映射。当初始格式化磁盘，或试图写一个扇区时，如果磁盘控制器检测到一个被损坏的扇区，它会把这个扇区逻辑上映射到另一个物理位置（从为此目的而留出的额外扇区中分配）。重映射被记录在磁盘或非易失性存储器中，且写操作在新的位置上执行。

图 10-3 表示磁盘是如何与计算机系统相连接的。与其他存储设备一样，磁盘通过高速互连与计算机系统或磁盘控制器相连接。一般小型计算机系统接口（SCSI）用来把磁盘与个人计算机和工作站相连，大型机和服务器系统则一般通过更快且更昂贵的总线与磁盘相连接。

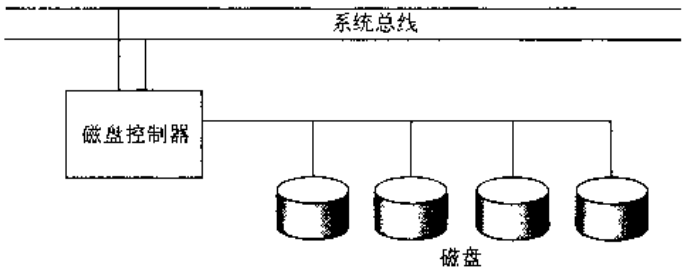


图 10-3　磁盘子系统

为了增大记录密度，读写头应尽可能地靠近磁盘盘片的表面。读写头一般浮于盘片表面之上几微米，被气垫所支撑。因为读写头离盘片表面靠得很近，所以盘片必须被制作得很平。读写头损坏一直是一个问题。如果读写头接触到盘片的表面（例如电源故障），读写头会刮掉磁盘上的记录介质，从而破坏掉存放在那里的数据。通常，读写头接触盘片表面所刮掉的介质会散落到其他盘片及其读写头之间，结果引起更多的损坏。正常情况下，一个读写头的损坏将导致整个磁盘失效，从而需要更换磁盘。

固定读写头的磁盘对每一个磁道都有一个独立的读写头，这样的结构可使计算机快速地从一个磁道切换到另一个磁道，而不需要移动读写头装置。但是由于它需要大量的读写头，从而使设备变得异常昂贵。一些磁盘系统有多个磁盘臂，允许同一时间访问同一盘片的多个磁道。固定读写头的磁盘和多臂磁盘曾经用于高性能的大型机系统，但在今天已经很少使用了。

10.2.2　磁盘性能的度量标准

磁盘质量的主要度量标准是容量、存取时间、数据传输率和可靠性。

存取时间是从发出读写请求到数据开始传输之间的时间。为了存取（即读或写）磁盘上指定扇区的数据，磁盘臂首先必须移动，以定位到正确的磁道，然后等待磁盘旋转到指定扇区出现在它下方。磁盘臂重定位的时间称为寻道时间，它随磁盘臂移动距离的增大而增大。一般寻道时间在 2～30 毫秒之间，依赖于目的磁道距离磁盘臂的初始位置有多远。较小的磁盘因为移动距离较短而寻道时间较小。

平均寻道时间是寻道时间的平均值，是基于一系列（均匀分布的）随机请求衡量得到的，大约是最坏情况下寻道时间的 1/3。如果读写头在存取前位于盘片的一端（内端或外端），平均寻道时间是最坏情况下寻道时间的 1/2；如果读写头恰好位于最内端磁道和最外端磁道之间的中点上，平均寻道时间是最坏情况下寻道时间的 1/4。对读写头所能处在的所有位置进行平均，平均寻道时间是最坏情况下寻道时间的 1/3。

一旦寻道结束，等待被存取的扇区出现在读写头下所花费的时间称为旋转等待时间。现在磁盘转速一般在每秒 60～120 转之间，等价于每转 8.35～16.7 毫秒。通常情况下，磁盘需要旋转半周才能使所要存取的扇区的开始处处于读写头的下方，因此磁盘的平均旋转等待时间是磁盘旋转一周时间的 1/2。

存取时间是寻道时间和旋转等待时间的总和，范围在 10～40 毫秒之间。一旦被存取数据的第一个扇区在读写头下经过，数据传输就开始了。数据传输率是从磁盘获得数据或者向磁盘存储数据的速率。当前磁盘系统所支持的传输率在每秒 1MB～5MB 之间。

最后一个经常使用的磁盘度量标准是平均故障时间，这是磁盘可靠性的度量标准。磁盘（或其他任何系统）的平均故障时间是预期系统无故障连续运行的平均时间。现在通常磁盘平均故障时间在 30 000～800 000 小时之间，大约为 3.4～91 年。

10.2.3　磁盘块存取的优化

在多数操作系统中，磁盘 I/O 请求是由文件系统和虚拟内存管理器产生的。每个请求指定了要存取的磁盘地址，这个地址是以块号的形式提供的。块是一个盘片的一条磁道内几个连续扇区构成的序列。块大小从 512 字节到若干 KB 不等。数据在磁盘和主存储器之间以块为单位传输。文件系统管理器的较低层将块地址转换成硬件层的柱面号、盘面号和扇区号。

因为存取磁盘上的数据比存取主存储器中的数据要慢几个数量级，所以要特别注意提高存取磁盘块的速度。可以在主存储器中对块进行缓冲以满足未来请求，这将在 10.5 节中讨论。这里讨论其他几种技术。

- 调度。如果需要把一个柱面上的几个块从磁盘传输到主存储器，那么按块经过读写头的顺序发出存取块的请求就可以节省存取时间。如果要存取的块在不同的柱面上，则按照使磁盘臂移动最短距离的顺序发出存取块的请求是非常有利的。*磁盘臂调度算法*试图把对磁道的访问按照能增加所处理的存取数量的方式排序。通常使用的算法是*电梯算法*，因为该算法的工作方式与大多数电梯非常相似。假设一开始，磁盘臂从最内端的磁道向最外端移动，遇到有存取请求的磁道，它就在那个磁道停下，为这个磁道的请求提供服务，然后继续向外移动，直到没有更外层磁道的请求在等待。这时，磁盘臂掉转方向，开始向内侧移动，依然在每个有请求的磁道处停下，直到没有更靠近中心的磁道上有请求在等待。此时，它掉转方向，开始一个新的周期。磁盘控制器通常对读请求进行重排序以提高性能，因为它最清楚磁盘块的组织、磁盘盘片的旋转位置和磁盘臂的位置。
- 文件组织。为了减少存取块的时间，可以按照与所预期的数据存取方式最接近的方式来组织磁盘上的块。例如，如果希望一个文件被顺序存取，那么理想情况下应该使文件的所有块存储在连续的相邻柱面上。旧的操作系统，如 IBM 大型机操作系统，给程序员提供了对文件位置的精确控制，允许程序员保留一组柱面来存储一个文件。然而，这种控制给程序员或系统管理员带来了决策上的负担，例如他们需要决定给一个文件分配多少柱面，并且当数据插入文件或从文件中删除时，重新组织文件还需要昂贵的代价。

随后的操作系统，如 Unix 和个人计算机操作系统，对使用者隐藏了磁盘的组织，而由操作系统内部来管理空间的分配。但采用这种方式经过一段时间后，一个连续的文件将变得碎片化，即它的块散布在整个磁盘上。为了减少碎片，系统可以先对磁盘上的数据进行一次备份，然后再恢复整个磁盘。恢复操作是将每个文件的块连续地（或几乎连续地）写回。一些系统（如 MS-DOS）提供工具扫描整个磁盘，然后移动块以减少碎片。通过这种技术所实现的性能提高是非常显著的。但是当这些工具在执行时，系统通常是不能使用的。

• 非易失性的写缓冲区。因为主存储器中的内容在发生电源故障时将全部丢失，所以关于数据库更新的信息必须被记录到磁盘上，这样信息才能在系统崩溃时得以保存。更新操作密集的数据库应用的性能，如事务处理系统的性能，高度倚赖于磁盘写操作的速度。

可以使用非易失性随机存取存储器（非易失性 RAM）大幅度地加快写磁盘的操作。非易失性 RAM 的内容在发生电源故障时不会丢失。一种实现非易失性 RAM 的常用方法是使用有后备电池的 RAM。非易失性 RAM 的思想是，当数据库系统（或操作系统）请求往磁盘上写一个块时，磁盘控制器将这个块写到非易失性 RAM 缓冲区中，然后立即通知操作系统写操作已成功完成。当磁盘上没有其他请求，或者当非易失性 RAM 缓冲区满时，磁盘控制器将这些数据写到磁盘上相应的目标地址处。可见，当数据库系统请求写一个块时，只有在非易失性 RAM 缓冲区满时，才有一个明显的延迟。从系统崩溃中进行恢复时，非易失性 RAM 中所有被缓冲的写操作将被写回到磁盘。

下面这个例子说明了非易失性 RAM 提高性能的程度。假设在随机（均匀分布）状态下接收到写请求，并且磁盘平均 90% 的时间内处于忙状态。如果我们有一个 50 个块大小的非易失性 RAM 缓冲区，那么每分钟只有一次写操作会发现缓冲区满了（因此不得不等待写磁盘操作的完成）。可见，在大多数情况下，写磁盘操作的执行不需要数据库系统去等待寻道和旋转延迟。如果磁盘和磁盘控制器没有非易失性 RAM 写缓冲区，一些操作系统（如 Solaris）会使用计算机系统自己的非易失性 RAM 来进行写缓冲。

• 磁盘日志。减少写等待延迟的另一种方法是使用*磁盘日志*（即一个专门用于写顺序日志的磁盘），这和非易失性 RAM 缓冲区的使用非常相似。所有对磁盘日志的存取都是顺序的，这从根本上消除了寻道时间，并且一次可以写几个连续的块，从而使写磁盘日志比随机写快上若干倍。和其他方法一样，数据也必须被写到它们在磁盘的实际位置，但是数据库系统不需要等待这种写操作的完成。系统崩溃后，系统读取磁盘日志，找到那些还没有完成的写操作，然后将它们完成。

*基于日志的文件系统*是磁盘日志方法的一种最典型的体现。数据不写回它在磁盘上的原始位置，而是文件系统跟踪块最近被写到磁盘日志的哪个地方，然后从这个地方取得数据。磁盘日志定期进行压缩，删除已被覆盖的旧的写操作。这种方法提高了写操作的性能，但是对经常更新的文件而言会使其高度碎片化，就像我们在前面注意到的，这样的碎片化增加了顺序读取文件时的寻道时间。

10.3 RAID

近几年来，磁盘驱动器变得越来越小且越来越便宜，因而一个计算机系统连接大量的磁盘在经济上是可行的。现在小磁盘大量生产且相对便宜，使用大量廉价的小磁盘存储数据比使用少量昂贵的大磁盘要合算得多。

如果磁盘能被并行存取的话，在一个系统中使用大量磁盘可以提供提高数据读写速率的机会。此外，这种组织方式提供了提高数据存储可靠性的潜力，因为冗余信息可以被存储在多个磁盘中，这样一个磁盘的故障不会导致数据的丢失。现在人们又提出了统称为*廉价磁盘冗余阵列*（RAID）的多种磁盘组织技术，以解决性能和可靠性问题。

过去，由小的廉价磁盘组成的 RAID 被视为是大而昂贵的磁盘的一种合算的替代方法。今天，大磁盘已经很少使用，RAID 被使用已不再是因为经济原因，而是因为有更高的可靠性和更高的数据传输率。因此 RAID 中的 I 代表独立，而不再是廉价。

10.3.1　通过冗余提高可靠性

让我们首先考虑可靠性问题。N 个磁盘组成的集合中某个磁盘发生故障的概率比特定的一个磁盘发生故障的概率要高。假设一个磁盘的平均故障时间为 100 000 小时，或大约 11.4 年，并假设在一个给定的时间单位中，磁盘发生故障的可能性不随时间而改变。（这个假设在实际中是不合理的，因为磁盘故障发生的可能性在它第一次使用时和靠近磁盘寿命终点时使用最高，此两者之间却相对较低。做这样的假设是为了简化问题。）这样，由 100 个磁盘组成的阵列发生磁盘故障的平均时间为 100 000/100 = 1000 小时，或 41.66 天，这个时间一点也不长！如果我们只存储了数据的一个拷贝，那么任何一个磁盘发生故障都将导致大量数据的丢失（正如前面在 10.2.1 节中讨论的那样），并且这样高的数据丢失率是不可接受的。

引入冗余是解决可靠性问题的方法，即存储一些通常情况下不需要的额外信息，但这些信息可在发生磁盘故障时用于重建丢失的信息。这样，即使有一个磁盘发生了故障，数据也不会丢失。于是，假如只统计导致数据丢失或数据不可用的磁盘故障，那么磁盘发生故障的有效平均时间将得到增加。

实现冗余最简单（但最昂贵）的方法是复制每一个磁盘，这种技术称为镜像或影像。一个逻辑上的磁盘由两个物理磁盘组成，并且每一个写操作都要在两个磁盘上执行。如果其中一个磁盘发生了故障，数据可以从另一个磁盘读出。只有在第一个磁盘的故障被修复之前，第二个磁盘也发生了故障，数据才会丢失。

采用镜像技术的磁盘的平均故障时间（这里的故障是指数据的丢失）依赖于每个磁盘的平均故障时间和平均修复时间，平均修复时间是替换发生故障的磁盘并且恢复这个磁盘上的数据的（平均）时间。假设两个磁盘发生故障是相互独立的，即一个磁盘的故障和另一个磁盘的故障没有联系。那么，如果一个磁盘的平均故障时间是 100 000 小时，平均修复时间是 10 小时，则镜像磁盘系统的平均数据丢失时间为 $100000^2/(2*10) = 500*10^6$ 小时，或 57 000 年！（这里就不再探究推导过程，文献注解中的参考文献里提供了细节。）

读者应该意识到磁盘故障之间的独立性假设是不合理的。电源故障和自然灾害，如地震、火灾、洪水，都将导致两个磁盘在同一时间被毁坏。随着磁盘逐渐老化，发生故障的可能性也跟着增加，同时也增加了在第一块磁盘被修复时，第二块磁盘也发生故障的机会。然而尽管有这些顾虑，镜像磁盘系统还是比单一磁盘系统提供了更高的可靠性。平均数据丢失时间在 500 000～1000 000 小时（或 55～110 年）之间的镜像磁盘系统现在是可以获得的。

电源故障需要特别考虑，因为它比自然灾害的发生要频繁得多。如果在电源发生故障时没有向磁盘传输数据，那么电源故障就不必考虑。然而，即使有磁盘镜像，如果正对两个磁盘上相同的块进行写操作，并且在两个块完全写完之前发生电源故障，那么这两个块将处于不一致的状态。解决这个问题的方法是，先写一个拷贝，再写另一个。这样，两个拷贝中有一个总是一致的。当在电源发生故障后重新启动时，需要做一些额外的动作，以从不完全的写操作中恢复，这个问题将在习题 10.4 中讨论。

10.3.2　通过并行提高性能

现在让我们看看对多个磁盘进行并行存取的好处。通过磁盘镜像，处理读请求的速度将翻倍，因为读请求可以被发送到任意一个磁盘上（只要组成一对的两个磁盘都是可用的，而且一般情况下总是如此）。每个读操作的传输速率和单一磁盘系统中的传输速率是一样的，但是每单位时间内读操作的数目将翻倍。

可以通过在多个磁盘上对数据进行拆分来提高传输速率。数据拆分最简单的形式是将每个字节按比特分开，存储到多个磁盘上，这种拆分称为比特级拆分。例如，如果有一个由八个磁盘组成的阵列，并且每个字节的第 i 个比特位写到第 i 个磁盘上，则这八个磁盘组成的阵列可以被看成是一个单一的磁盘，这个磁盘的一个扇区的大小是通常大小的八倍，而且更重要的是它的存取速度是通常速度的八倍。在这样的组织结构中，由于每个磁盘都参与每次的存取（读或写），所以它每秒所处理的存取操作的总数和一个磁盘所处理的是一样的，但是在相同时间内，它每次存取所读取的数据量却是一个磁盘上的八倍。

比特级拆分可以推广到磁盘总数为 8 的倍数或能被 8 整除的情况。例如，如果我们使用由四个磁盘组成的阵列，则每个字节的第 i 个比特位和第 $4+i$ 个比特位写到第 i 个磁盘上。此外，拆分并不一定在字节的比特级上进行。比如，在块级拆分中，文件的块被拆分存储到多个磁盘上。如果使用 n 个磁盘，则文件的第 i 块存储到第（$i \bmod n$）+1 个磁盘上。其他的拆分级别也是可能的，例如把扇区按字节拆分，或把块按扇区拆分。

总之，磁盘系统中的并行有两个主要目的：

1）负载平衡多个小的存取操作（即页面存取），以提高这种存取操作的吞吐量。

2）并行执行大的存取操作，以减少大的存取操作的反应时间。

10.3.3　RAID 级别

镜像提供了高可靠性，但它十分昂贵。拆分提供了高数据传输率，但没有提高可靠性。通过利用与奇偶校验（随后会讲到）相结合的磁盘拆分想法，人们已经提出了很多以较低成本提供冗余的方案。这些方案具有不同的成本和性能之间的权衡，并且被分为若干级别，称为 RAID 级别。这里我们将描述不同的级别，如图 10-4 所示（图中 P 表示纠错位，C 表示数据的第二个拷贝）。图中的所有情况中都存储了总量为四个磁盘的数据，其余的磁盘存储用于故障恢复的冗余信息。

• RAID 0 级。指块级拆分且没有任何冗余（例如镜像或奇偶校验位）的磁盘阵列。图 10-4a 给出了一个大小为 4 的磁盘阵列。

• RAID 1 级。指带块级拆分的磁盘镜像。图 10-4b 给出了一个有镜像的组织结构，其中存储四个磁盘的数据。

• RAID 2 级。也称为内存风格的纠错码（ECC）组织结构。长期以来，内存系统使用奇偶校验位来实现错误检测。内存系统中的每个字节都有一个与之相连的奇偶校验位，它记录了这个字节中为 1 的比特位的总数是偶数（奇偶校验位 = 0）还是奇数（奇偶校验位 = 1）。如果这个字节中有一位被破坏（1 变成 0，或 0 变成 1），那么这个字节的奇偶校验位就会改变，于是与存储的奇偶校验位就不再匹配。同样的，如果存储的奇偶校验位被破坏，它就不能和计算出的奇偶校验位相匹配。因此，存储器系统可以检测到所有的 1 位错误。纠错码机制存储 2 个或更多的附加位，这样如果有一位被破坏，它还可以重建数据。通过把字节拆分存储到多个磁盘上，纠错码的思想可以直接用于磁盘阵列。例如，每个字节的第一位存储在第一个磁盘中，第二位存储在第二个磁盘中，如此下去，直到第八位存储在第八个磁盘中，同时纠错位存储在其余的磁盘中。图 10-4 图形化地表示了这种方法，其中标记 P 的磁盘存储了纠错位。这样如果一个磁盘发生了故障，可以从其他磁盘中读出字节的其余位和相关的纠错位，并用它们来重建被破坏的数据。图 10-4c 显示了大小为 4 的磁盘阵列。注意，RAID 2 级对四个磁盘的数据只需要 3 个磁盘的额外开销，而不像 RAID 1 级需要 4 个磁盘的额外开销。

• RAID 3 级。也称为位交叉的奇偶校验组织结构。RAID 3 级在 RAID 2 级的基础上进行

a) RAID 0: 无冗余拆分

b) RAID 1: 镜像磁盘

c) RAID 2: 内存风格纠错码

d) RAID 3: 位交叉奇偶校验

e) RAID 4 块交叉奇偶校验

f) RAID 5: 块交叉分布奇偶校验

g) RAID 6: P + Q 冗余

图 10-4　RAID 的级别

了改进。此类级别与内存系统不同，磁盘控制器能够检测一个扇区是否被正确的读出，所以可以使用一个单一的奇偶校验位来检错和纠错。它的思想如下：如果一个扇区被破坏，我们先能准确地知道是哪个扇区坏了，然后对扇区的每一位，通过计算其他磁盘上对应扇区的对应位的奇偶值来推断该位是 1 还是 0。如果其余位的奇偶值等于存储的奇偶值，则丢失的位是 0，反之为 1。RAID 3 级和 RAID 2 级效果一样好，只是前者在额外磁盘所需数目方面更加节省（它只有一个磁盘的额外开销），所以 RAID 2 级在实际中并未使用。图 10-4d 中表示了这种方法。

RAID 3 级与 RAID 1 级相比有两个好处。一方面，RAID 3 级对多个常规磁盘只需要一个奇偶校验磁盘，而不像 RAID 1 级对每个磁盘都需要一个镜像磁盘，因此减少了存储的额外开销。此外由于使用 N 道数据拆分的 RAID 3 级对一个字节的读写散布在多个磁盘中，所以读写一个块的传输率是使用 N 道数据拆分的 RAID 1 级的 N 倍。另一方面，因为每个磁盘都要参与每个 I/O 请求，所以每秒 RAID 3 级支持的 I/O 数较少。

• RAID 4 级。也称为块交叉的奇偶校验组织结构。它像 RAID 0 级一样使用块级拆分，此外它还在一个独立的磁盘上为其他 N 个磁盘上对应的块保留一个奇偶校验块。图 10-4e 表示了这种方法。如果一个磁盘发生故障，可以使用奇偶校验块和其他磁盘上对应的块来恢复发生

故障的磁盘上的块。

读取一个块只访问一个磁盘，因此允许其他请求在其他磁盘上执行。这样，虽然每个存取操作的数据传输率较低，但可以并行地执行多个读操作，从而产生较高的总的 I/O 率。读取大量数据的操作有很高的传输率，因为所有磁盘可以并行地读。写入大量数据的操作也有很高的传输率，因为数据和奇偶校验位可以并行地写。

小的独立的写操作不能并行地执行。写一个块需要访问存储这个块的磁盘和存储奇偶校验位的磁盘，因为存储奇偶校验位的块需要更新。另外，为了计算新的奇偶校验位，还需要读出存储奇偶校验位的块的旧值和已写入的块的旧值。因此，一个写操作需要四次磁盘访问：两次读操作以读取两个旧块，两次写操作以写入两个新块。

• RAID 5 级。也称为块交叉的分布奇偶校验位组织结构。RAID 5 级在 RAID 4 级的基础上进行了改进，将数据和奇偶校验位都分布到所有的 $N+1$ 个磁盘中，而不是在 N 个磁盘上存储数据，在一个磁盘上存储奇偶校验位。在 RAID 5 级中，所有磁盘都能参与对读请求的服务，而不像 RAID 4 级中存储奇偶校验位的磁盘不参与读操作，因而 RAID 5 级增加了给定一段时间中所处理的请求总数。对每个块，一个磁盘存储奇偶校验位，其余磁盘存储数据。例如，在 5 个磁盘组成的阵列中，第 n 个块的奇偶校验位存储在第（n mod 5）+1 个磁盘中，其余 4 个磁盘的第 n 个块存储了对应这个块的实际数据。图 10-4f 表示了这种方法，其中 P 被分布到所有磁盘上。注意奇偶校验块不能和这个块所对应的数据块存储在同一个磁盘上，如果这样，一个磁盘发生故障会同时导致数据和奇偶校验位的丢失，于是数据将不能恢复。RAID 5 级包容了 RAID 4 级，又因为它在相同的成本下提供了更好的读写性能，所以 RAID 4 级在实际中未被使用。

• RAID 6 级。也称为 P+Q 冗余方案。它和 RAID 5 级非常相似，但它存储了额外的冗余信息，以防止多个磁盘发生故障。它不使用奇偶校验的方法，而是使用类似 Reed-Solomon 码（参见文献注解）的纠错码，如图 10-4g 所示。这种方法与 RAID 5 级的 1 位奇偶校验不同，它是对每 4 位数据存储 2 位的冗余信息，这样系统可容忍两个磁盘发生故障。

最后要注意的是，这里描述的基本 RAID 策略现在已经有了各种各样的改进方案。因此在不同 RAID 级别的精确定义上有时会存在混淆。

10.3.4　选择正确的 RAID 级别

如果磁盘发生故障，重建这个磁盘上数据的时间可能很长，并且根据使用的 RAID 级别的不同而不同。RAID 1 级的数据重建是最简单的，因为数据可以从另一个磁盘中拷贝得到。对其他级别，需要访问阵列中其他的所有磁盘来重建发生故障的磁盘上的数据。如果需要连续不断地提供数据，如高性能数据库或交互式数据库，那么 RAID 系统的重建性能将是一个重要因素。此外，重建性能还影响平均故障时间。

因为 RAID 2 级和 RAID 4 级被 RAID 3 级和 RAID 5 级所包容，RAID 级别的选择只需在剩下的级别中进行。RAID 0 级用于可以容忍数据丢失的高性能应用中。RAID 1 级广泛用于像数据库系统中日志文件的存储这样的应用中，因为它提供了最好的写性能。因为 RAID 1 级开销大，所以人们经常选用 RAID 3 级和 RAID 5 级来存储大量数据。RAID 3 级和 RAID 5 级之间的区别是在数据传输率和总的 I/O 率上它们各有所长。如果需要高数据传输率，RAID 3 级更合适；如果更注重随机读，则 RAID 5 级更合适，且大多数数据库系统都属于这种情况。现在很多 RAID 的实现都不支持 RAID 6 级，但 RAID 6 级提供比 RAID 5 级更高的可靠性。

RAID 系统的设计者还需要作出很多其他决定。例如，一个阵列中应该有多少磁盘？每个

奇偶校验位应保护几位数据？如果阵列中有更多的磁盘，数据传输率就更高，但是系统就更昂贵；如果一位奇偶校验位保护更多位的数据，存储奇偶校验位的空间开销就会更低，但这也会增加在第一个发生故障的磁盘被修复前第二个磁盘也发生故障的机会，从而导致数据丢失。

10.3.5　扩展

RAID 的概念可以推广到其他存储设备，如磁带阵列，甚至无线系统中的数据广播上。当应用于磁带阵列时，RAID 结构可以在磁带阵列中的一盘磁盘被破坏时恢复数据。当应用于数据广播时，数据块被分成更小的单元并和一个校验单元一起广播出去，如果其中一个单元由于某种原因而没有被接收到，那么该单元就可以利用其他单元来重建。

10.4　第三级存储

在大型数据库系统中，一些数据可能要存储在第三级存储中。两种最常用的第三级存储介质是光盘和磁带。

10.4.1　光盘

在发布软件、多媒体数据（如声音和图像）和其他电子出版物方面，CD-ROM 已经成为一种流行的介质。CD-ROM 的优点是光盘可以装入驱动器或者从驱动器中移出，并且具有较高的存储容量（一张光盘大约为 500MB，是一张软盘容量的几百倍）。此外，批量生产光盘十分便宜，并且光盘驱动器本身也比磁盘驱动器便宜。

CD-ROM 驱动器与磁盘驱动器相比其具有较长的寻道时间（通常为 250 毫秒）、较低的旋转速度（大约每分钟 400 转），从而导致较长的旋转等待时间以及较低的数据传输率（大约每秒 150KB）。旋转速度最初遵循音频 CD 的标准，但是现在出现了旋转速度是标准速度若干倍的驱动器，它们能提供更高的数据传输率（当前已能达到标准速率的 32 倍）。如今数字视频光盘（DVD）已成为新的光盘格式标准，这种光盘可存储 4.7GB～17GB 的数据。

“一次写，多次读”（WORM）光盘可以被同一台驱动器读写。与磁盘不同的是它只能写一次，并且它上面的数据不能被擦掉，但是它可以被任意读取多次。WORM 光盘广泛用于数据的归档存储，因为它具有较高的容量（每张盘大约为 500MB），具有比磁盘更长的寿命，并且和磁带一样可以从设备中取出。此外，因为它不能被重写，所以可以提供难以篡改的审计踪迹。WORM 自动光盘机是一种可以存放大量 WORM 光盘的设备，这种设备可以根据命令自动将光盘放入几个 WORM 驱动器中的其中一个。

10.4.2　磁带

磁带作为辅助存储介质已经有很长的历史。尽管磁带相对而言更持久些，并且能够存储大量的数据，但是它与磁盘和光盘相比速度较慢。更重要的是，磁带只能进行顺序存取。因此，磁带不能够提供随机存取以满足大多数辅助存储的需求。磁带主要用于备份、存储不经常使用的数据，以及允当将数据从一个系统转到另一个系统的脱机介质。

磁带被绕在一个轴上，通过卷绕或反卷来经过读写头。移动到磁带上的正确位置需要几分钟，而不是几毫秒。然而一旦定位完成，磁带设备能够以接近磁盘设备的密度和速度来写数据。磁带的容量根据磁带的长度、宽度和读写头所能读写的密度的不同而不同。磁带驱动器通常根据磁带的宽度来命名，因此有 8 毫米、1/4 英寸和 1/2 英寸（也称为 9 道）的磁带驱动器。

8毫米磁带驱动器因为所使用的技术而具有最高的密度，现在可以在106米长的磁带上存储5GB的数据。磁带设备是可靠的，但是磁带能可靠地读或写的次数却是有限制的。

磁带也用于存储量大的数据，如影像或图像数据，它们不需要快速地存取，且因为容量太大而使磁盘存储过于昂贵。自动磁带机存放大量的磁带，并且具备一些用于安装磁带的驱动器。该设备用于存储大量数据，甚至可达几TB（10^{12}字节），而存取时间则在秒和分钟这个数量级。需要如此巨大的数据存储的应用包括通过遥感卫星搜集数据的图像处理系统。

10.5　存储访问

一个数据库被映射到多个不同的文件，这些文件由提供支持的操作系统来管理。这些文件永久地存于磁盘上，它们的备份在磁带上。每个文件被分成定长的存储单元，称为块。块是存储分配和数据传输的基本单位。我们将在10.6节中讨论在文件中逻辑地组织数据的不同方法。

一个块可能包括很多数据项。块所包含的数据项的精确集合是由使用的物理数据组织形式决定的（见10.6节）。在这里假定没有数据项跨越两个或两个以上的块。这个假定对多数数据处理应用都是现实的，例如我们所举的银行的例子。

数据库系统的一个主要目标就是减少磁盘和存储器之间传输的块的数目。减少磁盘访问次数的一种方法是在主存储器中保留尽可能多的块。这样做的目的是增大访问的块已经在主存储器中的机会，这样就不再需要访问磁盘。

因为在主存储器中保留所有的块是不可能的，因此需要管理主存储器中用于存储块的可用空间的分配。缓冲区是主存储器中用于存储磁盘块的拷贝的部分。每个块总有一个拷贝存放在磁盘上，但是在磁盘上的拷贝可能比在缓冲区中的拷贝旧。负责缓冲区空间分配的子系统称为缓冲区管理器。

10.5.1　缓冲区管理器

当数据库系统中的程序需要磁盘上的块时，它向缓冲区管理器提出请求（或称为调用缓冲区管理器）。如果这个块已经在缓冲区中，这个块在主存储器中的地址就被传给请求者。如果这个块不在缓冲区中，缓冲区管理器首先在缓冲区中为这个块分配空间，如果需要的话，还会把其他块移出主存储器，为这个新块腾出空间。要从磁盘移出的块仅当它在最近一次写回磁盘时被修改过才被写回磁盘，然后缓冲区管理器把这个块从磁盘读入缓冲区，并将这个块在主存储器中的地址传给请求者。缓冲区管理器的内部动作对发出磁盘块请求的程序是透明的。

如果你熟悉操作系统，你会发现缓冲区管理器几乎和大多数操作系统中的虚拟内存管理器是一样的。它们的一处区别是前者数据库的大小会比机器硬件地址空间大得多，因此存储器地址不足以对所有磁盘块进行寻址。此外，为了更好地为数据库系统服务，缓冲区管理器必须使用比典型的虚拟内存管理策略更加复杂的技术：

- 替换策略。当缓冲区中没有剩余空间时，在新块读入缓冲区之前，必须把一个块从缓冲区中移出。操作系统通常使用“最近最少使用（LRU）”策略，即最近最少访问的块被写回磁盘，并从缓冲区中移走。这种简单的方法可以加以改进以用于数据库应用。
- 被钉住的块。为了使数据库系统能够从崩溃中恢复（见第15章），限制一个块写回磁盘的时间是十分必要的。不允许写回磁盘的块称为被钉住的块。尽管很多操作系统不提供对被钉住的块的支持，但是这个技术对实现可从崩溃中恢复的数据库系统十分重要。
- 块的强制写出。某些情况下，尽管不需要一个块所占用的缓冲区空间，仍必须把这个块写回磁盘。这样的写操作称为块的强制写出。我们将在第15章看到需要强制写出的原因。

简单地说，主存储器的内容，包括缓冲区的内容，在崩溃时将丢失，而磁盘上的数据一般在崩溃时可得以保留。

接下来将讨论缓冲区的替换问题。

10.5.2　缓冲区替换策略

缓冲区中块替换策略的目标是减少对磁盘的访问。对通用程序而言，精确预言哪个块将被访问是不可能的。因此，操作系统利用过去的块访问来预言未来的块访问。通常假定最近被访问过的块最有可能再次被访问。因此，如果必须替换一个块，则替换最近最少访问的块。这种方法称位 LRU 块替换策略。

在操作系统中，LRU 是一个可被接受的替换策略。然而，数据库系统能够比操作系统更精确地预言未来的访问模式。用户对数据库系统的请求包括若干步，通过查看执行用户请求所需操作的每一步，数据库系统通常可以预先确定哪些块会是需要的。因此，与依赖过去预言未来的操作系统不同，数据库系统至少还有关于短期内未来的信息。

为了说明关于未来块访问的信息如何使我们改进 LRU 策略，考虑如下关系代数表达式：

$$borrower \bowtie customer$$

处理这个请求所选择的策略由图 10-5 所示的伪码程序给出。（我们将在第 12 章研究其他策略。）

```
for each borrower 中的元组 b do
    for each customer 中的元组 c do
        if b[customer-name] = c[customer-name]
        then begin
                使 x 成为下列式子确定的元组：
                x[customer-name] := b[customer-name]
                x[loan-number] := b[loan-number]
                x[customer-street] := c[customer-street]
                x[customer-city] := c[customer-city]
                将元组 x 包含进 borrower ⋈ customer 的结果中
             end
    end
end
```

图 10-5　计算连接的过程

假设例子中的两个关系存储在不同的文件中，那么一旦 *borrower* 中的一个元组被处理过，这个元组就不再需要了。因此一旦处理完 *borrower* 元组构成的一个完整的块，这个块就不必再存于主存储器中，尽管它刚刚被使用过。当块中最后一个元组被处理完毕，就应该命令缓冲区管理器释放被这个 *borrower* 块所占用的空间。这个缓冲区管理策略称为*立即丢弃策略*。

现在考虑包含 *customer* 元组的块。对 *borrower* 关系中的每个元组需要查看一次 *customer* 元组的块。当 *customer* 块处理完毕后，我们知道这个块要到所有其他 *customer* 块处理完才会被再次访问。因此，最近使用的 *customer* 块也是再次访问的最后一块，最近最少使用的 *customer* 块是接着要访问的块。这个假设正好与构成 LRU 策略基础的假设相反。实际上，块替换的最优策略是“最近最常使用策略（*MRU*）。”如果必须从缓冲区中移出一个 *customer* 块，MRU 策略将选择最近最常使用的块。

在这个例子中，要使 MRU 策略正确地工作，系统必须把当前正在处理的 *customer* 块钉住。在块中最后一个 *customer* 元组处理完毕后，这个块就不再被钉住，成为最近最常使用的

块。

除了利用系统关于被处理的请求的知识，缓冲区管理器也可以使用有关请求将访问某个特定关系的可能性的统计信息。数据字典是数据库中最常访问的部分之一，因此，缓冲区管理器应尽力不把数据字典从主存储器中移出，除非有其他因素决定了非这样做不可。第 11 章将讨论文件的索引。因为对文件索引的访问比文件本身更频繁，所以不是迫不得已，缓冲区管理器一般不把索引块从主存储器中移出。

理想的数据库块替换策略需要正在执行的数据库操作的有关知识。没有哪个策略能够很好地处理所有可能的情况。实际上，绝大多数数据库系统都使用 LRU 策略，尽管这种策略有其缺点。习题将探究其他可选择的策略。

除了块被再次访问的时间以外，缓冲区管理器使用的块替换策略还被其他因素所影响。如果系统并发地处理多个用户的请求，并发控制子系统（第 14 章）将延迟某些请求，以保证数据库的一致性。如果缓冲区管理器从并发控制子系统那里获得了哪些请求被延迟的信息，它就利用这些信息来改变它的块替换策略。具体地说，活动（非延迟）的请求所需要的块可以因为被延迟的请求所需要的块被换出而在主存储器中得以保留。

崩溃恢复子系统（第 15 章）对块替换施加了严格的约束。如果一个块被修改了，缓冲区管理器不允许将这个块的新内容写回磁盘，因为这将破坏块的旧内容。而块管理器在写回块之前，必须从崩溃恢复子系统处获得许可。崩溃恢复子系统在允许缓冲区管理器写回需要的块之前，可能要求将某些其他块强制写出。第 15 章将精确定义缓冲区管理器与崩溃恢复子系统之间的相互作用。

10.6 文件组织

文件在逻辑上是记录的序列，这些记录被映射到磁盘块上。文件由操作系统作为一种基本结构提供，所以我们假定作为基础的文件系统是存在的。我们考虑用文件表示逻辑数据模型的不同方式。

尽管磁盘的物理特性和操作系统决定了一个块具有固定的大小，但是记录的大小可以不同。在关系数据库中，不同关系中的元组通常有不同的大小。

把数据库映射到文件的一种方法是使用多个文件，在所有文件中只存储同样长度的记录。另一种选择是构造自己的文件，使之能够容纳不同长度的记录。定长记录文件比变长记录文件容易实现，很多用于定长记录文件的技术可以应用到变长的情况。因此，我们首先考虑定长记录文件。

10.6.1 定长记录

让我们考虑由银行数据库的 *account* 记录组成的一个文件。文件中的各个记录定义如下：

```
type deposit = record
                 branch-name : char (22);
                 account-number : char (10);
                 balance : real;
               end
```

如果假设每个字符占 1 个字节，一个实数占 8 个字节，那么一个 *account* 记录为 40 字节

长。一种简单的方法是用头 40 个字节存储第一个记录，接着的 40 个字节存储第二个记录，依次类推（图 10-6）。然而这种简单的方法有两个问题：

1）从这个结构中删除一条记录十分困难。被删除记录所占据的空间必须由文件中的其他记录来填充，或者我们必须用一种方法标记被删除的记录使得它可以被忽略。

2）除非块的大小恰好是 40 的倍数（一般是不太可能的），否则一些记录会跨过块的边界，即一条记录的一部分存储在这个块中，而另一部分存储在另一个块中。于是，读写这样一条记录需要两次块访问。

记录 0	Perryridge	A-102	400
记录 1	Round Hill	A-305	350
记录 2	Mianus	A-215	700
记录 3	Downtown	A-101	500
记录 4	Redwood	A-222	700
记录 5	Perryridge	A-201	900
记录 6	Brighton	A-217	750
记录 7	Downtown	A-110	600
记录 8	Perryridge	A-218	700

图 10-6　包含 *account* 记录的文件

当一条记录被删除时，应该把紧跟其后的记录移动到被删记录先前占据的空间，依次类推，直到被删记录后面的每一条记录都向前做了移动（图 10-7）。这种方法需要移动大量的记录，因而将文件的最后一条记录移到被删记录所占据的空间可能更加简单，如图 10-8 所示。

记录 0	Perryridge	A-102	400
记录 1	Round Hill	A-305	350
记录 3	Downtown	A-101	500
记录 4	Redwood	A-222	700
记录 5	Perryridge	A-201	900
记录 6	Brighton	A-217	750
记录 7	Downtown	A-110	600
记录 8	Perryridge	A-218	700

图 10-7　图 10-6 中的文件删除了第二条记录并且移动所有记录

记录 0	Perryridge	A-102	400
记录 1	Round Hill	A-305	350
记录 8	Perryridge	A-218	700
记录 3	Downtown	A-101	500
记录 4	Redwood	A-222	700
记录 5	Perryridge	A-201	900
记录 6	Brighton	A-217	750
记录 7	Downtown	A-110	600

图 10-8　图 10-6 中的文件删除了第一条记录并且移动最后一条记录

移动记录以占据被删记录所释放的空间的做法并不理想，因为这样做需要额外的块访问操作。由于插入操作通常比删除操作更频繁，所以让被删记录占据的空间空着，一直等到随后进行的插入操作才重新使用这个空间的做法是可以接受的。仅在被删记录上做一个标记是不够的，因为当插入操作执行时，找到这个可用空间十分困难。因此我们需要引入额外的结构。

在文件的开始处，分配一定数量的字节作为文件头，文件头包含有关文件的各种信息。到目前为止，需要在文件头中存储的内容只有被删除的第一个记录的地址。用这第一个记录来存储第二个可用记录的地址，依次类推。可以直观地把这些存储的地址看作指针，因为它们指向一个记录的位置。于是，被删除的记录形成了一条链表，通常称为空闲列表。图 10-9 给出的是图 10-6 中的文件在删除第 1、4 和 6 条记录后的情况。

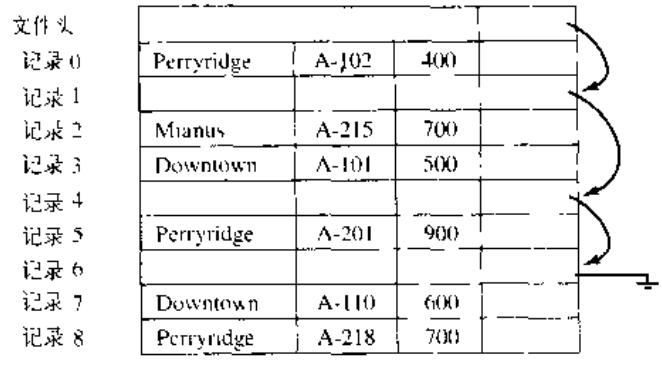


图 10-9　删除了第 1、4 和 6 条记录的图 10-6 中的文件

在插入一条新记录时，找到文件头所指向的可用记录，并改变文件头的指针以指向下一个可用记录。如果没有可用的空间，就把这条新记录加在文件末尾。

指针的使用要求编程要格外小心。如果移动或删除一条记录，而另一个记录里包含了指向这个记录的指针，那么这个指针就变得不正确了，因为它不再指向它所应该指向的记录。这样

的指针称为虚悬指针，它们实际上指向的是无用的东西。为了避免虚悬指针问题，必须避免移动或删除被其他记录所指向的记录，这样的记录称为被钉住的记录。

对定长记录文件的插入和删除是容易实现的，因为被删记录留出的可用空间恰好是插入记录所需要的空间。如果允许文件中包含不同长度的记录，这样的匹配将不再成立。被插入的记录可能放不进被删除记录所释放的空间，或者只能占用这个空间的一部分。

10.6.2　变长记录

变长记录出现在数据库系统中的下述情况中：

- 多种记录类型在一个文件中存储。
- 记录类型允许一个或多个字段是变长的。
- 记录类型允许可重复的字段。

实现变长记录有多种不同技术。为了便于说明，我们用一个例子来演示不同的实现技术。考虑存储在图 10-6 所示文件中存款数据的另一种不同表示，其中对于各个分支机构的名称及其所有帐户的信息，我们用一个变长记录来表示。记录的格式如下：

```
type account-list = record
                      branch-name : char (22);
                      account-info : array [1 .. ∞] of
                                       record
                                         account-number : char (10);
                                         balance : real;
                                       end
                    end
```

我们把 *account-info* 定义成一个有任意多个元素的数组，因此，一个记录有多大是没有限制的（当然，可以大到整个磁盘）。

1. 字节流表示

实现变长记录的一个简单方法是在每个记录的末尾附加一个特殊的记录终止符号（⊥），这样可以把每个记录作为一个连续的字节流来存储。图 10-10 给出了一种组织结构，这一结构通过使用变长记录来表示图 10-6 中的定长记录文件。另一种字节流表示方法是在每个记录的开始处存储记录的长度，而不再使用记录终止符号。

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

图 10-10　变长记录的字节流表示

字节流表示方法有以下不足：

- 重新使用被删除记录曾经占用的空间十分困难。尽管存在管理插入和删除的技术，但是这些技术将导致大量被浪费的磁盘存储碎片。
- 通常分配给记录的空间不能随记录长度的增长而增长。如果一个变长记录变长了，该记录就必须被移动，而如果这个记录是被钉住的，则移动的代价将很高。

因此，通常不用前面所描述的基本字节流表示方法实现变长记录，字节流表示方法的一种改进形式普遍用于块内部的记录组织，称为分槽的页结构。

分槽的页结构如图 10-11 所示，每个块的开始处有一个块头，其中包含以下信息：

1）块中记录条目的个数。

2）块中空闲空间的末尾地址。

3）一个由包含记录位置和大小的记录条目组成的数组。

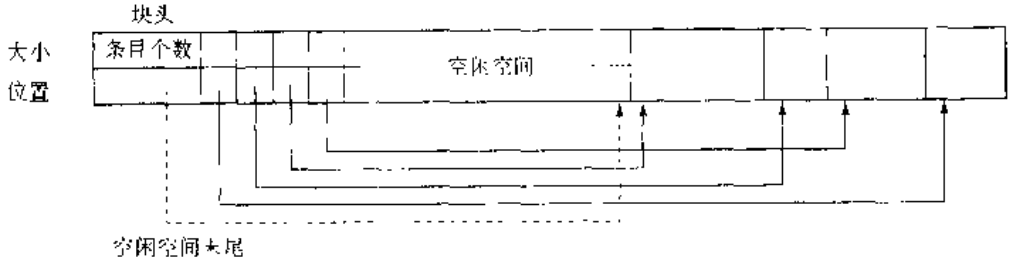


图 10-11　分槽的页结构

实际记录从块的尾部开始连续分配，块中空闲空间是连续的，处在块头数组的最后一个条目和第一条记录之间。如果插入一条记录，则在空闲空间的尾部给这条记录分配空间，并且包含这条记录大小和位置的条目被加到块头中。

如果一条记录被删除，它所占用的空间被释放，并且它的条目被置成删除状态（例如这条记录的大小被置为 -1）。此外，块中在删除记录之前的记录将被移动，使得由删除而产生的空闲空间被重新使用，并且所有空闲空间再一次处于块头数组的最后一个条目和第一条记录之间。块头中的空闲空间末尾指针也做适当修改。当块中存在空闲空间时，使用类似的技术可以使记录增长或缩短。移动记录的代价并不高，因为块的大小是有限制的，通常值为 4KB。

分槽的页结构不要求有直接指向记录的指针，但必须有指针指向块头中包含记录实际位置的条目。在支持指向记录的间接指针的同时，这种间接层次也防止在块的内部出现碎片空间而不得不移动记录。

2. *定长的表示方法*

另一种在文件系统中有效实现变长记录的方法是使用一个或多个定长记录来代表一个变长记录。

使用定长记录实现变长记录文件的技术有两种：

1）保留空间。如果设置一个永远不会被超过的最大记录长度，我们就可以使用长度为这个最大记录长度的定长记录。（对那些比最大空间短的记录）未使用的空间用特殊的空值或记录终结符号来填充。

2）指针。变长记录用一系列通过指针链接起来的定长记录来表示。

如果对所举的帐户例子采用保留空间的方法，我们需要选择一个最大记录长度。图 10-12

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

图 10-12　使用保留空间方法的图 10-6 中的文件

展示了当允许每个分支机构最多有三个帐户时，图 10-6 中的文件是如何表示的。这个文件中

的记录属于 *account-list* 类型，但其数组元素个数为 3。那些少于三个帐户的分支机构（例如，Round Hill）就会有包含空值域的记录。在图 10-12 中，使用符号⊥来表示空值域。实践中通常使用一个不代表任何实际数据的特殊值（例如，一个负的“*account number*”或一个以“*”开头的“*name*”）。

这种保留空间的方法在大多数记录都接近最大长度时非常有用。如果情况不是这样，就会浪费大量空间。在银行例子中，一些分支机构的帐户可能比其他分支机构要多得多，这种情况促使我们考虑使用指针的方法。为了用指针方法表示文件，我们增加一个指针域，就像在图 10-9 中所做的那样。产生的结构如图 10-13 所示。

实际上，图 10-9 和图 10-13 中的文件结构是一样的，只不过在图 10-9 中，我们仅仅用指针把被删除的记录链接在一起，而在图 10-13 中，我们把所有属于同一分支机构的记录链接在一起。

图 10-13 所示结构的一个缺点是除了链表中第一个记录以外的所有记录都浪费了空间。第一个记录需要有 *branch-name* 的值，但是后续的记录不需要。然而，我们需要在所有的记录中都包括 *branch-name* 域，否则记录就不是定长的了。这样浪费的空间是巨大的，因为在实际中每个分支机构都有大量的帐户。为解决这个问题，在文件中使用两种类型的块：

1）锚块。包含链表中第一个记录的块。

2）溢出块。包含链表中第一个记录以外的其他记录的块。

这样，一个块中的所有记录都具有相同的长度，尽管一个文件中所有的记录并不都具有相同的长度。图 10-14 表示了这种文件结构。

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

图 10-13　使用指针方法的图 10-6 中的文件

锚块

Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
Brighton	A-217	750	

溢出块

A-201	900	
A-218	700	
A-110	600	

图 10-14　锚块和溢出块结构

10.7　文件中记录的组织

迄今为止，我们研究了如何在一个文件结构中来表示记录。一个关系的实例是记录的集合。给定记录的集合，下一个问题就是如何在文件中组织它们。下面是在文件中组织记录的几种方法：

- 堆文件组织。在这种组织方式中，一条记录可以放在文件中的任何地方，只要那个地方有空间存放这条记录。记录是没有顺序的。通常一个关系是一个单独的文件。
- 顺序文件组织。在这种组织方式中，记录根据搜索码的值顺序存储。这种组织方式的实现在下节中描述。
- 散列文件组织。在这种组织方式中，各个记录的同一属性需要计算一个散列函数。散列函数的结果确定了记录应存储到文件的哪个块中。这种组织方式在第 11 章中描述，该结构与同一章中所描述的索引结构密切相关。

• 聚集文件组织。在这种组织方式中，不同关系的记录可以存储在同一个文件中。由于不同关系中的相关记录存储在相同的块中，于是一个 I/O 操作可以从所有关系中取到相关记录。这种组织方式在 10.7.2 节中描述。

10.7.1　顺序文件组织

顺序文件是为了高效处理按搜索码排序的记录而设计的。为了快速地按搜索码获取记录，我们通过指针把记录链接起来。每个记录的指针指向在搜索码顺序上的下一个记录。此外，为了减少顺序文件处理中的块访问的次数，我们在物理上按搜索码顺序存储记录，或尽可能的按照搜索码顺序。

图 10-15 表示了银行例子中由 *account* 记录组成的顺序文件。在该例中，记录使用 *branch-name* 作为搜索码并按其顺序存储。

Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	700	
Round Hill	A-305	350	

图 10-15　*account* 记录组成的顺序文件

顺序文件组织形式允许记录按排序的顺序读取，这对显示和特定的查询处理算法非常有用。这些内容将在第 12 章中研究。

然而，当记录被插入和删除后，维护物理上的顺序也是十分困难的，因为一个单一的插入或删除将导致很多记录的移动，而移动记录的代价很高。我们可以像前面那样，使用指针链表来管理删除。对插入，应用如下规则：

1）在文件中定位按搜索码顺序处于被插入记录之前的那条记录。

2）如果这条记录所在块中有一个空记录（即删除后留下来的空间），则就在那里插入新记录。否则，将新记录插入到一个溢出块中。不管哪种情况，都要调整按搜索码顺序把记录链接在一起的指针。

图 10-16 表示在图 10-15 所示文件中插入记录（North Town，A-888，800）后的情况。图 10-16 中的结构允许快速插入新记录，但却强迫顺序文件处理应用按不与记录的物理顺序相匹配的顺序来处理记录。

Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	700	
Round Hill	A-305	350	
North Town	A-888	800	

图 10-16　执行插入后的顺序文件

如果需要存储在溢出块中的记录相当少，这种方法将十分有效。然而，搜索码顺序和物理顺序之间的一致性最终将完全丧失，在这种情况下，顺序处理将明显变得效率低下。此时，文件应该重组，使得它再次能在物理上顺序存放。这种重组的代价是很高的，并且必须在系统负载很低的时候执行。需要重组的频率依赖于新记录插入的频率。在插入很少发生的极端情况下，使文件在物理上总保持排序的状态是可能的。此时图 10-15 中的指针域是不需要的。

10.7.2　聚集文件组织

很多关系数据库系统将各个关系存储在一个个独立的文件中，所以它们可以利用作为操作系统一部分的文件系统的所有好处。通常，关系的元组表示成定长记录，因此，关系可以映射为一个简单的文件结构。关系数据库系统的这种简单实现非常适合于为个人计算机设计的数据

库系统。在这种系统中，数据库的规模很小，复杂的文件结构不会带来什么好处。此外，在一些个人计算机中，必须使数据库系统目标代码量很小，简单的文件结构可以减少实现这个系统的代码量。

这种实现关系数据库的简单方法在数据库的规模增大时变得非常不合适。我们已经看到，在块间最佳地分配记录和组织块本身可以获得性能优势。因此，一个更复杂的文件结构将更有效是很明显的，即使保留在一个独立的文件中存储一个关系的策略。

然而，很多大型数据库系统在文件管理方面并不直接依赖于下层的操作系统。而是让操作系统分配给数据库系统一个大的操作系统文件，所有关系都存储在这个文件中，而这个文件的管理由数据库系统进行。为帮助理解在一个文件中存储多个关系的好处，考虑对银行数据库的如下 SQL 查询：

select *account-number*, *customer-name*, *customer-street*, *customer-city*
from *depositor*, *customer*
where *depositor*.*customer-name* = *customer*.*customer-name*

这个查询计算关系 *depositor* 和关系 *customer* 的连接。因此，对 *depositor* 的每个元组，系统必须找到具有相同 *customer-name* 的 *customer* 元组。理想状态下，这些记录通过索引来定位，这将在第 11 章中讨论。然而，不管这些记录如何定位，它们都需要从磁盘传输到主存储器中。最坏情况下，每个记录处在不同的块中，迫使我们为查询所需的每个记录执行一次读块操作。

作为一个具体的例子，考虑图 10-17 和图 10-18 中的关系 *depositor* 和 *customer*。图 10-19

customer-name	*account-number*
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

图 10-17　*depositor* 关系

customer-name	*customer-street*	*customer-city*
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

图 10-18　*customer* 关系

给出了一个为高效执行 *depositor* ⋈ *customer* 查询而设计的文件结构。对每个 *customer-name*，*depositor* 元组存储在对应 *customer-name* 的 *customer* 元组附近。这种结构将两个关系的元组混合在一起，但是允许对连接的高效处理。当读取关系 *customer* 的一个元组时，包含这个元组的整个块从磁盘拷贝到主存储器中。因为相应的 *depositor* 元组存储在靠近 *customer* 元组的磁盘上，所以包含 *customer* 元组的块也包含了处理查询所需的关系 *depositor* 的元组。如果一个顾客有太多的帐户，以至于 *depositor* 记录不能存储在一个块中，则其余的记录出现在临近的块中。这种文件结构称为聚集，它允许通过一次读块操作来读取更多需要的记录，由此可以更高效地处理这个特定的查询。

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

图 10-19　聚集文件结构

我们对聚集的使用加速了对特定连接（*depositor* ⋈ *customer*）的处理，但是它导致其他查询类型的处理变慢。例如：

select *
from *customer*

与在单一文件中存储单一关系的策略相比，需要访问更多的块。若干 *customer* 记录不是出现在一个块中，而是每个记录位于不同的块中。实际上，如果没有一些附加结构，想要很容易地找到所有 *customer* 记录是不可能的。为了在图 10-19 的结构中找到关系 *customer* 的所有元组，需要用指针把这个关系的所有记录链接起来，如图 10-20 所示。

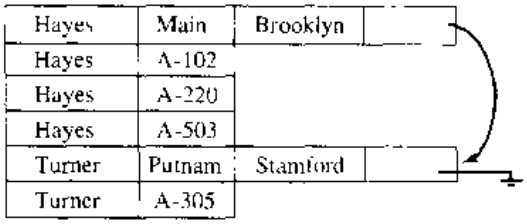


图 10-20　带指针链的聚集文件的结构

何时使用聚集的决定依赖于什么样的查询类型是数据库设计者所认为的最频繁的查询类型，聚集的谨慎使用可以导致查询中明显的性能提高。

10.8　数据字典的存储

到目前为止，我们只考虑了关系本身的表示。一个关系数据库系统需要维护有关关系的数据，如关系模式等。这类信息称为**数据字典**，或**系统目录**。系统必须存储的信息类型有：

- 关系的名字。
- 每个关系的属性的名字。
- 属性的域和长度。
- 在数据库上定义的视图的名字和这些视图的定义。
- 完整性约束（例如码的约束）。

此外，很多系统为系统用户保存了下列数据：

- 授权用户的名字。
- 关于用户的帐户信息。

此外，有关关系的统计数据和描述数据可以如下方式存在：

- 每个关系中元组的总数。
- 每个关系所使用的存储方法（例如，聚集或非聚集）。

第 11 章研究索引，在那里，我们将看到有必要存储关于每个关系的每个索引信息：

- 索引的名字。
- 被索引的关系的名字。
- 在其上定义索引的属性。
- 索引的类型。

实际上，所有这些信息组成了一个微型数据库。一些数据库系统使用专门的数据结构和代码来存储这些信息。通常人们更倾向于在数据库中存储关于数据库本身的数据。通过使用数据库存储系统数据简化了系统的总体结构，并且能够使用数据库的全部能力来对系统数据进行快速访问。

如何使用关系来表示系统数据，这一选择必须由系统设计者来决定。下面是一种可能的表示：

System-catalog-schema = (*relation-name*, *number-of-attributes*)

Attribute-schema = (*attribute-name*, *relation-name*, *domain-type*, *position*, *length*)

User-schema = (*user-name*, *encrypted-password*, *group*)

Index-schema = (*index-name*, *relation-name*, *index-type*, *index-attributes*)

View-schema = (*view-name*, *definition*)

10.9　面向对象数据库的存储结构

前面描述的文件组织技术（如堆、顺序、散列和聚集文件的组织）也可以用来在一个面向对象的数据库中存储对象。然而，为了支持面向对象数据库的特性，如以集合为值的字段和持久化指针，我们需要附加一些特性。

10.9.1　对象到文件的映射

对象到文件的映射与关系数据库中元组到文件的映射有很多相似之处。在数据表示的最低层，元组和对象的数据部分都只是字节的序列。因此可以用前几节中描述的文件结构来存储对象数据，要做的只是一些我们接下来将要提到的改动。

面向对象数据库中的对象可能缺少关系数据库中元组之间的一致性。例如，面向对象数据库中记录的字段可能是集合，而不像在关系数据库中，要求数据（至少）是第一范式。此外，对象可能非常大，这种对象的管理和关系数据库中记录的管理是不同的。

使用链表之类的数据结构可以实现具有少量元素的集合字段。具有大量元素的集合字段可以用B树或者数据库中一个单独关系来实现。集合字段也可以通过规范化在存储层消除。存储系统为数据库系统的更高层提供一个集合字段的视图，尽管集合字段实际上被规范化成一个新的关系。

一些应用包含了极大但却不易分解成更小成分的大对象，这样的大对象可以分别存储在单独的文件中。我们将在10.9.5节中讨论这一想法。

10.9.2　对象标识的实现

因为对象由对象标识（OID）确定，所以对象存储系统需要一种通过给定的OID来定位一个对象的机制。如果OID是逻辑的（即它没有指明对象的位置），则存储系统必须维护一个将OID映射到对象实际位置的索引。如果OID是物理的（即它的编码中包含对象的位置），则对象可以被直接找到。物理的OID一般包含以下三部分：

1）卷或文件标识。

2）卷或文件中页的标识。

3）页中的偏移量。

另外，物理的OID中可能还包含一个唯一标识，唯一标识是一个整数，用于区分这个OID和以前曾经恰好存储在相同位置并被删除或移动到别处的其他对象的标识。唯一标识也和对象存储在一起，并且OID中的标识和相应对象中的标识应该匹配。如果物理OID中的唯一标识和OID所指对象中的唯一标识不匹配，系统就认为这个指针是一个虚悬指针，并且发出一个错误信号。图10-21举例说明了这种机制。

这种指针错误在与已删除旧对象相对应的物理OID被意外使用时会发生。如果对象占用的空间被重新分配，这个位置上将有一个新的对象，并且可能通过旧对象的标识错误地访问这个新对象。如果虚悬指针的使用没有被检测到，将导致存储在同样位置上的新对象被破坏。唯一标识有助于检测这类错误，因为旧的物理OID中的唯一标识和新对象中的唯一标识不匹配。

如果一个对象不得不移到一个新的页，原因可能是因为这个对象的大小增加，而旧页中已没有额外的空间，那么物理OID将指向不再包含这个对象的旧页。我们采取的办法不是改变这个对象的OID（这将要改变指向这个对象的每个对象），而是在旧的位置留下一个转向地址。当数据库试图定位这个对象时，它将找到转向地址而不是这个对象，然后它使用转向地址找到

该对象。

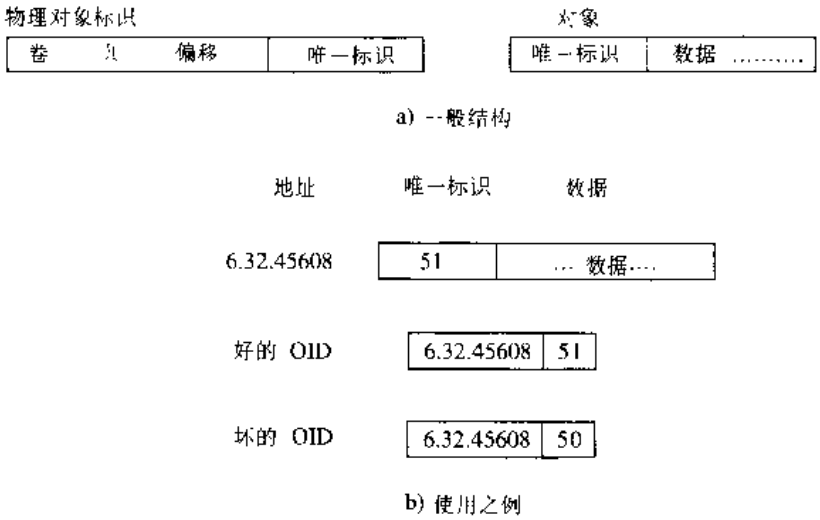


图 10-21　OID 中的唯一标识

10.9.3　持久化指针的管理

我们在一种支持持久性的编程语言中使用 OID 来实现持久化指针。在一些实现中，持久化指针是物理 OID；而在另一些实现中，持久化指针是逻辑 OID。持久化指针和内存指针的一个重要区别是指针的大小。内存指针所需大小只要可以在整个虚拟存储器中寻址即可。在如今的计算机中，内存指针为 4 个字节长，这足以对 4GB 的内存进行寻址。持久化指针需要寻址数据库中的所有数据。因为数据库系统通常大于 4GB，持久化指针一般至少为 8 字节长。许多面向对象的数据库在持久化指针中也提供唯一标识，以捕获虚悬访问。这个特性进一步增加了持久化指针的大小，因此，持久化指针比内存指针要长得多。

1. 指针混写

根据给定标识寻找对象的动作称为解引用。给定一个内存指针（如 C++ 中），查找对象只是对内存的一个引用。给定一个持久化指针，解引用一个对象需要额外的步骤——必须通过在一个表中查找持久化指针来找到对象在内存中的实际位置。如果对象不在内存中，则必须从磁盘上把它载入。可以使用散列来高效地实现对表的查找，但是与指针解引用相比，即使对象已经在内存中，查找仍然很慢。

指针混写是一种减少定位已经存在于内存中的持久化对象所需代价的方法。它的思想是，当一个持久化指针第一次解引用时，这个对象被定位，并且如果它不在内存中，就将它放入内存，此时开始一个额外的步骤——存储指向这个对象的内存指针以取代持久化指针。下一次当相同的持久化指针被解引用时，对象在内存中的位置可以被直接读出，这样就避免了定位对象的代价。（如果持久化对象为了给其他持久化对象腾出空间而从内存移回磁盘，那么还必须执行一个确保对象仍然在内存中的额外步骤。）相应地，当一个对象被写出时，它所包含的任何被混写的持久化指针需要反混写，即把它们转换回持久化表示。这里所描述的用于指针解引用的指针混写称为软件混写。

如果使用了指针混写，缓冲区的管理将十分复杂，因为一旦对象被放入缓冲区，这个对象的物理位置就不能被改变了。保证物理位置不被改变的一种方法是在缓冲池中钉住包含混写对象的页，使得它们不被替换，直到执行混写的程序结束为止。更复杂的缓冲区管理策略请参看文献注解，例如虚拟存储映射技术，它不需要钉住缓冲页。

2. 硬件混写

使用两种指针类型，即持久化指针和临时指针（内存中的），是十分不便的。程序员必须记住指针类型，而且可能不得不写两遍代码——一遍为持久化指针，一遍为内存指针。如果持久化指针和内存指针具有相同的类型，就会方便得多。

一个合并持久化指针和内存指针的简单方法是把内存指针的长度扩展到与持久化指针同样的大小，并且用标识中的 1 位来区分持久化指针和内存指针。然而，内存指针也将带来和持久化指针相同的存储代价，所以这种策略并不流行。

我们将描述一种称为硬件混写的技术，它利用当前大多数计算机系统中的存储器管理硬件来解决这个问题。

硬件混写与软件混写相比有两个主要优点。第一，它可以在对象中用与内存指针相同大小空间来存储持久化指针（同时在对象以外需要额外的存储）。第二，它以智能高效的方法，透明地在持久化指针和内存指针之间进行转换。这样，处理内存指针的软件不需要任何改变就可以用来处理持久化指针。

硬件混写用如下方式表示磁盘上的对象所包含的持久化指针。一个持久化指针在概念上划分为两部分：数据库中页的标识和页中的偏移量。持久化指针中的页标识实际上是一个小的间接指针，称之为短的页标识。每页（或其他存储单位）有一个转换表，提供从短的页标识到数据库中的完整页标识的映射。系统通过在转换表中查找持久化指针中的短的页标识来得到完整页标识。

最坏情况下，转换表也只是和一页中所有对象能包含的最大指针数一样大。如果页大小为 4096 字节，指针大小为 4 字节，则指针最多有 1024 个。实践中，转换表包含的元素个数远远少于最大数目（如 1024），并且不消耗过多的空间。短的页标识只需要足够的位来标识表中的一个记录；当表的最大长度为 1024 时，短的页标识只需要 10 位。因此，少量的位就足以存储短的页标识。这样，转换表使整个持久化指针可以和内存指针占据相同大小的空间。尽管短的页标识只要几位就够了，但是内存指针中除了表示页中偏移量的位以外，其他所有位都用来做短的页标识。这种体系结构可以使混写容易一些。

图 10-22 给出了持久化指针的表示方法，图中所表示的页里有三个对象，每个包含一个持久化指针。对这些持久化指针中的短的页标识，转换表给出了短的页标识和数据库的完整页标识之间的映射。数据库中页的标识以“卷．文件．偏移”的格式表示。

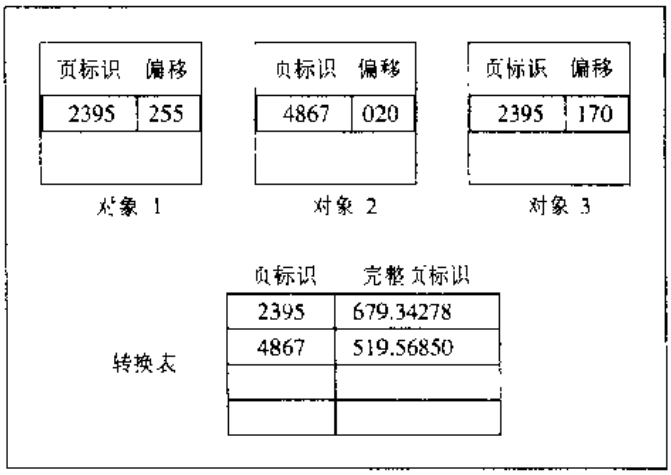


图 10-22　混写之前的页的影象

每个页需要维护一些额外的信息，以保证页中所有持久化指针都能被找到。当页中创建或删除一个对象时，这些信息被更新。定位页中所有持久化指针的需求在后面将变得十分明显。

当内存指针被解引用时，如果操作系统检测到并没有为它所指向的虚拟地址空间中的页分配存储，或者该页有访问保护，则称一个段违例发生了。（很多对硬件混写的描述使用术语缺页而不用段违例。）很多操作系统都提供了段违例发生时指定调用函数的机制，以及在虚拟地址空间中为页分配存储并设置页访问许可的机制。在大多数 Unix 系统中，系统调用 *mmap* 提供了后一种功能，并被用来实现硬件混写。

例如一个指向页 v 的内存指针在还没有为这个页分配存储时第一次被解引用。如前所述，一个段违例将会发生，并且导致数据库系统的一个函数调用。数据库系统首先决定哪个数据库页是分派给虚拟存储页 v 的，设这个数据库页的完整页标识为 P。如果没有数据库中的页分派给 v，就标记一个错误。否则，数据库系统给页 v 分配存储空间，并将数据库的页 P 载入到页 v。

现在开始进行对页 P 的指针混写，如下所述。通过使用存储在页中的额外信息，系统定位页 P 中所有对象所包含的所有持久化指针。考虑每个指针$<p_i, o_i>$，其中 p_i 为短的页标识，o_i 为页中的偏移量。设 P_i 是 p_i 在页 P 的转换表中的完整页标识。

如果还没有分配虚拟存储器中的页给页 P_i，则现在分配虚拟地址空间中的一个空闲页给它。如果页 P_i 被载入，那么它载入时将存放于这个虚拟地址空间。而此时，系统无论是存储器中还是磁盘上，并没有为虚拟地址空间中的这一页分配任何空间，而仅仅为数据库的页保留了一段地址。

假设（在前一步中或更早）分配给 Pi 的虚拟存储器的页为 v_i，则修改指针$<p_i, o_i>$，用 v_i 代替 p_i。最后，在混写 P 中所有持久化指针后，导致段违例的指针解引用就可以继续进行，并发现它所寻找的对象已经载入到内存中。

图 10-23 表示图 10-22 中的页载入内存且该页中的所有指针均被混写后的状态。这里，我们假设数据库中页标识为 679.34278 的页被映射到内存中的第 5001 页，然而标识为 519.56850 的页被映射到第 4867 页（与短的页标识相同）。对象中的所有指针都已经更新为能够反映新的映射，而且可以把它们当作内存指针来使用。

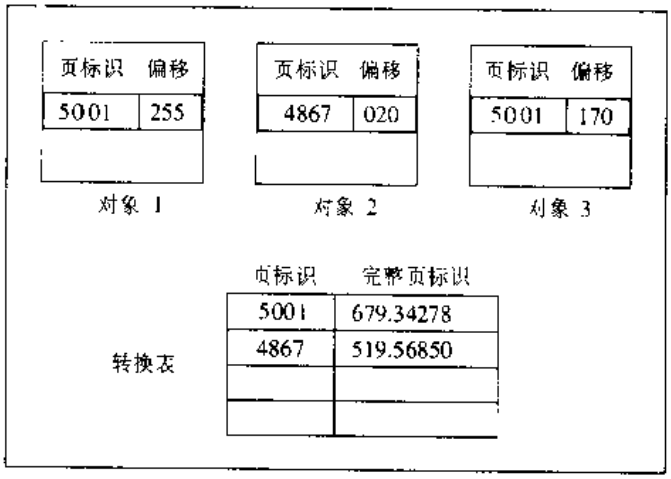


图 10-23　混写之后的页的影象

在对页进行的转换结束时，页中的对象满足一个重要特性：页中对象包含的所有持久化指针都被转换成了内存指针。因此，内存页中的对象只包含了内存指针。使用这些对象的程序不必知道持久化指针的存在。例如，已有的为内存对象书写的库可以不加修改地用于持久化对

象。这确实是一个重要的优点!

注意，如果这里描述的任一个内存页 v_i 在虚拟存储器中还没有被载入，那么，任何一个指向它的内存指针在第一次被解引用时，段违例都将发生。页 P_i 被载入存储器的页 v_i，并且对它进行指针混写，正如对页 P 所进行的那样。

当存储器中的页被写回数据库时，软件混写有一个反混写操作与之相联，即将内存指针转回持久化指针。硬件混写可以避开这一步——当对页完成指针混写后，只要简单地修改页的转换表，使被混写的内存指针的页标识部分可以用于表中查找。例如，如图 10-23 所示，数据库中的页 679.34278（在页中的短标识为 2395）被映射到虚拟存储器中的第 5001 页。这时，不仅对象 1 中的指针从 2395255 修改为 5001255，而且表中的短标识也被修改为 5001。于是，对象 1 中的短标识 5001 和表中又一次互相匹配了。因此，页可以写回磁盘而不需要任何反混写。

对这里描述的基本策略可以进行一些优化。当对页 P 进行了混写后，对 P 中每个持久化指针所指的页 P'，系统试图把 P' 分配到 P' 在页 P 中的短标识所指定的虚拟地址。如果这样的分配能如愿以偿，指针就不需要被修改。例如，短的页标识为 4867 的页 519.56850 被映射到虚拟存储器的页 4867，和短的页标识一样。我们可以看到对象 2 中指向这一页的指针在混写过程中就不需要任何改动。如果每一页在虚拟地址空间中都可以分配到合适的位置，那么就没有指针需要转换，混写的代价也将显著减少。

硬件混写可以在数据库大于虚拟存储器的情况下工作，但是必须满足特定进程存取的所有的页可被进程的虚拟存储器所容纳。如果不能做到这一点，一个已载入虚拟存储器的页将被替换，并且这个替换是很困难的，因为该页中可能有指向对象的内存指针。概念上，硬件混写也可以在页集合这一层（通常称为段）使用，而不是用在单独的页上，只要含有页内偏移量的短的页标识不超过内存指针的长度。

10.9.4　对象的磁盘结构与内存结构

存储在内存中的对象格式与存储在磁盘上数据库中的对象格式可能不同。一个原因可能是使用了软件混写，其中持久化指针和内存指针的结构是不一样的。另一个原因可能是我们希望数据库能被基于不同体系结构的机器、不同的语言以及不同编译器编译的程序所访问。所有这些导致了对象在内存中的表示不一样。

例如，我们来看编程语言如 C++ 中数据结构的定义。对象中的物理结构（如整型的大小和表示）依赖于程序所运行的机器[⊖]。此外，物理结构还依赖于所使用的编译器——像 C++ 那样复杂的语言中，从高层描述到物理结构的翻译可能有不同的选择，每个编译器可以做出自己的选择。

这个问题的解决方法是使数据库中对象的物理表示独立于机器和编译器。当这个对象被载入内存中时，对象可以从磁盘上的表示转换为特定机器、语言和编译器所需要的形式。这种转换可以在对象中指针进行混写的同时透明地进行，而程序员可以不必考虑。

实现这种策略的第一步是定义一种描述对象结构的通用语言——即数据定义语言。现在人们已经提出了很多建议，其中之一是由对象数据库管理组（ODMG）开发的对象定义语言（ODL）。ODL 有定义到 C++ 和 Smalltalk 上的映射，我们可以在一个遵从 ODMG 的数据库中

⊖ 例如，Motorola680x0、IBM360 和 Intel80386/80486/Pentium 体系结构中的整数都是 4 字节的。然而，它们在一个字中整数的各位如何排列上存在差异。在更早一代的个人计算机中，整数为 2 字节长。在较新的工作站体系结构如 DEC Alpha 中，整数可达 8 字节长。

使用 C++ 或 smalltalk 操纵对象。

数据库中每个类的结构定义被（逻辑地）存储在数据库中。将数据库中的对象转换为编程语言所操纵的对象表示（或相反方向的转换），其代码依赖于机器和这种语言的编译器。可以使用存储的对象类的定义来自动生成这些代码。

数据的磁盘表示和内存表示之间的一个让人意想不到的差异来自对象中的隐藏指针。隐藏指针是由编译器产生的并存储在对象中的临时指针。这些指针（间接地）指向用于实现对象的某些方法的表。这些表一般被编译成可执行目标代码，其确切位置依赖于这些可执行目标代码，因此对不同的进程而言，表的位置也可能不同。于是，当对象被一个进程访问时，隐藏指针必须被改变以指向正确的位置。隐藏指针可以在转换数据表示的同时进行初始化。

10.9.5　大对象

对象可能非常大，例如，多媒体对象可能占用几 MB 的空间。异常大的数据项，如视频序列，可以达到几 GB，尽管它们通常划分成多个对象，且每个对象在几 MB 的数量级或更少。大对象通常被称为二进制大对象，因为它们一般包含二进制数据。在关系数据库中，极大的记录字段也称为长字段。

大多数关系数据库限制记录的大小不超过一页，以简化缓冲区和空闲空间的管理。大对象和长字段通常被存储在为存储长字段而保留的特殊文件（或文件集合）中。

管理大对象的一个困难表现在缓冲页的分配上。大对象在被载入内存时，可能需要存储在连续的字节流序列中。这时，如果一个对象大于一页，则必须在缓冲池中分配连续的页来存储它，这使得缓冲区管理变得十分困难。

我们经常通过变更对象的一部分或通过插入、删除对象的部分来修改大对象，而不是通过重写整个对象。如果需要支持插入和删除，我们可以使用 B 树结构（第 11 章中研究）来实现大对象。B 树结构允许我们读整个对象，也允许插入和删除对象的部分。

由于实际原因，我们使用应用程序来操作大对象，而不是在数据库内部完成：

- 文本数据。文本通常被当作字节流，由编辑器和格式化工具来处理。
- 图像数据。图像数据被表示成位图，或者表示为线、矩形以及其他几何对象的集合。尽管一些图像数据通常由数据库本身来管理，但大多数情况下还是使用特殊的应用软件。例如 VLSI 的设计即是后一种情况。
- 音频和视频数据。音频和视频数据通常是数字化的压缩形式，由不同的应用软件产生和显示。数据的修改通常由数据库系统之外的专用编辑软件来执行。

修改这类数据时使用最广泛的方法是出库/入库方法。用户或应用程序出库长字段对象的一个拷贝，使用专用的应用程序来操作这个拷贝，然后入库这个被修改的拷贝，出库和入库这两个名词大致对应读和写。在一些系统中，入库可以创建对象的一个新版本而并不删除旧版本。

10.10　总结

大多数计算机系统中存在多种数据存储类型。根据数据被存取的速度、购买介质时每单位数据的成本和介质的可靠性，可对存储介质进行分类。通常的介质有高速缓冲存储器、主存储器、快闪存储器、磁盘、光盘和磁带。

存储介质的可靠性由两个因素决定：电源故障或系统崩溃是否导致数据丢失，存储设备发生物理故障的可能性有多大。通过保留数据的多个拷贝可以减少物理故障的可能性。对磁盘来说可以使用镜像。更复杂的一类方法是基于廉价磁盘冗余阵列（RAID）。通过把数据拆分到多

个磁盘上，可以提高大数据量访问的吞吐率。通过对多个磁盘引进冗余，可以显著提高可靠性。几种不同的 RAID 组织形式，具有不同的成本、性能和可靠性。最常用的是 RAID 1 级（镜像）和 RAID 5 级。

文件在逻辑上是映射到磁盘块上的一个记录序列。把数据库映射到文件的一种方法是使用多个文件，每个文件只存储某一确定长度的记录。另一种方法是构造文件，使之能存放不同长度的记录。实现变长记录的技术有多种，包括分槽页方法、指针方法和保留空间方法。

因为数据以块为单位在磁盘存储器和主存储器之间传输，按块分配文件记录比较合适，它使得单一一块可以包含相关的记录。如果只要进行一次块访问就可以存取多个记录，我们就节省了磁盘访问。因为磁盘访问通常是数据库系统性能的瓶颈，有效地按块分配记录可以获得显著的性能提高。

减少磁盘访问数量的一种方法是在主存储器中保留尽可能多的块。因为在主存储器中保留所有的块是不可能的，需要为块的存储而管理主存储器中可用空间的分配。缓冲区是主存储器的一部分，可用于存储磁盘块的拷贝。负责分配缓冲区空间的子系统称为缓冲区管理器。

面向对象数据库的存储系统不同于关系数据库的存储系统，例如，它们必须处理大对象，必须支持持久化指针。它们还必须有探测虚悬指针的方法。为了高效地进行持久化指针的解引用，它们可以使用基于软件和硬件的混写方法。基于硬件方法利用了对虚拟存储器管理的支持，这种支持由硬件实现且许多现代操作系统都允许用户程序去使用。

习题

10.1　列出你日常使用的计算机上用到的物理存储介质，给出每种介质上数据存取的速度。

10.2　磁盘控制器对坏扇区的重映射如何影响数据检索率？

10.3　考虑如下四个磁盘上的数据和奇偶校验块：

盘 1	盘 2	盘 3	盘 4
$B1$	$B2$	$B3$	$B4$
$P1$	$B5$	$B6$	$B7$
$B8$	$P2$	$B9$	$B10$
⋮	⋮	⋮	⋮

B_i 表示数据块，P_i 表示奇偶校验块。奇偶校验块 P_i 是数据块 B_{4i-3} 到 B_{4i} 的校验块。这种排列会带来什么问题？

10.4　一个磁盘块正在被写的时候发生电源故障会导致块只被写了一部分。假设部分写的块可以被探测出来。一个原子的写块操作是指或者完成对整块的写，或者什么都没有写（即不会部分写）。给出采用如下 RAID 方法时实现原子的写块操作的方法，要求其中包括从故障中的恢复工作。

(a) RAID 1 级（镜像）

(b) RAID 5 级（块级，分布奇偶校验）

10.5　RAID 系统通常允许你更换发生故障的磁盘而不需要停止对系统的访问。因此，在系统运行时，发生故障的磁盘上的数据必须被重建，并写到新更换的磁盘上。哪一级 RAID 在重建和持续磁盘访问之间干扰最小？解释你的答案。

10.6　在下面的每种情况中，给出由关系代数和查询处理策略构成的一个例子：

(a) MRU 优于 LRU。

(b) LRU 优于 MRU。

10.7　定义术语被钉住的块。

10.8　考虑从图 10-8 的文件中删除记录 5。比较下列实现删除的技术的相对优点：

(a) 移动记录 6 到记录 5 所占用的空间，然后移动记录 7 到记录 6 所占用的空间。

(b) 移动记录 7 到记录 5 所占用的空间。

(c) 标记记录 5 被删除，不移动任何记录。

10.9　给出经过下面每一步后图 10-9 中的文件结构：

(a) 插入 (Brighton, A-323, 1600)。

(b) 删除记录 2。

(c) 插入 (Brighton, A-626, 2000)。

10.10　给出一个数据库应用的例子，在这个例子中，用保留空间的方法表示变长记录优于用指针方法。解释你的答案。

10.11　给出一个数据库应用的例子，在这个例子中，用指针方法表示变长记录优于用保留空间的方法。解释你的答案。

10.12　给出经过下面每一步后图 10-12 中的文件结构：

(a) 插入 (Mianus, A-101, 2800)。

(b) 插入 (Brighton, A-323, 1600)。

(c) 删除 (Perryridge, A-102, 400)。

10.13　如果你试图向图 10-12 中的文件插入记录 (Perryridge, A-929, 3000)，会发生什么？

10.14　给出经过下面每一步后图 10-13 中的文件结构：

(a) 插入 (Mianus, A-101, 2800)。

(b) 插入 (Brighton, A-323, 1600)。

(c) 删除 (Perryridge, A-102, 400)。

10.15　解释为什么按块分配记录会显著影响数据库系统的性能。如果块经过删除变成空的，这个块为什么应被重新使用？解释你的答案。

10.16　如果可能，查出你的计算机系统上的操作系统所使用的缓冲区管理策略。讨论这种策略对数据库系统的实现如何有用。

10.17　在顺序文件组织中，为什么即使只有一条溢出记录，也要使用一个溢出块？

10.18　列出下列存储关系数据库的每个策略的两个优点和两个缺点：

(a) 在一个文件中存储一个关系。

(b) 在一个文件中存储多个关系。

10.19　考虑有两个关系的关系数据库：

course (*course-name*, *room*, *instructor*)

enrollment (*course-name*, *student-name*, *grade*)

定义这些关系的实例，其中有 3 门课程，每门课有 5 个学生。给出这些关系的一个使用聚集文件的结构。

10.20　解释为什么物理 OID 必须比一个指向物理存储位置的指针包含更多的信息。

10.21　如果使用物理 OID，有没有可能改变对象的位置？如果可能，解释系统必须进行什么动作才能正确改变对象位置。如果不能，解释为什么。

10.22　定义术语虚悬指针。描述唯一标识策略如何在面向对象的数据库中协助检测虚悬指针。

10.23　考虑 10.9.3 节中的第 2 部分关于使用硬件混写时不必反混写的例子。解释为什么在这个例子中将页 679.34278 的短标识从 2395 变到 5001 是安全的。会不会有其他的页已经有短标识 5001 了？如果有，你如何处理这种情况？

文献注解

高速缓冲存储器，包括关联存储器，在 Smith [1982] 中有描述和分析。这篇论文中还包含了有关这个主题的大量的参考文献。Patterson 和 Hennessy [1995] 讨论了关于旁视转换缓冲区、高速缓冲存储器和内存管理器单元的硬件方面。

关于磁盘技术的讨论在 Freedman [1983] 和 Harker 等 [1981] 中。关于光盘的讨论见 Kenville [1982]、Fujitani [1984]、O'Leary 和 Kitts [1985]、Gait [1988]，以及 Olsen 和 Kenley [1989]。Ruemmler 和 Wilkes [1994] 对磁盘技术做了很好的概述。Wiederhold [1983]、Bohl [1981]、Trivedi 等 [1980]、ElMasri 和 Navathe [1994] 讨论了磁盘的物理特性。Pechura 和 Schoeffler [1983] 和 Sarisky [1983] 提供了关于软盘的讨论。Dippert 和 Levy [1993] 讨论了快闪存储器。

Ammon 等 [1985] 讨论了高速大容量自动光盘机系统。Christodoulakis 和 Faloutsos [1986] 给出了一个光盘存储的应用实例。Chi [1982]、Hoagland [1985] 讨论了大数据量存储技术。

Gray 等 [1990]、Bitton 和 Gray [1988] 中提出了高度容错的不同磁盘组织结构。Salem 和 Garcia-Molina [1986] 描述了磁盘拆分。Patterson 等 [1988]、Chen 和 Patterson [1990] 讨论了廉价磁盘冗余阵列 (RAID)。Chen 等 [1994] 是有关 RAID 原则和实现的优秀综述。Pless [1989] 讨论了 Reed-Solomon 码。

Katz 等 [1989] 讨论了高性能计算机的磁盘系统体系结构。Nelson 和 Cheng [1992] 比较了实际系统中 IPI 和 SCSI 磁盘及控制器的性能。Loge project [English and Stepanov 1992] 进行了在磁盘控制器中加入智能的实验。Rosenblum 和 Ousterhout [1991] 描述了基于日志的文件系统，这种系统使磁盘存取顺序化。

在支持移动计算的系统中，数据可能被重复广播。广播介质可看作是存储层次中的一层——具有高度延迟的广播盘。Acharya 等 [1995] 讨论了这个问题。Barbara 和 Imielinski [1994] 讨论了移动计算的高速缓存和缓冲区管理。Douglis 等 [1994] 对移动计算的存储问题做了进一步的讨论。

Knuth [1973]、Aho 等 [1983]，以及 Horowitz 和 Sahni [1976] 讨论了基本的数据结构。Teorey 和 Fry [1982]、Smith 和 Barnes [1987] 以及 Ullman [1988] 讨论了数据库系统中文件的组织和存取方式。

有几篇论文描述了特定数据库系统的存储结构。Astrahan 等 [1976] 讨论了 System R、Chamberlin 等 [1981] 回顾了 System R。Stonebraker 等 [1976] 描述了 Ingres 的实现。"Oracle 7 Concepts Manual (Oracle [1992])" 描述了 Oracle 7 数据库系统的存储组织。Chou 等 [1985] 描述了 Wisconsin Storage System (WiSS)。Finkelstein 等 [1988] 描述了一个关系数据库物理设计的软件工具。

大多数操作系统的教科书都讨论了缓冲区管理，包括 Silberschatz 和 Galvin [1994]。Stonebraker [1981] 讨论了数据库系统缓冲区管理器和操作系统缓冲区管理器之间的关系。Chou 和 DeWitt [1985] 给出了数据库系统中缓冲区管理的算法，并做了性能评价。

Wilson [1990]、Moss [1990]、White 和 DeWitt [1992] 给出了不同混写技术的描述和性能

比较。White 和 DeWitt［1994］描述了 ObjectStore OODB 系统和 QuickStore 存储管理器中虚拟存储器映射的缓冲区管理方案，使用这种方案，我们可以映射磁盘页到固定的虚拟存储器地址，尽管它们在缓冲区中没有被钉住。Carey 等［1986］描述了 Exodus 对象存储管理器。Bilirs 和 Orenstein［1993］对面向对象数据库的存储系统作了综述。Jagadish 等［1994］描述了主存储器数据库的存储管理器。

第 11 章　索引和散列

许多查询只涉及文件中的少量记录。例如，查询“找出 Perryridge 分支机构的所有帐户信息”就只涉及所有帐户记录中的一部分。在这种情况下，如果系统将所有记录依次读出，并逐个检查各记录的 *branch-name* 字段是否为“Perryridge”，效率显然很低。理想的做法是系统能够直接定位这些记录。为了实现这种访问数据的方式，我们设计了附加结构，并将其同文件联系起来。

11.1　基本概念

系统中文件的索引在功能上类似于图书馆中书的目录。如果要查某个作者的著作，我们会先去查作者目录，目录中的某张卡片会告诉我们到哪里去找这本书。为了便于我们在目录中查找，图书馆将这些卡片按作者名字的字典顺序依次排放，这样我们不必检查每一张卡片就能获得所需信息。

在实际的数据库中，用上述方法建索引会使索引太大而不能有效处理，所以要采用一些更加复杂的索引方法。下面将讨论其中的几种技术。基本的索引有两种：

- 顺序索引。这种索引基于对值的一种排序。
- 散列索引。这种索引基于将值平均分布到若干散列桶中。一个值所属的散列桶由一个函数来决定，该函数称为散列函数。

我们将考虑顺序索引和散列索引的几种技术。没有哪一种技术是最好的，因为每种技术都有各自最适合的数据库应用。对这些技术的评价必须考虑下面一些因素：

- 访问类型。能有效支持的访问类型，包括根据指定的属性值找到相应记录，以及根据给定属性值的范围找到该范围中的所有记录。
- 访问时间。访问一个或多个数据项所需的时间。
- 插入时间。在索引中插入一个新数据项所需的时间，包括找到插入该项的正确位置的时间和修改索引结构所需的时间。
- 删除时间。在索引中删除一个数据项所需的时间，包括找到待删除项所需的时间和修改索引结构所需的时间。
- 空间开销。索引结构所需的额外的存储空间。如果较小，那么牺牲一定的空间来换取性能的提高是值得的。我们经常需要在一个表上有多个索引，就像图书馆中通常有多个目录：按作者的、按主题的、按题目的等。用于在文件中查找记录的属性或属性集称为搜索码。注意，码的这一含义与主码、候选码以及超码有所不同。码的多重含义在实践中已广为接受。根据搜索码的概念，不难发现，如果一个文件上有多个索引，那么也就有多个搜索码。

11.2　顺序索引

为了能迅速随机地访问文件中的记录，我们可以使用索引结构。每个索引结构有一个特定的搜索码与之关联。正如图书馆目录一样，索引按顺序存储搜索码的值，并将搜索码与包含该搜索码的记录关联起来

索引文件中的记录自身也可以按一定的顺序排列，正如图书馆中的书按某个属性如杜威十进制数（Dewey decimal number）顺序存放一样。一个文件可以有多个索引，分别对应于不同的搜索码。如果包含记录的文件按照某个搜索码指定的顺序排序，那么该搜索码对应的索引称为主索引。（有时将建立在主码上的索引称为主索引，这种叫法是不标准的，应该避免。）主索引也称为聚集索引。通常情况下主索引建在主码之上，但也并非总是如此。顺序与文件中记录的物理顺序不同的那些索引称为辅助索引或非聚集索引

11.2.1　主索引

本节假定所有文件都按照某个搜索码顺序存储，称这种在某个搜索码上有主索引的文件为索引顺序文件，它们代表数据库系统中最早采用的索引模式之一。这种模式针对既需对数据进行顺序处理又需对数据进行随机访问的应用而设计。

图 11-1 是银行例子中 *account* 记录的一个索引顺序文件。在图示的例子中，*branch-name* 是搜索码，而记录按照该搜索码顺序存放。

Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	700	
Round Hill	A-305	350	

图 11-1　*account* 记录的顺序文件

1. 稠密索引和稀疏索引

可以使用的顺序索引有两类：

- 稠密索引。对应文件中搜索码的每一个值有一个索引记录（或索引项）。索引记录包括搜索码值以及指向具有该搜索码值的第一个数据记录的指针。（有的作者用术语稠密索引表示对应文件中的每个记录有一个索引记录的索引。）
- 稀疏索引。只为搜索码的某些值建立索引记录。和稠密索引一样，每个索引记录也包括一个搜索码值和指向具有该搜索码值的第一个数据记录的指针。在寻找记录时，首先找到所有小于或等于所找记录搜索码值的索引项中具有最大搜索码值的那一项，从该索引项指向的记录开始，沿着文件中的指针找下去，直到找到所需记录为止。

图 11-2 和图 11-3 分别是为 *account* 文件建立的稠密索引和稀疏索引。假如现在要找出 Perryridge 分支机构的记录，利用图 11-2 的稠密索引，可以顺着指针直接从文件中找到 Perryridge 的第一条记录。处理完这条记录后，可以根据这条记录中的指针找到按搜索码（*branch-name*）值排在下一个的记录。继续对记录进行处理，直到遇到分支机构不是 Perryridge 的记录。如果我们利用的是稀疏索引（图 11-3），那么就没有一个索引项是对应于“Perryridge”的。由于（按字母顺序）在“Perryridge”前的最后一个索引项为“Mianus”，我们沿着这一指针得到它所指向的记录，接着顺序读入 *account* 文件，直到找到第一个 Perryridge 记录，这时开始进行处理。

如上所示，利用稠密索引通常可以比稀疏索引更快地定位一个记录。但是，稀疏索引也有

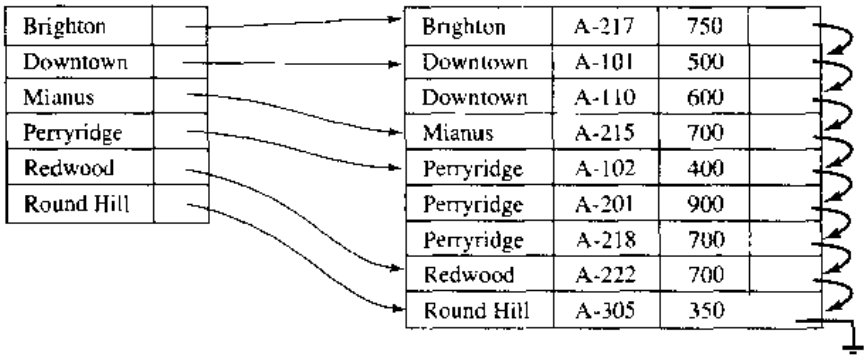


图 11-2　稠密索引

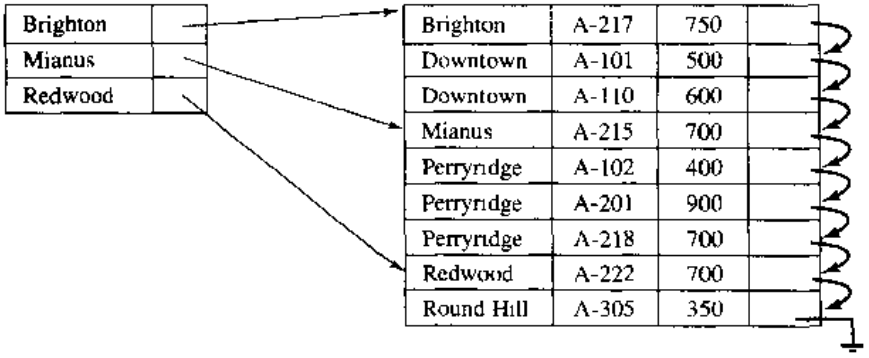


图 11-3　稀疏索引

比稠密索引优越的地方，主要表现在所占空间较小，并且插入和删除时的维护开销也较小。

系统设计者必须在访问时间和空间开销之间进行权衡。尽管有关这一权衡的决定依赖于具体的应用，但是为每个块建一个索引项的稀疏索引是一个较好的折中。原因在于，处理数据库查询的开销主要由把块从磁盘上取到主存中的时间决定。一旦将块放入主存，扫描整个块的时间是可以忽略的。使用这样的稀疏索引，可以定位包含所要查找记录的块。这样，只要记录不在溢出块（见 10.7.1 节）中，就既能尽量减少索引大小（因而也就减少空间开销），又能使块访问数最小。

要使上述技术完全通用，必须考虑一个搜索码值跨多个块的情况。通过对以上方法稍做修改就可以处理这样的情况。

2. *多级索引*

即使采用稀疏索引，索引本身有时也会变得非常大而难于有效处理。在实践中，一个文件有 100 000 个记录而每一块存储 10 个记录是有可能的。如果每一块有一个索引记录，那么索引就有 10 000 个记录。索引记录比数据记录小，因而不妨假设一块中能容纳 100 个索引记录。这样，索引将占据 100 块。这样大的索引在磁盘上以顺序文件存储。

如果索引小到能够放在主存中，搜索一个索引项的时间会很少。但是，如果索引过大而必须放在磁盘上，那么搜索一个索引项就必须几次读磁盘块。虽然在索引文件上可以用二分法来定位索引项，但是搜索的开销依然很大。如果索引占据 b 块，二分搜索需要读取的块数高达 $\lceil \log_2(b) \rceil$（$\lceil x \rceil$ 表示大于或等于 x 的最小整数，即向上取整）。对有 100 块的索引，二分搜索需要读块七次。在读一个块需要 30 毫秒的磁盘系统中，该搜索将耗时 210 毫秒，这是相当长的一段时间。注意，如果使用了溢出块，那么还不能使用二分搜索。这种情况下通常采用顺序搜

索，即需要读块 b 次，这将耗费更长的时间。因此，搜索一个大的索引可能是一个相当耗时的过程。

为了解决这个问题，我们像对待其他任何顺序文件那样来对待索引文件，即在主索引上构造一个稀疏索引，如图 11-4 所示。如果要搜索一个记录，首先在外层索引上用二分法找到不大于所需搜索码值的最大搜索码值所对应的记录。指针指向一个内层索引块。我们对这一块做扫描，直到找到不大于所需搜索码值的最大搜索码值所对应的记录。这一记录的指针指向包含我们所查找记录的文件块。

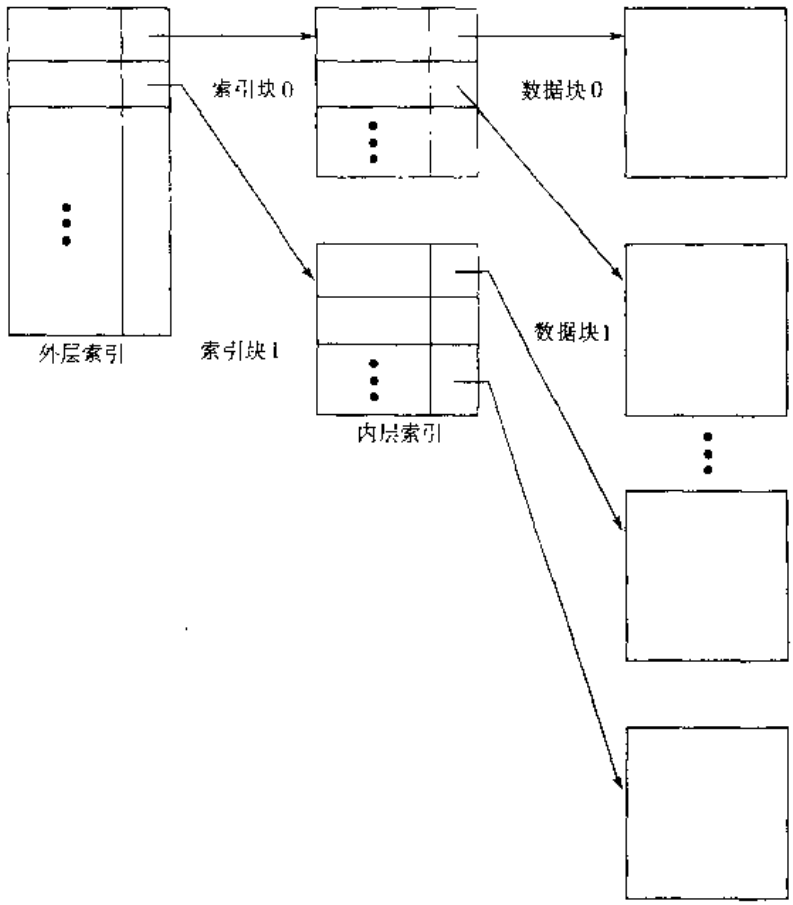


图 11-4　两级稀疏索引

如果假设外层索引已在主存中，那么使用两级索引时，我们只需读一次索引块，而不像使用二分搜索时那样需要七次。如果文件非常大，大到甚至外层索引也可能不为主存所容纳。在这种情况下，还可以创建另一级索引。实际上，可以根据需要多次重复此过程。具有两层或两层以上的索引称为多级索引。利用多级索引搜索记录比用二分法搜索要快得多。各级索引可以分别对应于一个物理存储单位，因此，可以有磁道级、柱面级和磁盘级的索引。

多级索引和树结构（如用于内存索引的二叉树）紧密相关。稍后我们将在 11.3 节讨论两者间的联系。

3. 索引的更新

无论采用何种形式的索引，每当文件中有记录插入或删除时，索引都需要更新。下面我们将介绍一级索引的更新算法。

• 删除。删除记录时，首先找到要删除的记录。如果要删除记录的搜索码值在文件中是唯一的，那么该搜索码值要从索引中删除。对稠密索引而言，删除一个搜索码值类似于从文件

中删除一条记录。对稀疏索引而言，如果被删除搜索码值在索引中有索引项，可以通过用（按搜索码顺序的）下一搜索码值替代该索引项来实现对码值的删除。如果下一搜索码值已经有其索引项，那么该索引项就应该删除而不是被替代。

• 插入。首先用出现在待插入记录中的搜索码值来进行查找。如果索引是稠密的，并且该搜索码值不在索引中，我们就把该搜索码值插入索引中。如果索引是为每个块保存一个索引项的稀疏索引，只要没有新块产生，索引就无需做任何改动。在产生新块的情况下，新块中（按搜索码顺序的）第一个搜索码值将被插入索引中。

多级索引的插入和删除算法只是对上述算法的一个简单扩充。在插入和删除时，底层索引的更新如上所述。而对于第二层而言，底层索引不过是一个包含记录的文件。因此，如果底层索引发生了改变，第二层索引就可以像上面描述的那样进行更新。如果还有更高层的索引，可以采用同样的技术继续进行下去。

11.2.2　辅助索引

候选码上的辅助索引看起来和稠密主索引没有太大的区别，只不过索引中一系列的后续值指向的记录不是连续存放的。然而辅助索引的结构通常可以和主索引不同。如果主索引的搜索码不是候选码，那么对于某个搜索码值而言，索引只要指向具有该值的第一个记录就足够了，因为其他记录可以通过对文件进行顺序扫描得到。

但是，如果辅助索引的搜索码不是一个候选码，那么仅仅指出各搜索码值的第一个记录是不够的。具有同一个索引码值的其他记录可能分布在文件的各个地方，因为记录按主索引而不是辅助索引的搜索码顺序存放。因此，辅助索引必须包含指向每一记录的指针。可以用增加一层间接的方法来实现非候选码上的辅助索引。在这样的辅助索引中，指针并不直接指向文件，而是每个指针指向一个包含文件指针的桶。图 11-5 给出了这样的一个辅助索引结构，它在 *account* 文件的搜索码 *balance* 上使用了一层额外间接。

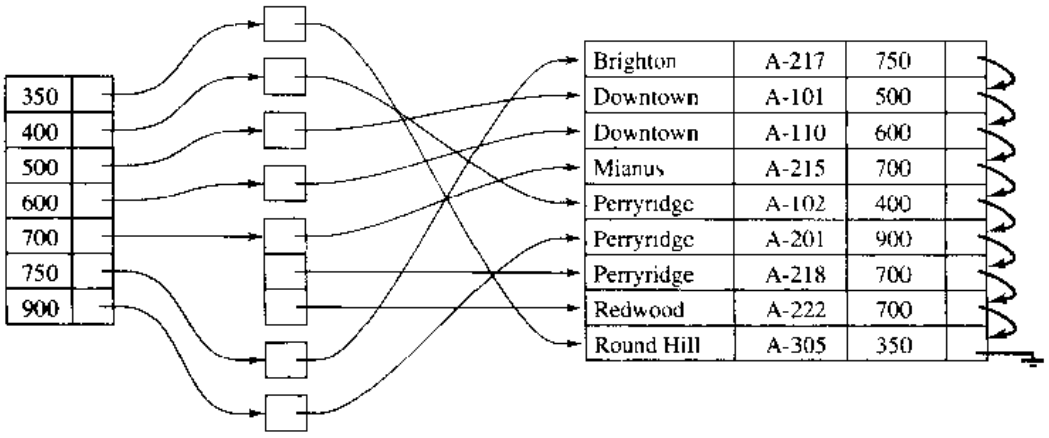


图 11-5　文件 *account* 上基于非候选码 *balance* 的辅助索引

按主索引顺序对文件进行顺序扫描是非常有效的，因为文件中记录的物理存储顺序和索引顺序一致。但是，除了极少数特殊情况外，我们不能使存储文件的物理顺序既和主索引的搜索码顺序相同，又和辅助索引的搜索码顺序相同。由于辅助码的顺序和物理码的顺序不同，因而如果想要按辅助码的顺序对文件进行顺序扫描，那么读每一条记录都很可能需要从磁盘读入一个新的块。

稀疏主索引可以只存储部分搜索码值，因为如前所述，通过顺序扫描文件的一部分，总可

以找到两个有索引项的搜索码值之间的搜索码值所对应的记录。如果辅助索引只存储部分搜索码值，两个有索引项的搜索码值之间的搜索码值所对应的记录可能存在于文件的任何地方，并且通常只能通过扫描整个文件才能找到它们。因此，辅助索引必须是稠密的，必须对应每个搜索码值都有一个索引项且对应文件中的每个记录都有一个指针。

前面所描述的关于删除和插入的过程也适用于具有多个索引的文件。每当文件被修改时，它的每个索引都必须被更新。

辅助索引能够提高没有使用主索引搜索码的查询的速度。但是，辅助索引明显增加了数据库更新的开销。数据库设计者根据对查询和更新相对频率的估计来决定哪些辅助索引是需要的。

11.3　B^+ 树索引文件

索引顺序文件组织最大的缺点在于随着文件的增大，索引查找性能和数据顺序扫描性能都会下降。虽然这种性能下降可以通过对文件进行重新组织来弥补，但是我们不希望频繁地进行重组。

一些索引结构能在有数据插入和删除的情况下保持其有效性，B^+ 树索引结构就是其中使用最广泛的一个。B^+ 树索引采用平衡树结构，其中每个叶结点到根的路径长度相同，每个非叶结点有 $\lceil n/2 \rceil \sim n$ 个子女，n 对特定的树是固定的。

B^+ 树结构会增加文件插入和删除处理的时间开销和空间开销。但是即使对更新频率较高的文件来说这种开销也是可以接受的，因为这样能够避免文件重组的开销。此外，由于结点在极端情况下可以是半空的（如果这些结点具有最少子结点数的话），这将造成空间的浪费。但是，考虑到 B^+ 树所带来的性能提高，这种空间浪费也是可以接受的。

11.3.1　B^+ 树的结构

B^+ 树索引是一个多级索引，但是其结构不同于多级索引顺序文件。典型的 B^+ 树结点结构如图 11-6 所示。它最多包含 $n-1$ 个搜索码值 K_1，K_2，…，K_{n-1}，以及 n 个指针 P_1，P_2，…，P_n。每个结点中的索引码值按次序存放：即如果 $i<j$，那么 $K_i<K_j$。

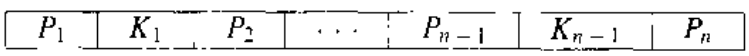


图 11-6　典型的 B^+ 树结点

我们首先考察叶结点的结构。对 $i=1, 2, \cdots, n-1$，指针 P_i 指向具有搜索码值 K_i 的一个文件记录或指向一个指针桶，桶中的每个指针指向具有搜索码值 K_i 的一个文件记录。指针桶只在搜索码不是主码且文件不按搜索码顺序存放时才使用。指针 P_n 有特殊的作用，我们稍后讨论。

图 11-7 是 *account* 文件的 B^+ 树的一个叶结点，其中 $n=3$，搜索码是 *branch-name*。注意，由于 *account* 文件按搜索码的顺序存放，所以叶结点中的指针直接指向文件。

知道了叶结点的结构之后，我们来看一看结点怎样得到其搜索码值。每个叶结点最多可有 $n-1$ 个值，我们允许叶结点包含值的个数最少为 $\lceil (n-1)/2 \rceil$。各叶结点中值的范围互不相交。因此，如果 L_i 和 L_j 是两个叶结点且 $i<j$，那么 L_i 中的所有搜索码值都小于 L_j 中的所有搜索码值。要使 B^+ 树索引成为稠密索引，各搜索码值都必须出现在某个叶结点中。

现在可以来解释指针 P_n 的作用了。既然各叶结点之间按照所含的搜索码值大小有一个线性的顺序，我们就用 P_n 将叶结点按搜索码顺序串在一起。这种排序使我们能够高效地对文件进行顺序处理。

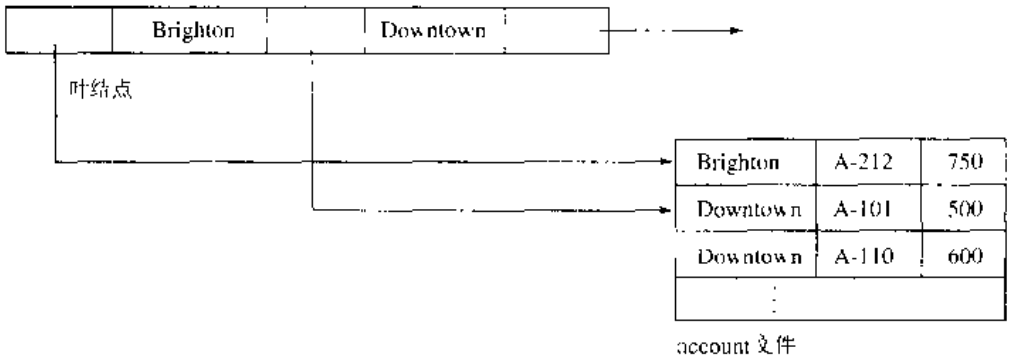


图 11-7　*account* 文件的 B⁺ 树索引（$n-3$）的一个叶结点

B⁺ 树的非叶结点形成叶结点上的一个多级（稀疏）索引。非叶结点的结构和叶结点相同，只不过非叶结点中的所有指针都指向树中结点。一个非叶结点至少包含 $\lceil n/2 \rceil$ 个指针，最多 n 个。结点的指针数称为该结点的扇出。

我们来看一个含 m 个指针的结点。对 $i=2, 3, \cdots, m-1$，指针 P_i 指向一棵子树，该子树所有结点的索引码值大于等于 K_{i-1} 且小于 K_i。指针 P_m 指向子树中所含搜索码值大于等于 K_{m-1} 的那一部分，而指针 P_1 指向子树中所含搜索码值小于 K_1 的那一部分。

根结点与其他非叶结点不同，根结点包含的指针数可以小于 $\lceil n/2 \rceil$。但是，除非整棵树只有一个结点，否则根结点必须至少包含两个指针。对任意 n，我们总可以构造满足上述要求的 B⁺ 树。图 11-8 给出了 *account* 文件的一棵完整 B⁺ 树，其中 $n=3$。为简单起见，我们省略了指向文件的指针和空指针。图 11-9 是根结点所含值的个数必须小于 $\lceil n/2 \rceil$ 的 B⁺ 树，它是 *account* 文件上的一棵 B⁺ 树，其中 $n=5$。

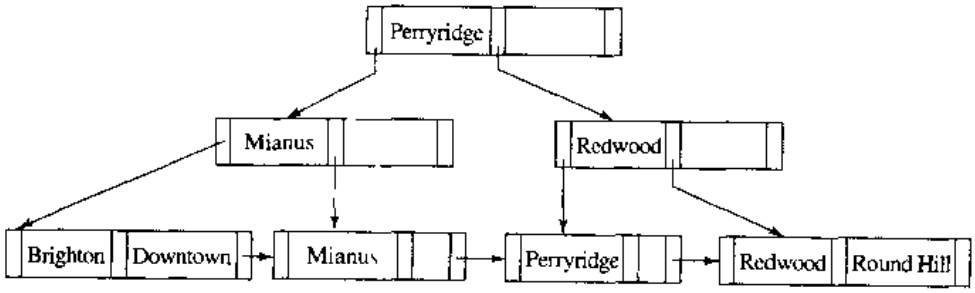


图 11-8　*account* 文件的 B⁺ 树（$n-3$）

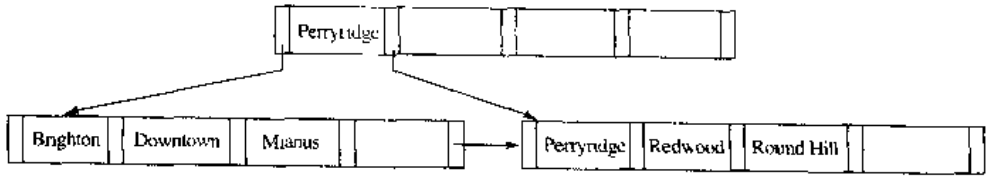


图 11-9　*account* 文件的 B⁺ 树（$n-5$）

我们所给 B⁺ 树的例子都是平衡树，即从根到叶结点的每条路径长度相同。对于 B⁺ 树来说这是一个必需的性质，实际上 B⁺ 树的 "B" 就表示 "平衡（balance）"。正是平衡这一性质保证了 B⁺ 树索引有良好的查找、插入和修改性能。

11.3.2　B⁺ 树上的查询

现在来看一看如何利用 B⁺ 树处理查询。假设要找出搜索码值为 k 的所有记录。首先我们

检查根结点，找到大于 k 的最小搜索码值，假设是 K_i，那么我们顺着指针 P_i 走到另一个结点；如果 $K < K_1$，那么我们顺着指针 P_1 走至另一个结点。如果找不到这样的值，且 $K \geqslant K_{m-1}$，其中 m 是该结点中的指针数，则这种情况下，我们沿着 P_m 走到另一个结点。在到达的结点中，我们再次寻找大于 k 的最小搜索码值，并且再次像上面那样沿相应结点而下。最终我们将到达一个叶结点。如果在该叶结点中有某个搜索码值 K_i 等于 k，那么指针 P_i 指向我们所需要的记录或指针桶。如果在该叶结点中找不到值 k，则不存在码值为 k 的记录。

因此在处理查询过程中，我们需要遍历树中从根到某个叶结点的一条路径。如果文件中有 K 个搜索码值，那么这条路径的长度不超过$\lceil \log_{n/2}(K) \rceil$。

实践中只需访问几个结点。结点的大小一般等于磁盘块大小，通常为 4KB。如果搜索码的大小为 12 字节，磁盘指针的大小为 8 字节，那么 n 大约为 200。即使采用更保守的估计，假设搜索码大小达到 32 字节，n 也大约为 100。如果文件中搜索码值共有 1 百万个，一次查找也只需要访问 $\lceil \log_{50}(1\,000\,000) \rceil = 4$ 个结点。因此，查找时最多只需要从磁盘读四个块。通常根结点访问最频繁，很可能已在缓冲中，因此一般只要从磁盘读取三块或更少。

B^+ 树结构与内存中树结构（如二叉树）的最大区别在于结点的大小，因而树的高度也不同。二叉树的结点很小，每个结点最多有两个指针。而 B^+ 树的结点非常大（一般是一个磁盘块），每个结点可以有大量指针。因此，B^+ 树一般矮而胖，不像二叉树那样瘦而高。在平衡二叉树中，查找路径的长度可达 $\lceil \log_2(K) \rceil$，其中 K 为搜索码值的个数。当 K 如上例中那样为 1 000 000 时，平衡二叉树大约需要访问 20 个结点，如果每个结点在不同的磁盘块中，处理一个查找需要读 20 个块，但对 B^+ 树来说只需读四个块。

11.3.3　B^+ 树的更新

插入和删除要比查找复杂得多，因为结点可能因为插入变得过大而需要分裂或因为删除变得过小（指针数少于 $\lceil n/2 \rceil$）而需要合并。此外，当一个结点分裂或一对结点合并时，必须保证 B^+ 树能保持平衡。为了说明 B^+ 树中处理插入和删除的思想，下面暂时假设结点不会变得过大或过小。在这个假设前提下，插入和删除将按下述方式进行。

- 插入。使用和查找一样的技术，先找到被插入的搜索码应在的叶结点。如果该值已经存在，那么在文件中加入一个新记录并且必要时在相应的指针桶中加入一个指针。如果该搜索码值在此叶结点中不存在，那么在此叶结点的适当位置加入该值以保证结点中搜索码的顺序。然后在文件中插入一个新记录并且必要时创建一个指针桶，指针桶中放入对应该新记录的指针。
- 删除。使用和查找一样的技术，先找到需要删除的记录并从文件中将该记录删除。如果该搜索码值没有对应的指针桶或者由于删除使该指针桶变空，那么从该搜索码值所在的叶结点中将其删除。

现在我们来看结点必须分裂的一个例子。假设需要往图 11-8 所示的 B^+ 树中加入 *branch-name* 值为“Clearview”的一条记录。按照查找算法，“Clearview”应出现在包含“Brighton”和“Downtown”的结点中。该结点已没有插入值“Clearview”的空间，因此该结点分裂为两个结点。图 11-10 表示了在包含“Brighton”和“Downtown”的结点中插入“Clearview”后分裂而成的两个结点。一般来说，我们将这 n 个值（叶结点中原有的 $n-1$ 个值再加上新插入的值）分为两组，前 $\lceil n/2 \rceil$ 个放在原来的结点中，剩下的放在新结点中。

在分裂结点后，我们必须将这个新结点加入 B^+ 树结构中。该例中，新结点的最小搜索码值是“Downtown”，我们需要将该搜索码值插入到被分裂结点的父结点中。图 11-11 给出了插

入后的结果，搜索码值“Downtown”插入到了父结点中。由于在父结点中有加入搜索码值的空间，所以能进行这样的插入。如果父结点也没有空间，那么该父结点也要进行分裂。最坏的情况下，从叶结点到根的路径上所有结点都需要分裂。如果根也分裂了，那么这棵树的深度就加大了。

图 11-10　插入“Clearview”时叶结点的分裂

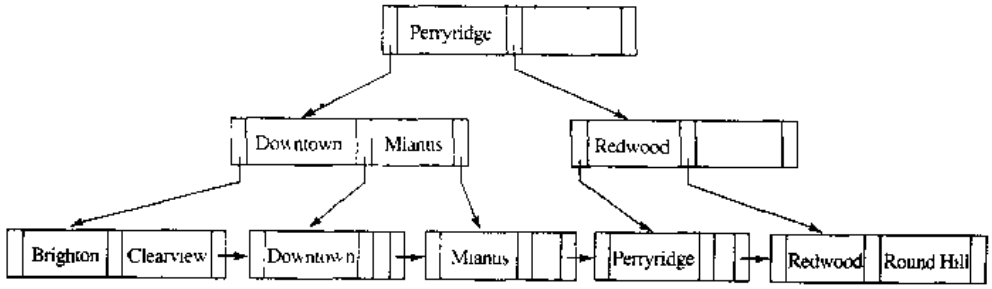


图 11-11　在图 11-8 的 B^+ 树中插入“Clearview”

往 B^+ 树中进行插入必须确定需要进行插入的叶结点 l。如果产生分裂，则将新结点插入结点 l 的父结点中。如果这一插入导致分裂，就递归处理下去，直到不再产生分裂或创建了一个新的根。

图 11-12 用伪码给出了插入算法。在伪码中，$L.K_i$ 和 $L.P_i$ 分别表示结点 L 中第 i 个值和第 i 个指针。伪码中用函数 *parent*（L）来得到结点 L 的父结点。在一开始寻找叶结点时，可以计算从根到叶的路径上的结点列表，以后就可以利用它来高效地寻找这条路径中任何一个结点的父结点。伪码把索引项（V，P）插入到一个结点中。对叶结点而言，指针实际存放在码值前，因此叶结点中 P 真正存放在 V 前。而在内部结点中，P 存放在 V 的后面。

现在来考虑使树结点中指针变少的删除操作。首先从图 11-11 的 B^+ 树中删除“Downtown”。我们用查找算法定位“Downtown”的索引项。当从“Downtown”索引项所在的叶结点中把它删除后，叶结点就变为空的。本例中，由于 $n=3$ 且 $0<\lceil (n-1)/2 \rceil$，这个结点必须从 B^+ 树中去除。要删除一个叶结点，必须从其父结点中删除指向它的指针。本例中，这一删除使原来有三个指针的父结点现在只剩下两个指针。由于 $2 \geqslant \lceil n/2 \rceil$，这个结点仍足够大，因此删除操作结束。产生的 B^+ 树见图 11-13。

当需要对叶结点的父结点做删除时，父结点本身可能会变得很小。这正是当我们从图 11-13 所示 B^+ 树中删除“Perryridge”时发生的情况。对 Perryridge 项的删除使一个叶结点变空。当删除该结点的父结点中指向它的指针时，父结点中只剩下一个指针。由于 $n=3$，$\lceil n/2 \rceil = 2$，因此只有一个指针就太少了。但由于父结点中包含有用的信息，因此我们不能简单地删除它，而要看它兄弟结点（包含一个搜索码 Mianus 的非叶结点）的情况。这个兄弟结点有空间容纳由于删除而变得太小的结点中的信息，因此合并这两个结点。现在这个兄弟结点包含了码“Mianus”和“Redwood”，从而使另一个结点（只包含搜索码“Redwood”）中的信息成为冗余的，因此可以从它的父结点（在这里正好是根结点）中删除。结果如图 11-14 所示。注意到在删除操作后根结点只有一个指针，因此根结点也要被删除，根结点唯一的孩子变成根，整个 B^+ 树的深度减少 1。

```
procedure insert(value V, pointer P)
    找到应该包含值 V 的叶结点 L
    insert_entry(L, V, P)
end procedure

procedure insert_entry(node L, value V, pointer P)
    if (L 有插入(V, P)的空间)
        then 在 L 中插入(V, P)
        else begin /* 分裂 L */
            创建结点 L'
            if (L 是叶结点) then begin
                令 V' 是满足这样一个条件的值:
                值 L.K1, ..., L.Kn-1, V 中有 ⌈n/2⌉ 个比 V' 小
                令 m 是使 L.Km ≥ V' 的最小值
                将 L.Pm, L.Km, ..., L.Pn-1, L.Kn-1 放入 L' 中
                if (V < V') then 在 L 中插入(P, V)
                else 在 L' 中插入(P, V)
            end
            else begin
                令 V' 是满足这样一个条件的值:
                值 L.K1, ..., L.Kn-1, V 中有 ⌈n/2⌉ 个大于或等于 V'
                令 m 是使 L.Km ≥ V' 的最小值
                将 Nil, L.Km, ..., L.Pn-1, L.Kn-1, L.Pn 加入 L' 中
                从 L 中删除 L.Km, ..., L.Pn-1, L.Kn-1, L.Pn
                if (V < V') then 在 L 中插入(V, P)
                else 在 L' 中插入(V, P)
                从 L' 中删除 (Nil, V') from L'
            end
            if (L 不是树根)
                then insert_entry(parent(L), V', L');
                else begin
                    创建有两个子结点 L 和 L' 以及一个值 V'
                    的新结点 R 使 R 成为树根
                end
            if (L) 是叶结点 then begin /* 修正下一子结点指针 */
                置 L'.Pn = L.Pn;
                置 L.Pn = L'
            end
        end
end procedure
```

图 11-12　往 B^+ 树中插入索引项

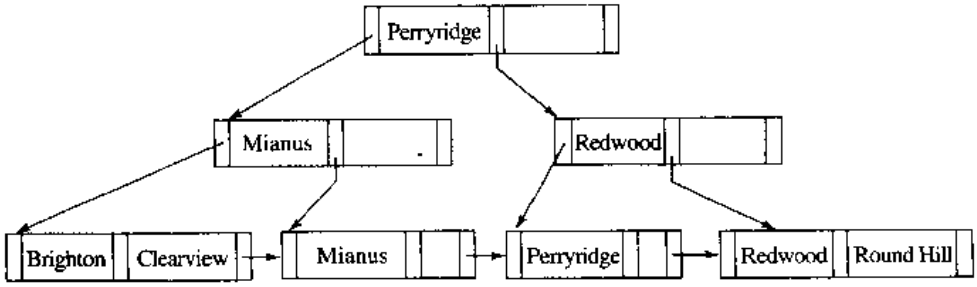


图 11-13　从图 11-11 所示 B^+ 树中删除 "Downtown"

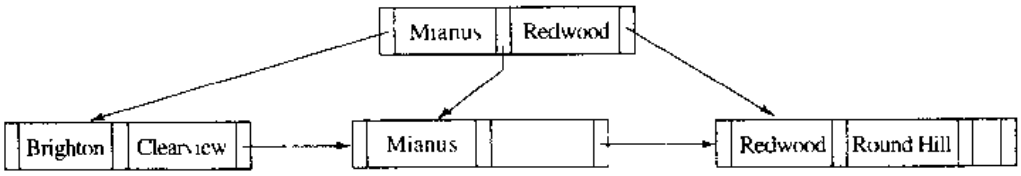


图 11-14　从图 11-13 所示 B^+ 树中删除 "Perryridge"

合并结点并不总是可行的。作为示例，考虑从图 11-11 的 B⁺ 树中删除“Perryridge”。这个例子中，“Downtown”索引项仍然是树的一部分。包含“Perryride”的结点这次又被删除为空，这个叶结点的父结点于是变得很小（只有一个指针）。但这次，它的兄弟结点中已经包含了最大数目的指针：三个，因此它不能再容纳更多的指针。这里的解决方法是重新分布指针，使每个结点包含两个指针，结果如图 11-15 所示。值得注意的是值的重新分布使两个兄弟结点的父结点中的搜索码也必须改变。

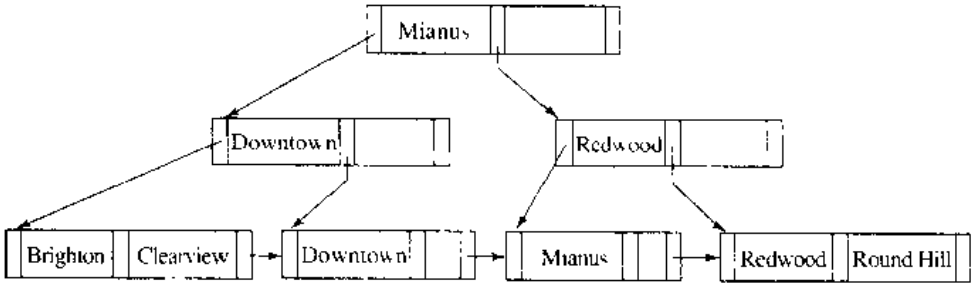


图 11-15　从图 11-11 所示 B⁺ 树中删除“Perryridge”

总的来说，为了在 B⁺ 树中删除一个值，我们要查找并删除该值。如果结点太小的话，我们从它的父结点中把它删除。这个删除导致删除算法的递归应用，直到到达树的根结点或父结点在删除后足够满或产生指针的重新分布。

图 11-16 给出了对 B⁺ 树进行删除的伪码。过程 swap _ variables（L，L'）仅仅交换两个（指针）变量 L 和 L' 的值，对树本身毫无影响。伪码使用条件“指针/值太少”。对非叶结点，这意味着少于 $\lceil n/2 \rceil$ 个指针；对叶结点，这意味着少于 $\lceil (n-1)/2 \rceil$ 个值。伪码通过向相邻的结点借一个索引项来实现重分布，也可以通过在两个结点间平分索引项来实现重分布。伪码从一个结点中删除索引项（V，P）。对叶结点而言，指针实际存放在码值前，因此叶结点中 P 真正存放在 V 前；而在内部结点中，P 存放在 V 的后面。

虽然对 B⁺ 树的插入和删除非常复杂，但它们只需较少的操作。可以证明最坏情况下插入和删除所需的操作数正比于 $\log_{\lceil n/2 \rceil}(K)$，其中 n 是结点中最大的指针数目，K 是搜索码值的个数。换句话说，插入和删除的代价正比于树的高度，因此 B⁺ 树总是很低。正是 B⁺ 树的操作速度使其成为数据库实现中常用的索引结构。

11.3.4　B⁺ 树文件组织

前面提到过，索引顺序文件组织的缺点是文件增大时性能下降。通过在文件上使用 B⁺ 树索引来解决索引查找时性能下降的问题。通过用 B⁺ 树的叶结点层次来组织包含真实记录的块，可以解决存储实际记录时的性能下降问题。在这样的结构中，B⁺ 树结构不仅用做索引，同时也是文件中记录的组织者。

在 B⁺ 树文件组织中，树叶结点中存储的是记录而不是指向记录的指针。图 11-17 给出了一个 B⁺ 树文件组织的例子。由于记录通常比指针大，一个叶结点中能存储的记录数目要比一个非叶结点中能存储的指针数目少。但叶结点仍然要求至少是半满的。

在 B⁺ 树文件组织中，插入和删除的处理与 B⁺ 树索引中索引项的插入和删除一样。当具有给定码值 v 的记录插入时，我们通过在 B⁺ 树中查找 $\leqslant v$ 的最大码来定位应该包含该记录的块。如果定位到的块有足够的空间来存放记录，记录就存放在该块中。否则，就像 B⁺ 树插入那样，该块被一分为二，其中记录（按 B⁺ 树码值的顺序）被重新分布，以给新记录创造空间。这种分裂以通常的形式在 B⁺ 树中向上传播。当一个记录被删除时，首先从包含它的块中将它删除。

```
procedure delete(value V, pointer P)
    找到包含(V,P)的叶结点 L
    delete_entry(L, V, P)
end procedure
procedure delete_entry(node L, value V, pointer P)
    从 I 中删除(V,P)
    if(L 是根且只剩下一个孩子)
    then 使 L 的孩子成为新的树根并删除 L
    else if(L 值/指针太少) then begin
    令 L'为 parent(L)的前一孩子或后一孩子
    令 V'为 parent(L)中介于指针 L 和 L'之间的值
    if(L 和 L'中的项能放在一个结点中)
    then begin /*合并结点*/
        if(L 是 L'的前驱) then swap_variables(L, L')
        if(L 非叶结点)
            then 将 V'以及 L 中所有指针和值附加到 L'中
            else 将 L 中所有(K_i, P_i)对附加到 L'中;置 L'.P_n = L.P_n
        delete_entry(parent(L), V'L);删除结点 L
    end
    else begin/*重新分布:从 L'借一个索引项*/
        if(L'是 L 的前驱)then begin
            if(L 是非叶结点)then begin
                令 m 满足 L'.P_m 是 L'的最后一个指针
                从 L'中去除(L'.K_{m-1}, L'.P_m)
                插入(L'.P_m, V')并通过将其他指针和值右移使之成为
                L 中的第一个指针和值
                用 L'.K_{m-1}替换 parent(L)中的 V'
            end
            else begin
                令 m 满足(L'.P_m, L'.K_m)是 L'中的最后一指针/值对
                从 L'中去除(L'.P_m, L'.K_m)
                插入(L'.P_m, L'.K_m)并通过将其它指针和值右移使之成为
                L 中的第一个指针和值
                用 L'.K_m 替换 parent(L)中的 V'
              end
            end
          else…与 then 的情况对称…
        end
      end
end procedure
```

图 11-16　从 B^+ 树中删除索引项

如果块 B 因此不到半满，它就会与相邻的块 B'重新分布记录。假定记录大小固定，那么每个块包含的记录数至少应是它所能包含的最大记录数的一半。B' 树的非叶结点接着以通常的方式更新。

当 B^+ 树用于文件组织时，空间的利用就变得非常重要，因为记录所占的空间常常比码和指针所占的空间要大得多。通过在分裂和合并时包含更多的兄弟结点，我们可以改善 B' 树的

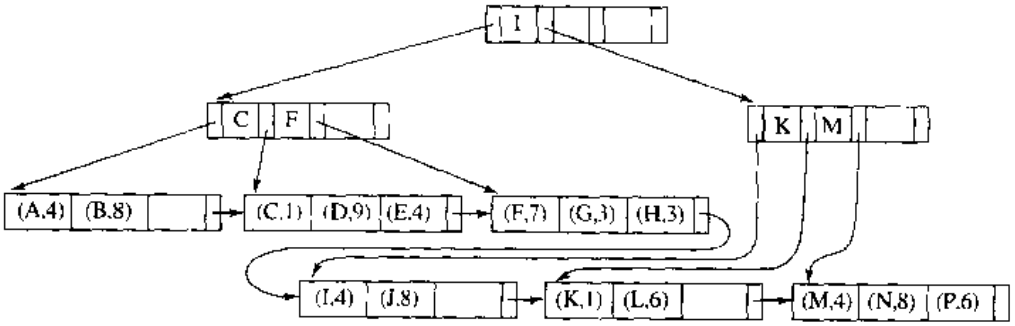


图 11-17　B⁻树文件组织

利用率。这个技术可以同时用于根结点和叶结点，具体方法如下所述。

在插入时，如果结点已满，我们就尽力把它的一些项重新分配到与它相邻的结点，以给新项腾出空间。如果因为相邻结点已满而导致这一重分配失败，我们就要分裂结点，并在由原始结点分裂所产生的两个结点和一个相邻结点之间均匀地分配所有项。由于三个结点总共包含的项比两个结点所能容纳的多 1，因此每个结点差不多都是 2/3 满。更确切地说，每个结点至少有$\lfloor 2n/3 \rfloor$个项，其中 n 是每个结点能容纳的最大项数。

在删除记录时，如果结点中的项数少于$\lfloor 2n/3 \rfloor$，我们就试图从相邻的结点借入一项。如果两个兄弟结点中都是$\lfloor 2n/3 \rfloor$个记录，那就不能借入一项，而要把这个结点及两个兄弟结点中的所有索引项均匀地分布到两个结点中，并删除第三个结点。能用这种方法是因为索引项的总数是$3\lfloor 2n/3 \rfloor - 1$，它小于 $2n$。当使用三个相邻结点重分布时，每个结点能保证有$\lfloor 3n/4 \rfloor$项。一般的，如果重分布涉及 m 个结点（$m-1$ 个兄弟结点），那么每个结点中能保证至少包含$\lfloor (m-1)n/m \rfloor$项。然而，参与重分布的兄弟结点越多，更新的代价也越高。

11.4　B 树索引文件

B 树索引和 B⁺树索引相似。两种方法的主要区别在于 B 树去除了搜索码值存储中的冗余。在图 11-11 的 B⁺树中，搜索码“Downtown”、“Mianus”、“Redwood”和“Perryride”出现了两次。每个搜索码值都出现在叶结点中，有的还重复出现在非叶结点中。

B 树允许搜索码值只出现一次。图 11-18 所示 B 树表示的搜索码值与图 11-11 相同。由于

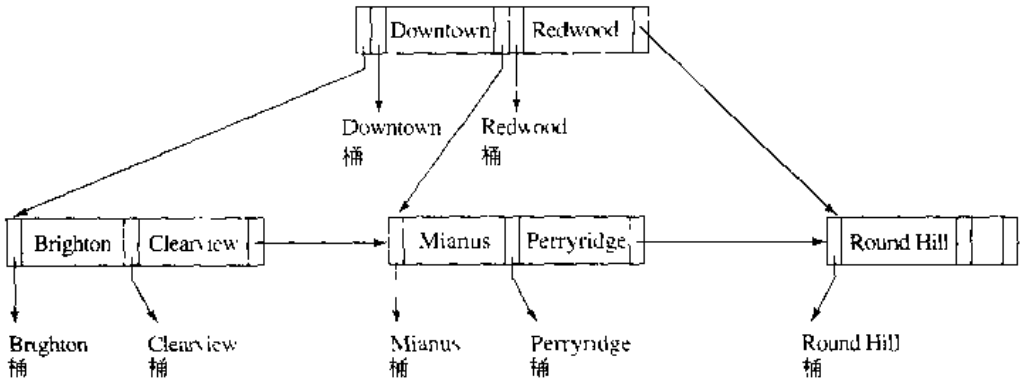
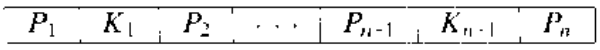


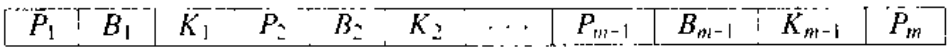
图 11-18　等价于图 11-11 中 B⁺树的 B 树

B 树中搜索码不重复，我们在用 B 树存储索引时可能比相应 B⁺树的结点要少。然而，由于非叶结点中出现的搜索码值不在 B 树中的其他任何地方出现，我们将不得不为非叶结点中的每个搜索码增加一个指针。附加的这些指针指向文件记录或相应搜索码所对应的桶。

B 树叶结点的一般形式如图 11-19a，非叶结点如图 11-19b。叶结点和 B^+ 树中的叶结点一样。在非叶结点中，指针 P_i 与 B^+ 树中使用的树指针一样，而指针 B_i 是桶或文件记录的指针。在图示的一般化的 B 树中，叶结点中有 $n-1$ 个码，而非叶结点中有 $m-1$ 个码。这个差异是因为非叶结点必须包含指针 B_i，这样就减少了结点所能容纳的搜索码个数。显然 $m<n$，且 m 和 n 的精确联系依赖于搜索码和指针的大小。



a)



b)

图 11-19　典型的 B 树结点　a)叶结点　b)非叶结点

在 B 树中进行一次查找所访问的结点数取决于搜索码的位置。对 B^+ 树的查找需要遍历从树根到叶结点的路径。与 B^+ 树相比，B 树中有时不需到达叶结点就能找到想要的值。然而，B 树中存储在叶级的码大约 n 倍于存储在非叶级的码，而 n 一般都相当大，因此较快找到特定值的好处将会比较小。而且比起 B^+ 树来，B 树的非叶结点中存储的搜索码较少，表明了 B 树扇出较小因而比相应的 B^+ 树深度大。因此，在 B 树中查询某些值时比 B^+ 树快，而另一些值比 B^+ 树慢，尽管总的来说查找时间与搜索码数目的对数成正比。

B 树中的删除更加复杂。在 B^+ 树中，要删除的项总是出现在叶结点中。在 B 树中，要删除的项却有可能出现在非叶结点中。我们必须从包含被删除项的子树中选择正确的值来作替代。具体来说，如果搜索码 K_i 被删除，指针 P_{i+1} 指向的子树中的最小搜索码必须移到以前 K_i 所占用的字段。如果叶结点因此包含的项太少的话，则需采取进一步的操作。相形之下，B 树中的插入只比 B^+ 树中的插入略复杂一些。

B 树在空间上的优势对大的索引来讲意义不大，通常还不能抵销我们已提到的那些不足。因此，B^+ 树结构上的简单性受到大多数数据库系统实现者的喜爱。B 树的插入和删除算法细节将在习题中讨论。

11.5　静态散列

顺序文件组织的一个缺点是我们必须访问索引结构来定位数据，或者使用二分法搜索，这将导致过多的 I/O 操作。基于散列的文件组织使我们能够避免访问索引结构。散列也提供了一种构造索引的方法。在接下来的几节中，我们将学习基于散列的文件组织和索引。

11.5.1　散列文件组织

在散列文件组织中，通过计算所需记录搜索码上的一个函数直接获得包含该记录的磁盘块地址。在对散列的描述中，将使用术语桶来表示能存储一条或多条记录的一个存储单位。通常一个桶就是一个磁盘块，但也可能小于或大于一个磁盘块。

形式化地，令 K 表示所有搜索码值的集合，令 B 表示所有桶地址的集合，散列函数 h 就是一个从 K 到 B 的函数。我们用 h 表示散列函数。

为了插入一个搜索码为 K_i 的记录，通过计算 $h(K_i)$ 来得到存放该记录的桶的地址。我们暂时假定桶中有容纳这条记录的空间，于是这条记录就被存储到该桶中。

为了进行一次基于搜索码的查找，我们只需计算 $h(K_i)$，然后搜索位于计算出的地址的桶。假定搜索码 K_5 和 K_7 有相同的散列值，即 $h(K_5)=h(K_7)$。如果我们执行对 K_5 的查找，则桶 $h(K_5)$ 包含搜索码是 K_5 或 K_7 的记录。因此，必须检查桶中每条记录的搜索码值，以确定该记录是否是我们要查找的记录。

删除也一样简单。如果待删除记录的搜索码值是 K_i，则我们计算 $h(K_i)$，然后在相应的桶中搜寻此记录并删除它。

1. 散列函数

最坏的可能是散列函数把所有搜索码值映射到同一桶中。这种函数并不是我们所期望的，因为所有的记录不得不存放到同一个桶里，查找一个需要的记录时将不得不遍历所有记录。理想的散列函数应把存储的码均匀地分布到所有桶中，使每个桶中含有相同数目的记录。

由于设计时无法精确知道文件中将存储哪些搜索码值，所以我们希望选择一个能按下面要求把搜索码值分配到桶中去的散列函数。

- 分布是均匀的。即每个桶从所有可能的搜索码值集合中分配到的值的数目相同。
- 分布是随机的。即一般情况下，不管搜索码值实际怎样分布，每个桶应分配到的搜索码值数目几乎相同。更确切地说，散列值不应与搜索码的任何外部可见的排序相关，例如按字母的顺序或按搜索码长度的顺序，散列函数应该表现为随机的。

如何达到以上要求，我们来为以 *branch-name* 为搜索码的 *account* 文件选择一个散列函数。我们选择的散列函数不仅要在曾使用的 *acount* 示例文件上具有良好特性，在拥有许多分支机构的大银行的实际大小的 *account* 文件上也应如此。

假设决定使用 26 个桶，并且定义的散列函数将首字母是字母表中第 i 个字母的名称映射到第 i 个桶。该散列函数虽具有简单这一优点，但是无法提供均匀分布，因为我们知道大多数分支机构名称会以"B"和"R"这样的字母开头而不用"Q"和"X"之类的字母。

现在假设希望找到搜索码 *balance* 上的一个散列函数。假设最小余额是 1，最大余额是 100 000。我们可以使用一个散列函数把这些值分成 10 个区间：1～10 000，10 001～20 000 等。对于该函数，搜索码值的分布是均匀的（因为每个桶具有相同数目的不同 balance 值），但不是随机的。然而，余额在 1～10 000 之间的记录比 90 001～100 000 之间的记录要普遍得多。因此，记录的分布是不均匀的，某些桶得到的记录比其他桶多。如果函数分布随机，甚至搜索码已有这种相关性，分布的随机性也将使所有桶拥有大致相等的记录数。

通常散列函数在搜索码字符的内部二进制表示上执行计算。这种类型的一个简单函数是先计算码中字符的二进制表示的总和，然后返回总和对桶数目的模。图 11-20 给出了该方案应用于 *account* 文件上的情况，其中使用了 10 个桶，并假设字母表中的第 i 个字母用整数 i 表示。

散列函数的设计需要认真仔细。一个糟糕的散列函数可能导致查找所花费时间与文件中搜索码数目成比例。一个设计良好的散列函数一般情况下查找所花费时间是一个（较小的）常数，而与文件中搜索码的个数无关。

2. 桶溢出控制

到目前为止，我们一直假设插入一个记录时，记录映射的桶中具有存储记录的空间。如果桶没有足够的空间，就会发生桶溢出。桶溢出的发生可能有以下几个原因：

- 桶不足。桶数目（用 n_B 表示）的选择必须使 $n_B > n_r/f_r$，其中 n_r 表示将被存储的记录总数，f_r 表示一个桶中能存放的记录数目。当然，n_B 的确定是以在选择散列函数时，记录总数已知为前提的。
- 偏斜。某些桶分配到的记录比其他桶多，所以即使其他桶仍有空间，某个桶仍可能溢出。

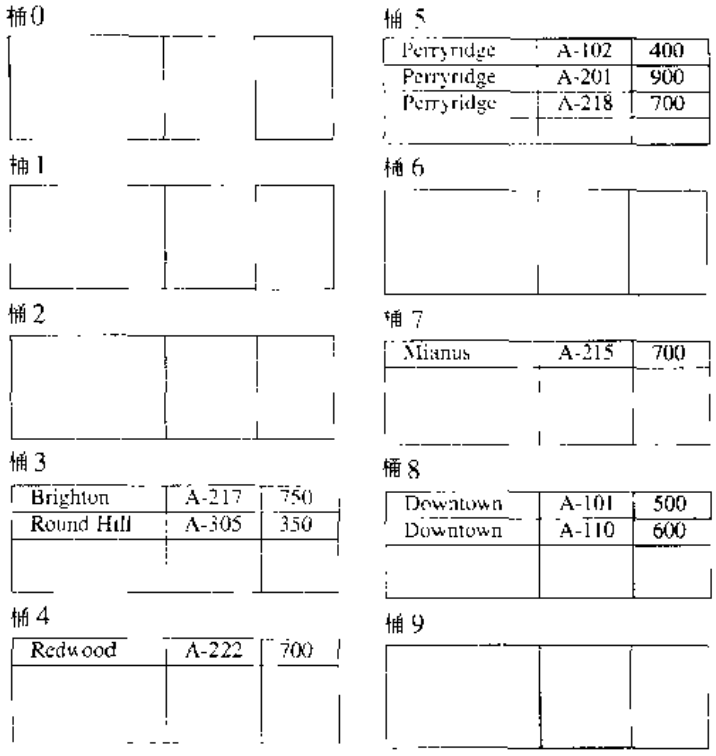


图 11-20　使用 *branch-name* 作为关键码的 *account* 文件的散列组织

偏斜发生的原因有两个：

- 多个记录可能具有相同的搜索码。
- 所选的散列函数造成搜索码的分布不均。

为了减少桶溢出的可能性，桶的数目选为 $(n_r/f_r)*(1+d)$，其中 d 是避让因子，其通常值约为 0.2。这样选择会有一些空间浪费，即桶中大约 20% 的空间是空的，但好处是减少了溢出的可能性。

尽管分配的桶比所需的桶多了一些，桶溢出还是可能发生。我们用溢出桶来处理桶溢出问题。如果一条记录必须插入桶 b，而桶 b 已满，一个溢出桶就会分配给桶 b，并将此记录插入到这个溢出桶中。如果溢出桶也满了，就会再分配一个溢出桶，如此继续下去。一个给定桶的所有溢出桶链接在一起，如图 11-21 所示。使用这种链表的溢出处理称为溢出链。

由于使用溢出链，需要将查找算法做轻微改动。和前面一样，将散列函数作用于搜索码得到桶号 b，然后检查 b 中的所有记录，看是否有匹配的搜索码。此外，如果桶 b 有溢出桶，则 b 的所有溢出桶中的记录也要被检查。

上面所描述的这种散列结构有时称为闭散列。另一种方法称为开散列，它的桶集合是固定的，也没有溢出链。当一个桶满了以后，记录将被插入到初始桶集合 B 的其他桶中。一种策略是使用下一个（按轮转顺序）有空间的桶，这个策略称为线性探索法。我们还可以使用其他策略如继续计算散列函数。开散列一般用于构造编译器和汇编器的符号表，而数据库系统中更适合采用闭散列。这样做的原因在于开散列中删除操作十分麻烦，且编译器和汇编器通常只在符号表上做查找和插入操作。可是在数据库系统中，处理删除的能力和插入一样重要，所以开散列在数据库实现中重要性不大。

前面介绍的散列方式的一个很大缺点是散列函数必须在实现系统时确定，此后若被索引的

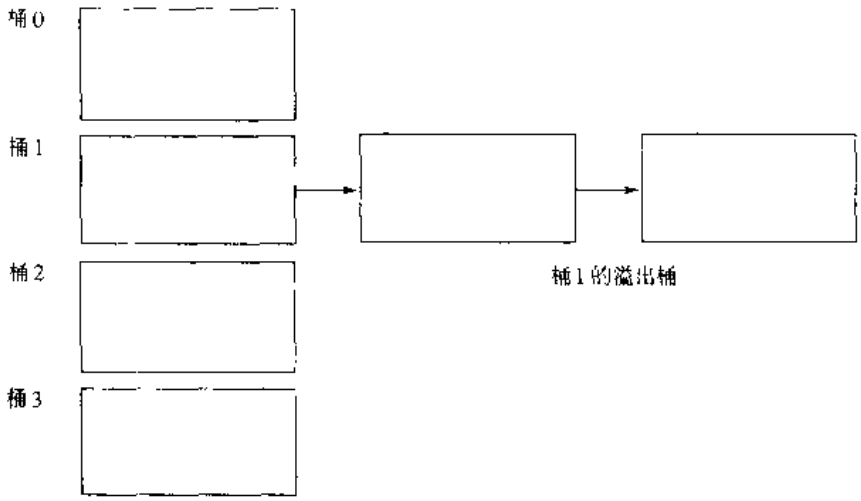


图 11-21　散列结构中的溢出链

文件变大或缩小，要想再改变它就不容易了。函数 h 将搜索码值映射到一个固定的桶地址集合 B 上，如果考虑到将来文件的增长而将 B 取得较大就会浪费空间；而如果 B 太小，一个桶中就会包含过多不同的搜索码值，也就可能发生溢出。当文件变大时，性能还会受到影响。稍后我们将在 11.6 节中介绍如何动态改变桶的数目和散列函数。

11.5.2　散列索引

散列不仅可以用于文件的组织，还可以用于索引结构的创建。*散列索引*将搜索码及相应指针组织成散列文件结构。散列索引的构造如下：将散列函数作用于搜索码以确定对应的桶，然后将此搜索码及相应指针存入此桶（或溢出桶）中。图 11-22 所示为 *account* 文件上搜索码

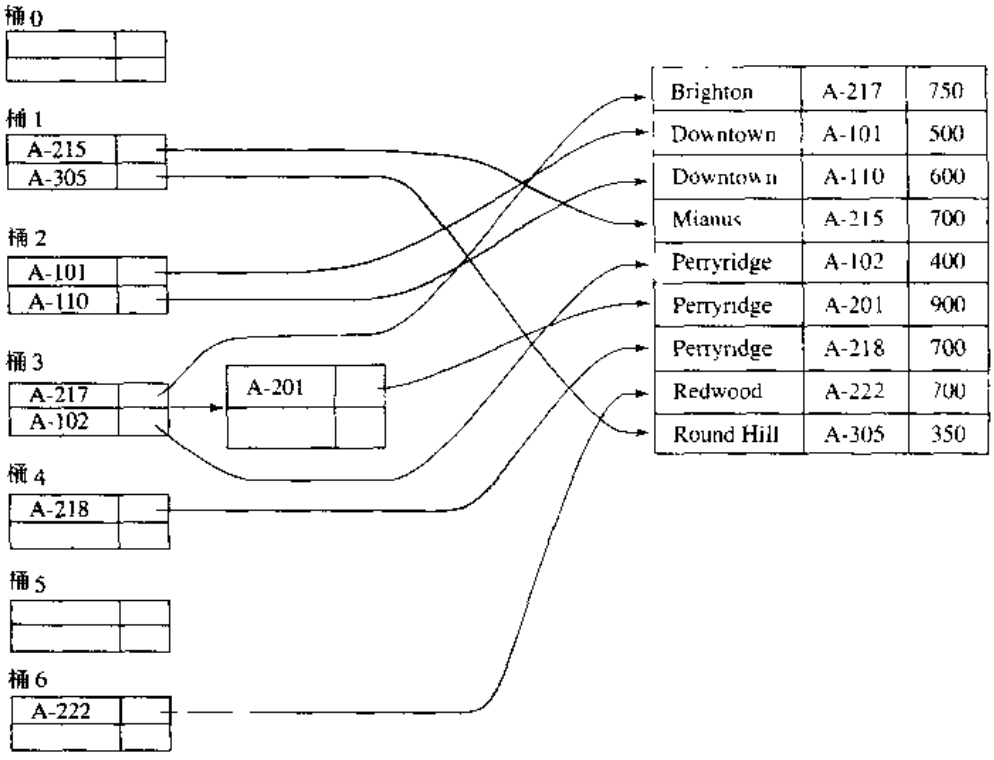


图 11-22　account 文件搜索码 account-number 上的散列索引

account-number 的一个辅助散列索引。所用散列函数为计算帐号各位数字之和后模 7，该散列索引有七个桶，每个桶的大小为 2（现实中索引的桶大小当然会大得多）。其中由三个码映射到一个桶，因此这个桶有一个溢出桶。本例中，*account-number* 是 *account* 的主码，所以每个搜索码只对应一个指针。一般情况下，每个码可能对应多个指针。

术语散列索引既用来表示散列文件结构又用来表示辅助散列索引。严格地说，散列索引应该只表示辅助索引结构。主索引结构不应该是散列索引，因为如果一个文件自身是按散列组织的，就不必在其上另外建立一个独立的索引结构。不过，既然散列文件组织能像索引那样提供对记录的直接访问，我们不妨认为以散列形式组织的文件上有一个虚拟的主散列索引。

11.6　动态散列法

如前所示，上节中的静态散列技术要求固定桶地址集合 B，这带来了一个很严重的问题。大多数数据库都会随时间而变大，如果要在这样的数据库上使用静态散列，有三种选择：

1）根据当前文件大小选择散列函数。这种选择会使性能随数据库增大而下降。

2）根据将来某个时刻文件的大小选择散列函数。尽管这样可以避免性能下降，但是初始时会造成相当大的空间浪费。

3）随着文件增大，周期性地对散列结构进行重组。这种重组涉及到一系列问题，包括新散列函数的选择、对文件中的每个记录上重新计算散列函数，以及生成新的桶分布。重组是一个复杂、耗时的操作，而且重组期间必须禁止对文件的访问。

几种动态散列技术允许散列函数动态改变，以适应数据库增大或缩小的需要。我们介绍一种称为可扩充散列的动态散列技术。文献注解中列出了其他动态散列技术的参考资料。

当数据库增大或缩小时，可扩充散列可以通过桶的分裂或合并来适应数据库大小的变化。这样一来，空间效率便得以保持。此外，由于重组每次仅作用于一个桶上，因而所带来的开销足够的低，在我们可以接受的范围内。

为了使用可扩充散列，所选择的散列函数 h 应当具有均匀性和随机性这两个良好特性。但是，此散列函数产生的值的范围也相当大，是 b 位二进制整数，而通常 b 值是 32。

用不着为每一个散列值创建一个桶。实际上，2^{32}超过 40 亿，除非是非常大的数据库，否则没有必要创建这么多桶。相反，我们是根据记录加入文件时的需要来创建桶的。开始时，散列值的 b 位不必全部使用，任一时刻使用的位数 i 满足 $0 \leqslant i \leqslant b$。这样的 i 个位作为附加的桶地址表的入口偏移量。i 的值随着数据库大小的变化而增大或减小。

图 11-23 所示为一般的可扩充散列结构。图中出现在桶地址表上方的 i 表明散列值 $h(K)$ 中有 i 位需要用来决定对应于 K 的桶。当然，这个值会随着文件变化而变化。尽管找出桶地址表中的正确表项需要 i 位，但几个连续的表项可能指向同一个桶。所有这样的表项有一个共同的散列前缀，其长度可能小于 i。因此，给每一个桶附加一个整数值，用来表明共同的散列前缀长度。在图 11-23 中，附加给桶 j 的整数是 i_j。桶地址表中指向桶 j 的表项数目为

$$2^{i-i_j}$$

为了确定含有搜索码值 K_l 的桶的位置，先找到 $h(K_l)$ 高 i 位二进制字串对应的表项，再根据表项中的指针来得到桶的位置。

要插入一个搜索码值为 K_l 的记录，我们利用相同的方法来定位桶，假定为桶 j。如果 j 桶中有剩余空间，将新记录插入该桶即可；如果桶 j 已满，必须分裂桶 j，并将该桶中现有记

录和新记录一起重新进行分配。在桶分裂之前，还必须根据散列值确定是否需要增加所使用的位数。

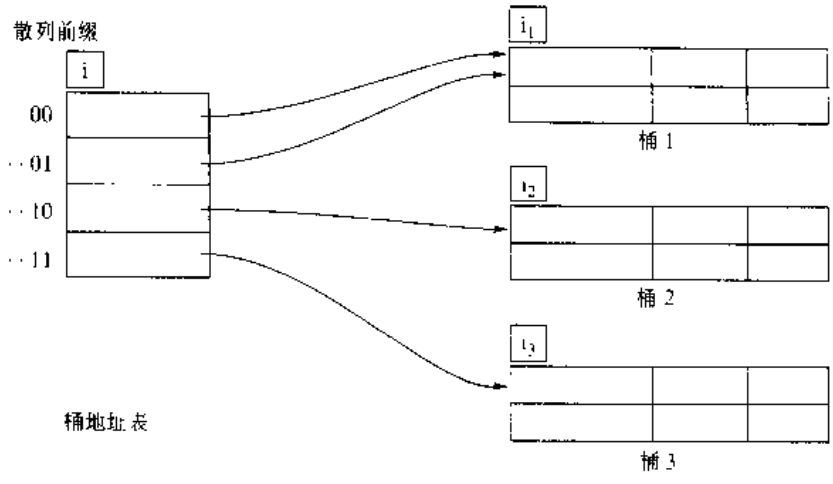


图 11-23　可扩充散列的一般结构

• 如果 $i = i_j$，那么在桶地址表中只有一个表项指向桶 j，所以需要增加桶地址表的大小，以容纳由于桶 j 分裂而产生的两个桶指针。为了达到这个目的，我们多引入散列值中的一位，将 i 的值加 1，从而使桶地址表的大小加倍。这样，原表中每个表项变成两个，两个表项都包含和原始表项一样的指针。现在桶地址表中有两个表项指向桶 j。这时，分配一个新的桶（桶 z），并让第二个表项指向此新桶。我们将 i_j 和 i_z 置为 i。接下来，桶 j 中的各条记录被重新散列，根据前 i 位（记住 i 已加 1）来确定这些记录是继续放在桶 j 中还是放到桶 z 中去。

现在再次尝试插入新记录，通常这次尝试会成功。但是，如果桶 j 中原有的所有记录和新插入的记录具有相同的散列前缀，那么这些记录会分配到同一个桶中，因此只有再次进行桶分裂。如果散列函数的选择较慎重，则一次插入导致两次或两次以上分裂是不太可能的，除非大量记录具有相同的搜索码。如果桶 j 中所有记录搜索码值相同，那么多少次分裂也不能解决问题。这种情况下采用溢出桶来存储记录，就像在静态散列中那样。

• 如果 $i > i_j$，那么在桶地址表中有多个表项指向桶 j，因此不需要扩大桶地址表就能分裂桶 j。不难发现指向桶 j 的这些表项的散列前缀的最左 i_j 位相同。我们分配一个新桶（桶 z），将 i_j 和 i_z 置为原 i_j 加 1 后得到的值。接下来需要调整桶地址表中原来指向桶 j 的表项。（注意，由于 i_j 有了新值，并非所有表项的散列前缀的最左 i_j 位都相同。）我们让这些表项的前一半仍指向桶 j，而使后一半指向新桶 z。然后就像上一种情况那样，桶 j 中的各条记录被重新散列，分配到桶 j 或新桶 z 中。

重新尝试插入新记录，失败的可能性微乎其微。如果失败，则根据情况 $i = i_j$ 还是 $i > i_j$ 继续做相应的处理。

注意，这两种情况下都只需对桶 j 中的记录重新计算散列函数。

要删除一个搜索码值为 K_l 的记录，可以按前面的查找过程找到相应的桶，不妨设为桶 j。我们不仅要把搜索码从桶中删除，还要把记录从文件中删除。如果这时桶成为空的，那么桶也需要删除。注意，此时某些桶可能被合并，桶地址表的大小也可能减半。何时及如何合并桶留作习题。

我们使用 *account* 文件的例子来说明插入操作（图 11-24）。*branch-name* 上的 32 位散列值如图 11-25 所示。假设该文件开始时是空的，如图 11-26 所示，将记录一条一条地插入其中。为了用一个很小的结构演示可扩充散列的所有特性，需要做一个不太实际的假设：一个桶只能容纳两条记录。

Brighton	A-217	750
Downtown	A-101	500
Downtown	A-110	600
Mianus	A-215	700
Perryridge	A-102	400
Perryridge	A-201	900
Perryridge	A-218	700
Redwood	A-222	700
Round Hill	A-305	350

11-24　*account* 例子文件

branch-name	*h*(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

图 11-25　*branch-name* 上的散列函数

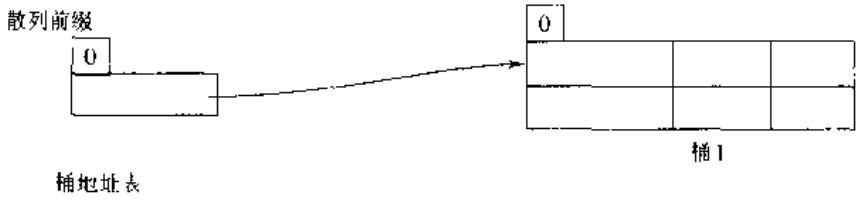


图 11-26　初始可扩充散列结构

让我们来插入记录(Brighton,A-217,750)。桶地址表包含一个指向桶的指针，并且记录已被插入。接着，再插入记录(Downtown,A-101,500)，该记录也已放在结构中的一个桶里。

当试图再插入记录（Downtown，A-110，600）时，发现桶已经满了。由于 $i = i_0$，我们需要增加所使用的散列值中的位数。现在我们使用 1 位，允许有 $2^1 = 2$ 个桶。这一位数的增加使桶地址表大小必然加倍，增加到有两个表项。这时分裂桶，在新桶中放置那些搜索码的散列值以 1 开始的记录，而其余记录留在原来的桶中。图 11-27 给出了分裂后结构的状态。

接下来插入（Mianus，A-215，700）。由于 h（Mianus）的第一位是 1，因而必须将该记录插入桶地址表中表项“1”指向的桶。这时桶满了并且 $i = i_1$，于是将使用的散列值位数增加到 2。这一位数增加使桶地址表大小必然加倍，增加到四个表项，如图 11-28 所示。由于图 11-27 中散列前缀为 0 的桶并未分裂，因而桶地址表中 00 和 01 表项都指向该桶。

对于图 11-27 中散列前缀为 1 的桶（正在被分裂的桶）中的每一条记录，检查其散列值中

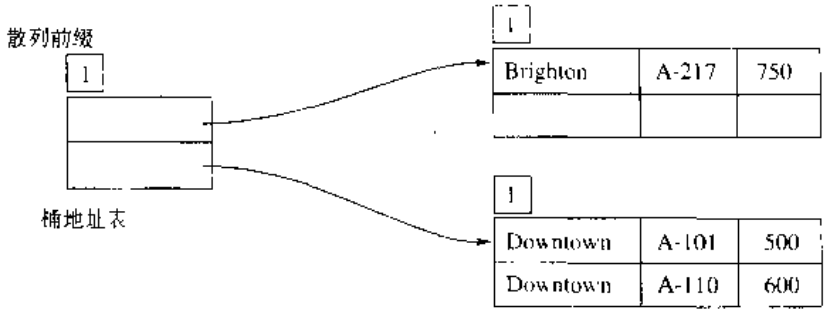


图 11-27　三次插入操作后的散列结构

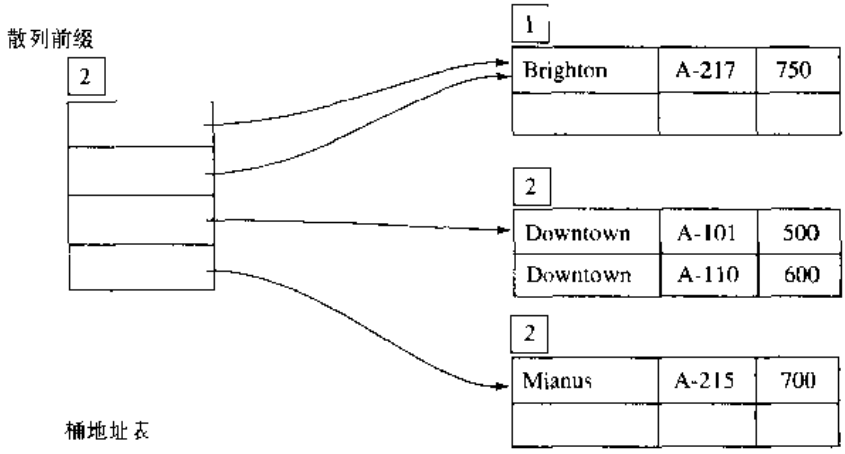


图 11-28　四次插入操作后的散列结构

的头两位，以决定在新结构的哪一个桶中存放它。

接下来插入（Perryridge，A-102，400），它和 Mianus 进入同一个桶。接着又插入（Perryridge，A-201，900），导致桶溢出，引起位数增加和桶地址表大小加倍。第三个 Perryridge 记录（Perryridge，A-218，700）的插入将引起另一次溢出。但是，这次溢出不能用增加位数来解决了，因为已有三个记录具有相同的散列值，因此使用一个溢出桶，如图 11-29 所示。

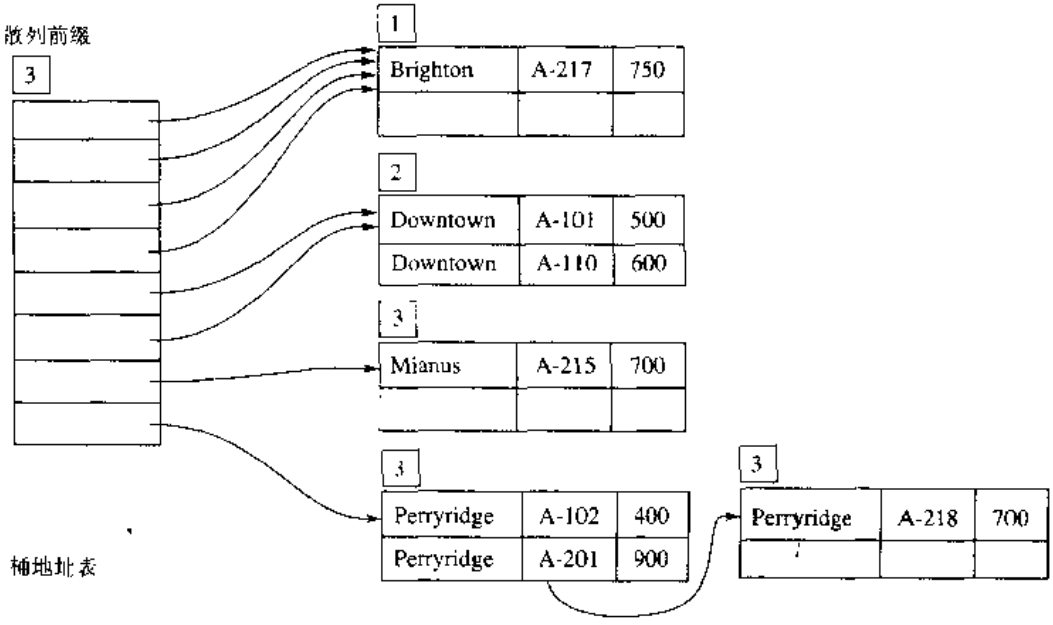


图 11-29　七次插入操作后的散列结构

继续按这种方法进行，直到插入图 11-24 中所有 *account* 记录为止。产生的结构如图11-30所示。

现在来看一看可扩充散列与讨论过的其他方案比较而言具有的优缺点。可扩充散列最主要的优点是其性能不随文件的增长而降低。此外，其空间开销是最小的。尽管桶地址表带来了额外的开销，但对于当前的前缀长度，该表为每个散列值只存放一个指针，因此该表还是较小的。可扩充散列与其他散列模式相比，空间的节约主要在于不必为将来的增长保留桶，更确切地说，可扩充散列能动态地分配桶。

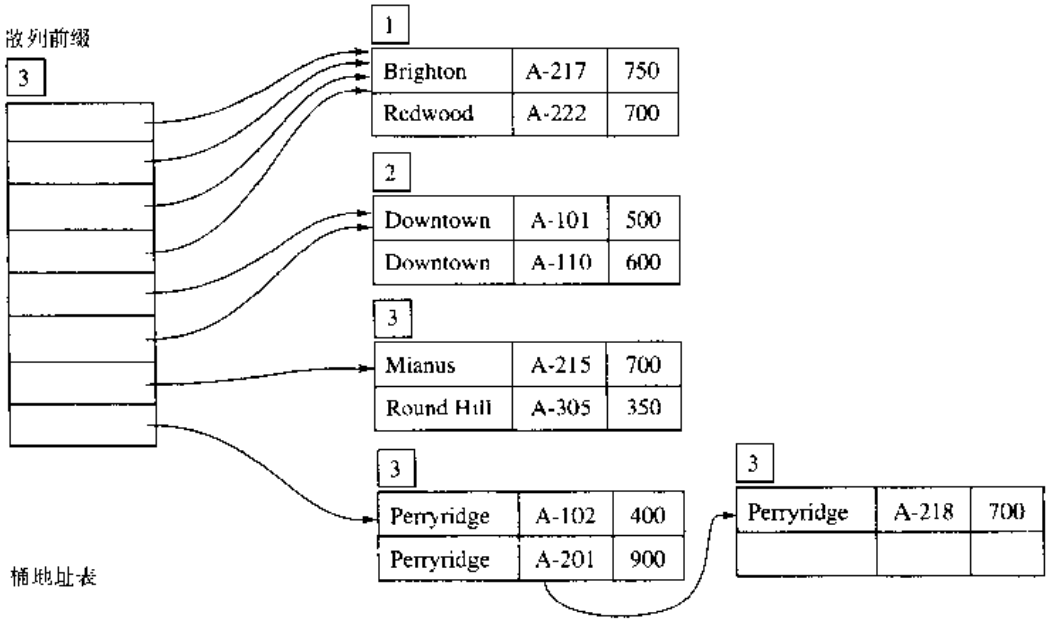


图 11-30　文件 *account* 的可扩充散列结构

可扩充散列的一个缺点在于查找时增加了一层间接，因为在访问桶之前必须先访问桶地址表。这一额外的访问对性能影响很小。尽管先前讨论的散列结构没有这一额外的间接层次，但当它们变满后就失去了性能上的小小优势。

由此看来，可扩充散列是一种非常吸引人的技术，只要我们能接受在实现它时增加的复杂性。关于可扩充散列实现的更详细资料参见文献注解。文献注解同时也提供了介绍另一种动态散列方法——线形散列的参考文献，这一方法以可能有更多的溢出桶代价为前提，避免了可扩充散列所具有的额外的间接层次。

11.7　顺序索引和散列的比较

我们已经看到了一些顺序索引方案和散列方案，可以用索引顺序组织或 B^+ 树组织将记录文件组织成顺序文件；另外，也可以用散列来组织文件；最后，还能将它们组织成堆文件，其中的记录不以任何方式排序。

各种方案在一定条件下各有其优点。数据库系统实现者可提供多种方案，而最终使用哪种方案由数据库设计者决定。但是，这种做法需要实现者写更多的代码，加大了系统的成本和系统所占用的空间。因此，大多数数据库系统只使用少数几种甚至是一种顺序索引或散列方案。

要做出明智的选择，实现者或数据库设计者必须考虑以下问题：

- 索引或散列组织的周期性重组代价可否接受？
- 插入和删除的相对频率怎样？
- 为了优化平均访问时间而增加最坏情况下的访问时间的做法是否可取？
- 用户可能提出哪些类型的查询？

我们已考虑过前三个问题，第一次在回顾具体索引技术的相对优点时，第二次在讨论散列技术时。第四个问题即预期的查询类型对选择顺序索引还是散列至关重要。

如果大多数查询形如

select A_1，A_2，…，A_n
from r
where $A_i = c$

那么，为了处理该查询，系统将在一个顺序索引或散列结构中为属性 A_i 查找值 c。对于这种形式的查询，散列的方案更可取。顺序索引查找所需时间与关系 r 中 A_i 值的个数的对数成正比；但在散列结构中，平均查找时间是一个与数据库大小无关的常数。对于这种形式的查询，索引比散列优越的唯一地方是使用索引时，最坏情况下的查找时间和关系 r 中 A_i 值的个数的对数成正比。但散列时最坏查找时间发生的可能性极小，因而在这种情况下散列更可取

顺序索引在指出了值范围的查询中比散列更可取。这样的查询形式如下：

select A_1，A_2，…，A_n
from r
where $A_i \leqslant c_2$ **and** $A_i \geqslant c_1$

换句话说，这一查询要找出 A_i 值在 c_1 和 c_2 之间的所有记录。

让我们考虑如何用顺序索引处理该查询。首先查找值 c_1。一旦找到值 c_1 的桶，可以顺着索引中的指针链读取下一个桶，如此下去直到到达 c_2。

如果不用顺序索引而用散列结构，查找 c_1 并确定其对应的桶，但一般来说，决定下一个必须检查的桶并不容易。困难产生的原因是因为好的散列函数总是将值随机地分散到各桶中，因而没有一个简单的概念能够表示“顺序排在下一个的桶”。不能将桶按 A_i 的大小顺序串在一起的原因是因为每个桶都分配了许多搜索码值。既然值由散列函数随机地分布，在一定范围内的值就很可能散布在很多甚至全部桶中。因而，为了找到所需搜索码，我们不得不读取所有的桶。

人们通常使用顺序索引，只在预先知道将来不会频繁使用范围查询的情况下才会使用散列。如果要基于码值查找而不进行范围查询，则散列组织对于查询执行过程中创建的临时文件来说特别有用。

11.8　SQL 中的索引定义

SQL 标准并未为数据库用户或管理员提供任何方法，以控制在数据库系统中创建和维护什么索引。由于索引是冗余的数据结构，因而索引对保证正确性来说并不是必需的。但是，索引对事务的高效处理十分重要，既包括更新事务又包括查询。索引对一致性约束的有效实施也很重要。例如，通过用定义的码作为搜索码创建一个索引，则强制实施了码定义（第 6 章）。

原则上数据库系统可以自动决定创建什么索引。但是，由于索引的空间代价和对更新操作的影响，对什么索引需做维护自动做出正确选择并不容易。因此，大多数 SQL 实现允许程序员通过数据定义语言的命令对索引的创建和删除进行控制。下面举例说明这些命令的语法。尽管给出的语法并不是 SQL-92 标准的一部分，但是很多数据库系统都使用并支持这样的语法。

索引由 create index 命令创建，它的形式为

create index <index-name> **on**<relation-name>（<attribute-list>）

attribute-list 是关系中构成索引搜索码的属性列表。

为了定义关系 *branch* 上以 *branch-name* 为搜索码的索引 *b-index*，我们写作

create index *b-index* **on** *branch* (*branch-name*)

如果想说明该搜索码是一个候选码，则在索引定义中加入属性 *unique*。因而，命令

create unique index *b-index* **on** *branch* (*branch-name*)

声明 *branch-name* 是 *branch* 的一个候选码。如果命令 create unique index 输入时 *branch-name* 不是候选码，就会显示错误消息，并且索引创建失败。如果该索引创建成功，接下来任何插入违反码定义的元组的企图都将失败。注意，如果数据库系统支持 SQL 标准的 unique 定义，那么这里的 unique 特性就是多余的。

为索引定义索引名是需要的，这样以后才能删除索引。drop index 命令形式如下：

drop index<index-name>

11.9　多码访问

到此为止，我们都隐含地假设执行关系上的一个查询时只使用一个索引（或散列表）。但对于某些类型的查询来说，如果存在多个索引，则使用多个索引比较有利。

假设 *account* 文件有两个索引，分别建立在 *branch-name* 和 *balance* 上。考虑如下查询："找出 Perryridge 分支机构中所有余额为 $1000 的帐户号码。"我们写作

select *loan-number*
from *account*
where *branch-name* = "Perryridge" **and** *balance* = 1000

处理这个查询可以有三种策略：

1）利用 *branch-name* 上的索引，找出 Perryridge 分支机构的所有记录。检查每条记录是否满足 *balance* = 1000。

2）利用 *balance* 上的索引，找出所有余额等于 $1000 的记录。检查每条记录是否满足 *branch name* = "Perryridge"。

3）利用 *branch-name* 上的索引找出指向属于 Perryridge 分支机构的所有记录的指针。同样，利用 *balance* 上的索引找出指向余额等于 $1000 的所有记录的指针。计算这两个指针集合的交。交集中的所有指针指向 Perryridge 分支机构中余额等于 $1000 的记录。

上面三种策略中只有第三种利用了存在的多个索引。然而，如果下面所有条件都成立，即使这种策略也可能是很糟糕的选择：

- 属于 Perryridge 分支机构的记录太多。
- 余额为 $1000 的记录太多。
- Perryridge 分支机构中余额为 $1000 的记录只有几个。

如果这些条件成立的话，为了得到一个很小的结果集，必须扫描大量指针。

这种情况下一个更有效的策略是在搜索码（*branch-name*，*balance*）上建立和使用索引，即这一搜索码由分支机构名和余额连接而成。这种索引结构和其他索引没有太大区别，唯一的不同是搜索码不是一个属性，而是一系列属性。这个搜索码可以表示成一个元组值，形式为$(a_1, \cdots, a_n)$，其对应的属性为$A_1, \cdots, A_n$。搜索码值的顺序是字典顺序。例如，若搜索码中有两个属性，$a_1 < b_1$，或$a_1 = b_1$且$a_2 < b_2$，那么$(a_1, a_2) < (b_1, b_2)$。字典顺序和把单词按字母排序类似。

使用多个属性上的顺序索引有几个缺点。例如，考虑如下查询：

select *loan-number*
from *account*
where *branch-name* < "Perryridge" **and** *balance* = 1000

可以使用建立在搜索码（*branch-name*，*balance*）上的索引来回答这个查询，如下所述：对于按字母顺序小于"Perryridge"的每个*branch-name*值，找出余额等于$1000的那些记录。然而，由于文件中记录的顺序，每条记录可能位于不同的磁盘块，因此导致大量I/O操作。

这个查询和前面的查询区别在于*branch-name*上的条件是比较条件而不是等于条件。

为了加速处理一般的有多个搜索码的查询（可能有一个或多个比较操作），可以使用若干特殊结构。我们将考虑这样两个结构：网格文件和分段散列技术。此处，还有一种称为R-树的结构也可用于这一目的，因为它相对较复杂，我们推迟到第21章再描述它。

11.9.1　网格文件

图11-31是account文件上基于搜索码*branch-name*和*balance*的网格文件的一部分。图中的二维数组称为网格数组，而一维数组称为线性标量。这个网格文件只有一个网格数组，并且对每个搜索码属性有一个线性标量。

搜索码按下面描述的方式映射到网格数组的单元中。网格数组的每个单元包含一个指向桶的指针，桶中包含若干搜索码值和记录指针。图中只显示了诸单元的桶和指针的一部分。为了节省空间，数组的多个元素可以指向相同的桶。图中用虚线围出的框指明哪些单元指向相同的桶。

假设想往网格文件索引中插入一条搜索码值为（"Brighton"，500000）的记录。为了找出这个码被映射到哪个单元，我们分别独立地计算出该单元的行和列。

首先用*branch-name*上的线性标量找出该搜索码值被映射到的单元位于哪一行。为了做到这一点，我们搜索该数组，找出比"Brighton"大的最小元素。本例中是第一个元素，所以该搜索码值被映射到标记为0的行。如果是第i个元素，该搜索码值被映射到的单元位于标记为$i-1$的行。如果该搜索码值大于等于线性标量中的所有元素，它将被映射到最后一行。接下来，以类似的方法，用*balance*上的线性标量来找出搜索码映射到的列。本例中，余额500 000被映射到列6。

因此，搜索码值（"Brighton"，500000）映射到行为0、列为6的单元。同样，（"Downtown"，60000）映射到行为1列为5的单元。两个单元都指向同一个桶（由图中虚线框可知），因此这两种情况下搜索码值和记录指针都存放在图中标记为B_j的桶中。

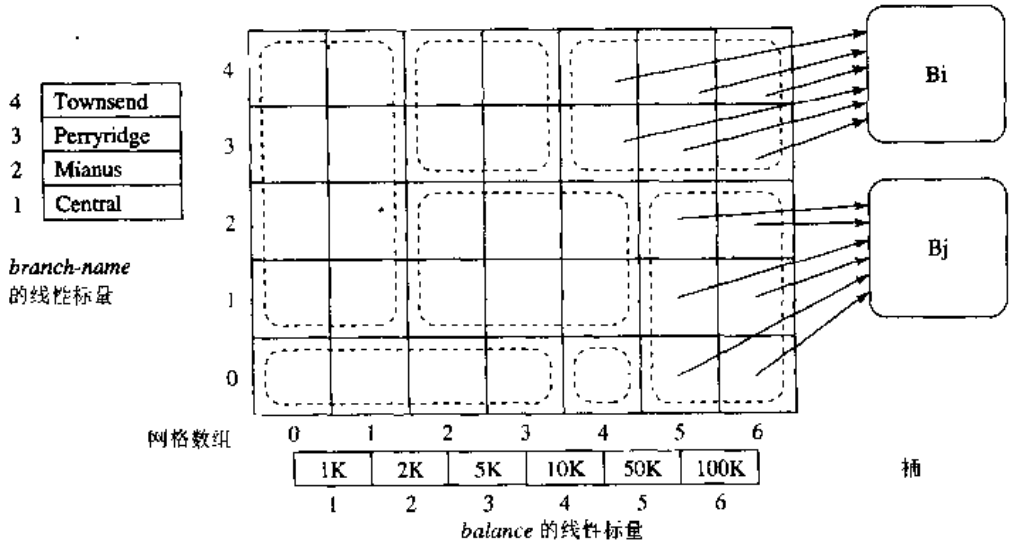


图 11-31　*account* 文件中基于码 *branch-name* 和 *balance* 的网格文件

为了处理此处查询，即条件为

$$branch\text{-}name < \text{“Perryridge”} \textbf{ and } balance = 1000$$

的查询，我们利用 *branch-name* 上的线性标量找出所含分支机构名比“Perryridge”小的所有行。本例中，这些行为 0、1 和 2。行 3 及以上行包含的分支机构名大于等于“Perryridge”。同样，只有列 1 的 *balance* 值可能为 1000，则只有列 1 满足条件。因此，列 1 上行为 0、1 和 2 的单元包含满足条件的项。

我们接着查找这三个单元指向的桶中的所有项。这里只有两个桶，因为根据图中虚线框，有两个单元指向同一个桶。桶中可能包含不符合条件的搜索码值，因此要再次检查桶中的每个搜索码值，看是否符合条件。此外，为了回答这个查询，只要检查很少几个桶。

线性标量的选择必须使记录在单元格中的分布是均匀的。如果桶 A 满了，此时又要向其中插入一项，则额外分配一个桶 B。如果不止一个单元指向 A，则这些单元指针改为一些指向 A，另一些指向 B。桶 A 中的项和新项在 A 和 B 中依据映射的单元重新分布。如果只有一个单元指向 A，B 就成为 A 的溢出桶。在这种情况下，为了提高性能，必须重新组织该网格文件，扩展网格数组和线性标量。这一过程类似于可扩充散列中桶地址表的扩展，我们将其留作习题。

把网格文件扩展为可容纳任意数目的搜索码在概念上很简单。如果想让结构能用于对 n 个码的查询，只需建立一个 n 维网格数组和 n 个线性标量。

网格结构也适用于只含一个搜索码的查询。考虑以下查询：

```
select *
from account
where branch-name = "Perryridge"
```

branch-name 上的线性标量告诉我们只有行为 3 的单元满足条件，因为对 *balance* 没有条件。检查行为 3 的单元指向的所有桶，找出属于 Perryridge 的项。这样，我们就可以使用两个搜索码上的网格文件索引来回答对其中任一个搜索码进行的查询和同时对两个搜索码进行的查询。因

此，一个单一的网格文件可以起到三个独立索引的作用。如果每个索引分别进行维护，那么三个索引合起来会占用更多的空间，对它们做更新的代价也更高。

网格文件大大减少了多码查询的处理时间。然而，它们增加了空间开销（网格目录可能变得很大），同时增加了记录插入和删除的开销。此外，很难选择适当的、保证记录均匀分布的分段范围。如果经常对文件做插入，那么必须周期地对文件进行重组，而这可能需要付出很高的代价。

11.9.2　分段散列

分段散列是对散列的扩展，它可以对多个属性进行索引。分段散列结构既允许单个属性上的查询，又允许多个属性上的查询，但它不支持范围查询。假设要为 *customer* 文件建立合适的结构，以适应该文件关于 *customer-name*、*customer-street*、*customer-city* 的查询。我们为码（*customer-street*，*customer-city*）建立散列结构。这里建立的结构和前面结构的区别在于我们增加了对散列函数 h 的限制。散列值被分割为两个部分，第一部分只依赖于 *customer-street* 的值；第二部分只依赖于 *customer-city* 的值。之所以称为分段的散列函数正是因为散列值被分割成几个部分，各部分分别依赖于码的各构成元素。

图 11-32 是一个示例的散列函数，其前三位只依赖于 *customer-street*，后三位只依赖于 *customer-city*。这样，（Main，Harrison）和（Main，Brooklyn）的散列值前三位相同，（Park，Palo Alto）和（Alma，Palo Alto）的散列值后三位相同。

可以用图 11-32 的分段散列函数回答查询“找出住在 Brooklyn 的 Main Street 的所有客户的名字”，只要计算 h（Main，Brooklyn），并访问该散列结构。这个散列结构对只基于两个搜索码之一的查询也适合。要找出住在 Brooklyn 的所有客户的记录，我们计算分段散列函数的一部分。因为只有 *customer-city* 的值，所以只能算出散列值的后三位。对于 Brooklyn，后三位是 001。我们访问散列结构，并扫描散列值后三位为 001 的桶。在图 11-32 中，我们将访问散列值为 101 001、011 001 和 001 001 的桶。

搜索码值	散列值
(Main, Harrison)	101 111
(North, Rye)	110 101
(Main, Brooklyn)	101 001
(North, Princeton)	110 000
(Park, Palo Alto)	010 010
(Putnam, Stamford)	011 001
(Nassau, Princeton)	011 000
(Spring, Brooklyn)	001 001
(Alma, Palo Alto)	110 010

图 11-32　对码（*customer-street*，*customer-city*）的分段散列函数

分段散列可扩展到任意数目的属性上。如果知道用户对不同属性的查询频率，我们就可以对分段散列进行一些改进。文献注解中提到了这些技术。分段散列的使用类似于网格文件，但有两个差别：第一，分段散列只用于码值，而不是区间。第二，分段散列中没有目录；而由散列值直接给出桶地址。

处理多码查询还有其他一些混合技术。如果系统实现者知道大部分查询限制在某种形式范围内，这样的技术可能在应用中很有用。和前面一样，涉及这些有趣技术的参考文献在文献注解中给出。

11.10　总结

许多查询只会访问文件中很少一部分记录。为了减少查找这些记录的开销，我们可以为存储数据库的文件创建索引。

索引顺序文件是数据库系统中最早的索引模式之一。这种索引方式是为既要求能顺序处理

整个文件又要求能随机处理单个记录的应用而设计的。为能按搜索码顺序快速检索记录，我们用指针把记录链接起来。为能快速随机访问，我们使用了索引结构。有两种类型的索引：稠密索引和稀疏索引。

在标准索引顺序文件中，只维持一个索引。如果在不同的搜索码上建立了不同的索引，那么有一个索引的搜索码决定了文件的排列顺序，这个索引是主索引，其余索引称为辅助索引。辅助索引可以提高对非主索引的搜索码进行查询的效率。然而，它们通常会增加数据库修改的开销。

索引顺序文件组织的主要缺陷是随着文件的增大，性能会下降。为了克服这个缺陷，可以使用 B^+ 树索引。B^+ 树索引是平衡树，即从树根到树叶的所有路径长度相等。这种模式下的查找是简单有效的，但插入和删除却较复杂。不过，在 B^+ 树中插入和删除所需操作数与数据库大小以 N 为底的对数成正比，其中 N 表示每一个非叶结点存储 N 个指针。因此，B^+ 树比其他平衡二叉树（如 AVL 树）要矮，故定位记录所需磁盘访问次数也少。可以用 B^+ 树组织记录文件的索引，也可以用它组织文件中的记录。

B 树索引和 B^+ 树索引类似。B 树的主要优点在于它去除了搜索码值存储中的冗余；主要缺陷在于整体的复杂性以及结点大小给定时减小了扇出。实际应用中，人们几乎总是更愿意使用 B^+ 树索引。

顺序文件组织需要一个索引结构来定位数据。相比之下，基于散列的文件组织允许我们通过计算所需记录搜索码值上的一个函数直接得到数据项地址。由于设计时不能精确知道哪些搜索码值将被存储在文件中，因而好的函数应该能均匀随机地把搜索码值分散到各桶中。

*静态散列*所用散列函数的桶地址集合是固定的，这样的散列函数不容易适应数据库随时间的显著增长。有几种允许修改散列函数的*动态散列*技术，*可扩充散列*是其中之一，在数据库增长或缩减时它可以通过分裂或合并桶来适应数据库大小的变化。

也可以用散列法创建辅助索引，这样的索引称为*散列索引*。为使记法简便，我们假定散列文件组织中用于散列的搜索码上有一个隐含的散列索引。

顺序索引（如 B^+ 树和散列索引）可以用于涉及多个属性的、基于相等性条件的索引。网格文件和分段散列等索引技术可以处理一般的有多个属性的索引。

习题

11.1　什么情况下应选用稠密索引而不是稀疏索引？给出你的解释。

11.2　既然索引可以加快查询处理，为什么不在多个搜索码建立索引？给出尽可能多的理由。

11.3　主索引和辅助索引有何区别？

11.4　在一个关系的不同搜索码上建立两个主索引一般来说是否可能？解释你的答案。

11.5　用下面的关键码值集合建立一个 B^+ 树

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

假设树初始为空，值按递增顺序加入。根据下列每个结点所能容纳的指针数分别构造 B^+ 树：

(a) 4

(b) 6

(c) 8

11.6　对习题 11.5 中的每一棵 B^+ 树，给出下列查询中涉及的步骤：

(a) 找出搜索码值为 11 的记录。

(b) 找出搜索码值在 7 和 17 之间（包括 7 和 17）的记录。

11.7　对练习 11.5 中的每一棵 B^+ 树，给出下列各操作后树的形状：

(a) 插入 9。

(b) 插入 10。

(c) 插入 8。

(d) 删除 23。

(e) 删除 19。

11.8　考虑 11.3.4 节修改后的 B^+ 树重分配模式。树的预期高度与 n 之间的函数关系是什么？

11.9　对练习 11.5 构造 B 树。

11.10　解释闭散列和开散列的区别。讨论在数据库应用中这两种技术相比较各有什么优点。

11.11　在散列文件组织中导致桶溢出的原因是什么？如何减少桶溢出的发生？

11.12　假设我们在一个文件上使用可扩充散列，该文件中所含记录的搜索码值如下：

2、3、5、7、11、17、19、23、29、31

假设散列函数为 $h(x) = x \bmod 8$，且每个桶最多容纳 3 条记录。给出此文件的可扩充散列结构。

11.13　在进行下列各步以后，习题 11.12 中的可扩充散列结构如何变化？

(a) 删除 11。

(b) 删除 31。

(c) 插入 1。

(d) 插入 15。

11.14　为什么对于一个可能进行范围查询的搜索码而言散列结构不是最佳选择？

11.15　考虑这样一个网格文件，由于一些性能因素，我们不希望在其中使用溢出桶。当需要溢出桶时，我们采用方法是对网格文件进行重组。给出用于这种重组的一个算法。

文献注解

索引和散列中所用基本数据结构的有关讨论可在 Cormen 等［1990］中找到。索引最先是由 Bayer［1972］以及 Bayer 和 McCreight［1972］引入的。Comer［1979］、Bayer 和 Unterauer［1977］以及 Knuth［1973］对 B^+ 树进行了讨论。Bayer 和 Schkolnick［1977］以及 Shasha 和 Goodman［1988］分析了对几个进程并发更新的文件上的 B 树索引进行管理的问题。对 B 树及类似数据结构中并发控制的其他讨论包括 Lehman 和 Yao［1981］、Kung 和 Lehman［1980］，以及 Ford 和 Calhoun［1984］。Gray 和 Reuter［1993］为 B^+ 树实现问题提供了很好的描述。

几种树和类树搜索结构已被提出。Tries 树结构基于码中的“数字”（例如字典中的 thumb 索引对每个字母有一个索引项）。这种树不像 B^+ 树那样是平衡的。Tries 在 Ramesh 等［1989］、Orestein［1982］、Litwin［1981］以及 Fredkin［1960］中进行了讨论。相关技术包括 Lomet［1981］中的数字 B^+ 树。

Knuth［1973］对大量不同的散列技术做了分析。动态散列模式有几种，可扩充散列由 Fagin 等［1979］引入，线性散列由 Litwin［1978，1980］引入，Larson［1982］对其性能进行了

分析。Ellis [1987] 讨论了使用线性散列时的并发。线性散列的一种变体在 Larson [1988] 中给出。此外 Larson [1978] 提出了另一种称为动态散列的模式。Ramakrishna 和 Larson [1989] 给出的另一种模式以少数数据库更新开销很大为代价，使检索只需访问一次磁盘。

网格文件结构见 Nievergelt 等 [1984] 以及 Hinrichs [1985]。分段散列函数已经在一些散列结构中得到了应用。部分匹配检索使用分段散列函数来限制处理多码查询时所需搜索的桶的数目，见 Rivest [1976]、Burkhard [1976, 1979] 以及 Ullman [1988]。King 等 [1983] 在为包括子查询的多码查询所设计的一个索引结构中混合使用了可扩展散列和分段散列函数（正如基于关键词的检索那样）。分段散列函数的最佳选择在 Bolour [1979] 以及 Aho 和 Ullman [1979] 中进行了讨论。

第12章 查询处理

查询处理是指从数据库中提取数据的一系列活动。这一系列活动包括：将用高层数据库语言表示的查询语句翻译为能在文件系统这一物理层次上实现的表达式、为优化查询进行各种转换，以及查询的实际执行。

查询处理的代价通常取决于磁盘访问，磁盘访问比内存访问速度要慢。对于一个给定的查询，通常会有许多可能的处理策略，复杂查询更是如此。就所需磁盘访问次数而言，策略好坏差别甚大，有时甚至相差几个数量级。因此，系统多花点时间在选择一个较好的查询处理策略上是值得的，即便该查询语句只执行一次。

12.1 概述

查询处理步骤如图 12-1 所示。基本步骤包括：

1）语法分析与翻译。

2）优化。

3）执行。

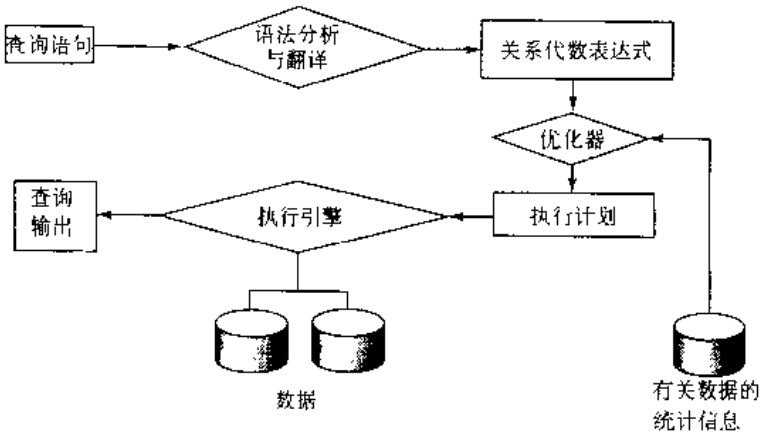


图 12-1 查询处理的步骤

查询处理开始之前，系统必须将查询语句翻译成可使用的形式。适合人们使用的查询语言如 SQL 并不适合用来在系统内部表示查询。一种更有用的内部表示建立在扩展关系代数的基础上。

因此，查询处理中系统首先必须把查询语句翻译成系统的内部表示形式。翻译过程类似于编译器语法分析器的工作。语法分析器在将查询翻译成系统的内部表示形式时，对用户的查询进行语法检查，如确保在查询中出现的关系名是数据库中已存在的关系名、等等。构造该查询语句的语法分析树表示，并将其翻译成关系代数表达式。如果查询是用视图表示的，翻译阶段还要用定义该视图的关系代数表达式来替换所有对该视图的引用[○]。语法分析在大多数有关编译的教科书中进行了讨论（参见本章文献注解），不属于本书的讨论范围。

○ 对于实体化视图，定义视图的表达式已经执行并存储了结果。故此，可以使用存储的关系，而无需用定义视图的表达式转换视图。递归视图的处理与此不同，它是通过不动点过程来进行，如 5.3.6 节所述。

在网状模型与层次模型中（见附录A和B），查询优化的大部分工作留给了编写应用程序的程序员。原因在于，这两种模型的数据操纵语言通常嵌入到宿主语言中，并且在不知道整个应用的情形下要把网状模型与层次模型的查询转换成等价的查询是很困难的。而关系查询语言或者是声明性的或者是代数的。声明性语言使用户可以说明查询要产生什么，而不必告诉系统如何产生；代数语言则可对用户的查询进行代数转换。基于查询说明，优化器可以比较容易地为查询产生多个等价的查询计划，并选出其中代价最小的一个。

本章针对关系模型进行论述。我们将看到关系模型的代数基础给查询优化带来了诸多便利。给定一个查询，通常有多种方法可以计算其结果。例如，我们已经看到，一个查询在SQL中可用多种方式表达，而每个SQL语句又可以按多种方法中的一个翻译成关系代数表达式。此外，查询语句的关系代数表达式表示只是部分地描述了如何计算一个查询。通常计算关系代数表达式有多种方法。作为例子，考虑下面这个查询语句：

select *balance*
from *account*
where *balance* <2500

该查询语句可翻译成下面两个关系代数表达式中的任何一个：

- $\sigma_{balance<2500}(\Pi_{balance}(account))$。
- $\Pi_{balance}(\sigma_{balance<2500}(account))$。

进一步，我们可用多个算法之一来执行关系代数运算。例如，实施前面的选择可以通过搜索*account*的每个元组找出满足*balance*<2500条件的元组。如果存在属性*balance*上的B^+树索引，还可利用它来定位元组。

要全面说明如何计算一个查询，不仅要提供关系代数表达式，还要对该表达式加上注释，这些注释是用于说明如何实施每个操作的命令。注释可以说明某个具体运算所采用的算法或将要使用的一个或多个特定的索引。加了有关“如何执行”的注释的关系代数运算称为执行原语。几个原语可以聚集起来形成一个流水线，流水线中的每个原语甚至在作为其输入的元组正在被另一原语产生时就可以对输入元组进行处理。用于计算一个查询的原语序列称为查询执行计划或查询计算计划。图12-2表示了所举查询例子的一个计算计划。图中为选择运算指定了一个具体的索引（图中用“索引1”表示）。查询执行引擎接受一个查询执行计划，执行该计划并把结果返回给查询。

给定查询的不同执行计划可能会有不同的代价。不能指望用户写出对应于最高效计算计划的查询语句。相反，构造具有最小查询执行代价的查询执行计划应当是系统的职责。如前所述，磁盘访问数常常是衡量性能最恰当的标准。

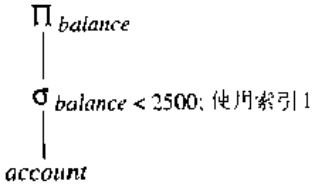


图12-2 一个查询执行计划

查询优化是为查询选择最有效的查询执行计划的过程。查询优化的一方面是在关系代数级进行优化，要做的是力图找出与给定表达式等价、但执行效率更高的一个表达式。我们将在12.9节更深入地讨论等价表达式。查询优化的其他方面涉及查询语句处理的详细策略的选择，例如选择执行运算所采用的具体算法以及将使用的特定索引等等。

为了在诸多查询执行计划中做出选择，优化器必须估计每个查询执行计划的代价。在没有真正执行查询计划之前，准确计算出执行该查询计划的代价通常是不可能的。因此，优化器要

利用各关系的统计信息，如关系大小、索引深度等，来对计划做出最佳估计。

考虑前面的例子，它将一个选择运算作用到 *account* 关系上。优化器对不同查询执行计划的代价做出估计。若 *account* 的属性 *balance* 上建立了索引，图 12-2 中给出的执行计划——其中选择运算使用了该索引——其代价可能是最小的，从而该执行计划被选中。

一旦选定了查询计划，查询就用该计划来计算，查询的结果被输出。

以上对查询语句处理步骤的描述是具有代表性的，但并非所有数据库系统都完全遵从这些步骤。例如，许多数据库系统就不用关系代数表达式来表示查询，而是采用基于给定 SQL 查询结构的带注释的语法分析树来表示。然而，此处所讲述的概念构成了数据库中查询处理的基础。

下节将构造一个代价模型，利用该模型我们可以对各种运算的代价做出估计。使用这个代价度量，我们将在 12.3～12.7 节讲述单个操作的最优计算。12.8 节将验证把多个操作组合成流水线操作时效率的提高。利用这些手段，我们可以最佳地确定给定关系代数表达式的近似代价。最后，12.9 及 12.10 节将讲述关系代数表达式间的等价性。利用这些等价性，我们可以将由用户查询构造的关系代数表达式替换成与之等价的但执行代价估计较低的表达式。

12.2　用于估计代价的目录信息

为查询执行所选策略取决于该策略的代价估计。查询优化器利用存储在 DBMS 目录中的统计信息来估计计划代价。关系的相关目录信息包括：

- n_r ——关系 r 中的元组数目。
- b_r ——含有关系 r 的元组的块数目。
- s_r ——关系 r 中一个元组的大小。
- fr ——关系 r 的块因子，即一个块中能存放的关系 r 的元组数。
- $V(A, r)$ ——关系 r 中属性 A 所具有的不同值的数目。该数目与 $\Pi_A(r)$ 的大小相同。若 A 为关系 r 的码，$V(A,r)$即为 n_r。
- $SC(A, r)$——关系 r 的属性 A 的选择基数。给定关系 r 及其属性 A，假定至少有一条记录满足等值条件，那么$SC(A,r)$表示在属性 A 上满足某个等值条件的平均记录数。例如，若 A 为 r 的码属性，则$SC(A,r)=1$；若 A 为非码属性，并假定$V(A,r)$个不同值在多个元组中均匀分布，则$SC(A,r)=(n_r/V(A,r))$。

根据需要，最后两个统计值 $V(A, r)$ 与 $SC(A, r)$ 可以针对属性组而不用仅仅针对单个属性。因此，给定属性组A，$V(A,r)$就是 $\Pi_A(r)$的大小。

若假定关系 r 的元组物理上存于同一文件中，则下面的等式成立：

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

除了使用关系的目录信息外，下面这些关于索引的目录信息也会用到：

- f_i ——树形结构（如 B^+ 树）索引 i 的内部结点的平均扇出。
- HT_i——索引 i 的层数，即索引 i 的高度。对于关系 r 的属性 A 上所建的平衡树索引（如 B^+ 树），$HT_i=\lceil \log_{f_i}(V(A,r)) \rceil$；对于散列索引，$HT_i=1$。
- LB_i——索引 i 中最低层索引块数目，即索引叶层的块数。对于散列索引，LB_i 就是索引中的块数，而不是目录块数。

如下所述，我们利用这些统计变量来估计结果集的大小、各种运算与算法的代价。我们把算法 A 的代价估计记为 E_A。

如果想要维护准确的统计值，那么每当关系修改时，相应的统计值也必须进行修改。这种更新会带来很大的代价，因此大部分系统并不是每次修改时都对统计值进行更新，而是留到系统负载较轻时进行。所以，用于选择查询处理计划的统计数据不一定完全准确。然而，只要在两次统计数据更新之间没有太多的更新发生，这些统计数据的准确度足以用来对不同计划的相对代价做出较好估计。

这里提到的统计信息是经过简化的，实际系统的优化器通常包含更多的统计信息，以提高计划代价估计的准确性。

12.3 查询代价的度量

查询处理的代价可以通过该查询对各种资源的使用情况进行衡量，资源包括磁盘存取、执行一个查询所用CPU时间、在并行/分布式数据库系统中的通信开销（第17、18章中讨论）。假定计算机中没有其他任务，查询执行计划的响应时间（即执行该计划所需时间）则包含以上所有类型的代价，同时也可作为查询执行计划代价的一个较好的度量。

然而在大型数据库系统中，磁盘访问（用从磁盘中传送的块数来度量）通常是最主要的代价，因为磁盘存取比内存操作的速度慢得多。此外，CPU速度的提升比磁盘速度提升要快。这样，花费在磁盘存取上的时间仍旧是整个查询执行时间中的主要部分。再者与磁盘存取代价估计相比，CPU时间的估计相对较难。因此，磁盘存取代价被认为是查询执行计划代价的一个合理度量。

为简化磁盘存取代价的计算，假定所有块传送的代价相同。该假定忽略由旋转延迟（等待所需数据旋转到读写头下的时间延迟）与搜索时间（将磁头移到所期望的磁道或柱面的时间）所引起的差异。虽然这些因素很重要，但在一个共享系统中它们是很难估计的。因此，我们简单地用磁盘块传送数作为实际代价的一个度量。

我们还忽略了将操作的最终结果写回磁盘的代价。不管采用了什么查询执行计划，这个代价是不变的，因此，忽略它并不影响查询计划的选择。

我们所考虑的算法的代价很大程度上取决于主存中缓冲区的大小。最好的情形是所有的数据都能读入到缓冲区中，外存不必再访问。最坏的情形是缓冲区只能容纳数目不多的数据块——大约每个关系一块。当要进行代价估计时，通常假定是最坏的情形。

12.4 选择运算

查询处理中，文件扫描是存取数据最低级的操作。文件扫描是用于定位、检索满足选择条件的记录的搜索算法。在关系系统中，若关系保存在单个专用的文件中，文件扫描允许读取整个关系。

12.4.1 基本算法

假定关系保存在单个文件中，现考虑对该关系的选择运算。用于实现选择运算的两个扫描算法如下：

• $A1$（线性搜索）。在线性搜索中，每个文件块均被扫描，且所有记录[○]都被测试，看它们是否满足选择条件。由于所有块都要读取，所以 $E_{A1}=b_r$（前面讲过，E_{A1}表示算法 $A1$ 的代价估计）。对于在码属性上的选择运算，假定当找到指定的记录时需要扫描一半的数据块，此

○ 我们假定文件包含了一个关系，每个记录正好与一个元组相对应。

时扫描就可以中止。算法 A1 的代价估计值为 $E_{A1}=(b_r/2)$。虽然多数情况下线性搜索算法效率低，但它可用于任何文件，且不管该文件是否有索引。

• A2（二分法搜索）。若文件按某一属性排序，且选择条件是该属性上的等值比较，则可用二分法搜索来定位符合选择条件的记录。二分法搜索是针对文件的数据块进行的，扫描的文件块数目估计如下：

$$E_{A2}=\lceil \log_2(b_r)\rceil+\left\lceil \frac{SC(A,r)}{f_r}\right\rceil-1$$

第一项 $\lceil \log_2(b_r)\rceil$ 表示用二分法搜索算法对数据块进行查找获得的第一个元组的代价。满足选择条件的记录数是 $SC(A,r)$，这些记录将占 $\lceil SC(A,r)/fr\rceil$ 块⊖，其中有一块已被检索到，从而有前面的估计。

若等值条件是对码属性进行比较，则有 $SC(A,r)=1$，因此以上估计减少到 $E_{A2}=\lceil \log_2(b_r)\rceil$。

二分法搜索算法的代价估计基于关系的所有数据块在磁盘上连续存放这一假设。如果没有这一假设，定位文件中一个数据块的物理地址需要查找文件存取结构（该结构可能在磁盘上），这项代价必须加到估计值中去。此外，代价估计还依赖于选择运算结果集的大小。

如果假定属性值均匀分布（即值的出现是等概率的），a 是关系 r 的某条记录中属性 A 的值，则查询 $\sigma_{A=a}(r)$ 估计会得到

$$SC(A,r)=\frac{n_r}{V(A,r)}$$

个元组。其中选择条件中值 a 出现于关系 r 的某个元组中，这一假设一般是成立的，代价估计中常做此假设。然而，假定每个值等概率出现则却是不现实的。如 *account* 关系中的 *branch-name* 属性就不满足等概率条件。每个帐户在 *account* 关系中有一个元组，大的分支机构很可能会比小的分支机构有更多的帐户，因此某些 *branch-name* 出现的概率相对较大。尽管均匀分布假设通常不成立，但在许多情况下是对现实的合理近似，有助于简化表示。

举一个代价估计的例子，假定有 *account* 关系的如下统计信息：

- $f_{account}=20$，即每个数据块中存放 20 个 *account* 元组。
- $V(branch\text{-}name, account)=50$，即有 50 个不同的分支机构。
- $V(balance, account)=500$，即有 500 个不同的 *balance* 值。
- $n_{account}=10000$，即 *account* 关系有 10 000 个元组。

考虑查询

$$\sigma_{branch\text{-}name=\text{“Perryridge”}}(account)$$

由于关系中有 10 000 个元组，每块可容纳 20 个元组，因此所需块数 $b_{account}=500$。对 *account* 的一次简单文件扫描将有 500 次块存取。

假设 *account* 按 *branch-name* 排序，由于 $V(branch\text{-}name, account)=50$，预计有 10000/50=200 个 *account* 元组与“Perryridge”分支机构有关，这些元组将占 200/20=10 块。二分法搜索找到第一条记录的时间将需 $\lceil \log_2(500)\rceil=9$ 次块存取数，因此，总的代价估计为：9+10−1=18 次块存取。

⊖ 该公式基于这样一个（通常有效的）假定：具有给定值的记录集合从块的边界开始。若该假定不成立，最坏情况下，要增加一次块存取。

12.4.2 利用索引的选择

索引结构称为存取路径，因为索引结构提供了定位和存取数据的一条路径。第11章曾指出以与物理顺序紧密相关的次序去读取文件记录效率较高，同时我们也讲过，主索引使得文件记录可以按与其在文件中的物理顺序相应的次序进行读取。非主索引称为辅助索引。

使用索引的搜索算法称为索引扫描。有序索引，如 B^+ 树，还允许按次序存取元组，这对于实现范围查询是很有帮助的。虽然使用索引可以快速、直接、有序地存取元组，但索引的使用带来了存取那些包含索引的数据块的代价。在估计使用索引的查询策略的代价时，应把索引块的存取代价考虑进去。我们用选择谓词来指导在查询处理中选择使用哪个索引。

- *A3*（主索引，码属性等值比较）。对于具有主索引的码属性的等值比较，可以使用索引检索到满足对应等值条件的一条记录。要检索到这条记录，所需的块存取数比索引层数（HT_i）多一，代价为 $E_{A3} = HT_i + 1$。
- *A4*（主索引，非码属性等值比较）。当选择条件是关于非码属性 A 的等值比较时，可以利用主索引检索到多条记录。$SC(A, r)$ 条记录满足等值条件，则需存取「$SC(A, r) / f_r$」个文件块。因此，

$$E_{A4} = HT_i + \left\lceil \frac{SC(A, r)}{f_r} \right\rceil$$

- *A5*（辅助索引，等值比较）。使用等值条件的选择可用辅助索引。若索引字段是码属性，该策略可检索到满足等值条件的一条记录；若索引字段是非码属性，则可检索到多条记录。对于属性 A 上的一个等值条件，有 $SC(A, r)$ 条记录满足等值条件。在已知所用索引是辅助索引的情形下，假设发生最坏的情形：每条满足等值条件的记录存在于不同的块上，于是得到 $E_{A5} = HT_i + SC(A, r)$；或当 A 是码索引属性时，$E_{A5} = HT_i + 1$。

假设 *account* 的统计信息同前例，并且假设 *account* 有如下索引：

- *branch-name* 属性上建立了 B^+ 树主索引。
- *balance* 属性上建立了 B^+ 树辅助索引[○]。

如前所述，为了简单起见，假设属性值均匀分布。

考虑查询

$$\sigma_{branch\text{-}name = \text{"Perryridge"}}(account)$$

由于 $V(branch\text{-}name, account) = 50$，我们估计将得到 *account* 关系中“Perryridge”分支机构的 10000/50 = 200 个元组。假设我们使用 *branch-name* 属性上的索引。由于该索引是聚集索引，因而读取 *account* 元组需要 200/20 = 10 次块存取。另外，还必须读取多个索引块。假定 B^+ 树索引每个结点存储 20 个指针。由于有 50 个不同分支机构名，B^+ 树索引必有 3～5 个叶结点。对于这样的叶结点数，整棵树的高度为 2，因此必须读 2 个索引块。从而，上述策略总共需要 12 次块读取。

12.4.3 涉及比较的选择

考虑形如 $\sigma_{A \geqslant v}(r)$ 的选择。在没有关于比较的更多信息的情况下，假定大约有一半记录满足比较条件，故所得结果有 $n_r/2$ 个元组。

○ 实际中，在 *balance* 属性上维护索引的可能性不大，因为 *balance* 属性值频繁变更。尽管如此，这个例子在这里可以用来说明不同存取计划的相对代价。

代价估计时，若比较表达式中 v 实际值已知，则可以做更准确的估计。属性的最小、最大值（min（A，r）、max（A，r））可以存储在目录中。假设属性值均匀分布，那么可以估计满足条件 $A \leqslant v$ 的记录数在 $v < \min(A, r)$ 时是 0，在 $v \geqslant \max(A, r)$ 时是 n_r，其他情况是 $n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$。

可以用线性搜索或二分法搜索来实现涉及比较的选择运算，或按以下方法之一使用索引：

• *A6*（主索引，比较）。当选择条件是比较时，可使用有序主索引（如 B^+ 树主索引）。形如 $A > v$ 或 $A \geqslant v$ 的比较条件，可按以下方式使用主索引来指导元组的检索：对于 $A \geqslant v$，在索引中查找 v 值以检索出满足条件 $A = v$ 的首条记录，从该元组开始一直扫描到文件尾可得到满足该条件的所有记录；对于 $A > v$，文件扫描从第一条满足 $A > v$ 的记录处开始。

对于形如 $A < v$ 或 $A \leqslant v$ 的比较式，没有必要查找索引。对于 $A < v$，只是简单地从文件头开始进行文件扫描，直到遇上（但不包含）首条满足 $A = v$ 的元组为止。$A \leqslant v$ 的情形类似，不过扫描是直到遇上（但不包含）首条满足 $A > v$ 的元组为止。这两种情况下，索引都没有什么用处。

假定大约一半记录满足比较条件之一。在这个假设下，使用索引进行检索的代价为

$$E_{A6} = HT_i + \frac{b_r}{2}$$

代价估计时，若比较表达式中 v 实际值已知，则可以做更准确的估计。令估计的满足条件的值个数（如前所述）为 c，则

$$E_{A6} = HT_i + \left\lceil \frac{c}{f_r} \right\rceil$$

• *A7*（辅助索引，比较）。可以使用有序辅助索引指导涉及 $<$、$\leqslant$、$\geqslant$、$>$ 的比较条件的检索。最低层索引块的扫描，对于 $<$ 及 $\leqslant$ 情形，是从最小值开始直到 v 为止；对于 $>$ 及 $\geqslant$ 情形，是从 v 开始直到最大值为止。对于这些比较，若假定至少有一半记录满足条件，则要存取一半的最底层索引块，并且通过索引存取一半数目的文件记录。此外，必须经过从索引根块到使用的第一个索引叶块的一条路径。因此代价估计如下：

$$E_{A7} = HT_i + \frac{LB_i}{2} + \frac{n_r}{2}$$

正如在聚集索引上非等值比较时所做的一样，如果在代价估计时已知比较时实际所用的值，则可以做更准确的估计。

在 Tandem 的 Non-Stop SQL 系统中，B^+ 树既用于主数据存储又用作辅助存取路径。主索引是聚集索引，而辅助索引则不然。辅助索引中包含的不是指向记录的物理位置的指针，而是搜索主 B^+ 树的码。如果使用了这样的索引，则上述用于辅助索引的代价公式要稍做修改。

尽管前面的算法表明索引在处理带比较的选择时很有用，但并不总是如此。作为例子，考虑查询

$$\sigma_{balance < 1200}(account)$$

假设关系所用统计信息与前面一样。如果不知道 *account* 关系中的最大、最小余额，那么我们假定有一半记录满足选择条件。

如果使用 *balance* 上的索引，则可以如下估计块存取数：假定 *balance* 的 B^+ 树索引结点可存放 20 个指针。由于有 500 个不同的 *balance* 值，并且每个叶结点至少是半满的，故该树约有 25～50 个叶结点。因此，*balance* 上的索引深度为 3，读取第一个索引块需 3 次块存取。最坏的

情形下有 50 个叶结点，需存取其中的一半结点，即还要 24 次块读取。最后，对于在索引中定位的每个元组，还得在关系中检索该元组。我们估计 10 000 个元组当中有一半即 5000 个满足条件。由于是非聚集索引，最坏的情况下存取每条记录要单独存取一块，这样，共需 5027 次块存取。

相反地，一次简单的文件扫描只需 10000/20 = 500 次块存取。这种情况下使用索引不合算，而应当使用文件扫描。

12.4.4 复杂选择的实现

到此为止，我们只考虑了形如 $A\ op\ B$ 的最简单的选择条件，其中 op 是等值或比较运算。现在我们来看看更复杂的选择谓词。

• 合取。合取选择形式如下

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$$

对该选择的结果集大小可做如下估计：对于每个 θ_i，我们对选择 $\sigma_{\theta_i}(r)$ 的结果集大小进行估计，方法同前，并将其大小记为 s_i。因此，关系中的一个元组满足选择条件 θ_i 的概率为 s_i/nr。

上述概率称为选择 $\sigma_{\theta_i}(r)$ 的中选率。假设各条件相互独立，则某条记录满足全部条件的概率是全体概率的乘积。所以整个选择的结果集大小估计为：

$$n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$

• 析取。析取选择形式如下

$$\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n}(r)$$

满足单个简单条件 θ_i 的所有记录的并也满足析取条件。

与前面一样，s_i/n_r 表示某元组满足条件 θ_i 的概率。元组满足整个析取式的概率就是 1 减去条件全不满足时的概率，即：

$$1-(1-\frac{s_1}{n_r})*(1-\frac{s_2}{n_r})*\ldots*(1-\frac{s_n}{n_r})$$

将此值乘上 n_r 即得到满足该选择的全部记录数。

• 取反。选择 $\sigma_{\neg\theta}(r)$ 的结果就是那些不在$\sigma_\theta(r)$中的元组。我们已经知道如何对$\sigma_\theta(r)$的大小作估计。因此，$\sigma_{\neg\theta}(r)$ 的大小可估计为

$$size(r) - size(\sigma_\theta(r))$$

可以用下列算法之一来实现涉及多个简单条件的合取或析取的选择：

• *A8*（利用一个索引的合取选择）。首先我们判断是否存在某个简单条件中的某个属性上的一条存取路径。若存在，则 *A2*～*A7* 选择算法之一可以用来检索满足该条件的记录。在内存缓冲区中，通过测试每条检索到的记录是否满足其余条件，可以完成这个操作。

为减少代价，我们选择某个 θ_i 及 *A1*～*A7* 算法之一，其组合可使 $\sigma_{\theta_i}(r)$ 的代价达到最小。对于其余条件，那些选择最少元组数的条件应首先测试。这样，只需做较少的测试就可将那些不满足合取式的记录剔除，从而减少 CPU 代价。

• *A9*（使用组合索引的合取选择）。对某些合取选择来说，可能存在合适的组合索引（即在多个属性上建立的一个索引）供其使用。如果选择声明的是一个或多个属性上的等值条件，并且在这些属性字段的组合上又存在组合索引，则可以直接搜索索引。索引的类型将决定

使用 A3、A4、A5 算法中的哪一个。

• A10（通过标识符的交集实现合取选择）。另一种实现合取选择的方法是利用记录指针或记录标识。该算法要求各个条件所涉及的字段上有带记录指针的索引。对每个索引进行扫描，获取那些指向满足单个条件的记录的指针。所有检索到的指针的交集就是满足合取条件的指针集合。然后利用该指针集合去检索实际的记录。如果并非各个条件上均存在索引，则要用剩余条件对所检索到的记录进行测试。

• A11（通过标识符的并集实施析取选择）。如果析取选择中所有条件均有相应的存取路径存在，则逐个扫描索引获取满足单个条件的指针。检索到的所有指针的并集就是指向满足析取条件的全体元组的指针集，然后利用这些指针去检索实际的记录。

然而，即使其中只有一个条件不存在存取路径，我们也不得不对这个关系进行线性扫描以找出那些满足该条件的元组。故此，只要析取式中有一个这样的条件，最有效的存取方法就是线性扫描，并对每条记录进行析取条件测试。

具有取反条件的选择的实现留作练习（习题 12.15）。另外，包含析取、合取条件的组合条件也留作练习（习题 12.16）。

为了说明前面的算法，我们来看下面的查询

select *account-number*
from *account*
where *branch-name* = "Perryridge" **and** *balance* = 1200

假定有关 *account* 关系的统计信息同前面的例子一样。

如果使用 *branch-name* 上的索引，如前所述，所需代价是 12 次块读取。如果利用 *balance* 上的索引，所需块存取数估计如下：由于 V（*balance*，*account*）= 500，预计有 10000/500 = 20 个元组满足 *balance* = $1200 的条件。然而，由于 *balance* 索引非聚集索引，因而估计每个元组需一次块读取。因此，检索 *account* 的元组共需 20 次块读取。

假定 *balance* 的 B^+ 树索引结点可容纳 20 个指针。由于有 500 个不同的 *balance* 值，因而该树大约有 25～50 个叶结点。这样，与 *branch-name* 上的 B^+ 树索引一样，*balance* 索引深度为 2，读必要的索引块需 2 次块存取。因此，这个策略共需 22 次块读取。

这样得到的结论是利用 *branch-name* 索引比较可取。注意，如果两个索引均为非聚集索引，则我们宁愿用 *balance* 上的索引，原因是满足 *balance* = 1200 的元组预计只有 20 个，而 *branch-name* = "Perryridge" 的元组预计会有 200 个。没有聚集特性时，第一种策略读取数据需要 200 次块读取，因为最坏的情况下每个记录各在一块，此外加上 2 次索引块存取，共需 202 次块存取。然而，由于 *branch-name* 索引的聚集特性，本例中使用 *branch-name* 索引时事实上代价相对较小。

用索引处理本查询的另一种方法是通过标识符的交集来进行。使用 *balance* 索引检索出指向满足条件 *balance* = 1200 的记录指针，而不是检索记录本身。令 S_1 表示该指针集合。同样，用 *branch-name* 索引检索出指向满足条件 *branch-name* = "Perryridge" 的记录的指针集合，记为 S_2。那么 $S_1 \cap S_2$ 就是指向满足条件 *branch-name* = "Perryridge" 和 *balance* = 1200 的记录的指针集合。

这个技术要访问两个索引。两个索引高度均为 2。两个索引各自检索到的指针数，前面已估计分别为 20 与 200，都可以放入单个叶结点页面中。因此，为获得两个指针集合共需进行 4

次块读取。计算两个指针集合的交集不需要进一步的磁盘 I/O。通过估计 $S_1 \cap S_2$ 中的指针数，可以估计从 *account* 读取的块数。

根据 V（*branch-name*，*account*）= 50 和 V（*balance*，*account*）= 500，我们估计每 50 * 500 = 25000 个元组中即有一个能同时满足 *branch-name* = "Perryridge" 及 *balance* = 1200。该估计是基于前面所做的均匀分布假设以及 *branch-name* 与 *balance* 的分布相互独立的假设。基于这些假设，对 *account* 的 10 000 条记录而言，我们保守地估计 $S_1 \cap S_2$ 中含有一个指针，因此，只需读取一次 *account* 的块。该策略总代价估计值是 5 次块读取。

12.5 排序

数据排序在数据库子系统中有重要的作用，其原因有两个：第一，SQL 查询可以指明对结果进行排序。第二，关系运算中的一些运算（如连接运算）在作为输入的关系已排序时实现的效率可以更高。对查询处理而言，这个原因和第一个原因一样重要。因此，我们先讨论排序，然后在 12.6 节中讨论连接运算。

通过在排序码上建立索引，然后使用该索引按序读取关系，我们可以完成排序。当然，这一过程通过索引对关系进行的是逻辑排序，而没有对关系进行物理排序。因此，顺序读取元组可能导致每读一个元组就要访问一次磁盘。由于这个缘故，有时还需要在物理上排序。

人们已对关系排序的有关问题进行了广泛的研究，既包括整个关系在内存中的情况又包括关系不能完全被内存容纳的情况。第一种情况下可以利用标准的排序技术如快速排序。在这里我们讨论如何处理第二种情况。

对不能全部放在内存中的关系进行排序称为外排序。外排序中最常用的技术是外部排序归并算法，下面我们讲述该算法。令 M 表示内存中用于缓冲的页面数（即内存缓冲能容纳的磁盘块数）。

1）第一阶段，建立多个排序的归并段文件。

```
i←0;
repeat
    读入 M 块关系数据或剩下的不足 M 块的数据;
    在内存中对关系的这一部分进行排序;
    将排好序的数据写到归并段文件 R_i 中;
    i = i + 1;
until 到达关系末尾
```

2）第二阶段，对归并段文件进行归并。暂时假定归并段文件总数 N 少于 M，这样可以为每个归并段文件分配一个内存页，此外剩下的空间还应能容纳存放结果的一页。归并阶段的工作流程如下：

为 N 个归并段文件 R_i 各分配一页内存缓冲页，并分别读入一数据块；

```
repeat
    在所有缓冲页中按序挑选第一个元组;
把该元组作为输出写出，并将其从缓冲页中删除;
if 任何一个归并段文件 R_i 的缓冲页为空并且没有到达 R_i 末尾
then 读入 R_i 的下一块到相应的缓冲页;
```

until 所有的缓冲页均为空

归并阶段的输出是已排序的关系。输出文件是经缓冲的，这样可以减少写盘次数。前面的归并算法是对标准内存排序算法——二路归并的推广，由于该算法对 N 个归并段文件进行归并，故称它为 n 路归并。

一般而言，若关系比内存大得多，则在第一阶段可能需要 M 个甚至更多个归并段文件，那么在归并阶段给每个归并段文件分配一个缓冲页是不可能的。这种情况下，归并操作需要多趟才能完成。由于内存足以容纳 $M-1$ 个缓冲页，每趟归并可以以 $M-1$ 个归并段文件为输入。

第一趟归并过程如下。头 $M-1$ 个归并段文件如前所述进行归并得到一个归并段文件作为下一趟的输入。接下来的 $M-1$ 个归并段文件以同样方法归并，如此下去，直到所有的初始归并段文件被处理过为止。此时，归并段文件的数目减少到原来的 $1/(M-1)$。如果归并后的归并段文件数目仍大于等于 M，则以上一趟创建的归并段文件作为输入进行下一趟归并。每一趟归并段文件的数目均减少到上一趟归并输入文件数的 $1/(M-1)$。根据需要的趟数重复进行归并，直到归并段文件数目小于 M，此时做最后一趟归并，得到排序的输出结果。

图 12-3 展示了对一个示例关系进行外部归并-排序的过程。为方便说明，假定每块只能容纳 1 个元组（$f_r=1$），同时假定内存最多只能提供 3 页缓冲。在归并阶段，两个缓冲页用于输入，另一页用于输出。

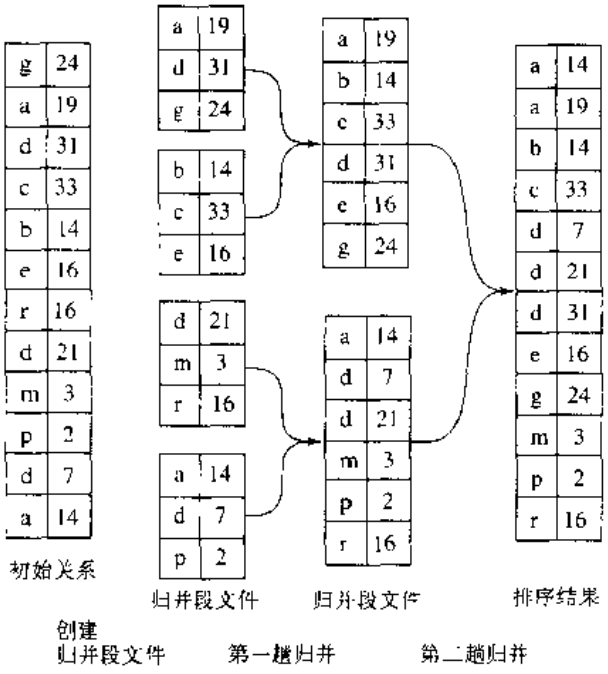


图 12-3　使用排序-归并的外排序

下面计算外部归并排序需多少次块传送。在第一阶段，关系的每一数据块需读入并写出，共需 $2b_r$ 次磁盘访问。初始归并段文件数为 $\lceil b_r/M \rceil$。由于每一趟归并段文件的数目均减少为原来的 $1/(M-1)$，总共所需归并趟数为 $\lceil \log_{M-1}(b_r/M) \rceil$。每一趟归并，关系的每一数据块读写各一次，其中有两趟例外。首先，最后一趟可以只产生排序结果而不写入磁盘。其次，可能存在某一趟中不用读入或不用写出的归并段文件。例如，某一趟有 M 个归并段文件需归并，其中 $M-1$ 个被读入并归并，而另一个归并段文件在该趟归并中却未被访问。忽略后

一种特殊情况所能减少的访问，关系外排序磁盘存取总数为

$$b_r(2\lceil \log_{M-1}(b_r/M)\rceil + 1)$$

把该公式用到如图 12-3 所示的例子上，算出共需 $12 * (4 + 1) = 60$ 次块传送，这一结果可以在图中得到验证。注意，该值不包括将最后结果写出的开销。

12.6　连接运算

本节首先讲述如何估计连接运算结果集的大小，然后探讨计算关系连接的几个算法，并分析各种算法的代价。我们用等值连接这个词来表示形如 $r \bowtie_{r.A=s.B} s$ 的连接，其中 A、B 分别为关系 r 与 s 的属性或属性组。

来看表达式

$$depositor \bowtie customer$$

我们假定这两个关系的目录信息如下：

- $n_{customer} = 10\ 000$。
- $f_{customer} = 25$，意指 $b_{customer} = 10000/25 = 400$。
- $n_{depositor} = 5000$。
- $f_{depositor} = 50$，意指 $b_{depositor} = 5000/50 = 100$。
- V（*customer-name*，*depositor*）$= 2500$，意指平均每个客户有 2 个帐户。

我们还假定 *depositor* 的属性 *customer-name* 是参照 *customer* 的外码。

12.6.1　连接结果集大小的估计

笛卡儿积 $r \times s$ 包含 $n_r * n_s$ 个元组。$r \times s$ 的每个元组占 $S_r + S_s$ 个字节，据此可以计算笛卡儿积结果集的大小。

对自然连接结果集大小的估计比选择运算或笛卡儿积运算结果集大小的估计更复杂。设 r（R）与 s（S）是关系。

- 若 $R \cap S = \varnothing$ 即关系之间无公共属性，则 $r \bowtie s$ 与 $r \times s$ 结果一样，可用对笛卡儿积结果集大小一样的方法估计。
- 若 $R \cap S$ 是 R 的码，则可知 s 的一个元组至多与 r 的一个元组连接。因此，$r \bowtie s$ 的元组数不会超过 s 中的元组数。若 $R \cap S$ 构成了 S 中参照 R 的外码，则 $r \bowtie s$ 中的元组数正好与 s 中的元组数相等。$R \cap S$ 是 S 的码的情形同 $R \cap S$ 是 R 的码的情形是对称的。

我们所举的例子 *depositor* $\bowtie$ *customer* 中，*depositor* 中的属性 *customer-name* 是参照 *customer* 的外码。因此，结果集大小正好是 $n_{depositor}$，即 5000。

- 最难考虑的情形是 $R \cap S$ 既不是 R 的码也不是 S 的码。这种情况与在选择运算的情况下一样，我们假定每个值等概率出现。考虑 r 的元组 t，假定 $R \cap S = \{A\}$。我们估计元组 t 在 $r \bowtie s$ 中产生

$$\frac{n_s}{V(A, s)}$$

个元组，该值就是 S 关系中属性 A 的值给定情况下平均的元组数目。考虑 r 中的所有元组，我们估计在 $r \bowtie s$ 中有

$$\frac{n_r * n_s}{V(A, s)}$$

个元组。注意，如果将 r 与 s 的角色颠倒，那么估计在 $r \bowtie s$ 中有

$$\frac{n_r * n_s}{V(A, r)}$$

个元组。当 $V(A, s) \neq V(A, r)$ 时，这两个估计是不同的。若发生这种情况，就可能有未参加连接的悬挂元组存在，由此这两个估计值中较小者可能比较准确。

当 r 中属性 A 的 $V(A, r)$ 个值与 s 中属性 A 的 $V(A, s)$ 个值相等的很少时，上述连接结果集大小的估计值可能太高。然而，实际中这种情形很少发生，因为大部分实际关系中悬挂元组要么不存在要么只占总元组数的一小部分。更重要的一点是，前面的估计是在各个值等概率这一假设前提下作出的。若这个假设不成立，则必须用更复杂的估算方法。

下面在不使用外码信息情况下计算 *depositor* ⋈ *customer* 的大小估计值。根据 V (*customer-name*, *depositor*) = 2500 及 V (*customer-name*, *customer*) = 10000，我们得到两个估计值：5000 * 10000/2500 = 20 000 及 5000 * 10000/10000 = 5000，我们取较低者。这里，估计值的较低者与我们早先用外码信息计算所得相同。

12.6.2　嵌套循环连接

图 12-4 所示过程是计算关系 r 和 s 的 theta 连接 $r \bowtie_\theta s$ 的一个简单算法。由于该算法主要由两个嵌套的 for 循环构成，故称为嵌套循环连接。从该过程中可以看到，算法中有关 r 的循环包含了有关 s 的循环，因而 r 称为连接的外层关系，s 称为连接的内层关系。算法中 t_r、t_s 表示 r、s 的元组，$t_r \cdot t_s$ 表示将 t_r、t_s 元组的属性值拼接而成的一个新元组。

```
for each 元组 t_r in r do
begin
  for each 元组 t_s in s do
  begin
    测试元组对 (t_r、t_s) 是否满足连接条件 θ
    如果满足，把 t_r·t_s 加到结果中
  end
end
```

图 12-4　嵌套循环连接

与选择算法中使用的线性文件扫描算法类似，嵌套循环连接算法不要求有索引，并且不管是什么连接条件，该算法均可使用。对此算法做简单的扩展就可以计算自然连接，因为自然连接可表示为 theta 连接再加上消除重名属性的投影运算。所需的修改只是在将 $t_r \cdot t_s$ 放入结果集之前删除 $t_r \cdot t_s$ 的重复属性。

嵌套循环连接算法的代价很大，因为该算法逐个检查两个关系中每一对元组。让我们看一下嵌套循环连接算法的代价。若元组对数目是 $n_r * n_s$。对于关系 r 中的每一条记录，我们必须对 s 做一次完整的扫描。最坏的情况下，缓冲区只能容纳每个关系的一个数据块，这时共需 $n_r * b_s + b_r$ 次块存取。最好的情况下，内存空间容纳两个关系，此时每一数据块只需读一次，从而只需 $b_s + b_r$ 次块存取。

如果较小的那个关系能完全放在内存中，把这个关系作为内层关系来处理是有好处的。因为内层循环关系只需读一次，这种方式大大地加快了查询处理。因此，如果 s 足够小，可以装入内存，那么我们的策略只需 $b_s + b_r$ 次块存取，其代价与两个关系能同时装入内存的情形相同。

现在考虑 *depositor* 与 *customer* 的自然连接。暂时假设两个关系中没有任何索引可供利用，并且也不想创建任何索引，我们可以用嵌套循环计算连接。假定连接中 *depositor* 是外层关系，*customer* 是内层关系，这时我们需要检查 $5000 * 10000 = 50 * 10^6$ 对元组。最坏的情况下，块读

写数是 5000 * 400 + 100 = 2 000 100；最好的情况下，两个关系只需读一次，然后进行计算，其代价最多只要 100 + 400 = 500 次块访问，大大好于最坏的情况。如果把 *customer* 作为外层关系，把 *depositor* 作为内层关系，则此策略在最坏情况下的代价会低一些，为 10000 * 100 + 400 = 1 000 400。

12.6.3 块嵌套循环连接

当缓冲区太小从而使内存中不能同时容纳两个关系时，如果以块的方式而不是以元组的方式处理关系，仍然可以省去不少块读写次数。图 12-5 所示过程是嵌套循环连接的一个变种，其中内层关系的每一块与外层关系的每一块形成一对。每个块对中，一个块中的每一个元组与另一块中的每一个元组形成元组对，这样就得到了全体元组对。和前面一样，满足连接条件的所有元组对被添加到结果中去。

```
for each 块 B_r of r do
begin
  for each 块 B_s of s do
  begin
    for each 元组 t_r in B_r do
    begin
      for each 元组 t_s in B_s do
      begin
        测试元组对 (t_r, t_s) 是否满足连接条件 θ
        如果满足，把 t_r·t_s 加到结果中
      end
    end
  end
end
```

图 12-5 块嵌套循环连接

块嵌套循环连接与基本嵌套循环连接算法的主要差别在于：最坏的情况下，对于外层关系的每一块，内层关系 s 的每一块只需读一次，而不是对外层关系的每一个元组读一次。因此，最坏的情况下共需 $b_r * b_s + b_r$ 次块访问。显然，使用较小的关系作为外层关系是最有效的，最好的情况下只需 $b_r + b_s$ 次块存取。

回到计算 *depositor* ⋈ *customer* 的例子，现在用块嵌套循环连接算法来计算。最坏的情况下，必须为每一个 *depositor* 块读一次 *customer* 块，所以共需 100 * 400 + 100 = 40 100 次块访问。这个代价与采用基本嵌套循环连接算法的 5000 * 400 + 100 = 2 000 100 次块访问相比，已经有很大的改进。但最好的情况下，块读写次数还和原来的一样，是 100 + 400 = 500 次。

对嵌套循环与块嵌套循环连接算法的性能可以做进一步的改进：

- 如果自然连接或等值连接中的连接属性是内层关系的码，则内层循环一旦找到了首条匹配元组就可以终止。
- 在块嵌套循环连接算法中，外层关系的块可以不用磁盘块作单位，而以内存中最多能容纳的大小为单位，当然这是在保留足够的缓冲给内层关系及输出结果使用的前提下。这种改进减少了内层关系的扫描次数。
- 对内层循环轮流做前向、后向扫描。该扫描方法对磁盘块读写请求进行排序，使得上一次扫描后留在缓冲区中的数据可以被重用，从而减少磁盘存取次数。
- 若内层循环连接属性上有索引，可以用更有效的索引查找法替代文件扫描法。这一改

进在下节中讲述。

12.6.4　索引嵌套循环连接

在图 12-4 的嵌套循环连接中，若存在内层循环连接属性上的索引，则无论是永久的还是临时的索引，我们都可以用索引查找替代文件扫描。对于外层关系 r 的每一个元组 t_r，可以利用索引查找 s 中和元组 t_r 满足连接条件的元组。

上述连接方法称为*索引嵌套循环连接*，它可以在已有索引或是为了计算该连接而专门建立临时索引的情况下使用。

给定元组 t_r 的情况下，在关系 s 中查找满足连接条件的元组本质上是在 s 上做选择运算。例如，考虑 *depositor* ⋈ *customer*。假设 *depositor* 中有一个 *customer-name* 为“John”的元组，那么 s 中的相关元组就是那些满足选择条件 *customer-name* = “John”的元组。

索引嵌套循环连接的代价计算如下。对于外层关系 r 的每一个元组，需要在关系 s 的索引上进行查找，检索相关元组。最坏的情况下，缓冲区只能容纳关系 r 的一页和索引的一页。此时，读取关系 r 需 b_r 次磁盘存取，且对应关系 r 上的每个元组都进行一次 s 上索引的查找。连接的代价可用 $b_r + n_r * c$ 来计算，其中 c 是使用连接条件对关系 s 进行单个选择运算的代价。我们已经讲述了使用各种索引时如何计算对关系进行单个选择运算的代价，这可以使我们计算出 c。代价计算公式表明，如果两个关系 r、s 上均有索引，一般而言把元组较少的关系作为外层关系效果更好。

例如，考虑 *depositor* ⋈ *customer* 的索引嵌套循环连接，其中 *depositor* 作为外层关系。假设关系 *customer* 在连接属性 *customer-name* 上有 B^+ 树主索引，每个索引结点可容纳 20 项。由于 *customer* 有 10 000 个元组，因而树深度为 4，存取实际数据时还多需一次磁盘访问。又由于 $n_{depositor} = 5000$，总代价为 $100 + 5000 * 5 = 25\ 100$ 次磁盘访问。这一代价小于块嵌套循环连接算法的 40 100 次磁盘访问。

12.6.5　归并连接

归并连接算法（又称*排序-归并-连接算法*）可用于计算自然连接和等值连接。假设要计算关系 r（R）和 s（S）的自然连接，$R \cap S$ 表示两个关系的公共属性。假定两个关系均按 $R \cap S$ 排序，那么可用与归并排序算法中归并阶段极为类似的处理过程来计算它们的连接。

归并连接算法如图 12-6 所示。在这个算法中，*JoinAttrs* 表示 $R \cap S$ 中的属性，$t_s \bowtie t_r$ 表示具有相同 *JoinAttrs* 属性的两个元组 t_s、t_r 的拼接，其中重复的属性已通过投影去除。归并连接算法为每个关系分配一个指针。这些指针一开始指向对应关系的第一个元组。随着算法的进行，指针遍历整个关系。其中一个关系中在连接属性上有相同值的所有元组被加入到 S_s 中。图 12-6 所示算法要求每个 S_s 元组集合都能全部装入内存；本节稍后讲述如何扩展该算法以避免这一限制。接下来，另一关系中的相应元组（如果有的话）被读入，并在读入同时加以处理。

图 12-7 给出了两个在公共属性 $a1$ 上排序的关系。利用图中的两个关系具体地走一遍归并连接算法是很有启发的。

由于关系已排序，所以在连接属性上有相同值的元组连续存放。这样已排序的每一元组只需读一次，从而每一块也只需读一次。两个文件都只需读一遍，可知归并连接算法是高效的，所需磁盘访问次数是两个文件块数之和：$b_r + b_s$。

如果输入关系 r 或 s 未按连接属性排序，那么在使用归并连接算法之前可先对其排序。归并连接算法可以很容易地从自然连接拓展到更一般的等值连接情形。

```
pr := r 的第一个元组的地址;
ps := s 的第一个元组的地址;
while (ps≠null and pr≠null) do
  begin
    t_s := ps 所指向的元组;
    S_s := {t_s};
    让 ps 指向关系 s 的下一个元组;
    done := false;
    while (not done and ps≠null) do
      begin
        t_s' := ps 所指向的元组;
        if (t_s [JoinAttrs] = t_s' [JoinAttrs])
          then begin
              S_s := S_s ∪ {t_s'};
              让 ps 指向关系 s 的下一个元组;
            end
          else done := true;
      end
    t_r := pr 所指向的元组;
    while (pr≠null and t_r [JoinAttrs] < t_s [JoinAttrs]) do
      begin
    让 pr 指向关系 r 的下一个元组;
    t_r := pr 所指向的元组;
  end
  while (pr≠null and t_s [JoinAttrs] = t_s [JoinAttrs]) do
  begin
    for each t_s in S_s do
      begin
        将 t_s ⋈ t_r 加入结果中;
      end
    让 pr 指向关系 r 的下一个元组;
    t_r := pr 所指向的元组;
  end
end
```

图 12-6　归并连接

假设将归并连接算法应用到 *depositor* ⋈ *customer* 例子上，连接属性是 *customer-name*。假定两个关系已按连接属性 *customer-name* 排序，这种情况下，归并连接的代价为 400 + 100 = 500 次块访问。假设两个关系未排序，内存大小也属于最差情形——只有 3 块，那么对 *customer* 进行排序的代价是 $400 * (2\lceil \log_2 (400/3) \rceil + 1) = 6800$ 次块存取，此外还再加上 400 次将结果写出的块传送。同样对 *depositor* 进行排序要 $100 * (2\lceil \log_2 (100/3) \rceil + 1) = 1300$ 次块传送及 100 次将结果写出的块传送。因此，若关系未排序，则归并连接共需 9100 次块传送。

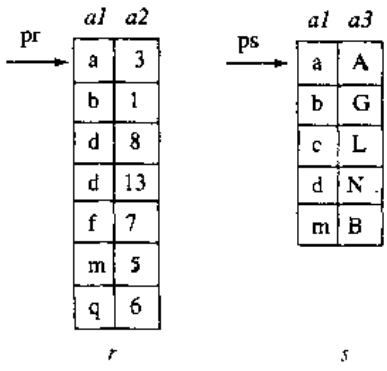


图 12-7　在归并连接中使用的已排序关系

如前所述，图 12-6 所示归并连接算法要求内存中能容纳

在连接属性上有相同值的元组集 S_s。这个要求通常可以满足，即使关系 s 较大时亦如此。若不能满足，则必须在 S_s 与关系 r 中具有相同连接属性值的元组之间采用块嵌套循环连接，归并连接总代价将因此而增加。

当两个关系的连接属性上存在辅助索引时，也可能对未排序的元组执行变相的归并连接运算。通过索引扫描记录，效果相当于顺序检索元组。但这种变相导致更大的缺陷，因为记录可能分散在文件块中，从而元组的每次存取都要导致一次磁盘访问，这样代价是很大的。

为避免这种代价，我们可以使用一种混合归并连接技术，该技术把索引与归并连接相结合。假设两个关系中有一个已排序，另一个未排序，但未排序的关系在连接属性上有 B^+ 树辅助索引。混合归并连接算法把已排序关系和另一关系的 B^+ 树辅助索引叶结点项进行归并，所得结果文件中包含了来自已排序关系的元组及未排序关系的元组地址。为完成连接运算，将该文件按未排序关系元组的地址进行排序，并用此文件有效地按物理存储顺序检索相应元组。将这个技术拓展用于处理两个未排序关系的工作留作练习。

12.6.6　散列连接

类似归并连接算法，散列连接算法可用于实现自然连接和等值连接。在散列连接算法中，散列函数 h 用来对两个关系进行划分。此算法的基本思想是把两个关系按连接属性值划分成一些有相同散列值的元组集合。

假设：

- h 是将 *JoinAttrs* 值映射到{0，1，2，…，max} 的散列函数，其中 *JoinAttrs* 表示 r 与 s 自然连接中的公共属性组。
- H_{r_0}，H_{r_1}，…，$H_{r_{max}}$ 表示关系 r 的元组划分，初始均为空集。每个元组 $t_r \in r$ 被放入分划 H_{r_i} 中，其中 $i = h\ (t_r\ [JoinAttrs])$。
- H_{s_0}，H_{s_1}，…，$H_{s_{max}}$ 表示关系 s 的元组划分，初始均为空集。每个元组 $t_s \in s$ 被放入分划 H_{s_i} 中，其中 $i = h\ (t_s\ [JoinAttrs])$。

散列函数 h 应当具有在第 11 章讨论过的良好特性——随机性、均匀性。关系的划分过程如图 12-8 所示。

散列连接算法的基本思想如下。如果关系 r 的一个元组与关系 s 的一个元组满足连接条件，那么它们在连接属性上有相同的值。若该值经散列函数映射为 i，则关系 r 的那个元组必在 H_{r_i} 中，关系 s 的那个元组必在 H_{s_i} 中。因此，H_{r_i} 中的元组只需与 H_{s_i} 中的元组相比较，而无需与 s 的其他任何分划比较。

例如，如果 d 是 *depositor* 中的一个元组，c 是 *customer* 中的一个元组，h 是元组属性 *customer-name* 上的散列函数，那么只有在 $h\ (c) = h\ (d)$ 时 d 与 c 才需比较。若 $h\ (c) \neq h\ (d)$，则 d 与 c 在属性 *customer-name* 上的值必不相等。之所以 $h\ (c) = h\ (d)$ 时必须检查 d 与 c 在连接属性上的值，是因为 d 与 c 有相同的散列值却可能有不同的 *customer-name* 值。

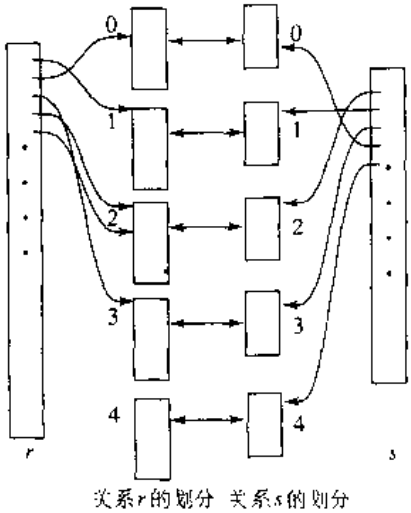


图 12-8　关系的散列划分

图 12-9 显示了散列连接算法计算关系 r 与关系 s 自然

连接的详细过程。和归并连接算法一样， $t_r \bowtie t_s$ 表示元组 t_r、t_s 属性拼接后滤除重复属性的投影。将关系进行划分后，散列连接代码的其余部分分别完成各个分划对 i（$i=0, 1, \cdots, max$）的索引嵌套循环连接。为此，算法为每个 H_{s_i} 建立散列索引，然后在 H_{r_i} 中查找与 H_{s_i} 中元组匹配的元组。关系 s 称为构造用输入，关系 r 称为查找用输入。

```
/* 对关系 s 进行划分 */
for each 元组 t_s in s do
  begin i := h (t_s[JoinAttrs]);
  H_si := H_si ∪ {t_s};
end
/* 对关系 r 进行划分 */
for each 元组 t_r in r do
  begin i := h (t_r [JoinAttrs]);
  H_ri := H_ri ∪ {t_r};
end
/* 对每一分划进行连接 */
for i := 0 to max do
  begin 读 H_si 在内存中建立其散列索引;
  for each 元组 t_r in H_ri do
  begin 检索 H_si 的散列索引，定位所有满足 t_s [JoinAttrs] = t_r [JoinAttrs]
  的元组 t_s
  for each 匹配的元组 t_s in H_si do
  begin 将 t_r ⋈ t_s 加入结果中
  end
 end
end
```

图 12-9　散列连接

H_{s_i} 的散列索引是在内存中建立的，因此检索元组并不需要访问磁盘。用于构造此索引的散列函数与前面使用的散列函数 h 是不同的，但仍是对连接属性进行散列映射。在进行索引嵌套循环连接时，系统将使用该索引检索那些与查找用输入关系相匹配的记录。

构造阶段与检索阶段只需对构造用与查找用输入关系做一次扫描。将散列连接算法推广到计算更普遍的等值连接是很容易的。

max 值的选择应足够大，以使对任意 i，内存中可以容纳构造用输入关系的任一 H_{s_i} 以及其上的散列索引。内存中可以不必容纳查找用输入关系的分划。显然，最好用较小的输入关系作为构造用输入关系。如果构造用关系有 b_s 块，那么要使 max 个分划中每一个分划小于等于 M，max 值至少应是 $\lceil b_s/M \rceil$。更准确些，还应当考虑分划上散列索引将占用的内存空间。因此，max 值相应取大一点。为简单起见，分析中有时我们忽略散列索引将使用的内存空间。

1. 递归划分

如果 max 的值大于等于内存页帧数，关系的划分不可能一趟完成，因为没有足够的缓冲页。这时，完成关系的划分需要重复多趟。每一趟中，输入的可最多分裂的分划数不超过用于

输出的缓冲页数。每一趟产生的存储桶又在下一趟中分别读入并再次划分，创建较小的分划。当然，每趟划分中所用的散列函数与上一趟所用的散列函数是不同的。分裂过程不断重复直到构造用输入关系的各个分划都能被内存容纳为止。这种划分方法称为递归划分。

当 $M > max + 1$ 即 $M > (b_s/M) + 1$（近似简化为：$M > \sqrt{b_s}$）时，关系不需要进行递归划分。例如，考虑如下情形：内存大小是 12MB，分成 4KB 大小的块，共有 3000 块。我们可用这些内存对含有 900 万块（36GB）的关系进行划分。同样，1GB 大小的关系需 $\sqrt{250\,000}$ 个内存块，约 2MB。

2. 溢出处理

当 H_i 的散列索引大于内存时，构造用输入关系 s 的分划 i 发生散列表溢出。如果构造用输入关系在连接属性上具有相同值的元组数很多，或所选散列函数没有随机性、均匀性，那么就会发生散列表溢出。这两种情况下，某些分划所含元组数多于平均数，而另一些分划所含元组数少于平均数。这样的划分称为是有偏的。

少量的偏斜可以做如下处理。增加分划个数，使得每个分划的大小（包括该分划上的散列索引在内）小于内存容量。分划因此的少量增加称为避让因子，数目通常是按前面所描述方法计算出的散列分划数的 20% 左右。

即使我们利用避让因子，在分划大小上采取了比较保守的态度，但散列表溢出仍然在所难免。散列表溢出可以通过分解或避让的方法来进行处理。在构造阶段若发现了散列索引溢出，则进行溢出分解。溢出分解过程如下。任给 i，若发现 H_{s_i} 太大，我们就用另一个散列函数进一步将之划分成更小的分划。同样，我们用新的散列函数对 H_{r_i} 进行划分，且只有那些在匹配的分划中的元组才需连接。

与溢出分解法不同，溢出避让法对划分进行了细致的考虑，保证构造阶段没有溢出发生。我们可以这样实现溢出避让法。首先将关系 s 划分成许多小分划，然后把某些分划组合在一起，确保组合后的分划能被内存容纳。查找用关系 r 必须按照与关系 s 上的组合分划相同的方式进行划分，但 H_{r_i} 的大小无关紧要。

如果 s 中有大量元组在连接属性上有相同的值，溢出分解与溢出避让法可能在某些分划上失效。这种情况下，我们不采用在内存中创建散列索引，然后用嵌套循环连接算法对分划进行连接的方法，而是在那些分划上使用其他连接技术，如块嵌套循环连接。

3. 散列连接的代价

下面考虑散列连接的代价，分析中假定没有散列表溢出发生。首先考虑不需递归划分的情形。对两个关系 r、s 的划分需要对这两个关系分别进行一次完整的读入与写出，该操作需要 $2(b_r + b_s)$ 次块访问。在构造与检索阶段每个分划分别读入一次，又需要 $b_r + b_s$ 次块访问。分划所占用的块数可能比 $b_r + b_s$ 略多，因为有的块只是半满的。由于 max 个分划中每个分划可能有一个部分满的块，而该块需写出、读入各一次，因此对每个关系而言存取这些部分满的块的代价最多不超过 $2 * max$ 次。从而散列连接的代价估计是：

$$3(b_r + b_s) + 2 * max$$

其中 $2 * max$ 与 $b_r + b_s$ 相比是很小的，可以忽略。

现在来看看需要递归划分的情形。每一趟预计可将分划的大小减小为原来的 $1/(M-1)$。分划大小不断减小，直到每个分划最多占 M 块为止。划分所需的趟数预计为 $\lceil \log_{M-1}(b_s) - 1 \rceil$。由于每一趟中，关系 s 的每一块需读入、写出，划分过程总共所需块传送数是 $2b_s\lceil \log_{M-1}(b_s) - 1 \rceil$。对关系 r 进行划分所需趟数与关系 s 是类似的，由此得到如下的连接代价估计：

$$2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

例如，考虑连接 *depositor* ⋈ *customer*。内存有 20 块，*depositor* 划分成 5 个分划，每个分划占 20 块，正好能装入内存，划分时只需一趟。*customer* 同样划分成 5 个分划，每个分划占 80 块。忽略写出半满块的代价，则共需 3（100 + 400）= 1500 次块传送。

若内存较大，散列连接性能可以得到提高。当内存中可以容纳整个构造用输入关系时，*max* 可置为 0。此时，不管查找用输入关系的大小如何，都不必将关系划分成临时文件，因而散列连接可以快速执行，其代价估计降为 $b_r + b_s$。

4. 混合散列连接

混合散列连接算法是另一种改进。当内存相对较大，但还不足以容纳整个构造用输入关系时，该算法是很有用的。在划分阶段，散列连接算法需为创建的每一分划提供一内存块缓冲，另外还需一内存块作为输入缓冲区，因此划分两关系都共需 *max* + 1 内存块。如果内存大于 *max* + 1 块，可以用剩余内存块（$M - max - 1$ 块）对构造用输入关系的第一个分划（即 H_{s_0}）进行缓冲，从而避免将其写出后再读入。更进一步，可以适当设计散列函数，使得 H_{r_0} 上的散列索引能被 $M - max - 1$ 块内存所容纳。这样，在关系 s 划分结束后，H_{s_0} 完全在内存中且可以在 H_{r_0} 上建立散列索引。

当对 r 进行划分时，H_{r_0} 的元组也不必写回磁盘，而是在元组产生时，系统利用它们去查找驻留在内存中的 H_{s_0} 散列索引，并得到连接后的元组。H_{r_0} 的元组使用后就可以抛弃了，因此 H_{r_0} 分划不占内存空间。这样，对于 H_{r_0} 与 H_{s_0} 的每一块可以节省读、写各一次。其他分划的元组按普通方式写出，以后再连接。在构造用输入关系只是略大于内存时，混合连接算法可以大大降低代价。

若构造用输入关系大小为 b_s，则 *max* 近似为 b_s/M。因此，混合连接算法在 $M \gg b_s/M$ 或 $M \gg \sqrt{b_s}$ 时非常有用，记号≫表示远大于。例如，设块大小为 4KB，构造用输入关系大小为 1GB，那么混合连接算法在内存大于 2MB 时是很有用的。现今计算机通常拥有 50MB～100MB 或更多内存。

再来看看 *depositor* ⋈ *customer* 连接的例子。如果内存有 25 块，则 *depositor* 可划分为 5 个分划，每个分划大小为 20 块。第一个分划可以驻留内存，占 20 块，其次一个内存块用于输入，其他 4 个分划各占一块缓冲区。关系 *customer* 同样划分成 5 个分划，每个分划大小为 80 块，其中第一块立即用于查找，而不是先写出再读入。忽略写出半满块的代价，总代价是 3（80 + 320）+ 20 + 80 = 1300 次块传送，这在不用混合连接算法时的代价是 1500 次块传送。

12.6.7 复杂连接

嵌套循环连接与块嵌套循环连接不管在什么连接条件下均可使用。其他连接技术比嵌套循环连接及其变种更有效，但只能处理简单的连接条件，如自然连接或等值连接。利用 12.4.4 节中用于处理复杂选择的技术，我们可以用有效的连接技术实现具有复杂连接条件如合取式和析取式的连接。

来看下面带有合取条件的连接：

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$$

前面所述的一种或多种连接技术可以用于单个连接条件的计算，如 $r \bowtie_{\theta_1} s$、$r \bowtie_{\theta_2} s$、$r \bowtie_{\theta_3} s$ 等等。我们通过先计算这些较简单的 $r \bowtie_{\theta_i} s$ 连接中的某一个来计算整个连接。中间

结果元组对由 r 中的一个元组与 s 中的一个元组组成。完整的连接结果是中间结果中满足以下剩余条件的那些记录：

$$\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

这些条件可在产生 $r \bowtie_\theta s$ 元组时进行测试。

具有析取条件的连接可按如下方式计算。例如：

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n} s$$

该连接可以通过对各个连接 $r \bowtie \theta_i s$ 的元组取并来计算：

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \cdots \cup (r \bowtie_{\theta_n} s)$$

计算关系的并的算法将在 12.7 节中讲述。

下面考虑含有 3 个关系的连接运算：

$$account \bowtie depositor \bowtie customer$$

我们不仅可以选择连接处理策略，而且可以决定先计算哪个连接。可选策略可能有许多，在这里我们仅讨论其中的几个，并且留了一个作为练习（习题 12.4）。

- 策略 1。用我们曾讲述过技术之一计算连接 *depositor* ⋈ *customer*；利用该中间结果计算

$$account \bowtie (depositor \bowtie customer)$$

- 策略 2。类似策略 1，但首先计算 *account* ⋈ *depositor*，然后将结果与 *customer* 连接。

连接还可用其他的次序计算。

- 策略 3。不是分别计算两个连接，而是同时计算。该技术要先建立两个索引
  - 对 *account* 的 *account-number* 建立索引
  - 对 *customer* 的 *customer-name* 建立索引

下一步，考虑 *depositor* 中的每一个元组 t，寻找 *customer* 及 *account* 中对应的元组。这样，*depositor* 的每一个元组只需检索一次。

策略 3 所采用的方法是前面未曾讲述的。它不直接对应于一个关系代数运算，而把两个运算结合为一个特殊目的的运算。使用该策略计算三个关系的连接运算通常比两次执行两个关系的连接运算更有效，其相对代价取决于关系存储的方式、列值的分布方式以及是否有索引。习题 12.5 是计算这些代价的好的尝试。

12.7　其他运算

其他运算以及扩展关系运算（如消除重复、投影、集合运算、外连接、聚集）可以按 12.7.1～12.7.5 节所述加以实现。

12.7.1　消除重复

用排序方法可以很容易地消除重复。排序时等值元组相互邻近，删除其他副本只留一个元组副本即可。对于外部归并排序而言，归并段文件创建时就可以发现重复元组，可在将归并段文件写回磁盘之前去除重复元组，从而减少块传输次数。剩余的重复元组可在归并时去除，最后经排序的归并段将没有重复元组。消除重复的最坏情形估计与该关系排序时最坏情形代价估计一样。

也可使用和散列连接算法相似的散列方法来实现重复的消除。首先，整个关系基于整个元组上的一个散列函数进行划分。接下来每个分划被读入内存，建立内存中的散列索引。在创建散列索引时，只有不在索引中的元组才被插入。否则，元组就被抛弃。分划中的所有元组处理

完后，散列索引中的元组被写到结果中。其代价与散列连接中构造用输入关系的处理（划分以及读入每个分划）代价一样。

由于消除重复的代价相对较大，因而商用查询语言要求用户显式指明需要消除重复，若不指明则保留重复。

12.7.2 投影

可以通过如下方式比较容易地实现投影。首先对每个元组做投影，所得结果关系中可能有重复记录，然后消除重复记录。消除重复可按 12.7.1 节中所述的方法进行。若投影属性列表中含有码，则结果中不会有重复元组，因此不必做重复的消除。3.5.1 节中所述的广义投影可用与投影一样的方式来实现。形如 $\Pi_A(r)$ 的投影其结果大小估计为 $V(A, r)$，因为投影是消除重复的。

12.7.3 集合运算

要实现并、交、差集合运算，首先对两个关系进行排序，然后对每个已排序的关系扫描一次，产生所需结果。在 $r \cup s$ 中，当同时对两个文件进行扫描发现有相同元组时，只需保留其中一个。$r \cap s$ 的结果中只包含同时出现在两个关系中的元组。同样，通过只保留 r 中那些不属于 s 的元组，可以实现集合差运算。

对所有这些运算，两个输入关系仅需扫描一次，故代价为 $b_r + b_s$。若关系一开始未排序，则还要考虑排序的代价。在执行集合运算时，任何排序顺序均可，只是两个输入关系有相同的排序顺序。

散列提供了实现集合运算的另一种方法。对于每种运算，首先用相同的散列函数对两个关系进行划分，由此得到分划 H_{r_0}，H_{r_1}，…，$H_{r_{max}}$ 以及 H_{s_0}，H_{s_1}，…，$H_{s_{max}}$。然后对 $i = 0 \cdots max$ 的每一分划执行以下步骤：

- $r \cup s$
  1）对 H_{r_i} 建立内存中的散列索引。
  2）把 H_{s_i} 中的元组加入到以上散列索引中，条件是元组不在散列索引中。
  3）把散列索引中的元组加入结果中。
- $r \cap s$
  1）对 H_{r_i} 建立内存中的散列索引。
  2）对 H_{s_i} 的每个元组，检索散列索引，若它出现在其中，则将该元组写到结果中。
- $r - s$
  1）对 H_{r_i} 建立内存中的散列索引。
  2）对 H_{s_i} 的每个元组，检索散列索引，若它出现在其中，则将它从散列索引中删除。
  3）把散列索引中剩余的元组加入结果中。

12.7.4 外连接

回想一下 3.5.2 节中所描述的外连接运算。例如，自然左外连接 *customer* ⟕ *depositor* 包含了 *customer* 与 *depositor* 的连接。此外，对于 *customer* 中每一个在 *depositor* 中没有匹配元组的元组 t（即 t 中的 *customer-name* 不在 *depositor* 中），如下的元组 t_1 被加入结果中：对于 *customer* 模式的全部属性，元组 t_1 与 t 有相同的值；t_1 的其余属性（来自 *depositor* 模式）取空

值。

外连接可用以下两种策略之一来实现。第一种策略是计算相应的连接，然后将适当的元组加入到连接结果中以得到外连接结果。例如左外连接运算与两个关系 r（R）和 s（S）。为计算 $r \supset\!\bowtie_\theta s$，首先计算 $r \ltimes_\theta s$，将结果存为临时关系 q_1。接着，计算 $r - \Pi_R(q_1)$，得到所有属于 r 但未参与连接的元组。外连接的计算可以采用前面说过的用于计算连接、投影和集合差的任何算法。我们用空值填充这些元组中属于关系 s 的属性，然后将元组加到 q_1 中，得到外连接的结果。

右外连接运算 $r \bowtie\!\subset_\theta s$ 等价于 $s \supset\!\bowtie_\theta r$，可用与左外连接对称的方式实现。要实现全外连接运算 $r \supset\!\subset s$，可以先计算 $r \bowtie s$，然后和前面一样将左、右外连接的额外元组加入。

实现外连接的第二种策略是对连接算法加以修改。将嵌套循环连接算法扩展为计算左外连接的算法很容易，只要把与内层关系的任何元组都不匹配的外层关系元组在填充空值后写到结果中即可。然而，要将嵌套循环连接扩展为计算全外连接的算法却是很困难的。

自然外连接与具有等值连接条件的外连接可以通过扩展归并连接与散列连接算法来计算。对归并连接加以扩展，可以用来计算全外连接，方法如下。当两个关系的归并完成后，两个关系中那些不与另一个关系的任何元组相匹配的元组在填充空值后写到结果中。同样，我们可以扩展归并连接以计算左、右外连接，其方法是只将一个关系中不匹配的元组在填充空值后写到结果中。由于关系是排序的，很容易判断一个元组是否与另一个关系的元组相匹配。例如，当 *customer* 与 *depositor* 归并连接完成后，元组按 *customer-name* 的顺序读入，这样对于每个元组就很容易判断在另一个关系中是否有与之匹配的元组。

使用归并连接算法实现外连接的代价估计同相应连接的代价估计是一样的。唯一差异在于结果的大小以及由此导致的写出结果的块传送次数，而这一点在早先的代价估计中并没有考虑。

用扩展散列连接算法计算外连接留作为练习（习题 12.17）。

12.7.5　聚集

回忆 3.5.3 节中讲述的聚集运算符 $\mathcal{G}$。例如，运算

$$_{branch\text{-}name}\mathcal{G}_{\mathbf{sum}(balance)}(account)$$

根据分支机构对 *account* 元组进行分组，并且计算出每个分支机构中所有帐户的总余额。

聚集运算可以用与消除重复相类似的方法来实现。我们使用排序或散列，就像在消除重复时那样，不同的是这里基于分组属性进行（上面的例子中是 *branch-name*）。而且，我们不是去除在分组属性上有相同值的元组，而是将之聚集成组，并对每一组应用聚集运算以获取结果。$_A\mathcal{G}_F(r)$ 的大小就是 $V(A, r)$，因为对应 A 的每一个不同的值在 $_A\mathcal{G}_F(r)$ 中有一元组。对 min、max、sum、count 和 avg 等聚集函数而言，实施聚集运算的代价估计和消除重复的代价是一样的。

不必等到所有元组聚集成组后再施加聚集运算，可以在组的构造过程中就实施聚集运算，如 sum、min、max、count 以及 avg。对于 sum、min 与 max，当在同一组中发现了两个元组时，它们就被替换为被聚集列上包含 sum、min 或 max 的单个元组。对于 count 运算，我们为每一已发现元组的组维护一个计数值。最后，avg 运算可以实现如下。在组构造过程中，我们计算每一组的 sum 及 count，结束时将 sum 除以 count 即获得平均值。

如果结果中 $V(A, r)$ 个元组可以装入内存，则基于排序的实施方法与基于散列的实施方法都不必将元组写到磁盘上。当元组读入时，就可以把它插入到一个有序树结构中或插入到

一个散列索引中。当使用以上聚集技术时，对于 $V(A, r)$ 个组中的每一个组只需保存一个元组。因此，有序树结构或散列索引可在内存中容纳。聚集可在 b_r 次块存取中完成，而用其他方法时是 $3b_r$ 次块传送。

12.8 表达式计算

目前为止，我们只研究了单个关系运算如何执行，现在我们考虑如何计算包含多个运算的表达式。一种显而易见的方法是以适当的顺序每次执行一个操作，每次计算的结果被实体化到一个临时关系中以备后用。这种方法的缺点是需要构造临时关系，这些临时关系除非很小，否则必须写到磁盘上。另一种方法是在流水线上同时执行多个运算，一个运算结果传递给下一个，而不必保存到临时关系中。

下面两节将介绍实体化方法与流水线方法。我们将看到这两种方法的代价相差很大，并且有的情况下只能用实体化方法。

12.8.1 实体化

要直观地理解如何计算一个表达式，最容易的方法是看一看运算符树对表达式所做的图形化表示。考虑图 12-10 所示表达式：

$$\Pi_{customer\text{-}name}(\sigma_{balance<2500}(account) \bowtie (customer))$$

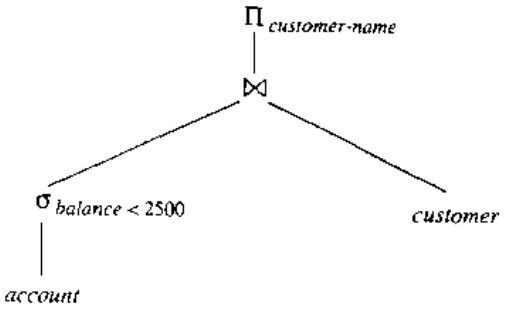


图 12-10 一个表达式的图形化表示

当采用实体化方法时，从表达式的最底层运算（在树的底部）开始。本例只有一个底层运算：*account* 上的选择运算。底层运算的输入是数据库中的关系。我们用前面学过的算法执行这个运算，并将结果存储在临时关系中。在树的更高一层，使用临时关系进行计算，这时的输入要么是临时关系，要么是来自数据库的关系。本例连接运算的输入是 *customer* 关系及在 *account* 关系上进行选择时所建立的临时关系。连接运算的结果是建立起另一个临时关系。

通过重复这一过程，最终可以执行位于树根的运算，得到表达式的最终结果。本例位于树根的运算是投影，以连接运算产生的临时关系作为输入执行投影，我们就可以得到最终结果。

上述计算方法被称为实体化计算，这是因为运算的中间结果被创建（实体化），然后用于下一层的运算。

实体化计算的代价不仅仅是那些所涉及的运算代价的总和，当估计各算法的代价时，我们忽略了将运算结果写到磁盘上的代价。为了计算按如上所述执行一个表达式的代价，必须把所有运算的代价相加，同时还要加上把中间结果写回磁盘的代价。假设结果记录在缓冲区中积聚，当缓冲区满时，再把它们写到磁盘上。写出结果的代价可按 n_r/f_r 来估算，其中 n_r 是结果关系估计的元组数，f_r 是结果关系的块因子。双缓冲技术（即使用两个缓冲区，其中一个用

于连续执行算法，另一个用于写出结果）使 CPU 活动与 I/O 活动并行，从而提高算法执行速度。

12.8.2 流水线

通过减少查询执行中产生的临时文件数可以提高查询执行的效率。减少临时文件数是通过将多个关系操作组合成一个操作的流水线来实现的，即将一个操作结果传送到下一个操作。将操作组合成流水线可以去除读写临时关系的代价。

例如，在两个关系连接后进行一个投影操作（$\Pi_{a1,a2}(r \bowtie s)$）。如果使用实体化方法，执行中将创建存放连接结果的临时关系，然后在执行投影时又从临时关系中读入连接结果。这些操作可按如下方式组合：当连接操作产生一个结果元组时，该元组马上传送给投影操作去处理。通过将连接操作与投影操作组合起来，可以避免中间结果的创建，从而直接产生最后结果。

1. 流水线的实现

通过构造一个复杂的、将构成流水线的所有操作组合起来的操作，可以实现流水线。尽管对于各种频繁发生的情况，这个方法是可行的，但在构造一个流水线时仍希望能重用各个操作的代码。因此，流水线中的每一操作都被建模，作为系统内独立的进程或线程，它从流水化的输入中接受元组流，并产生一个元组流作为其输出。对于流水线中的每对相邻操作，我们都创建一个缓冲区来保存从一个操作传送到另一个操作的元组。

在图 12-10 的例子中，三个操作均可放入流水线。在流水线中，选择操作的结果产生后传送给连接操作，连接操作的结果产生后又传送给投影操作。由于一个操作的结果不必长时间保存，因此对内存的要求不高。然而，由于是流水线，各个操作并非总是能立即获得输入来进行处理。

流水线可按以下两种方式之一来执行：

1）需求驱动。

2）生产者驱动。

在一个需求驱动的流水线中，系统不停地向位于流水线顶端的操作发出需要元组的请求。每当一个操作收到需要元组的请求，它就计算下一个或多个元组并返回这个（些）元组。若该操作的输入不是来自流水线，则下一个（组）元组可以由输入关系中计算得到，并记载目前为止已返回了哪些元组。若该操作的部分输入来自流水线，那么该操作也为来自流水线输入的元组发出请求。通过使用来自流水线输入的元组，该操作计算输出，然后把输出传给父层。

在一个生产者驱动的流水线中，各操作并不等待元组请求，而是不停地产生元组。流水线底部的每个操作不断地产生元组并将它们放在输出缓冲区中，直到缓冲区满为止。其他层的流水线的任何操作在获得较低层的输入元组时产生输出元组，直到其输出缓冲区满为止。一旦操作使用流水线输入的一个元组，就将其从输入缓冲区中删除。无论哪种情况，一旦输出缓冲区已满，操作都必须等待，直到其父操作将元组从该缓冲区中取走而腾出空间。此时，该操作接着产生更多的元组，直到缓冲区再次满了为止。这个过程不断重复，直到该操作已产生了所有的输出元组为止。

只有当一个输出缓冲区已满，或一个输入缓冲区已空，需更多的输入元组用于产生输出元组时，系统才需在各操作之间切换。在一个并行处理系统中，流水线中的操作可在不同的处理器上并发执行（见第 17 章）。

使用生产者驱动的流水线方法可以被看成将数据从一棵操作树的底层往上推的过程，而使

用需求驱动的流水线方法可被看成从顶端将数据拉上操作树的过程。需求驱动的流水线中的每个操作可以迭代算子实现，迭代算子提供以下函数：open、next及close。调用open后，对next的每次调用返回该操作的下一个输出元组。该操作接着又调用next，获取所需输入元组。函数close告知迭代算子不再需要元组了，迭代算子维护两次调用之间的执行状态，这样下一个next调用请求可以获取下一个（组）结果元组。迭代算子的实现细节留作练习（习题12.18）。需求驱动的流水线方法比生产者驱动的流水线方法使用更广泛，因为它更容易实现。

2. 流水线的执行算法

给定一个连接运算，其左端输入来自流水线。由于是流水线输入，处理连接运算所需的输入不能一下子全部获得。这种不可获性限制了所能使用的连接算法。例如，归并连接在其输入未排序时是不能使用的，因为在没有获得全部元组之前对一个关系进行排序是不可能的，这样实际上就将流水线方法变成了实体化方法。不过，使用索引嵌套循环连接算法是可以的，在获取连接左边所需元组后，可以利用这些元组对右边关系进行索引，从而得到连接结果中的元组。这个例子表明，一个操作所用算法的选择与流水线技术的选择不是相互独立的。

对可用执行算法的约束对流水线而言是一个限制因素。因此，尽管流水化有明显优点，但有时实体化方法代价反而更低。假设 r 与 s 要进行连接，其中 r 是来自流水线的输入。如果索引嵌套循环连接用于支持流水线，则对于流水线输入关系的每一个元组可能都需要一次磁盘访问。该技术的代价是 $n_r * HT_i$，其中 HT_i 是 s 上索引的高度。若用实体化方法，则将关系 r 写出的代价是 b_r。若采用索引技术如散列连接，则执行连接的代价为 $3(b_r + b_s)$。当 n_r 比 $4b_r + 3b_s$ 大得多时，实体化方法的代价较小。

为了有效地利用流水技术，执行算法的使用必须满足即使在操作输入接收元组过程中也能产生输出元组。可以区分以下两种情形：

1）连接运算的输入中只有一个来自流水线。

2）连接运算的两个输入均来自流水线。

如果连接运算只有一个流水线输入，那么索引嵌套循环连接算法是一个自然选择。如果流水线输入的元组按连接属性排序，并且连接条件是等值连接，则也可使用归并连接。若流水线输入作为查找用关系，则可用混合散列连接。然而，不在第一分划中的元组只有在获得整个流水线输入关系后才能输出。当非流水线输入可被内存所容纳，或至少大部分元组可装入内存时，混合散列连接算法才会很有用。

若两个输入均为流水线输入，连接算法的选择就受到限制。当两个输入均在连接属性上排序且连接条件是等值连接时，可用归并连接，同时这时还可以采用流水线连接算法，如图12-11所示。该算法假设两个关系 r、s 的输入元组均为流水线输入。两个关系的可用输入元组在同一队列中排队等待处理。特殊的队列项 End_r、

```
done_r := false;
done_s := false;
r := ∅;
s := ∅;
result := ∅;
while not done_r or not done_s do
  begin
    if 队列空 then 等待，直到队列非空;
    t := 队列中的头一项;
    if t = End_r then done_r := true
      else if t = End_s then done_s := true
        evse if t 属于输入关系 r then
          begin
            r := r ∪ {t};
            result := result ∪ ({t} ⋈ s);
          end
          else/* t 属于输入关系 s */
            begin
              s := s ∪ {t};
              result := result ∪ (r ⋈ {t});
            end
end
```

图 12-11 流水连接算法

End。在 r、s 所需的全部元组已产生后被插入队列中，作为文件结束标记。为有效执行算法，应当在 r、s 上建立适当索引。当元组添加到 r、s 中时，相应的索引也必须更新。

12.9　关系表达式的转换

至此，我们已研究了计算扩展关系代数运算的算法，并对其代价做了估计。前面提到，一个查询可以用多种方式表示，不同表示方式有不同的执行代价。本节不采用给定的关系表达式，而是考虑其等价表达式。

12.9.1　表达式的等价性

考虑查询“找出在 Brooklyn 的任何分支机构中开设了帐户的所有客户的姓名”。这个查询关系表达式为

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch \bowtie (account \bowtie depositor)))$$

该表达式需构造一个很大的中间关系：$branch \bowtie account \bowtie depositor$。然而，我们只对这个关系的少数元组感兴趣，即那些与 Brooklyn 的分支机构有关的元组，并且只对这个关系的 6 个属性中的 1 个感兴趣。由于我们只关心那些位于 Brooklyn 的分支机构的元组，因而没有必要考虑不满足 *branch-city* = “Brooklyn” 条件的元组。通过减少要存取的 *branch* 关系的元组数，也就减少了中间结果的大小。现在将查询表示为如下的关系代数表达式：

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)) \bowtie (account \bowtie depositor))$$

它与我们前面的表达式等价，但产生的中间关系较小。初始表达式以及变换后表达式的图形化表示在图 12-12 中。

给定一个关系代数表达式，查询优化器的任务是产生一个查询执行计划，该计划能获得与原关系表达式相同的结果，但其代价是能求得该结果的所有计划中最小的，或至少不比最小代价大多少。

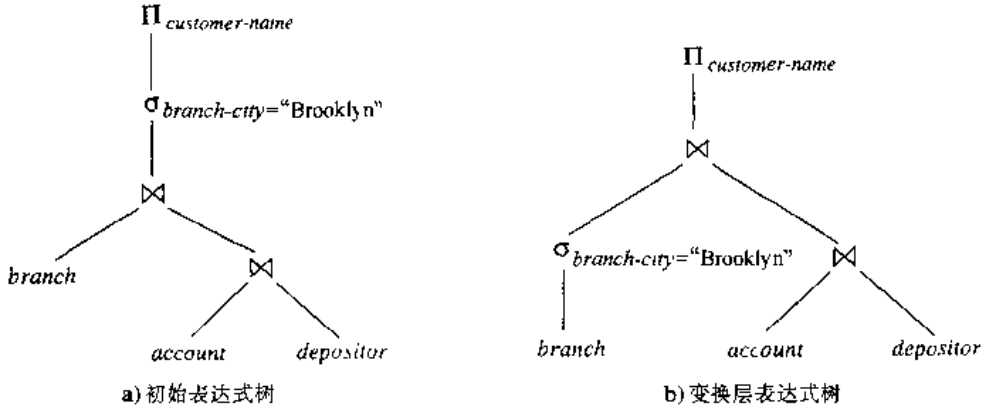


图 12-12　等价表达式

为寻找代价最小的查询执行计划，优化器需要产生一些能和给定表达式得到相同结果的候选计划，从中选择代价最小的一个。查询执行计划的产生有两步：1）产生逻辑上与给定表达式等价的表达式；2）对所产生的表达式做不同方式的注释，产生候选查询计划。查询优化器中这

两步是交叉的，产生一些表达式并注释，然后又产生一些表达式并注释，依此类推。

要完成第一步，查询优化器需要产生与给定表达式等价的表达式，这是通过等价规则来进行的。等价规则说明如何将一个表达式转换成逻辑上等价的另一个表达式。接下来我们来讲述这些规则。12.10节讲述如何选择一个查询执行计划。我们可以基于执行计划的代价估计来进行选择。由于代价是估计值，所选计划未必就是代价最小的计划，但是，只要估计值较准确，该计划很可能就是代价最小的计划，或至少不比最小值大多少。这种优化称为基于代价的优化，在12.10.2节中讲述。

12.9.2 等价规则

等价规则顾名思义就是指两种不同形式的表达式是等价的，我们可以将其中一个转换成另一个表达式，而又保持等价。所谓保持等价是指两个表达式所产生的结果关系具有相同的属性集以及相同的元组集，尽管其属性出现的次序可以不同。优化器利用等价规则将表达式转换成逻辑上等价的另一表达式。

下面列出关系代数表达式的一些通用等价规则，其中一些等价式在图12-13也做了说明。

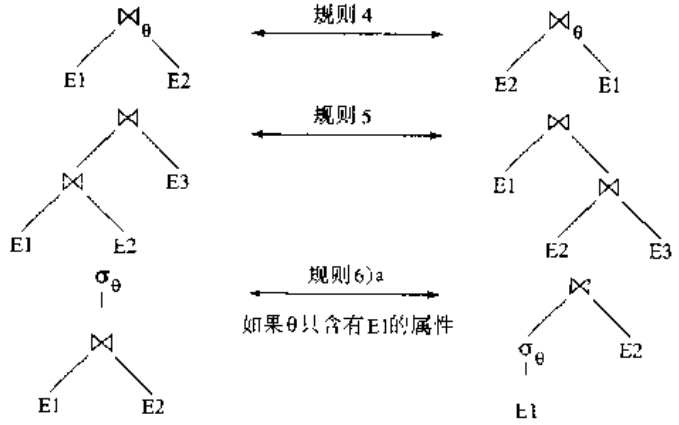


图12-13 等价表达式的图形化表示

用θ、θ_1、θ_2等表示谓词，L_1、L_2、L_3等表示属性列表，E、E_1、E_2等表示关系代数表达式。关系名r是关系代数表达式的特例，在E可以出现的地方它就可以出现。

1）合取选择运算可分解为单个选择运算的序列。该变换称为σ的级联

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2）选择运算满足交换律

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3）投影运算序列中只有最后一个运算是需要的，其余的可省略。该转换也可称为Π的级联

$$\Pi_{L_1}(\Pi_{L_2}(\cdots(\Pi_{L_n}(E))\cdots)) = \Pi_{L_1}(E)$$

4）选择可与笛卡儿积以及theta连接相结合：

a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$。该表达式就是theta连接的定义。

b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$。

5）theta连接运算满足交换律

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

前面讲过，自然连接是 theta 连接的特例，因此自然连接也满足交换律。

6）a. 自然连接运算满足结合律

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

b. theta 连接具有以下方式的结合律

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E3)$$

其中 θ_2 只涉及 E_2 与 E_3 的属性。由于任意一个条件都可为空，因此笛卡儿积运算也具有结合律。连接运算满足结合律、交换律对于在查询优化中重排连接顺序是很重要的。

7）选择运算在下面两个条件下对 theta 连接运算具有分配律：

a. 当选择条件 θ_0 的所有属性只涉及参与连接运算的表达式之一（E_1）时：

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$$

b. 当选择条件 θ_1 只涉及 E_1 的属性，选择条件 θ_2 只涉及 E_2 的属性时：

$$\sigma_{\theta_1 \wedge \theta_2} (E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

8）投影运算对 theta 连接运算具有分配律：

a. 令 L_1、L_2 分别是 E_1、E_2 的属性。假设连接条件 θ 只涉及 $L_1 \cup L_2$ 中的属性，则

$$\Pi_{L_1 \cup L_2} (E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1} (E_1)) \bowtie_{\theta} (\Pi_{L_2} (E_2))$$

b. 考虑连接 $E_1 \bowtie_{\theta} E_2$。令 L_1、L_2 分别是 E_1、E_2 的属性，L_3 是 E_1 中出现在连接条件 θ 中但不在 $L_1 \cup L_2$ 中的属性，L_4 是 E_2 中出现在连接条件 θ 中但不在 $L_1 \cup L_2$ 中的属性，那么

$$\Pi_{L_1 \cup L_2} (E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2} ((\Pi_{L_1 \cup L_3} (E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4} (E_2)))$$

9）集合运算并与交满足交换律

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

集合差运算不满足交换律。

10）集合运算并与交满足结合律

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11）选择运算对并、交、差运算具有分配律

$$\sigma_p(E_1 - E_2) = \sigma_p(E_1) - \sigma_p(E_2)$$

同样，这一等价式在将“－”替换成∪或∩时也成立。进一步有

$$\sigma_p(E_1 - E_2) = \sigma_p(E_1) - E_2$$

这一等价式将“－”替换成∩时也成立，但将“－”替换成∪时不成立。

12）投影运算对并运算具有分配律

$$\Pi_L (E_1 \cup E_2) = (\Pi_L (E_1)) \cup (\Pi_L (E_2))$$

以上列表只是等价式的一部分。有关扩展关系代数运算符的更多等价式如外连接、聚集等在习题中讨论。

12.9.3　变换的一些例子

下面举例说明等价式的用法。还用银行这个例子，其关系模式如下：

$$Branch\text{-}schema = (branch\text{-}name,\ branch\text{-}city,\ assets)$$
$$Account\text{-}schema = (branch\text{-}name,\ account\text{-}number,\ balance)$$
$$Depositor\text{-}schema = (customer\text{-}name,\ account\text{-}number)$$

关系 *branch*、*account*、*depositor* 是以上模式的实例。

前面的例子中，表达式

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch \bowtie (account \bowtie depositor)))$$

被转换成以下表达式：

$$\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch)) \bowtie (account \bowtie depositor))$$

转换后表达式等价于原始关系代数表达式，但却产生较小的中间关系。这一转换可用规则7)a来实现。记住，规则只是说两个表达式等价，而未表明孰优孰劣。

对一个查询或查询的一部分，可以连续使用多个等价规则。例如，假设我们对前面提到的查询加以修改，将注意力集中到那些存款余额超过 $1000 的客户，新的关系代数查询可写为：

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$$

我们不能把选择谓词直接作用到 *branch* 关系上，因为谓词牵涉到 *branch* 和 *account* 两个关系中的属性。然而，我们可以首先用规则 6)a(自然连接的结合律）把连接 $branch \bowtie (account \bowtie depositor)$ 转换成 $(branch \bowtie account) \bowtie depositor$：

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"} \wedge balance > 1000}((branch \bowtie account) \bowtie depositor))$$

然后使用规则 7)a 将查询重写为

$$\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

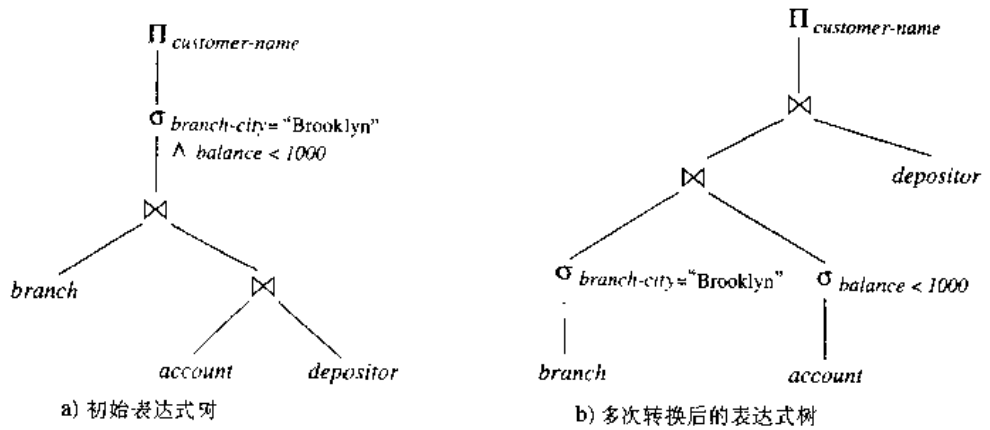
我们再看该表达式的选择子表达式。利用规则 1)，我们可以把选择条件分解为两个选择条件，得到以下子表达式：

$$\sigma_{branch\text{-}city = \text{"Brooklyn"}}(\sigma_{balance > 1000}(branch \bowtie account))$$

上述两个表达式均是用于选择满足 *branch-city* = “Brooklyn” 及 *balance* > 1000 条件的元组。不过，后一表达式形式可以让我们应用“尽早执行选择”这条规则，得到如下子表达式：

$$\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch) \bowtie \sigma_{balance > 1000}(account)$$

经过上述转换后的最终表达式及初始表达式的图形化表示如图 12-14 所示。实际上还可以



a) 初始表达式树　　b) 多次转换后的表达式树

图 12-14　多次转换

应用规则7)b直接得到最后表达式，而不必用规则1）将选择分解为两个选择。事实上，规则7)b本身可由规则1)及7)a导出。

若等价规则集合中任意一条规则都不能由其他规则结合起来导出，则这组等价规则称为最小的等价规则集。上述表明12.9.2节中的等价规则集不是最小的。

对前面提到的查询，现在考虑如下形式：

$$\Pi_{customer\text{-}name}\ (\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch)\ \bowtie\ account)\ \bowtie\ depositor)$$

当计算子表达式

$$(\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch)\ \bowtie\ account)$$

时，我们得到具有如下模式的关系：

$$(branch\text{-}name,\ branch\text{-}city,\ assets,\ account\text{-}number,\ balance)$$

通过使用等价规则8)a及8)b先做投影运算，可从模式中去除几个属性。要保留的属性是那些出现在查询结果中或在后续运算中要处理的属性。通过去除不必要的属性，可以减少中间结果的列数，从而减少中间结果的大小。本例 *branch* 与 *account* 的连接中我们只需属性 *account-number*，因此可将表达式修改为

$$\Pi_{customer\text{-}name}\ (\ (\Pi_{account\text{-}number}\ (\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch))\ \bowtie\ account))\ \bowtie\ depositor)$$

投影 $\Pi_{account\text{-}number}$ 减少了中间连接结果的大小。

12.9.4　连接的次序

一个好的连接运算顺序对于减小临时结果的大小是很重要的，因此大部分查询优化器在连接次序上花了较多功夫。正如在第3章及在等价规则6)a中提到的，自然连接运算满足结合律。所以，任意给关系 r_1、r_2、r_3

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

虽然这两个表达式等价，但它们各自的代价可能不同。我们再来考虑表达式

$$\Pi_{customer\text{-}name}\ (\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch))\ \bowtie\ account\ \bowtie\ depositor)$$

可以选择先计算 $account \bowtie depositor$，然后将结果与 $\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch)$ 相连接。不过，$account \bowtie depositor$ 可能是一个很大的关系，因为每个帐户均有一个元组。与此相反，$(\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch)) \bowtie account$ 很可能是一个小关系。要理解这一点，我们应当注意到银行拥有大量分布广泛的分支机构，但很可能只有一小部分银行客户在位于Brooklyn的分支机构中设立了帐户。因此，这个表达式对应Brooklyn居民的每个帐户有一个元组。这样，需要保存的临时关系比先计算 $account \bowtie depositor$ 连接时要小。

执行查询还需考虑其他因素。我们不太关心属性在连接中出现的次序，因为在显示结果之前改变属性的显示顺序是很容易的。因此，对于任何关系 r_1、r_2

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

即自然连接满足交换律（规则5））。

利用自然连接满足结合律与交换律（规则5）与6））的特性，可以将前面的关系代数表达式重写为

$$\Pi_{customer\text{-}name}((\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))\ \bowtie\ depositor)\ \bowtie\ account)$$

也就是说，我们可以先计算

$$(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch)) \bowtie depositor$$

然后将结果与 *account* 连接。但是注意，*Branch* 模式与 *Depositor* 模式没有公共属性，所以以上连接为笛卡儿积。若 Brooklyn 有 b 个分支机构，*depositor* 关系中有 d 个元组，该笛卡儿积将产生 $b * d$ 个元组，*depositor* 的每一个元组与 *branch* 的每一个元组所形成的元组对均为结果中的一个元组(该计算没有考虑 *depositor* 中的帐户是否由 *branch* 中的分支机构维护)。因此，看来该笛卡儿积将产生较大的临时关系，所以我们不采用这一策略。不过，如果用户输入如上表达式，我们可以用自然连接的结合律与交换律把这个表达式转换成前面我们所用的更有效的表达式。

12.9.5 等价表达式的枚举

查询优化器使用等价规则系统地产生与给定表达式等价的表达式。概念上，处理过程如下。给定一个表达式，只要其中有任何子表达式与等价规则的某一边相匹配就产生一个新的表达式，其中子表达式被替换成规则中与它相匹配的那一边。该处理过程不断进行下去，直到不再有新表达式产生为止。

上述处理方式无论在时间上还是在空间上代价都很大。对空间的要求可以按如下方式减小。如果使用等价规则把表达式 E_2 转换成 E_1，则 E_1 与 E_2 在结构上是类似的，并且具有一些相同的子表达式。表达式可以采用一定的表示技术，让有公共子表达式的表达式指向共享子表达式，这样可以大大减少对空间的需求。许多查询优化器都采用了这种技术。

此外，用等价规则产生所有的表达式并不总是必要的。若把执行代价估计考虑进去，优化器可以避免检查某些表达式，正如我们将在下节中看到的那样。利用如上所述的各种技术，我们可以减少优化所需时间。

12.10 选择执行计划

产生表达式只是查询优化过程的一部分。表达式中的每个运算可用不同的算法实现，执行计划准确定义每个运算使用什么算法以及如何协调各运算的执行。图 12-15 表示了图 12-14 中所示表达式的一个可能的执行计划。正如我们所知，每个关系运算可以有许多不同的算法，从而有多个可选的执行计划。另外，计划中还要决定是否采用流水技术。图 12-15 中，从选择运算到归并连接运算的边上标记了“流水线”，当选择运算产生按连接属性排序的输出时，流水技术就是可行的。当 *branch* 与 *account* 上的索引为按 *branch-name* 排序的索引属性保存等值记录时，选择运算就产生按连接属性排序的输出。

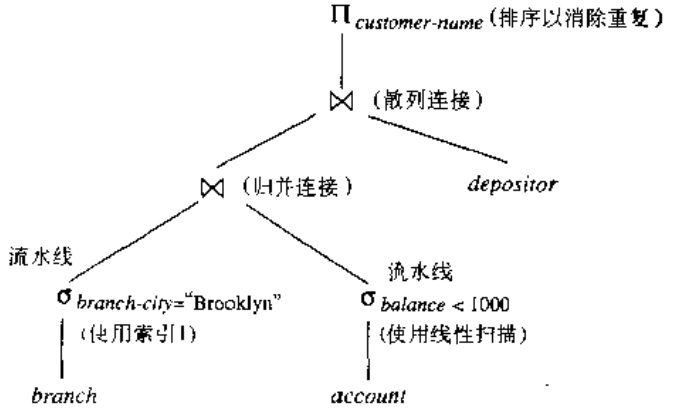


图 12-15 一个执行计划

12.10.1　执行技术的相互作用

为查询表达式选择执行计划的方法之一是简单地为每个运算选择一个代价最小的算法。在表达式树中，只要层次较低的运算比层次较高的运算先执行，我们就可选择任意的运算执行顺序。

但是，为每个运算独立地选择最小代价的算法不一定总是好事。尽管在给定层次上归并连接比散列连接代价大，但它却产生有序的输出，从而使以后的运算（如消除重复、交或另一个归并连接）代价变小。同样，索引嵌套循环连接可以提供把结果流水地传给下一个运算的机会，因此，即使它不是实现连接代价最小的算法，却也可能很有用。要选择一个总体上最佳的算法，有时甚至也要考虑对于单个运算而言并不是最优的算法。

所以，除了考虑一个查询的候选表达式之外，还必须为表达式中的每一个运算考虑可选的算法。可用类似等价规则的规则来定义哪些是一个运算可能的算法，包括是否将某个运算与另一个表达式构成流水线。我们可用这些规则系统地评估给定表达式的所有查询执行计划。

给定一个执行计划，可用本章前面讨论过的技术来估计其代价，那样仍然存在为一个查询选择最佳执行计划的问题。有两大类方法：第一种是搜索所有计划，按基于代价的方式选择最佳计划；第二种是使用启发式方法选择计划。接下来将讨论这些方法，实际中的查询优化器将两者结合在一起。

12.10.2　基于代价的优化

基于代价的优化器通过使用等价规则为给定的查询语句产生一系列查询执行计划，并选择其中代价最小的一个。对于一个复杂的查询，等价于给定查询计划的不同查询计划可能很多。为说明这一点，考虑表达式

$$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

其中连接运算没有指定任何顺序。当 $n=3$ 时，有 12 种不同的连接顺序：

$$r_1 \bowtie (r_2 \bowtie r_3) \quad r_1 \bowtie (r_3 \bowtie r_2) \quad (r_2 \bowtie r_3) \bowtie r_1 \quad (r_3 \bowtie r_2) \bowtie r_1$$
$$r_2 \bowtie (r_1 \bowtie r_3) \quad r_2 \bowtie (r_3 \bowtie r_1) \quad (r_1 \bowtie r_3) \bowtie r_2 \quad (r_3 \bowtie r_1) \bowtie r_2$$
$$r_3 \bowtie (r_1 \bowtie r_2) \quad r_3 \bowtie (r_2 \bowtie r_1) \quad (r_1 \bowtie r_2) \bowtie r_3 \quad (r_2 \bowtie r_1) \bowtie r_3$$

一般而言，对于 n 个关系，有 $(2(n-1))!/(n-1)!$ 个不同的连接顺序。将该式的计算留作练习（习题 12.23）。就少量关系的连接而言，其数目是可以接受的。例如 $n=5$，此数是 1680。然而，当 n 增加时，此数迅速增长。当 $n=7$，此数变为 665 280；当 $n=10$，此数大于 1760 亿！

所幸的是，我们不必产生与给定表达式等价的所有表达式。例如，假设想找到以下表达式的最佳连接顺序：

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

该表达式表示 r_1、r_2、r_3 首先进行连接（以一定顺序），其结果再与 r_4、r_5 以某种顺序进行连接。计算 $r_1 \bowtie r_2 \bowtie r_3$ 有 12 种不同的连接次序，其结果再与 r_4、r_5 进行连接时又有 12 种顺序，因此，似乎需要检查 144 种连接顺序。然而，一旦给子集 $\{r_1, r_2, r_3\}$ 找到了最佳连接顺序，可以用此顺序进一步与 r_4、r_5 进行连接，从而舍弃 $r_1 \bowtie r_2 \bowtie r_3$ 中代价较大的连

接顺序。这样，我们就不必一一检查 144 种连接顺序，而只需检查 12 + 12 = 24 种顺序。利用这种想法，可以开发一个动态编程算法来寻找最佳连接顺序，该算法将大大减少需检查的表达式数目。给定 n 个关系的集合，该算法计算每个子集的最佳连接顺序（参见习题 12.24）。

实际上，由连接 $r_1 \bowtie r_2 \bowtie r_3$ 产生的元组的顺序对于寻找总体上最佳的连接顺序也是很重要的，因为它可以进一步影响连接的代价（如使用归并连接时）。若某个具体的元组排序顺序对后面的运算有用的话，我们称该顺序是一个感兴趣的排序顺序。例如，连接 $r_1 \bowtie r_2 \bowtie r_3$ 所产生的结果在与 r_4 或 r_5 的公共属性上进行排序是有用的。而此结果若对 r_1 与 r_2 的公共属性进行了排序，就没有什么用处。在计算 $r_1 \bowtie r_2 \bowtie r_3$ 时，使用归并连接可能比使用其他连接技术代价大，但其输出结果可能是感兴趣的排序顺序。

由此可知，仅为 n 个给定关系集合的每个子集找出最佳连接顺序是不够的，应当为每个子集、该子集连接结果的每个感兴趣的排序顺序找出最佳连接顺序。n 个关系的子集总数是 2^n，但感兴趣的排序顺序数目一般不大。因此，约有 2^n 个连接表达式需要存储。用于寻找最佳连接顺序的动态编程算法可以容易地加以扩展，用于处理排序顺序，其时间代价约为 3^n（参见习题 12.25）。

当 $n = 10$，该数约为 59 000，较之于 1760 亿个不同的连接顺序简直好多了。更重要的是，所需的存储也比原先少得多，因为对于 r_1，…，r_{10}的 1024 个子集的每一个感兴趣次序，我们只需保存一次连接顺序。虽然这两个数均随 n 迅速增长，但是通常参与连接的关系数少于 10，因而可以很容易地处理。

可以使用其他几项技术进一步减少搜索大量计划的代价。例如，在检查一个表达式的计划时，如果检查了该表达式的某部分后发现这一部分的最小代价已经比先前已检查过的某个整体表达式的执行计划的最小代价要大，那么可以终止对这个表达式的检查。同样，假设发现计算一个子表达式的所用代价最小的方法比先前已检查过的某个整体表达式的最小执行代价所需代价还大，则没有必要对包含该子表达式的任何完整表达式进行检查。通过一开始就对好的计划做启发式猜测，并对其代价做出估计，可以进一步减少需要完整考虑的执行计划的数目。这样，我们只需对少数有竞争性的计划做全面的代价分析，这些改进可以大大减少查询优化的开销。

12.10.3 启发式优化

基于代价的优化的一个缺点是优化本身的代价。虽然查询处理的代价可以通过各种巧妙的优化技术来减少，但基于代价的优化开销仍然很大。因此，许多系统用启发式方法来减少基于代价的方式中可选择的方案数。有的系统甚至将基于代价的优化方法完全弃之不用，而只使用启发式方法。

下面是一条启发式规则，该规则用于对关系代数查询进行转换。

- 尽早执行选择运算。

一个启发式优化器不验证采用本规则转换后代价是否减少，而是直接加以使用。在 12.9 节的第一个转换例子里，选择运算直接用于连接之中。

我们说这条规则是启发式的，因为它通常但不总是有助于减少代价。来看一个使用这条规则导致代价增加的例子。考虑表达式 $\sigma_\theta (r \bowtie s)$，其中 θ 只引用了 s 的属性。很明显，选择可以先于连接计算。然而，若 r 与 s 相比而言非常小，并且在 s 的连接属性上有索引，且 θ 所

引用的属性上无索引，那么先做选择很可能不是个好主意。先做选择运算意味着直接对 s 进行选择，这需要对 s 的全体元组进行一次扫描。就本例而言，先用索引计算连接然后再做选择运算，代价可能更小。

投影运算像选择运算一样可以减少关系的大小。因此，无论何时，当产生临时关系时，只要可能就立即执行投影是有好处的。这种好处告诉我们，同前面的“尽早执行选择运算”相并列的还有一条规则：

- 尽早执行投影运算。

通常选择运算优先于投影运算执行比较好，因为选择运算可能会大大减小关系，并且选择运算可以利用索引存取元组。可以用类似启发式选择规则中使用的例子来说明该启发式规则不总是有助于减少代价。

利用 12.9.2 节中讲述的等价规则，启发式优化算法将对初始查询树的成分重新排序，改进查询执行效率。下面我们概述典型的启发式优化算法所采取的步骤。你可以像前面那样将查询表达式图形化地表示为一棵树，以帮助理解启发式算法。

1）将合取选择分解为单个选择运算的序列。根据等价规则 1），这一步有助于将选择运算往查询树下层移。

2）把选择运算在查询树中下推到最早可能执行的地方。这一步使用了选择运算的可交换性和可分配性，它们在等价规则 2）、7)a、7)b 与 11)中有说明。

例如，只要有可能，$\sigma_\theta (r \bowtie s)$ 转换成 $\sigma_\theta (r) \bowtie s$ 或 $r \bowtie \sigma_\theta (s)$。尽早执行基于值的选择运算可以减少对中间结果进行排序和归并的代价。对于一个具体选择，允许重新排序的程度取决于选择条件中涉及的属性。

3）确定哪些选择运算与连接运算将产生最小的关系，即所含元组数最少的关系。通过使用连接操作的结合律，重新组织查询树，使得限制比较严格的选择运算的叶结点关系首先执行。

这一步考虑的是满足选择或连接条件的元组比例。前面讲过，限制最严的选择（即满足选择条件的元组比例最小）检索出的记录数最少。这一步依赖于等价规则 6）中所给出的二元运算结合律。

4）将其后跟有选择条件的笛卡儿积运算（规则 4）a）替换成连接运算。正如 12.9.4 节中说明的那样，笛卡儿积的实现代价很大。如计算 $r_1 \times r_2$，不仅包含了所有可能的 r_1 元组与 r_2 元组的组合对，而且在结果关系的属性中包含了 r_1、r_2 两关系的全部属性。因此，明智的做法是避免使用笛卡儿积运算。

5）将投影属性加以分解并在查询树上尽可能往下推。若有必要，可以引入新的投影运算。这一步所用的投影运算特性在等价规则 3）、8)a、8)b 与 12)中给出。

6）识别那些可用流水方式执行其运算的子树，并采用流水线方法执行之。

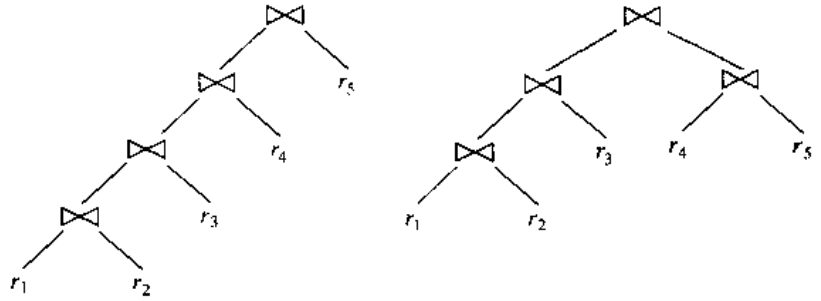
总之，上述启发式规则重新组织初始查询树表示，让可以减少中间结果的运算首先执行。早用选择减少元组数，早用投影减少属性数。启发式转换对树的重构还包括使限制最严格的选择与连接运算先于其他同类运算执行。

启发式优化接下来将经启发式规则转换产生的查询表达式映射成运算序列，以产生一系列候选执行计划。执行计划不仅包含要计算的关系运算，还包括要使用的索引、访问元组的次序、执行运算的次序。启发式优化器的存取－计划－选择阶段为每个运算选择最有效的策略。

12.10.4 查询优化器的结构

至此，我们已讲述了选择执行计划的两个基本方法。大部分实际的查询优化器将这两种方法结合起来。某些查询优化器，如 System R 优化器，并不考虑所有的连接顺序，而是只对特殊类型的连接次序进行搜索。System R 优化器仅考虑右操作数是原始关系 r_1，r_2，…，r_n 的那些连接顺序，这种连接顺序称为*左深连接顺序*。左深连接顺序用于流水线计算特别方便，因为右操作数是一个已存储的关系，因此每个连接只有一个输入来自流水线。

图 12-16 说明了左深连接树与非左深连接树的区别。全体左深连接顺序所需时间代价是 $O(n!)$，比所有连接顺序少得多。由于使用了某些优化技术，System R 优化器可以在近似 $O(2^n)$ 的时间内找到最佳连接顺序。可把这一代价同找出总体上最佳连接顺序所需时间 3^n 相比较。System R 优化器使用启发式规则在查询树中将选择与投影运算往下推。



a) 左深连接树　　b)非左深连接树

图 12-16 左深连接树

对使用辅助索引进行的扫描做代价估计时，假定每个元组的存取都会导致一次 I/O 操作。缓冲区较少时，这个假设可能是对的；但对于较大缓冲区，包含所要存取元组的页可能已在缓冲区中。Sybase 优化器对这种扫描采用了较好的代价估计技术，它考虑了包含该元组的页在缓冲区中的概率。

许多系统采用将启发式选择与候选存取计划产生集成在一起的查询优化方法。System R 及其后续的 Starburst 项目采用的方法由基于 SQL 嵌套块概念的一个层次过程组成。我们所讲述的基于代价优化技术被独立地应用到每个查询块上。Ingres 使用启发式分解方法将每个查询分解为包含的关系不多于两个的子查询集合，然后为每个子查询产生详细的存取计划。

Oracle7 优化器有两个部件：一个基于启发式规则；另一个基于代价。启发式方法使用情况大致如下。对于有 n 个连接的一个查询，启发式方法考虑 n 个按如下方式构造的执行计划。每个计划使用一个左深连接顺序，各个计划的第一个关系不相同。基于已有存取路径的排序，通过重复选择参加下一个连接的最佳关系，启发式方法分别为 n 个执行计划构造连接顺序。基于已有存取路径的每个连接选择嵌套循环连接算法或排序归并连接算法。最后，用启发式方法从 n 个执行计划中选一个，该选择基于使内层关系上没有索引的嵌套循环连接个数最小，同时也基于排序归并连接的个数。

SQL 本身的复杂性给查询优化器带来不少困难，特别是 SQL 子查询很难翻译成关系代数。很多优化器，如 System R 与 Starburst，试图重写 SQL 查询，只要有可能就把子查询转换成连接。当不能转换时，这些子查询就作为独立的表达式，并独立地进行优化。被选中的执行计划通过专门的执行算法组合起来。同样，对于复合 SQL 查询语句（使用了∩、∪、－运算），每

个成分均分别进行优化，然后所有执行计划结合在一起形成总的执行计划。

即使使用启发式方法，基于代价的查询优化仍给查询处理带来不少开销。然而，增加的开销通常小于查询执行时间的节省，这里查询执行时间主要花在慢速的磁盘存取上。好的计划与坏的计划在查询执行时间上差别可能很大，因此查询优化非常重要。如果应用中有许多经常运行的操作，那么其中的查询可以优化一次，选中的查询计划在以后每一次运行时使用。这样的应用中查询优化所能带来的代价节省更为明显。因此，大部分商业系统都包含了相当复杂的优化器。文献注解给出了对实际数据库系统查询优化器进行讲解的参考资料。

12.11　总结

对于一个查询，系统首先要做的事就是把它翻译成系统内部的表示形式。对于关系数据库系统而言，内部形式通常是基于关系代数的。在产生查询内部形式过程中，语法分析器检查用户查询语句的语法，验证出现在查询语句中的关系名是否是数据库中的关系名等等。如果查询语句是用视图表达的，语法分析器就把所有对视图名的引用替换成计算该视图的关系代数表达式。

给定一个查询，通常有许多计算它的方法。将用户输入的查询语句转换成等价的、执行效率更高的查询语句是系统的责任。优化过程，更准确地说，即改进查询处理策略的过程称为查询优化。

复杂查询语句的执行涉及多次磁盘存取。由于磁盘存取比计算机内存存取及计算机系统的CPU速度慢，因而花相当的处理时间来选择一种最小磁盘存取次数的方法是值得的。

为一个操作选择的策略取决于每个关系的大小以及各列上值的分布。为了能在可靠的信息基础上选择策略，数据库系统为每个关系 r 保存统计信息。这些统计信息包括：

- 关系 r 中的元组数。
- 关系 r 的一个记录（元组）按字节计算的大小。
- 关系 r 中某个属性上出现的不同值的个数。

有了这些统计数据，就可以估计各种运算的结果大小以及执行代价。当存在有助于查询处理的多个索引时，关系的统计信息尤其有用。这些结构的存在对选择查询处理策略大有帮助。

对于包含简单选择的查询语句，可以通过线性扫描、二分搜索或索引来处理。通过计算简单选择结果的并和交，可以处理复杂的选择操作。使用外部归并排序算法，可以对大于内存的关系进行排序。涉及自然连接的查询语句可以有多种处理方法，如何处理取决于是否有索引以及关系的物理存储形式。若连接的结果几乎与两个关系的笛卡儿积相当，则采用块嵌套循环连接策略较好。若存在索引，则可用索引嵌套循环连接。若两个关系已排序，则归并连接更可取。有些情况下连接计算前先对关系排序可能有好处（为了能使用归并连接算法）。为了使用更有效的连接策略而建立临时索引也可能是有益的。散列连接算法把关系划分成多个部分，使每个部分都能被内存所容纳。划分过程是通过连接属性上的散列函数来进行的，这样对应的分划对可以独立地进行连接。散列与排序是可并用的，因为许多操作如消除重复、聚集、连接、外连接等均可用散列或排序来实现。

每个关系代数表达式表示了一个特定的运算序列。选择查询策略的第一步是找出与给定查询等价，但执行代价估计较小的关系代数表达式。将表达式转换成其等价式可以使用许多等价规则。使用这些等价规则系统地产生与给定查询语句等价的所有表达式，然后选择其中执行代

价最小的一个。对于每个表达式，可以用类似的规则产生各种候选执行计划，然后选择所有表达式中代价最小的执行计划。多种优化技术可用来减少所需产生的候选表达式及计划的个数。

采用启发的办法减少需要考虑的计划个数，从而减少优化的代价。关系代数转换的启发式规则包括“尽早执行选择运算”、“尽早执行投影”及“避免使用笛卡儿积”等。

习题

12.1 查询处理中什么时候进行优化？

12.2 为什么不应要求用户显式地选择查询处理计划？是否存在希望用户知道两个候选查询处理计划代价的情形？请做出解释。

12.3 就书中所举银行数据库例子考虑下面的 SQL 语句：

select $T.branch\text{-}name$
from $branch\ T$, $branch\ S$
where $T.assets > S.assets$ **and** $S.branch\text{-}city =$ “Brooklyn”

写一个与此等价的、高效的关系代数表达式，并解释为什么要选这样一个表达式。

12.4 考虑关系 r_1 (A, B, C)、r_2 (C, D, E) 及 r_3 (E, F)，它们的主码分别为 A、C、E。假设 r_1 有 1000 个元组，r_2 有 1500 个元组，r_3 有 750 个元组。估计 $r_1 \bowtie r_2 \bowtie r_3$ 的大小，给出一个有效地计算这个连接的策略。

12.5 考虑习题 12.4 中的关系 r_1 (A, B, C)、r_2 (C, D, E) 及 r_3 (E, F)。假设除了整个模式外没有主码。令 $V(C, r_1)$ 为 900、$V(C, r_2)$ 为 1100、$V(E, r_2)$ 为 50，$V(E, r_3)$ 为 100。假设 r_1 有 1000 个元组，r_2 有 1500 个元组，r_3 有 750 个元组。估计 $r_1 \bowtie r_2 \bowtie r_3$ 的大小，给出一个有效地计算这个连接的策略。

12.6 聚集索引可以提供比非聚集索引更快的数据存取。虽然聚集索引有此好处，但什么时候我们必须建立非聚集索引？为什么？

12.7 与 B^+ 树索引相比，散列索引的优缺点是什么？已有索引类型会怎样影响查询处理策略的选择？

12.8 为了简单起见，本题假设一块中只有一个元组，内存最多容纳 3 页。当使用排序归并算法时，给出下列元组按第一个属性排序各趟所产生的归并段文件：(kangaroo, 17)、(wallaby, 21)、(emu, 1)、(wombat, 13)、(platypus, 3)、(lion, 8)、(warthog, 4)、(zebra, 11)、(meerkat, 6)、(hyena, 9)、(hornbill, 2)、(baboon, 12)。

12.9 设关系 r_1 (A, B, C)、r_2 (C, D, E) 有如下特性：r_1 有 20 000 个元组，r_2 有 45 000个元组，一块中可容纳 25 条 r_1 元组，或 30 条 r_2 元组。使用以下连接策略计算 $r_1 \bowtie r_2$：

(a) 嵌套循环连接。

(b) 块嵌套循环连接。

(c) 归并连接。

(d) 散列连接。

估计每种方法所需的块存取数。

12.10　修改混合连接算法以处理这种情形：做连接的两个关系在物理上未排序，但两个关系在连接属性上有辅助索引。

12.11　如果索引是辅助索引并且在连接属性上多个元组有相同值，则 12.6.4 节中讲述的索引嵌套循环连接算法效率不高，为什么？找出一种利用排序减少内层关系元组检索代价的方法。在什么条件下该算法比混合连接算法更有效？

12.12　估计你在习题 12.10 中给出的计算 $r_1 \bowtie r_2$ 的方法所需的块存取数，其中 r_1、r_2 按照习题 12.9 的定义。

12.13　考虑习题 12.9 所给关系 r_1、r_2，再加上关系 r_3 (E, F)。假设 r_3 有 30 000 个元组，一块中可容纳 40 个 r_3 元组。估计 12.6.7 节中用于计算 $r_1 \bowtie r_2 \bowtie r_3$ 连接的三种策略的代价。

12.14　令 r、s 是没有索引的两个关系，并且两个关系也没有排序。假设内存无限大，那么计算 $r \bowtie s$ 代价最小（就 I/O 操作而言）的方法是什么？该算法需多大内存？

12.15　假设关系 *branch* 在 *branch-city* 属性上有 B^+ 树索引，此外别无其他索引。处理下列包含否定选择条件的选择操作的最佳方法是什么？

(a) $\sigma_{\neg(branch\text{-}city < \text{"Brooklyn"})}(branch)$

(b) $\sigma_{\neg(branch\text{-}city = \text{"Brooklyn"})}(branch)$

(c) $\sigma_{\neg(branch\text{-}city < \text{"Brooklyn"} \vee assets < 5000)}(branch)$

12.16　假设关系 *branch* 在 (*branch-name*, *branch-city*) 属性上有 B^+ 树索引。处理下列选择操作的最佳方法是什么？

$$\sigma_{(branch\text{-}city < \text{"Brooklyn"}) \wedge (assets < 5000) \wedge (branch\text{-}name \neq \text{"Downtown"})}(branch)$$

12.17　12.6.6 节中讲述的散列连接算法用以计算两个关系的自然连接。请说明如何扩展散列连接算法，以计算自然左外连接、自然右外连接和自然全外连接。

(提示：在散列索引中保存每个元组的额外信息，判断在查找用关系中是否存在与散列索引中元组相匹配的元组。）将你的算法用到 *customer* 与 *depositor* 上。

12.18　用伪码书写实现索引嵌套循环连接的迭代算子，其中外层关系是流水线输入。在你的伪码中使用标准的迭代函数，说明迭代函数在两次调用之间应保存什么状态信息。

12.19　证明以下等价式成立。解释如何用它们提高某些查询的效率。

(a) $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$

(b) $\sigma_\theta ({}_A\mathcal{G}_F (E)) = {}_A\mathcal{G}_F (\sigma_\theta (E))$，其中 θ 仅使用 A 的属性。

(c) $\sigma_\theta (E_1 \mathbin{⟕} E_2) = \sigma_\theta (E_1) \mathbin{⟕} E_2$，其中 θ 仅使用 E_1 的属性。

12.20　使用 12.9.2 节中的等价规则，证明如何通过一系列转换导出下列等价式：

(a) $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3} (E) = \sigma_{\theta_1} (\sigma_{\theta_2} (\sigma_{\theta_3} (E)))$

(b) $\sigma_{\theta_1 \wedge \theta_2} (E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1} (E_1 \bowtie_{\theta_3} (\sigma_{\theta_2} (E2)))$，其中 θ_2 仅使用 E_2 的属性

12.21　对于下面每对表达式，给出关系实例说明表达式不等价：

(a) $\Pi_A (R - S)$ 与 $\Pi_A (R) - \Pi_A (S)$

(b) $\sigma_{B<4} ({}_A\mathcal{G}_{max(B)} (R))$ 与 ${}_A\mathcal{G}_{max(B)} (\sigma_{B<4} (R))$

(c) 若上式 *max* 全改为 *min*，等价吗？

(d) $(R \mathbin{⟕} S) \mathbin{⟕} T$ 与 $R \mathbin{⟕} (S \mathbin{⟕} T)$。换句话说，自然左外连接不满足结合律。

(提示：假设三个关系的模式分别为 R $(a, b1)$、S $(a, b2)$，T $(a, b3)$。)

(e) σ_θ $(E_1 \supset\!\bowtie E_2)$ 与 $E_1 \supset\!\bowtie \sigma_\theta$ (E_2)，其中 θ 仅使用 E_2 的属性

12.22 SQL 语言允许关系有重复元组（见第 4 章）。

(a) 对有重复元组的关系，给出基本关系代数运算 σ、Π、$\times$、$\bowtie$、$-$、$\cup$ 和 $\cap$ 在这种情况下的定义（多重集版本的定义），定义应与 SQL 一致。

(b) 等价规则 1) ~7) 中哪一个满足 a 中定义的多重集版本的关系代数运算。

12.23 证明：对于 n 个关系，有 $(2(n-1))!/(n-1)!$ 种不同的连接次序。

12.24 给出一种寻找 n 个关系连接的最佳连接次序的动态编程算法。假设只有一个感兴趣的排序次序。

(提示：在已知 n 个关系的每个子集的最小代价连接次序的情况下，说明如何导出 n 个关系的最小代价连接次序。)

12.25 证明最小代价连接次序的时间是 $O(3^n)$，假设保存和查看关系集合的有关信息（如该集合的最佳连接次序以及该连接次序的代价）所需时间是常量（如果你感觉做本题有困难，至少证明更宽松的时间界限 $O(2^{2n})$。）

12.26 如果像 System R 优化器那样只考虑左深连接树，证明寻找最有效连接次序的时间大约为 2^n。这里假定只有一个感兴趣的排序次序。

12.27 如果只要两个表达式等价，其中一个就可以通过使用一系列等价规则从另一个表达式导出，则这组等价规则是完备的。我们在 12.9.2 节中讲述的那一组等价规则完备吗？

(提示：考虑等式 $\sigma_{3=5}(r) = r$。)

文献注解

查询处理器必须对查询语言中的语句做语法分析，并且必须将它们转换成某种内部表示。查询语言的语法分析与传统编程语言的语法分析几乎没有区别。大多数有关编译器的书籍中都覆盖了主要的语法分析技术，其中包括 Aho 等 [1986] 和 Tremblay 与 Sorenson [1985]。这两本书还从编程语言的角度谈到了优化。

Knuth [1973] 给出了关于外排序算法的极好描述，包括创建大小（平均来说）为内存大小两倍的初始归并段文件的有关改进。根据 70 年代中期所进行的性能研究，数据库系统只使用嵌套循环连接和归并连接。这些同 System R 的开发相联系的研究表明，嵌套循环连接或归并连接几乎总能提供最佳连接方式 [Blasgen 和 Eswaran 1976]。因此，System R 中实现的连接算法只有这两种。

然而，System R 的研究中没有包括对散列连接算法的分析。现在，散列连接被认为是效率很高的。散列连接最初为并行数据库系统而设计。使用散列连接方法的实验系统在 80 年代中期开发出来，其中最有名的是 Grace database machine（Kitsuregawa 等 [1983]、Fushimi 等 [1986]）和 Gamma database machine（DeWitt 等 [1986, 1990]）。散列连接技术在 Kitsuregawa 等 [1983] 中描述，而其扩展（包括混合散列连接）在 Shapiro [1986] 中描述。更近期的结果是 Zeller 和 Gray [1990] 以及 Davison 和 Graefe [1994] 所描述的散列连接技术，这些技术可以根据所获得的内存做调整，而这一点在多个查询可同时运行的系统上非常重要。

主存数据库中的查询处理在 DeWitt 等 [1984] 以及 Whang 和 Krishnamurthy [1990] 中讨论。Kim [1982, 1984] 描述了连接策略以及对可用内存的最佳使用。对使用外连接的查询如

何处理在 Rosenthal 和 Reiner［1984］、Galindo-Legaria 和 Rosenthal［1992］以及 Galindo-Legaria［1994］中讲述。Klug［1982］讨论了对含有聚集函数的关系代数表达式的优化。关系代数运算计算复杂性的理论结果在 Gotlieb［1975］、Pecherer［1975］以及 Blasgen 和 Eswaran［1976］中。Yao［1979b］给出了各种查询执行算法的比较，Kim 等［1985］对其他方面进行了讨论。

Graefe［1993］为查询执行技术提供了一个极好的概览。此前 Jarke 和 Koch［1984］中也有查询处理技术的概览。

Selinger 等［1979］这富有启发性的书中描述了 System R 优化器的存取路径选择，System R 优化器是最早的关系查询优化器之一。Wong 和 Youssefi［1976］描述了 Ingres 查询优化器采用的一种叫做分解的技术。Starburst 查询处理在 Haas 等［1989］中描述。Oracle［1992］扼要列出了 Oracle 的查询优化。

Graefe 和 McKenna［1993］描述了 Volcano 的可扩展性和搜索高效性，Volcano 是一个基于规则的优化器。对涉及多个关系的连接做优化时，遍历所有查询计划是不现实的，因此有人提出了不需检查所有可能计划的随机搜索技术。Ioannidis 和 Wong［1987］、Swami 和 Gupta［1988］以及 Ioannidis 和 Kang［1990］给出了这一领域的成果。Ioannidis 等［1992］提出了参数化的查询优化技术，解决关系大小频繁变化时的查询处理问题。优化器在编译时先计算出一系列的计划，每个计划针对不同大小关系中的一种，然后将这些计划保存下来。执行时根据关系的实际大小选出其中之一，避免运行时做完整优化导致的开销。值的非均匀分布为查询结果大小及代价估计带来问题。为了解决这个问题，人们又提出了使用值分布直方图的代价估计技术。Ioannidis 和 Christodoulakis［1993］、Ioannidis 和 Poosala［1995］，以及 Poosala 等［1996］给出了这一领域的成果。

SQL 语言给查询优化带来了一些挑战，包括重复值和空值，还有嵌套子查询的语义。将关系代数扩展至允许重复的情况在 Dayal 等［1982］中描述。嵌套子查询的优化在 Kim［1982］、Ganski 和 Wong［1987］以及 Dayal［1987］中讨论。Chaudhuri 和 Shim［1994］描述了对使用聚集的查询进行优化的技术。

Sellis［1988］描述了多查询的优化，这是关于将几个查询视为一组来执行时如何优化的问题。如果考虑整组查询，则有可能发现对整个组来说只需计算一次的公共子表达式。Finkelstein［1982］和 Hall［1976］讲述了对一组查询的优化以及公共子表达式的用法。

查询优化可以利用语义信息，如函数依赖和其他完整性约束。King［1981］讨论了关系数据库中的语义查询优化。Malley 和 Zdonick［1986］给出了基于知识的一种查询优化方法。Chakravarthy 等［1990］使用完整性约束来帮助进行查询优化。

面向对象数据库的查询处理在 Maier 和 Stein［1986］、Beech［1988］、Bertino 和 Kim［1989］、Cluet 等［1989］、Kim［1989］以及 Kim 等［1989］中讨论。

当查询通过视图产生时，连接的关系数通常大于计算该查询时需要连接的关系数。一些用于极小化连接的技术被人们统称为表格优化。表格的概念由 Aho 等［1979a，1979c］引入，Sagiv 和 Yannakakis［1981］做了进一步扩展。Ullman［1988］和 Maier［1983］是讨论表格的教科书。

第13章 事 务

从数据库用户的观点看，数据库上一些操作的集合通常被认为是一个独立单元。比如，顾客认为从支票帐户到储蓄帐户的资金转帐是一次单独操作，而在数据库系统中这是由几个操作组成的。显然，这些操作要么全都发生，要么由于出错而全不发生。保证这一点非常重要，我们无法接受资金从支票帐户支出而未转入储蓄帐户的情况。

事务是构成单一逻辑工作单元的操作集合。不论有无故障，数据库系统必须保证事务的正确执行——或者执行整个事务或者属于该事务的操作一个也不执行。此外，数据库系统必须以一种能避免不一致性的引入的方式来管理事务的并发执行。再看这个资金转帐的例子，如果计算顾客总金额的事务在资金转帐事务从支票帐户支出金额之前查看支票帐户余额，而在资金存入储蓄帐户之后查看储蓄帐户余额，该事务就会得到不正确的结果。

本章介绍事务处理的一些基本概念。并发事务处理和故障恢复的细节问题分别在 14 和 15 章讨论。事务处理的更进一步讨论放在第 20 章。

13.1 事务概念

事务是访问并可能更新各种数据项的一个程序执行单元。事务通常由高级数据操纵语言或编程语言（如 SQL、COBOL、C 或 Pascal）书写的用户程序的执行所引起，并用形如 begin transaction 和 end transaction 的语句（或函数调用）来界定。事务由事务开始与事务结束之间执行的全体操作组成。

为了保证数据完整性，我们要求数据库系统维护以下事务性质：

- 原子性。事务的所有操作在数据库中要么全部正确反映出来要么全部不反映。
- 一致性。事务的隔离执行（即没有并发执行的其他事务）保持数据库的一致性。
- 隔离性。尽管多个事务可以并发执行，但系统必须保证对任一事务对 T_i 和 T_j，在 T_i 看来，T_j 或者在 T_i 开始之前已经停止执行，或者在 T_i 完成之后开始执行。这样，每个事务都感觉不到系统中有其他事务在并发地执行。
- 持久性。一个事务成功完成后，它对数据库的改变必须是永久的，即使系统可能出现故障。

这些特性通常被称为 ACID 特性，这一缩写来自四条性质的第一个英文字母。

为了加深对 ACID 性质及其必要性的理解，考虑一个简化的银行系统，这个系统由几个帐户和访问、更新这些帐户的一组事务组成。我们暂且假设数据库永久驻留在磁盘上，而它的某些部分临时驻留在主存中。

数据库的访问由以下两个操作完成：

- read（X）。从数据库传送数据项 X 到执行 read 操作的事务的一个局部缓冲区中。
- write（X）。从执行 write 的事务的局部缓冲区把数据项 X 传回数据库。

在实际数据库系统中，write 操作不一定立即将数据更新到磁盘上，write 操作的结果可以临时存储在内存中，以后再写到磁盘上。但现在我们假设 write 操作立即更新数据库。我们将在第 15 章重提这个话题。

设 T_i 是从帐户 A 过户 $50 到帐户 B 的事务。这个事务可以被定义为

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

现在我们考虑每一个 ACID 要求。(为便于讲解，我们不按 A-C-I-D 的次序来讲述)。

• 一致性。在这里，一致性要求事务的执行不改变 A、B 之和。如果没有一致性要求，事务就会创造或销毁钱！容易验证，如果数据库在事务执行前是一致的，那么事务执行后仍将保持一致性。

确保单个事务的一致性是对该事务编码的应用程序员的责任。完整性约束的自动检查给这项工作带来了便利，我们在第 6 章已经讨论了这个问题。

• 原子性。假设事务 T_2 执行前，帐户 A 和帐户 B 分别有 $1000 和 $2000。现在假设事务 T_1 执行时，系统出现故障，导致 T_1 的执行没有成功完成。这种故障可能是电源故障、硬件故障或软件错误等。我们再假设故障发生在 write (A) 操作执行之后，write (B) 操作执行之前。这种情况下，数据库反映出来的是帐户 A 有 $950 而帐户 B 有 $2000。这次故障的结果是销毁了 $50。还要特别指出的是，$A + B$ 的总和已经改变。

这样，由于系统故障，系统的状态不再反映数据库应当描述的现实世界的真实状态。我们把这种状态称为不一致状态。必须保证这种不一致性在数据库中是不可见的。但是注意，系统必然会在某一时刻处于不一致状态。即使事务 T_1 能执行完，仍然存在某一时刻帐户 A 的金额是 $950 而帐户 B 的金额是 $2000，这显然是一个不一致状态。但这一状态最终会被帐户 A 的金额是 $950 且帐户 B 的金额是 $2050 这个一致状态所代替。这样，如果一个事务要么不开始，要么保证能完成的话，那么不一致状态除了在事务执行当中外，在其他时刻都是不可见的。这就是需要原子性的原因：如果具有原子性，某个事务的所有活动要么在数据库中全部反映要么全部不反映。

保证原子性的基本思路如下。对于事务要执行写操作的数据项，数据库系统在磁盘上记录其旧值，而如果事务没能完成执行，旧值将被恢复，好像事务从未执行。我们将在下节进一步讨论这些想法。保证原子性是数据库系统本身的责任，具体来说，这项工作由事务管理部件处理，我们将在第 15 章详细讲述。

• 持久性。一旦事务成功地完成执行，并且发起事务的用户已经被告知资金转帐已经发生，系统就必须保证任何系统故障都不会再引起与这次转帐相关的数据的丢失。

持久性保证一旦事务成功完成，该事务对数据库施加的所有更新就都是永久的，即使事务执行完成后出现系统故障。

现在假设计算机系统故障将会导致内存数据丢失，但已写入磁盘的数据不会丢失。可以通过确保以下两条中的任何一条来达到持久性：

1) 事务做的更新在事务结束前已经写入磁盘。

2) 有关事务已执行的更新和已写到磁盘上的更新的信息必须充分，能让数据库在系统出现故障后重新启动时重新构造更新。

确保持久性是数据库系统中恢复管理部件的责任。事务管理部件和恢复管理部件之间的关系密切，我们在第 15 章讨论它们的实现。

• 隔离性。即使每个事务都能确保一致性和原子性，但如果几个事务并发执行，它们的操作会以人们所不希望的某种方式交叉执行，这也会导致不一致的状态。

例如，在 A 至 B 转帐事务执行过程中，当 A 中总金额已被减去转帐额并已写回 A，而 B 中总金额被加上转帐额但还未被写回时，数据库暂时是不一致的。如果另一个并发运行的事务在这个中间时刻读取 A 和 B 的值并计算 $A+B$，它将会得到不一致的值。此外，如果第二个事务基于它读取的不一致的值对 A 和 B 进行更新，即使两个事务都已完成，数据库仍可能处于不一致状态。

解决事务并发执行问题的一种方法是串行地执行事务——即一个接一个执行。但我们会在 13.4 节看到，事务并发执行能显著改善性能。因此人们提出了许多其他的解决方法，这些方法允许多个事务并发执行。

13.4 节讲述事务并发执行所引起的问题。事务隔离性确保事务并发执行后的系统状态与这些事务以某种次序一个接一个地执行后的状态是等价的。13.5 节进一步讨论隔离的原则。确保隔离性是数据库系统中并发控制部件的责任，我们将在第 14 章讨论。

13.2 事务状态

在不出现故障的情况下，所有事务都能成功完成。但是，事务并非总能顺利执行完成，这种事务称为中止事务。如果要确保原子性，中止事务必须对数据库的状态不造成影响。这样，中止事务对数据库所做的任何改变必须撤销。一旦中止事务造成的变更被撤销，我们就说事务已回滚。恢复机制负责管理事务中止。

成功完成执行的事务称为已提交事务。对数据库进行更新的已提交事务使数据库进入一个新的一致状态，即使出现系统故障，这个状态也必须保持。

一旦事务已提交，不能通过中止它来撤销其造成的影响。撤销已提交事务所造成影响的唯一方法是执行一个补偿事务，但有时我们不能创建这样的补偿事务。因此，书写和执行补偿事务的责任就留给了用户，而不是由数据库系统处理。第 20 章包括有对补偿事务的讨论。

一个事务成功完成，其含义需进一步明确。为此我们建立了一个简单的抽象事务模型。事务必须处于以下状态之一：

• 活动状态。初始状态，事务执行时处于这个状态。

• 部分提交状态。最后一条语句被执行后。

• 失败状态。发现正常的执行不能继续后。

• 中止状态。事务回滚并且数据库已被恢复到事务开始执行前的状态后。

• 提交状态。成功完成后。

事务相应的状态图如图 13-1 所示。只有在事务已进入提交状态后，才说事务已提交。同样，具有当事务已进入中止状态，才说事务已中止。提交的或中止的事务称为已经结束的事务。

事务从活动状态开始。当事务完成它的最后一条语句后就进入了部分提交状态。此时，事务虽已经完成执行，但由于实际输出可能临时驻留在主存中，而在事务成功完成前还可能出现硬件故障，因此事务仍有可能不得不中止。

接着数据库系统往磁盘上写入足够信息，确保即使出现故障事务所做的更新也能在系统重启后重新创建。当最后一条这样的信息写入后，事务就进入了提交状态。

现在假设故障不会引起磁盘数据丢失。磁盘数据丢失的处理技术将在第 15 章讨论。

系统判定事务不能继续正常执行后（如由于硬件或逻辑错误），事务就进入了失败状态，这种事务必须回滚。这样，事务就进入了中止状态。此刻，系统有两种选择：

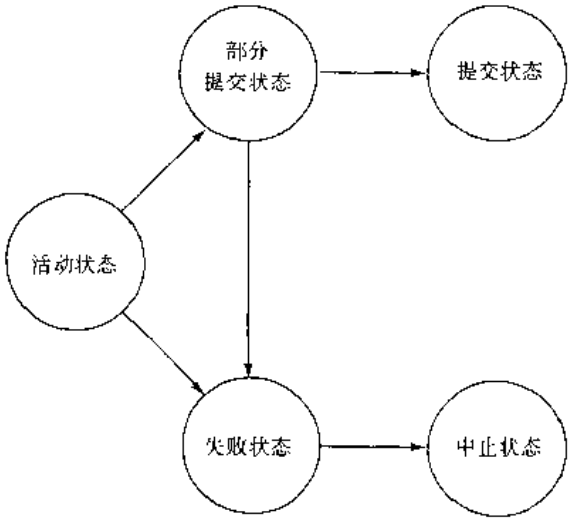


图 13-1　事务状态图

• 重启事务。这仅当引起事务中止的软硬件错误不是由事务的内部逻辑所产生时。重启的事务被看成是一个新事务。

• 杀死事务。系统这样做通常是由于事务的内部逻辑造成的错误，这只有重写应用程序才能改正；或者由于输入错误；或者所需数据在数据库中没有找到。

当处理可见的外部写，比如写至终端或打印机时，我们必须小心。由于写的结果可能已经在数据库系统之外被看到，所以一旦发生这种写操作，就不能再被抹去。大多数系统只允许这种写在事务进入提交状态后发生。实现这种模式的一种方法是在非易失性存储设备中临时写下与外部写相关的所有数据，然后在事务进入提交态后再执行真正的写操作。如果在事务已进入提交态而外部写尚未完成时，系统出现了故障，数据库系统可以在重启后（用存储在非易失性设备中的数据）执行外部写。

对于某些特定应用，允许处于活动状态的事务向用户显示数据也是我们所期望的，特别是对将运行几分钟或几小时的长事务来说。不幸的是，除非牺牲事务原子性，否则不能允许这种可见的数据输出。现在的大多数事务系统为了确保原子性，禁止这种和用户交互的形式。第 20 章会讨论一种支持交互式长事务的事务模型。

13.3　原子性和持久性的实现

数据库系统的恢复管理部件实现对原子性和持久性的支持。首先考虑一个简单但效率极低的方案。这个方案假设某一时刻只有一个活动事务，并且这个方案是基于对数据库做的拷贝（称为*影子拷贝*）。该方案还假设要处理的数据库只是磁盘上的一个文件。磁盘上维护一个称为 db-pointer 的指针，它指向数据库的当前拷贝。

在影子数据库方案中，欲更新数据库的事务首先创建数据库的一个完整拷贝，所有的更新在新建的拷贝上进行，而原始数据库（被称为影子拷贝）原封不动。如果任何时候事务不得不中止，新拷贝简单地被删除，原数据库没有受到任何影响。

如果事务完成，它的提交过程如下所述。首先，操作系统确保数据库新拷贝上的所有页已被写到磁盘上。在 Unix 系统中，flush 命令被用来达到这个目的。在刷新完成后，db-pointer 指针被修改为指向数据库的新拷贝，这个新拷贝于是就成了数据库的当前拷贝。而数据库旧拷贝被删除。这个方案的图形化表示如图 13-2 所示，图中显示了数据库更新前后的状态。

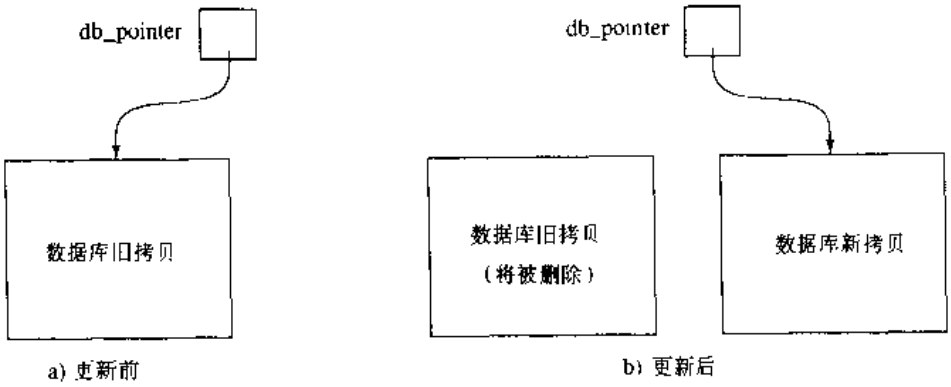


图 13-2 保证原子性和持久性的影子拷贝技术

在修改后的 db-pointer 指针写到磁盘上之后，就说事务已提交了。我们现在来看这种技术是怎样处理事务和系统故障的。首先考虑事务故障。如果事务故障在 db-pointer 指针修改之前发生，数据库的原始内容并未受影响，我们可以简单地通过删除数据库的新拷贝来中止事务。一旦事务已提交，所有的更新就都在 db-pointer 指针所指向的数据库中。这样，不管有没有事务故障，事务更新或者全部反映到数据库中，或者全部不反映。

现在考虑系统故障。假设在修改后的 db-pointer 指针被写到磁盘上之前出现了系统故障。当系统重启后，读取的 db-pointer 指针仍是原值，于是我们看到的仍是数据库的原始内容，事务的影响对数据库无效。下面假设在 db-pointer 指针被写到磁盘上之后出现了系统故障。在指针被修改之前，数据库新拷贝所有被更新的页已被写到磁盘上。如前所述，假设一旦文件被写到磁盘上，即使系统发生故障也不会破坏其内容。这样，系统重启时，读取 db-pointer 指针，我们看到的将是事务更新完成后的数据库内容。

以上实现实际取决于对 db-pointer 写操作的原子性，即要么写入 db-pointer 的所有字节，要么没有写入任何字节。如果 db-pointer 指针的某些字节已写出，而其他字节没有写出，则该指针没有任何意义。这种情况下，系统重启后新、旧版本的数据库都找不到。幸运的是，磁盘系统提供了对块更新的原子性，或至少是支持磁盘扇区更新的原子性。换句话说，磁盘系统将保证 db-pointer 更新的原子性。

这样，数据库系统的恢复管理部件通过影子拷贝实现了事务的原子性和持久性。

举个数据库领域之外的事务处理的简单例子，例如一个文本编辑会话，整个编辑会话可以模型化为一个事务。该事务所执行的动作是读文件与更新文件。编辑结束时保存文件相当于编辑事务的提交，未保存文件退出编辑器对应于事务的中止。

许多文本编辑器基本上采用上述方法来保证一个编辑会话的事务性。一个新文件被用来存储修改后的文件。编辑结束时，若要保存修改过的文件，则使用文件重命名命令将新文件重命名为真正的文件名。文件重命名由文件系统作为原子操作实现，而且删除旧文件。

不幸的是，这种实现方法在大型数据库的情形下效率很低，因为一个事务的执行就要求复制整个数据库。此外，该实现方法不允许事务并发执行。但尽管如此，仍有许多实现事务原子性和持久性的可行方法，这些方法所需代价比这里所用方法小得多且功能更强。这样的恢复技术在第 15 章讲述。

13.4 并发执行

事务处理系统通常允许多个事务并发执行。如前所述，多个事务并发更新数据容易引起数

据一致性的问题。在事务并发执行的情况下保证一致性需进行特殊处理。如果要求事务串行执行，事情就变得很简单了。事务串行执行指一次执行一个事务，每个事务仅当前一事务执行完后才开始。然而，有两条理由促使我们考虑并发。

• 一个事务由多个步骤组成，一些步骤涉及 I/O 活动，而另一些涉及 CPU 活动。计算机系统中 CPU 与磁盘可以并行运作，因此，I/O 活动可以与 CPU 处理并行进行。利用 CPU 与 I/O 系统的并行性，多个事务可并行执行。当一个事务在一个磁盘上进行读写时，另一个事务可在 CPU 上运行，此外第三个事务又可在另一磁盘上进行读写。这样系统的吞吐量增加——即给定时间内执行的事务数增加。相应地，处理器与磁盘利用率也提高，换句话说，处理器与磁盘空闲（没有做有用的工作）的时间较少。

• 系统中可能运行着各种各样的事务，一些较短，一些较长。如果事务串行执行，短事务可能得等待它前面的长事务完成，这可能导致难以预测的时延。如果各事务是针对数据库的不同部分进行操作，事务并发执行会更好，这样各事务可以共享 CPU 周期与磁盘存取。并发执行可以减少不可预测的事务执行时延。此外，并发执行也可减少平均响应时间，即一个事务从开始到完成所需的平均时间。

在数据库中使用并发执行的动机本质上与操作系统中使用多道程序的动机是一样的。

当多个事务并发执行时，即使每个事务都正确执行，数据库的一致性也可能被破坏。本节我们将讲述调度的概念，调度用于确定那些可以保证一致性的执行序列。

数据库系统必须控制事务之间的相互影响，防止它们破坏数据库的一致性。系统通过称为并发控制机制的各种机制来实现。我们将在第 14 章研究并发控制机制。现在我们集中研究正确的并发执行这一概念。

再来看 13.1 节中简化了的银行系统，其中包含多个帐户以及存取、更新这些帐户的一组事务。设 T_1、T_2 是将资金从一个帐户转移到另一个帐户的两个事务。事务 T_1 是从帐户 A 过户 \$ 50 到帐户 B 的事务，这个事务定义为：

T_1: **read**(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

事务 T_2 是从帐户 A 过户 10% 的存款余额到帐户 B 的事务，这个事务定义为：

T_2: **read**(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

设帐户 A 和帐户 B 当前分别有 \$ 1000 和 \$ 2000。假设两个事务一次执行一个，先是 T_1，然

后是 T_2。该执行顺序如图 13-3 所示。图中，指令序列自顶向下按时间顺序排列，T_1 的指令出现在左栏，T_2 的指令出现在右栏。按图 13-3 的顺序执行后，帐户 A 与 B 的最终值分别为 $855 与 $2145。因此，帐户 A 与 B 的资金总数（即 $A+B$ 之和）在两个事务执行后保持不变。

同样，如果事务一次执行一个，先是 T_2，然后是 T_1，那么相应的执行顺序如图 13-4 所示。正如所预期的那样，$A+B$ 之和仍维持不变。帐户 A 与 B 中最终的值分别为 $850 与 $2150。

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$;
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

图 13-3 调度 1——一个串行调度，T_2 跟在 T_1 之后

T_1	T_2
	read(A)
	$temp := A * 0.1$;
	$A := A \quad temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

图 13-4 调度 2——一个串行调度，T_1 跟在 T_2 之后

前面所说的执行顺序称为调度，它们表示指令在系统中执行的时间顺序。显然，一组事务的一个调度必须包含这一组事务的全部指令，并且必须保持指令在各个事务中出现的顺序。例如，在任何一个有效调度中，事务 T_1 中指令 write（A）必须出现在指令 read（B）之前。在下面的讨论中，我们称第一种执行顺序为调度 1（T_2 跟在 T_1 之后），称第二种执行顺序为调度 2（T_1 跟在 T_2 之后）。

这两个调度是串行的。串行调度由来自各事务的指令序列组成，其中属于同一事务的指令在调度中紧挨在一起。因此，对于有 n 个事务的事务组，共有 $n!$ 个有效的串行调度。

当多个事务并发执行时，相应的调度不必是串行的。若有两个并发执行的事务，操作系统可能先选其中的一个事务执行一小段时间，然后切换上下文，执行第二个事务一段时间，接着又切换到第一个事务执行一段时间，如此下去。多个事务的情形下，所有事务共享 CPU 时间。

执行顺序可能会有多种，因为来自两个事务的各条指令可以交叉执行。一般而言，在 CPU 切换到另一事务之前准确预测 CPU 将执行多少条某个事务的指令是不可能的。因此，对于有 n 个事务的事务组，可能的调度总数要比 $n!$ 大得多。

回到前面的例子，假设两个事务并发执行，一种可能的调度如图 13-5 所示。执行完成后，得到的状态与先执行 T_1 再执行 T_2 的串行调度得到的一样。$A+B$ 之和保持不变。

不是所有的并发执行都能得到正确的结果。举个例子，考虑如图 13-6 所示的调度。该调度执行后，到达的状态是帐户 A 与 B 中最终的值分别为 $950 与 $2100。实际上，这个最终状态是一个不一致状态，因为在并发执行过程中多出了 $50，两个事务执行后 $A+B$ 之和未能保持不变。

如果并发执行的控制完全由操作系统负责，许多调度都是可能的，包括上述那种使数据库处于不一致状态的调度。保证任何调度执行后数据库总处于一致状态是数据库系统的职责，数据库系统中完成此任务的部件是*并发控制部件*。

在并发执行中，通过保证任何调度执行的效果与没有并发的调度的执行效果一样，可以确保数据库的一致性。也就是说，调度在某种意义上等价于一个串行调度。我们在下节探讨这个问题。

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$;
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

图 13-5　调度 3——等价于调度 1 的一个并发调度

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$;
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

图 13-6　调度 4 —— 一个并发调度

13.5　可串行化

数据库系统必须控制事务的并发执行，保证数据库处于一致的状态。在研究数据库系统如何执行该任务前，首先必须理解哪些调度能确保一致性，哪些调度不能。

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

图 13-7　调度 3—— 只显示 **read** 与 **write** 指令

由于事务就是程序，在计算上要确定一个事务有哪些操作、多个事务的操作如何相互作用是有困难的。由于这个缘故，我们不解释一个事务对某一数据项所能执行操作的类型，而只考虑 read 与 write 这两种操作。我们认为数据项 Q 上的 read（Q）和 write（Q）指令之间，事务可以对驻留在事务局部缓冲中的 Q 的拷贝执行任意操作序列。所以，从调度的角度来看，事务仅有的重要操作是 read 与 write 指令。因此，调度中通常只显示 read 与 write 指令，正如图 13-7 所示调度 3 中所表示的那样。

本节我们讨论不同形式的等价调度，由此引出冲突可串行化与视图可串行化的概念。

13.5.1　冲突可串行化

现在考虑一个调度 S，其中含有分别属于 T_i 与 T_j 的两条连续指令 I_i 与 I_j（$i \neq j$）。如果 I_i 与 I_j 引用不同的数据项，则交换 I_i 与 I_j 不会影响调度中任何指令的结果。然而，若 I_i 与 I_j 引用相同的数据项 Q，则两者的顺序就需考虑。由于我们只处理 read 与 write 指令，因而只需考虑以下四种情形：

1）I_i = read（Q），I_j = read（Q）。I_i 与 I_j 的次序无关紧要，因为不论其次序如何，T_i 与 T_j 读取的 Q 值总是相同的。

2）I_i = read（Q），I_j = write（Q）。若 I_i 先于 I_j，则 T_i 不会读取由 T_j 的指令 I_j 写入的 Q 值；若 I_j 先于 I_i，则 T_i 读到由 T_j 的指令 I_j 写入的 Q 值。因此 I_i 与 I_j 的次序是重要的。

3）I_i = write（Q），I_j = read（Q）。I_i 与 I_j 的次序是重要的，原因类似前一情形。

4）I_i = write（Q），I_j = write（Q）。由于两条指令均为 write 指令，指令的顺序对 T_i 与 T_j

没什么影响。然而，调度 S 的下一条 read（Q）指令读取的值将受到影响，因为数据库电只保留了两条 write 指令中后一条的结果。如果在调度 S 中的指令 I_i 与 I_j 之后没有其他的 write（Q）指令，则 I_i 与 I_j 的顺序直接影响调度 S 产生的数据库状态中 Q 的最终值。

因此，只有在 I_i 与 I_j 全为 read 指令时，两条指令的执行顺序才是无关紧要的。

当 I_i 与 I_j 是不同事务在相同数据项上的操作，并且其中至少有一个是 write 指令时，我们说 I_i 与 I_j 是冲突的。

为了说明冲突指令的概念，来看图 13-7 中的调度 3。T_1 的 write（A）指令与 T_2 的 read（A）指令相冲突。然而，T_2 的 write（A）指令与 T_1 的 read（B）指令不冲突，因为这两条指令访问不同的数据项。

设 I_i 与 I_j 是调度 S 的两条连续指令。若 I_i 与 I_j，属于不同事务且不冲突，则可以交换 I_i 与 I_j 的顺序得到一个新的调度 S'。我们认为 S 与 S'等价，因为除了 I_i 与 I_j 外，其他指令的次序与原来相同，而 I_i 与 I_j 的顺序无关紧要。

在图 13-7 的调度 3 中，由于 T_2 的 write(A)指令与 T_1 的 read(B)指令不冲突，因而可以交换这两条指令得到一个等价的调度——图 13-8 所示的调度 5。不管系统初始状态如何，调度 3 与调度 5 均得到相同的系统最终状态。

继续交换非冲突指令如下：

- 交换 T_1 的 read（B）指令与 T_2 的 read（A）指令。
- 交换 T_1 的 write（B）指令与 T_2 的 write（A）指令。
- 交换 T_1 的 write（B）指令与 T_2 的 read（A）指令。

经过以上交换的结果是一个串行调度，即图 13-9 所示的调度 6。这样，我们证明了调度 3 等价于一个串行调度。该等价性意味着不管初始系统状态如何，调度 3 总与某个串行调度得到相同的终态。

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

图 13-8 调度 5——交换调度 3 的一对指令得到的调度

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

图 13-9 调度 6——与调度 3 等价的一个串行调度

如果调度 S 可以经过一系列非冲突指令交换转换成 S'，我们称 S 与 S'是冲突等价的。

再看前面那些例子，我们注意到调度 1 与调度 2 不是冲突等价的。然而，调度 1 与调度 3 冲突等价，因为 T_1 的指令 read（B）、write（B）可以和 T_2 的指令 read（A）、write（A）相交换。

冲突等价的概念引出了冲突可串行化的概念。称一个调度 S 是冲突可串行化的是指该调度与一个串行调度冲突等价。因此，调度 3 是冲突可串行化的，因为它等价于串行调度 1。

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

图 13-10 调度 7

最后考虑图 13-10 所示调度 7，该调度仅包含事务 T_3 与 T_4 的重要操作（即 read 与 write）。这个调度不是冲突可串行化的，因为它既不等价于串行调度$<T_3, T_4>$也不等价于串行调度$<T_4, T_3>$。

有可能存在两个调度，它们产生相同的结果，但它们不是冲突等价的。例如事务 T_5，它从帐户 B 过户 \$10 到帐户 A。设调度 8 如图 13-11 所示。调度 8 不与串行调度 $<T_1, T_5>$ 冲突等价是因为在调度 8 中，T_5 的 write（B）指令与 T_1 的 read（B）指令冲突。因此，不能通过交换连续的非冲突指令将 T_1 的所有指令移到 T_5 之前。然而，执行调度 8 或串行调度 $<T_1, T_5>$ 后，帐户 A 与 B 的最终值是相同的，即分别为 \$960 与 \$2040。

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

图 13-11　调度 8

从这个例子可以看出，存在比冲突等价定义限制松一些的调度等价定义。对于系统来说，要确定调度 8 与串行调度 $<T_1, T_5>$ 产生的结果相同，系统必须分析 T_1 与 T_5 所进行的计算，而不只是分析其读写操作。一般而言，这种分析难于实现且计算代价很大。不过，还是存在一些别的纯粹基于 read 与 write 操作的调度等价定义，下一节我们考察其中的一个。

13.5.2　视图可串行化

本节考虑比冲突等价限制要宽松的一种等价形式，但这种等价形式与冲突等价一样基于事务的 read 与 write 操作。

考虑两个调度 S 与 S'，其中参与两个调度的事务集是相同的。若调度 S 与 S' 满足下面三个条件，则称 S 与 S' 为视图等价的。

1）对于每个数据项 Q，若事务 T_i 在调度 S 中读取了 Q 的初始值，那么 T_i 在调度 S' 中也必须读取 Q 的初始值。

2）对于每个数据项 Q，若事务 T_i 在调度 S 中执行了 read（Q）并且读取的值是由 T_j 产生的，则 T_i 在调度 S' 中读取的值也必须是由 T_j 产生的。

3）对于每个数据项 Q，若在调度 S 中有事务执行了最后的 write（Q）操作，则在调度 S' 中该事务也必须执行最后的 write（Q）操作。

条件 1）与 2）保证了两个调度中的每个事务都读取相同的值，从而进行相同的计算。条件 3）与条件 1）、2）一起保证两个调度得到相同的最终系统状态。

回到前面的例子，注意到调度 1 与调度 2 并不视图等价，因为在调度 1 中，T_2 读取的帐户 A 的值是由 T_1 产生的，而调度 2 不满足这一点。然而，调度 1 与调度 3 是视图等价的，因为在两个调度中，事务 T_2 读取的帐户 A 与 B 的值都是由 T_1 产生的。

视图等价的概念引出了视图可串行化的概念。如果某个调度视图等价于一个串行调度，则说这个调度是视图可串行化的。

举个例子，假设在调度 7 中增加事务 6，由此得到调度 9，如图 13-12 所示。则调度 9 是视图可串行化的。实际上，调度 9 视图等价于串行调度 $<T_3, T_4, T_6>$，因为两个调度中的 read（Q）指令均是读取 Q 的初始值，两个调度中的 T_6 均最后写入 Q 值。

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

图 13-12　调度 9——一个视图可串行化调度

每个冲突可串行化调度都是视图可串行化的，但也存在非冲突可串行化的视图可串行化调度。事实上，调度9就不是冲突可串行化的，因为每对连续指令均冲突，从而交换指令是不可能的。

注意，调度 9 中，事务 T_4 与 T_6 执行 write（Q）操作之前没有执行 read（Q）操作，这样的写操作称作盲目写操作。盲目写操作存在于任何非冲突可串行化的视图可串行化调度中。

13.6 可恢复性

至此，我们在隐含假定在无事务故障的前提下，从数据库一致性的观点出发，对什么样的调度可以接受进行了研究。现在讨论在并发执行过程中事务故障所产生的影响。

无论什么原因，如果事务 T_i 失败了，我们必须撤销该事务的影响以确保事务的原子性。在允许并发执行的系统中，还必须确保依赖于 T_i 的任何事务 T_j（即 T_j 读取了 T_i 写的数据）也中止。为确保这一点，需要对系统所允许的调度类型做一些限制。

在下面两小节中，我们根据以上观点讲述什么样的调度是可接受的。前面已经提到，我们将在第 14 章中讲述如何保证只产生可接受的调度。

13.6.1 可恢复调度

考虑图 13-13 所示的调度 11，其中事务 T_9 只执行一条指令：read（A）。假定系统允许 T_9 在执行完 read（A）后立即提交，则 T_9 将先于 T_8 提交。现假定 T_8 在提交前发生故障。由于 T_9 已读取了由 T_8 写入的数据项 A，因而我们必须中止 T_9 以保证事务的原子性。但 T_9 已提交，不能再中止，这样就出现了 T_8 发生故障后不能正确恢复的情形。

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

图 13-13 调度 11

调度 11 中执行完 read（A）指令后立即提交，是不可恢复调度的一个例子，这是不允许的。大多数数据库系统要求所有调度都是可恢复的。可恢复调度应满足：对于每对事务 T_i 和 T_j，如果 T_j 读取了由 T_i 所写的数据项，则 T_i 先于 T_j 提交。

13.6.2 无级联调度

即使一个调度是可恢复调度，要从事务 T_i 故障中正确恢复，我们可能不得不回滚若干事务。当其他事务读取了由事务 T_i 所写的数据项时就会发生这种情形。举一个例子，考虑图 13-14 所示的部分调度。事务 T_{10} 写入 A，事务 T_{11} 读取 A；事务 T_{11} 写入 A，事务 T_{12} 读取 A。假定现在事务 T_{10} 失败，T_{10} 必须回滚。由于 T_{11} 依赖于 T_{10}，则事务 T_{11} 必须回滚。由于 T_{12} 依赖于 T_{11}，则事务 T_{12} 必须回滚。这种因一个事务故障导致一系列事务回滚的现象称为级联回滚。

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

图 13-14 调度 12

由于级联回滚导致撤销大量工作，因而我们不希望发生级联回滚。我们希望对调度加以限制，从而避免级联回滚发生。像这样的调度称为*无级联调度*。无级联调度应满足：对于每对事务 T_i 和 T_j，如果 T_j 读取了由 T_i 所写的数据项，则 T_i 必须在 T_j 读取之前提交。容易验证无级联调度总是可恢复的。

13.7 隔离性的实现

至此，我们已经知道一个调度应具有什么样的特性才能保证数据库处于一致状态，保证事务故障得以安全处理。具体来说，冲突或视图可串行化且无级联的调度满足这些要求。

可以使用多种并发控制机制来保证只产生可接受的调度。不论是否有多个事务并发执行，不论操作系统在事务之间如何分时共享资源，我们都可做到这一点。

举一个并发控制机制的简单例子。考虑如下机制：一个事务在它开始前获得整个数据库的锁，在它提交之后又释放这个锁。当一个事务持有锁时，其他事务就不允许获得这个锁，因此

必须等待释放锁。由于采用了封锁策略，一次只能执行一个事务，所以只会产生串行调度。这些调度很明显是可串行化的，并且容易证明它们也是无级联的。

这样的并发控制机制导致性能低下，因为这种机制迫使事务必须等到前面的事务结束后才能开始。换句话说，这种机制提供的并发程度很低。其实，正如 13.4 节谈到的那样，并发执行有诸多性能方面的益处。

并发控制机制的目标是获得高度的并发性，同时保证所产生的调度是冲突或视图可串行化的，并且是无级联的。

第 14 章我们将研究几个并发控制机制。在所允许的并发程度和所导致的开销之间，这些机制有不同的权衡。某些机制只允许产生冲突可串行化调度，而另一些则允许产生非冲突可串行化的视图可串行化调度。

13.8 SQL 中的事务定义

数据操纵语言必须包含用于说明构成事务的活动集合的结构。

SQL 标准规定事务的开始是隐含的，事务的结束用下列 SQL 语句之一来表示：

- Commit work。提交当前事务并开始一个新的事务。
- Rollback work。中止当前事务。

这两个语句中的关键字 work 是可选的。如果一个程序结束前没用这两条命令，那么其更新或者提交或者回滚——具体哪一个会发生，标准没有规定，视具体实现而定。

标准还规定系统必须保证可串行化和不存在级联回滚。标准中使用的可串行化定义是指一个调度的执行必须与一个串行调度的执行有相同的效果。因此，冲突与视图可串行化均是可接受的。

SQL-92 标准还允许当某一事务与别的事务并发时，该事务可以某种非可串行化的方式执行。例如，一个事务可能运行在 read uncommitted 级，这个级别的事务甚至可以读取未曾提交的记录。这种特性是为那些结果不必太精确的长事务提供的。例如，查询优化中使用的统计数据通常用近似信息就足够了。如果这些事务以可串行方式执行，它们可能与其他事务的执行发生冲突，引起其他事务延迟执行。

以下是 SQL-92 标准规定的一致性级别：

- serializable。是默认的。
- repeatable read。只允许读取已提交的记录，并要求一个事务对同一记录的两次读取之间，其他事务不能对该记录进行更新。然而，该事务与其他事务可能是不可串行化的。例如，当某个事务搜索满足某些条件的记录时，它找到的可能是已提交事务插入的那些记录，但却可能找不到其他记录。
- read committed。只允许读取已提交的记录，但不要求可重复读。例如，事务对同一记录的两次读取之间，记录可能被已提交的事务更新。
- read uncommitted。允许读取未提交的记录。它是 SQL-92 标准中所允许的最低一致性级别。

13.9 可串行化判定

在设计并发控制机制时，我们必须证明该机制产生的调度是可串行化的。要做到这一点，我们首先必须知道如何确定一个给定的调度 S 是否是可串行化的。本节讲述用于确定冲突与视图可串行化的方法。我们将证明存在一个简单有效的确定冲突可串行化的算法。然而，判定视

图可串行化的有效算法是不存在的。

13.9.1　冲突可串行化的判定

设 S 是一个调度。由 S 构造一个有向图，称为优先图。该图由两部分 $G=(V, E)$ 组成，其中 V 是顶点集，E 是边集。顶点集由所有参与调度的事务组成。边集由满足下列三个条件之一的边 $T_i \rightarrow T_j$ 组成：

1）在 T_j 执行 read（Q）之前，T_i 执行 write（Q）。

2）在 T_j 执行 write（Q）之前，T_i 执行 read（Q）。

3）在 T_j 执行 write（Q）之前，T_i 执行 write（Q）。

如果优先图中存在边 $T_i \rightarrow T_j$，则在任何等价于 S 的串行调度 S' 中，T_i 都必出现在 T_j 之前。

例如，调度 1 的优先图如图 13-15a 所示。图中只有一条边 $T_1 \rightarrow T_2$，因为 T_1 的所有指令均在 T_2 的首条指令之前执行。图 13-15b 表示的是调度 2 的优先图，图中仅含一条边 $T_2 \rightarrow T_1$，因为 T_2 的所有指令均在 T_1 的首条指令之前执行。

调度 4 的优先图如图 13-16 所示。因为 T_1 执行 read（A）先于 T_2 执行 write（A），所以图中含有边 $T_1 \rightarrow T_2$。又因 T_2 执行 read（B）先于 T_1 执行 write（B），所以图中含有边 $T_2 \rightarrow T_1$。

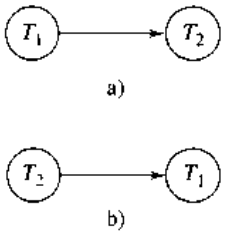


图 13-15　a）调度 1 与 b）调度 2 的优先图

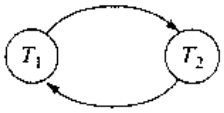


图 13-16　调度 4 的优先图

如果调度 S 的优先图中有环，则调度 S 是非冲突可串行化的。如果图中无环，则调度 S 是冲突可串行化的。串行化顺序可通过拓扑排序得到，拓扑排序用于计算与优先图的偏序相一致的线序。一般而言，通过拓扑排序可以获得多个线序。例如，图 13-17a 有两种可接受的线序，如图 13-17b 与图 13-17c 所示。

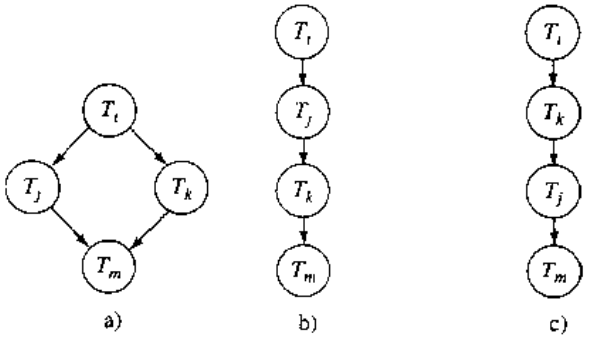


图 13-17　拓扑排序示例

因此，要判定冲突可串行化，需要构造优先图并调用一个环检测算法。环检测算法可在常见的算法课本中找到。环检测算法（例如那些基于深度优先搜索的环检测算法）需要 n^2 数量级的运算，其中 n 是图中顶点数（即事务数）。这样，我们有了判定冲突可串行化的实际可行

的方法。

回到前面的例子，发现调度 1 与调度 2 的优先图（图 13-15）无环，而调度 4 的优先图（图 13-16）有环，可见调度 4 是非冲突可串行化的。

13.9.2 视图可串行化的判定

下面讲述如何修改冲突可串行化的判定方法来判定视图可串行化，然而，这样所得到的判定方法将花费大量 CPU 时间。实际上，我们在后面会看到判定视图可串行化是计算代价很高的一个问题。

当判定冲突可串行化时，我们知道如果两个事务 T_i 和 T_j 都访问数据项 Q，并且两个事务中至少有一个事务有写 Q 的操作，则优先图中必然插入边 $T_i \rightarrow T_j$ 或 $T_j \rightarrow T_i$。然而在判定视图可串行化时，情况并非如此。我们很快会看到，这种差别使我们无法找到高效的判定算法。

考虑前面图 13-12 中给出的调度 9。按照判定冲突可串行化的规则构造优先图，我们得到图 13-18。该图包含了一个环，表明调度 9 不是冲突可串行化的。然而如前所述，调度 9 是视图可串行化的，因为它视图等价于串行调度 $<T_3, T_4, T_6>$。边 $T_4 \rightarrow T_3$ 不应加入图中，因为 T_3、T_4 产生的数据项值 Q 没被其他事务引用，T_6 产生了 Q 的一个新最终值。T_3、T_4 的 write（Q）指令被称为无用的写操作。

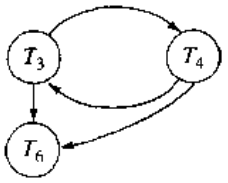


图 13-18　调度 9 的优先图

前面已经说过，不能简单地使用上节的优先图方法来判定视图是否可串行化。我们需要设计一种用来判定一条边是否有必要加入优先图中的方法。

设 S 是一个调度。假定事务 T_j 读取事务 T_i 写入的数据项 Q。显然，如果调度 S 是视图可串行化的，则在任何与调度 S 等价的串行调度 S' 中 T_i 必然先于 T_j。现在假定在调度 S 中事务 T_k 执行了 write（Q）。则在调度 S' 中，T_k 要么先于 T_i，要么在 T_j 之后。T_k 不可能出现在 T_i 与 T_j 之间，否则 T_j 将不能读取事务 T_i 写入的数据项 Q，那么 S 不会视图等价于 S'。

这些约束不能通过前面讲述的简单优先图模型表示出来。造成困难的原因可从前面的例子中看出，边 $T_k \rightarrow T_i$ 与 $T_j \rightarrow T_k$ 两者之一必须加入优先图中，但我们还未制定出规则来确定哪一个选择合适。

要制定这样的规则，必须扩充优先图，在优先图中引入带标记的边。我们称这样的图为带标记的优先图。和前面一样，图中的结点是参与调度的事务。接下来我们讲述插入带标记边的规则。

设调度 S 包含了事务 $\{T_1, T_2, \cdots, T_n\}$。设 T_b 与 T_f 是两个虚事务，其中 T_b 为 S 中访问的每个 Q 执行 write（Q）操作，T_f 为 S 中访问的每个 Q 执行 read（Q）操作。通过在调度 S 的开头插入 T_b，在调度 S 的末尾添加 T_f，我们得到一个新调度 S'。我们按下面的方法构造带标记的优先图：

1）如果事务 T_j 读取事务 T_i 写入的数据项 Q 的值，则加入边 $T_i \xrightarrow{0} T_j$。

2）删除所有关联于无用事务的边。如果在优先图中不存在从 T_i 到 T_f 的通路，则事务 T_i 是无用事务。

3）对于每个数据项 Q，如果事务 T_j 读取事务 T_i 写入的 Q 值，T_k 执行 write（Q）操作且 $T_k \neq T_b$，则做以下事情：

a. 如果 $T_i = T_b$ 且 $T_j \neq T_f$，则在带标记的优先图中插入边 $T_j \xrightarrow{0} T_k$。

b. 如果 $T_i \neq T_b$ 且 $T_j = T_f$，则在带标记的优先图中插入边 $T_k \xrightarrow{0} T_i$。

c. 如果 $T_i \neq T_b$ 且 $T_j \neq T_f$，则在带标记的优先图中插入边 $T_k \xrightarrow{p} T_i$ 与 $T_j \xrightarrow{p} T_k$，其中 p 是一个唯一的、在前面标记边时未曾用过的大于零的整数。

规则 3)c 意思是若 T_j 读取事务 T_i 写入的数据项，则写入同一数据项的事务 T_k 必须在 T_i 之前或 T_j 之后。由于 T_b 与 T_f 必须分别是第一与最后一个事务，因而规则 3)a 与 3)b 是由此引起的特殊情形。当我们应用规则 3)c 时，并不要求 T_k 既在 T_i 之前又在 T_j 之后，而是可以对 T_k 在等价串行顺序中的位置进行选择。

作为示例，再考虑调度 7（图 13-10）。用第 1)、2）步构造出的图如图 13-19a 所示，该图包含边 $T_b \xrightarrow{0} T_3$，因为 T_3 读取事务 T_b 写入的数据项 Q 的值。该图还包含边 $T_3 \xrightarrow{0} T_f$，因为 T_3 是写入 Q 的最后事务，从而 T_f 读取该 Q 值。对应于调度 7 的最终图见图 13-19b。它包含边 $T_3 \xrightarrow{0} T_4$，这是由步骤 3)a 得到的。它也包含边 $T_4 \xrightarrow{0} T_3$，这是由步骤 3)b 得到的。

现在考虑调度 9（图 13-12）。用第 1)、2）步构造出的图如图 13-20a 所示。最终图如图 13-20b 所示，该图包含边 $T_3 \xrightarrow{0} T_4$ 与 $T_3 \xrightarrow{0} T_6$，这是由步骤 3)a 得到的。由步骤 3)b 可得到边 $T_3 \xrightarrow{0} T_6$（已在图中）与边 $T_4 \xrightarrow{0} T_6$。

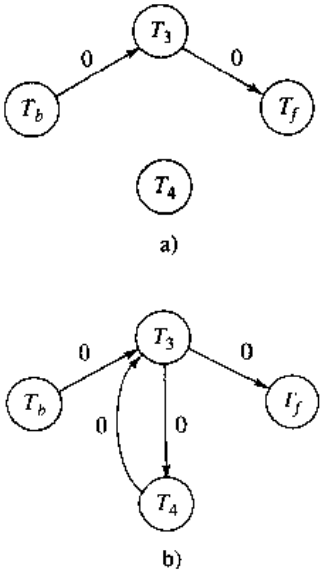


图 13-19 调度 7 的带标记的优先图

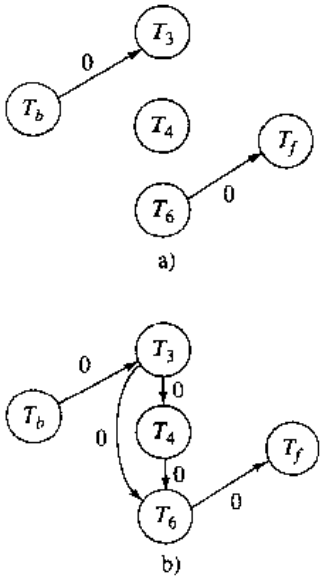


图 13-20 调度 9 的带标记的优先图

最后考虑图 13-21 所示的调度 10。调度 10 是视图可串行化的，因为它视图等价于串行调度 $< T_3, T_4, T_7 >$。用第 1)、2）步构造出的相应的带标记优先图如图 13-22a 所示。最终图如图 13-22b 所示，使用规则 3)a，该图插入了边 $T_3 \xrightarrow{0} T_4$ 与 $T_3 \xrightarrow{0} T_7$。使用规则 3)c 一次，图中加入边 $T_3 \xrightarrow{1} T_4$ 与 $T_7 \xrightarrow{1} T_3$。

在图 13-19b 与 13-22b 中分别含有以下最小环：

- $T_3 \xrightarrow{0} T_4 \xrightarrow{0} T_3$。
- $T_3 \xrightarrow{0} T_7 \xrightarrow{0} T_3$。

图 13-20b 中却不包含环。

如果图中无环，则对应的调度是视图可串行化的。例如，图 13-20b 中无环，同时其对应

的调度 9 是视图可串行化的。当然，图中包含了环未必意味着对应的调度非视图可串行化。例如，图 13-19b 中的优先图有环，与其对应的调度 7 是非视图可串行化的。图 13-22b 中的优先图也有环，但其对应的调度 10 却是视图可串行化的。

那么怎样判定一个调度是否是视图可串行化的呢？答案取决于对优先图的适当解释。假设有 n 对不同的边对，即在构造优先图过程中使用了 $3n$ 次 3）c 规则。这时存在 2^n 个不同的图，每个图中只包含每对边中的一条。这些图中的任何一个若无环，则对应的调度是视图可串行化的。串行化顺序通过将虚事务 T_b 与 T_f 从图中删除后对剩余的无环图进行拓扑排序得到。

回到图 13-22b 所示的优先图。由于只有一个边对，所以有两个不同的图需要考虑，这两个图在图 13-23 中表示。图 13-23a 中的优先图是无环的，因此我们知道与其对应的调度 10 是视图可串行化的。

T_3	T_4	T_7
read(Q)		
	write(Q)	
		read(Q)
write(Q)		
		write(Q)

图 13-21　调度 10

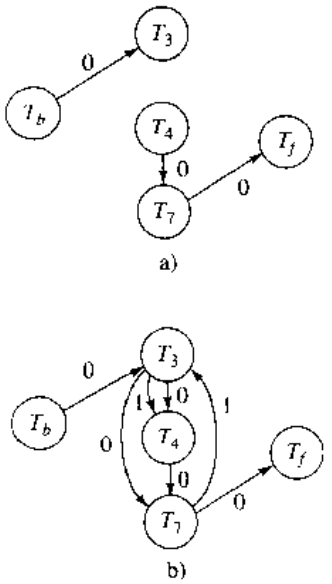


图 13-22　调度 10 的带标记的优先图

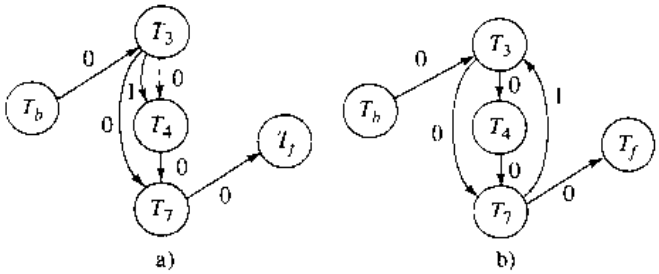


图 13-23　两个不同的优先图

以上讲述的算法需要穷尽对所有可能的不同优先图进行的测试。这类判定无环图的问题已被证明是 *NP*-完全问题（关于 *NP*-完全问题理论的参考书见文献注解）。*NP*-完全问题的任何算法所耗时间几乎总是与问题的规模呈指数关系。

事实上，有关判定调度是否视图可串行化的问题已被证明属于 NP-完全问题，因此几乎肯定不存在有效的判定视图可串行化的算法。然而，并发控制机制仍然可以利用充分条件判定视

图可串行化。也就是说，如果调度满足充分条件，则该调度是视图可串行化的，但是存在不满足该充分条件的视图可串行化调度。

13.10 总结

事务是一个程序执行单元，它访问且可能更新不同的数据项。理解事务这个概念对于理解与实现数据库中的数据更新是很关键的，只有这样才能保证并发执行与各种故障不会导致数据库处于不一致状态。事务具有ACID特性：原子性、一致性、隔离性和持久性。

原子性保证事务的所有操作在数据库中全部反映出来或根本不反映，事务故障不会使数据库处于事务部分执行后的状态。一致性保证若数据库一开始是一致的，则事务（自己）执行后数据库仍处于一致状态。隔离性保证并发执行的事务相互隔离，从而使得每个事务感觉不到系统中其他事务的并发执行。持久性保证一旦某个事务提交后，它对数据库的改变不会丢失，即使系统可能出现故障。

当多个事务在数据库中并发执行时，数据的一致性可能受到破坏，因此系统必须控制各并发事务之间的相互作用。由于事务是保持一致性的单元，多个事务的串行执行可保持一致性，因而我们要求事务集的并发处理所产生的任何调度的执行效果等价于这些事务按某种串行顺序执行的效果。确保这种特性的系统称为保证可串行化的系统。不同的等价概念引出了冲突可串行化与视图可串行化的概念。

并发执行事务所产生的可串行化调度可以通过多个并发控制机制中的一个来加以保证。数据库的并发控制管理部件负责并发控制机制。数据库的恢复管理部件负责保证事务的原子性与持久性。

给定一个调度，我们可以通过为该调度构造优先图及搜索是否无环来判定它是否冲突可串行化。判定所用时间是n^2数量级，其中n是事务数。通过构造带标记的优先图以及在所有可能的不同的带标记的优先图中搜索一个无环图，我们可以判定一个调度是否视图可串行化，其时间开销是2^n数量级。

习题

13.1 列出ACID特性，解释每一特性的用途。

13.2 假设存在永远不出现故障的数据库系统，对这样的系统还需要故障恢复管理器吗？

13.3 考虑一种文件系统，比如你最喜欢的某种操作系统的文件系统。原子性和持久性问题与下列因素相关吗？解释你的回答。

(a) 文件系统的一个用户。

(b) 文件系统的一种实现。

13.4 数据库系统实现者比文件系统实现者更注重ACID特性，这是为什么？

13.5 事务从开始执行到提交或终止，其间要经过几个状态。列出所有可能出现的事务状态序列，解释每一种状态出现的原因。

13.6 解释串行调度和可串行化调度的区别。

13.7 考虑以下两个事务：

T_1: **read**(A);
read(B);
if $A = 0$ **then** $B := B + 1$;

write(B);
T_2: read(B);
read(A);
if $B = 0$ then $A := A + 1$;
write(A);

设一致性需求为 $A = 0 \vee B = 0$，初值是 $A = B = 0$。

(a) 说明包括这两个事务的任意串行执行保持数据库的一致性。

(b) 给出 T_1 和 T_2 的一次并发执行，该执行可产生不可串行化调度。

(c) 存在可产生可串行化调度的 T_1 和 T_2 的并发执行吗?

13.8 既然每一个冲突可串行化调度都是视图可串行化的，那么我们为什么强调冲突可串行化而非视图可串行化呢?

13.9 考虑图 13-24 所示的优先图，对应的调度是视图可串行化的吗? 解释你的问答。

13.10 考虑图 13-25 所示的优先图，对应的调度是视图可串行化的吗? 解释你的回答。

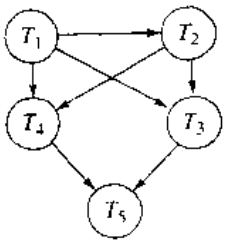


图 13-24　优先图

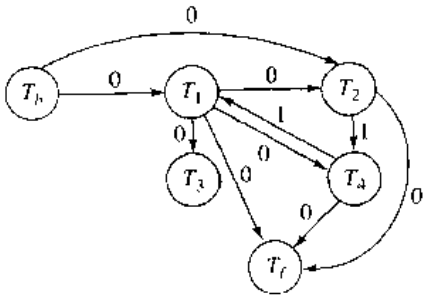


图 13-25　带标记的优先图

13.11 什么是可恢复调度? 为什么需要可恢复性调度? 存在希望出现不可恢复调度的情况吗? 解释你的回答。

13.12 什么是无级联调度? 为什么要求无级联调度? 存在希望出现级联调度的情况吗? 解释你的回答。

文献注解

讨论并发控制和恢复的教科书由 Bernstein 等 [1987] 以及 Papadimitrious [1986] 给出。关于并发控制和恢复的实现问题方面的综述报告由 Gray [1978] 给出。Gray 和 Reuter [1993] 在其教科书中全面讨论了事务处理的概念和技术，包括并发控制和恢复问题。Lynch 等 [1994] 是全面讨论原子事务的教科书。

Eswaran 等 [1976] 对可串行化的概念进行了规范，这是同他们对系统 R 的并发控制的工作相联系的。关于可串行化判定的结果参考了 Papadimitrious 等 [1977] 以及 Papadimitrious [1979]。环检测算法可在一般教科书中找到，如 Cormen 等 [1990] 和 Aho 等 [1983]。

Papadimitrious [1979] 证明视图可串行化的判定是 *NP*-完全的。(Aho 等 [1974] 以及 Garey 和 Johnson [1979] 中有关于 *NP*-完全的理论)。

事务处理各个具体方面的参考文献见第 14、15 和 20 章。

第14章 并发控制

在第13章中我们知道了事务最基本的特性之一是隔离性。当数据库中有多个事务并发执行时，事务的隔离性不一定能保持。系统必须对并发事务之间的相互作用加以控制，这是通过称为并发控制机制的各种机制中的一个来实现的。

本章论述的并发控制机制全部基于可串行性这个特性。也就是说，本章论述的机制保证调度是可串行化的。第20章讨论允许非可串行化调度的并发控制机制。本章讲述并发执行事务的管理，并忽略故障。第15章讨论系统如何从故障中恢复。

14.1 基于锁的协议

确保可串行化的方法之一是对数据项的访问以互斥的方式进行，即当一个事务访问某个数据项时，其他任何事务都不能修改该数据项。实现该要求最常用的方法是只允许事务访问当前该事务锁住的数据项。

14.1.1 锁

给数据项加锁的方式有多种。本节只讲两种：

1）共享锁。如果事务 T_i 获得了数据项 Q 上的共享型锁（记为 S），则 T_i 可读 Q 但不能写 Q。

2）排它锁。如果事务 T_i 获得了数据项 Q 上的排它型锁（记为 X），则 T_i 既可读 Q 又可写 Q。

我们要求每个事务都要根据自己将对数据项 Q 进行的操作类型申请适当的锁。该请求是发送给并发控制管理器的。只有在并发控制管理器授予所需锁后，事务才能继续其操作。

对于给定的一个锁类型集合，可在其上按如下方式定义一个相容函数。令 A 与 B 代表任意的锁类型。假设事务 T_i 请求对数据项 Q 加 A 类型锁，而事务 T_j（$T_i \neq T_j$）当前在数据项 Q 上拥有 B 类型锁。尽管数据项 Q 上存在 B 类型锁，但如果事务 T_i 可以立即获得数据项 Q 上的锁，则我们说 A 类型锁与 B 类型锁相容。这样一个函数可以通过矩阵方便地表示出来。本节所用的两类锁的相容关系由图14-1所示的矩阵 comp 给出。该矩阵的一个元素 comp（A，B）具有 true 值当且仅当类型 A 与类型 B 相容。

注意共享型与共享型相容，而与排它型不相容。任何时候，一个具体的数据项上可同时拥有多个（被不同事务拥有的）共享锁。此后的排它锁请求必须等到该数据项上的所有共享锁释放。

	S	X
S	true	false
X	false	false

图14-1 锁相容矩阵 comp

事务通过执行 lock-S（Q）指令来申请数据项 Q 上的共享锁。同样，排它锁通过执行 lock-X（Q）指令来申请。数据项 Q 上的锁通过 unlock（Q）指令来释放。

要访问一个数据项，事务 T_i 必须首先给该数据项加锁。如果该数据项已被另一事务加上了不相容类型的锁，则在所有不相容类型锁被释放之前，并发控制管理器不会授予事务 T_i 锁。因此，T_i 必须等待，直到所有不相容类型锁被释放。

事务 T_i 可以释放先前它加在某个数据项上的锁。注意，一个事务只要还在访问某数据项，它就必须拥有该数据项上的锁。此外，让事务做完对数据项的最后一次访问后立即释放该数据项上的锁也是不现实的，因为这样有可能破坏事务的可串行性。

作为例子，再来考虑第13章中介绍的简化的银行业务系统。设 A 与 B 是事务 T_1 与 T_2 访问的两个帐户。事务 T_1 从帐户 B 转 \$50 到帐户 A 上，定义如下：

T_1: **lock-X**(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A);

事务 T_2 显示帐户 A 与 B 上的总金额（即 $A + B$ 之和），定义如下：

T_2: **lock-S**(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$);

假设帐户 A 与 B 的金额分别为 \$100 与 \$200。如果这两个事务串行执行，即以 T_1, T_2 或 T_2, T_1 的顺序执行，则事务 T_2 将显示值 \$300。然而，如果两个事务并发执行，则有可能出现如图14-2所示的调度1。这种情况下，事务 T_2 将显示 \$250，而这是不对的。出现这种错误的原因是由于事务 T_1 过早释放了数据项 B 上的锁，从而导致事务 T_2 面对不一致的状态。

该调度显示了事务执行的动作以及并发控制管理器授权加锁的时刻。申请加锁的事务在并发控制管理器授权加锁之前不能执行下一个动作，因此，锁的授予必然是在事务申请锁操作与事务下一动作的间隔内。至于在此期间内授权加锁的准确时间则并不重要，我们不妨假设锁就在事务的下一动作前获得。因此，本章余下部分所示调度中我们将去掉并发控制管理器授予锁的那一栏，而让读者自己去推断加锁的时机。

现在假定事务结束后才释放锁。T_3 是对应于 T_1 的事务且延迟了锁释放，定义如下：

T_3: **lock-X**(B);
read(B);
$B := B - 50$;
write(B);

lock-X(A);
read(A);
$A := A + 50$;
Write(A);
unlock(B);
unlock(A).

事务 T_4 是对应于 T_2 的事务且延迟 3 锁释放，定义如下：

T_4: **lock-S**(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

T_1	T_2	并发控制管理器
lock-X(B)		
		grant-X(B,T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A,T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B,T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A,T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

图 14-2　调度 1

可以验证在调度 1 中导致不准确总和 $250 的读写顺序对事务 T_3 与 T_4 来说不再可能出现。我们也可以使用其他一些调度。在这些调度当中，T_4 将不会显示不一致的结果，稍后我们会明白其中缘由。

遗憾的是，使用锁会引起我们所不希望的情形。考察如图 14-3 所示的事务 T_3 与 T_4 的部分调度。由于 T_3 在 B 上拥有排它锁，而 T_4 正在申请 B 上的共享锁，所以 T_4 须等待 T_3 释放 B 上的锁。同样，由于 T_4 在 A 上拥有共享锁，而 T_3 正在申请 A 上的排它锁，所以 T_3 也须等待 T_4 释放 A 上的锁。于是，我们进入了一种哪个事务都不能继续执行的状态，这种状态称为死锁。当死锁发生时，系统必须回滚其中一个事务。一旦某个事务被回滚，该事务锁住的数据项就被解锁，其他事务就可以访问这些数据项，继续自己的执行。我们将在 14.6 节讨论死

锁处理这个问题。

如果不使用封锁，或者如果对数据项进行读写之后立即解锁，那么我们可能会进入不一致状态。另一方面，在申请对另一数据项加锁之前如果不对当前锁住的数据项解锁，则可能会发生死锁。某些情形下，有许多方法可以避免死锁，我们将在 14.1.4 节讨论这些方法。然而，一般而言如果我们为了避免不一致状态而采取封锁，则死锁总是随之而来。产生死锁显然比产生不一致状态要好，因为死锁可以通过回滚事务加以解决，而不一致状态可能引起的现实中的问题是数据库系统所不能处理的。

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

图 14-3　调度 2

我们要求系统中的每一个事务遵从称为封锁协议的一组规则，这些规则规定事务何时对各数据项加锁、解锁。封锁协议限制了可能的调度数目，这些调度组成的集合是所有可能的可串行化调度的一个真子集。我们将讲述几个封锁协议，它们只允许冲突可串行化调度。在此之前，我们需要先给出几个定义。

令 $\{T_0, T_1, \cdots, T_n\}$ 是参与调度 S 的一个事务集。如果存在数据项 Q，T_i 在 Q 上加 A 型锁，T_j 在 Q 上加 B 型锁，且 comp (A, B) = false，则称 T_i 先于 T_j，记为 $T_i \rightarrow T_j$。如果 $T_i \rightarrow T_j$，则该居先意味着在任何等价的串行调度中 T_i 必须出现在 T_j 之前。注意这个图与我们在 13.9.1 节中用于检测冲突可串行性的优先图是类似的。指令之间的冲突对应于锁类型之间的不相容性。

如果 S 是那些遵从封锁协议规则的事务集的可能调度之一，则称调度 S 在给定的封锁协议下是合法的。我们称一个封锁协议保证冲突可串行性，当且仅当对于所有合法的调度而言，由 $\rightarrow$ 所形成的关系是无环的。

14.1.2　锁的授予

当事务申请对一个数据项加某一类型锁，且没有其他事务在该数据项上已加上与此类型相冲突的锁时，则锁可以被授予。然而，必须小心防止出现下面的情形。假设事务 T_2 在某一数据项上持有共享锁，而另一事务 T_1 申请在该数据项上加排它锁。显然，事务 T_1 必须等待事务 T_2 释放共享锁。同时，如果事务 T_3 又申请对该数据项加共享锁，加锁请求与 T_2 在该数据项上所加锁是相容的，因此 T_3 可以被授权加共享锁。此时，T_2 可能释放锁，但 T_1 又必须等待 T_3 完成。可是，如果又有一个新的事务 T_4 申请对该数据项加共享锁，并在 T_3 完成之前被授予锁。事实上，有可能存在一个事务序列，其中每个事务都申请对该数据项加共享锁，且每个事务在授权加锁后一小段时间内释放锁，这样事务 T_1 总是不能在该数据项上加排它锁，事务 T_1 也就永远不能取得进展。这称为“饿死”。

可以按如下方式授权加锁来避免事务饿死。当事务 T_i 申请对数据项 Q 加 M 型锁时，授权加锁的条件是：

1）不存在在数据项 Q 上持有与 M 型锁冲突的锁的其他事务。

2）不存在等待对数据项 Q 加锁且先于 T_i 申请加锁的事务。

14.1.3　两阶段封锁协议

保证可串行性的一个协议是两阶段封锁协议。该协议要求每个事务分两个阶段提出加锁和解锁申请：

1）增长阶段。事务可以获得锁，但不能释放锁。

2）缩减阶段。事务可以释放锁，但不能获得新锁。

一开始，事务处于增长阶段，事务根据需要获得锁。一旦该事务释放了锁，它就进入了缩减阶段，不能再发出加锁请求。

例如，前面所说事务 T_3 与 T_4（见14.1.1节）是两阶段的；同时，事务 T_1 与 T_2 不是两阶段的。注意，解锁指令不必非得出现在事务末尾。例如，就事务 T_3 而言，可以把 unlock（B）指令紧跟指令 lock-X（A）之后，这样仍然保持了两阶段封锁特性。

我们可以证明两阶段封锁协议保证冲突可串行性。对于任何事务，调度中该事务获得其最后加锁的时刻（增长阶段结束点）称为事务的封锁点。这样，多个事务可以根据它们的封锁点进行排序——实际上，这个顺序就是事务的一个可串行性次序。我们将此证明留作练习（见习题14.1）。

两阶段封锁并不保证不会发生死锁。不难发现事务 T_3 与 T_4 是两阶段的，但在调度2中（图14-3所示），它们却发生死锁。

在13.6.2节中，我们曾除了希望调度可串行化外，还希望它们是无级联的。在两阶段封锁协议下，级联回滚可能发生。作为示例，来看如图14-4所示的部分调度。每个事务都遵从两阶段封锁协议，但在事务 T_7 的 read（A）指令之后事务 T_5 发生故障，从而导致 T_6 与 T_7 级联回滚。

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

图14-4 在两阶段封锁下的部分调度

级联回滚可以通过将两阶段封锁修改为严格两阶段封锁协议加以避免。严格两阶段封锁协议除了要求封锁是两阶段之外，还要求事务持有的所有排它锁必须在事务提交后方可释放。这个要求保证未提交事务所写的任何数据在该事务提交之前均以排它方式加锁，防止了其他事务读取这些数据。

另一个两阶段封锁的变体是强两阶段封锁协议，它要求事务提交之前不得释放任何锁。很容易验证在强两阶段封锁条件下，事务可以按其提交的顺序串行化。大部分数据库系统要么采用严格两阶段封锁要么采用强两阶段封锁。

来看下面两个事务，我们只写出了其中较为重要的 read 与 write 指令。事务 T_8 定义为：

$$
\begin{aligned}
T_8: \quad & \textbf{read}(a_1);\\
& \textbf{read}(a_2);\\
& \cdots\\
& \textbf{read}(a_n);\\
& \textbf{write}(a_1).
\end{aligned}
$$

事务 T_9 定义为：

$$
\begin{aligned}
T_9: \quad & \textbf{read}(a_1);\\
& \textbf{read}(a_2);\\
& \textbf{display}(a_1+a_2).
\end{aligned}
$$

如果采用两阶段封锁协议，则 T_8 必须对 a_1 加排它锁。因此两个事务的任何并发执行方式都相当于串行执行。注意，T_8 只需在其执行结束写 a_1 时才对 a_1 加排它锁。因此，如果 T_8 开始时就对 a_1 加共享锁，然后在需要时将其变更为排它锁，那么我们可以获得更多的并发性，

这也因为 T_8与 T_9 可以同时访问 a_1 与 a_2。

以上示例告诉我们对基本的两阶段封锁协议加以修改，就能执行锁转换。我们将提供一种将共享锁提升为排它锁以及将排它锁降级为共享锁的机制。我们用 upgrade 表示将共享锁提升为排它锁，用 downgrade 表示将排它锁降级为共享锁。锁转换不能随意进行，锁提升只能发生在增长阶段，而锁降级只能发生在缩减阶段。

再看上面的例子，事务 T_8 与 T_9 可在修改后的两阶段封锁协议下并发执行，如图 14-5 所示，其中的调度是不完整的，它只给出了一些封锁指令。

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

图 14-5　带有锁转换的不完整调度

注意，事务者要提升数据项 Q 上的锁吮可能不得不等待。这种情况发生在另一个事务当前对 Q 持有共享锁时。

与基本的两阶段封锁协议一样，具有锁转换的两阶段封锁协议只产生冲突可串行化调度，并且事务可以根据其封锁点做串行化。此外，如果排它锁直到事务结束时才释放，则调度是无级联的。

现在介绍一个简单却广泛使用的机制，它自动为事务产生加锁、解锁指令。当事务 T_i 进行 read(Q)操作时，系统就产生一个 lock-S(Q)指令，后接 read（Q）指令。当事务 T_i 进行 write（Q）操作时，系统就检查 T_i 是否已在 Q 上持有共享锁。若有，则系统发出 upgrade（Q）指令，后接 write（Q）指令，否则系统发出 lock-X（Q）指令，后接 write（Q）指令。当一个事务提交或回滚后，该事务持有的所有锁都被释放。

对于一个事务集，可能存在某些不能通过两阶段封锁协议得到的冲突可串行化调度。然而，如果想通过非两阶段封锁协议得到冲突可串行化调度，我们或者需要事务的附加信息，或者要求数据库中的数据项集合有一定的结构或顺序。没有这些信息，两阶段封锁对于冲突可串行性就是必不可少的——如果 T_i 是一个非两阶段事务，则总可以找到另一两阶段事务 T_j，且 T_i与 T_j 的某个调度是非冲突可串行化的。

严格两阶段封锁与强两阶段封锁（含锁转换）在商用数据库系统中被广泛使用。

14.1.4　基于图的协议

如前所述，在缺少有关数据项存取方式的信息时，两阶段封锁协议对保证可串行化来说不仅是必要的而且是充分的。因此，如果要开发非两阶段协议，就需要有关事务如何存取数据库的附加信息。不同模型所需信息量的大小不同，最简单的模型要求我们事先知道访问数据项的顺序。在已知这些信息情况下，构造非两阶段封锁协议是可能的，并且这样的协议保证冲突可串行性。

为获取这些事先知识，要求数据项集合 D= $\{d_1, d_2, \cdots, d_n\}$ 的所有数据项满足偏序 $\rightarrow$。如果 $d_i \rightarrow d_j$，则任何既访问 d_i 又访问 d_j 的事务必须首先访问 d_i，然后访问 d_j。这种偏序可以是数据的逻辑或物理组织的结果，也可以只是为了并发控制而加上的。

偏序意味着集合 D 可被视为有向无环图，称为数据库图。本节为了简单起见，只关心那些带根树。我们将给出一个称为树形协议的简单协议，该协议只使用排它锁。其他更复杂的基于图的封锁协议可以参阅文献注解中给出的有关文献。

在树形协议中，可用的加锁指令只有 lock-X。每个事务 T_i 对一个数据项最多能加一次锁，并且必须遵从以下规则：

1）T_i 的首次加锁可以对任何数据项进行。

2）此后，T_i 对数据项 Q 加锁的前提是 T_i 持有 Q 的父项上的锁。

3）对数据项解锁可以随时进行。

4）数据项被 T_i 加锁并解锁后，T_i 不能再对该数据项加锁。

如前所述，所有满足树形协议的调度是冲突可串行化的。

为了说明这个协议，来看如图 14-6 所示的数据库图。图下的 4 个事务遵从该图的树形协议。我们只列出了加锁、解锁指令：

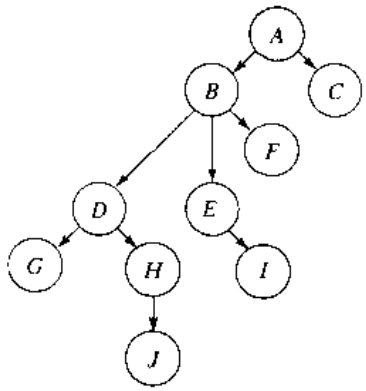


图 14-6　具有树形结构的数据库图

T_{10}: lock-X (B); lock-X (E); lock-X (D); unlock (B); unlock (E); lock-X (G); unlock (D); unlock (G).

T_{11}: lock-X (D); lock-X (H); unlock (D); unlock (H).

T_{12}: lock-X (B); lock-X (E); unlock (E); unlock (B).

T_{13}: lock-X (D); lock-X (H); unlock (D); unlock (H)

这 4 个事务参与的一个可能的调度如图 14-7 所示。注意，执行过程中事务 T_{10}持有两棵分离子树的锁。

不难发现图 14-7 所示的调度是冲突可串行化的。可以证明树形协议不仅保证冲突可串行性，而且保证不会产生死锁。

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

图 14-7　在树型协议下的可串行化调度

树形封锁协议可以在较早时候释放锁，这点优于两阶段封锁协议。较早解锁减少了等待时间，增加了并发性。此外，树形协议不会产生死锁，毋需回滚。然而，该协议也有其缺点，有时事务必须给那些根本不访问的数据项加锁。例如，某事务要访问图 14-6 所示数据库图中的数据项 A 与 J，则该事务不仅要给 A 与 J 加锁，而且还要给 B、D、H 加锁。这种额外的封锁不仅导致封锁开销增加及可能额外的等待时间，而且还会使并发性降低。此外，如果事先没有得到哪些数据项需要加锁的知识，事务就必须给树根加锁，这会大大降低并发性。

对于一事务集，可能存在某些不能通过树形封锁协议得到的冲突可串行化调度。事实上，一些在两阶段封锁协议中可行的调度在树形封锁协议中是不可行的，反之亦然。习题中对这种调度的例子进行了探讨。

14.2　基于时间戳的协议

到此为止我们所讲述的封锁协议中，每一对冲突事务的次序是在执行时由第一个两者都申请的，但与其他类型不相容的锁决定的。另一种决定事务可串行化次序的方法是事先选定事务的次序，其中最常用的方法是时间戳排序机制。

14.2.1　时间戳

对于系统中每个事务 T_i，我们把一个唯一的固定的时间戳和它联系起来，此时间戳记为 TS（T_i）。该时间戳是在事务 T_i 开始执行前由数据库系统赋予的。若事务 T_i 已被赋予时间戳 TS（T_i），同时有一新事务 T_j 进入系统，则 TS（T_i）< TS（T_j）。实现这种机制可以采用下面两个简单的方法：

1）使用*系统时钟值*作为时间戳；即事务的时间戳等于该事务进入系统时的时钟值。

2）使用*逻辑计数器*，每赋予一个时间戳，计数器增加一次。即事务的时间戳等于该事务进入系统时的计数器值。

事务的时间戳决定了串行化顺序。因此，若 TS（T_i）<TS（T_j），则系统必须保证所产生的调度等价于事务 T_i 出现在事务 T_j 之前的某个串行调度。

要实现这个机制，每个数据项 Q 需要与两个时间戳相关联：

- W-timestamp（Q）表示成功执行 write（Q）的所有事务的最大时间戳。
- R-timestamp（Q）表示成功执行 read（Q）的所有事务的最大时间戳。

每当有新的 read（Q）或 write（Q）指令执行时，这些时间戳就被更新。

14.2.2　时间戳排序协议

时间戳排序协议保证任何有冲突的 read 和 write 操作按时间戳顺序执行。该协议运作方式如下：

1）假设事务 T_i 发出 read（Q）操作：

a. 若 TS（T_i）<W-timestamp（Q），则 T_i 需读入的 Q 值已被覆盖。因此，read 操作被拒绝，T_i 回滚。

b. 若 TS（T_i）≥W-timestamp（Q），则执行 read 操作，R-timestamp（Q）被设为 R-timestamp（Q）与 TS（T_i）两者中的最大值。

2）假设事务 T_i 发出 write（Q）操作：

a. 若 TS（T_i）< R-timestamp（Q），则 T_i 产生的 Q 值是先前所需要的值，且系统已

假定该值不会被产生。因此，write操作被拒绝，T_i 回滚。

b. 若 TS（T_i）< W-timestamp（Q），则 T_i 想写入的 Q 值已过时。因此，write 操作被拒绝，T_i 回滚。

c. 其他情况时执行 write 操作，将 W-timestamp（Q）设为 TS（T_i）。

事务 T_i 如果由于发出 read 或 write 操作而被并发控制机制滚回，则被赋予新的时间戳并重新启动。

为说明这个协议，考虑事务 T_{14}与 T_{15}。事务 T_{14}显示帐户 A 与 B 的内容，定义如下：

T_{14}: **read**(B);
read(A);
display($A+B$).

事务 T_{15}从帐户 A 转 $50 到帐户 B，然后显示两个帐户的内容，定义如下：

T_{15}: **read**(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A+B$).

在说明时间戳协议下的调度时，假设事务在第一条指令执行之前的那一刻被赋予时间戳。因此，如图 14-8 所示的调度 3 中，TS（T_{14}）<TS（T_{15}），同时该调度是满足时间戳协议的一个可能的调度。

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A+B$)	
	$A := A + 50$
	write(A)
	display($A+B$)

图 14-8 调度 3

我们注意到上述执行过程也可以由两阶段封锁协议产生。不过，存在满足两阶段封锁协议却不满足时间戳协议的调度，反之亦然（见习题 14.20）。

时间戳排序协议保证冲突可串行化，这一断言来自冲突操作按时间戳顺序进行处理这一事实。该协议保证无死锁，因为不存在等待的事务。协议可能产生不可恢复的调度，然而协议可以进行扩展，保证调度可恢复且无级联，这可以通过只在事务末尾执行写操作或使用一种受限封锁形式来实现（见习题 14.23）。

14.2.3 Thomas 写规则

现在给出对时间戳排序协议的一种修改，它允许的并发程度比上节中的协议要高。让我们来看图 14-9 所示调度 4，并应用时间戳排序协议。由于 T_{16}先于 T_{17}开始，我们可假定 TS（T_{16}）<TS（T_{17}）。若 T_{16}的 read（Q）操作成功，T_{17}的 write（Q）操作也成功。当 T_{16}试图进行 write（Q）操作时，TS（T_{16}）< W-timestamp（Q），因为 W-timestamp（Q） TS（T_{17}）。所以，T_{16}的 write（Q）操作被拒绝且事务 T_{16}回滚。

虽然事务 T_{16}回滚是时间戳排序协议所要求的，但也是不必要的。因为 T_{17}已经写入了 Q，

T_{16}想要写入的值将永远不会被读到。满足 TS（T_i）<TS（T_{17}）的任何事务 T_i 试图进行 read（Q）操作时均被回滚，因为 TS（T_i）<W-timestamp（Q）。满足 TS（T_j）>TS（T_{17}）的任何事务 T_j 必须读由 T_{17}写入的 Q 值，而不是由 T_{16}写入的值。

根据以上分析，可以修改时间戳排序协议而得到一个新版本协议，该协议在某些特定的情况下忽略过时的 write 操作。协议中有关 read 操作的规则保持不变，但有关 write 操作的规则与上节中的时间戳排序协议略有区别。

假设事务 T_i 发出 write（Q）操作：

1）若 TS（T_i）<R-timestamp（Q），则 T_i 产生的 Q 值是先前所需要的值，但系统已假定该值不会被产生。因此，write 操作被拒绝，T_i 回滚。

2）若 TS（T_i）<W-timestamp（Q），则 T_i 试图写入的 Q 值已过时。因此，这个 write 操作可被忽略。

3）其他情况时执行 write 操作，将 W-timestamp（Q）设为 TS（T_i）。

以上规则与上节中的规则区别在第二条。时间戳排序协议在 T_i 发出 write（Q）操作且 TS（T_i）<W-timestamp（Q）时要求 T_i 回滚。修改后的协议对这种情况的处理是，在 TS（T_i）≥R-timestamp（Q）时，忽略过时的 write 操作。对时间戳排序协议所做的这种修改称为 Thomas 写规则。

Thomas 写规则实际上是通过删除事务发出的过时 write 操作来利用视图可串行性。对事务的这种修改使得系统可以产生本章中其他协议所不能产生的可串行化调度。例如，图 14-9 所示调度 4 是非冲突可串行化的，因此该调度在两阶段封锁协议、树形封锁协议或时间戳排序协议中都是不可能的。在 Thomas 写规则下，T_{16}的write(Q)操作将被忽略，产生视图等价于串行调度<T_{16}，T_{17}>的一个调度。

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

图 14-9　调度 4

14.3　基于有效性检查的协议

在大部分事务是只读事务的情况下，事务发生冲突的频率较低。因此，许多这样的事务，即使在没有并发控制机制监控的情况下执行，也不会破坏系统的一致状态。并发控制机制会带来代码执行开销及可能的事务延迟，采用开销较小的机制是我们所希望的。减小开销面临的困难是我们事先并不知道哪些事务将陷入冲突中。为获得这些知识，需要一种监控系统的机制。

我们假定事务 T_i 在其生命周期中按两个或三个阶段执行，这取决于该事务是一个只读事务还是一个更新事务。这些阶段如下顺序列出：

1）读阶段。在这一阶段中执行事务 T_i。各数据项值被读入并保存在事务 T_i 的局部变量中。所有 write 操作都是对局部临时变量进行的，并不对数据库进行真正的更新。

2）有效性检查阶段。事务 T_i 进行有效性检查，判定是否可以将 write 操作所更新的临时局部变量值拷入数据库同时又不违反可串行性。

3）写阶段。若事务 T_i 已在第二步通过有效性检查，则实际的更新就可写入数据库中；否则，事务 T_i 回滚。

每个事务必须按以上顺序经历三个阶段。然而，并发执行诸事务的三个阶段可以是交叉执行的。

读、写阶段毋需更多说明，需要进一步讨论的是有效性检查阶段。为进行有效性检测，需要知道事务 T_i 的各个阶段何时进行。为此，我们将三个不同的时间戳与事务 T_i 相关联：

1）Start（T_i）。事务 T_i 开始执行的时间。

2）Validation（T_i），事务 T_i 完成读阶段并开始其有效性检查阶段的时间。

3）Finish（T_i）。事务 T_i 完成写阶段的时间。

我们利用时间戳 Validation（T_i）的值通过时间戳排序技术决定可串行性顺序。因此，值 TS（T_i）＝Validation（T_i）并且若 TS（T_j）＜TS（T_k），则产生的任何调度必定等价于事务 T_j 出现在事务 T_k 之前的某个串行调度。我们选择 Validation（T_i）而不是 Start（T_i）作为事务 T_i 的时间戳是因为在冲突频度很低的情况下可能有更快的响应时间。

事务 T_j 的有效性测试要求满足 TS(T_i)＜TS(T_j)的事务 T_i 必须满足下面两条件之一：

1）Finish（T_i）＜Start（T_j）。因为 T_i 在 T_j 开始之前完成其执行，故可串行性次序得到了保持。

2）T_i 所写数据项集与 T_j 所读数据项集不相交，并且 T_i 的写阶段在 T_j 开始其有效性检查阶段之前完成（Start（T_j）＜Finish（T_i）＜Validation（T_j））。这个条件保证 T_i 与 T_j 的写不重叠。因为 T_i 写不影响 T_j 读，又因为 T_j 不影响 T_i 读，所以保持了可串行性次序。

作为例子，我们再次考虑事务 T_{14} 与事务 T_{15}。假设 TS(T_{14})＜TS(T_{15})。则有效性检查阶段成功地产生调度 5，如图 14-10 所示。注意，只有在 T_{15} 的有效性检查阶段执行之后才写实际的变量。因此，T_{14} 读取 B 与 A 的旧值，且该调度是可串行化的。

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
(*validate*)	
display($A + B$)	
	(*validate*)
	write(B)
	write(A)

图 14-10　调度 5——使用 validation 得到的一个调度

有效性检查机制自动预防级联回滚，因为只有发出写操作的事务提交后实际的写才发生。

14.4　多粒度

以上所讲的并发控制机制中，我们把一个个数据项作为可执行同步的单元。

然而，某些情况下我们需要把多个数据项聚为一组，将它们作为一个同步单元，因为这样效果可能更好。例如，如果事务 T_i 需要访问整个数据库，并使用一种封锁协议，则事务 T_i 必须给数据库中的每个数据项加锁。显然，执行这些加锁操作是很费时的。要是 T_i 能够发出一个封锁整个数据库的加锁请求就更好了。另一方面，如果事务 T_i 只需存取少量数据项，就不应要求 T_i 给整个数据库加锁，否则并发性就丧失了。

我们需要的是一种允许系统定义多级粒度的机制。通过允许存在各种大小的数据项并定义数据粒度的层次结构，其中小粒度数据项嵌套在大粒度数据项中，我们可以构造出这样一种机制。这种层次结构可以图形化地表示为树。注意，这里所描述的树与树形协议（14.1.4 节）所用的树的概念完全不同。多粒度树中的非叶结点表示了与其后代相联系的数据。在树形协议中，每个结点是一个相互独立的数据项。

例如图 14-11 所示的树，它由四层结点组成。最高层表示整个数据库，其下是区域类型的结点，数据库恰好由这些域组成。每个域又以文件类型结点作为子结点，每个区域恰好由作为其子结点的文件结点组成。任何文件都不能处于两个或两个以上区域中。最后，每个文件由记录类型的结点组成。和前面一样，文件恰好由作为其子结点的记录组成，并且任何记录不能同

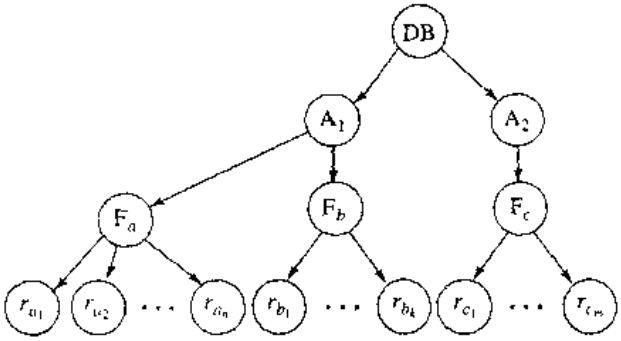


图 14-11　粒度层次图

时属于多个文件。

树中每个结点都可以单独加锁。正如在两阶段封锁协议中所做的那样，我们将使用共享锁与排它锁。当事务对一个结点加锁（共享锁或排它锁），该事务也以同样类型的锁封锁这个结点的全部后代结点。例如，若事务 T_i 显式给图 14-11 中的文件 F_c 加排它锁，则事务 T_i 也隐含地给所有属于该文件的记录加排它锁，这里没有必要显式地给 F_c 的记录逐个加锁。

假设事务 T_j 希望封锁文件 F_b 的记录 r_{b_6}。由于 T_i 显式地给 F_b 加锁，意味着 r_{b_6} 也被加锁（隐式地）。但是，当 T_j 发出对 r_{b_6} 加锁的请求时，r_{b_6} 却没有被显式加锁！那么系统如何判定 T_j 是否可以封锁 r_{b_6} 呢？T_j 必须从树根走到 r_{b_6}，只要发现此路径上某个结点的锁与要加的锁类型不相容，则 T_j 必须延迟。

现假设事务 T_k 希望封锁整个数据库。为此，它只需给层次结构图的根结点加锁。不过，注意 T_k 给根结点加锁不会成功，因为目前 T_i 在树的某部分持有锁（具体地说，对文件 F_b 持有锁）。但是，系统怎样判定根结点是否可以加锁呢？一种可能的方法是搜索整棵树，可惜这个方法破坏了多粒度封锁机制的初衷。获取这种知识的一个更有效的方法是引入一种新的锁类型，即意向锁。如果一个结点加上了意向锁，则意味着要在树的较低层进行显式加锁（也就是说，以更小的粒度加锁）。在一个结点显式加锁之前，该结点的全部祖先结点均加上了意向锁。因此，事务在判定能否成功地给一个结点加锁时不必去搜索整棵树。希望给某个结点如 Q 加锁的事务必须遍历从根到 Q 的路径，在此过程中，该事务给各结点加上意向锁。

与共享锁相关联的是一种意向锁，与排它锁相关联的是另一种意向锁。如果一个结点加上了共享型意向（IS）锁，那么将在树的较低一层进行显式封锁，但只能加共享锁。同样，如果一个结点加上了排它型意向（IX）锁，那么将在树的较低一层进行显式封锁，此时排它锁或共享锁均可。最后，若一个结点加上了共享排它型意向（SIX）锁，则以该结点为根的子树被显式地加了共享锁，并且将在树的较低一层显式地加排它锁。这些锁类型的相容函数如图 14-12 所示。

下面的多粒度封锁协议保证可串行性。每个事务 T_i 可按如下规则对数据项 Q 加锁：

1）必须遵从图 14-12 所示的锁类型的相容函数。

2）根结点必须首先加锁，且可以加任意类型的锁。

3）仅当 T_i 当前对 Q 的父结点持有 IX 或 IS 型锁时，T_i 对结点 Q 才可加 S 或 IS 型锁。

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

图 14-12　相容矩阵

4）仅当 T_i 当前对 Q 的父结点持有 IX 或 SIX 型锁时，T_i 对结点 Q 才可加 X、SIX 或 IX 型锁。

5）仅当 T_i 未曾对任何结点解锁时，T_i 才可对结点加锁（也就是说，T_i 是两阶段的）。

6）仅当 T_i 当前不持有 Q 的子结点的锁时，T_i 才可对结点 Q 解锁。

注意，多粒度协议要求加锁按自顶向下的顺序（根到叶），锁的释放则按自底向上的顺序（叶到根）。

作为该协议的例子，来看图 14-11 所示的树及下面的事务：

- 假设事务 T_{18} 读文件 F_a 的记录 r_{a_2}。那么，T_{18} 需给数据库、区域 A_1、以及 F_a 加 IS 锁（按此顺序），最后给 r_{a_2} 加 S 锁。
- 假设事务 T_{19} 要修改文件 F_a 的记录 r_{a_9}。那么，T_{19} 需给数据库、区域 A_1、以及文件 F_a 加 IX 锁，最后给记录 r_{a_9} 加 X 锁。
- 假设事务 T_{20} 要读取文件 F_a 的所有记录。那么，T_{20} 需给数据库和区域 A_1（按此顺序）加

IS锁，最后给 F_a 加S锁。

• 假设事务 T_{21}要读取整个数据库。它在给数据库加S锁后就可读取。

我们注意到事务 T_{18}、T_{19}与 T_{21}可以并发地存取数据库。事务 T_{19}可以与 T_{18}并发执行，但不能与 T_{20}或 T_{21}并发执行。

该协议增强了并发性，减少了锁开销。该协议在由如下事务类型混合而成的应用中尤其有用：

• 只存取几个数据项的短事务。

• 由整个文件或一组文件来形成报表的长事务。

还有其他相类似的封锁协议可用于数据粒度按有向无环图组织的数据库系统中，相应参考文献见文献注解。与在两阶段封锁协议中一样，这里所讲述的协议中也可能存在死锁。有多种技术可用来减少多粒度协议中死锁发生的频度，甚至可以彻底消除死锁。文献注解中给出了这些技术的相关资料。

14.5 多版本机制

目前为止我们讲述的并发控制机制要么延迟一项操作要么中止发出该操作的事务来保证可串行性。例如，read操作可能由于相应值还未写入而延迟，或因为它要读取的值已被覆盖而被拒绝执行（即发出read操作的事务被中止）。如果每一数据项的旧值拷贝被保存在系统中，这些问题就可以避免。

在多版本数据库系统中，每个write（Q）操作创建 Q 的一个新版本。当发出read（Q）操作时，系统选择 Q 的一个版本进行读取。并发控制机制必须保证用于读取的版本的选择能保持可串行性。出于性能考虑，一个事务能容易而快速地判定读取哪个版本的数据项也是很关键的。

14.5.1 多版本的时间戳排序

多版本机制中最常用的技术是时间戳。对于系统中的每个事务 T_i，我们将一个静态的唯一的时间戳与之关联，记为TS（T_i）。如14.2节所述，该时间戳在事务开始前赋予。

对于每个数据项 Q，有一个版本序列$<Q_1, Q_2, \cdots, Q_m>$与之关联。每个版本 Q_k 包含三个数据字段：

• Content是 Q_k 的版本值。

• W-timestamp（Q）是创建 Q_k 版本的事务的时间戳。

• R-timestamp（Q）是所有成功读取 Q_k 版本的事务的最大时间戳。

事务（如 T_i）通过发出write（Q）操作创建数据项 Q 的一个新版本 Q_k。版本的Content字段保存事务 T_i 写入的值。W-timestamp与R-timestamp初始化为TS（T_i）。每当事务 T_j 读取 Q_k 的值且R-timestamp（Q_k）$<$TS（T_j）时，R-timestamp的值就被更新。

下面的多版本时间戳机制保证可串行性，该机制运作如下。假设事务 T_i 发出read（Q）或write（Q）操作，令 Q_k 表示 Q 的版本，其写时间戳小于或等于TS（T_i）的最大写时间戳。

1）如果事务 T_i 发出read（Q）操作，则返回值是 Q_k 的内容。

2）如果事务 T_i 发出read（Q）操作且若TS（T_i）$<$R-timestamp（Q_k），则事务 T_i 回滚，否则，若TS（T_i）$=$W-timestamp（Q_k），则 Q_k 的内容被覆盖；否则，创建 Q 的一个新版本。

规则1）的理由是显然的，一个事务读取在它之前的最近版本。规则2）则说明当一个事务执行写操作“太迟”时，将迫使该事务中止。更确切地说，如果 T_i 试图写入其他事务读取的版本，则不允许该写操作成功。

不再需要的版本根据以下规则删除。假设有某数据项的两个版本 Q_k 与 Q_j，这两个版本的 W-timestamp 都小于系统中最老的事务的时间戳，那么 Q_k 和 Q_j 中较旧的版本将不再会被用到，因而可以删除。

多版本时间戳排序机制有一个很好的特性：读请求从不失败且不必等待。在典型的数据库系统中，读操作比写操作频繁，因而这个优点对于实践来说至关重要。

然而这个机制也存在两个不好的特性。首先读取数据项要求更新 R-timestamp 字段，于是产生两次潜在的磁盘访问而不是一次。其次，事务间的冲突通过回滚解决而不是等待，这种做法开销可能很大。下节讲述一个可以减轻这个问题的算法。

14.5.2　多版本两阶段封锁

多版本两阶段封锁协议可将多版本并发控制的优点与两阶段封锁的优点结合起来。该协议对只读事务与更新事务加以区分。更新事务执行强两阶段封锁协议，即它们持有全部锁直到事务结束。因此，它们可以按提交的次序进行串行化。数据项的每个版本有一个时间戳，这种时间戳不是真正基于时钟的时间戳，而是一个计数器，我们称之为 ts _ counter，这个计数器在事务提交时增加计数。

在只读事务开始执行前，我们读取 ts _ counter 的当前值来作为该事务的时间戳。只读事务在执行读操作时遵从多版本时间戳排序协议。因此，当只读事务 T_i 发出 read（Q）时，返回值是时戳小于 TS（T_i）时间戳的最大版本的内容。

当更新事务读取一个数据项时，它在获得该数据项上的共享锁后读取该数据项最新版本的值。当更新事务想写一个数据项时，它首先要获得该数据项上的排它锁，然后为此数据项创建一个新版本，写操作在新版本上进行。新版本的时间戳最初置为∞，它大于任何可能的时间戳值。

当更新事务 T_i 完成其任务后，它按如下方式进行提交。首先，T_i 将它创建的每一版本的时间戳设为 ts _ counter + 1。然后，T_i 将 ts _ counter 增加 1。这里一次只允许有一个更新事务进行提交。

这样，在 ts _ counter 增加之后启动的只读事务将看到被 T_i 更新的值，而那些在 ts _ counter 增加之前就启动的只读事务将看到被 T_i 更新之前的值。无论哪种情况，只读事务均不必等待加锁。

版本删除类似于多版本时间戳排序协议中采用的方式。假设有某数据项的两个版本 Q_k 与 Q_j，两个版本的时间戳都小于系统中最老的只读事务的时间戳，那么两个版本中较旧的版本将不再会被使用，因而可以删除。

多版本两阶段封锁协议或其变种已用于某些商用数据库系统中。

14.6　死锁处理

如果存在一个事务集，该集合中的每个事务都在等待该集合中的另外一个事务，那么我们说系统处于死锁状态。更确切地说，存在一个等待事务集 $\{T_0, T_1, \cdots, T_n\}$，其中 T_0 正等待被 T_1 锁住的数据项，T_1 正等待被 T_2 锁住的数据项，…，T_{n-1} 正等待被 T_n 锁住的数据项，且 T_n 正等待被 T_0 锁住的数据项。这种情况下，没有一个事务能取得进展。此时，系统唯一的补救措施是采取激烈的动作，如回滚某些陷于死锁的事务。

处理死锁问题有两种主要的方法，其一我们可以使用死锁预防协议保证系统永不进入死锁状态。其二我们允许系统进入死锁状态，然后试着用死锁检测与死锁恢复机制进行恢复。如下所

述，两种方法均会引起事务回滚。如果系统进入死锁状态的概率相对较高，则通常使用死锁预防机制；否则，使用死锁检测与死锁恢复机制会更有效。

注意，检测与恢复机制带来各种开销，不仅包括在运行时维护必要信息及执行检测算法的代价，而且包括从死锁中恢复所固有的潜在损失。

14.6.1 死锁预防

预防死锁有两种方法。第一种方法是通过对加锁请求进行排序或要求同时获得所有锁来保证不会发生循环等待。另一种方法比较接近死锁恢复，每当等待有可能导致死锁时，进行事务回滚而不是等待加锁。

第一种方法的最简单的机制是要求每个事务在开始之前封锁它的所有数据项。并且，要么一次全部封锁要么全不封锁。这个协议有两个主要缺点。首先，在事务开始前通常很难预知哪些数据项需要封锁。其次，数据项使用率可能很低，因为许多数据项虽被封锁但长时间不被用到。

防止死锁的另一种机制是对所有的数据项定义一个偏序，同时要求事务只能按偏序规定的顺序封锁数据项。我们曾经在树形协议中讲述过这样一个机制。

防止死锁的第二种方法是使用抢占与事务回滚。在抢占机制中，若事务 T_2 所申请的锁已被事务 T_1 持有，则授予 T_1 的锁可能通过回滚事务 T_1 被抢占，并授予 T_2。为控制抢占，我们给每个事务赋与一个唯一的时间戳，系统使用时间戳来决定事务应当等待还是回滚。并发控制仍使用封锁机制。若一个事务回滚，该事务重启时保持原有的时间戳。利用时间戳的两种不同的死锁预防机制已被提出：

1）wait-die 机制基于非抢占技术。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 T_i 的时间戳小于 T_j 的时间戳（即 T_i 比 T_j 老）时，允许 T_i 等待。否则，T_i 回滚（死亡）。例如，假设事务 T_{22}、T_{23}、及 T_{24}的时间戳分别为 5、10、与 15。如果 T_{22}申请的数据项当前被 T_{23}持有，则 T_{22}将等待。如果 T_{24}申请的数据项当前被 T_{23}持有，则 T_{24}将回滚。

2）wound-wait 机制基于抢占技术，是与 wait-die 相反的机制。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 T_i 的时间戳大于 T_j 的时间戳（即 T_i 比 T_j 年轻）时，允许 T_i 等待。否则，T_j 回滚（T_j损伤了 T_i）。再看上面的例子，对于事务 T_{22}、T_{23}及 T_{24}，如果 T_{22}申请的数据项当前被 T_{23}持有，则 T_{22}将从 T_{23}抢占该数据项，T_{23}回滚。如果 T_{24}申请的数据项当前被 T_{23}持有，T_{24}将等待。

只要存在事务回滚，保证无饿死发生就已非常重要，即必须保证不存在某个事务重复回滚而总是不能取得进展。

wound-wait 与 wait-die 机制两者均可避免饿死；因为任何时候均存在一个时间戳最小的事务，在两个机制中，这个事务都不允许回滚。由于时间戳总是增长，而回滚的事务不赋予新时间戳，因此被回滚的事务最终会变成最小时间戳事务，从而不会再次回滚。

然而，两个机制的运作方式有明显区别：

- 在 wait-die 机制中，较老的事务必须等待较新的事务释放它所持有的数据项。因此，事务变得越老，它越要等待。相反，在 wound-wait 机制中，较老的事务从不等待较新的事务。
- 在 wait-die 机制中，如果事务 T_i 由于申请的数据项当前被 T_j 持有而死亡并回滚，则当事务 T_i 重启时它可能重新发出相同的申请序列。如果该数据项仍被 T_j 持有，则 T_i 将再度死亡。因此，T_i 在获得所需数据项之前可能多次死亡。将这一系列事件与 wound-wait 机制中发生的事件相比较，后者机制中事务 T_j 申请的数据项当前被 T_i 持有引起事务 T_i 受损与回滚；当事务 T_i

重启并申请当前被 T_j 持有的数据项时，T_i 等待。因此，在 wound-wait 机制中可能回滚较少。

两种机制都面临的主要问题是可能发生不必要的回滚。

14.6.2 基于超时的机制

另一种处理死锁的简单方法基于锁超时。这种方法中，申请锁的事务至多等待一个给定的时间。若在此期间内锁未授予该事务，则称该事务超时，此时该事务自己回滚并重启。如果确实存在死锁，卷入死锁的一个或多个事务将超时并回滚，从而使其他事务继续。该机制介于死锁预防（不会发生死锁）与死锁检测及恢复之间，死锁检测与恢复在下节讲述。

超时机制的实现极其容易，并且在事务是短事务或长时间等待是由死锁引起时该机制运作良好。但是，一般而言很难确定一个事务超时之前应等待多长时间。如果已发生死锁，则等待时间太长而导致不必要的延迟。如果等待时间太短，即便没有死锁，也可能引起事务回滚，造成资源浪费。该机制还可能会产生饥饿。故此，基于超时的机制应用不多。

14.6.3 死锁检测与恢复

如果系统没有采用能保证不产生死锁的协议，那么系统必须采用检测与恢复机制。检查系统状态的算法周期性地被激活，判断有无死锁发生。如果发生死锁，则系统必须试着从死锁中恢复。为实现这一点，系统必须：

- 维护当前分配给事务的数据项的有关信息以及任何尚未解决的数据项请求信息。
- 提供一个使用这些信息判断系统是否进入了死锁状态的算法。
- 如果检测算法判定存在死锁，则从死锁中恢复。

以下详细讲述上述问题。

1. 死锁检测

死锁可以用称为等待图的有向图来精确描述。该图由两部分 $G=(V, E)$ 组成，其中 V 是顶点集，E 是边集。顶点集由系统中的所有事务组成。边集 E 的每一元素是一个有序对 $T_i \rightarrow T_j$。如果 $T_i \rightarrow T_j$ 属于 E，则存在从事务 T_i 到 T_j 的一条有向边，表示事务 T_i 在等待 T_j 释放所需数据项。

当事务 T_i 申请的数据项当前被 T_j 持有时，边 $T_i \rightarrow T_j$ 被插入等待图中。只有当事务 T_j 不再持有事务 T_i 所需数据项时这条边才从等待图中删除。

系统中存在死锁当且仅当等待图中包含环。环中的事务称为处于死锁状态。要检测死锁，系统需要维护等待图，并周期性地激活一个在等待图中搜索环的算法。

为说明这些概念，考虑图 14-13 所示的等待图，它说明了下面这些情形：

- 事务 T_{25} 在等待事务 T_{26} 与 T_{27}。
- 事务 T_{27} 在等待事务 T_{26}。
- 事务 T_{26} 在等待事务 T_{28}。

由于该等待图无环，系统不处于死锁状态。

假设事务 T_{28} 申请被事务 T_{27} 持有的数据项，则边 $T_{28} \rightarrow T_{27}$ 加入等待图中，从而得到图 14-14 所示的系统新状态。此时，该等待图中包含了环：

$$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$$

意味着事务 T_{26}、T_{27} 与 T_{28} 都处于死锁状态。

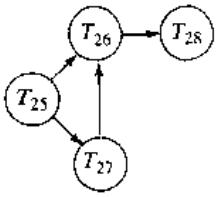


图 14-13 无环等待图

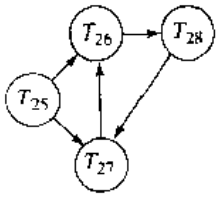


图 14-14 有环等待图

由此引出下面的问题：何时激活检测算法？答案取决于两个因素：

1）死锁发生频率怎样？

2）有多少事务将受到死锁的影响？

如果死锁频繁发生，则检测算法应比通常情况下激活得更频繁。分配给处于死锁状态的事务的数据项在死锁解除之前不能为其他事务获取。此外，等待图中环的数目也可能增长。最坏的情况下，每个分配请求都不能立即满足。

2. 死锁恢复

当检测算法判定存在死锁时，系统必须从死锁中恢复。解除死锁最通常的做法是回滚一至多个事务。需采取的行动有三个：

1）选择牺牲者。给定处于死锁状态的事务集，为解除死锁，必须决定回滚哪一个或哪一些事务以打破死锁。我们应使事务回滚带来的代价最小。可惜“最小代价”这个词并不准确。很多因素影响事务回滚代价，其中包括：

a. 事务已计算了多久，在完成其指定任务之前该事务还将计算多长时间。

b. 该事务已使用了多少数据项。

c. 为完成事务还需使用多少数据项。

d. 回滚时将牵涉多少事务。

2）回滚。一旦我们决定了要回滚哪个事务，我们必须决定该事务回滚多远。最简单的方法是彻底回滚；即中止该事务，尔后重新开始。然而，事务者只回滚到可以解除死锁处会更有效。但是，这种方法要求系统维护所有正在运行事务的额外的状态信息。有关参考文献见文献注解。

3）饿死。在系统中，如果选择牺牲者主要基于代价因素，有可能同一事务总是被选为牺牲者，于是该事务总是不能完成其指定任务。这种情况称为饿死。必须保证一个事务被选为牺牲者的次数有限（且较少）。最常用的方法是在代价因素中包含回滚次数。

14.7 插入与删除操作

目前为止我们一直把注意力集中在 read 与 write 操作上，这就限制了事务只能处理已存在于数据库中的数据。一些事务不仅要求访问已存在的数据项，而且要求创建新的数据项。还有一些事务要求能删除数据项。为考察这样的事务如何影响并发控制，我们引入如下操作：

- delete（Q）。从数据库中删除数据项 Q。
- insert（Q）。插入一个新的数据项 Q 到数据库中并赋予 Q 一个初值。

事务 T_i 在 Q 被删除后执行 read(Q)操作将导致 T_i 中的逻辑错误。同样，事务 T_i 在 Q 被插入之前执行 read(Q)操作也将导致 T_i 中的逻辑错误。试图删除并不存在的数据项也是一个逻辑错误。

14.7.1 删除

要理解删除指令怎样影响并发控制，必须弄清删除指令何时与另一指令发生冲突。令 I_i

与 I_j 分别是 T_i 与 T_j 的指令，它们连续地出现于调度 S 中。令 I_i = delete（Q），我们考虑如下几个 I_j 指令：

• I_j = read（Q）。I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_j 可以成功执行 read 操作。

• I_j = write（Q）。I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_j 可以成功执行 write 操作。

• I_j = delete（Q）。I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_i 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_i 将出现逻辑错误。

• I_j = insert（Q）。I_i 与 I_j 冲突。假设数据项 Q 在执行 I_i 与 I_j 之前不存在。那么，若 I_i 出现在 I_j 之前，事务 T_i 将出现逻辑错误。如果 I_j 出现在 I_i 之前，则没有逻辑错误发生。同样，如果 Q 在执行 I_i 与 I_j 之前已存在，则如果 I_j 出现在 I_i 之前会出现逻辑错误，反过来 I_i 出现在 I_j 之前则不会。

可以得出结论：如果使用两阶段封锁，在一数据项删除之前，在该数据项上必须请求加排它锁。在时间戳排序协议下，必须执行类似于为 write 操作进行的测试。假设事务 T_i 发出 delete（Q）：

• 如果 TS（T_i）＜R-timestamp（Q），则 T_i 将要删除的 Q 值已被满足 TS（T_j）＞TS（T_i）的事务 T_j 读取。因此，delete 操作被拒绝，T_i 回滚。

• 如果 TS（T_i）＜W-timestamp（Q），则满足 TS（T_j）＞TS（T_i）的事务 T_j 已经写过 Q。因此，delete 操作被拒绝，T_i 回滚。

• 否则，执行 delete 操作。

14.7.2　插入

我们已经看到 insert（Q）操作与 delete 操作冲突。同样，insert（Q）操作与 read（Q）操作或 write（Q）操作冲突。在数据项存在之前不能进行 read 或 write 操作。

由于 insert（Q）给数据项 Q 赋值，在并发控制中应当像处理 **write** 操作那样处理 **insert** 操作：

• 在两阶段封锁协议下，如果 T_i 执行 insert（Q）操作，T_i 在新创建的数据项 Q 上赋予排它锁。

• 在时间戳排序协议下，如果 T_i 执行 insert（Q）操作，R-timestamp（Q）与 W-timestamp（Q）的值被设成 TS（T_i）。

14.7.3　幻象现象

考虑事务 T_{29}，它在银行数据库上执行以下 SQL 查询：

select sum(*balance*)
from *account*
where *branch-name* = "Perryridge"

事务 T_{29}需要访问关系 *account* 有关 Perryridge 分支机构的全部元组。

考虑事务 T_{30}，它执行以下 SQL 插入：

insert into *account*
values("Perryridge",A-201,900)

令 S 是包含事务 T_{29}与 T_{30}的调度。由于以下原因，我们预计冲突是可能存在的：

• 如果 T_{29}在计算 sum (*balance*) 时使用了 T_{30}新近插入的元组，则 T_{29}读取被 T_{30}写入的值。因此，在等价于 S 的串行调度中，T_{30}必须先于 T_{29}。

• 如果 T_{29}在计算 sum (*balance*) 时未使用 T_{30}新近插入的元组，则在等价于 S 的串行调度中，T_{29}必须先于 T_{30}。

这两种情况中的第二种让人感到奇怪。T_{29}与 T_{30}没有访问共同的元组，但它们却相互冲突！事实上，T_{29}与 T_{30}在一个幻象元组上发生冲突，我们刚才描述的现象称为幻象现象。如果并发控制在元组级粒度上进行，该冲突将难以发现。

为防止幻象现象，我们允许 T_{29}阻止其他事务在关系 *account* 上创建 *branch-name* = "Perryridge" 的新元组。

为找到关系 *account* 中满足条件 *branch-name* = "Perryridge" 的所有元组，T_{29}必须搜索整个 *account* 关系，或至少搜索关系上的一个索引。目前为止，我们隐含地假设事务所访问的数据项仅仅是元组。然而，事务 T_{29}是一个读取关系中有哪些元组这一信息的事务的例子，而事务 T_{30}则是更新关系中有哪些元组这一信息的事务的例子。显然，仅仅封锁所访问的元组是不够的，封锁关系中有哪些元组这一信息也是需要的。

解决这个问题的最简单方法是将一个数据项与关系本身联系起来。读取关系中有哪些元组这一信息的事务，如事务 T_{29}，必须给对应关系 *account* 的数据项加共享锁。更新关系中有哪些元组这一信息的事务，如事务 T_{30}，必须给对应关系 *account* 的数据项加排它锁。这样，事务 T_{29}与 T_{30}将在真实的数据项上发生冲突，而不是在幻象上发生冲突。

不要把多粒度封锁中整个数据库的封锁与对应关系的数据项的封锁相混淆。通过封锁数据项，事务只是阻止其他事务对关系中有哪些元组这一信息的修改。对元组的封锁仍是需要的。即使另一个事务在对应关系本身的数据项持有排它锁，直接访问某个元组的事务也可以被授予该元组上的排它锁。

封锁对应关系本身的数据项的主要缺点是并发程度低——在关系中插入两个不同元组的两个事务也不能并发执行。

较好的解决方法是使用索引封锁技术。在关系中插入元组的任何事务必须在该关系的每一个索引中插入有关信息。通过使用索引的封锁协议，我们可以消除幻象现象。为简单起见，我们只考虑 B^+ 树索引。

正如在 11 章所看见的那样，每一个搜索码值对应于索引的一个叶结点。查询通常使用一至多个索引来访问关系。插入操作必须在关系的所有索引中插入新元组。本例中，我们假定在关系 *account* 的 *branch-name* 上建立了索引。那么，事务 T_{30}必须修改包含码值 "Perryridge" 的叶结点。如果 T_{29}读取同一叶结点，查找有关 Perryridge 分支机构的所有元组，则 T_{29}与 T_{30}在该叶结点上发生冲突。

通过将幻象现象实例转换为对索引叶结点上封锁的冲突，索引封锁协议就利用了关系上索引的可获性。该协议运作如下：

• 每个关系至少有一个索引。

• 只有首先在关系的一个或多个索引上找到元组后，这些元组才能被访问。

• 进行查找（不管是区间查找还是点查找）的事务 T_i 必须要在它访问的所有索引叶结点

上获得共享锁。

• 没有更新关系 r 的所有索引之前，事务 T_i 不能插入、删除或更新关系 r 中的元组 t_i。该事务必须获得受插入、删除或更新影响的所有索引叶结点上的排它锁。对于插入与删除，受影响的叶结点是那些（插入后）或（删除前）包含元组搜索码值的叶结点。对于更新，受影响的叶结点是那些（修改前）包含搜索码旧值的叶结点以及（修改后）包含搜索码新值的叶结点。

• 必须遵循两阶段封锁协议规则。

索引封锁技术存在一些变种，可以消除本章中其他并发控制协议下的幻象现象。

14.8 索引结构中的并发

索引结构的访问处理可以像其他数据库结构的访问处理那样进行，并应用前面讲述过的并发控制技术。但是，由于索引访问频繁，它们将成为封锁竞争的集中点，从而导致低并发度。幸运的是，索引不必像其他数据库结构那样处理，因为它们提供将搜索码映射到数据库元组的高层抽象。事务在两次索引查找期间，发现索引结构发生了变化是完全可以接受的，只要索引查找返回正确的元组集。因此，只要维护索引的准确性，对索引进行非可串行化并发存取是可以接受的。

我们讲述并发存取 B^+ 树的处理技术。B^+ 树的其他处理技术以及其他索引结构的处理技术的相关文献见文献注解。

我们所讲述的处理 B^+ 树的技术基于封锁机制，但既不采用两阶段封锁也不采用树形协议。用于查找、插入与删除的算法是第 11 章中使用的算法，这里只做了小小的修改。我们要求每个结点（不仅仅是叶结点）维护一个指向右兄弟结点的指针，满足这个条件的树称为 B-link 树。这个指针是必要的，因为在一个结点分裂时进行的查找可能不仅要查找该结点而且还可能要查找该结点的右兄弟结点（如果存在的话）。在列出修改后的过程以后，我们将用一个例子来说明这个技术。

• 查找。B^+ 树的每个结点在访问之前必须加共享锁。该锁在对 B^+ 树的其他任何结点发出加锁请求前被释放。如果结点分裂与查找同时发生，所希望的搜索码值可能不再位于查找过程中所访问的某个结点所代表的那些值的范围内。在这种情况下，搜索码值由一个右兄弟结点代替，这是系统循着指向右兄弟结点的指针而找到的。不过，叶结点的封锁遵循两阶段封锁协议以避免幻象现象，如 14.7.3 节所述。

• 插入与删除。系统遵循查找规则，定位要进行插入或删除的叶结点。该结点的共享锁升级为排它锁，然后进行插入或删除。受插入或删除影响的叶结点封锁遵循两阶段封锁协议以避免幻象现象，如 14.7.3 节所述。

• 分裂。如果一个结点分裂，则按 11.3 节的算法创建新结点。设置原始结点与新产生结点的右兄弟指针。接着，释放原始结点的排它锁，并发出对父结点加排它锁的请求，以便插入指向新结点的指针。

• 合并。执行删除后，如果一个结点的搜索码值太少，则必须给将要与之合并的那个结点加上排它锁。一旦这两个结点合并，则发出对父结点加排它锁的请求，以便删除要被删除的结点。此时，释放合并结点的锁。如果父结点不需再合并，则释放父结点的锁。

插入与删除操作可能封锁一个结点，释放该结点，然后又对它重新封锁，注意到这一点是很重要的。此外，与分裂或合并操作并发执行的查找可能发现所需搜索码值被分裂或合并操作移到右兄弟结点。

为说明这一点，考虑图 14-15 所示的 B^+ 树。假设有两个针对该 B^+ 树的并发操作：

1）插入 “Clearview”。

2）查找 “Downtown”。

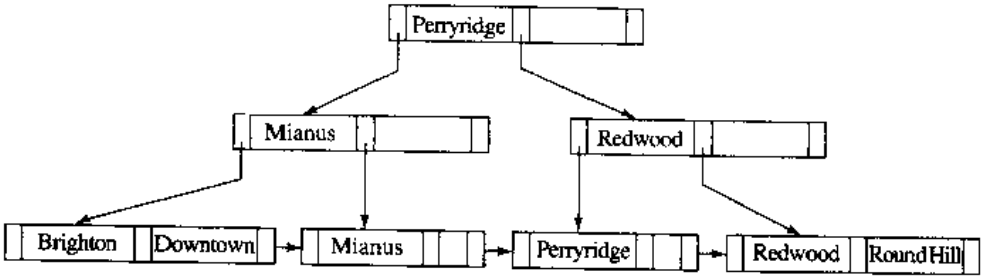


图 14-15　*account* 文件 $n=3$ 时的 B^+ 树

假设首先开始插入操作。该操作对 “Clearview” 执行查找，发现要插入 “Clearview” 的结点已满，于是插入操作将该结点的共享锁转换为排它锁，并创建新结点。这样，原始结点包含搜索码值 “Brighton” 与 “Clearview”，新结点包含搜索码值 “Downtown”。

现在假设发生上下文切换，控制传递给查找操作。该查找操作访问根结点，然后沿指向左子结点的指针而下。接着访问那个结点，并得到指向左子结点的指针。该左子结点原先包含搜索码值 “Brighton” 与 “Downtown”。由于该结点目前被插入操作以排它方式封锁，查找操作必须等待。注意，此时查找操作不持有任何锁！

插入操作现在释放了叶结点并重新对父结点加锁，这次以排它方式封锁。该操作完成了插入，得到图 14-16 所示的 B^+ 树。查找操作继续进行。然而，查找操作拥有的指针指向错误的叶结点，于是它顺着右兄弟结点指针定位到下一个结点。如果该结点仍不正确，则继续顺着该结点的右兄弟指针查找。可以证明，如果查找操作拥有指向错误结点的指针，则沿着右兄弟结点指针，查找操作最终可以到达正确的结点。

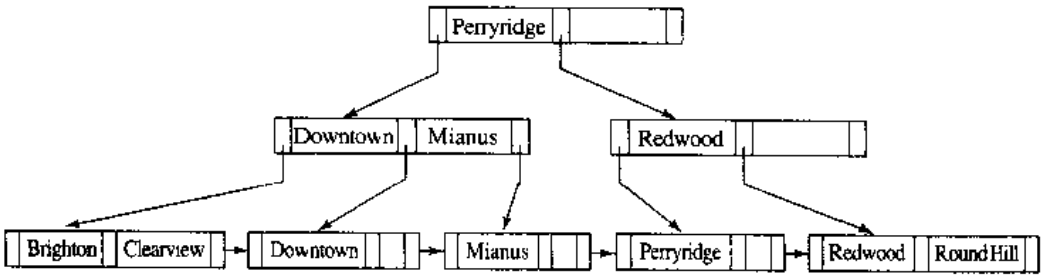


图 14-16　插入 “Clearview” 到图 14-15 所示的 B^+ 树中

查找与插入不会引起死锁。删除操作时的结点合并可能引起不一致性，因为查找操作可能在父结点被更新前已从父结点读取了指向被删除结点的指针，且曾试图访问被删除的结点。查找操作将不得不从根结点重新开始。不对结点合并可以避免这样的不一致性，但这个解决方案使某些结点包含的搜索码值太少，违背了 B^+ 树的某些特性。然而，大部分数据库中插入操作比删除操作频繁，因此包含搜索码值太少的结点可能会较快地得到更多的值。

14.9　总结

当多个事务在数据库中并发执行时，数据的一致性可能受到破坏。系统有必要控制各事务之间的相互作用，这是通过称为并发控制机制的多种机制中的一种来实现的。

为保证可串行性，我们可以使用多种并发控制机制。所有这些机制或拖延一个操作中止发出该操作的事务。最常用的机制是各种封锁协议、时间戳排序机制、有效性检查技术与多版本机制。

封锁协议是一组规则，这些规则阐明了事务何时对数据库中的数据项加锁解锁。两阶段封锁协议仅在一个事务未曾释放任何数据项时允许该事务封锁新数据项。该协议保证可串行性，但不能避免死锁。在没有有关数据项访问方式信息的情况下，两阶段封锁协议对于保证可串行性既是必要的又是充分的。

时间戳排序机制通过事先在每对事务之间选择一个顺序来保证可串行性。系统中的每个事务对应一个唯一的固定的时间戳。事务的时间戳决定了事务的可串行化顺序。这样，如果事务 T_i 的时间戳小于事务 T_j 时间戳，则该机制保证产生的调度等价于事务 T_i 出现在事务 T_j 之前的一个串行调度。该机制通过回滚违反该次序的事务来保证这一点。

在大部分事务是只读事务，这样事务间冲突频度较低的情形下，有效性检查机制是一个适当的并发控制机制。系统中的每个事务对应一个唯一的固定的时间戳，串行性次序是由事务的时间戳决定的。在该机制中，事务不会被延迟。不过，事务要完成必须通过有效性检查，如果事务未通过有效性检查，则该事务回滚到初始状态。

某些情况下把多个数据项聚为一组，将它们作为聚集数据项来处理效果可能更好，这就导致了多级粒度。我们允许存在各种大小的数据项，并定义数据项的层次，其中小数据项嵌套于大数据项之中。这种层次结构可以图形化地表示为树。封锁按从根结点到叶结点的顺序进行，解锁则按从叶结点到根结点的顺序进行。该协议保证可串行性，但不能避免死锁。

多版本并发控制机制基于每个事务写数据项时为该数据项创建一个新版本。读操作发出时，系统选择其中的一个版本进行读取。利用时间戳，并发控制机制保证按确保可串行性的方式选取要读取的版本。读操作总能成功。在多版本时间戳排序协议中，写操作可能引起事务的回滚。在多版本两阶段封锁协议中，写操作可能导致封锁等待或死锁。

许多封锁协议都不能防止死锁。防止死锁的一种方法是使用抢占与事务回滚。为控制抢占，我们给每个事务赋予唯一一个时间戳，这些时间戳用于决定事务是等待还是回滚。如果一个事务回滚，它在重启时保持原有时间戳。Wait-die 与 wound-wait 机制是两个抢占机制。另一种用于处理死锁的方法是死锁检测与恢复机制，等待图是为此而构造的。系统处于死锁状态当且仅当等待图中包含环。当一个检测算法判定死锁存在时，系统必须从死锁中恢复。系统通过回滚一至多个事务来解除死锁。

仅当要删除元组的事务在该元组上具有排它锁时，delete 操作才能够进行。在数据库中插入新元组的事务在该元组上授予排它锁。插入操作可能导致幻象现象，这时插入操作与查询发生冲突，尽管两个事务可能没有存取共同的元组。索引封锁技术通过要求对某些索引存储桶加锁来解决这个问题。所加的锁保证所有事务在实际的数据项上发生冲突，而不是在幻象上。

我们可为特殊的数据结构开发特殊的并发控制技术。通常，特殊的技术被用到 B^+ 树上，以允许较大的并发性。这些技术允许对 B^+ 树进行非可串行化访问，且它们保证 B^+ 树结构是正确的，并保证对数据库本身的存取是可串行化的。

习题

14.1 证明两阶段封锁协议保证冲突可串行化，并且事务可以根据其封锁点串行化。

14.2 考虑下面两个事务：

T_{31}: **read**(A);
read(B);
if $A=0$ **then** $B:=B+1$;
write(B);

T_{32}: **read**(B);
read(A);
if $B=0$ **then** $A:=A+1$;
write(A);

给事务 T_{31}与 T_{32}增加加锁、解锁指令，使它们遵从两阶段封锁协议。这两个事务会引起死锁吗？

14.3 严格两阶段封锁协议带来什么好处？会产生哪些弊端？

14.4 强两阶段封锁协议带来什么好处？它与其他形式的两阶段封锁协议相比怎样？

14.5 数据库系统的并发控制管理器必须决定是满足锁请求还是让发出请求的事务等待。设计一个节省空间的数据结构，使并发控制管理器能够快速作出决定。

14.6 大部分数据库系统实现采用严格两阶段封锁协议。说明该协议受欢迎的三点理由。

14.7 考虑以下对树形封锁协议的扩展，它既允许使用共享锁又允许使用排它锁：

- 事务可以是只读事务，此时它只申请共享锁；也可以是更新事务，此时它只申请排它锁。
- 每个事务必须遵从树形协议规则。只读事务可以首先封锁任何数据项，而更新事务必须首先封锁根结点。

证明该协议保证可串行性并能避免死锁。

14.8 考虑一个按带根树方式组织的数据库。假设我们在每对结点之间插入一个空结点。证明如果我们在由此构成的新树上遵从树形协议，我们得到的并发性可以比在原始树上遵从树形协议得到的更高。

14.9 考虑以下基于图的封锁协议，它只允许加排它锁，并且在带根有向无环数据图上运作：

- 事务首先可以封锁任何结点。
- 要封锁任何其他结点，事务必须在该结点的大部分父结点上持有锁。

证明该协议保证可串行性并能避免死锁。

14.10 通过举例证明：存在在树形封锁协议下可行，而在两阶段封锁协议下不可行的调度，反之亦然。

14.11 考虑以下基于图的封锁协议，它只允许加排它锁，并且在带根有向无环数据图上运作：

- 事务首先可以封锁任何结点。
- 要封锁任何其他结点，事务必须已经访问该结点的所有父结点，并且必须在该结点的一个父结点上持有锁。

证明该协议保证可串行性并能避免死锁。

14.12 考虑树形协议的一个变种，它称为森林协议。数据库按带根树的森林方式组织。每个事务必须遵从以下规则：

- 每棵树上的首次封锁可以在任何数据项上进行。
- 树上的第二次以及此后的封锁申请仅当被申请结点的父结点上有锁时才能发出。
- 数据项的解锁可在任何时候。

• 事务 T_i 释放数据项后不能再次封锁该数据项。

证明森林协议不保证可串行性。

14.13 考虑除 read 与 write 之外还包含原子操作 increment 的一个数据库系统。令 V 是数据项 X 的值。操作：

increment（X）by C

在一个原子步骤中将 X 的值设为 $V+C$。如果事务不执行 read（X），则事务不能获知 X 的值。图 14-17 表示了三种锁类型的锁相容阵：共享型、排它型、增加型。

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

图 14-17　锁相容阵

(a) 证明：如果所有事务按相应的类型封锁它们所访问的数据项，则两阶段封锁保证可串行性。

(b) 证明：包含增加型锁可以增加并发性。

(提示：考虑所举银行例子中的支票交换事务。)

14.14 在时间戳排序中，W-timestamp（Q）表示成功执行 write（Q）的所有事务的最大时间戳。现在，假设我们将之定义为最近成功执行 write（Q）的事务的时间戳，这种措辞上的变化会带来什么不同？解释你的答案。

14.15 当一个事务在时间戳排序协议下回滚，它被赋予新时间戳。为什么它不能简单地保持原有时间戳？

14.16 在多粒度封锁中，隐式封锁与显式封锁有什么不同？

14.17 虽然 SIX 锁在多粒度封锁中很有用，但排它共享意向锁（XIS）却无用。为什么？

14.18 采用多粒度封锁机制比采用单封锁粒度的等价系统需要更多或更少的锁。针对每种情况各举一例，并比较所允许的相对的并发量。

14.19 考虑 14.3 节基于有效性检查的并发控制机制。证明：若选择 Validation（T_i）而不是 Start（T_i）作为事务 T_i 的时间戳，则如果事务间发生冲突的次数确实很低，那么我们能够获得较好的响应时间。

14.20 证明：存在满足两阶段封锁协议却不满足时间戳协议的调度，反之亦然。

14.21 对于下面的每个协议，说明促使你使用某个协议的实际应用原因以及不使用的原因：

• 两阶段封锁。
• 具有多粒度封锁的两阶段封锁。
• 树形协议。
• 时间戳排序。
• 有效性检查。
• 多版本时间戳排序。
• 多版本两阶段封锁。

14.22 对于每个基于时间戳的协议，判定它是否无级联以及是否可恢复。

14.23 在时间戳协议的一个修改版中，我们要求测试提交位以判定 read 请求是否必须等待。解释提交位如何防止级联中止。为什么该测试对 write 请求是不必要的。

14.24 在什么条件下避免死锁比允许死锁发生然后检测的方式代价更小？

14.25 避免死锁后，饿死仍有可能吗？解释你的答案。

14.26 解释幻象现象。为什么尽管采用两阶段封锁协议，该现象仍可能导致不正确的并发执行？

14.27 设计一个基于时间戳的能避免幻象现象的协议。

14.28 假设我们采用 14.1.4 节的树形协议来管理对 B^+ 树的并发访问。由于影响根结点的插

入可能导致分裂，插入操作似乎在完成整个操作之前不能释放任何锁。那么在什么情况下可以较早地释放锁？

14.29 在持久编程语言中封锁不是显式进行的。对象（或相应页）被访问时必须加锁。大部分现代操作系统允许用户对页面设置访问保护（不许访问、读、写），并且违反存取保护的内存访问将导致违反保护错误（如参见 Unix 的 mprotect 命令）。说明访问保护机制在持久编程语言中如何用于页级封锁。

（提示：该技术在某种程度上类似于 10.9.3 节中用于硬件混写的技术。）

文献注解

两阶段封锁协议由 Eswaran 等［1976］引入。树型协议来自 Silberschatz 与 Kedem［1980］。其他在更一般图上执行的非两阶段封锁协议由 Yannakakis 等［1979］、Kedem 与 Silberschatz［1983］、以及 Buckley 与 Silberschatz［1985］提出。Lien 和 Weinberger［1978］、Yannakakis 等［1979］、Yannakakis［1981］、以及 Papadimitriou［1982］给出了对封锁协议的更一般的讨论。Korth［1983］探讨了从基本的共享和排它锁方式中可以得到的多种封锁方式。动态搜索树并发访问的各种算法由 Bayer 与 Schkolnick［1977］、Ellis［1980a，1980b］、Lehman 与 Yao［1981］、以及 Manber 与 Ladner［1984］提出。习题 14.7 取自 Kedem 与 Silberschatz［1983］，习题 14.8 取自 Buckley 与 Silberschatz［1984］，习题 14.9 取自 Kedem 与 Silberschatz［1979］，习题 14.11 来自 Yannakakis 等［1979］，习题 14.13 取自 Korth［1983］。

基于时间戳的并发控制机制取自 Reed［1983］。Bernstein 与 Goodman［1980a］对各种基于时间戳的机制进行了说明。Buckley 与 Silberschatz［1983］给出了一种不需回滚且保证可串行性的算法。有效性检查机制来自 Kung 与 Robinson［1981］。单站点的和分布式的有效性检查并发控制机制由 Bassiouni［1988］给出。

多粒度数据项封锁协议来自 Gray 等［1975］，Gray 等［1976］给出了详细的描述。封锁粒度的作用在 Ries 与 Stonebraker［1977］中讨论。Korth［1983］对任意封锁方式（允许更多的语义而不仅仅是读和写）的多粒度封锁做了规范。这种方法包括称为更新型锁的一类锁，以处理锁转换。Carey［1983］将多粒度的想法扩展到基于时间戳的并发控制。保证不产生死锁的一种扩展协议由 Korth［1982］给出。面向对象数据库系统中的多粒度封锁在 Lee 与 Liou［1996］中进行了讨论。

关于多版本并发控制的讨论见 Bernstein 等［1983］。Silberschatz［1982］中给出了一种多版本树封锁协议。多版本时间戳排序在 Reed［1978，1983］介绍。Lai 与 Wilkinson［1984］对多版本两阶段封锁的证明进行了描述。

Dijkstra［1965］是死锁领域中最早也最有影响的研究人员之一。Holt［1971，1972］第一个用类似本章所给的图的模型给出了死锁的形式化概念。时间戳死锁检测算法来自 Rosenkrantz 等［1978］。Gray 等［1981b］对等待和死锁的可能性进行了分析。关于死锁和可串行化的理论成果见 Fussell 等［1981］和 Yannakakis［1981］。

环检测算法可以在一般的算法教科书中找到，如 Cormen 等［1990］和 Aho 等［1983］。

Bayer 与 Schkolnick［1977］和 Johnson 与 Shasha［1993］对 B^+ 树中的并发做了研究。14.8 节中所给技术基于 Kung 与 Lehman［1980］，以及 Lehman 与 Yao［1981］。相关工作包括 Kwong 与 Wood［1982］，Manber 与 Ladner［1984］，以及 Ford 与 Calhoun［1984］。Shasha 与 Goodman［1988］对索引结构的并发协议的特征做了很好的描述。Ellis［1987］给出了一个用于线性散列的并发控制技术。B-link 树在 Lomet 与 Salzberg［1992］中进行了讨论。Ellis

[1980a，1980b] 给出了其他索引结构的并发控制算法。

Gray [1978] 给出了关于并发控制与恢复的一个综述报告。这方面讨论的教科书包括 Data [1983a]、Bernstein 等 [1987]、Papadimitriou [1986]，以及 Gray 与 Reuter [1993]。Garza 与 Kim [1988] 对面向对象数据库中的事务处理进行了讨论。设计和软件工程应用中的事务处理在 Korth 等 [1988]、Kaiser [1990]，以及 Weikum [1991] 中进行了讨论。多级事务理论方面的内容由 Lynch 等 [1988] 以及 Weihl 与 Liskov [1990] 给出。

IBM Almaden 研究中心开发的 ARIES 事务管理器对事务管理产生了重要影响。这一管理器在一系列论文中进行了讨论，包括 Mohan [1990a，1990b]、Mohan 等 [1991，1992]，以及 Mohan 与 Narang [1994，1995]。

第15章 恢复系统

计算机系统与其他任何设备一样易发生故障。故障的原因多种多样，包括磁盘故障、电源故障、软件错误、机房失火、甚至人为破坏。这些情况一旦发生，就可能导致丢失信息。因此，数据库系统必须预先采取措施，以保证即使发生故障，也可以保持第13章中所讲的事务的原子性和持久性。恢复机制是数据库系统必不可少的组成部分，它负责将数据库恢复到故障发生前的某个一致状态。

15.1 故障分类

系统可能发生的故障有很多种，每种故障需要不同的方法来处理。最容易处理的故障类型是不会导致系统信息丢失的故障；较难处理的故障是导致信息丢失的故障。本章将只考虑如下类型的故障：

- 事务故障。有两种错误可能造成事务执行失败：
  - 逻辑错误。事务由于某些内部条件而无法继续正常执行，这样的内部条件如非法输入、找不到数据、溢出、或超出资源限制。
  - 系统错误。系统进入一种不良状态（如死锁），结果事务无法继续正常执行。但该事务可以在以后的某个时间重新执行。
- 系统崩溃。硬件故障，或数据库软件或操作系统的漏洞，导致易失性存储器内容的丢失，并使得事务处理终止。而非易失性存储器仍完好无损。

硬件错误和软件漏洞致使系统终止，而不破坏非易失性存储器内容的假设称为故障-停止假设。设计良好的系统在硬件或软件一级有大量的内部检查，一旦有错误发生系统就会终止。因此，故障-停止假设是合理的。

- 磁盘故障。在数据传送操作过程中由于磁头损坏或故障造成磁盘块上的内容丢失。其他磁盘上的数据拷贝，或三级介质（如磁带）上的归档备份可用于从这种故障中恢复。

要确定系统如何从故障中恢复，首先需要确定用于存储数据的设备的故障状态。其次，必须考虑这些故障状态对数据库内容有什么影响。然后我们可以设计在故障发生后仍保证数据库一致性以及事务原子性的算法。这些算法称为恢复算法，它由两部分组成：

1）在正常事务处理时采取的措施，保证有足够的信息可用于故障恢复。

2）故障发生后采取的措施，将数据库内容恢复到某个保证数据库一致性，事务原子性及持久性的状态。

15.2 存储器结构

正如第10章中所看到的，数据库中的各种数据项可在多种不同存储介质上存储访问。为便于理解如何保证事务的原子性和持久性，我们必须对存储介质及其存取方法有更深入的了解。

15.2.1 存储器类型

存储介质的类型有很多，它们可以从相对速度、容量及对故障的敏感程度上加以区分。

• 易失性存储器。易失性存储器中的信息在系统崩溃时通常无法保存下来。这类存储器的例子有主存和高速缓存。对易失性存储器的访问是极快的，因为它本身的存取速度很快并且可以直接访问其中的任意数据项。

• 非易失性存储器。非易失性存储器中的信息在系统崩溃时可以保存下来。这类存储器的例子有磁盘和磁带。磁盘用于联机存储，磁带用于归档存储，但它们都容易出故障（如，磁头损坏）而导致信息丢失。就目前的技术状况，非易失性存储器比易失性存储器慢几个数量级，这是由于磁盘和磁带设备是机电的，而不像易失性存储器那样完全基于芯片。在数据库系统中，大多数非易失性存储器采用磁盘。其他非易失性介质通常只用于备份数据。

• 稳定存储器。稳定存储器中的信息永不丢失（“永不”也不是绝对的，因为理论上无法保证什么事永不发生　例如，有可能黑洞吞噬了地球，永远地摧毁了所有数据。尽管这种可能性微乎其微!）。虽然稳定存储器理论上不可能得到，但可以通过技术手段使数据极不可能丢失。稳定存储器的实现将在下节讨论。

各种存储器类型的区别在实际中往往不像前面介绍的那么清楚。某些系统提供电池备份，这样某些主存信息在系统崩溃和电源故障后仍可以保存。其他类型的非易失性存储器，如光学介质，甚至能提供比磁盘更高的可靠性。

15.2.2　稳定存储器的实现

要实现稳定存储器，需要在多个非易失性存储介质（通常是磁盘）上以独立的故障模式复制所需信息，并且以某种受控的方式更新信息，保证数据传送过程中发生的故障不会破坏所需信息。

前面（第 10 章）提到的 RAID 系统保证了单个磁盘故障（即使发生在数据传送过程中）不会导致数据丢失。最简单并且最快的 RAID 形式是磁盘镜像，即在不同的磁盘上为每个磁盘块保存两个拷贝。RAID 的其他形式花费虽低一些，但性能也差一些。

但是，RAID 系统不能防止由于灾难如大火或洪水而导致的数据丢失。许多系统通过将归档备份存储在磁带上并转移到其他地方来防止这种灾难。但是，由于磁带不能连续地来回移动，因而磁带被移至其他地方以后最近所做的更新可能会在这样的灾难中丢失。更安全的系统在远程为稳定存储器的每一个块保存一份拷贝，除在本地磁盘系统进行存储外，还通过网络存储到远程。由于往本地存储器输出块的同时也要输出到远程系统，因而一旦输出操作完成，输出结果将不会丢失，即使发生大火或洪水这样的灾难。

本节剩余部分讨论如何在数据传送发生时保护存储介质不受故障损害。内存和磁盘存储器间进行块传送有以下几种可能结果：

• 成功完成。被传送的信息安全到达目的地。

• 部分失败。传送过程中发生故障，并且目标块中有不正确信息。

• 完全失败。故障发生在传送的较早阶段，目标块中未被写入任何信息。

我们要求如果数据传送发生故障，系统应能检测到并且调用恢复过程将块恢复成为一致状态。为达到这个要求，系统必须为每个逻辑数据库块维护两个物理块。若是镜像磁盘，则两个块在同一个地点；若是远程备份，则一个块在本地，另一块在远程。输出操作的执行如下：

1）将信息写入第一个物理块。

2）当第一次写成功完成时，将相同信息写入第二个物理块。

3）只有第二次写成功完成时，输出才算完成。

恢复时要检查一对物理块。如果它们相同并且没有检测到错误，则不需要采取进一步动

作。(前面提到，磁盘块中的某些错误，如部分写块，可通过在每个块上存储校验和来检测)。如果检测到一个块上有错误，则可用另一块的内容替换此块的内容。如果两个块都没有校验错但内容不一致，则可用第二块的内容替换第一块的内容。该恢复过程保证，对稳定存储器的写要么完全成功（即更新所有拷贝），要么没有任何改变。

在恢复过程中比较每一对块的开销太大。通过使用少量非易失性 RAM，跟踪正在进行的对块的写操作，我们可以大大降低开销。恢复时，只需比较正在写的块。

将块写到远程的协议类似于第 10 章，特别是习题 10.4 中所讨论的向镜像磁盘系统写块的协议。

我们可以将这个过程很容易地推广为允许为稳定存储器的每一个块使用任意多的拷贝。尽管使用大量拷贝比使用两个拷贝发生故障的可能性要低，但通常只用两个拷贝模拟稳定存储器是合理的。

15.2.3 数据访问

正如第 10 章中所看到的，数据库系统常驻于非易失性存储器（通常为磁盘），并且分成称为块的定长存储单位。块是磁盘数据传送的单位，可能包含多个数据项。我们假设没有数据项跨两个或多个块。这个假设对于大多数数据处理应用，如银行例子，都是正确的。

事务由磁盘向主存输入信息，然后再将信息输出回磁盘。输入和输出操作以块为单位完成。位于磁盘上的块称为*物理块*，临时位于主存的块称为*缓冲块*。内存中用于临时存放块的区域称为*磁盘缓冲区*。

磁盘和主存间的块移动是由下面两个操作引发的：

1) input (B) 传送物理块 B 至主存。

2) output (B) 传送缓冲块 B 至磁盘，并替换磁盘上相应的物理块。

这一机制如图 15-1 所示。

每个事务 T_i 有一个私有工作区，用于保存 T_i 所访问及更新的所有数据项的拷贝。该工作区在事务初始化时创建，在事务提交或中止时删除。事务 T_i 的工作区中保存的每一个数据项 X 记为 x_i。事务 T_i 通过在其工作区和系统缓冲区之间传送数据，与数据库系统进行交互。我们使用下面两个操作传送数据：

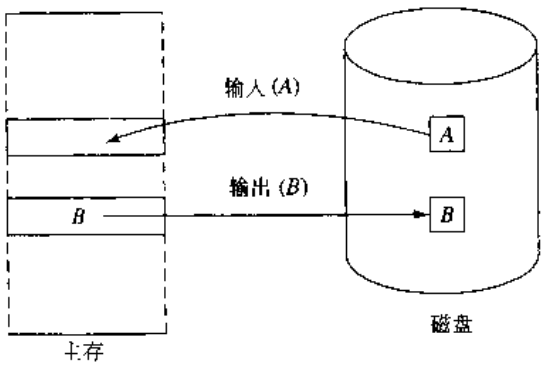


图 15-1 块存储操作

1) read (X) 将数据项 X 的值赋予局部变量 x_i。该操作执行如下：

a. 若 X 所在块 B_X 不在主存，则发指令执行 input (B_X)。

b. 将缓冲块 B_X 中 X 的值赋予 x_i。

2) write (X) 将局部变量 x_i 的值赋予缓冲块中的数据项 X。该操作执行如下：

a. 若 X 所在块 B_X 不在主存，则发指令执行 input (B_X)。

b. 将 x_i 的值赋予缓冲块 B_X 中的 X。

注意这两个操作都可能需要将块从磁盘传送到主存。但是，它们都没有特别指明需要将块从主存传送到磁盘。

缓冲块被最终写到磁盘，要么是因为缓冲区管理器因其他用途需要内存空间，要么是数据

库系统希望将 B 的变化反映到磁盘上。如果数据库系统发指令执行 output（B），则我们称数据库系统强制输出缓冲块 B。

当事务第一次需要访问数据项 X，它必须执行 read（X），然后把对 X 的所有更新都作用于 x_i。事务最后一次访问 X 后，它必须执行 write（X），以在数据库中反映 X 的变化。

X 所在缓冲块 B_X 上的操作 output（B_X）不需要在 write（X）执行后立即执行，因为块 B_X 可能包含其他仍在被访问的数据项。因此一段时间后才可能真正执行输出。注意，如果在操作 write（X）执行后但在操作 output（B_X）执行前系统崩溃，那么 X 的新值未写入磁盘，也就意味着丢失了。

15.3　恢复与原子性

再来考虑简化的银行系统和事务 T_i，事务 T_i 将 $50 从帐户 A 转到帐户 B，A 和 B 的初始值分别为 $1000 和 $2000。假设 T_i 执行过程中系统崩溃，且发生在 output（B_A）之后，output（B_B）之前，其中 B_A 和 B_B 代表 A 和 B 所在的缓冲块。由于内存的内容丢失，我们无法知道事务的结局。因此，可以调用以下其中一个可能的恢复过程：

- 重新执行 T_i。该过程使 A 的值变为 $900，而不是 $950。于是，系统进入不一致状态。
- 不重新执行 T_i。当前系统状态则是 A 和 B 的值分别为 $950 和 $2000。于是，系统进入不一致状态。

不论哪种情况，数据库均进入了不一致状态，因此这样简单的恢复机制是不行的。问题的原因在于我们在没有保证事务真正提交的情况下修改了数据库。我们的目标是要么执行 T_i 对数据库的所有修改，要么什么都不执行。但是，若 T_i 执行多处数据库修改，就可能需要多个输出操作，但故障也恰可能发生于某些修改完成后而全部修改完成前。

为达到保持原子性的目标，必须在修改数据库本身之前，首先输出对稳定存储器做修改的描述信息。我们将看到，这一过程使我们可以输出一个已提交事务的所有修改，而不管是否有故障发生。执行这种输出有两种方式，我们将在下两节讨论。在这两节中，我们将假设事务是串行执行的，即在某一时刻只有一个事务是活跃的。我们将在 15.6 节阐述如何处理并发执行的事务。

15.4　基于日志的恢复

使用最为广泛的记录数据库修改的结构是日志。日志是日志记录的序列，它记录了数据库中的所有更新活动。日志记录有几种，更新日志记录描述一次数据库写操作，它具有如下几个字段：

- 事务标识是执行 write 操作的事务的唯一标识。
- 数据项标识是所写数据项的唯一标识，通常是数据项在磁盘上的位置。
- 旧值是数据项的写前值。
- 新值是数据项的写后值。

其他特殊的日志记录用于记录事务处理过程中的重要事件，如事务的开始以及事务的提交或中止。我们将各种类型的日志记录记为：

- $<T_i$ start$>$。事务 T_i 开始。
- $<T_i, X_j, V_1, V_2>$。事务 T_i 对数据项 X_j 执行写操作。X_j 的写前值是 V_1，写后值是 V_2。

- $<T_i$ commit$>$。事务 T_i 提交。
- $<T_i$ abort$>$。事务 T_i 中止。

每次事务执行写操作之前，必须在数据库修改前生成该次写操作的日志记录。一旦日志记录已创建，就可以根据需要对数据库做修改。并且，我们能够利用日志记录中的旧值字段取消已对数据库做的修改。

为了在从系统和磁盘故障中恢复时能使用日志记录，日志必须放在稳定存储器上。现在假设每一个日志记录创建后立即写入稳定存储器上日志的尾部。15.7节我们将看到可在什么时候放宽这个要求，以减少日志带来的开销。以下两小节将介绍两个技术，它们可以利用日志保证即使发生故障也保持事务的原子性。由于日志包含了所有数据库活动的完整记录，所以日志中存储的数据量会变得很大。15.4.3小节我们将看到什么时候可以删掉日志信息。

15.4.1 延迟的数据库修改

延迟修改技术通过在日志中记录所有数据库修改，将一个事务的所有write操作拖延到事务部分提交时才执行，从而保证事务的原子性。前面提到，事务的最后一个动作一旦被执行，该事务就称为部分提交。本节所描述的延迟修改技术假设事务是串行执行的。

当事务部分提交时，日志上有关该事务的信息被用来执行延迟写。如果在事务完成其执行前系统崩溃，或事务中止，则只要忽略日志上的信息。

事务 T_i 的执行过程如下。T_i 开始执行前，向日志中写入记录$<T_i$ start$>$。T_i 的一次write（X）操作导致向日志中写入一条新记录。最后，当 T_i 部分提交时，向日志中写入记录$<T_i$ commit$>$。

当事务 T_i 部分提交时，日志中相关的记录被用来执行延迟写。由于执行该更新时可能发生故障，必须保证在开始更新前所有的日志记录已写到了稳定存储器上。一旦日志记录都写到了稳定存储器上，就真正执行更新，并且事务进入提交状态。

注意延迟修改技术只需要数据项的新值。因此，可以简化上一节中通用的更新日志记录结构，将旧值字段省去。

让我们用简化的银行系统例子加以阐述。令 T_0 为一个事务，它将$50从帐户 A 转到帐户 B。该事务定义如下：

$$
\begin{aligned}
T_0: \quad & \textbf{read}(A)\\
& A := A - 50\\
& \textbf{write}(A)\\
& \textbf{read}(B)\\
& B := B + 50\\
& \textbf{write}(B)
\end{aligned}
$$

令 T_1 为一个事务，它从帐户 C 取出$100。该事务定义如下：

$$
\begin{aligned}
T_1: \quad & \textbf{read}(C)\\
& C := C \quad 100\\
& \textbf{write}(C)
\end{aligned}
$$

假设这些事务串行执行，先执行 T_0 后执行 T_1，并且执行前帐户 A、B 和 C 的值分别为

$1000、$2000和$700。日志中包含这两个事务相关信息的部分如图15-2所示。

T_0和T_1的执行可以产生多种数据库系统和日志实际输出时采纳的顺序。其中一种顺序如图15-3所示。注意到只有将记录$<T_0, A, 950>$放入日志后，才能改变数据库中A的值。

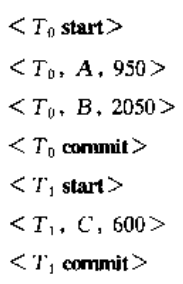


图15-2　数据库日志中与T_0和T_1有关的部分

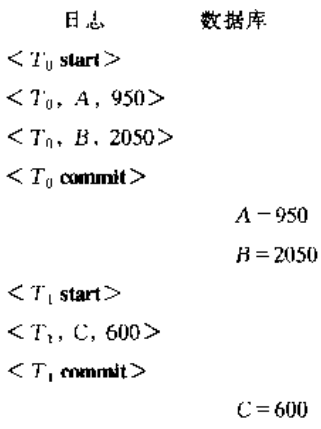


图15-3　与T_0和T_1有关的日志和数据库状态

利用日志，系统可以处理任何导致易失性存储器上信息丢失的故障。该恢复机制使用如下的恢复过程：

• redo(T_i)将事务T_i更新的所有数据项的值置为新值。

T_i所更新的数据项的集合及其相应的新值可以在日志中找到。

redo操作必须是幂等的，即执行它多次等价于执行它一次。为保证即使在恢复过程中发生故障也能正确，这个性质是必需的。

故障发生后，恢复子系统检查日志，看哪个事务需要重新执行。事务T_i需要重新执行当且仅当日志中既包含记录$<T_i$ start$>$又包含记录$<T_i$ commit$>$。因此，如果系统在事务执行完成后崩溃，日志中的信息将用来将系统恢复到事务完成后的一致状态。

例如，再看银行的例子，事务T_0和T_1以先T_0后T_1的次序串行执行。图15-2表示了T_0和T_1执行完成后的日志信息。假设系统在事务完成前崩溃，我们来看恢复技术如何将数据库恢复到一个一致的状态。假设对应于事务T_0中的

$$\text{write}(B)$$

的日志记录刚刚写入稳定存储器就发生了系统崩溃，崩溃时的日志如图15-4a所示。当系统重新启动后，不必做任何redo操作，因为日志中没有出现提交记录。帐户A和B的值仍保持$1000和$2000。未完成的事务T_0的日志记录可以从日志中删掉。

现在，假设对应于事务T_1中的

$$\text{write}(C)$$

的日志记录刚刚写入稳定存储器就发生了系统崩溃，崩溃时的日志如图15-4b所示。当系统重新启动后，要执行操作redo(T_0)，因为磁盘上的日志中有记录

$$<T_0 \text{ commit}>$$

该操作执行完，帐户 A 和 B 的值分别为 \$950 和 \$2050，帐户 C 的值仍为 \$700。和前面一样，未完成的事务 T_1 的日志记录可以从日志中删除。

最后，假设日志记录

$$<T_1 \text{ commit}>$$

刚刚写入稳定存储器就发生了系统崩溃，崩溃时的日志如图 15-4c 所示。当系统重新启动后，日志中有两个提交记录：一个对应于 T_0，一个对应于 T_1。因此，必须以提交记录在日志中出现的顺序执行操作 redo(T_0)和 redo(T_1)。这些操作执行以后，帐户 A、B 和 C 的值分别为 \$950、\$2050 和 \$600。

a)	b)	c)
$<T_0$ start$>$	$<T_0$ start$>$	$<T_0$ start$>$
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit$>$	$<T_0$ commit$>$
	$<T_1$ start$>$	$<T_1$ start$>$
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit$>$

图 15-4 与图 15-3 相同的日志，但分别在三个不同时刻

最后再考虑一种情况，第一次崩溃后的恢复过程中又发生了第二次系统崩溃。redo 操作可能导致对数据库的一些修改，但可能又没有完成所有修改。当系统在第二次崩溃后重新启动时，恢复过程和上一例子完全一样，对日志中的每一个提交记录

$$<T_i \text{ commit}>$$

执行操作 redo (T_i)。也就是说，恢复过程完全从头再执行一遍。由于 redo 写入数据库的值与数据库中的当前值无关，因此第二次成功执行 redo 的结果与第一次就成功的结果是一样的。

15.4.2 立即的数据库修改

立即更新技术允许数据库修改在事务仍处于活跃状态时就输出到数据库中。活跃事务所做的数据修改称为未提交修改。在发生系统崩溃或事务故障时，系统必须使用 15.4 节中所描述的日志记录中的旧值字段将修改的数据项恢复到事务开始以前的值。这种恢复是通过下面所描述的 undo 操作来完成的。

在事务 T_i 开始执行以前，日志中写入记录$<T_i$ start$>$。执行过程中，T_i 执行任何 write (X) 操作前先要向日志中写入适当的新的更新记录。当 T_i 部分提交时，日志中写入记录$<T_i$ commit$>$。

由于日志中的信息要用于恢复数据库的状态，因此不允许在相应日志记录写到稳定存储器前对数据库做实际的更新。这样，我们要求在执行操作 output (B) 以前，与 B 相关的日志记

录必须已经写到了稳定存储器上。我们在 15.7 节还要再讨论这个问题。

例如，再考虑简化的银行系统，事务 T_0 和 T_1 以先 T_0 后 T_1 的次序串行执行。日志中包含这两个事务有关信息的部分如图 15-5 所示。

T_0 和 T_1 的执行产生的数据库系统和日志实际输出时采纳的可能的一种顺序如图 15-6 所示。注意，如果用 15.4.1 小节中的延迟修改技术则不会产生这个顺序。

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

图 15-5　系统日志中与 T_0 和 T_1 有关的部分

日志	数据库
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

图 15-6　与 T_0 和 T_1 有关的系统日志和数据库状态

利用日志，系统可以解决任何不造成非易失性存储器上信息丢失的故障。恢复机制调用两个过程：

- undo（T_i）将事务 T_i 所更新的所有数据项的值恢复成旧值。
- redo（T_i）将事务 T_i 所更新的所有数据项的值置为新值。

T_i 所更新的数据项的集合及其相应的旧值和新值均能在日志中找到。

操作 undo 和 redo 必须是幂等的，这样即使在恢复过程中发生故障也能保证正确性。

故障发生后，恢复机制检查日志，决定哪个事务需要 redo，哪个需要 undo。这是通过如下原则划分的：

- 事务 T_i 需要 undo，如果日志包含记录<T_i start>，但不包含记录<T_i commit>。
- 事务 T_i 需要 redo，如果日志既包含记录<T_i start>又包含记录<T_i commit>。

例如，再看银行的例子，事务 T_0 和 T_1 以先 T_0 后 T_1 的次序串行执行。假设系统在事务完成之前崩溃，我们考虑三种情况，这三种情况下日志的状态如图 15-7 所示。

a)	b)	c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

图 15-7　相同的日志，但分别在三个不同时刻

首先，假设对应于事务 T_0 中的

write (B)

的日志记录刚刚写入稳定存储器后就发生系统崩溃（图 15-7a)。当系统重新启动时，发现日志中有记录<T_0 start>，但没有相应的<T_0 commit>记录，事务 T_0 的操作必须取消，于是执行 undo (T_0)。这样，帐户 A 和 B 的值（在磁盘上）分别恢复为 $1000 和 $2000。

其次，假设对应于事务 T_1 中的

write (C)

的日志记录刚刚写入稳定存储器后就发生系统崩溃（图 15-7b)。当系统重新启动时，需要执行两个恢复动作：一个执行操作 undo (T_1)，因为日志中有记录<T_1 start>，但没有记录<T_1 commit>；另一个执行操作 redo (T_0)，因为日志中既包含记录<T_0 start>又包含记录<T_0 commit>。整个恢复过程的最后，帐户 A、B 和 C 的值分别为 $950、$2050 和 $700。注意，操作 undo ($T_1$) 是在 redo ($T_0$) 之前执行的。本例中，如果颠倒顺序结果是一样的。但是，先做操作 undo 后做操作 redo 在恢复算法中很重要，我们将在 15.6 节中讨论。

最后，假设日志记录

<T_1 commit>

在刚刚写入稳定存储器后就发生系统崩溃（图 15-7c)。当系统重新启动时，事务 T_0 和 T_1 都需要重做，因为日志中有记录<T_0 start>和<T_0 commit>以及记录<T_1 start>和<T_1 commit>。执行恢复过程 redo (T_0) 和 redo (T_1) 后，帐户 A、B 和 C 的值分别为 $950、$2050 和 $600。

15.4.3 检查点

当系统故障发生时，必须检查日志，决定哪些事务需要 redo，哪些需要 undo。原则上需要搜索整个日志来决定该信息。这样有两个主要困难：

1）搜索过程太耗时。

2）根据提供的算法，大多数需要 redo 的事务都将其更新写入了数据库中。尽管对它们做 redo 操作不会造成不良后果，但会使恢复过程变得更长。

为降低这种开销，我们引入检查点。在执行时，系统使用上两节中所描述的两种技术之一维护日志。另外，系统周期性地执行检查点，它需要执行如下动作序列：

1）将当前位于主存的所有日志记录输出到稳定存储器上。

2）将所有修改了的缓冲块输出到磁盘上。

3）将日志记录<checkpoint>输出到稳定存储器。

检查点执行过程中，不允许事务执行任何更新动作，如写缓冲块或写日志记录。

日志中加入记录<checkpoint>使得系统提高了恢复过程的效率。例如在检查点前提交的事务 T_i，对于这个事务，记录<T_i commit>在日志中出现在记录<checkpoint>前。T_i 所做的任何数据库修改都必然已在检查点前或作为检查点本身的一部分写入了数据库。因此，在恢复时就不必再对 T_i 执行 redo 操作了。

这使得我们可以简化前面的恢复机制。（我们仍然假设事务串行执行。）故障发生后，恢复机制检查日志来确定最近一次检查点发生前开始执行的最近一个事务 T_i。要找到这个事务只

需从日志的尾部由后至前搜索日志，直到找到第一个<checkpoint>记录（由于我们由后至前搜索，该记录就是日志中的最后一个<checkpoint>记录），然后继续向前搜索直至发现下一个<T_i **start**>记录，该记录确定了事务 T_i。

一旦事务 T_i 被确定，则只需对事务 T_i 和事务 T_i 后开始执行的所有事务 T_j 执行 redo 和 undo 操作。让我们用集合 T 代表这些事务。日志的剩余部分（前面的部分）可以被忽略，并且可以根据需要随时被删除。具体执行什么恢复操作取决于是采用立即修改技术还是采用延迟修改技术。如果采用立即修改技术，所需恢复操作如下：

- 对 T 中所有事务 T_k，若日志中没有记录<T_k commit>，则执行 undo（T_k）。
- 对 T 中所有事务 T_k，若日志中有记录<T_k commit>，则执行 redo（T_k）。

显然，如果采用延迟修改技术，则不必执行 undo 操作。

例如，考虑按下标顺序执行的事务集合 $\{T_0, T_1, \cdots, T_{100}\}$。假设最近的检查点发生在事务 T_{67}执行的过程中。于是，恢复机制中只需要考虑事务 T_{67}，T_{68}，…，T_{100}。其中已提交的需要重做；否则需要取消操作。

在 15.6.3 节中，我们将讲述扩充检查点技术以用于并发事务处理。

15.5　影子分页

与基于日志的故障恢复技术不同的另一种技术叫做影子分页。影子分页技术是 13.3 节中的影子拷贝技术的重要改进。某些情况下，影子分页技术可能比前面所讨论的基于日志的方法还需要更少的磁盘访问。但我们也将看到影子分页技术有一些缺点。例如，扩充影子分页技术很难扩充到允许多个事务并发执行的情况。

和前面一样，数据库分成若干定长的块，称为页。页这个概念来源于操作系统，因为我们用分页技术进行内存管理。假设有 n 个页，标号为 1～n。（实践中，n 可能是几十万。）这些页不必按某种特定顺序存储在磁盘上（这有很多原因，我们已在第 10 章讨论过），但是，必须有一种方法使对任意给定的 i 能找到数据库中的第 i 页。我们用页表解决这个问题，如图 15-8 所示。页表有 n 项，每个数据库页对应一项。每一项包含一个指针指向磁盘上的页。第一项的指针指向数据库的第一页，第二项指向第二页，以此类推。图 15-8 中的例子表示数据库页的逻辑顺序不必与磁盘上页的物理顺序一致。

影子分页技术的主要思想是在一个事务的生存周期中维护两张页表：当前页表和影子页表。当事务开始时，两张页表是相同的。影子页表在事务执行过程中是从不改变的，当前页表在事务执行 write 操作时可能改变。所有 input 和 output 操作使用当前页表定位磁盘上的数据库页。

假设事务 T_j 执行一个 write（X）操作，并且 X 位于第 i 页。write 操作执行如下：

1）如果第 i 页（即 X 所在页）还不在主存，则发指令执行 input（X）。

2）如果这是该事务第一次向第 i 页执行写操作，那么按如下方式修改当前页表：

a. 找出磁盘中未使用的页。通常，数据库系统利用一个未使用（空闲）页的列表，正如第 10 章所看到的那样。

b. 将第 2）a 步找到的页从空闲页列表中删除，将第 i 页的内容拷贝到第 2）a 步找到的页中。

c. 修改当前页表使得第 i 项指向第 2)a 步找到的页。

3）将 x_j 的值赋给缓冲页中的 X。

将写操作的上述动作与 15.2.3 节中所描述的动作相比较，唯一的区别是我们新加入了一

步。上面的第 1）步和第 3）步对应于 15.2.3 节中的第 1）步和第 2）步。新加入的一步，即第 2）步对当前页表进行操作。图 15-9 表示了一个事务的影子页表和当前页表，该事务对只包含 10 页的数据库中的第 4 页执行了一次写操作。

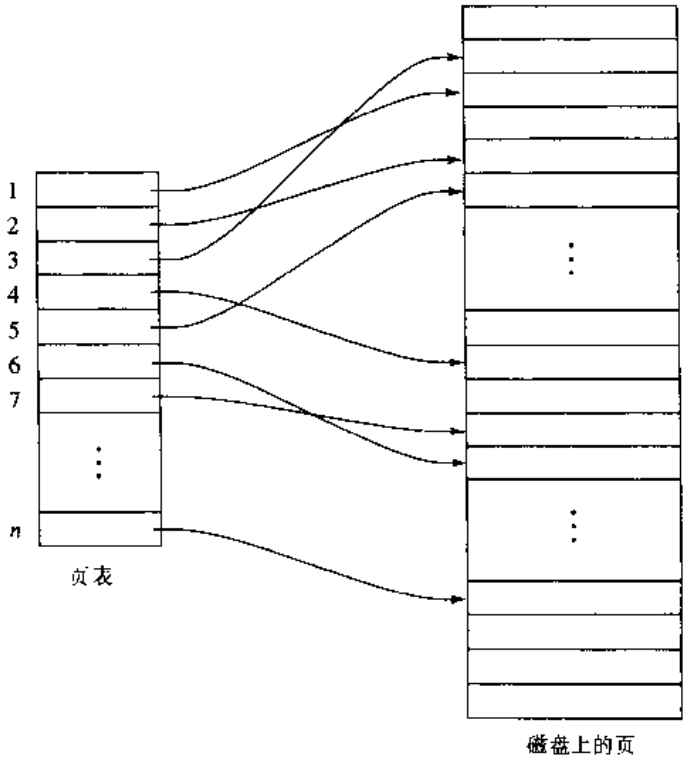


图 15-8　示例页表

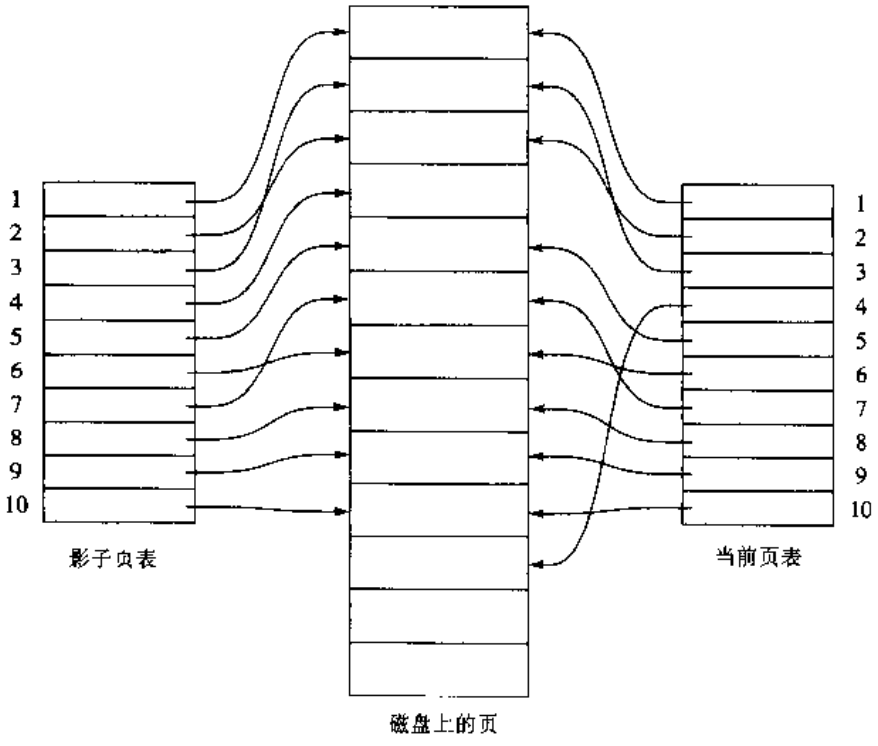


图 15-9　影子页表和当前页表

直观地，影子分页恢复方法是将影子页表存入非易失性存储器，使得执行事务前的数据库状态可以在发生系统崩溃或事务中止时得到恢复。当事务提交时，当前页表被写入非易失性存储器中。于是当前页表成为新的影子页表，下一个事务开始执行。这里重要的一点是影子页表存储在非易失性存储器上，因为这是对数据库页定位的唯一手段。当前页表可以保存在主存中（易失性存储器），我们不担心当前页表在系统崩溃时丢失，因为系统可以用影子页表进行恢复。

成功的恢复要求我们在系统崩溃后能在磁盘上找到影子页表。要找到它的简单方法是在稳定存储器上设置固定的区域记录影子页表的磁盘地址。当系统崩溃后重新启动时，拷贝影子页表至主存，并且用它进行后续的事务处理。根据我们对写操作过程的规定，我们可以保证影子页表指向的数据库页对应崩溃时活跃事务执行前的数据库状态。因此，中止事务的恢复是自动的，不像基于日志的机制那样需要调用 undo 操作。

要提交一个事务，必须做下列事情：

1）保证主存中所有被该事务修改过的缓冲页都被输出到了磁盘上。（注意这些输出操作不会修改影子页表中的某些项所指向的数据库页）。

2）将当前页表输出到磁盘上。注意不应覆盖影子页表，因为我们需要用它恢复。

3）将当前页表的磁盘地址输出到记录影子页表地址的稳定存储器的固定区域。这将覆盖旧的影子页表的地址。于是，当前页表就成了影子页表，事务提交完毕。

如果崩溃发生在第 3）步结束前，我们将状态恢复到该事务执行前的时刻。如果崩溃发生在第 3）步结束后，该事务的执行结果将被保存，不必调用 redo 操作。

影子分页与基于日志的技术相比有几点优势。它消除了日志记录输出的开销，并且从崩溃中恢复的速度也明显加快（因为不需要 undo 和 redo 操作）。但是，影子分页技术也有缺点：

• 提交开销。使用影子分页的事务提交时要求输出多个块——实际的数据块、当前页表和当前页表的磁盘地址。基于日志的机制只需要输出日志记录，对于通常的小事务来说，这些日志记录都位于一个块中。

• 数据分片。在第 10 章中，我们将相关的数据库页物理地放置在磁盘上相距较近的地方，这种局部性可加快数据传送。影子分页使得数据库页在更新时改变了位置。结果，我们要么会丧失页的局部特性，要么就必须寻求更复杂、开销更大的物理存储管理机制。（参见后面列出的文献注解）。

• 垃圾回收。每当一个事务提交时，包含该事务所修改数据旧值的数据库页就不可再访问了。在图 15-9 中，一旦例子中的事务提交，影子页表的第 4 项所指向的页就变成不可访问的页。这种页称为垃圾，因为它们不是空闲空间的一部分，并且不包含可用信息。垃圾也可能是系统崩溃的一个副产品。这就需要周期性地找出所有垃圾页，并且将它们加入空闲页的列表。这个过程称为垃圾回收，它增加了系统的开销和复杂性。垃圾回收有一些标准算法（参见后面列出的文献注解）。

除了这些缺点外，我们刚才还提到在支持多个事务并发执行的系统中采用影子分页比采用日志机制要难。在这种系统中，即使使用了影子分页，通常也还需要使用一些日志。例如，System R 原型就结合了影子分页技术与日志机制，其日志机制与 15.4.2 节中所讲类似。扩展基于日志的恢复机制以使它允许并发事务相对而言比较容易，我们将在 15.6 节中讨论。正由于这些原因，影子分页技术没有被广泛采用。

15.6　并发事务的恢复

到目前为止，我们只考虑了每一时刻就只有一个事务在执行的情况下的恢复。现在讨论如

何修改和扩展基于日志的恢复机制以处理多个事务并发的情况。不论并发事务几个，系统只有一个磁盘缓冲区和一个日志。缓冲块由所有事务共享。我们允许立即更新，并允许一个缓冲块中的数据项被一个或多个事务更新。

15.6.1 与并发控制的关系

恢复机制很大程度依赖于所用的并发控制机制。为回滚一个失败事务，必须撤消该事务所做的更新。假设事务 T_0 必须被回滚，并且被 T_0 更新的数据项 Q 必须恢复成旧值。当使用基于日志的机制进行恢复时，我们利用日志记录中的撤消信息进行恢复。现在假设第二个事务 T_1 在 T_0 回滚前对 Q 也做了一次更新。于是，如果 T_0 被回滚，T_1 所做的更新就会丢失。

因此我们要求，如果事务 T 更新了数据项 Q，在 T 提交或回滚前不允许其他事务更新该数据项。只要使用严格两阶段封锁协议就可以保证满足这个要求。严格两阶段封锁是在事务结束前使用排它锁的两阶段封锁。

15.6.2 事务回滚

我们利用日志回滚失败的事务 T_i。日志的扫描由后至前进行。对日志中的所有形如$<T_i, X_j, V_1, V_2>$的日志记录，数据项 X_j 被恢复成旧值 V_1。当发现日志记录$<T_i, \text{start}>$时，停止日志扫描。

由后至前扫描日志非常重要，因为一个事务可能多次更新一个数据项。例如，考虑两个日志记录

$$<T_i, A, 10, 20>$$
$$<T_i, A, 20, 30>$$

它们表示 T_i 对数据项 A 做了一次修改后又做了一次修改。由后至前扫描日志可以将 A 的值正确地恢复成 10。如果由前向后扫描日志，A 将被恢复成 20，但这个结果是不正确的。

如果并发控制用的是两阶段封锁协议，那么只有当事务 T 被回滚后，它所占用的锁才被释放。一旦事务 T（正在回滚的）已经更新了一个数据项，其他事务就不能再更新该数据项，因为要满足前面提到的并发控制要求。因此，恢复数据项的旧值不会影响其他事务的执行结果。

15.6.3 检查点

在 15.4.3 节中，我们利用检查点减少系统从崩溃中恢复时必须扫描的日志记录的数目。由于我们假设没有并发，恢复时我们只要考虑如下事务：

- 最近一次检查点后开始的那些事务。
- 若有的话，最近一次检查点时活跃的那个事务。

如果允许事务并发执行，情况会更为复杂，因为在最近一次检查点时可能有几个事务都是活跃的。

在并发事务处理系统中，我们要求检查点日志记录的形式为$<\text{checkpoint } L>$，其中 L 为检查点时活跃事务的列表。另外，我们假设事务在检查点进程中不执行更新操作，不论在缓冲块上还是日志上。

要求事务在检查点进程中不必向缓冲块或日志做更新可能有些麻烦，因为事务处理在检查

点进程中必须停止。模糊检查点则允许事务执行更新操作，甚至在缓冲块被写出时也如此。文献注解中提供了有关扩展恢复技术以允许模糊检查点的参考文献。模糊检查点机制在15.9.5节中阐述。

15.6.4 重启动恢复

当系统从崩溃中恢复时，它构造两个列表：*undo-list* 由需要撤消的事务构成，*redo-list* 由需要重做的事务构成。

这两个列表在恢复时按如下步骤构造。开始时它们都为空，然后我们由后至前扫描日志，检查每一记录，直至发现第一个<checkpoint>记录：

- 对每一个形如<T_i commit>的记录，将 T_i 加入 *redo-list*。
- 对每一个形如<T_i start>的记录，如果 T_i 不属于 *redo-list*，则将 T_i 加入 *undo-list*。

当所有相应的日志记录都被检查后，我们看检查点记录中的列表 L。对 L 中的每一个事务 T_i，如果 T_i 不属于 *redo-list*，则将 Ti 加入 *undo-list*。

一旦 redo-list 和 undo-list 构造完毕，恢复过程继续做如下工作：

1）从最后一个记录开始由后至前重新扫描日志，并且对 *undo-list* 中事务 T_i 的每一个日志记录执行 undo 操作。*redo-list* 中事务的日志记录在本步骤中忽略。当 *undo-list* 中的所有事务 T_i 所对应的<T_i start>记录都被找到时，扫描停止。

2）找出日志中的最后一个<checkpoint L>记录。注意，如果步骤1）已经经过了最近的检查点记录，这一步可能要由前至后扫描日志。

3）由最近一个<checkpoint L>记录开始由前至后扫描日志，并且对 *redo-list* 中事务 T_i 的每一个日志记录执行 redo 操作。本步骤中忽略 *undo-list* 中的事务的日志记录。

步骤1）中由后至前处理日志十分重要，它保证数据库的结果状态都是正确的。

当 *undo-list* 中的所有事务被撤消后，重做 *redo-list* 中的事务，这时重要的是要由前至后处理日志。当恢复过程完成时，事务处理重新开始。

若使用上述算法，在重做 *redo-list* 中的事务前撤消 *undo-list* 中的事务是很重要的；否则，将可能发生下面的问题。假设数据项 A 的初始值为10，并且事务 T_i 将数据项 A 更新为20后中止，那么事务回滚应将 A 的值恢复为10。假设另一事务 T_j 随后将数据项 A 的值更新为30并提交，这时系统崩溃。系统崩溃时的日志状态是

$$<T_i, A, 10, 20>$$
$$<T_j, A, 10, 30>$$
$$<T_j \text{ commit}>$$

如果先进行 redo 扫描，A 被置为30；然后，在 undo 扫描中，A 将被置为10，这样就出现了错误。A 最后的值应为30，我们只要先做 undo 再做 redo 就可以保证这个结果。

15.7 缓冲区管理

本节考虑几个微妙的细节，它们对实现能保证数据一致性但只增加少量与数据库交互的开销的故障恢复机制非常重要。

15.7.1 日志记录缓冲

前面，我们假设每个日志记录在创建时都被输出到了稳定存储器上。该假设大大增加了系

统执行的开销，其原因是通常向稳定存储器的输出是以块为单位进行的。大多数情况下，一个日志记录比一个块要小得多。因此，每个日志记录的输出在物理上被转化成了大得多的输出。另外，正如15.2.2节中所看到的，向稳定存储器输出一块可能在物理上涉及了几个输出操作。

将一个块输出到稳定存储器上的开销非常高，因此最好一次输出多个日志记录。为了达到这个目的，我们将日志记录写到主存的日志缓冲区中，日志记录在输出至稳定存储器以前临时保存在那里。多个日志记录可以集中在日志缓冲区中，用一次输出操作输出到稳定存储器中。稳定存储器中的日志记录顺序必须与写入日志缓冲区的顺序完全一样。

由于使用了日志缓冲区，日志记录在输出到稳定存储器前可能有一段时间只存在于主存（易失性存储器）中。由于系统发生崩溃时这种日志记录会丢失，因而必须对恢复技术增加一些要求以保证事务的原子性：

- 在日志记录<T_i commit>输出到稳定存储器后，事务 T_i 进入提交状态。
- 在日志记录<T_i commit>输出到稳定存储器前，与事务 T_i 有关的所有日志记录必须已经输出到稳定存储器中。
- 在主存中的数据块输出到数据库（非易失性存储器）前，所有与该数据块中数据有关的日志记录必须已经输出到稳定存储器中。

后两个规则称为先写日志（WAL）规则。（严格地说，WAL规则只要求日志中的undo信息已经被输出到了稳定存储器中，而redo信息允许以后再写。对undo信息和redo信息分别存储在不同的日志记录中的系统而言，这一区别并非无关紧要的）。

将缓冲的日志写到磁盘上有时称为强制日志。前面的规则表明在某些情况下，某些日志记录必须已经输出到了稳定存储器上。而提前输出日志记录不会造成任何问题。因此，当系统发现需要将日志记录输出到稳定存储器上时，如果主存中有足够的日志记录可以填满整个日志记录块，就将其整个输出。如果没有足够的日志记录填入该块，那么就将主存中的所有日志记录填入半满的块，再输出到稳定存储器上。

15.7.2 数据库缓冲

15.2节描述了两层存储结构的使用。数据库存储在非易失性存储器（磁盘）上，并且在需要时将数据块调入主存。由于主存通常比整个数据库小得多，所以可能在需要将块 B_2 调入主存时覆盖主存中的块 B_1。如果 B_1 已经被修改过，那么 B_1 必须在输入 B_2 前就输出。与10.5.1节中讨论的一样，这种存储结构是操作系统中标准的虚拟内存概念。

输出日志记录的规则限制了系统输出数据块的自由。如果块 B_2 的输入使得必须输出块 B_1，那么所有与 B_1 中数据有关的日志记录必须在 B_1 输出前输出到稳定存储器上。因此，系统将采取如下一系列动作：

- 输出日志记录至稳定存储器，直至所有与块 B_1 有关的日志记录都已被输出。
- 将块 B_1 输出到磁盘上。
- 将块 B_2 由磁盘输入到主存中。

重要的是，当上述动作执行时，不能对块 B_1 进行写操作。我们可以使用下面一种特殊的封锁机制来加以保证。在事务对数据项执行写操作前，它必须获得该数据项所处块上的排它锁，这个锁在更新执行后可以立即释放。在一个块输出前，系统获得该块上的排它锁，以保证没有事务更新该块。一旦该块的输出结束，锁就可以释放。保持时间短的锁通常称为闩锁。闩锁的处理不同于并发控制系统所使用的锁，因此，它们的释放不必遵守任何封锁协议，如并发控制系统中的两阶段封锁协议。

为阐明上述一系列动作的必要性，我们考虑银行例子中的事务 T_0 和 T_1。假设日志的状态是

$$<T_0 \text{ start}>$$
$$<T_0, A, 1000, 950>$$

并假设事务 T_0 发出指令执行 read（B）。假设 B 所在的块不在主存中，并且主存已满。另外还假设选择了 A 所在的块输出到磁盘上。如果将该块输出到磁盘上后系统崩溃，数据库中帐户 A、B 和 C 的值分别就是 $950、$2000 和 $700，这是不一致的数据库状态。但是，由于上述要求，日志记录$<T_0, A, 1000, 950>$必须在输出 A 所在块之前输出到稳定存储器中，因此系统可以在恢复时使用该日志记录将数据库恢复到一致状态。

15.7.3　操作系统在缓冲区管理中的作用

我们可以用下面两种方法之一管理数据库缓冲区：

1）数据库系统保留部分主存作为缓冲区，并对它进行管理，而不是让操作系统来管理。数据库系统按照前面讨论的那些要求管理数据块的传送。

这种方法的缺点是限制了主存使用的灵活性。缓冲区必须足够小，以使其他应用有足够的主存满足需要。但是，即使其他应用并未运行，数据库系统也不能利用所有可用内存。同样，非数据库应用也不能使用为数据库缓冲区保留的那部分主存，即使数据库缓冲区中的一些页并未被使用。

2）数据库系统在操作系统提供的虚拟内存中实现其缓冲区。由于操作系统知道系统中的所有进程的内存需求，所以最好由它决定哪个缓冲块在什么时候必须被强制输出到磁盘上。但是，为保证前面提到的先写日志要求，不应由操作系统自己写数据库缓冲页，而应由数据库系统强制输出缓冲块。数据库系统在将相关日志记录写入稳定存储器后，强制输出缓冲块至数据库中。

遗憾的是，几乎所有当前操作系统都完全控制虚拟内存。操作系统保留磁盘空间以存储当前不在主存的那些虚拟内存页，这些空间称为交换区。如果操作系统决定输出一个块 B_X，该块就被输出到磁盘交换区中，这样就没有办法让数据库系统控制缓冲块的输出。

因此，如果数据库缓冲区在虚拟内存中，那么数据库文件和虚拟内存缓冲区之间的数据传送就必由数据库系统来管理，它可以实现前面提到的先写日志的要求。

这种方法可能导致更多的数据到磁盘的输出。如果块 B_X 由操作系统输出，则该块不是输出到数据库中，而是输出到为操作系统的虚拟内存准备的交换区中。当数据库系统需要输出 B_X，操作系统可能需要先从交换区输入 B_X。因此，这里可能不只一次的 B_X 输出，而是两次 B_X 输出（一次由操作系统进行，一次由数据库系统进行）和一次 B_X 输入。

尽管两种方法都有一些缺点，但总要选择其一，除非操作系统设计成支持数据库日志的要求。当今只有几个操作系统支持这种要求，如 Mach 操作系统。

15.8　非易失性存储器数据丢失的故障

到目前为止，我们只考虑了一种情况，即故障导致了易失性存储器上的信息丢失，但非易失性存储器保持完好。尽管导致非易失性存储器上内容丢失的故障极少，我们仍需时刻准备对这种类型的故障加以处理。本节只讨论磁盘存储器，这些讨论也适用于其他类型的非易失性存

储器。

基本方法是周期性地将整个数据库的内容转储到稳定存储器上，比如一天一次。可以将数据库转储到一个或多个磁带上。如果发生了一个导致物理数据库块丢失的故障，就可以用最后一次转储将数据库恢复到故障前的一致状态。一旦恢复结束，系统再利用日志将数据库系统恢复到最近的一致状态。

更精确地说，转储过程中没有事务处于活跃状态，并且必须执行一个类似于检查点的过程：

1）将所有当前位于上存的日志记录输出到稳定存储器中。

2）将所有缓冲块输出到磁盘上。

3）将数据库的内容拷贝到稳定存储器中。

4）将日志记录<dump>输出到稳定存储器中。

第1）、2）和4）步对应于15.4.3节中用于检查点的那三步。

为从非易失性存储器数据丢失中恢复，我们用最后一次转储将数据库恢复到磁盘上。然后，根据日志，重做所有最后一次转储后提交的事务。注意，这里不必执行任何undo操作。

数据库内容的转储也称为*归档转储*，因为我们可以将转储归档，以后用它们来查看数据库的旧状态。数据库转储和缓冲区的检查点机制很相似。

这里描述的简单转储过程开销较大，原因有以下两个。首先，整个数据库都必须拷贝到稳定存储器中，这导致大量的数据传送。其次，由于在转储过程中要中止事务处理，这样就浪费了CPU周期。*模糊转储*机制对此做了改进，它允许转储过程中事务仍是活跃的。这一点类似于模糊检查点机制，详细情况可参见后面列出的文献注解。

15.9 高级恢复技术

15.6节中描述的恢复技术要求一旦事务更新了一个数据项，其他事务就都不能更新该数据项，直至第一个事务提交或回滚。我们通过使用严格两阶段封锁协议保证满足该要求。尽管如14.8节中所讨论的那样，严格两阶段封锁协议对关系中的记录是可以接受的，但当将它应用到某些特殊结构如B^+树索引页时，并发性极度下降。

为提高并发性，我们可以使用14.8节中所描述的B^+树并发控制算法，通过非两段方式使锁可以较早释放，但这导致15.6节中的恢复技术无法使用。现在已提出了其他几种可用的恢复技术，它们允许锁较早释放。本节我们描述其中一种恢复技术。

15.9.1 逻辑Undo日志

对于锁提前释放的情况，我们执行撤消操作时不能只是将数据项的旧值写回。假如事务T向B^+树插入了一项，并且根据B^+树的并发控制协议，在插入操作结束后但在事务提交前释放了某些锁。在锁释放后，其他事务可以执行插入或删除操作，于是造成对B^+树页的进一步改变。

尽管操作提前释放了某些锁，但它必须保持必要的锁以保证其他事务不会执行任何冲突操作（如读插入值，或删除插入值）。因此，前面讨论的B^+树并发控制协议要保持B^+树叶层的锁直至事务结束。

现在我们考虑如何执行事务回滚。如果事务回滚时将B^+树内部结点（执行插入操作前）的旧值写回，那么其他事务在其后执行的插入或删除操作所做的某些更新可能会丢失。因此我们不这样做，代之的是逻辑地撤消插入操作，即通过执行一次删除操作撤消。

因此，当插入操作结束时，在释放任何锁之前，它写入一条日志记录<T_i，O_j，operation-end，U>，其中 U 代表 undo 信息，O_j 代表某次操作的唯一标识。例如，如果某操作往 B^+ 树中插入了一项，则 undo 信息 U 将指明需要执行的一次删除操作，并指明要从 B^+ 树中删除什么。这种有关操作的日志信息称为逻辑日志。反之，在日志中记录旧值和新值信息称为物理日志，相应的日志记录称为物理日志记录。

由于提前释放锁而需要逻辑 undo 操作，插入和删除操作就是其中的例子，我们称这种操作为逻辑操作。在逻辑操作开始前，它写入日志记录<T_i，O_j，operation-begin>，其中 O_j 是该操作的唯一标识。在操作执行时，该操作的所有更新以正常方式记录在日志中。于是，每一次更新的旧值和新值信息都被写入。当操作结束时，一个 operation-end 日志记录像前面所述那样被写入。

15.9.2　事务回滚

我们先考虑正常操作时（即不是在系统故障恢复时）的事务回滚。日志由后至前扫描，并用属于该事务的日志记录恢复数据项的旧值。与前面不同，如果回滚时数据项 X_j 的值恢复成 V，我们就写入特殊的形如<T_i，X_j，V>的只读日志记录。这些日志记录有时被称为补偿日志记录。每当发现日志记录<T_i，O_j，operation-end，U>，就要执行如下特殊动作：

1）我们利用日志记录中的 undo 信息 U 回滚操作。回滚该操作时执行的更新被记入日志，就像记录第一次执行该操作时所做的更新。只不过不是写入日志记录<T_i，O_j，operation-end，U>，而是写入日志记录<T_i，O_j，operation-abort>。

2）当继续由后至前扫描日志时，跳过该事务的所有日志记录，直至发现日志记录<T_i，O_i，operation-begin>。发现日志记录 operation-begin 后，该事务的日志记录以通常的方式再次被处理。

如果发现记录<T_i，O_j，operation-abort>，则所有前面的记录都被跳过直至发现记录<T_i，O_j，operation-begin>。这些日志记录必须跳过，以防止多次回滚同一个操作，比方说前一次回滚时发生了故障，同时事务已经被部分回滚。当事务 T_i 回滚后，记录<T_i abort>加入日志中。

如果逻辑操作过程中发生了故障，则事务回滚时将找不到该操作的日志记录 operation-end。但是，对该操作执行的所有更新，undo 信息——在物理日志记录中以旧值形式存在——都可在日志中找到。这些物理日志记录将用于回滚不完全的操作。注意，在回滚过程中当发现日志记录 operation-end 时就跳过物理日志记录，以保证一旦操作结束，物理日志记录中的旧值不被用于回滚。

如果并发控制使用了封锁机制，那么事务 T 所拥有的锁可能在事务回滚后才被释放。

15.9.3　检查点

检查点的执行正如 15.6 节中所描述的那样。对数据库的更新暂时中止，并执行如下动作：

1）将当前位于主存的所有日志记录输出到稳定存储器上。

2）将日志记录<checkpoint L>输出到稳定存储器上，其中 L 是所有活跃事务的列表。

3）将所有修改了的缓冲块输出到磁盘上。

15.9.4　重启动恢复

当数据库系统在故障后重新启动时，恢复动作分为两个阶段：

1）在 redo 阶段，通过从最新一个检查点向后扫描日志来重新执行所有事务的更新。重新执行的日志记录包括系统崩溃前回滚事务的日志记录，以及系统崩溃发生时还未提交事务的日志记录。其中有普通的形如$<T_i, X_j, V_1, V_2>$的日志记录和特殊的形如$<T_i, X_j, V_2>$的日志记录，它们都表示 V_2 的值写入数据项 X_j。这一阶段也找出了日志中所有要么在检查点记录的事务列表中，要么在其后开始，但在日志中既无$<T_i$ abort$>$记录又无$<T_i$ commit$>$记录的事务。所有这些事务都必须回滚，其事务标识被放入 *undo-list*。

2）在 undo 阶段，回滚 *undo-list* 中的所有事务。我们通过由后至前扫描日志执行回滚。每当发现属于 *undo-list* 中事务的日志记录，就以失败事务回滚过程中发现日志记录一样的方式执行 undo 动作。也就是说，一个事务在 operation-end 记录前但在相应的 operation-begin 记录后的日志记录被忽略掉。

当发现 *undo-list* 中事务 T_i 的日志记录$<T_i$ start$>$，则往日志中写入日志记录$<T_i$ abort$>$。当 *undo-list* 中所有事务的日志记录$<T_i$ start$>$都被找到时，就停止对日志的扫描。

重启动恢复的 redo 阶段重新执行最新一个检查点后的每个物理日志记录。换句话说，重启动恢复的这个阶段重复执行该检查点后执行的并且其日志记录已写入稳定存储器的所有更新动作。这些动作包括未结束事务的动作以及为回滚失败事务而执行的动作。它们重复执行的顺序与以前执行时的顺序相同，因此，该过程称为重复历史。重复历史大大简化了恢复机制。

15.9.5 模糊检查点

15.6.3 节中所描述的检查点技术要求对数据库的所有更新在检查点进行过程中暂时中止。其实可以改进这个技术，以允许在 checkpoint 记录写入日志后但在修改过的缓冲块写到磁盘前做更新。这样产生的检查点称为*模糊检查点*。

具体想法如下。我们不是由后至前地扫描日志以找出检查点记录，而是将最后一个检查点记录在日志中的位置存到磁盘上固定的位置 last_checkpoint 上。但是，该信息在 checkpoint 记录写入时不更新，而是在写 checkpoint 记录前，创建所有修改过的缓冲块的列表。只有该列表中的所有缓冲块都输出到了磁盘上以后，last_checkpoint 信息才会更新。缓冲块在输出到磁盘时不能更新，并且必须遵守先写日志协议，使得与该块有关的（undo）日志记录在该块输出前已写入了稳定存储器。注意，在本书所讲述的机制中，逻辑日志只用于 undo 操作，而物理日志用于 redo 和 undo 操作。有一些恢复机制用逻辑日志做 redo 操作，但这种机制不能用于模糊检查点，因此并未得到广泛应用。

15.10 总结

计算机系统与其他机械或电子设备一样容易发生故障。造成故障的原因有很多，包括磁盘故障、电源故障和软件错误。这些故障造成了数据库系统信息的丢失。除系统故障外，事务也可能因各种原因而失败，如破坏了完整性约束或发生死锁。数据库系统的一个重要组成部分就是恢复机制，它负责检测故障以及将数据库恢复至故障发生前的某一状态。

计算机中的存储器类型有易失性存储器、非易失性存储器和稳定存储器。易失性存储器如 RAM 中的数据在计算机发生故障时会丢失。非易失性存储器如磁盘中的数据在计算机发生故障时一般不会丢失，只是偶尔由于某些故障如磁盘故障才会丢失。稳定存储器中的数据从不丢失。必须能联机访问的稳定存储器用于磁盘镜像或 RAID 的其他形式模拟，它们提供冗余数据存储。脱机成归档稳定存储器可能是数据的多个磁带备份，并被存放在物理上安全的地方。

一旦故障发生，数据库系统的状态可能不再一致，即它不能反映数据库试图保存的现实世

界的状态。为保持一致性，我们要求每个事务都必须是原子的。恢复机制的责任就是要保证原子性。保证原子性主要有两种不同机制。

• 基于日志的。所有更新都记入日志，并被存放在稳定存储器上。在延迟修改机制中，事务执行时所有 write 操作要延迟到事务部分提交后才执行，那样，日志中与该事务有关的信息用于执行延迟写。在立即修改机制中，所有更新直接作用于数据库。如果故障发生，日志中的信息用于将数据库的状态恢复到故障前的一致状态。为减少搜索日志和重做事务的开销，我们可以使用检查点技术。

• 影子分页。在事务的生命周期中维护两张页表：当前页表和影子页表。当事务开始时，两张页表是相同的。影子页表在事务执行过程中从不改变。事务执行 write 操作时当前页表可能改变。所有 input 和 output 操作使用当前页表定位磁盘上的数据库页。当事务部分提交时，影子页表被丢弃，当前页表成为新的影子页表。如果事务中止，则只要丢弃当前页表。

如果允许多个事务并发执行，则不能用影子分页技术，但可以使用基于日志的技术。尚未结束的事务更新的数据项不允许其他事务再做更新。我们可以用严格两阶段封锁协议来保证这一要求。

事务处理基于主存中有日志缓冲区、数据库缓冲区和系统缓冲区这一存储模型。系统缓冲区的页中有系统目标代码和事务的局部工作区。

恢复机制的有效实现要求写数据库和稳定存储器的次数尽量少。日志记录可能一开始放在易失性日志缓冲区中，但必须在下列情况之一发生时写入稳定存储器：

• 在日志记录$<T_i$ commit$>$输出到稳定存储器前，所有与事务 T_i 有关的日志记录必须已被输出到稳定存储器中。

• 在主存中的数据块输出到数据库（非易失性存储器）中之前，所有与该块中数据有关的日志记录必须已被输出到稳定存储器中。

为从导致非易失性存储器中数据丢失的故障中恢复，可以周期性地将整个数据库的内容转储到稳定存储器上——例如，每天一次。如果发生了导致物理数据库块丢失的故障，可以用最后一次转储将数据库恢复至故障前的某个一致状态。一旦完成该恢复，可以再用日志将数据库系统恢复至最近的一致状态。

现在已设计出一些高级恢复技术来支持高并发性封锁技术，如用于 B^+ 树并发控制。这些技术基于逻辑（操作）undo，并且遵照重复历史的原则。当从系统故障中进行恢复时，我们用日志执行一遍 redo，然后用日志执行一遍 undo 以回滚未完成的事务。

习题

15.1　从 I/O 开销的角度解释易失性存储器、非易失性存储器和稳定存储器三种存储器的区别。

15.2　稳定存储器是不可能实现的。

(a) 解释为什么。

(b) 解释数据库系统如何解决这个问题。

15.3　从实现的难易程度和开销的角度比较延迟修改和立即修改这两个基于日志的恢复机制。

15.4　假设系统使用立即修改方式，用例子说明如果在事务更新的数据写入磁盘前有关该事务的日志记录未输出至稳定存储器中，为什么会造成不一致的数据库状态。

15.5　解释检查点机制的目的。检查点应多长时间执行一次？检查点的频率如何影响：

• 没有故障发生时系统的性能。

- 从系统故障中恢复所占用的时间。
- 从磁盘故障中恢复所占用的时间。

15.6 当系统从故障中恢复时（参见15.6.4节），它构造一个undo-list和一个redo-list。解释为什么undo-list中事务的日志记录必须由后至前进行处理，而redo-list中事务的日志记录则必须由前至后进行处理。

15.7 从实现的难易程度和开销的角度比较影子分页恢复机制和基于日志的恢复机制。

15.8 考虑一个包含10个连续磁盘块（块1，块2，…，块10）的数据库。假设采用影子分页技术，主存中的缓冲区只能容纳三个块，并且缓冲区管理采用最近最少使用策略(LRU)。给出在下列更新后的缓冲区状态和块的可能的物理顺序。

read block 3
read block 7
read block 5
read block 3
read block 1
modify block 1
read block 10
modify block 5

15.9 解释如果在块输出到磁盘前，与该块有关的某些日志记录还未输出到稳定存储器上，缓冲区管理器如何造成数据库状态的不一致。

15.10 解释磁盘故障后需要的恢复过程。

15.11 解释逻辑日志的好处。给出一个例子说明逻辑日志胜于物理日志，另一个例子说明物理日志胜于逻辑日志。

15.12 解释为什么交互事务的恢复比批处理事务的恢复更难处理，是否有简单的方法处理该困难？
（提示：参考用于提取现金的自动取款机事务。）

15.13 更新操作的日志在持久编程语言中不显式执行。说明现代操作系统所提供的页存取保护机制如何用于创建更新页的前映像和后映像。
（提示：参见习题14.29。）

文献注解

有关恢复的理论工作最初分别是在Davies［1973］和Bjork［1973］的两篇早期论文中阐述的。Chandy等［1975］所阐述的数据库系统中的回滚和恢复策略的分析模型是该领域的另一个早期著作。

本章中引入的大多数恢复机制基于System R中所用的恢复机制。有关该系统的恢复机制的概述在Gray等［1981a］的书中做了阐述。System R的影子分页机制由Lorie［1977］描述。Reuter［1980］描述了快速的面向事务的日志机制。

Gray［1978］、Lindsay等［1980］和Verhofstad［1978］中有关于数据库系统的各种恢复技术的指导性和概述性文章。模糊检查点和模糊转储的概念是在Lindsay等［1980］中阐述的。Haerder和Reuter［1983］中有恢复原理的全面阐述。Date［1983a］和Bernstein等［1987］可作

为教材。Gray 和 Reuter［1993］是有关恢复的很不错的教材，它包含了实现和历史发展的详细资料。

恢复方法的技巧在 Mohan 等［1992］、Mohan 和 Narang［1995］以及 Mohan［1990b］的 ARIES 恢复方法中说明得最清楚。15.9 节中描述的恢复技术部分基于 ARIES 恢复方法。索引结构的特殊恢复技术在 Mohan 和 Levine［1992］，以及 Mohan［1993］中做了阐述。Mohan 和 Narang［1994］描述了客户－服务器结构的恢复技术。

如何为灾难性恢复做远程备份（由于火灾、洪水、地震等原因造成的整个计算设备的损坏）在 Polyzois 和 Garcia-Molina［1994］以及 King 等［1991］中做了讨论。

数据库管理中涉及的操作系统问题在 Stonebraker［1981］、Traiger［1982，1983］以及 Haskin 等［1988］中做了讨论。与长事务及其相关恢复问题有关的文献注解将在第 20 章列出。

第 16 章　数据库系统体系结构

数据库系统运行于计算机系统之上，数据库系统的体系结构与计算机系统的体系结构密切相关。计算机体系结构的各个方面，如网络、并行、分布等，都反映到数据库系统的体系结构中。

• 计算机连网可以使得某些任务在服务器系统上执行，而另一些任务在客户系统上执行。这种工作任务的划分导致了客户－服务器数据库系统的发展。

• 计算机系统中的并行处理能力能够加速数据库系统的活动，对事务做出更快速的响应，在单位时间内处理更多的事务。查询能够充分利用计算机系统所提供的并行能力来处理。并行查询处理的需要导致了并行数据库系统的发展。

• 在一个组织机构的多个节点或部门间对数据进行分布，可以使得数据能驻留在该数据的产生地或最需要该数据的地方，而同时又支持其他节点部门对该数据的访问。在不同节点上保存数据库的多个副本还使得大型组织机构能够在一个节点受水灾、火灾、地震等自然灾害影响而遭到破坏的情况下继续进行数据库操作。人们开发分布式数据库系统来管理分布在多个数据库系统中的数据。这种分布可以是地理上的，也可以是行政管理上的。本章我们研究数据库系统的体系结构，首先讨论集中式系统，然后讨论客户-服务器系统、并行系统、以及分布式系统。

16.1　集中式系统

集中式系统是指运行在一台计算机上，不与其他计算机系统交互的数据库系统。这样的系统范围很广，既包括运行在个人计算机上的单用户数据库系统，也包括运行在大型主机上的高性能数据库系统。

现代通用的计算机系统包括一到多个 CPU，以及若干个设备控制器，它们通过公共总线连接在一起，从而实现对共享主存储器的访问（如图 16-1 所示）。CPU 具有本地的高速缓冲存储器，用于存放主存储器中某些数据的本地拷贝，从而加快对数据的访问。每个设备控制器负责一种类型的设备（例如磁盘设备、音响设备、或影视播放设备）。CPU 和设备控制器可以并行工作，它们的竞争主要针对于主存储器的访问。高速缓冲存储器减轻了对主存储器的访问强度，因为它减少了 CPU 需要访问共享主存储器的次数。

我们使用计算机的方式分为两类：单用户系统和多用户系统。个人计算机和工作站属于第一类。典型的单用户系统是个人使用的桌面系统，它包括一个 CPU 和 1～2 个磁盘，以及仅支持一个用户的操作系统。典型的多用户系统有多个磁盘和多个主存储器，还可能有多个 CPU，并且有一个多用户操作系统。它为大量的用户服务，这些用户通过终端与系统相连。这样的系统通常称作服务器系统。

为个人计算机这样的单用户系统设计的数据库系统一般不提供多用户数据库系统所提供的许多特性。特别地，它们不支持并发控制，因为当仅有一个用户才能进行更新时并发控制是不需要的。在这样的系统中，故障恢复能力或者没有，或者非常有限，例如，仅在任何更新之前做一个简单的数据库备份。许多这样的系统也不支持 SQL，而是提供一个简单的查询语言，例

如 QBE 的变种。

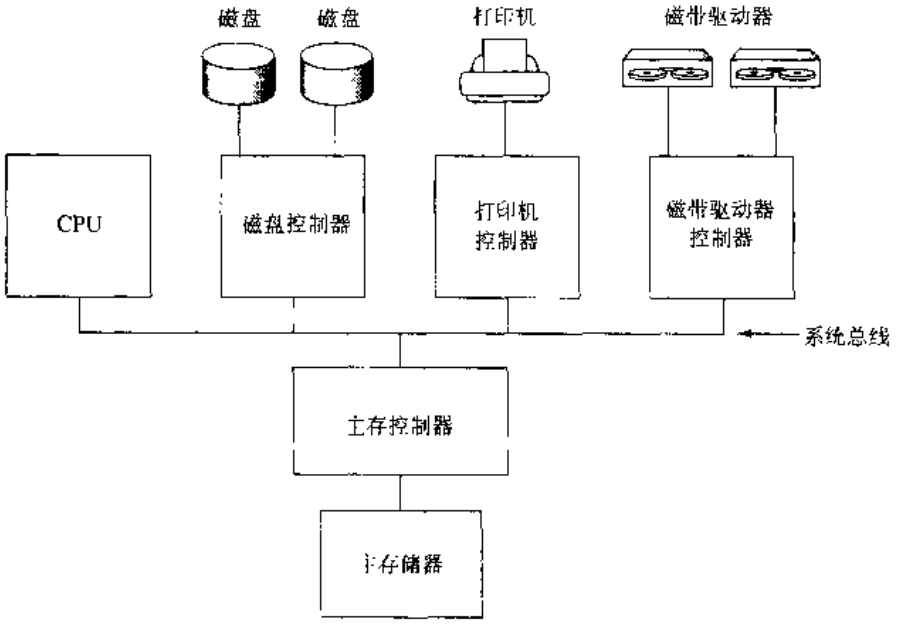


图 16-1　集中式计算机系统

虽然当今通用的计算机系统有多个处理器，但它们是粗粒度的并行，只具有几个处理器（一般 2～4 个），共享一个主存。在这种机器上运行的数据库一般不将一个查询分配到多个处理器上，而是在每个处理器上运行一个查询，从而使多个查询能并行地运行。因此这样的系统能提供较高的吞吐量，即尽管单个事务没有运行得更快，但在每秒钟里却运行了更大量的事务。

为单处理器机器设计的数据库已经能够提供多任务的能力，从而使得多个进程能以分时的方式运行在同一个处理器上，这样用户看起来像是多个进程在并行地运行。因此，粗粒度的并行机逻辑上几乎和单处理器机器一样，为分时机器设计的数据库系统很容易就能适应在粗粒度并行机上运行。

与此相反，细粒度并行机拥有大量的处理器，在这种机器上运行的数据库系统将用户提交的单个任务（例如查询）并行地执行。我们在 16.3 节中研究并行数据库系统。

16.2　客户－服务器系统

由于个人计算机变得速度更快、能力更强、价格更低。因此集中式体系结构发生了变化。连接到集中式系统的终端现在被个人计算机所代替。相应地，以前由集中式系统直接执行的用户界面功能现在越来越多地由个人计算机来处理。其结果是，集中式系统已变成起服务器系统的作用，它满足由客户系统产生的请求。客户－服务器系统的一般结构如图 16-2 所示。

数据库功能可以大致地分为两个部分：前端和后端，如图 16-3 所示的那样。后端负责存取结构、查询计算和优化、并发控制，以及故障恢复。数据库系统的前端包括表格生成工具、报表书写工具、图形用户界面工具等。前端与后端之间通过 SQL 或应用程序来接口。

服务器可以大致分为事务服务器和数据服务器两类。

• 事务服务器系统也称作查询服务器系统，它提供一个接口，使得客户可以发出执行一个动作的请求，服务器响应客户的请求，执行该动作，并将结果送回给客户。用户可以用 SQL 表达请求，也可以通过应用程序接口，使用远程过程调用机制来表达请求。

• 数据服务器系统使得客户可以向服务器发出请求，以文件或页面等为单位对数据进行读取或更新。例如，文件服务器提供文件系统界面，使客户能够进行文件的创建、读取、更新、删除。数据库系统的数据服务器提供更强的功能，所支持的数据单位比文件要小，可以是页面、元组、对象等。它还提供数据的索引机制，以及事务机制，从而保证即使客户机器或进程发生故障，数据也不会处于不一致状态。

我们在下两节对事务服务器和数据服务器进行详细描述。

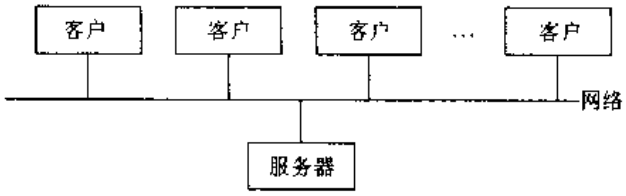


图 16-2　客户 - 服务器系统的一般结构

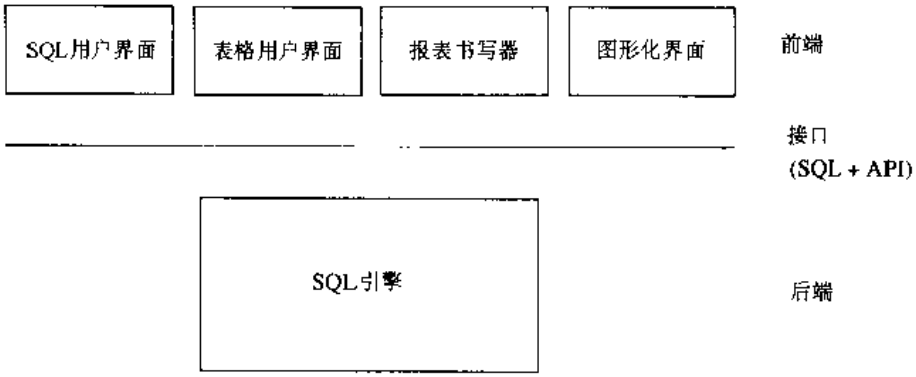


图 16-3　前端功能和后端功能

16.2.1　事务服务器

对于集中式系统，前端和后端都在同一个系统中运行。然而，事务服务器体系结构能够在前端和后端之间进行功能划分。由于图形用户界面代码具有更多的处理需求，也由于个人计算机的能力越来越强，所以前端的功能由个人计算机来支持。个人计算机充当服务器系统的客户，服务器中存储大量数据，并提供后端功能。客户将事务送给服务器系统，由服务器系统来执行事务并把结果送回给客户，由客户负责数据的显示。

关于客户与服务器之间的接口已经制定了一些标准，例如开放数据库互连（ODBC）标准。ODBC 是一个应用程序接口，它使得客户可以生成 SQL 语句，送到服务器去执行。使用 ODBC 接口的任何客户可以与提供 ODBC 接口的任何服务器连接。在早先的数据库系统中，由于缺乏这样的标准，前端与后端必须是同一个软件厂商提供，它们之间才能连接。

由于接口标准的发展，各个不同厂商提供了越来越多的前端工具和后端服务器。Gupta SQL 和 Power-Builder 就是独立于后端服务器的前端系统的典型代表。而且，像电子表格和统计软件包这样特定的应用程序也直接使用客户 - 服务器接口去访问后端服务器中的数据。实际上，这些应用程序提供了专用于特定任务的前端。

除 ODBC 之外，某些事务处理系统中也使用其他 -些客户 - 服务器接口。这些接口通过应用程序接口来定义，客户使用这些接口对服务器进行事务远程过程调用。这些调用在程序员看

来和通常的过程调用一样，但是同一个客户发出的所有远程过程调用在服务器方都包括在单一的一个事务中，这样，如果事务中止，服务器将消除各个远程过程调用所带来的影响。

单台小机器的购买和维护价格都越来越便宜，这一事实导致了整个产业朝低档机器发展的趋势。许多公司正用连接到后端服务器机器的工作站或个人计算机网络代替他们原来的大型主机。这样做的好处很多，包括付出较低的代价而获得较强的功能、资源配置和功能扩展方面更大的灵活性、更好的用户界面和更易于维护。

16.2.2　数据服务器

数据服务器系统使用在局域网中，客户与服务器之间具有高速的连接，客户机器在处理能力上与服务器机器是相当的，并且其执行的任务主要以计算为主。在这样的环境中，通常采用如下做法：数据传送到客户机器，在客户机器上进行所有的处理（这样的处理可能要花一些时间），然后再把数据传回到服务器机器。注意，这种体系结构需要将后端的所有功能（参见图16-3）都放到客户端。在面向对象数据库系统中数据服务器体系结构更为常见。

由于客户与服务器之间通信的时间代价与本地主存储器引用的代价相比要高得多（数百万秒相对于不到100纳秒），因此，这样的体系结构引发了一些很有意思的问题。

• 页面传送与项传送。数据通信的单位可以是粗粒度的，例如页面；也可以是细粒度的，例如元组（或面向对象数据库系统中的对象）。我们使用项这个词来代表元组和对象。

如果通信的单位是单个的项，那么消息传送的开销就大于数据传输的数量。而当请求一个项时，将不久的将来很有可能会用到的其他项也传送回去是有好处的。将有些项在请求之前就取过来称作预读取。如果多个项驻留在一个页面上，那么页面传送也可以看作是一种预读取，因为当进程要求访问页面上的一个项时，也将该页面上的所有项都传送过去了。

• 封锁。对于传送给客户机器的数据项的锁通常由服务器给予。页面传送的一个缺点是给予服务器机器的锁的粒度可能太大，加在页面上的锁意味着锁住了该页面上的所有项。即使客户不访问该页面上的某些项，它也隐含地得到了预读取的所有项的锁。真正需要对这些项加锁的其他客户机器可能被阻塞，而这是不必要的。基于这些原因提出了锁逐步降级的技术，即服务器可以要求它的客户将预读取的项上的锁传回来。如果客户机器不需要某个预读取的项，它就可以把该项上的锁传回给服务器，然后服务器就可把该锁分配给其他客户。

• 数据缓冲存储。因一个事务的需要而传送到客户的数据可以缓冲存储在客户端，即使该事务已经完成了，只要存储空间够用，数据仍可以缓冲存储在那里。同一个客户上后续的事务可以使用缓冲存储的数据。然而，那样的话存在一个缓存相关性问题：即就算事务找到了缓冲存储的数据，它还必须确认这些数据是最新的，因为在这些数据被缓冲存储之后可能另一个客户对它们进行了更新。所以，仍然需要与服务器交换一条消息以确认数据的有效性，并且请求对该数据的锁。

• 锁缓冲存储。如果各客户对数据基本上是分割使用的，即一个客户很少需要其他客户也需要的数据，那么锁也可以缓冲存储在客户机器中。假设在某缓冲存储中发现了一个数据项，并且访问该数据项所需要的锁也在该缓冲存储中，那么不需要与服务器通信就可以进行访问了。然而，服务器必须保持缓冲存储的锁的踪迹。如果一个客户向服务器请求某个锁，服务器必须从那些缓冲存储了锁的客户机器收回该数据项上所有冲突的锁。如果考虑到机器故障的可能性，这个任务就变得更加复杂。锁缓冲存储是跨事务的，它与锁逐步降级技术不同，否则，这两项技术就是相似的了。

关于客户-服务器数据库系统，文献注解中提供了更多的信息。

16.3　并行系统

并行系统通过并行地使用多个 CPU 和磁盘来提高处理速度和 I/O 速度。并行计算机正变得越来越普及，相应地并行数据库系统的研究也变得更加重要。有些应用需要查询非常大的数据库（字节数达到 10^{12} 以上），有些应用需要在每秒钟里处理很大数量的事务（每秒数千个事务），这些应用的需求推动了并行数据库系统的发展。集中式和客户－服务器数据库系统的能力不够强大，不足以支持这样的应用。

在并行处理中，许多操作是同时执行的，而不是串行处理的，即不是顺序地执行各个计算步骤。粗粒度并行机只有少数几个能力强大的处理器，大规模并行机或细粒度并行机具有数千个较小的处理器。当今大多数高端机器提供一定程度的粗粒度并行，它们通常具有 2～4 个处理器。大规模并行计算机与粗粒度并行机器的区别在于前者支持的并行程度要高得多。现在市场上可以买到具有数百个 CPU 和磁盘的并行计算机。

对数据库系统的性能有两种主要度量。第一种是*吞吐量*，在给定的时间区间里所能完成的任务的数量。第二种是*响应时间*，单个任务从提交到完成所需的时间。对于处理大量小事务的系统，通过并行地处理多个事务可以提高它的吞吐量。对于处理大事务的系统，通过并行地执行每个事务中的子任务可以缩短它的响应时间，同时提高它的吞吐量。

16.3.1　加速比和扩展性

并行性研究中的两个重要问题是加速比和扩展性。*加速比*指的是通过增加并行度来在更短的时间里运行一个给定的任务。*扩展性*指的是通过增加并行度来处理更大的任务。

考虑在一个具有一定数量的处理器和磁盘的并行系统中运行的数据库应用程序。现在假设我们通过增加处理器、磁盘，以及系统中其他成分的数量来扩大系统的规模。我们的目标是使处理任务所需时间与所分配的处理器和磁盘的数量成反比。假设在较大的机器上执行一个任务的时间是 T_L，在较小的机器上执行同样的任务的时间是 T_S。由于并行性而获得的加速比定义为 T_S/T_L。当较大系统的资源（CPU、磁盘等）是较小系统的资源的 N 倍，且获得的加速比也是 N 时，称并行系统实现了线性的加速比。如果获得的加速比小于 N，则称系统实现了准线性的加速比。图 16-4 描述了线性的和准线性的加速比。

扩展性指的是通过提供更多的资源以在相同的时间里处理更大任务的能力。令 Q 是一个任务，Q_N 是比 Q 大 N 倍的一个任务。假设 Q 在给定的机器 M_S 上的执行时间是 T_S，任务 Q_N 在比 M_S 大 N 倍的并行机器 M_L 上的执行时间是 T_L，于是扩展性定义为 T_S/T_L。如果 $T_L = T_S$，则称并行系统 M_L 对于任务 Q 实现了线性的扩展性。如果 $T_L > T_S$，则称系统实现了准线性的扩展性。图 16-5 描述了线性的和准线性的扩展性。根据度量任务规模的方法不同，并行数据库系统中有两种类型的扩展性。

- *批量型扩展性*。数据库的规模增大，任务是那些其运行时间与数据库的大小密切相关的应用，例如，扫描一个其大小与数据库大小成正比的关系。于是，可以用数据库的大小作为问题大小的度量。批量型扩展性也适合于科学计算应用，例如，执行的查询的精细度提高 N 倍，或进行的模拟的长度增大 N 倍。
- *事务型扩展性*。事务提交率增高，并且数据库的大小也与事务提交率成正比地增长。这类扩展性主要应用在事务都是小更新的那种事务处理系统中，例如银行帐户中的存款和取款，而且，帐户越多，事务也就越多。这些事务特别适合于并行执行，因为不同的事务可以在不同的处理器上并发地、互相独立地运行，每个事务大致耗费同样多的时间，即使数据库的规模增

大了也没有什么影响。

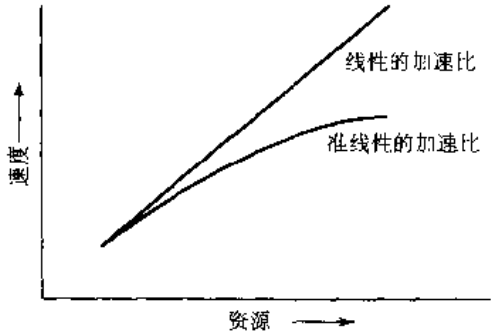


图 16-4　资源增加时的加速比

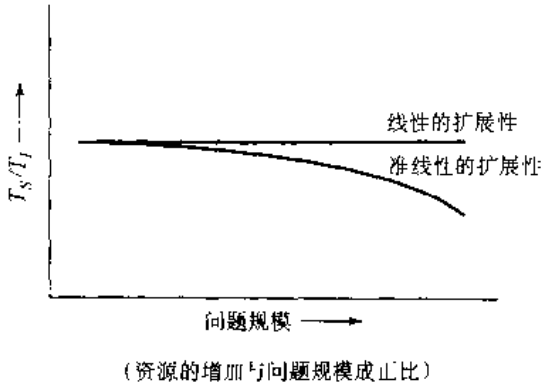


图 16-5　问题规模增大时的扩展性

扩展性通常是并行数据库系统效率的一个更重要的度量。数据库系统中引入并行性的目的通常是为了保证即使在数据库的规模和事务的数量都大大增长时，数据库系统仍能以可接受的速度运行。通过增加并行度来增强系统的能力为企业的增长提供了一条平稳的路径，比在集中式系统中用速度更快的机器（即使有这样的机器存在）替换原来的机器的方法要好。然而，在使用扩展性进行度量时，我们还必须关心绝对性能表现：线性扩展性的机器可能比准线性扩展性的机器执行效果差，原因很简单，后者的启动速度快。

有若干个因素影响并行操作的效率，既降低加速比，又降低扩展性。

• 启动代价。各个进程初始化时需要启动代价。在数以千计的进程构成的并行操作中，启动时间可能掩盖实际的处理时间。对于加速比，这是个不利的影响。

• 干扰。由于在并行系统中执行的进程通常需要访问共享的资源，因此在新的进程与原有进程竞争共享资源，例如系统总线、共享磁盘、甚至锁时，就会发生干扰，使速度下降。这种现象既会影响加速比也会影响扩展性。

• 偏斜。我们通过将一个任务分解为多个并行的步骤来减小各个步骤的大小。然而，最慢的那个步骤的服务时间将决定整个任务的服务时间。通常很难将一个任务分解为大小完全相等的多个部分，因而大小分配常常是有偏斜的。例如，如果将大小为 100 的一个任务分为 10 个部分，并且分解是偏斜的，那么有些任务的大小小于 10，而另一些任务的大小大于 10；即使只有一个任务的大小是 20，那么并行地运行这些任务所得到的加速比就是 5，而不像我们期望的那样是 10。

16.3.2　互连网络

并行系统包括若干个成分（处理器、存储器和磁盘），这些成分之间通过互连网络互相通信。以下是互连网络的一些例子：

• 总线。所有的系统成分通过单一的一个通信总线来发送数据和接收数据。总线可以是以太网，或并行互连。总线结构非常适合于只有少量处理器的情况。然而，当并行度增大时，总线结构就不能很好地适应了，因为总线在同一时间只能处理一个成分的通信要求。

• 网格。将各种成分都安排成网格中的结点，每个结点都和网格中它的所有邻接成分相连接。二维网格中，每个结点都和 4 个邻接结点相连接；三维网格中，每个结点都和 6 个邻接结点相连接。相互间没有直接连接的结点间的通信可以通过将消息由一系列相互间有直接连接的结点来传送的方式进行。当系统成分的数目增大时，通信链的数目也随之增大，因此当并行度

增大时，网格的通信能力能较好地随之增强。

• 超立方体。系统的各个成分按二进制编号，如果某两个成分的二进制编号正好相差 1 位，那么它们之间连接起来。于是，n 个成分中的每一个将与 $log\ (n)$ 个其他成分相连接。可以证明，在超立方体互连中，每一个成分发出的消息至多经由 $log\ (n)$ 个链就可以到达任何一个其他成分。而在网格互连中，一个成分与另一个成分的距离可能是 $2\ (\sqrt{n}-1)$ 个链。（如果网格互连在网格的边缘进行绕接，那么距离可能是 $\sqrt{n}$ 个链。）因此，超立方体互连中的通信延迟显著地低于网格互连。

16.3.3　并行数据库体系结构

并行机器有若干种体系结构模式。图 16-6 所示的是其中最重要的几种。（图中，M 表示主存储器，P 表示处理器，圆柱形表示磁盘）。

• 共享内存。所有的处理器共享一个公共的主存储器。这种模式如图 16-6a 所示。

• 共享磁盘。所有的处理器共享公共的磁盘。这种模式如图 16-6b 所示。共享磁盘系统有时又称做群机。

• 无共享。各处理器既不共享公共的主存储器，也不共享公共的磁盘。这种模式如图 16-6c 所示。

• 层次的。这种模式如图 16-6d 所示。它是前几种体系结构的混合。

我们在下面四部分中对这几种模式进行详细讨论。

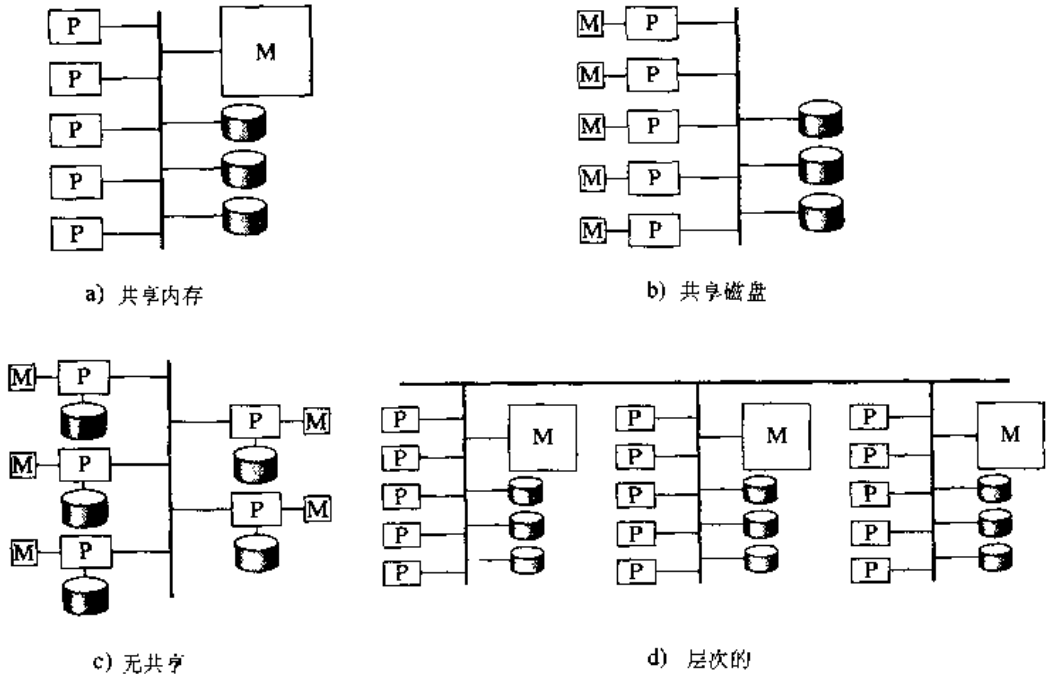


图 16-6　并行数据库体系结构

1. 共享内存

在共享内存体系结构中，所有的处理器和磁盘访问一个公共的主存储器，通常是通过总线或互连网络来进行访问的。共享内存的优点在于处理器之间的通信效率极高，存放在共享主存储器中的数据可以被任何一个处理器访问，而不需要由软件将它们移来移去。一个处理器可以

通过往主存储器中写的办法来向其他处理器传送消息，这通常只需要不到一个微秒的时间，比通过通信机制来传递消息要快得多。共享内存机器的缺点是这种体系结构的规模不能超过32或64个处理器，因为总线或互连网络是由所有的处理器共享的，太多时它们会变成瓶颈。处理器增加至某一点后，再增加更多也没有什么好处，因为各个处理器会将它的大多数时间花在等轮到它通过总线去访问主存储器上。共享内存体系结构通常在每个处理器上有很大的高速缓冲存储器，从而尽量减少对共享内存的访问。然而，不管怎样至少总有一些数据不在高速缓冲存储器中，必须对共享内存进行访问。于是，共享内存机器的规模扩大不能超过某一个点，当前共享内存机器不能支持多于64个处理器的情况。

2. *共享磁盘*

在共享磁盘模式中，所有的处理器都可以通过互联网络直接访问所有的磁盘，但是每个处理器有自己私有的主存储器。共享磁盘体系结构与共享内存体系结构相比有两点好处。第一，由于每个处理器有自己的主存储器，存储器总线就不是瓶颈了。第二，这种体系结构提供了一种较为经济的方法来提供一定程度的容错性：如果一个处理器（或它的主存储器）发生故障，其他处理器可以接替它的工作，因为数据库驻留在磁盘上，而磁盘是所有处理器都可以访问的。我们可以通过使用第10章介绍的RAID体系结构来使磁盘子系统自身是容错的。在许多应用环境中共享磁盘体系结构都是可接受的，按这种体系结构建立的系统通常称做群机。

共享磁盘系统的主要问题也是可扩展性。虽然存储器总线不再是瓶颈了，但与磁盘子系统之间的连接现在却成为了瓶颈，尤其在数据库需要对磁盘进行大量访问的情况下更是这样。与共享内存系统相比，共享磁盘系统中处理器的数目可以更大，但是处理器之间的通信要通过通信网络，这样通信速度就要慢一些（在没有专用的通信硬件的情况下会达到几个毫秒）。

运行Rdb的DEC群机是共享磁盘数据库体系结构的早期商品化应用之一。（Rdb现在为Oracle所有，称做Oracle Rdb。）

3. *无共享*

在无共享的系统中，机器的每一个结点包括一个处理器，一个主存储器，以及一个或多个磁盘。一个结点上的处理器可以通过高速互连网络与另一个结点上的另一个处理器通信。一个结点充当该结点所拥有的磁盘上的数据的服务器。由于局部磁盘访问由各个处理器的本地磁盘提供，所以无共享模式克服了所有的I/O都要通过同一个互连网络的缺点，只有那些访问非本地磁盘的查询及其结果关系需要通过网络传送。而且，无共享系统中的互联网络通常设计成可扩展的，这样当结点数目增加时，传输能力也随之增强。可见，无共享体系结构更具可扩展性，可以很容易地支持大量处理器。无共享系统的主要缺点是通信的代价和非本地磁盘访问的代价，这些代价比共享内存或共享磁盘体系结构中的代价要高，因为数据的传送涉及到两端软件的交互。

Teradata数据库机器是采用无共享数据库体系结构最早期的商品化系统之一。Grace和Gamma研究原型也是建立在无共享体系结构之上的。

4. *层次的*

层次的体系结构综合了共享内存、共享磁盘，和无共享体系结构的特点。在最上层，系统由通过互连网络连接起来的若干个结点组成，它们之间不共享磁盘或主存储器，因此最上层是一个无共享的体系结构。系统的每一个结点实际上可以是一个包括少量处理器的共享内存系统。或者，每一个结点可以是一个共享磁盘系统，而共享一组磁盘系统中的每一个又可以是一个共享主存储器的系统。于是，整个系统可以构造成层次的，其基础是包括少量处理器的共享内存体系结构，其顶层是无共享的体系结构，或许其中还有一个共享磁盘体系结构的中间层。

图 16-6d 描述了一个层次的体系结构，它具有若干个共享内存的结点，通过无共享的体系结构连接在一起。当今有几个运行在这样体系结构上的商品化的并行数据库系统。

为了减少这种系统的程序设计复杂性，提出了分布式虚拟存储器体系结构，在这种体系结构中逻辑上有一个共享的主存储器，而物理上有多个互不相交的主存储器系统。虚拟存储器映象硬件与附加软件共同支持这些互不相交的主存储器的一个单一的虚拟存储区域视图。

16.4　分布式系统

在分布式数据库系统中，数据库存储在几台计算机中。分布式系统中的计算机之间通过高速网络或电话线等各种通信媒介互相通信。这些计算机不共享公共的内存或磁盘。分布式系统中的计算机的规模和功能可大可小，小到工作站，大到大型主机系统。

分布式系统中的计算机有多种不同的称呼，例如节点或结点，根据讲述时的上下文不同而异。我们主要采用节点这个称呼，以强调系统的物理分布。分布式系统的一般结构如图 16-7 所示。

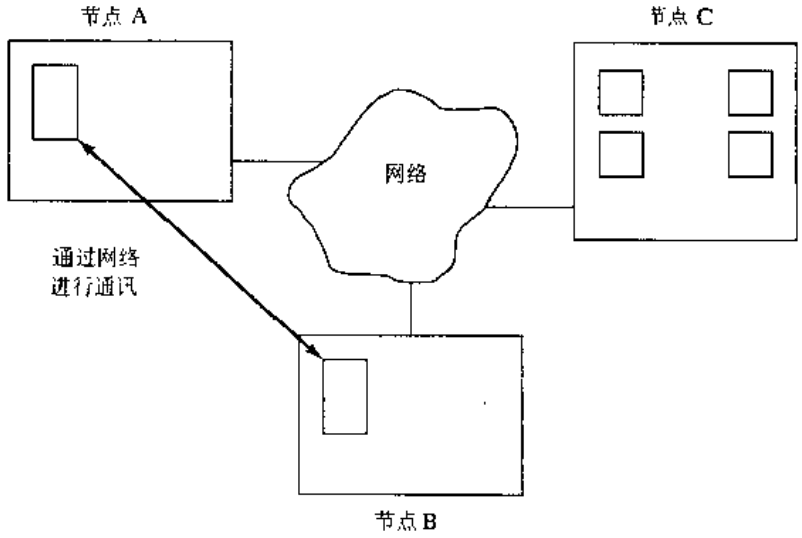


图 16-7　分布式系统

无共享的并行数据库与分布式数据库之间的主要区别在于，分布式数据库一般是地理上分开的，分别管理的，并且是以较低的速度互相连接的。另外，在分布式系统中，我们将事务区分为*局部事务*和*全局事务*。局部事务是仅访问事务被发起的单个节点上的数据的事务，而全局事务或者需要访问发起事务的节点之外的某节点上存放的数据，或者需要访问几个不同节点上的数据。

16.4.1　说明性的例子

考虑一个银行系统，它有 4 个分支机构，位于 4 个不同的城市。每个分支机构有它自己的计算机，计算机中维护该分支机构的所有帐户的一个数据库。每个这样的配置称做一个节点。另外还有一个节点，它维护该银行的所有分支机构的信息。每个分支机构的数据库中有多个关系，其中关系 *account*（*Account-schema*）的关系模型如下：

$$Account\text{-}schema = (branch\text{-}name, account\text{-}number, balance)$$

关于4个分支机构的信息的节点维护的关系 *branch*（*Branch-schema*）如下

$$Branch\text{-}schema = (branch\text{-}name, branch\text{-}city, assets)$$

各个节点上还有一些其他关系，这里我们不再举例。

为说明两类事务的差异，考虑如下事务：增加 $50 到 valleyview 分支机构中的帐户 A-177。如果在 valleyview 分支机构发起该事务，则它是一个局部事务，否则它就是一个全局事务。将 $50 从帐户 A-177 转到 Hillside 分支机构中的帐户 A-305 的事务是一个全局事务，因为事务的执行要访问两个不同节点上的帐户。

如下事实决定了上述配置是一个分布式数据库系统：

- 各个节点互相知道对方的存在。
- 所有节点共享一个公共的全局模式，尽管有些关系可能只存放在其中某些节点上。
- 每个节点提供一个环境，既能执行局部事务，又能执行全局事务。
- 每个节点运行相同的分布式数据库管理软件。

如果不同的节点运行不同的数据库管理软件，那么管理全局事务就很困难。这样的系统称作*多数据库系统*或*异构的分布式数据库系统*。我们将在18.9节中讨论这样的系统，讲述在作为各组成成分的系统异构的情况下，如何达到一定程度的全局控制。

16.4.2　利弊权衡

建立分布式数据库系统的原因包括数据共享、自治性、可用性等。

- 数据共享。建立分布式数据库系统的主要优点是提供一个环境，使得一个节点上的用户可以访问存放在其他节点上的数据。例如，在上节的分布式银行系统例子中，一个分支机构的用户可以访问另一个分支机构的数据。如果没有这种能力，那么想要将资金从一个分支机构转到另一个分支机构的用户就必须求助于某种将已存在的系统互相关联起来的外部机制。
- 自治性。通过数据分布的方法来共享数据，其主要优越性是每个节点可以对局部存储的数据保持一定程度的控制。在集中式系统中，中心节点的数据库管理员对数据库进行控制。在分布式系统中，一个全局数据库管理员对整个系统负责，部分职责委派给每个节点的局部数据库管理员。每个数据库管理员可以有不同程度的局部自治，程度的大小依赖于分布式数据库系统的设计。局部自治的可能性是分布式数据库的一个主要优点。
- 可用性。在分布式系统中，如果一个节点发生故障，其他节点还能继续运行。特别地，如果数据项在几个节点上进行了复制，则需要该特定数据项的事务可以在这几个节点中的任何一个上找到。这样，一个节点的故障不一定意味着整个系统停止运转。

系统必须能检测到一个节点发生了故障，并采取适当的动作使从故障中恢复。系统不能再使用故障节点的服务。最后，当故障节点恢复或修理好了，还需要一定的机制来将它平滑地重新集成到系统中。

虽然分布式系统的故障恢复比集中式系统复杂，但分布式系统中一个节点发生了故障，系统的绝大部分还能继续运行，这一能力使系统的可用性大大增强。对用于实时应用的数据库系统来说，可用性是至关重要的。例如，如果不能对航班数据进行访问，顾客可能就到竞争对手那里去买票了。

分布式数据库系统的主要缺点是由于要保证各节点间的正确合作而增加了复杂性。增加的复杂性表现为以下几种形式：

• 软件开发代价。实现一个分布式数据库系统会更加复杂，因此，代价更高。

• 出现错误的可能性更大。由于构成分布式系统的各个节点并行地运行，因此更难于保证算法的正确性，尤其难于保证当系统的一部分发生故障时的运行，以及从故障中的恢复。其中都会出现很微妙的错误。

• 处理开销增大。消息的交换，以及为了达到节点间的合作而需要的附加的计算，这些都是集中式系统中所没有的开销。

在选择数据库系统的设计时，设计者必须在数据分布的优点和缺点间进行权衡。分布式数据库设计的方法有好几种，可以是完全分布式设计，也可以是有很大程度集中的设计。我们将在第 18 章中对这些方法进行研究。

16.5　网络类型

分布式数据库和客户－服务器系统的建立都以通信网络为基础。有两种基本的网络类型：局域网和广域网。两者的主要区别在于它们的地理分布方式不同。局域网由小的地理范围（例如一个建筑物，或几个相邻的建筑物）内分布的若干个处理器构成，广域网由大的地理范围（例如全美国，或全世界）内分布的若干个自治的处理器构成。这些差别意味着通信网络的速度和可靠性方面的显著差异，并且也反映到分布式系统的设计中。

16.5.1　局域网

局域网（LAN）作为计算机之间互相通信和共享数据的一种手段，出现于 70 年代初期。人们意识到，对于许多企业来说，使用大量的小计算机，每台计算机上有它自含的应用，比使用一台大系统要更经济。由于每一台小计算机都可能需要访问全套的外围设备（例如磁盘和打印机），又由于在一个企业中可能需要某种形式的数据共享，因此很自然地要把这些小系统连接成一个网络。

通常将 LAN 设计成覆盖一个小的地理范围（例如一个建筑物，或几个相邻的建筑物），LAN 一般用于办公室环境中。在这样的环境中，所有节点之间的距离都很近，因此其通信链接的速度比广域网高，且错误率也比广域网低。局域网中最常用的链接是双绞线、基带同轴电缆、宽带同轴电缆、以及光纤。通信速度从每秒 1M 字节（例如 Appletalk 和 IBM 的低速令牌环网络）到每秒 10^9 位（例如试验光纤网络）。最常见的是每秒 10M 位，这是标准以太网的速度。基于光纤的 FDDI 网络和高速以太网的速度是每秒 100M 位。基于一种称做异步传输模式（ATM）的协议的网络能运行得更快，达到每秒 144M 位，这种网络正变得越来越普遍。

一个典型的 LAN 可能包括若干个不同的工作站、一个或多个服务器系统、多种共享的外围设备（例如激光机或磁带设备）、以及支持对其他网络进行访问的一个或多个网关（专用的处理器），如图 16-8 所示。在构造 LAN 时通常采用以太网。

16.5.2　广域网

广域网（WAN）出现在 60 年代后期，开始主要作为一个学术研究项目，提供节点间高效的通信，使得大范围的用户能够方便地和经济地共享硬件和软件。在 60 年代初期开发的能够通过电话线路将远程终端连接到中央计算机的系统不是真正的 WAN。阿帕网是设计和开发的第一个广域网，阿帕网的工作始于 1968 年，现在阿帕网已经从一个包括 4 个节点的试验性网络成长为一个世界范围的网络中的网络——因特网，它包括数以百万计的计算机系统。WAN

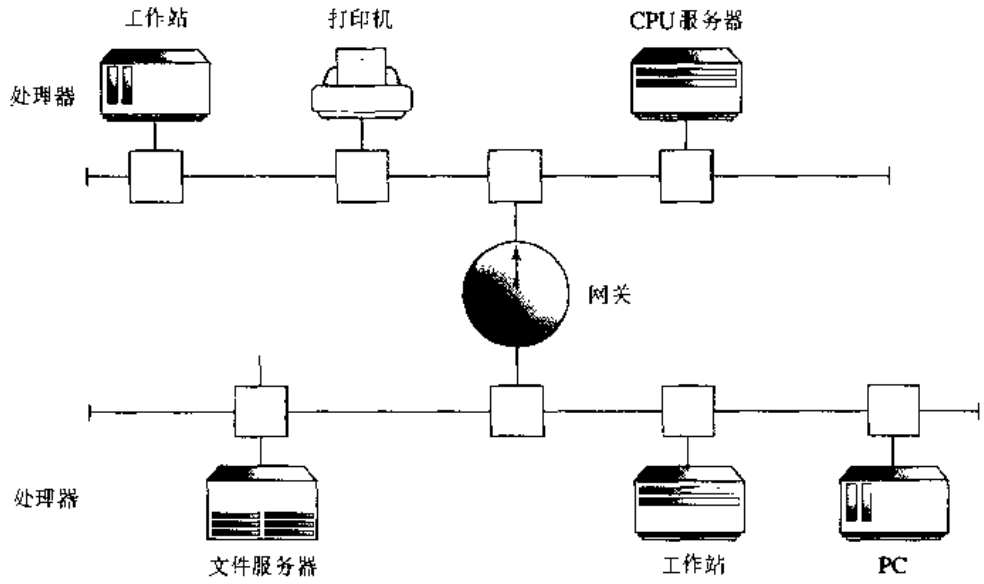


图 16-8　局域网

中典型的连接是电话线路，建立在光纤线上，有时在卫星信道上。

作为一个例子，让我们考虑因特网，它将全球的计算机连接在一起。该系统为地理上分布的节点中的主机提供互相通信的能力。一般说来，这些主机的类型、速度、字长、操作系统等各不相同。通常主机在局域网中，局域网连接到地区网中，地区网通过路由器互相连接，形成世界范围的网络。网络间的连接通常使用一种称作 T1 的电话系统服务，它提供每秒 1.544M 位的传输率；或使用 T3 电话服务系统，它提供每秒 44.736M 位的传输率。网络中传送的消息由称作路由器的系统来确定传送路线，路由器控制每一条消息在网络中走过的路径。这种路线控制可以是动态的，以提高通信效率；也可以是静态的，以减小安全性方面的风险，或者使得通信计费更容易些。

其他的 WAN 使用标准的电话线路作为主要的通信方式。调制解调器从计算机端接收数字化的数据，将这些数据转换为电话系统使用的模拟信号。目的端节点的调制解调器将模拟信号转换回数字信号，于是目的端接收到数据。调制解调器的速度范围从每秒 2400 位到每秒大约 32K 位。在支持综合服务数字网（ISDN）标准的电话系统中数据能够高速地进行点到点传输，一般速度为每秒 128K 位。

在 Unix 网络 UUCP 中，系统之间能够在限定的时间（通常是预先确定的时间）内，通过调制解调器交换消息。然后消息被传送到附近的其他系统中。以这种方式，消息或者被传播到网络中的所有主机（公共消息），或者被传送到它们的目的地（私有消息）。

WAN 可以划分为两种类型：

- 非持续连接的 WAN，例如基于 UUCP 的 WAN，主机只是在部分时间里与网络相连接。
- 持续连接的 WAN，例如因特网，主机在所有时间里都与网络相连接。

分布式数据库系统的设计受作为基础的 WAN 的影响很大。真正的分布式数据库系统只能在持续连接的网络（或至少当分布式数据库运行时处于连接状态的网络）中运行。

非持续连接的网络通常不允许跨节点的事务，但是可以保持远程数据的本地副本，并且周期性地（例如每天的夜间）刷新这些副本。对于那些一致性要求不是很强的应用，例如文档共享系统，Lotus Notes 之类的组件系统，允许在本地对远程数据进行更新，然后再周期性地将更

新传播回到远程节点。有可能在不同节点上发生互相冲突的更新，需要检测这种情况并加以解决。在 21.7.4 节中将描述一种检测互相冲突的更新的机制，然而对于互相冲突的更新的解决方案依赖于不同的应用系统。

16.6　总结

集中式数据库系统完全运行在一台计算机中。最初所有数据库系统都是集中式的。随着个人计算机和局域网的发展，数据库的前端功能不断移向客户机，而后端功能由服务器系统提供。客户－服务器接口协议推动了客户－服务器数据库系统的发展。服务器可以是事务服务器，也可以是数据服务器。

并行数据库系统由通过高速互连网络连接在一起的多台处理器和多个磁盘构成。加速比用来衡量通过并行可以得到的对单个事务的处理速度的增长。扩展性用来衡量通过并行可以得到的处理大量事务的能力。并行数据库体系结构包括共享内存、共享磁盘、无共享，以及层次的体系结构。这些体系结构在可扩展性以及通信速度方面各有千秋。

分布式数据库是局部独立的数据库的集合，它们共享一个公共的模式，并且互相协作处理访问非局部数据的事务。处理器之间通过通信网络相互通信，通信网络负责确定传送路线和连接策略。

大体上有两类通信网络：局域网和广域网。局域网由分布在小地理范围（例如一个建筑物或相邻的几个建筑物）内的若干个处理器构成。广域网由分布在大地理范围内的若干个处理器构成。广域网中的计算机之间可以持续地连接，也可以只是周期性地连接，例如每天连接一次，或每天连接几次。

习题

16.1　如果不需要对单个查询进行并行计算，则将数据库系统从单处理器机器移植到多处理器机器相对比较容易，为什么？

16.2　对于处理短事务的客户－服务器关系数据库，常采用事务服务器体系结构，而对于处理较长事务的面向对象数据库系统，常采用数据服务器体系结构。请给出两条理由，解释为什么数据服务器适合于面向对象数据库而不适合于关系数据库。

16.3　在一般的客户－服务器系统中，服务器机器比客户机的能力要强得多，也就是说，处理器速度更快，可能有多个处理器，有更大的内存和磁盘容量。请考虑另外一种情况，即客户机和服务器机器能力相同的情况。建立这样一个客户－服务器系统有意义吗？哪一种情况更适合于数据服务器体系结构？

16.4　考虑一个基于客户－服务器体系结构的面向对象的数据库系统，其服务器是一个数据服务器。

(a) 客户机和服务器间的互相连接的速度对于选择是进行对象传送还是进行页面传送有什么影响？

(b) 如果采用页面传送，数据在客户端的缓冲存储可以组织成对象缓存，也可以组织成页面缓存。页面缓存以页面为单位存储数据，而对象缓存以对象为单位存储数据，假设对象比页面小，请描述对象缓存比页面缓存优越的一处。

16.5　什么是锁降级？什么情况下需要锁降级？为什么当数据传送的单位是数据项时不需要锁降级？

16.6　假设你负责一个公司的数据库的运行，主要任务是处理事务。假设该公司每年都在迅速

增长，公司当前的计算机系统也该升级了。当你选择一台新的并行计算机时，你最关注下列哪一项：加速比，批量型扩展性，事务型扩展性？为什么？

16.7　假设一个事务是用 SQL 嵌入到 C 中书写的，且大约 80% 的时间花在运行 SQL 代码上，20% 的时间花在运行 C 代码上。如果仅仅对 SQL 代码实施了并行，那么可期望得到多大的加速比？解释你的答案。

16.8　在事务处理系统中，对于线性的加速比起负面影响的有哪些因素？对于下面的每一种体系结构，说明哪个因素是最重要的因素：共享内存，共享磁盘，无共享？

16.9　假设一个银行，它有若干个节点，每个节点运行一个数据库系统，假设这些节点之间唯一的交互是用电子方式相互传送款项，这样的系统称得上是分布式数据库吗？为什么？

16.10　考虑一个基于拨号电话线的网络，节点之间周期性地通信，例如，每天夜里通信。这样的网络通常配置成包括一个服务器节点和多个客户节点。客户节点仅与服务器通信，与其他客户交换数据的方式是将数据存储到服务器中，然后从服务器中检索其他节点存储的数据。与那种要与另一个节点交换数据就必须先拨它的电话号码的体系结构相比，上述体系结构有什么优越性？

文献注解

Patterson 和 Hennessy［1995］、Stone［1993］都是对计算机体系结构做了很好介绍的教科书。

Gray 和 Reuter［1993］是一本优秀的描述事务处理的教科书，包括对客户－服务器体系结构和分布式系统的描述。Geiger［1995］和 Signore 等［1995］介绍了客户－服务器连接的 ODBC 标准。North［1995］介绍了对客户－服务器数据库进行访问的各种工具的使用。

Garey 等［1991］和 Franklin 等［1993］描述了用于客户－服务器数据库系统的数据缓冲存储技术。Biliris 和 Orenstein［1994］对对象存储管理系统进行了综述，包括与客户－服务器结构相关的问题。Franklin 等［1992］以及 Mohan 和 Narang［1994］描述了客户－服务器系统的恢复技术。

Duncan［1990］对并行计算机体系结构进行了综述。Jones 和 Schwarz［1980］对多处理器结构进行了一般性讨论。Satyanarayanan［1980］讨论了多处理器硬件结构。Agrawal 等［1986］和 Bhuyan 等［1989］对多处理器系统的性能进行了讨论。Dubois 和 Thakkar［1992］是一本关于可伸缩的共享内存体系结构的论文集。

DeWitt 和 Gray［1992］对并行数据库系统进行了描述，包括其体系结构和性能度量。Ceri 和 Pelagatti［1984］以及 Bell 和 Grimson［1992］都是关于分布式数据库系统的教科书。关于并行数据库系统和分布式数据库系统的更深一步的参考文献在第 17 章和第 18 章的文献注解中分别列出。

Comer［1995］和 Thomas［1996］描述了计算机网络和互联网。Tanenbaum［1996］和 Halsall［1992］对计算机网络进行了一般性概述。Fortier［1989］对网络硬件和软件进行了详细讨论。Prycker［1993］对 ATM 网络和转换器进行了讨论。Quarterman［1990］描述了网络的总体情况。

第 17 章　并行数据库

本章讨论基于关系数据模型的并行数据库系统中所使用的基本算法。我们特别将重点放在多个磁盘上的数据放置，以及关系操作的并行计算上，它们对于并行数据库系统的成功是非常重要的。

17.1　引言

在过去的 10 年中，并行数据库系统取得了巨大进展，从最初它的一些最坚定的倡导者都几乎要将它彻底否定的地步，发展到今天被 Teradata、Tandem 和 Oracle 等公司成功地推向市场的场景。促使这种转变的是以下趋向：

- 随着计算机的更大量使用，企业组织的事务需求增长了。
- 企业组织正在使用越来越大量的事务处理数据（例如关于人们购买什么商品的数据，或人们什么时间打电话的数据）来策划他们的活动和定价。为这种目的服务的查询称作决策支持查询，而这样的查询所需要的数据量可能高达 10^{12} 字节。单处理器系统没有能力按所需要的速度处理如此大量的数据。
- 商品化的并行数据库系统，例如 Teradata DBC 系列计算机，以及研究性系统，例如 Grace 和 Gamma，已经清楚地展示了并行数据库查询的可行性。事实上，数据库查询的面向集合的特性很自然地将它引向并行处理。
- 随着微处理器价格的日益下降，并行机器已变得很普遍，而且价格并不昂贵。

正如在第 16 章中所讨论的，我们利用并行来提高速度，即由于有了更多的处理器、磁盘等资源而使查询执行得更快。我们还利用并行来提供扩展性，即通过增大并行度来处理更大的工作负载，同时不加长响应时间。

我们在第 16 章中列举了并行数据库系统的几种不同结构：共享内存、共享磁盘、无共享，和层次结构。简单地说，在共享内存结构中，所有的处理器共享一个公共的主存储器和所有的磁盘；在共享磁盘结构中，各处理器有自己独立的主存储器，但共享公共的磁盘；在无共享的结构中，处理器间既不共享主存储器，也不共享磁盘；层次的结构含有多个结点，这些结点间既不共享主存储器，也不共享磁盘，但是每个结点内部是共享内存或共享磁盘的结构。

17.2　I/O 并行

I/O 并行的最简单的形式是指通过在多个磁盘上对关系进行划分，来缩减从磁盘对关系进行检索所需的时间。并行数据库环境中数据划分最通用的形式是水平分布，该分布中将一个关系中的诸元组划分（或分散）到多个磁盘，使每个元组驻留在其中一个磁盘上。另外还提出了好几种划分的策略。

17.2.1　划分技术

我们给出三种基本的划分策略。假定数据分布到 n 个磁盘 D_0，D_1，…，D_{n-1} 上。

- 轮转法。对关系进行任意顺序扫描，将第 i 个元组送到标号为 $D_{i \bmod n}$ 的磁盘上。轮转

模式保证了元组在多个磁盘上的平均分布，即每个磁盘上大致具有相同数目的元组。

• 散列分布。这种策略中，给定的关系模式中的一个或多个属性被指定为划分属性。选定一个值域为 $\{0, 1, \cdots, n-1\}$ 的散列函数。原来关系中的各个元组基于划分属性进行散列。如果散列函数返回 i，则把该元组放到磁盘 D_i 中。

• 范围分布。这种策略往每一个磁盘上分配连续的属性值范围。选定一个划分属性 A，形成划分向量。令 $[V_0, V_1, \cdots, V_{n-2}]$ 表示该划分向量，其满足条件：若 $i < j$，则 $V_i < V_j$。对关系进行如下分配：对元组 t，$t[A] = x$。如果 $x < V_0$，则将 t 放置在磁盘 D_0 中；如果 $X \geqslant V_{n-2}$，则将 t 放置在磁盘 D_{n-1} 中；如果 $V_i \leqslant x < V_{i+1}$，则将 t 放置在磁盘 D_{i+1} 中。

17.2.2　划分技术比较

一旦将一个关系分布到多个磁盘上，就可以使用所有的磁盘来对它进行并行检索。同样，当对一个关系进行了划分时，也可以并行地将它写到多个磁盘上。因此，有 I/O 并行与没有 I/O 并行相比，读/写整个关系的传输速度要快得多。然而，读整个关系，或扫描一个关系，仅是存取数据的一种类型。数据存取可以分类如下：

1）扫描整个关系。

2）按内容找出一个元组（例如，*employee-name* = “Campbell”）。这样的查询称作点查询，它寻找在特定属性上有特定值的元组。

3）找出给定属性的值处于指定范围的所有元组（例如，10000 < salary < 20000），这样的查询称作范围查询。

不同的划分技术支持这些存取类型的效率不同：

• 轮转法。这种模式非常适合于对于每一个查询都希望顺序地读整个关系的那些应用。然而，若采用这种模式，点查询和范围查询的处理都很复杂，因为这种查找 n 个磁盘都要用到。

• 散列分布。这种模式非常适合于基于划分属性的点查询。例如，如果一个关系基于属性 *telephone-number* 进行了分布，那么我们回答“找出 *telephone-number* = 555-3333 的职工记录”的查询时，就可以将用于划分的散列函数应用到 555-3333 上，然后查找该磁盘。将查询引向单个磁盘，比启动多个磁盘要节省启动代价，并且使得其他磁盘可以空闲下来，处理其他查询。

散列分布对于顺序扫描整个关系也很有用。如果散列函数是一个好的随机函数，并且划分属性构成关系的一个码，那么每个磁盘上的元组数目就会大致相同，不会有太大差异。因此，扫描关系所需的时间大致就是关系处于单个磁盘系统情况下所需时间的 $1/n$。

然而，这种模式不适合于非划分属性上的点查询，同时基于散列的分布也不适合范围查询，因为一般说来，散列函数不保持一个范围内的互相贴近，因此，回答范围查询需要扫描所有磁盘。

• 范围分布。这种模式很适合于划分属性上的点查询和范围查询。对于点查询，可以参考划分向量找到元组所在的磁盘。对于范围查询，参考划分向量找出元组可能驻留的磁盘的范围。这两种情况下，我们都将查询范围缩小到了仅包含可能想要找的元组的那些磁盘。

这种特性既是优点，也是缺点。优点在于，如果查询范围内只有少量元组，那么一般说来查询只会送到一个磁盘去，而不是送到所有磁盘。这样可以将其他磁盘用于回答其他查询，因此范围分布可以有较好的响应时间和较高的吞吐量。与此相反，如果查询范围内有许多元组（当查询范围占关系域的较大份额时会是这样），那么就会有许多元组从少量磁盘中检索出来，从而导致这些磁盘上的 I/O 瓶颈（热点）。在这种执行偏斜的情况中，所有的处理发生在一个

或少量的几个分片中。与此相对照的是，对于这样的查询，散列分布和轮转法分布会用到所有磁盘，其结果是几乎同样的吞吐量，但却有较快的响应时间。

划分的类型还影响其他关系操作，例如连接，我们将在 17.5 节中讨论。因此，划分技术的选择也依赖了需要执行的操作。一般说来，更倾向于采用散列分布和范围分布，而不是轮转法分布。

在一个拥有许多磁盘的系统中，可以用如下的方法来确定要往多少个磁盘上分布一个关系。如果一个关系仅包含少量元组，在一个磁盘块中就能放下，那么最好将该关系放到单个磁盘中。大的关系最好分布到所有的可用磁盘中。如果一个关系包括 m 个磁盘块，而系统中有 n 个磁盘可用，那么应该分配给这个关系 min（m，n）个磁盘。

17.2.3　偏斜的处理

当把一个关系中的元组分布到多个磁盘时，元组的分布可能偏斜（轮转法不会），即某些磁盘中元组较多，而另一些磁盘中元组较少。偏斜的表现形式分类如下：

- 属性值偏斜。
- 划分偏斜。

*属性值偏斜*指的是某些值出现在许多元组的划分属性中，划分属性上具有相同值的所有元组落入同一分片中，这就导致了偏斜。*划分偏斜*指的是即使在不存在属性值偏斜的情况下，也可能出现的划分中的负载不平衡。

不管采用范围分布，还是采用散列分布，属性值偏斜都会导致分布的偏斜。如果没有选择好划分向量，那么范围分布会发生划分偏斜。如果散列函数选得好，那么散列分布不太可能发生划分偏斜。

我们在第 16.3.1 节中已经谈到，即使一个很小的偏斜也可能导致性能的显著降低。随着并行度的提高，偏斜成了更加严重的问题。例如，如果将一个包含 1000 个元组的关系分成 10 个部分，并且分布发生了偏斜，那么有些分片的元组数就会大于 100，而另外一些分片的元组数会小于 100，即使只有一个分片的元组数达到 200，那么并行地存取这些分片所能得到的加速比只能是 5，而不是所期望的 10。还是这个关系，如果将它分成 100 个部分，平均每个分片 10 个元组。如果有一个分片的元组数达到 40（这是有可能的，因为分片的数目很大），那么并行地存取这些分片所能得到的加速比就只能是 25，而不是 100。这样，我们看到随着并行度的增大，由于偏斜而导致的加速比的损失也增大。

如果划分属性构成关系的一个码，那么通过排序可以构造一个好的范围划分向量，如下所述。首先将关系在划分属性上排序，然后按排好的顺序扫描该关系，每读过该关系的 $1/n$，其中 n 表示要构造的分片的数目，就将下一个元组的划分属性值加入到划分向量中。这种方法的缺点在于进行初始排序需要额外的 I/O 开销。

17.3　查询间并行

*查询间并行*指的是不同的查询或不同的事务互相并行地执行，这种形式的并行可以提高事务吞吐量。然而，单个事务的响应时间不会比事务孤立运行时快。因此，查询间并行的主要用处是扩展事务处理系统，使它在每秒种里能支持更大数量的事务。

查询间并行是数据库系统中最易被支持的一种并行形式，在共享内存的并行系统中尤其如此。为单处理器系统设计的数据库系统可以不做改动，或者做很少的改动，就能用到共享内存的并行结构中，因为即使在串行的数据库系统中也支持并行处理。在串行机器中以分时并发方

式运行的多个事务在共享内存的并行结构中将以并行的方式运行。

在共享磁盘或无共享的体系结构中支持查询间并行要复杂些。各处理器必须以一种协同的方式来执行某些任务，例如封锁和日志，而这就需要它们互相之间能传送消息。并行数据库系统还必须保证两个处理器不会相互独立地同时更新相同的数据。而且，当一个处理器访问或更新数据时，数据库系统必须保证该处理器在它的缓冲池中也拥有该数据的最新版本。这个问题称作缓存相关问题。

已经制定了多种协议来保证缓存相关，通常将缓存相关协议与并发控制协议集成在一起，以减小它们的开销。用于共享磁盘系统中的一个协议如下：

1）一个事务对一个页面进行任何读写访问之前，先根据需要用共享或排它方式封锁该页面。当事务获得了该页面的共享锁或排他锁，则立刻从共享磁盘中读到该页面的最新版本。

2）在事务释放一个页面的排他锁之前，它将页面刷新到共享磁盘中，然后再释放封锁。

这个协议保证，当一个事务对某页面设置了共享锁或排他锁时，它得到该页面的正确版本。

为避免上述协议所需要的对磁盘的反复读写，我们又制定了更复杂的协议。基于这个协议，当释放共享锁时并不将页面写回到磁盘。当事务获得共享锁或排他锁时，如果页面的最新版本已在某个处理器的缓冲池中，事务就从该缓冲池中获得该页面。这种协议的设计必须考虑到对于并发请求的处理。对于共享磁盘体系结构的协议进行如下扩充，使其能适用于无共享的体系结构：每一个页面有一个主处理器 P_i，并且这些页面存储在磁盘 D_i 中。当其他处理器想对该页面进行读写时，因为它们不能与页面所在的磁盘直接通信，它们就将请求发给页面的主处理器 P_i。其他动作与共享磁盘体系结构中的协议一样。

Oracle 7 和 Oracle Rdb 系统是支持查询间并行的共享磁盘数据库系统的实例。

17.4　查询内并行

*查询内并行*指的是单个查询在多个处理器和磁盘上并行执行。若要加快运行时间长的查询的速度，采用查询内并行非常重要。查询间并行对此无能为力，因为每一个查询都是串行地运行。

为了说明查询的并行计算，我们来看一个要求对某关系进行排序的查询。假设该关系已经基于某个属性进行范围分布，且放置到多个磁盘上，如果要进行的排序是基于划分属性的，那么排序运算可以如下进行：对每个分片并行地进行排序，然后将排好序的各个分片串接在一起，从而得到最终排好序的关系。因此，可以通过并行地执行各个操作来使一个查询并行化。还有一个对单个查询进行并行计算的途径：一个查询的操作树中可能包含多个操作，可以对其中某些互不依赖的操作并行地执行，从而使操作树的计算并行化。而且，正如第 12 章中所提到的，可以采用流水线方式，将一个操作的输出传送给另一个操作，这两个操作可以在各自的处理器上并行地执行，一个操作产生的输出，马上被另一个操作使用。

综上所述，单个查询的执行可以有两种并行方式：

- *操作内并行*。可以通过并行地执行每一个操作，如排序、选择、投影、连接等，来加快一个查询的处理速度。我们将在下节讨论操作内并行。
- *操作间并行*。可以通过并行地执行一个查询表达式中多个不同的操作，来加快一个查询的处理速度。我们将在第 17.6 节讨论这种形式的并行。

并行的这两种形式是互相补充的，并可以同时应用在一个查询中。通常由于一个查询中操作的数目与每一个操作中所处理的元组数目相比是很小的，所以当并行度增大时，第一种形式的并行效果更佳。然而，由于当前一般并行系统中处理器的数目都不大，所以两种形式的并行

都很重要。

在下面关于查询的并行处理的讨论中，假设查询是只读的。并行查询计算的算法选择依赖于机器的结构。在下面的描述中，采用无共享的体系结构模式，并不对每一种体系结构都分别给出算法。因此，我们显式地给出什么时候需要将数据从一个处理器传送给另一个处理器。采用其他体系结构，可以很容易地模拟这种模式，因为在共享内存的体系结构中，数据的传送可以通过共享内存来进行；在共享磁盘的体系结构中，数据的传送可以通过共享磁盘来进行。因此，为无共享的体系结构设计的算法也可以应用到其他体系结构中。到时我们还会提到，在共享内存或共享磁盘的系统中如何进一步对算法进行优化。

为了简化算法的表示，我们假定有 n 个处理器 P_0，…，P_{n-1}和 n 个磁盘 D_0，…，D_{n-1}，其中磁盘 D_i 是与处理器 P_i 相关的。实际的系统中每个处理器可以有多个磁盘。将算法扩展到允许每个处理器有多个磁盘并不困难，例如干脆允许 D_i 是一组磁盘不就行了。然而，为简单表示起见，我们在这里假定 D_i 是单个磁盘。

17.5　操作内并行

因为关系操作在包含大量元组的关系上进行，所以可以通过在关系的不同子集上并行地执行来实现操作的并行化。由于一个关系中元组的数目可以很大，所以并行度也有可能达到很大。因此，在数据库系统中，操作内并行是很自然的。在 17.5.1～17.5.3 节中，我们将研究一些常用的关系操作的并行版本。

17.5.1　并行排序

假设希望对存放在 n 个磁盘 D_0，D_1，…，D_{n-1}中的一个关系进行排序。如果该关系进行的是范围分布，且划分属性正是排序要基于的属性，那么，正如 17.2.2 节所谈到的，可以对每一个分片分别进行排序，然后把各个排序结果串接起来，从而得到完全排好序的关系。由于元组分布在 n 个磁盘中，因此并行的存取减少了读取整个关系所需的时间。

如果关系是按其他某种方式分布的，那么可以采用下面的方法之一来进行排序：

1）按排序属性对它进行范围分布，然后分别排序各个分片。

2）采用外部排序－合并算法的并行版本。

1. 基于范围分布的排序

当采用对关系进行范围分布的方法进行排序时，没有必要将关系仍然分布到它所存放的那些处理器或磁盘上。假设我们确定用处理器 P_0，…，P_m（其中 $m<n$）来对关系进行排序。排序操作中涉及两个步骤：

1）采用范围分布策略对关系进行重新分布，使得处于第 i 个范围的所有元组都送给处理器 P_i，它将关系临时地存放在磁盘 D_i 中。这一步骤需要磁盘 I/O 和通信开销。

2）每个处理器对送给它的关系的分片进行排序，这一排序在本地进行，不和其他处理器交互。每个处理器对于不同的数据集执行相同的操作——排序。（在不同的数据集上并行地执行相同的操作称为数据并行）。

最后的合并操作非常简单，因为一开始的范围分布就保证了对于 $1\leqslant i<j\leqslant m$，处理器 P_i 中的所有关键码值都小于处理器 P_j 中的所有关键码值。

正如在 17.2.3 节中所提到的，我们必须使用一个负载均衡的划分向量来进行范围分布，这样每个分片中的元组数目就会大致相同。

2. 并行的外部排序－合并

除了范围分布外，另外一种方法是采用并行的外部排序 - 合并。假设关系已经分布到磁盘 D_0，…，D_{n-1}中，（分布的方法是哪一种无关紧要）。并行的外部排序 - 合并按如下方式进行：

1）每一个处理器 P_i 在它的本地对磁盘 D_i 中的数据进行排序。

2）然后对每个处理器中排好序的文件进行合并，得到最终排好序的输出。

第 2）步中对排好序的文件的合并可以并行执行如下：

a. 每个处理器 P_i 中的排好序的分片使用相同的划分向量，用范围分布的办法分布到处理器 P_0，…，P_{m-1}中。元组按照排好的序列发送，所以每个处理器接收的都是排好序的元组流。

b. 每个处理器 P_i 在收到元组流时进行合并，得到单个的排好序的文件。

c. 将处理器 P_0，…，P_{m-1}中排好序的文件串接起来，得到最终的结果。

有些机器，例如 Teradata DBC 系列机器，采用专门的硬件来执行合并。Teradata DBC 机器中的 Y-net 互连网络可以合并来自多个处理器的输出，得到单个的排好序的输出。

17.5.2　并行连接

连接操作需要对一个个元组对进行检测，看它们是否满足连接条件，如果满足，就把这个元组对加到连接的输出中。并行连接算法设法将这些要检测的元组对分到多个处理器中，然后每个处理器在本地进行连接操作的一部分。最后，我们从各个处理器收集计算结果，以产生最终的结果。

1. 基于划分的连接

对于某些类型的连接，例如等值连接和自然连接，可以将两个输入关系分布到多个处理器中，然后在每个处理器做本地连接。假定使用 n 个处理器，要连接的关系是 r 和 s，则将每个关系都划分成 n 个部分，分别记为 r_0，r_1，…，r_{n-1}和 s_0，s_1，…，s_{n-1}。将分片 r_i 和 s_i 送到处理器 P_i，并在那儿进行本地连接。

上述技术只有在等值连接（例如，$r \bowtie_{r.A=s.B} s$）且我们在连接属性上使用相同的划分函数来划分 r 和 s 的情况下才能正确工作。划分思想与散列连接划分步骤的思想完全一样。然而，有两种不同的方法来划分 r 和 s：

- 基于连接属性进行范围分布。
- 基于连接属性进行散列分布。

不管采用上述哪一种划分，都必须在两个关系上使用相同的划分函数。对于范围分布，两个关系所用的划分向量必须相同；对于散列分布，两个关系所用的散列函数必须相同。图 17-1 描述了基于划分的并行连接中所采用的划分方法。

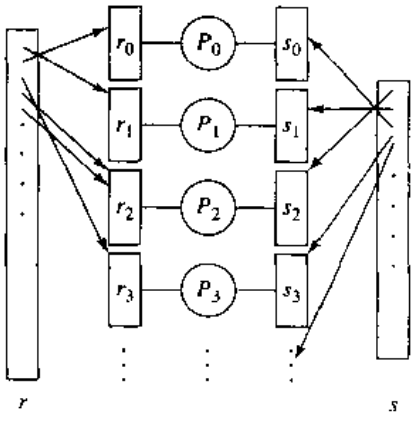


图 17-1　基于划分的并行连接

一旦对关系进行了划分，就可以在每个处理器 P_i 中采用任何一种连接技术来本地地计算 r_i 和 s_i 的连接。例如，采用散列连接、合并连接、或嵌套循环连接。这样，我们可以采用划分的方法使任何连接技术并行化。

如果关系 r 和 s 中的一个，或两个都已经基于连接属性进行了划分（通过散列分布或范围分布），那么划分所需做的工作就大大减少了。如果关系没有进行划分，或者

不是基于连接属性进行的划分，那么就需要对元组重新进行划分。每个处理器 P_i 读入磁盘 D_i 中的元组，对于每一个元组 t，计算出它所应该属于的分片 j，并把元组 t 发送给处理器 P_j，处理器 P_j 将元组存放到磁盘 D_j。

每个处理器中，如果将某些元组不是写到磁盘中，而是放在内存的缓冲区中，则可以减少 I/O，从而优化本地使用的连接算法。我们将在第 3 部分中讨论这样的优化。

当采用范围分布时，偏斜表现为一个特殊的问题，因为将连接中的一个关系划分为相等大小的分片的划分向量可能将另一个关系划分为大小差别很大的分片。划分向量应该使 $|r_i|+|s_i|$（即 r_i 和 s_i 的大小之和）对于所有的 $i=0, 1, \cdots, n-1$ 大致相等。如果散列函数选得好，则散列分布产生的偏斜较小，除非有许多元组在连接属性上具有相同的值。

2. 分片 - 复制连接

基于划分的连接方法并不适合于所有类型的连接。例如，如果连接条件不是相等，像 $r \bowtie_{r.a<s.b} s$，那么可能 r 中的所有元组与 s 中的某些元组相连接（反之亦然）。于是，可能找不到可行的划分 r 和 s 的方法，使得分片 r_i 中的元组只与分片 s_i 中的元组相连接。

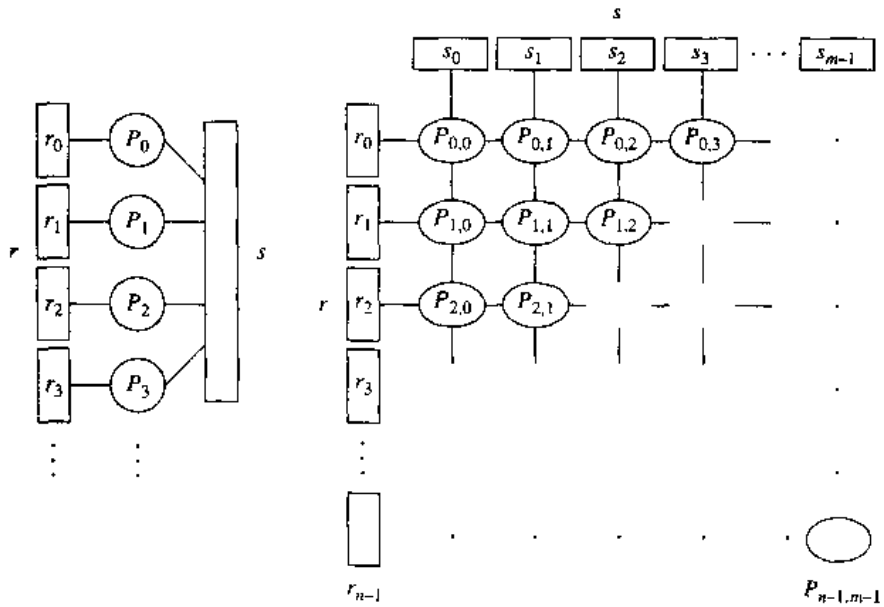
我们可以采用一种称为分片 - 复制的技术来并行地执行这样的连接。首先考虑分片 - 复制的一种特殊情况：非对称的分片-复制，方法如下：

1）对其中的一个关系，例如 r，进行划分。对 r 可以采用任何一种划分技术，包括轮转法分布。

2）将另一个关系 s 复制到所有处理器。

3）处理器 P_i 采用任何一种连接技术本地地进行 r_i 和整个 s 的连接。

非对称的分片 - 复制模式如图 17-2a 所示。如果 r 已经是经过划分存储的，则不必执行第 1）步划分，只要将 s 复制到所有的处理器就可以了。



a) 非对称的分片和复制　　b) 分片和复制

图 17-2　分片 - 复制模式

分片 - 复制的一般情况如图 17-2b 所示，方法如下：将关系 r 划分为 n 片 $r_0, r_1, \cdots, r_{n-1}$，将关系 s 划分为 m 片 $s_0, s_1, \cdots, s_{m-1}$，同前面一样，对 r 和 s 采用哪种划分技术都行。m 和 n 的值不必相等，但必须保证至少有 $m * n$ 个处理器。非对称的分片 - 复制是一般的分片 - 复制的一个特殊情况，即 $m=1$。与非对称的分片 - 复制相比，分片 - 复制减小了每个处理器中

关系的大小。

让我们将处理器标记为 $P_{0,0}$，$P_{0,1}$，…，$P_{0,m-1}$，$P_{1,0}$，…，$P_{n-1,m-1}$。处理器 $P_{i,j}$ 计算 r_i 和 s_j 的连接。为了能做到这一点，我们将 r_i 复制到处理器 $P_{i,0}$，$P_{i,1}$，…，$P_{i,m-1}$（这组成了图 17-2b 中的一个行），将 s_i 复制到处理器 $P_{0,i}$，$P_{1,i}$，…，$P_{n-1,i}$（这组成了图 17-2b 中的一个列）。在各个处理器 $P_{i,j}$ 上可以采用任何一种连接技术。

对于任何连接条件，都可以采用分片 - 复制方法，因为 r 中的每一个元组都能与 s 中的每一个元组一同进行检测。所以，在不能采用基于划分的连接的情况下可以采用分片 - 复制连接。

当两个关系的大小基本相同时，分片 - 复制连接的代价比基于划分的连接的代价要高，因为至少其中的一个关系要被复制。然而，如果其中的一个关系如 s 很小，那么将 s 复制到所有处理器的代价可能比基于连接属性对 r 和 s 进行划分的代价要小。在这种情况下，即使能够采用基于划分的连接，也最好采用非对称的分片 - 复制连接。

3. 基于划分的并行散列连接

本节说明如何并行地执行 12.6 节中所讲的基于划分的散列连接。

假设有 n 个处理器 P_0，P_1，…，P_{n-1}，有两个关系 r 和 s，r 和 s 分布到多个磁盘中。在 12.6 节中我们讲到，选用较小的关系作为建立散列表的关系。如果 s 比 r 小，那么并行的散列连接算法按如下步骤进行：

1）选择一个散列函数，例如 h_1，它用 r 和 s 中每个元组的连接属性的值作为自变量，将元组映象到 n 个处理器中的一个。令 r_i 表示关系 r 映象到处理器 P_i 中的元组；同样，令 s_i 表示关系 s 映象到处理器 P_i 中的元组。每个处理器 P_i 读它的磁盘 D_i 中 s 的元组，并基于散列函数 h_1 将每一个元组送到适当的处理器中去。

2）当 s_i 中的元组到达目的地处理器 P_i 时，用另一个散列函数 h_2 对它们进一步划分，h_2 是该处理器用来进行本地散列连接的。这一步的划分与串行的散列连接算法的划分阶段完全一样。每一个处理器在执行这个步骤时与其他处理器完全独立。

3）将 s 的元组进行了分布后，系统采用与前面相同的方法，用散列函数 h_1 将较大的关系 r 在 n 个处理器上进行重新分布。目的地处理器在接收到每一个元组时，用散列函数 h_2 对它进行重新划分，就像串行散列连接算法中对探查关系进行划分一样。

4）每个处理器 P_i 对 r 和 s 的本地分片 r_i 和 s_i 执行散列连接算法的建表阶段和探查阶段，产生散列连接的最终结果的一个分片。

各个不同的处理器中执行的散列连接相互独立，接收 r_i 和 s_i 中的元组类似于从磁盘中读取元组。因此，第 12 章中描述的散列连接的任何一种优化都可以应用到并行的情况。特别地，可以采用混合式散列连接算法，将输入的一些元组缓冲存储在内存中，从而避免将它们写出再读入的代价。

4. 并行的嵌套循环连接

为了对基于分片 - 复制方法的并行的使用进行描述，考虑关系 s 比关系 r 小得多的情况，同时还假设关系 r 是分片存储的，至于它是基于哪个属性进行划分的则没有什么关系。最后，我们还假设在关系 r 的每一个分片上存在连接属性上的一个索引。

我们采用非对称的分片 - 复制方法，将关系 s 进行复制，关系 r 则利用它现有的分片。每一个存放了关系 s 的一个分片的处理器 P_j 读取存放在 D_j 上的关系 s 中的元组，将这些元组复制到其余每一个处理器 P_i 中。在这个阶段的最后，关系 s 复制到了存放有关系 r 的元组的每一个节点中。

现在，每个处理器 P_i 对关系 r 的第 i 个分片做利用索引的嵌套循环连接。可以将利用索引的嵌套循环连接与关系 s 的元组的分布重叠起来做，从而减少将 s 的元组写到磁盘再读回来的代价。然而，关系的复制必须与连接同步，从而在每个处理器 P_i 的内存缓冲区中能有足够的空间来存放已经接收到的，但尚未用于连接的关系 s 中的元组。

17.5.3　其他关系操作

我们也可以将其他关系操作的执行并行化：

• 选择。令选择为 $\sigma_\theta(r)$，首先考虑 θ 形为 $a_i = v$ 的情况，其中 a_i 是一个属性，v 是一个值。如果关系 s 是基于属性 a_i 进行划分的，那么选择在单个处理器中执行。如果 θ 形为 $l \leqslant a_i \leqslant u$，即 θ 是一个范围选择，而且该关系基于 a_i 进行了范围分布，那么选择在其分片与指定的值的范围相重叠的每一个处理器中执行。其他所有情况下，选择在所有的处理器中并行地执行。

• 消除重复。消除重复可以通过排序来进行，哪一种并行排序技术都行，在排序过程中一旦发现重复就消除掉。我们还可以将元组进行分布（采用范围分布或散列分布），然后在每个处理器中本地地进行消除重复，从而使消除重复并行化。

• 投影。没有进行消除重复的投影可以在元组并行地从磁盘读入时进行。如果要消除重复，那么可以采用上述技术。

• 聚集。可以通过将关系基于分组属性进行划分后在每个处理器中本地地计算聚集的值，来使操作并行化。采用散列分布或者范围分布都可以。如果关系已经基于分组属性进行了分布，那么可以跳过第一步。

我们可以在对元组进行分布前部分地计算聚集的值，从而减少分布时元组传输的代价，对于常用的聚集函数至少可以这样做。考虑关系 r 上的一个聚集操作，它基于属性 A 分组，且在属性 B 上使用聚集函数 sum。每个处理器 P_i 可以对存储在磁盘 D_i 中的 r 元组执行该操作。其结果是每个处理器中的部分和，对于磁盘 D_i 中存放的每一个 r 元组中属性 A 的值，在 D_i 上都有一个结果元组。局部聚集的结果基于分组属性 A 进行分布，然后在每个处理器 P_i 上再次进行聚集运算（对有部分和的元组进行），从而得到最终结果。

这样的优化减少了分布过程中需要送到其他处理器的元组数目。可以很容易地将这一想法扩充到 min 和 max 聚集函数，练习 17.8 要求你扩充到 count 和 avg 聚集函数。

练习中还包括了其他操作的并行化。

17.5.4　操作并行计算的代价

我们通过将 I/O 分布到多个磁盘上，将 CPU 工作分布到多个处理器上来达到并行。如果这样的划分不需要开销就能达到，且在工作负载的划分中没有偏斜，那么使用 n 个处理器的并行操作所用的时间就是单个处理器执行相同操作所用时间的 $1/n$。我们已经知道如何估算连接、选择等操作的代价。这样，并行处理的时间代价就是串行处理相同操作的时间代价的 $1/n$。

我们还必须将以下代价计算在内：

• 在多个处理器中启动该操作的启动代价。

• 在多个处理器中分布工作负载的偏斜，有些处理器得到的元组多，有些处理器得到的元组少。

• 对资源（内存、磁盘、通信网络等）的竞争所导致的延迟。

• 从每个处理器传送部分结果以组成最终结果所花费的代价。

并行操作所需的时间可以估算为

$$T_{part} + T_{asm} + \max\ (T_0,\ T_1,\ \cdots,\ T_{n-1})$$

其中 T_{part} 是对关系进行分布所需的时间，T_{asm} 是组成最终结果所需的时间，T_i 是处理器 P_i 执行操作所需的时间。假设元组的分布没有任何偏斜，那么送到每个处理器的元组数可以估算为总的元组数的 $1/n$。再忽略掉竞争，于是每个处理器 P_i 执行操作的代价 T_i 就可以通过第 12 章所描述的技术估算出来。

上面的估算是一个乐观的估算，因为通常总是有偏斜的。即使将一个查询分解成若干个并行的步骤减少了每个步骤的平均大小，但处理整个查询所需的时间是由其中最慢的那个步骤所需的时间来决定的。例如，基于划分的并行计算的速度仅与并行执行中最慢的一个速度相同。因此，在处理器间分布工作负载时的任何偏斜都会对性能造成很大的影响。

分布中的偏斜问题与串行散列连接中的分布溢出问题（第 12 章）密切相关，当进行散列分布时，可以采用散列连接所用的解决溢出和避免溢出的技术来处理偏斜。

采用范围分布的连接中处理偏斜的一种技术是为每一个关系的每一个属性的属性值构造并保存一个频率表，或直方图。图 17-3 给出了一个直方图的例子，这是一个以整数为值的属性，取值范围在 1～25 之间。然后我们再利用该关系的连接属性上的直方图构造一个负载均衡的范围分布函数，从而使得元组的分布更加均匀。我们已经提出了一些减少 I/O 开销的构造直方图的方法，这些方法基于对关系进行抽样，只使用关系的磁盘块随机选取的子集。

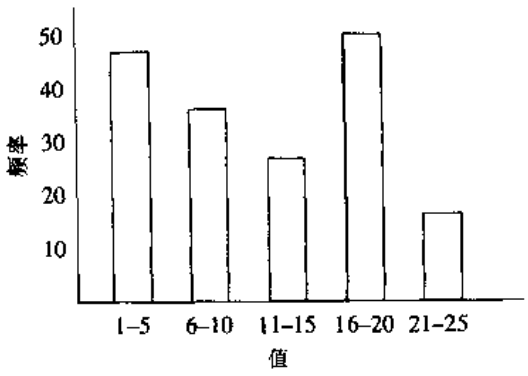


图 17-3　直方图的例子

17.6　操作间并行

操作间并行有两种形式：流水线并行，独立的并行。

17.6.1　流水线并行

正如我们在第 12 章中所讨论的，流水线能有效地减少数据库查询处理的计算代价。在流水线中，一个操作 A 的输出元组作为第二个操作 B 的输入，在第一个操作尚未产生完全的输出元组集合之前，第二个操作就可以在它的输入上进行工作。在串行计算中采用流水线执行的主要优点是不必往磁盘中写任何中间结果，就可以将一系列的操作进行下去。

在并行系统中，采用流水线执行的主要原因与串行系统相同。但同时，流水线也可以提供并行性，就像硬件设计中采用指令的流水线以提供并行性一样。在前面的例子中，我们可以在不同的处理器上同时运行 A 和 B，在 A 产生结果元组的同时，B 来使用它们。这种形式的并行称作流水线并行。

让我们考虑四个关系的连接：

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

可以建立一个流水线，让三个连接并行地执行。指定处理器 P_1 去计算 $temp_1 \leftarrow r_1 \bowtie r_2$，处理器 P_2 计算 $r_3 \bowtie temp_1$。当 P_1 计算出 $r_1 \bowtie r_2$ 中的元组时，它就将这些元组提供给处理器 P_2。于是在 P_1 完成它的计算之前，P_2 就得到了 $r_1 \bowtie r_2$ 中的一些元组。P_2 利用它得到的元组开始计算 $r_3 \bowtie temp_1$，尽管这时 P_1 还没有计算出完整的 $r_1 \bowtie r_2$。与此类似，当 P_2 计算出 $(r_1 \bowtie r_2) \bowtie r_3$ 中的元组时，它就将这些元组提供给 P_3，由 P_3 来计算这些元组与 r_4 的连接。

当只有少量的处理器时，流水线很适合，但它的可扩展性不好。首先，流水线链达不到足够的长度来提供较高的并行度。其次，对于那些必须访问了整个输入才能开始产生输出的操作（例如集合的差运算），就不能采用流水线。第三，对于常见的一个操作的代价比其他操作的代价高得多的情况，流水线方法只能获得很小的速度提高。因此，当并行度较高时，流水线对于提高并行度的重要性不如基于划分的并行。采用流水线方法最重要的原因是流水线执行可以避免将中间结果写到磁盘中。

17.6.2 独立的并行

查询表达式中那些相互之间没有依赖关系的操作可以并行地执行。这种并行形式称作独立的并行。

让我们再来考虑连接 $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$。显然，我们可以并行地计算 $temp_1 \leftarrow r_1 \bowtie r_2$ 和 $temp_2 \leftarrow r_3 \bowtie r_4$。当这两个计算完成后，我们计算 $temp_1 \bowtie temp_2$。为进一步提高并行度，我们可以通过将 $temp_1$ 和 $temp_2$ 中的元组以流水线方式送给连接来计算 $temp_1 \bowtie temp_2$（见 12.8.2 节）。

独立的并行与流水线并行一样不能提供较高的并行度，在高度并行的系统中没有太大用处，尽管在低度并行的系统中它是有用的。

17.6.3 查询优化

关系技术之所以能够取得成功，一个重要的因素是成功地设计了查询优化器。查询优化器分析一个查询，在给出相同结果的多个可能的执行计划中找出代价最低的执行计划。

并行查询计算的优化器比串行查询计算的优化器更为复杂。首先，代价模型更复杂，因为不仅必须将划分的代价考虑在内，而且还要考虑偏斜、资源竞争等代价。其次，更重要的是如何并行地计算一个查询的问题。假设我们以某种方式从一个查询的多个等价的表达式中选出了一个来计算该查询。在 12.8 节中我们已经讨论过，可以用操作树来表示这个表达式。

为了在并行系统中计算一个操作树，必须做出以下决定：

- 如何并行地计算每一个操作，用多少个处理器来计算。
- 哪些操作需要在多个处理器上流水线地执行，哪些操作需要独立地并行执行，哪些操作需要串行地，一个跟在一个后面地执行。

这些决定构成执行树的调度任务。

优化问题的另一个方面是确定应该分配给每一个运算的各种类型资源，如处理器、磁盘和内存等。例如，看起来可能最好是最大限度地利用并行资源，但另一个好的想法却是某些操作不要并行地执行。那些计算需求比通信开销小得多的操作应该和邻近的操作聚集在一起，否则并行的好处就被通信的开销掩盖了。

采用流水线方法时需要考虑的一个问题是较长的流水线不能很好地利用资源，除非操作是粗粒度的，否则流水线中最后一个操作可能要等待很长时间才能得到输入，然而它却占据着宝

贵的资源，例如内存。因此应该避免较长的流水线。

候选并行执行计划的数目比串行执行计划的数目要大得多。因此，通过考虑所有的候选计划来优化并行查询的代价比优化串行查询的代价要高得多。所以，我们通常采用启发式方法来减少要考虑的并行执行计划的数目。在这里我们描述两种常用的启发式规则。

第一种启发式规则是仅考虑那些利用所有的处理器，对每一个操作都并行计算，并且不采用流水线的执行计划。Teradata DBC 系列采用的是这种方法。在这种执行计划中寻找最佳者类似于在串行系统做查询优化，二者主要的区别在于如何进行划分，以及采用什么代价估算公式。

第二种启发式规则是选择最高效的串行执行计划，然后对该执行计划中的各个操作进行并行处理。

优化的另一个方面是设计物理存储组织，以加快查询处理。不同的查询有不同的最佳物理组织。数据库管理员必须选择一种适合于预期数据库查询集合的物理组织。因此，并行查询优化领域很复杂，现在仍然是一个活跃的研究领域。

17.7 并行系统设计

本章迄今为止一直将注意力集中于数据存储的并行化和查询处理的并行化。因为大型并行数据库系统主要用于存储大量数据，以及基于这些数据的决策支持查询处理，所以上述议题是并行数据库系统中最重要的议题。如果要处理大量的输入数据，那么来自外部数据源数据的并行装入是一个重要的需求。

一个大型的并行数据库还必须注意下列可用性问题：

- 某些处理器和磁盘发生故障时的适应能力。
- 数据和模式发生改变时的联机重组织。

下面我们来讨论这些问题。

由于具有大量的处理器和磁盘，那么至少有一个处理器发生故障的概率就会比单处理器单磁盘的系统要大得多。如果并行系统设计得不好，那么当任何一个成分（处理器或磁盘）发生故障时，系统就会停止工作。假设单个处理器或磁盘发生故障的可能性很小，随着处理器和磁盘数目的增长，系统发生故障的概率也线性地增长。如果一个单处理器单磁盘系统每 5 年发生一次故障，那么具有 100 个处理器的系统每 18 天就会发生一次故障。

因此，像 Tandem 和 Teradata 这样的大型并行数据库系统都设计成即使一个处理器或磁盘发生了故障，系统还能运行。数据至少复制到两个服务器中，如果一个处理器发生故障，它所存储的数据还能从另一个处理器中访问到。系统记住哪些处理器发生了故障，并将工作分配给能正常运行的处理器。对于发生故障的节点中存放数据的请求被自动地送到存放了该数据副本的节点中去。如果处理器 A 中的所有数据都复制到处理器 B 中，那么 B 将处理对 A 的所有请求，同时还要处理对它自己的请求，这就会使 B 成为瓶颈。因此，应该将一个处理器中数据的副本分配到多个其他处理器中。

当处理大量数据（tera 字节规模）时，像建立索引这样的简单操作，以及像往关系中增加一个列这样的模式改变，都会花费很长的时间，可能要数小时甚至数天。因此要在进行这样的操作时让数据库停止运行是不可接受的。像 Tandem 这样的并行数据库系统允许这种操作联机地进行。

例如，来看一个联机的索引建立。支持这一特性的系统即使正在某个关系上建立索引时，也允许在该关系上进行插入、删除和更新。因此，建立索引这一操作不能像一般情况那样以共

享方式封锁整个关系，而是需要该进程记住在它活动期间所发生的更新，并将这些变化映射到正在建立的索引中去。

17.8 总结

在过去 15 年中，并行数据库已经得到了广泛的商业认同。在 I/O 并行方面，关系被划分存储到多个可用磁盘中，从而使检索速度更快。有三种常用的划分技术：轮转法分布，散列分布和范围分布。偏斜是一个主要的问题，特别是当并行度增高时该问题更为突出。

查询间并行是同时运行多个不同的查询，以提高吞吐量。查询内并行是为了减少运行一个查询的代价。有两类查询内并行：操作内并行和操作间并行。

我们采用操作内并行来并行地执行关系操作，例如排序、连接等。因为关系操作是面向集合的操作，所以操作内并行是很自然的。对于像连接这样的二元操作，有两种基本的方法使其并行化。基于划分的并行是将两个关系都分成几个部分，r_i 中的元组仅与 s_i 中的元组连接。非对称的分片 - 复制并行是将一个关系进行划分，而将另一个关系进行复制。对于任何连接条件，都可以采用分片 - 复制方法，这一点与基于划分的并行不同。此外，这两种并行技术都可以与任何一种连接技术结合使用。

独立的并行是将互不依赖的多个不同的操作并行地执行。流水线并行是让处理器在完成整个操作之前，就将计算出的结果送给另一个操作。并行数据库中的查询优化比串行数据库中的查询优化要复杂得多。

习题

17.1 对于轮转法、散列分布和范围分布这三种划分技术，请各给出一个使用该划分技术能提供最快响应的查询例子。

17.2 当在范围划分属性上进行范围选择时，数据存取可能仅在一个磁盘上进行，请描述这一特性的优点和缺点。

17.3 当关系基于它的一个属性采用：

(a) 散列分布

(b) 范围分布

进行划分时，哪些因素会导致偏斜？对于上述两种方法，可以分别采取什么措施来减小偏斜？

17.4 对于下述的每一种任务

(a) 提高执行许多小查询的系统的吞吐量。

(b) 在磁盘和处理器数目都很大的情况下，提高执行少量大查询的系统的吞吐量。

哪一种并行形式（查询间并行、操作间并行、操作内并行）最重要？

17.5 对于流水线并行，即使当有许多处理器可用时，往往也将一个流水线中的几个操作只在一个处理器上执行。

(a) 解释这是为什么。

(b) 如果所使用的机器采用共享内存体系结构，上述说法还正确吗？为什么？

(c) 对于独立的并行，上述说法正确吗？（也就是说，即使在不将操作组织到流水线中，而且有许多处理器可用的情况下，是否仍然最好将几个操作放在同一个处理器中执行？）

17.6 请给出一个不是等值连接，但仍然可以采用基于划分的并行的例子。应该基于什么属性来进行划分？

17.7　考虑采用范围分布的对称的分片－复制连接处理，如果连接条件的形式为$|r.A\text{-}s.B| \leqslant K$，其中$K$是一个小常数，$|x|$表示取$x$的绝对值，（这样条件的连接称作带状连接），那么你如何来优化查询计算？

17.8　对于下面的每一种操作，给出一种好的并行方法。

(a) 差运算。

(b) 使用 Count 运算的聚集。

(c) 使用 Count distinct 运算的聚集。

(d) 使用 avg 运算的聚集。

(e) 左外连接，其连接条件只涉及相等比较。

(f) 左外连接，其连接条件涉及相等之外的比较。

(g) 完全外连接，其连接条件涉及相等之外的比较。

17.9　我们讲到可以利用直方图来建立负载均衡的范围分布：

(a) 假设你有一个取值范围为 1～100 的直方图，划分为 10 个范围，1～10，11～20，…，91～100，其出现率分别为 15，5，20，10，10，5，5，20，5，5，请给出一个负载均衡的范围分布函数，将这些值划分为 5 个部分。

(b) 请给出一个算法，对于给定的包括n个范围的属性值出现率直方图，计算出划分为P个部分的一个均衡的范围分布。

17.10　请叙述采用流水线并行的优点和缺点。

17.11　有些并行数据库系统对于每一个数据项都在附属于另一个处理器的磁盘中存放它的一个额外的拷贝，以避免当一个处理器发生故障时造成的数据丢失。

(a) 一个好的办法是将一个处理器的数据拷贝划分存储到多个处理器中，为什么？

(b) 如果采用 RAID 存储，而不是存储每一个数据项的一个额外拷贝，有什么优点和缺点？

文献注解

1983 年，关系数据库系统在市场中出现，现在，它已主宰了市场。70 年代末，80 年代初，当关系模型取得了相当稳固的地位时，人们意识到关系操作是高度可并行的，并且具有很好的数据流特性。商品化系统 Teradata 和几个研究项目，例如 GRACE（Kitsuregawa 等［1983］、Fushimi 等［1986］）、GAMMA（DeWitt 等［1986，1990］）、Bubba（Boral 等［1990］等）接连启动。研究人员利用这些并行数据库系统来研究对关系操作进行并行执行的可能性。而后，在 80 年代末和 90 年代，更多的公司，例如 Tandem、Oracle、Informix、Red-Brick 等也进入了这个市场。学术界的研究项目包括 XPRS（Stonebraker 等［1989］）和 Volcano（Graefe［1990］）。

Joshi［1991］、Mohan 和 Narang［1991，1992b］讨论了并行数据库中的封锁问题。Dias 等［1989］、Mohan 和 Narang［1991，1992］、Rahm［1993］讨论了并行数据库系统中的缓存相关协议。Carey 等［1991］讨论了客户－服务器系统中的缓存问题。Bayer 等［1980］讨论了数据库系统中的并行和恢复。

Graefe［1993］对查询处理，包括查询的并行处理给了一个极好的综述。DeWitt 等［1992］讨论了并行排序。讨论并行连接算法的文献很多，包括 Nakayama 等［1984］、Kitsuregawa 等［1983］、Richardson 等［1987］、Schneider 和 DeWitt［1989］、Kitsuregawa 和 Ogawa［1990］、Lin 等［1994］、Wilschut 等［1995］，以及其他文献。Tsukuda 等［1992］、Deshpande 和 Larson

[1992]、Shatdal 和 Naughton [1993] 讨论了共享内存体系结构的并行连接算法。Walton 等 [1991]、Wolf [1991]、DeWitt 等 [1992] 描述了并行连接中的偏斜处理。Seshadri 和 Naughton [1992]、Ganguly 等 [1996] 描述了并行数据库中的采样技术。

Graefe [1990, 1993] 提出了一个称作交换操作模型的并行模型，它使用操作的已有实现，在数据的本地拷贝上进行操作，同时伴随着一个将数据移来移去的交换操作。关于操作的并行化描述一定程度上基于该模型。

Lu 等 [1991]、Hong 和 Stonebraker [1991]、Ganguly 等 [1992]、Lanzelotte 等 [1993]、Hasan 和 Motwani [1995] 描述了并行查询优化技术。

第 18 章　分布式数据库

在第 16 章里，我们讨论了分布式系统的基本结构。分布式系统与并行系统不同，并行系统中处理器是紧耦合的，它们形成了单一的数据库系统；而分布式数据库系统由一些松耦合的节点组成，这些节点不共享任何物理部件。此外，分布式系统中每个节点上运行的数据库系统之间做到了真正意义上的相互独立。

每个节点都可以参与事务的执行，这些事务所访问的数据可以位于一个节点上也可以位于几个节点上。集中式数据库系统和分布式数据库系统的主要差别在于，前者数据存在于唯一一个地方，而后者数据存在于几个地方。数据的分布给事务处理和查询处理带来很多困难。本章将谈到这些困难。

首先在 18.1 节中，我们看一看如何在分布式数据库中存储数据的问题。18.2 节中，我们考虑关于网络透明性和数据项命名的问题。分布式数据库中的一些查询处理技术已经得到了发展，我们将在 18.3 节中讨论。

接下来的几节是关于事务处理的。18.4 节讨论基本模型和由各种故障引起的问题。18.5 节描述提交协议，协议的目的是即使故障引起这些问题，也要提供原子事务提交。18.6 节考虑被指定作为协调器的节点发生故障时选择新协调器的恢复问题。分布式系统中的并发控制在 18.7 节讨论。关于死锁问题的处理在 18.8 节讨论。

近年来出现了访问和更新来自先前存在的多种数据库中数据的需求，这些先前存在的数据库在硬件和软件环境以及数据存储的模式上各不相同。多数据库系统是一个软件层，它使得我们可以像对待同构分布式数据库那样对待这种异构数据库集合。18.9 节讨论多数据库系统的查询处理和事务处理的有关问题。

18.1　分布式数据存储

考虑一个要存储到数据库中的关系 r。在分布式数据库中存储这个关系可以有几种方法：

- 复制。系统维护这个关系的几个完全相同的副本（拷贝），各个副本存储在不同的节点上，这就是数据复制。与复制相对的方式是只存储关系 r 的一个拷贝。
- 分片。关系被划分为几个片段，各个片段存储在不同的节点上。
- 复制和分片。关系被划分为几个片段，系统为每个片段维护几个副本。

在下面几个小节中，我们将详细阐述这些技术。

18.1.1　数据复制

如果关系 r 被复制，则在两个或两个以上的节点存有关系 r 的拷贝。极端情况下，我们采用全复制，这时系统中的每个节点都存有 r 的一个拷贝。

复制有优点也有缺点。

- 可用性。当包含关系 r 的节点之一发生故障时，关系 r 可以在另一个节点上找到。因此，尽管一个节点上发生了故障，系统仍可以继续处理涉及 r 的查询。
- 增加的并行度。如果绝大多数对关系 r 的访问只导致对关系的读取，那么几个节点就

可以并行地处理涉及 r 的查询。r 的副本越多，在事务执行的节点上发现所需数据的可能性就越大。这样，数据复制减少了数据在节点之间的移动。

• 增加的更新开销。系统必须保证关系 r 的所有副本是一致的，否则就可能产生错误的计算。因此，只要 r 被更新了，更新就必须传播到包含副本的所有节点。这会导致开销的增加。例如，在某个银行系统中，一个帐户信息在不同节点进行了复制，则有必要保证在所有节点上同一帐户的余额保持一致。

通常情况下，复制可以提高 read 操作的性能，并且增加只读事务的数据可用性。但是，更新事务的开销会增大。控制几个事务对复制数据的并发更新比我们在第 14 章所看到的采用集中式的方法进行并发控制更加复杂。通过选择副本中的一个来作为 r 的主副本，我们可以简化关系 r 的副本管理。例如在银行系统中，帐户可以同其开户节点联系起来。同样，在机票预定系统中，航班可以同其起飞的节点联系起来。我们将在 18.7 节中讨论分布式并发控制可选择的方法。

18.1.2　数据分片

对关系 r 进行分片是将 r 划分为多个片段 r_1，r_2，…，r_n。这些片段中包含足够的信息，使得我们能够重构原始关系 r。如下所述，这种重构可以通过在各个片段上做并运算或一种特殊的连接运算来实现。对一个关系进行分片可以有两种不同的模式：水平分片和垂直分片。水平分片通过将 r 的每个元组分到一个或多个片段中来对关系进行划分；垂直分片则是利用我们在后面将讨论的某种方式对关系 r 的模式 R 进行分解来对关系进行划分。这两种模式都可以连续地用于同一个关系，产生一些不同的片段。注意，某些信息可能在几个片段中出现。

从下三部分讨论对一个关系进行分片的不同方法。我们将通过对关系 *account* 进行分片来说明这些方法，关系 *account* 的模式为 *Account-schema* = (*branch-name*, *account-number*, *balance*)，关系 *account* (*Account-schema*) 如图 18-1 所示。

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Valleyview	A-177	205
Valleyview	A-402	10000
Hillside	A-155	62
Valleyview	A-408	1123
Valleyview	A-639	750

图 18-1　作为例子的 *account* 关系

1. 水平分片

关系 r 被划分为多个子集 r_1，r_2，…，r_n。关系 r 的每个元组必须至少属于一个片段，以便在需要时可以重构原始关系。

一个片段可以被定义为整个关系 r 上的一个选择。也就是说，可以用谓词 Pi 构造片段 ri 如下：

$$r_1 = \sigma_{Pi}(r)$$

我们可以通过将所有片段相并来实现关系 r 的重构，即

$$r = r_1 \cup r_2 \cup \cdots \cup r_n$$

我们来看一个例子。假设关系 r 是图 18-1 中的 *account* 关系。这个关系可以划分为 n 个不同的片段，每个片段中包含属于某个分支机构的 account 元组。如果银行系统中只有两个分支机构 (Hillside 和 Valleyview)，那么就会产生两个不同的片段：

$$account_1 = \sigma_{branch\ name = \text{"Hillside"}}(account)$$

$$account_2 = \sigma_{branch\text{-}name = \text{"Vallegview"}}(account)$$

这两个片段如图 18-2 所示。片段 $account_1$ 存储在 Hillside 节点，片段 $account_2$ 存储在 Valleyview 节点。

这个例子中，片段是不相交的。通过改变用于构造片段的选择谓词，我们可以让 r 的某个元组出现在不止一个 r_i 中。这种数据复制形式在本节末尾会进一步讨论。

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_1$

branch-name	*account-number*	*balance*
Valleyview	A-117	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$account_2$

图 18-2　关系 *account* 的水平分片

2. 垂直分片

垂直分片最简单的形式和分解（见第 7 章）一样。r（R）的垂直分片需要定义模式 R 的几个属性子集 R_1，R_2，…，R_n，使得

$$R = R_1 \cup R_2 \cup \cdots \cup R_n$$

片段 r_i 被定义为

$$r_i = \Pi_{R_i}(r)$$

分片应使得我们可以通过采用自然连接

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \cdots \bowtie r_n$$

从分片所产生的片段重构关系 r。

保证关系 r 能被重构的一种方法是在每个 R_i 中都包含 R 的主码属性。更一般地，我们可以使用任何超码。通常在模式 R 中加入一个特殊的，被称为 *tuple-id* 的属性会比较方便。元组的 *tuple-id* 的值是唯一的，用来将一个元组同其他所有元组区别开来。这样，*tuple-id* 就成为扩展后模式的一个候选码，并被包含到每个 R_i 中。元组的物理地址或逻辑地址都可用作 *tuple-id*，因为每个元组都有唯一的地址。

为了说明垂直分片，我们以银行数据库为例。我们采用另一种数据库设计，其中包括模式[⊖]

Deposit-schema = (*branch-name*, *account-number*, *customer-name*, *balance*)

branch-name	*account-number*	*customer-name*	*balance*
Hillside	A-305	Lowman	500
Hillside	A-226	Camp	336
Valleyview	A-177	Camp	205
Valleyview	A-402	Kahn	10000
Hillside	A-155	Kahn	62
Valleyview	A 408	Kanh	1123
Valleyview	A-639	Green	750

图 18-3　作为例子的 *deposit* 关系

图 18-3 所示为 *deposit* 关系。图 18-4 中给出了关系 *deposit′*，它是在图 18-3 所示关系 *deposit* 上增加 *tuple-id* 得到的。图 18-5 所示将模式 *Deposit-schema* $\cup$ {*tuple-id*} 垂直分解为

⊖ 尽管这本书的其他地方所用的高度规范化数据库设计也可以进行垂直分片，但这种分片不会有多大用处。垂直分片对这里所用的这种模式更有意义。

$$Deposit\text{-}schema\text{-}1 = (branch\text{-}name,\ customer\text{-}name,\ tuple\text{-}id)$$

$$Deposit\text{-}schema\text{-}2 = (account\text{-}number,\ balance,\ tuple\text{-}id)$$

图 18-5 所示的两个关系由下面的计算产生

$$deposit_1 = \Pi_{Deposit\ schema\text{-}1}\ (deposit')$$

$$deposit_2 = \Pi_{Deposit\ schema\ 2}\ (deposit')$$

要从这两个片段重构原始的 *deposit* 关系，需要计算

$$\Pi_{Deposit\ schema}\ (deposit_1 \bowtie deposit_2)$$

注意表达式

$$deposit_1 \bowtie deposit_2$$

是自然连接的一种特殊形式，连接属性是 *tuple-id*。尽管 *tuple-id* 属性有助于垂直划分的实现，但对用户来说它必须是不可见的，因为它是实现时人为地在内部制造出来的，并且它违反了数据独立性——这是关系模型的主要优点之一。

branch-name	*account-number*	*customer-name*	*balance*	*tuple-id*
Hillside	A-305	Lowman	500	1
Hillside	A-226	Camp	336	2
Valleyview	A-177	Camp	205	3
Valleyview	A-402	Kahn	10000	4
Hillside	A-155	Kahn	62	5
Valleyview	A-408	Kahn	1123	6
Valleyview	A-639	Green	750	7

图 18-4　在图 18-3 中 *deposit* 关系上加 tuple-id

3. 混合分片

关系 r 被划分为一些片段关系 r_1，r_2，…，r_n。每个片段都是在关系 r 或先前产生的 r 的某个片段上运用水平分片或垂直分片产生的结果。我们来看一个例子。假设关系 r 是图 18-3 所示 *deposit* 关系。这个关系最初像前面定义的那样被划分为 $deposit_1$ 和 $deposit_2$。我们现在可以进一步来划分 $deposit_1$ 片段，采用水平分片模式，可以把它划分为两个片段：

$$deposit_{1a} = \sigma_{branch\text{-}name = \text{"Hillside"}}\ (deposit_1)$$

$$deposit_{1b} = \sigma_{branch\text{-}name = \text{"Valleyview"}}\ (deposit_1)$$

这样，关系 r 就被划分为了三个片段：$deposit_{1a}$，$deposit_{1b}$ 和 $deposit_2$。每个片段可以位于一个不同的节点。

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1$

account-number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2$

图 18-5　关系 *deposit* 的垂直分片

18.1.3　数据复制与分片

上两节描述的数据复制和数据分片技术可以连续地作用于同一个关系。也就是说，片段可

以被复制，片段的副本也可以进一步被分片，等等。例如，我们来看一个包含节点 S_1，S_2，…，S_{10}的分布式系统。我们可以将 *deposit* 分片为 $deposit_{1a}$，$deposit_{1b}$和 $deposit_2$。然后呢？举例来说，我们可以在节点 S_1，S_3 和 S_7 上存储 $deposit_{1a}$的一个拷贝；在节点 S_7 和 S_{10}上存储 $deposit_{1b}$的一个拷贝；在节点 S_2，S_8 和 S_9 上存储 $deposit_2$ 的一个拷贝。

18.2　网络透明性

在上节中我们看到，关系 r 可以多种方式存储在分布式数据库系统中。系统应减少用户对关系如何存储这个问题所要了解的程度，这是非常基本的。如下所述，系统可以屏蔽网络中数据分布的细节。我们称这种屏蔽为网络透明性，并用系统用户对数据项如何存储以及存在什么地方不了解的程度来定义它。

我们将从以下几点来考虑透明性问题：

- 数据项的命名。
- 数据项的复制。
- 数据项的分片。
- 片段和副本的位置。

18.2.1　数据项的命名

数据项（如关系、片段和副本）必须有唯一的名字。集中式数据库中这一特性很容易保证，然而在分布式数据库中我们必须小心地保证不同节点上不会用同一个名字来代表不同的数据项。

这个问题的一个解决方法是要求所有的名字都在一个中央名字服务器中注册。名字服务器有助于保证同一名字不会用于不同的数据项。我们还可以利用名字服务器在已知数据项名字时定位数据项。但是，这种方法有两个主要缺点。首先，当数据项通过名字来定位时，名字服务器可能成为性能瓶颈，从而导致性能低下。其次，如果名字服务器崩溃，分布式系统中的任何节点都不可能继续运行下去。另一种方法是要求每个节点都将其节点标识作为前缀加到该节点所产生的所有名字上。这种方法可以保证任何两个节点不会产生相同的名字（因为每个节点的标识都是唯一的），并且不需要中央控制。但是，这种方法不能达到网络透明性，因为节点标识被附加到名字上。这样，关系 *account* 就可能需要用 *site*17.*account* 来引用，而不是简单地用 *account* 来引用。

为了解决这个问题，数据库系统可以为数据项创建一个可用的另外的名字或别名集合。这样，用户可以用简单的名字来引用数据项，而简单的名字被系统翻译为完整的名字。从别名到真实名字的映射可以存储在每个节点上。有了别名，用户就可以不必知道数据项的物理位置。此外，当数据库管理员决定将数据项从一个节点移到另一个节点时，这种移动也不会影响到用户。

数据项的每个副本以及数据项的每个片段也必须有唯一的名字。对系统来说，能够确定多个副本是同一数据项的副本或多个片段是同一数据项的片段是非常重要的。我们采用后缀“.*f*1”，“.*f*2”，…，“.*fn*”表示数据项的多个片段，后缀“.*r*1”，“.*r*2”…，“.*rn*”表示数据项的多个副本。因此，

$$site17.account.f3.r2$$

指的就是 *account* 的片断 3 的副本 2，并告诉我们这个数据项是节点 17 产生的。

我们不能指望用户去引用数据项的具体副本。相反地，系统对 read 请求应该能确定需引用哪个副本，而对 write 请求则应该更新所有副本。我们可以通过维护一个目录表来保证这一点，系统用目录表来确定数据项的所有副本。

我们同样也不能要求用户知道数据项是如何被分片的。如前所述，垂直分片可能包含 *tuple-id*，水平分片可能涉及复杂的选择谓词。因此，分布式数据库系统应允许用未分片的数据项来表达请求。这一要求不会带来什么太大问题，因为我们总可以从片段重构原始数据项。但是，从片段重构数据的做法效率可能低下。回头再看对 *account* 的水平分片，考虑查询

$$\sigma_{branch\text{-}name = \text{“Hillside”}}(account)$$

我们只用片段 $account_1$ 就能回答这个查询。但是，分片透明性要求用户不必知道片段 $account_1$ 和 $account_2$ 的存在。如果在处理查询前先重构 *account*，那么我们得到表达式

$$\sigma_{branch\text{-}name = \text{“Hillside”}}(account_1 \cup account_2)$$

此表达式的优化留给查询优化器去完成（见 18.3 节）。

图 18-6 给出了对给定数据项名字进行翻译的整个模式。为了说明此模式如何运转，我们来看一个位于 Hillside 分支机构（节点 S_1）的用户。对 *account* 关系的本地片段 *account*.*f*1，此用户使用别名 *local-account*。当此用户引用别名 *local-account* 时，查询处理子系统在别名表中查找 *local-account*，然后用 S1.*account*.*f*1 替代 *local-account*。S1.*account*.*f*1 可能被复制，如果这样，系统必须查找副本表来选择一个副本。但选出的副本自身又可能被分片，这就要求检查分片表。大多数情况下只需要查 1～2 个表。但是，图 18-6 的名字翻译模式非常一般化，足够处理对关系进行连续复制和分片的任意组合。

```
if name 出现在别名表中
    thenexpression := map (name)
    elseexpression := name;

functionmap (n)
    ifn 出现在副本表中
        thenresult := n 的一个副本的名字;
    ifn 出现在分片表中
        then begin
            result := 构造片段的表达式
            for each n'in result do begin
                用 map (n') 代替结果中的 n'
            end
        end
    return result;
```

图 18-6　名字翻译算法

18.2.2　透明性与更新

为更新数据库的用户提供透明性比为读数据库的用户提供透明性要稍微困难一些。主要问题在于必须保证数据项的所有副本都要被更新，并且所有影响到的片段也都要被更新。

一般说来，被分片和复制的数据的更新问题同视图维护问题相关——即同数据库关系更新时保持实体化视图最新这个问题相关（见 3.7.1 节）。考虑 *account* 关系的例子，我们要插入元组

（“Valleyview”，A-733，600）

如果 *account* 水平分片，则有谓词 P_i 同第 i 个片段相关。我们把 P_i 用在元组（"Valleyview"，A-733，600）上，检查元组是否需要被插入到第 i 个片段中。如果像前面例子中那样将 *account* 分片为

$$account_1 \quad \sigma_{branch\text{-}name = \text{"Hillside"}}(account)$$

$$account_2 \quad \sigma_{branch\text{-}name = \text{"Valleyview"}}(account)$$

则元组应被插入 $account_2$ 中。

现在我们来看将 *deposit* 垂直分片为 $deposit_1$ 和 $deposit_2$ 的情况。元组（"Valleyview"，A-733，"Jones"，600）必须被分裂为两个片段：一个插入 $deposit_1$ 中，一个插入 $deposit_2$ 中。

如果对一个被复制的关系进行更新，则更新必须被施加到每个副本上。如果存在对关系的并发访问，这一要求就会带来问题，因为有可能一个副本比另一个副本更新得早，我们将在 18.7 节讨论这个问题。

18.3　分布式查询处理

第 12 章中我们看到计算一个查询的结果可以有不同的方法，那里我们讨论了一些用来选择处理查询策略以减少计算结果时间的技术。在集中式系统中，衡量某个策略所需代价的基本准则是磁盘访问量。在分布式系统中，我们还必须考虑另外一些因素，包括

- 数据在网络上传输的代价。
- 通过在几个节点同时分别处理查询的一部分，性能可能得到的提高。

数据在网络上传输和数据传入传出磁盘的相对代价同网络类型以及磁盘速度相关。因此，通常情况下我们不仅注意磁盘开销或网络开销，而且必须在这两者间取得良好的折衷。

18.3.1　查询转换

我们来看一个最简单的查询："找出 *account* 关系中的所有元组"。虽然这个查询很简单（甚至实际上是微不足道的），但对它的处理却不是微不足道的，因为正如我们在 18.1 节中所看到的那样，*account* 关系可能被分片、被复制、也可能既被分片又被复制。如果 *account* 关系被复制，就需要对副本进行选择。如果所有副本都没有被分片，就选择使传输代价最小的那个副本。可是，如果副本被分片了，选择就不那么容易了，因为我们需要进行一些连接或并运算来重构 *account* 关系。这种情况下，该简单例子的策略可能会很多。这时，通过穷举所有可能的策略来对查询进行优化是不可行的。

分片透明性意味着用户可以书写这样的查询

$$\sigma_{branch\ name = \text{"Hillside"}}(account)$$

因为 *account* 被定义为

$$account_1 \cup account_2$$

由名字翻译模式产生的表达式为

$$\sigma_{branch\text{-}name = \text{"Hillside"}}(account_1 \cup account_2)$$

采用第 12 章的查询优化技术，可以自动简化上述表达式，产生结果为

$$\sigma_{branch\ name = \text{"Hillside"}}(account_1) \cup \sigma_{branch\text{-}name = \text{"Hillside"}}(account_2)$$

此表达式包括了两个子表达式。第一个只涉及 $account_1$，因此可以在 Hillside 节点求值。第二个只涉及 $account_2$，因此可以在 Valleyview 节点求值。

在对

$$\sigma_{branch\text{-}name = \text{“Hillside”}}\ (account_1)$$

求值时还可以进行进一步的优化。由于 $account_1$ 只包含属于 Hillside 分支机构的元组，因此可以去掉选择运算。在对

$$\sigma_{branch\ name = \text{“Hillside”}}\ (account_2)$$

求值时，我们可以运用 $account_2$ 的定义得到

$$\sigma_{branch\text{-}name = \text{“Hillside”}}\ (\sigma_{branch\ name = \text{“Valleyview”}}\ (account))$$

不管 *account* 关系的内容是什么，此表达式都是空集。

这样，我们最终的策略是让 Hillside 节点将 $account_1$ 作为查询结果返回。

18.3.2　简单的连接处理

正如我们在第 12 章看到的那样，选择查询处理策略的一个主要方面就是选择连接策略。考虑下面的关系代数表达式：

$$account \bowtie depositor \bowtie branch$$

假设这三个关系都既没被复制又没被分片，且 *account* 存储在节点 S_1，*depositor* 存储在节点 S_2，*branch* 存储在节点 S_3。设 S_I 表示提出查询的节点，系统需要在节点 S_I 产生结果。下面是处理这个查询可采取策略中的几个：

- 将三个关系的拷贝都送到节点 S_1。使用第 12 章中的技术，在节点 S_1 为本地地处理整个查询选择一个策略。
- 将关系 *account* 的一个拷贝送到节点 S_2，在 S_2 计算 $temp_1 = account \bowtie depositor$。将 $temp_1$ 从 S_2 送到 S_3，在 S_3 计算 $temp_2 = temp_1 \bowtie branch$。最后将结果 $temp_2$ 送到 S_1。
- 设计和上一个类似的策略，只是改变 S_1，S_2，S_3 的角色。

任何一个策略都不会总是最好的。我们需要考虑的因素中包括运送的数据量、在一对节点间传输数据块的代价以及各个节点处理速度的相对快慢。考虑上面列出的前两个策略。如果我们将三个关系都送到节点 S_1，并且这些关系上存在索引，那么在 S_1 我们可能需要重新建立这些索引。重建索引带来了额外的处理开销和磁盘访问。然而，第二种策略也存在缺点，那就是可能一个很大的关系（*customer* ⋈ *account*）必须从 S_2 送到 S_3。此关系中，客户每拥有一个帐户，其地址就要重复一次。因此同第一种策略相比，第二种策略可能导致额外的网络传输。

18.3.3　半连接策略

假设我们希望求表达式 $r_1 \bowtie r_2$ 的值，其中 r_1 和 r_2 分别存储在节点 S_1 和 S_2 上。设 r_1 和 r_2 的模式分别为 R_1 和 R_2。并假设我们希望在节点 S_1 得到结果。如果 r_2 中的许多元组都不能和 r_1 中的任何元组相连接，那么把 r_2 送到 S_1 会使我们必须传输那些对结果毫无贡献的元组。最好我们能在数据送到 S_1 以前去掉这些元组，特别是在网络代价较高的时候。

我们可以实现一个这样的策略：

1）在 S_1 计算 $temp_1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$。

2）将 $temp_1$ 从 S_1 送到 S_2。

3）在 S_2 计算 $temp_2 \leftarrow r_2 \bowtie temp_1$。

4）将 $temp_2$ 从 S_2 送到 S_1。

5）在 S_1 计算 $r_1 \bowtie temp_2$，产生的关系和 $r_1 \bowtie r_2$ 一样。

在考虑这个策略的效率以前，先要验证这个策略能否计算出正确的结果。在第 3）步，$temp_2$

是 $r_1 \bowtie \Pi_{R_1 \cap R_2}(r_1)$ 的结果。在第 5）步，我们计算

$$r_1 \bowtie r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$$

由于连接是可结合并且可交换的，因此可以将此表达式重写为

$$(r_1 \bowtie \Pi_{R_1 \cap R_2}(r_1)) \bowtie r_2$$

由于 $r_1 \bowtie \Pi_{R_1 \cap R_2}(r_1) = r_1$，因此这个表达式事实上是等于 $r_1 \bowtie r_2$ 的。

当 r_2 中只有较少元组对连接有贡献时，这个策略的优势尤其明显。如果 r_1 是一个涉及选择的关系代数表达式的结果，这种情况就可能发生。这种情况下，$temp_2$ 的元组数可能明显少于 r_2。这种策略节省的开销源于只需将 $temp_2$ 而不是 r_2 的全部送到 S_1，而增加的开销源于将 $temp_1$ 送到 S_2。如果 r_2 中对连接有贡献的元组比例足够小，传输 $temp_1$ 带来的额外开销就会远远小于由于只传输 r_2 的部分所节省的开销。

这种策略称为半连接策略，与关系代数中的半连接运算符一致，我们用 $\ltimes$ 表示。$r_1 \ltimes r_2$ 表示 r_1 和 r_2 的半连接，也就是

$$\Pi_{R_1}(r_1 \bowtie r_2)$$

因此，$r_1 \ltimes r_2$ 选出了 r_1 中对 $r_1 \bowtie r_2$ 有贡献的那些元组。在第 3）步中，$temp_2 = r_2 \ltimes r_1$。

对于几个关系之间的连接来说，这种策略可以扩展到一系列的半连接步骤。一个将半连接用于查询优化的充实的理论体系已经发展起来了，在文献注解中我们提到了此理论的一部分。

18.3.4　利用并行性的连接策略

在分布式系统中，通过重新分布元组来实现操作内的并行通常认为是不可行的，因为并行程度小且通信代价高。但操作间的并行，包括流水线并行和独立的并行（17.6 节），却是可以在分布式系统中发挥作用的。

例如，我们来看一个四个关系的连接：

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

其中关系 r_i 存储在节点 S_i 上。假设结果必须在 S_1 给出，我们可以有很多可能的并行求值策略，例如 17.6 节中描述的任一策略。利用这些策略，r_1 被送到 S_2 并在 S_2 计算 $r_1 \bowtie r_2$，同时 r_3 送到 S_4 并在 S_4 计算 $r_3 \bowtie r_4$。在 $r_1 \bowtie r_2$ 计算过程中节点 S_2 就可以把已经计算出来的元组送到 S_1，而不需要等到整个连接计算完。同样，S_4 也可以这样把 $r_3 \bowtie r_4$ 中的元组送到 S_1。采用 12.8.2 节的流水线连接技术，一旦 $(r_1 \bowtie r_2)$ 和 $(r_3 \bowtie r_4)$ 的元组到达 S_1，就可以开始计算 $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$。因此，$S_1$ 上最终连接结果的计算可以同 S_2 上 $r_1 \bowtie r_2$ 的计算以及 S_4 上 $r_3 \bowtie r_4$ 的计算并行地进行。

18.4　分布式事务模型

分布式系统中对各种数据项的访问常常通过事务来完成，事务必须保持 ACID 特性（见 13.1 节）。我们需要考虑两种类型的事务。局部事务是仅访问和更新一个局部数据库中数据的事务；全局事务是访问和更新多个局部数据库中数据的事务。局部事务的 ACID 特性可以用类似第 13、14 和 15 章中讨论的方式来保证，但在全局事务的情况下这种特性的保证就复杂得多，因为可能有几个节点参与执行。其中一个节点的故障，或者将这些节点连起来的一条通信链路的故障，都可能导致错误的计算。

18.4.1　系统结构

每个节点都有自己的局部事务管理器，其功能是保证该节点上执行的事务的 ACID 特性。各个事务管理器互相协作执行全局事务。为了帮助理解这种管理器怎样实现，我们定义事务系统的一个抽象模型。系统中的每个节点包括两个子系统：

- 事务管理器管理那些访问存储在一个局部节点中的数据的事务（或子事务）的执行。注意，这样的事务既可以是局部事务（即只在该节点上执行的事务），也可以是全局事务（即在几个节点上执行的事务）的一部分。
- 事务协调器协调该节点上发起的各事务（既有局部的又有全局的）的执行。

整个系统的体系结构如图 18-7 所示。

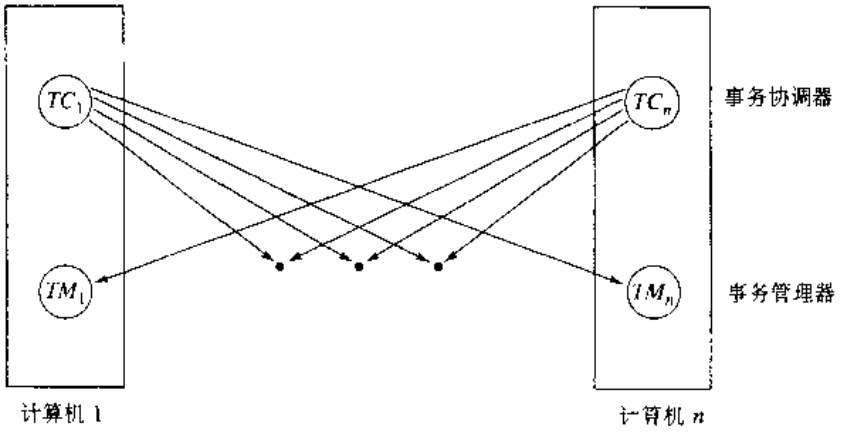


图 18-7　系统体系结构

事务管理器的结构在许多方面同在集中式系统的情况下所使用的结构类似。每个事务管理器都要负责

- 维护一个用于恢复的日志。
- 参与适当的并发控制模式，以协调在该节点上执行的事务的并发执行。

如下所述，为了支持事务的分布，我们既需要修改恢复模式，又需要修改并发模式。

集中式环境中不需要事务协调器子系统，因为一个事务只会访问唯一一个节点上的数据。正如其名字所暗含的那样，事务协调器负责协调该节点上发起的所有事务的执行。对每个这样的事务，协调器负责

- 启动事务的执行。
- 将事务分裂为一些子事务，并将这些子事务分派到恰当的节点上去执行。
- 协调事务的终止，其结果是事务在所有节点上都提交或事务在所有节点上都中止。

18.4.2　系统故障模式

分布式系统可能遭受和集中式系统所遭受的同样类型的故障（例如软件错误、硬件错误或磁盘崩溃），但分布式系统还有另外一些我们需要处理的故障类型。基本故障类型包括：

- 节点故障。
- 消息丢失。
- 通信链路故障。
- 网络分割。

分布式系统中总是可能发生消息的丢失或损坏。系统采用传输控制协议，如 TCP/IP，来处理这样的错误。关于这类协议的信息在一般的网络教科书中都可以找到（见文献注解）。

为了理解通信链路故障和网络分割的影响，首先必须懂得分布式系统中的节点是如何相互连接的。系统中的节点在物理上可以有多种连接方式。一些最常见的结构如图 18-8 所示。

每种结构都既有优点又有缺点。可以基于下列判断标准对这些结构进行比较：

- 安装开销。物理上将系统中节点链接起来所需要的开销。
- 通信开销。把消息从节点 A 发到节点 B 所需时间和金钱的开销。
- 可用性。即便某些节点或链路发生故障的情况下数据也可以被访问的程度。

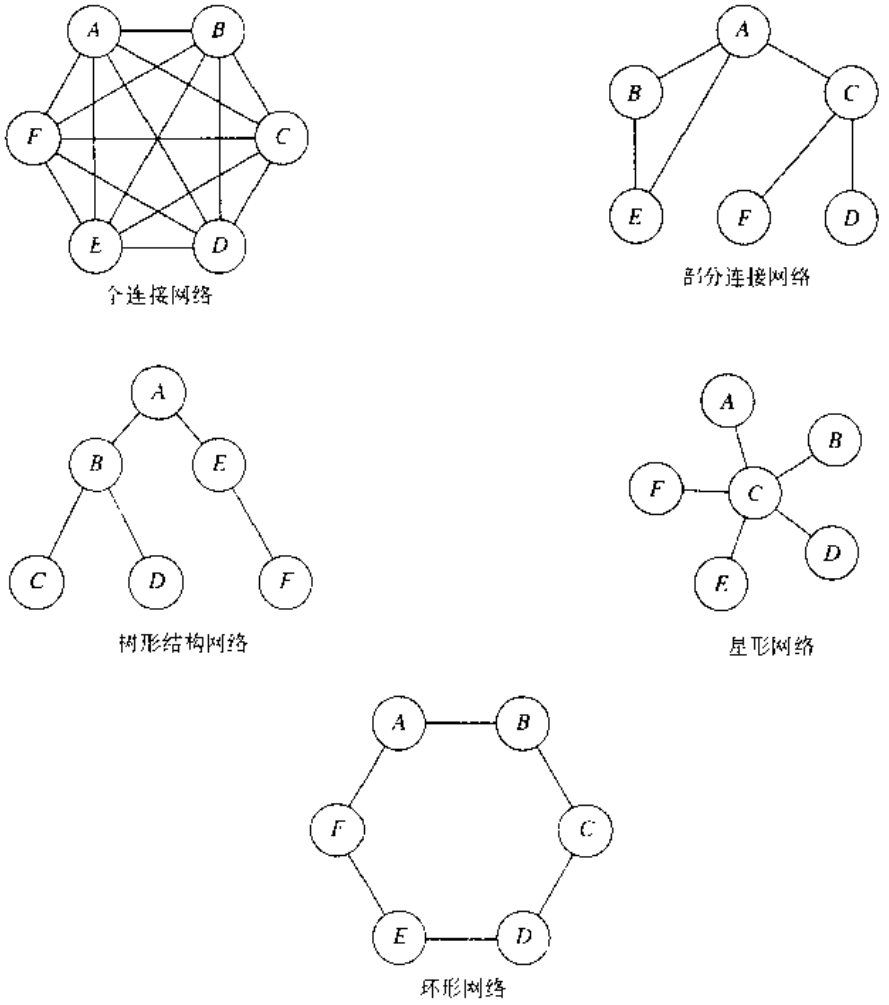


图 18-8　网络拓扑

图 18-8 表示了各种拓扑结构，其中结点对应于节点。从结点 A 到结点 B 的边对应于两个节点间的直接通信链路。在全连接网络中，每个节点同其他任何一个节点都是直接相连的。可是，由于链路数按节点数的平方增长，从而导致了昂贵的安装开销。因此，全连接网络在任何大系统中都是不实用的。

在部分连接网络中，直接链路存在于某些（但不是所有的）节点对之间。因此，这种结构的安装开销比全连接网络的安装开销要小。但是，如果节点 A 和节点 B 不是直接相连的，那么从一个节点到另一个节点的消息就必须通过由一系列通信链路构成的路径。这种要求带来了更高的通信开销。

如果某个通信链路发生故障，本该通过这条链路传输的消息就必须重新寻径。某些情况下可以找到通过网络的另一条路径，这时消息就可以到达目的地。然而在另一些情况下，故障可能导致某些节点对之间不再存在连接。如果一个系统被分裂为两个（或更多）子系统，那么这个系统是被分割的系统，这些子系统称为分区，分区之间缺乏相互连接。注意，在这种定义下，一个子系统中可以只包含一个结点。

图 18-8 所示各种不同的部分连接网络类型有各自的故障特性及安装和通信开销。树形结构网络的安装开销和通信开销相对较小。然而，树形结构网络中一条链路的故障就会导致网络分割。在环形网络中，分割的发生至少需要两条链路故障，因此，环形网络比树形网络的可用性程度高。但环形网络通信开销也高，因为消息必须跨过大量的链路。星形网络中一条链路的故障也会导致网络分割，但是其中的一个分区中只有一个节点。星形网络的通信开销也小，因为每个节点到其他任何一个节点至多两条链路。但是星形网络中，中央节点的故障将导致系统中所有节点的相互分离。

18.4.3　强壮性

分布式系统要做到强壮，就必须能检测故障，能重构系统以使计算继续进行，同时还能在处理器或链路修复后进行恢复。

不同的故障类型处理方式也不同。消息丢失通过重传来处理。对跨过某链路的一个消息进行多次重传后，却总得不到确认，这常常是链路故障的征兆。通常，网络试图为此消息寻找另一条路径。寻找这样一条路径的失败常常是网络分割的征兆。

但是，明确区分节点故障和网络分割通常是不可能的。系统可以检测到故障的发生，但却可能不能确定故障类型。例如，假设节点 S_1 不能同节点 S_2 通信，可能是 S_2 发生了故障，但也可能是由于 S_1 和 S_2 间的链路发生故障而导致了网络分割。

假设节点 S_1 已经发现有故障发生，这时 S_1 就必须发起一个过程，此过程使得系统可以重构并继续通常的操作模式。

- 如果被复制的数据存储在故障节点，就应修改目录，使得查询不再引用该故障节点上的拷贝。
- 如果发生故障时故障节点上有事务在进行，这些事务就应中止。最好能迅速中止事务，因为它们可能拥有仍然活跃的节点的数据上的锁。
- 如果故障节点是某些子系统的中央服务器，就必须进行选举来决定新的服务器（见 18.6.2 节）。中央服务器包括名字服务器、并发协调器以及全局死锁检测器。

由于区分网络链路故障和节点故障通常是不可能的，因此任何重构模式的设计都应使之在网络分割的情况下也能正确运行。特别地，我们应避免下面的情况：

- 两个甚至更多中央服务器在不同的分区中选出。
- 不止一个分区更新某个被复制的数据项。

将修复的节点或链路重新集成到系统中也要小心谨慎。故障节点恢复后，必须发起一个过程来更新系统表，使之能反映该节点停止工作后所发生的变化。如果该节点上有数据项的副本，它就必须获得这些数据项的当前值，并保证它能接收到以后的所有更新。节点的集成绝不像第一眼看起来那么简单，因为节点在恢复中时可能发生了对数据项更新的处理。一个简单的解决办法是当故障节点重新连入时暂时让整个系统停止。但是，在大多数应用中，这种暂时的停止可能导致混乱而不被接受。在允许更新数据项的同时使故障节点重新集成的技术已经得到了发展。如果一个故障链路恢复，两个或更多分区就能重新连接起来。由于网络分割限制了部

分节点或所有节点可做的操作，因而最好能将链路的恢复迅速通知所有节点。关于分布式系统恢复的更多信息请参看文献注解。

18.5 提交协议

如果要保证原子性，执行事务 T 的所有节点就必须在 T 执行的最终结果上取得一致。T 必须要么在所有节点上都提交，要么在所有节点上都中止。为了保证这一特性，T 的事务协调器必须执行一个提交协议。

两阶段提交协议（2PC）是最简单且使用最广泛的提交协议之一，将在下一小节描述。另外一种是三阶段提交协议（3PC），它避免了 2PC 的某些缺点但同时也增加了复杂性和开销，3PC 将在 18.5.2 小节描述。

18.5.1 两阶段提交

设事务 T 在节点 S_i 发起，并设 S_i 的事务协调器是 C_i。

1. 提交协议

当 T 完成其执行时（即执行 T 的所有节点都通知 C_i 已完成了 T 的执行时），C_i 启动 2PC 协议。

• 阶段 1。C_i 将记录<prepare T>加到日志中，并强制该日志写入稳定存储器中。接着它将一条 prepare T 消息发送到执行 T 的所有节点上。当收到这样一条消息时，节点上的事务管理器确定是否愿意提交 T 中属于它的那部分，如果答案是“不”，事务管理器就把记录<no T>加到日志中，并通过向 C_i 发送一条 abort T 消息来作出回答；如果答案是“是”，事务管理器就把记录<ready T>加到日志中，并将日志（包括所有与 T 相关的日志项）强制写入稳定存储器中，然后通过向 C_i 发送一条 ready T 消息来作出回答。

• 阶段 2。当 C_i 收到所有节点对 prepare T 消息的回答时，或者 prepare T 消息发送后一个预定的时间间隔已经过去时，C_i 就可以确定是将事务 T 提交还是中止。如果 C_i 从所有参与的节点处都收到 ready T 消息，事务 T 就可以被提交。否则，事务 T 必须被中止。根据结论的不同，记录<commit T>或记录<abort T>加到日志中，并将日志强制写入稳定存储器中。此时，事务的命运已经被决定了。此后，协调器向参与的所有节点发送消息 commit T 或消息 abort T。当节点收到此消息时，就把此消息记录到日志中。

在向协调器发送 ready T 消息之前，执行 T 的节点任何时候都可以无条件地将 T 中止。ready T 消息在效果上相当于节点所做的承诺，承诺的内容是保证按照协调器的命令来提交 T 或中止 T。节点能做出这样一个承诺的唯一依据是所需信息都存储在稳定存储器中。否则，如果节点在发送 ready T 后崩溃，就不能实现自己的承诺。

由于事务的提交需要全体一致地提交，因而只要有一个节点回答 abort T，事务 T 的命运就决定了。由于作为协调器的节点 S_i 是执行 T 的节点之一，所以协调器可以单方面决定将 T 中止。关于 T 的最后结论在协调器将此结论写入日志并强制写入稳定存储器时就已经确定了。在 2PC 协议的某些实现中，节点在协议第二阶段的最后向协调器发送 acknowledge T 消息。当协调器收到所有节点发来的 acknowledge T 消息后，就把记录<complete T>加到日志中。

2. 故障处理

我们现在来详细讨论 2PC 协议如何对各种类型的故障做出反应。

• 参与节点的故障。如果协调器 C_i 检测到某个节点发生了故障，它将采取如下行动。如果节点在用 ready T 消息回答 C_i 前发生故障，就假定该节点是用 abort T 消息来回答的。如果

节点在协调器从该节点接收到 ready T 消息后发生故障，提交协议的剩余部分就按照通常的形式执行，忽略该节点的故障。

如果参与节点 S_k 从故障中恢复，它必须检查它的日志，看看故障发生时正在执行的事务命运怎样。设 T 是这样一个事务，我们来考虑每种可能的情况：

- 日志中包含<commit T>记录。在这种情况下，该节点执行 redo (T)。
- 日志中包含<abort T>记录。在这种情况下，该节点执行 undo (T)。
- 日志中包含<ready T>记录。在这种情况下，该节点必须询问 C_i 以确定 T 的命运。如果 C_i 仍在工作，C_i 就告诉 S_k 关于 T 是提交还是中止的信息。在前一种情况下，S_k 执行 redo (T)；在后一种情况下，S_k 执行 undo (T)。如果 C_i 停止工作，S_k 就必须试着从别的节点上发现 T 的命运。它通过向系统中所有节点发送 query-status T 消息来执行这一操作。收到这样一条消息后，节点就必须查阅其日志看一看 T 是否在该节点上执行过，如果是，还要看一看 T 是提交了还是中止了，接着它把查阅结果告诉 S_k。如果每个节点都没有恰当的信息（即 T 是提交还是中止），那么 S_k 既不能中止 T 也不能提交 T，关于 T 的决定需要推迟到 S_k 能得到所需信息时再做。因此，S_k 必须定期地向其他节点重发 query-status T 消息。它持续不断地这样做，直到某个节点发现自己包含了所需信息。注意，C_i 所位于的节点通常会有所需信息。
- 日志中没有包含关于 T 的控制记录（abort，commit，ready）。这时，我们知道 S_k 在回答来自 C_i 的 prepare T 消息前发生故障。由于 S_k 的故障使得这样的回答不会被发送，根据算法，C_i 必然中止 T。因此，S_k 必须执行 undo (T)。

- 协调器的故障。如果协调器在为事务 T 执行提交协议的过程中发生故障，那么就必须由参与节点来决定事务 T 的命运。我们将会看见，某些情况下参与节点不能决定是提交 T 还是中止 T，因此这些节点必须等待发生故障的协调器恢复。
  - 如果某个活跃节点的日志中包含<commit T>记录，则 T 必须被提交。
  - 如果某个活跃节点的日志中包含<abort T>记录，则 T 必须被中止。
  - 如果某个活跃节点的日志中不包含<ready T>记录，则发生故障的协调器不可能已决定将 T 提交，因为日志中不包含<ready T>记录的节点不可能已向 C_i 发送了 ready T 消息。但是，协调器可能已经决定中止 T 而不是提交 T。同等待 C_i 恢复相比，中止 T 更可取一些。
  - 如果上述情况均不成立，则所有活跃节点的日志中都有<ready T>记录，但此外就没有别的控制记录（如<commit T>或<abort T>）。由于协调器已发生故障，因此不等到协调器恢复，就不可能知道协调器是否已做出决定，或者已做出的决定是什么。因此，活跃节点必须等待 C_i 的恢复。由于 T 的命运仍然是一个疑问，T 可能会继续占用系统资源。例如，如果使用了锁，T 可能拥有活跃节点的数据上的锁。这种情况不是我们所希望的，因为有可能需要几小时甚至几天的时间 C_i 才能重新变得活跃。在这段时间内，别的事务也可能被迫等待 T。其结果是，数据项不仅在发生故障的节点（C_i）上不能获得，在活跃的节点上也不能获得。这种情况称为阻塞问题，因为 T 由于等待节点 C_i 恢复而被阻塞。
- 网络分割。当网络被分割时，有两种可能性：
  1）协调器和它所有参与者处于一个分区中。这种情况下，故障对提交协议没有影响。
  2）协调器和它的参与者属于不同分区。从其中一个分区中的节点观点来看，就好像

其他分区中的节点发生了故障一样。不在协调器所位于的分区中的节点只需要执行处理协调器故障协议。协调器以及与协调器在同一分区中的节点遵循平常的提交协议，并假定其他分区中的节点发生了故障。

因此，2PC的主要缺陷在于协调器故障可能导致阻塞，这种情况下必须等到 C_i 恢复时才能做出提交 T 还是中止 T 的决定。

3. 恢复与并发控制

当故障节点重新启动时，我们可以使用某些算法，例如15.9节中所描述的算法，来进行恢复。为了处理分布式提交协议（如2PC和3PC），恢复过程必须对疑问事务进行特殊对待，疑问事务是发现有<ready T>日志记录，但既未发现<commit T>日志记录又未发现<abort T>日志记录的事务。如同上一部分所描述的那样，恢复节点必须同其他节点进行联系，以确定这种事务的提交-中止状态。

但是，如果恢复像刚才所描述的那样来完成，该节点上正常的事务处理就只有等所有疑问事务被提交或回滚后才能开始。找出疑问事务的状态可能会很慢，因为不得不与多个节点进行联系。此外，如果协调器发生了故障，而别的任何节点上都没有关于未完成事务提交-中止状态的信息，那么使用2PC时就存在着恢复被阻塞的潜在可能，这使得执行重启动恢复的节点在很长一段时间内都保持不可使用的状态。

为了克服这个问题，恢复算法通常提供对日志中记载封锁信息的支持。（我们这里假设锁被用于并发控制。）这时所写的日志记录不再是<ready T>，而是<ready T，L>，其中 L 是日志记录写入时事务拥有的所有锁的列表。恢复时即执行局部恢复行动后，对所有疑问事务而言，<ready T，L>日志记录（从日志中读到）中所记载的所有锁都需要重新获取。

当所有疑问事务重新获取锁完成后，该节点上的事务处理就可以开始，即使疑问事务的提交-中止状态还不能确定。疑问事务的提交或回滚与新事务的执行是并发的，这样，节点恢复可以加快，并且不会再被阻塞。注意，与疑问事务存在锁冲突的新事务只有在该疑问事务提交或回滚后才能取得进展。

18.5.2　三阶段提交

设计3PC协议的目的是为了在有限的故障可能情况下避免阻塞的可能性。我们所描述的3PC协议版本要求

- 不会发生网络分割。
- 当为事务执行3PC时至多 K 个参与节点发生故障，K 是一个表明协议对节点故障回复力的参数。
- 任何时候都至少有 $K+1$ 个节点处于工作状态。

协议的非阻塞特性是通过加入一个额外阶段来实现的，这个阶段对 T 的命运做出了初步决定。由于有了这个初步决定，参与节点就可获得一些信息，这使得不管协调器是否发生故障它都能作出决定。

1. 提交协议

和前面一样，设事务 T 在节点 S_i 发起，并设 S_i 的事务协调器是 C_i。

- 阶段1。这个阶段和2PC协议中的阶段1相同。
- 阶段2。如果 C_i 从某个参与节点收到abort T 消息，或者 C_i 在某个预定时间间隔内没有从参与节点收到任何回答，那么 C_i 就决定中止 T。实现中止决定所采用的方式与2PC中一样。如果 C_i 从每个参与节点都收到ready T 消息，C_i 就决定预提交 T。预提交与提交的差别

在于 T 最终仍然可能被中止。预提交决定使得协调器可以把所有参与节点都已准备好的消息告诉每个参与节点。C_i 将记录<precommit T>加到日志中并强制该日志写入稳定存储器中。接着，C_i 向所有参与节点发送 precommit T 消息。当节点从协调器收到一条消息（abort T 或 precommit T），就把它记录在日志中并强制此信息写入稳定存储器中，然后该节点把消息 acknowledge T 发给协调器。

• 阶段 3。这一阶段只在阶段 2 决定预提交时才执行。在向所有参与节点发 precommit T 消息后，协调器必须等待直到它收到至少 K 个 acknowledge T 消息。接着，协调器作出提交决定。它在日志中加入一条<commit T>记录，并强制将该日志写入稳定存储器。然后，C_i 向所有参与节点发 commit T 消息。当节点收到此消息后，就把这条信息记录到它的日志中。

和 2PC 协议中一样，在向协调器发送 ready T 消息之前，执行 T 的节点任何时候都可以无条件地将 T 中止。ready T 消息在效果上相当于节点所做的承诺，承诺的内容是保证按照协调器的命令来提交 T 或中止 T。和 2PC 协议中不同的是，2PC 中协调器在发出消息 commit T 之前的任何时候都可以无条件地中止 T，而 3PC 协议中 precommit T 消息是协调器所做的承诺，承诺的内容是保证按照参与者的命令来提交 T。

由于阶段 3 总是产生提交决定，因而这个阶段似乎没什么用。但是当我们看到 3PC 协议是怎样处理故障时，第三阶段的作用就变得明显起来。

在 3PC 协议的某些实现中，节点在收到 commit T 消息后向协调器发送 ack T 消息（注意我们用 ack 将这里的表示法同我们在阶段 2 中所用的 acknowledge 消息区别开来）。当协调器收到所有节点发来的 ack T 消息后，就把记录<complete T>加到日志中。

2. 故障处理

我们现在来详细讨论 3PC 协议如何对各种类型的故障作出反应。

• 参与节点的故障。如果协调器 C_i 检测到某个节点发生故障，它所采取的行动类似于 2PC 中采取的行动。如果节点在用 ready T 消息回答 C_i 前发生故障，就假定该节点是用 abort T 消息来回答的。否则，提交协议的剩余部分就按照通常的形式执行，忽略该节点的故障。

如果参与节点 S_j 从故障中恢复，它必须检查它的日志，看看故障发生时正在执行的事务命运怎样。设 T 是这样一个事务，我们来考虑每种可能的情况：

- 日志中包含<commit T>记录。在这种情况下，该节点执行 redo（T）。
- 日志中包含<abort T>记录。在这种情况下，该节点执行 undo（T）。
- 日志中包含<ready T>记录，但既无<abort T>记录又无<precommit T>记录。在这种情况下，该节点必须询问 C_i 以确定 T 的命运。如果 C_i 回答的消息是 T 中止，该节点就执行 undo（T）。如果 C_i 回答的消息是 precommit T，该节点（和阶段 2 中一样）将此信息记录在日志中，并通过向协调器发 acknowledge T 消息来继续协议的执行。如果 C_i 回答的消息是 T 提交，该节点就执行 redo（T）。如果在某个时间间隔后 C_i 仍未作出回答，该节点就执行协调器故障协议（见下一列表项）。
- 日志中包含<precommit T>记录，但既无<abort T>记录又无<commit T>记录。和前面一样，该节点需要询问 C_i。如果 C_i 回答的消息是 T 中止或 T 提交，该节点就分别执行 undo（T）或 redo（T）。如果 C_i 回答的消息是 T 仍处于预提交状态，该节点就从这一刻开始继续协议的执行。如果在某个时间间隔后 C_i 仍未作出回答，节点就需要找出新的协调器（如果需要则等待协调器被选出），然后像前面那样询问新的协调器。

• 协调器的故障。如果参与节点不能从协调器那里得到回答，不管这是什么原因造成的，该节点都执行协调器故障协议，此协议的结果是选出新的协调器。发生故障的协调器在恢复后成为一个参与节点，它不再担任协调器，但必须对新协调器作出的决定进行确定。

3. 协调器故障协议

协调器故障协议由预定时间内不能从协调器那里得到回答的参与节点激发。由于我们假设不存在网络分割，产生这一情况的唯一原因就是协调器发生了故障。

1）活跃节点采用选举协议来选出一个新的协调器（见 18.6 节）。

2）新协调器 C_{new} 向每个参与节点发送消息，询问 T 的局部状态。

3）每个参与节点，包括 C_{new}，各自确定 T 的局部状态：

- Committed。日志中包含<commit T>记录。
- Aborted。日志中包含<abort T>记录。
- Ready。日志中包含<ready T>记录，但既未包含<abort T>记录又未包含<precommit T>记录。
- Precommitted。日志中包含<precommit T>记录，但既未包含<abort T>记录又未包含<commit T>记录。
- Not ready。日志中既未包含<ready T>记录又未包含<abort T>记录。每个参与节点都把局部状态发送到 C_{new}。

4）根据收到的回答，C_{new} 决定提交 T、中止 T 还是重新启动 3PC 协议：

- 如果至少一个节点中 T 的局部状态 = committed，C_{new} 就提交 T。
- 如果至少一个节点中 T 的局部状态 = aborted，C_{new} 就中止 T。（注意，某个节点局部状态 = committed 而另一个节点局部状态 = aborted 的情况是不可能发生的。）
- 如果没有一个节点的局部状态 = aborted，也没有一个节点的局部状态 = committed，但至少有一个节点的局部状态 = precommitted，则 C_{new} 从发送新的 precommit 消息开始继续 3PC 的执行。
- 其他情况下，C_{new} 中止 T。

协调器故障协议使得新协调器可以获得有关发生故障的协调器 C_i 的状态的知识。

只要有一个活跃节点的日志中有<commit T>，C_i 就必然已决定提交 T。如果某个活跃节点的日志中有<precommit T>，则 C_i 必然已作出预提交 T 的初步决定，这意味着所有节点，包括可能发生了故障的节点，都已经到达 ready 状态。另外，事务不可能已经被中止，因为任何较早的协调器（后来发生了故障）都会发现此活跃节点处于预提交状态，并会继续执行 3PC 去提交 T。因此提交 T 是安全的。但是，C_{new} 不会片面地提交 T，这样做会在 C_{new} 发生故障时产生和 2PC 中一样的阻塞问题。这就是 C_{new} 重新执行阶段 3 的原因。

考虑没有一个节点得到从 C_i 发出的预提交消息的情况。我们必须考虑三种可能性：

1）C_i 在发生故障前决定提交 T。

2）C_i 在发生故障前决定中止 T。

3）C_i 还没有决定 T 的命运。

我们可以说明这些可能性中的第一个是不可能的，因此中止 T 是安全的。

假设 C_i 决定提交 T，那么必须至少有 K 个节点已经决定预提交 T 并将确认消息发给了 C_i。由于 C_i 发生了故障，并且我们又假设在为事务执行 3PC 协议时至多允许 K 个节点（包括协调器）发生故障，那么发送确认消息的 K 个节点中至少有一个还是活跃的，这样也就至少有一个活跃节点会通知 C_{new} 它收到了预提交消息。因此，如果没有活跃节点收到预提交消息，

C_i 显然就不可能已经作出提交决定，中止 T 因而是真正安全的。由于 C_i 还未决定中止 T 是可能的，因而也就可能去提交 T。但是，要侦测到 C_i 还未决定中止 T 就需要等到 C_i（或其他已收到预提交消息的节点）恢复。所以，如果没有活跃节点收到预提交消息，协议就中止 T。

在前面的讨论中，如果在为事务执行 3PC 协议时允许多于 K 个节点（包括协调器）发生故障，存活下来的参与者就不可能确定 C_i 发生故障前采取的行动，这种情况将迫使阻塞存在直至 C_i 恢复。尽管从这点来看 K 取一个很大的值是最好的，但这却会迫使协调器在决定提交之前需要等待更多的回答——这样就延缓了日常（无故障的）处理。此外，如果少于 K 个参与者（不包括协调器）是活跃的，协调器就不可能完成提交协议，因而导致了阻塞。因此，K 值的选定是至关紧要的，它决定了协议能在多大程度上避免阻塞。

我们所做的不存在网络分割的假设对于我们的讨论来说至关重要。如前所述，通常情况下要区别网络故障和节点故障是不可能的。因此，网络分割可能导致选出两个新的协调器（每一个都认为和自己不在同一分区中的所有节点都发生了故障）。两个协调器的决定可能不一致，导致事务在一些节点上提交而在另一些节点上中止。

18.5.3　协议的比较

尽管 2PC 协议存在阻塞的潜在威胁，但它的使用很广泛。实践中阻塞发生的可能性很低，因而 3PC 协议的额外开销不合算。3PC 在链路故障方面的弱点是实践中的另一个问题。这些不足虽可由网络层次上的协议来克服，但这样的解决方法会增加开销。

这两个协议都可以简化，以减少发送的消息数和记录必须被强制写入稳定存储器的次数。3PC 协议可以进行扩展，以允许多于 K 的故障，只要在新协调器作出决定前发生故障的节点不多于 K 个。文献注解中提到了几种这样的技术。

18.6　协调器选择

我们所给的几种算法需要用到协调器。如果协调器因为它所位于的节点发生故障而停止活动，系统就只有在重新启动另一节点上的一个新协调器情况下才能继续执行。系统可以通过维持协调器的一个备份来实现，这个备份在协调器发生故障时能立即承担起责任。系统也可以在协调器发生故障后选择一个新的协调器。确定协调器的拷贝应在什么地方被重新启动的算法称为选举算法。

18.6.1　备份协调器

备份协调器是这样一个节点，它在完成其他任务的同时，还必须在本地维护足够的信息，使它可以在给分布式系统带来最小限度混乱的情况下担负起协调器的角色。所有发给协调器的消息，协调器和它的备份都会收到。备份协调器和实际协调器执行相同的算法，维护相同的内部状态信息（例如并发协调器使用的封锁表）。协调器和它的备份在功能上的唯一区别在于，备份不会采取任何影响到其他节点的行动，这些行动由实际协调器来进行。

如果备份协调器检测到实际协调器发生了故障，它就担负起协调器的角色。由于故障协调器拥有的所有信息对备份来说都是可以得到的，因而处理可以不中断地继续下去。

备份方式的主要优点在于它立刻将处理继续下去的能力。如果备份不能立刻承担协调器的责任，新近指派的协调器为了执行协调任务，就不得不从系统的所有节点中寻找信息。常常发生故障的协调器就是某些所需信息的唯一来源。这种情况下，中断几个（或所有）活动事务并

在新协调器的控制下重新启动它们可能是必要的。

因此，备份协调器方式避免了分布式系统从协调器故障恢复所需的大量延迟，其缺点在于重复执行协调器任务的开销。此外，协调器和其备份需要有规律地通信，以保证它们的活动是同步的。

简言之，备份协调器方式引入正常处理时的额外开销能使系统可以从协调器故障中迅速恢复。下节将考虑一个低开销的模式，它在故障恢复时需要做的工作稍微多一些。

18.6.2　选举算法

选举算法要求系统中的每个活跃节点都对应一个唯一的标识数字。为使记法方便，我们假设节点 S_i 的标识数字就是 i。同时为了简化讨论，我们假设协调器总位于标识数字最大的节点上。选举算法的目的是为新协调器选择一个节点。因此，当协调器发生故障时，算法必须选出具有最大标识数字的活跃节点。这个数字必须被送到系统中的每个节点。另外，系统还必须提供一种能使从故障中恢复的节点辨认出当前协调器的机制。

各种选举算法通常依据网络结构的不同而不同。本节中，我们给出这些算法中的一个：威逼算法。

假设节点 S_i 发出的请求在预定的时间间隔内没有得到协调器的回答，这种情况下就假设协调器发生了故障，S_i 会试着选举自己来作为新协调器的节点。

节点 S_i 给每个具有更大标识数字的节点发一条选举消息，接着 S_i 就等待一个时间间隔 T，等这些节点中的任何一个作出回答。如果在时间 T 内没有收到任何回答，它就假设具有大于 i 的数字的所有节点都已经发生故障，于是选举自己来作为新协调器的节点，同时给具有小于 i 的标识数字的所有节点发送消息，通知它们它已成为新协调器所位于的节点。

如果 S_i 收到回答，它就开始一个时间间隔 T' 来接收通知自己一个具有更大标识数字的节点已被选中的消息。（别的某个节点正选举自己为协调器，并应该在时间 T' 内报告结果。）如果在 T' 内没有收到消息，就假设具有更大数字的节点发生了故障，节点 S_i 重新开始算法的执行。

故障节点恢复后，马上开始执行相同的算法。如果没有具有更大数字的活跃节点，恢复后的节点就强迫具有较小数字的所有节点让自己成为协调器节点，即使当前有一个具有较小数字的活跃协调器。正因为如此，这个算法称为威逼算法。

18.7　并发控制

本节说明前面所讨论的某些并发控制模式可以怎样进行修改，以适用于分布式环境。

我们假设所有节点都参与到提交协议的执行中，以保证全局事务的原子性。

18.7.1　封锁协议

第 14 章所描述的各种封锁协议都可以用于分布式环境。需要做的唯一改变是锁管理器的实现方式。我们给出对数据能在几个节点上被复制的环境适用的几种可能模式。和第 14 章中一样，我们假设存在共享和排它两种封锁方式。

1. 单一锁管理器方式

系统维护位于选定的单一节点（如 S_i）上的一个单一锁管理器。所有封锁和解锁请求都在节点 S_i 处理。当事务需要给某个数据项上锁时，就向 S_i 发封锁请求。锁管理器决定锁是否能被立即授予。如果锁能被授予，锁管理器就向发起封锁请求的节点发一条告知此结果的消息。否则，请求就被推迟到锁能被授予时，这时才能发送消息给发起封锁请求的节点。事务可

以从该数据项的副本所处的任何一个节点上读取该数据项。在写情况下，所有存在该数据项副本的节点都必须参与到写操作中。

这个模式有如下优点：

• 实现简单。这个模式中处理封锁请求只需要两条消息，处理解锁请求只需要一条消息。

• 死锁处理简单。由于所有封锁和解锁请求在一个节点上处理，第 14 章所讨论的死锁处理算法可以直接运用在此环境中。

这个模式的缺点包括如下：

• 瓶颈。节点 S_i 成为瓶颈，因为所有请求都必须在其上处理。

• 脆弱性。如果 S_i 发生故障，就不再有并发控制器，或者必须将处理停止，或者必须像 18.6 节描述的那样使用恢复模式来使一个新的节点从 S_i 那里接过封锁管理。

2. 多协调器

刚才提到的优缺点之间的一个折衷可以通过多协调器方式达到，在这种方式下锁管理器功能被分布到几个节点上去。

每个锁管理器管理数据项封锁和解锁请求的一个子集，每个锁管理器位于不同的节点，这种方式减少了协调器成为瓶颈的程度，但使死锁处理变得复杂，因为封锁和解锁请求不是在一个节点上处理的。

3. 多数协议

在多数协议中，每个节点维护自己的局部锁管理器，其功能是管理对存储在该节点上的数据项所进行的封锁和解锁请求。当事务希望封锁存储在节点 S_i 且未被复制的数据项 Q 时，请求封锁（以某种封锁方式）的消息就被发送给节点 S_i 上的锁管理器。如果数据项 Q 以不兼容的方式上了锁，请求就被推迟到锁能被授予时。一旦锁管理器确定可以满足封锁请求，就给请求发起者发送消息，告知它的封锁请求已被同意。此模式具有实现简单的优点。处理封锁请求时它需要传输两条消息，处理解锁请求时它需要传输一条消息。但是，死锁处理变得更加复杂。由于封锁请求和解锁请求不再是在单一节点上处理，第 14 章所讨论的死锁处理算法必须修改，我们将在 18.8 节讨论这个问题。

如果数据项 Q 在 n 个不同的节点上被复制，则封锁请求必须送到存储 Q 的 n 个节点当中一半以上的节点上去。每个锁管理器（只要是牵涉到的）都要确定锁是否能被立即授予。和前面一样，对请求的响应被推迟到请求能被满足时。直到获得 Q 的多数副本上的封锁，事务才开始 Q 上的操作。

此模式以一种分散的方式处理复制数据，这样可以避免集中控制的缺点。但是，当数据副本存在时，此模式就被如下缺点所烦扰：

• 实现。多数协议比前面的那些协议在实现上更复杂，处理封锁请求它需要传输 $2(n/2+1)$ 条消息，处理解锁请求它需要传输 $(n/2-1)$ 条消息。

• 死锁处理。由于封锁和解锁请求不是在一个节点上处理，死锁处理算法必须做修改（见 18.8 节）。另外，即使只有一个数据项被封锁时也可能发生死锁。作为例子，我们来看一个有四个节点和完整副本的系统。假设事务 T_1 和 T_2 希望以排它方式封锁数据项 Q。事务 T_1 可能在节点 S_1 和 S_3 封锁成功，事务 T_2 可能在节点 S_2 和 S_4 封锁成功。下一步两个事务都必须等待第三个锁，于是发生死锁。所幸的是，通过要求所有节点按照相同的预定顺序请求数据项副本上的封锁，我们可以比较容易地避免这样的死锁。

4. 有偏协议

有偏协议基于类似多数协议的一个模型。区别在于，共享锁请求受到的待遇比排它锁请求

受到的待遇要优越一些。系统在每个节点上维护一个锁管理器，锁管理器管理存储在该节点上的所有数据项上的锁。共享锁和排它锁的处理是不同的。

- 共享锁。事务需要封锁数据项 Q 时，只需向包含 Q 的副本的一个节点上的锁管理器请求封锁。
- 排它锁。事务需要封锁数据项 Q 时，需要向包含 Q 的副本的所有节点上的锁管理器请求封锁。

和前面一样，对请求的响应被推迟到请求能被满足时。

有偏模式加到 read 操作上的开销比多数协议要小，这是它的一个优点。通常情况下 read 操作频率比 write 操作频率高得多，从而使这种开销的节省尤其明显。但是，附加到写操作上的开销是此模式的一个缺点。此外，有偏协议和多数协议一样具有死锁处理复杂的不足。

5. 主副本

在有数据复制的情况下，我们可以选择一个副本来作为主副本。因此，对每个数据项 Q 而言，Q 的主副本必然仅位于一个节点上，我们称其为 Q 的主节点。

事务需要封锁数据项 Q 时，就在 Q 的主节点上请求封锁。和前面一样，对请求的响应被推迟到请求能被满足时。

这样，主副本就使得复制数据的并发控制可以用与处理未复制数据相类似的方式来处理。这种类似使实现变得简单。然而，当 Q 的主节点发生故障时，即使包含副本的其他节点是可以访问的，Q 也不能被访问了。

18.7.2　时间戳

隐藏在 14.2 节所讨论的时间戳模式后面的基本思想是：每个事务得到一个唯一的时间戳，系统用时间戳来决定串行化顺序。这样在将集中的模式推广到分布的模式时，我们第一个任务就是设计一个产生唯一时间戳的模式。然后，前面我们所给的那些协议就可以直接适用于无复制的环境。

产生唯一时间戳有两种基本方法，一种是集中的，而另一种是分布的。集中模式中选出一个节点来分发时间戳。这个节点可以利用一个逻辑计数器或自己本地的时钟来达到此目的。

分布模式中，每个节点用逻辑计数器或本地时钟产生唯一的局部时间戳。通过将唯一的局部时间戳和唯一的节点标识符（图 18-9）连接起来，我们可以得到唯一的全局时间戳。连接的顺序是很重要的！我们把节点标识符放在最不重要的位置上，以保证一个节点上产生的全局时间戳不至于总是比另一个节点上产生的全局时间戳大。比较这个产生唯一时间戳的技术和前面我们所给的产生唯一名字的技术。

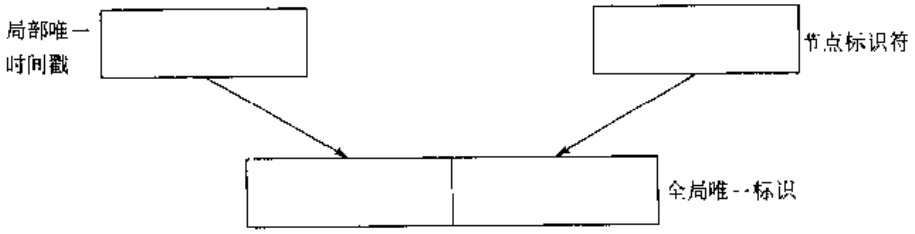


图 18-9　唯一时间戳的产生

如果一个节点产生局部时间戳的速率比别的节点高，这里就还是存在问题。这种情况下，快节点的逻辑计数器将比其他节点的大。因此，快节点所产生的所有时间戳也将比别的节点产生的时间戳大。我们需要一种机制来保证整个系统中局部时间戳能公平地产生。我们在每个节点 S_i 内定义一个产生唯一局部时间戳的逻辑时钟（LC_i）。逻辑时钟可以实现为每产生一个新的局部时间戳就加 1 的计数器。为了保证不同的逻辑时钟同步，我们要求无论什么时候，只要具

有时间戳$<x, y>$的事务 T_i 访问节点 S_i 并且 x 大于 LC_i 的当前值，S_i 就必须增大其逻辑时钟。这种情况下，节点 S_i 将其逻辑时钟增大到值 $x+1$。

如果系统时钟被用来产生时间戳，那么只要任何一个节点的系统时钟都不会运行得过快或过慢，时间戳的分配就是公平的。由于时钟可能不是非常精确，因此必须采用和逻辑时钟情况下所用技术类似的技术，以保证没有时钟走到别的时钟前面或后面。

18.8　死锁处理

第 14 章所给的死锁预防和死锁检测算法只要做相应的修改，就可以用于分布式系统。例如，我们可以通过在系统的数据项当中定义一棵全局树来使用树协议。同样，正如我们在上小节中所看到的那样，时间戳排序方式可以直接用于分布式环境。

死锁预防可能导致不必要的等待和回滚。此外，某些死锁预防技术与不采用这些技术的情况相比，可能还需要将更多的节点牵扯到事务执行中来。

如果我们允许死锁发生然后去依赖死锁检测，那么分布式系统中的主要问题就是决定如何维护等待图。常用的处理这个问题的技术是要求每个节点都维护一个局部等待图。图中节点对应于目前占有或请求该节点上任何数据项的所有事务（局部的或非局部的）。例如，图 18-10 描述了具有两个节点的一个系统，每个节点维护自己的局部等待图。注意，事务 T_2 和 T_3 在两个图中都出现，表明它们在两个节点上都有数据项请求。

对局部事务和数据项来说，局部等待图用普通方法构造。当节点 S_1 上的事务 T_i 需要 S_2 上的资源时，T_i 就向节点 S_2 发出请求消息。如果资源被 T_j 占用，边 $T_i \rightarrow T_j$ 就被插入节点 S_2 的等待图中。

显然，只要某个局部等待图中存在环，就已经发生了死锁。相反地，任何局部等待图中都不存在环并不意味着死锁没有发生。为了说明这个问题，我们来看图 18-10 中的等待图。每个等待图都是无环的，然而系统中却存在死锁，因为局部等待图的并包含了一个环。这个图在图 18-11 中给出。

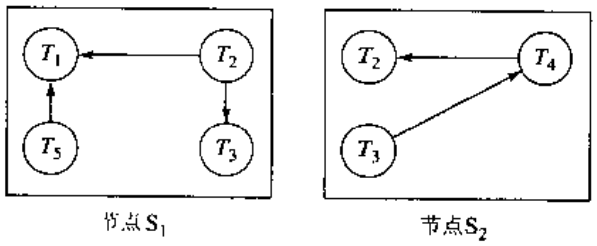


图 18-10　局部等待图

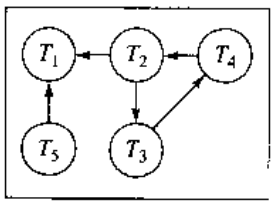


图 18-11　图 18-10 的全局等待图

下两小节将描述分布式系统中常见的几种组织等待图的模式。

18.8.1　集中方式

集中方式中，全局等待图（所有局部图的并）的构造和维护由单一节点——死锁检测协调器来完成。由于系统中存在通信延迟，我们还必须区分两类等待图：*真实图*在任意时刻描述系统真实的但也是不为人知的状态，就像无所不能的观察者所能看到的那样。*构造图*是控制者在控制者算法执行中产生的一个近似。显然，产生的构造图必须使得检测算法无论什么时候被激发，报告的结果都是正确的。报告结果的正确性基于这样一种意义：只要死锁存在，就必须被迅速报告，并且只要系统报告存在死锁，系统就确实处于死锁状态。

全局等待图可以在下述情况下构造：

- 每当一条新边插入局部等待图或从图中删除时。
- 周期的，当局部等待图发生变化达到一定次数时。
- 每当协调器需要激发环检测算法时。

当死锁检测算法被激发时，协调器搜寻它的全局图。如果发现环，就选出一个牺牲品并将其回滚。协调器必须告知所有节点哪个事务被选作牺牲品，于是这些节点将这个作出牺牲的事务回滚。

可以发现这种模式可能产生不必要的回滚，这样的回滚是由以下原因之一导致的：

- 假环可能存在于全局等待图中。作为例子，来看图 18-12 中局部等待图所代表的系统快照。假设 T_2 释放它在节点 S_1 所占用的资源，导致 S_1 上边 $T_1 \rightarrow T_2$ 的删除。事务 T_2 接着请求节点 S_2 上被 T_3 所占用的一个资源，导致 S_2 上边 $T_2 \rightarrow T_3$ 的插入。如果来自 S_2 的消息 insert $T_2 \rightarrow T_3$ 比来自 S_1 的消息 remove $T_1 \rightarrow T_2$ 先到达协调器，那么协调器在 insert 后（但 remove 前）可能会发现假环 $T_1 \rightarrow T_2 \rightarrow T_3$。死锁恢复会被启动，尽管实际上并没有发生死锁。

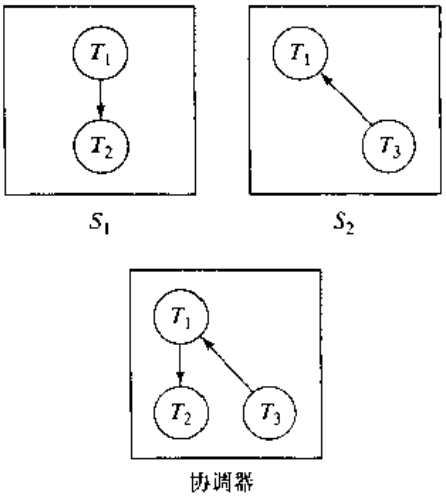


图 18-12　全局等待图中的错环

注意上面的例子在两阶段封锁中是不会发生的。事实上，假环的可能性通常很小，不足以造成性能的严重问题。

- 在死锁确已发生且牺牲品已经选定的情况下仍有可能发生不必要的回滚，这样某个事务就会因和死锁无关的原因而中止。例如，假设图 18-10 中节点 S_1 决定将 T_2 中止。同时，协调器发现了环，选出 T_3 作为牺牲品。此时 T_2 和 T_3 都被回滚，尽管实际上只有 T_2 需要回滚。

18.8.2　全分布方式

全分布方式死锁检测算法中，所有控制者平等地承担检测死锁的责任。在这种模式中，每个节点根据系统的动态行为构造一个代表整个图的一部分的等待图。其思想是，如果死锁存在，则（至少）一个部分图中会存在环。我们给出这样的一个每个节点上构造部分图的算法。

假设一个事务在单独一个节点上运行，并在需要访问非局部数据时向别的节点提出请求。每个节点为局部数据上的锁维护自己的局部等待图，因此如果 T_i 在等待 T_j 拥有的锁，节点 S_k 上就会有边 $T_i \rightarrow T_j$。注意 T_i 和 T_j 可以是非局部的事务。此外，局部等待图有一个附加结点 T_{ex}。如果 a) T_i 在节点 S_k 上执行，且正在等待它在另一节点上所做请求的回答；或 b) T_i 对节点 S_k 来说是非局部的，且在 S_k 上为 T_i 授予了锁，那么图中会存在弧 $T_i \rightarrow T_{ex}$。同样，图中会存在弧 $T_{ex} \rightarrow T_i$，如果 a) T_i 对节点 S_k 来说是非局部的，且正在等待节点 S_k 上的数据上的锁；或 b) T_i 对节点 S_k 来说是局部的，且正在访问外部节点上的数据。

作为例子，考虑图 18-10 中的两个局部等待图。在两个图中都加入 T_{ex}，产生的局部等待图如图 18-13 所示。

如果某个局部等待图中包含不涉及节点 T_{ex} 的环，则系统处于死锁状态。但是，涉及结点

T_{ex}的环只是意味着存在死锁的可能性。为了确定是否真的存在死锁，我们必须激发一个分布式死锁检测算法。

假设在节点 S_i 的局部等待图中包含涉及结点 T_{ex}的环，此环的形式必然是

$$T_{ex} \rightarrow T_{k_1} \rightarrow T_{k_2} \rightarrow \cdots \rightarrow T_{k_n} \rightarrow T_{ex}$$

这意味着 S_i 中事务 T_{k_n} 正在等待获得另一个节点例如 S_j 上的数据项。

在发现这样一个环时，节点 S_i 向节点 S_j 发送包含此环信息的一条死锁检测消息。

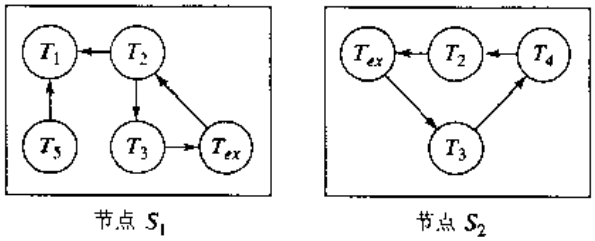


图 18-13　局部等待图

当节点 S_j 收到该死锁检测消息时，它就用所获得的新信息来更新自己的局部等待图。接下来，它在新建的等待图中搜寻不涉及 T_{ex}的环。如果存在这样一个环，就发现了死锁，于是激发适当的恢复模式。如果涉及 T_{ex}的环被发现，那么 S_j 就传输一条死锁检测消息到恰当的节点如 S_k 上去。节点 S_k 重复上述过程。这样，在经过有限轮以后，如果死锁存在则必然会被发现，如果死锁不存在死锁检测计算就会终止。

作为例子，考虑图 18-13 中的局部等待图。假设节点 S_1 发现环

$$T_{ex} \rightarrow T_2 \rightarrow T_3 \rightarrow T_{ex}$$

由于 T_3 正在等待获得节点 S_2 上的数据项，则描述此环的一条死锁检测消息从节点 S_1 传到节点 S_2。当节点 S_2 收到此消息时，就更新自己的局部等待图，得到图 18-14 中的等待图。此图中包含环

$$T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$$

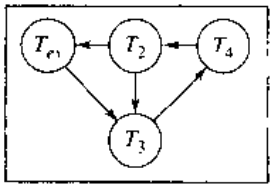


图 18-14　局部等待图

其中不包括结点 T_{ex}。因此，系统处于死锁状态，必须激发适当的恢复模式。

注意，如果节点 S_2 先发现其局部等待图中的环，然后把死锁检测消息传给节点 S_1，结果也是一样的。最坏的情况是，两个节点几乎同时发现环，于是发出两条死锁检测消息，一条从 S_1 到 S_2，另一条从 S_2 到 S_1。其中不必要的消息传输增加了在两个节点上更新等待图以及搜寻等待图的开销。

为了减少消息流量，我们为每个事务 T_i 指定一个唯一的标识符，此标识符用 ID（T_i）表示。当节点 S_k 发现其局部等待图中包含形式如下的涉及 T_{ex} 的环时

$$T_{ex} \rightarrow T_{k_1} \rightarrow T_{k_2} \rightarrow \cdots \rightarrow T_{k_n} \rightarrow T_{ex}$$

那么只在

$$\text{ID}(T_{k_n}) < \text{ID}(T_{k_1})$$

时 S_k 才发送死锁检测消息给别的节点。否则，节点 S_k 继续它正常的执行，而将发起死锁检测算法的任务留给其他节点。

我们再来看图 18-13 所示的由节点 S_1 和节点 S_2 所维护的局部等待图。假设

$$\text{ID}(T_1) < \text{ID}(T_2) < \text{ID}(T_3) < \text{ID}(T_4)$$

再假定两个节点几乎同时发现局部环。节点 S_1 上的环形式为

$$T_{ex} \rightarrow T_2 \rightarrow T_3 \rightarrow T_{ex}$$

由于 ID（T_3）>ID（T_2），节点 S_1 不向节点 S_2 发送死锁消息。

节点 S_2 上的环形式为

$$T_{ex} \rightarrow T_3 \rightarrow T_4 \rightarrow T_2 \rightarrow T_{ex}$$

由于 ID（T_2）＜ID（T_3），节点 S_2 向节点 S_1 发送死锁消息。节点 S_1 在收到此消息时更新其局部等待图，在图中搜寻环，并发现系统处于死锁状态。

18.9　多数据库系统

近年来，开发了许多新的数据库应用系统，它们需要来自多个先前存在的数据库中的数据，这些先前存在的数据库位于一个异构的硬件及软件环境集合中。操纵异构数据库中的信息需要在已有数据库系统之上增加一个软件层，此软件层称为*多数据库系统*。这些局部的数据库系统可能采用不同的逻辑模型以及数据定义和数据操纵语言，而且可能在并发控制和事务管理机制上存在差别。多数据库系统造成逻辑上数据库集成的假象，而不需要将数据库在物理上集成。

将已存在的系统完全集成为一个同构的分布式数据库（到目前为止我们所考虑的那一类分布式环境）通常是困难的甚至是不可能的：

- 技术上的困难。基于已有数据库系统的应用程序的投资可能会是巨大的，而将这些应用进行转化的代价可能就高不可及了。
- 组织上的困难。即使集成在技术上是可行的，在政策上也可能是不可行的，这是由于已有的数据库系统属于不同的公司或组织。

第二种情况下，对多数据库系统来说，允许局部数据库系统在局部数据库及在其数据上运行的事务上保持较高程度的自治是很重要的。

由于这些原因，多数据库系统提供了超出其开销的重要优点。以下我们从数据定义和事务管理的角度，给出构造多数据库系统时所面临的挑战的一个概观。

18.9.1　数据的一致视图

各个 DBMS 可能使用不同的数据模型。也就是说，有的 DBMS 可能采用现代数据模型如关系模型，而另外的 DBMS 可能采用较早的数据模型如网状模型（见附录 A）或层次模型（见附录 B）。

由于多数据库系统会提供单一集成的数据库系统的假象，因此必须使用一个共同的数据模型。我们当然会选择关系模型，并将 SQL 作为共同的查询语言。事实上，现在存在多种允许 SQL 查询的非关系 DBMS 系统。

提供共同的概念模式是另外一个难题。每个局部 DBMS 提供自己的概念模式，多数据库系统必须将这些分离的模式集成为一个共同的模式。模式集成是一个复杂的工作，主要是由于语义的不同。

模式集成绝不仅仅是数据定义语言之间的直接转换。同一属性名可能出现在多个局部 DBMS 中但意义不同。一个系统中使用的数据类型可能在另一个系统中不被支持，因此类型转换可能也不简单。即使是完全相同的数据类型，也会由于数据的物理表示而产生问题，一个系统可能用 ASCII，而另一个系统可能用 EBCDIC。浮点表示可能不同。整数可以用高位在前或低位在前的形式表示。在语义层上，一个系统中表示长度的整数值可能是英寸而另一个系统中可能是毫米，这样出现了整数相等却只能近似表示的尴尬境地（正如浮点数通常的情况那样）。相同的名字可能出现在不同系统的不同语言中。例如，基于美国的系统用“Cologne”表示城市科隆，而基于德国的系统用“Köln”表示。

所有这些看似很小的区别必须被正确记录到共同的全局概念模式中，必须提供转换函数。对于与系统相关的行为，必须为其加注标志（例如，非字母字符的排序在 ASCII 中和 EBCDIC 中是不同的）。另一种方法是将各个数据库转换为共同的格式，但前面我们已经注意到，在不

废弃已有应用程序的前提下这通常是不可行的。

这些情况下，查询处理很复杂。在全局层次上进行查询优化非常困难，通常的解决方案是只依赖局部层次的优化。

18.9.2　事务管理

多数据库系统支持两类事务：

1）局部事务。这些事务由各个局部 DBMS 执行，不在多数据库系统的控制下。

2）全局事务。这些事务在多数据库系统的控制下执行。

多数据库系统虽能知道局部事务在局部节点上运行，但它不知道有哪些具体的事务正在执行或它们要访问什么数据。

要保证每个 DBMS 的局部自治，就不能对局部 DBMS 软件做任何修改。因此，一个节点上的 DBMS 不能通过与另一个节点上的 DBMS 的直接通信来同步几个节点上活动的全局事务的执行。

由于多数据库系统不能对局部事务的执行进行控制，因此每个局部 DBMS 必须使用某种并发控制模式（例如，两阶段封锁或时间戳）来保证它的调度是可串行的。另外，在使用锁的情况下，局部 DBMS 必须能防止局部死锁的可能性。

保证了局部可串行性并不足以保证全局可串行性。作为例子，考虑两个全局事务 T_1 和 T_2，二者都访问分别位于节点 S_1 和 S_2 的两个数据项 A 和 B。假设局部调度是可串行的，但这时仍可能发生节点 S_1 上 T_2 在 T_1 后，节点 S_2 上 T_1 在 T_2 后的情况，这将导致全局调度不可串行，但事实上，即使不存在全局事务的并发（即一个全局事务只在前一事务提交或中止后才提交），局部可串行性仍不足以保证全局可串行性（见练习 18.20）。

即使有全局事务和局部事务的并发执行情况，很多协议在多数据库系统中仍可以保证一致性。其中一些基于施加足够多的条件来保证全局可串行性；另外一些只保证某种比可串行性要弱的一致性，但实现这样的一致性可以用一些限制较少的方法。我们考虑后一类模式中的一种：*两级可串行性*。20.4 节描述在不保证可串行性时达到一致性的进一步的方法，文献注解中提到了别的一些方法。

1. *两级可串行性*

两级可串行性（2LSR）在系统的两个级别上保证可串行性：

- 每个局部 DBMS 在其局部事务间（包括作为全局事务一部分的那些事务）保证局部可串行性。
- 多数据库系统仅仅在全局事务间保证可串行性——忽略由局部事务引起的顺序问题。

这两个可串行性级别中的每一级都易于实施。局部系统已经提供可串行性保证，因此第一个要求容易达到。第二个要求只应用于全局调度中不出现局部事务的部分。因此，MBDS 可以用标准的并发控制技术（技术的具体选择是无关紧要的）来保证第二个要求。

2LSR 的两个要求并不足以保证全局可串行性。然而，在基于 2LSR 的方式下，我们采用比可串行性弱的要求，称为*强正确性*：

1）保持由一系列一致性约束说明的一致性。

2）保证每个事务所读的数据项集合是一致的。

可以证明，对事务行为做特定限制，再加上 2LSR，就足以保证强正确性（尽管不必保证可串行性）。我们列举这些限制中的几个。

在每个协议中，我们区别局部和全局数据。局部数据项属于某个节点且在该节点单独的控

制下。注意不同节点的局部数据项之间不可能存在任何一致性约束。全局数据项属于多数据库系统，并且尽管它们可能处于不同的节点，它们都在多数据库系统的控制下。

全局读协议允许全局事务读局部数据项，但不允许更新，同时不允许局部事务对全局数据进行任何访问。全局读协议在下列条件都成立时可保证强正确性：

1）局部事务只访问局部数据项。

2）全局事务可以访问全局数据项，可以读局部数据项（尽管不能写局部数据项）。

3）局部数据项与全局数据项之间没有任何一致性约束。

局部读协议允许局部事务对全局数据进行读访问，但不允许全局事务对局部数据的任何访问。在这个协议中，我们需要引入值依赖的概念。如果事务在一个节点上所写数据项的值依赖于它在另一个节点上所读数据项的值，则此事务存在值依赖。

局部读协议在下列条件都成立时可保证强正确性：

1）局部事务可以访问局部数据项，可以读存储在该节点上的全局数据项（尽管不能写全局数据项）。

2）全局事务只访问全局数据项。

3）任何事务都不能存在值依赖。

全局读写/局部读协议从数据访问这一方面来说是我们所考虑协议中最慷慨的。该协议允许全局事务读写局部数据，并允许局部事物读全局数据。但是，它既要求局部读协议中值依赖的条件，又要求全局读协议中局部数据和全局数据间不存在一致性约束的条件。

全局读写/局部读协议在下列条件都成立时可保证强正确性：

1）局部事务可以访问局部数据项，可以读存储在该节点上的全局数据项（尽管不能写全局数据项）。

2）全局事务既能访问全局数据项又能访问局部数据项（即它们可以读写任何数据）。

3）局部数据项与全局数据项之间没有任何一致性约束。

4）任何事务都不能存在值依赖。

2. 全局可串行性的保证

早期多数据库系统限制全局事务必须是只读的，这样它们虽避免了全局事务给数据带来不一致的可能性，但却不足以保证全局可串行性。说明这样的全局调度是真正可行的并设计一个保证全局可串行性的模式的任务，我们留给读者来完成（练习 18.21）。

有一些通用的模式可以在更新事务和只读事物同时执行的环境中保证全局可串行性，其中的几个模式基于标签的思想。每个 DBMS 上都建立称为标签的特殊数据项。访问某个节点上数据的全局事务必须去写该节点上的标签。这一要求保证全局事务在它们访问的每个节点上直接发生冲突。此外，全局事务管理器可以通过控制标签被访问的次序来控制全局事务被串行化的顺序。文献注解中谈到了这样的模式。

如果需要在各节点上不发生直接局部冲突的环境中保证全局可串行性，必须做一些关于局部 DBMS 允许调度的假设。例如，如果局部调度保证提交顺序和串行化顺序总保持一致，我们就可以通过只控制事务提交的顺序来保证可串行性。

保证全局可串行性的模式可能存在过度限制并发的问题。它们之所以很可能存在这样的问题，是因为大多数事务把 SQL 语句递交给下面的 DBMS，而不是递交一个个 read、write、commit、和 abort 步骤。尽管这样的假设下仍有可能保证全局可串行性，但由于并发程度太低，以至于使别的模式，如前面讨论的两级可串行性技术，成为更具吸引力的选择。

18.10　总结

分布式数据库系统由节点集合构成，每个节点维护一个局部的数据库系统。各个节点能够处理局部事务，即那些只访问该节点上数据的事务。此外，节点可以参与全局事务的执行，全局事务访问多个节点上的数据。全局事务的执行需要在节点之间进行通信。

建立分布式数据库系统的原因有几点，包括数据共享，可靠性和可用性，以及查询处理的加速。但是，伴随这些优点而来的还有一些缺点，包括更高的软件开发代价、程序错误存在的更大可能性，以及增大的处理开销。分布式数据库系统的主要缺点是为了在节点间保证正确的协调而需要增加的复杂性。

关于在分布式数据库中存储关系有几个问题，包括复制和分片。系统应尽量减小用户对关系如何存储需要知道的程度。

分布式系统可能遭受与集中式系统所面对的故障类型相同的故障。但是，分布式环境中还有另外一些我们需要处理的故障，包括节点故障、链路故障、消息丢失，以及网络分割。分布式的恢复模式的设计需要考虑这些故障中的每个问题。因此，如果系统要做到强壮，就必须能检测所有这些故障，重新配置自己以使计算继续，并在处理器或链路修复后进行恢复。

如果要保证原子性，执行事务 T 的所有节点就必须在 T 执行的最终结果上取得一致。T 必须要么在所有节点上均提交，要么在所有节点上均中止。为了保证这一特性，T 的事务协调器必须执行一个提交协议。使用最广泛的提交协议是两阶段提交。

两阶段协议可能导致阻塞——事务的命运必须等到故障节点（协调器）恢复后才能确定的状态。为了避免阻塞，我们可以使用三阶段提交协议。

集中式系统中所用的各种并发控制模式修改后可用于分布式环境。对于封锁协议的情况，需做的唯一改变是锁管理器的实现方式，这里可以有多种不同的方式，可能要用到一个或多个协调器。如果采用分布的方式，复制数据就必须被特别对待。处理复制数据的协议包括多数协议、有偏协议和主副本协议。在时间戳和验证模式情况下，所需的唯一修改是设计一个产生全局唯一时间戳的机制。通过将局部时间戳和节点标识符相连接，或者在具有较大时间戳的消息到来时增大局部时钟，我们可以设计出这样一个机制。

分布式环境中处理死锁的主要方法是死锁检测，主要的问题是要决定等待图该怎样维护。组织等待图的方式包括集中的方式、层次的方式，以及全分布的方式。

一些分布式算法需要使用协调器。如果协调器由于它所位于的节点发生故障而停止工作，系统只有通过在另一节点上重新启动协调器的新备份来继续其执行。系统可以通过维护协调器的一个备份来完成这一任务，备份协调器在协调器发生故障时可以立刻担负起责任。另一种方法是在协调器发生故障后选出新协调器。确定协调器的新备份应该在什么地方重新启动的算法称为选举算法。

多数据库系统提供给新数据库应用一个可以访问来自多个先前存在的、位于多种异构软硬件环境中的数据库中数据的环境。局部数据库系统可能采用不同的逻辑模型以及数据定义和数据操纵语言，并可能在并发控制和事务管理机制上存在差别。多数据库系统造成逻辑上数据库集成的假象，而不需要物理上数据库的集成。

习题

18.1　讨论集中式数据库和分布式数据库相对各自的优点。

18.2　解释以下说法的区别：分片透明性，副本透明性，位置透明性。

18.3　为局域网设计的分布式数据库系统与为广域网设计的分布式数据库系统应有怎样的区别？

18.4　数据复制和分片在什么时候有用？解释你的答案。

18.5　解释透明性和自治的概念。为什么从人的因素的角度来说这些概念是我们需要的。

18.6　考虑按 *plant-number* 水平分片的关系：

employee (*name*, *address*, *salary*, *plant-number*)

假设每个片段有两个副本：一个存在 New York 节点，另一个存在工厂的节点。为以下在 San Jose 节点提出的查询设计一个好的处理策略。

(a) 找出 Boca 厂的所有员工。

(b) 找出所有员工的平均工资。

(c) 找出以下每个节点报酬最高的员工：Toronto、Edmonton、Vancouver、Montreal。

(d) 找出公司中报酬最低的员工

18.7　考虑关系：

employee (*name*, *address*, *salary*, *plant-number*)

machine (*machine-number*, *type*, *plant-number*)

假设 *employee* 关系按 *plant-number* 水平分片，且每个片段本地地存在于它所对应的工厂节点。假设整个 *machine* 关系存在 Armonk 节点。为以下查询设计一个好的处理策略。

(a) 找出包含机器号 1130 的工厂的所有员工。

(b) 找出包含机器类型“milling machine”的工厂的所有员工。

(c) 找出 Almaden 工厂的所有机器。

(d) 找出 *employee* ⋈ *machine*。

18.8　对练习 18.7 中每个策略，说明你选择的策略对下列因素的依赖：

(a) 提出查询的节点。

(b) 需要结果的节点。

18.9　对下列关系计算 $r \ltimes s$：

r

A	*B*	*C*
1	2	3
4	5	6
1	2	4
5	3	2
8	9	7

s

C	*D*	*E*
3	4	5
3	6	8
2	3	2
1	4	1
1	2	3

18.10　$r_i \ltimes r_j$ 必然等于 $r_j \ltimes r_i$ 吗？什么条件下 $r_i \ltimes r_j = r_j \ltimes r_i$？

18.11　要建立一个强壮的分布式系统，你必须知道会出现什么类型的故障。

(a) 列出分布式系统中可能的故障类型。

(b) 你的上小题列表中哪些故障也可能出现在集中式系统中？

18.12　考虑为事务执行 2PC 时出现的故障。对你在练习 18.11a 中列出的每种可能的故障，解释尽管存在这些故障可能，2PC 怎样保证事务的原子性。

18.13　对 3PC 重复练习 18.12。

18.14　列出 3PC 不能处理的故障类型。描述这些故障怎样被低级别协议处理。

18.15　在节点按层次组织的情况下，考虑分布式死锁检测算法。每个节点检查本节点中的局

部死锁和涉及到层次结构中后代节点的全局死锁。给出此算法的详细描述，证明此算法能检测所有死锁。比较这种层次模式和集中模式、完全分布式模式的优缺点。

18.16　考虑一个有两个节点 A 和 B 的分布式系统。A 能区别出下列情况吗？

- B 崩溃。
- A 和 B 间的连接断开。
- B 极度过载，响应时间比正常情况下长 100 倍。

18.17　如果把第 14 章中多粒度协议的分布式版本应用于分布式数据库，DAG 图的根节点可能成为瓶颈。假设我们修改协议如下：

- 根节点中只允许有意向锁。
- 所有事务被自动地授予在根节点上所有可能的意向锁。

18.18　讨论我们在 18.7.2 节中给出的用来产生全局唯一的时间戳的两个模型的优缺点。

18.19　考虑下面的死锁检测算法。当节点 S_1 的事务 T_i 请求节点 S_3 的事务 T_j 的资源时，发送带时间戳 n 的请求消息，边 (T_i, T_j, n) 被插入到 S_1 的局部等待图中。仅当 T_j 接受到请求消息并且不能立即给予资源时，边 (T_i, T_j, n) 才被插入到 S_3 的局部等待图中。同一节点上事务 T_i 到 T_j 的请求按照通常的方式处理，边 (T_i, T_j) 不带时间戳。中央协调器通过发送启动消息给系统的每个节点来激发检测算法。

一旦接收到这个消息，节点就发送它自己的局部等待图给协调器。注意此图中包含该节点所拥有的关于真实图状态的所有局部信息。等待图反映了节点的瞬间状态，但是它没有与任何其他节点同步。

当控制者从每个节点都接收到回答后，它构造图如下：

- 对于系统中每个事务，图中包含一个顶点。
- 该图中有边 (T_i, T_j) 当且仅当
  - 在某个等待图中有边 (T_i, T_j)。
  - 边 (T_i, T_j, n)（对某个 n）在不止一个等待图中出现。

证明，如果在构造的图中有环，那么系统处于死锁状态，并且如果在构造的图中没有环，那么系统在算法执行开始时没有处于死锁状态。

18.20　考虑一个多数据库系统，它保证任何时候至多只有一个全局事务在活动，并且每个局部节点保证局部串行性。

(a) 给出多数据库系统能用来保证任何时候至多只有一个全局活动事务的方法。

(b) 用例子说明尽管有这些假设，仍可能产生非串行性的全局调度。

18.21　考虑一个多数据库系统，其中，每个局部节点保证局部串行性，所有全局事务都是只读的。

(a) 用例子说明在这样的系统中，可能产生非串行性执行。

(b) 证明可以用标签模式保证全局串行性。

文献注解

Tanenbaum [1996] 和 Halsall [1992] 讨论了计算机网络。Rothnie 和 Goodman [1977] 综述了分布式数据库系统的主要问题。Bray [1982]、Date [1983]、Ceri 和 Pelagatti [1984]、Ozsu 和 Valduriez [1991] 都是相关教材。

Wong [1977]、Epstein 等 [1978]、Hevner 和 Yao [1979]、Epstein 和 Stonebraker [1980]、Apers 等 [1983]、Ceri 和 Pelagatti [1983] 讨论了分布式查询处理。Wong [1983]、Selinger 和

Adiba [1980]、Daniels 等 [1982] 讨论了 R* (System R 的分布式版本) 采用的分布式查询处理的方法。Mackert 和 Lohman [1986] 提供了 R* 中查询处理算法的性能评价。性能评价结果也可以用来检验 R* 中查询优化器所使用的代价模型。有关半连接的理论结果由 Bernstein 和 Chiu [1981]、Chiu 和 Ho [1980]、Bernstein 和 Goodman [1981b]、Kambayashi 等 [1982] 提供。

分布式数据库中事务概念的实现由 Gray [1981]、Traiger 等 [1982]、Spector 和 Schwarz [1983]、Eppinger 等 [1991] 给出。2PC 协议由 Lampson 和 Sturgis [1976]、Gray [1978] 提出。三阶段提交协议来自 Skeen [1981]。Mohan 和 Lindsay [1983] 讨论了 2PC 协议的两种修正版：假设提交和假设中止。对事务命运它们通过定义默认假设减少了 2PC 的开销。

18.6.2 节中给出的威逼算法来自 Garcia-Molina [1982]。分布式时钟同步在 Lamport [1978] 中讨论。

讨论分布式并发控制的论文由 Rosenkrantz 等 [1978]、Bernstein 等 [1978, 1980a]、Menasce 等 [1980]、Bernstein 和 Goodman [1980a, 1981a, 1982]、Garcia-Molina 和 Wiederhold [1982] 提供。R* 的事务管理器在 Mohan 等 [1986] 中描述。

基于投票概念的数据复制的并发控制由 Gifford [1979] 和 Thomas [1979] 给出。分布式并发控制模式的验证技术在 Schlageter [1981]、Ceri 和 Owicki [1983]、Bassiouni [1988] 中描述。有关基于语义的事务管理技术的讨论在 Garcia-Molina [1983]、Kumar 和 Stonebraker [1988] 中给出。近来，复制数据的并发更新问题已经在数据仓库环境中作为一个重要的研究课题出现。Gray 等 [1996] 讨论了在此环境中的问题。

Attar 等 [1984] 讨论了带数据复制的数据库系统中分布式恢复中事务的作用。Kohler [1981] 综述了分布式数据库系统中的恢复技术。

分布式死锁检测算法由 Gray [1978]、Rosenkrantz 等 [1978]、Menasce 和 Muntz [1979]、Gligor 和 Shattuck [1980]、Chandy 和 Misra [1982]、Chandy 等 [1983] 给出。Knapp [1987] 概括了分布式死锁检测的文献。18.8.2 节中给出的算法取自 Obermark [1982]，习题 18.19 取自 Stuart 等 [1984]。

多数据库系统中的事务处理在 Breitbart [1990]、Breibart 等 [1991, 1992]、Soparkar 等 [1991]、Mehrotra 等 [1992a, 1992b] 中讨论。标签模式在 Georgakopoulos 等 [1994] 中给出，2LSR 在 Mehrotra 等 [1991] 中介绍，更早的叫作准可串行性的方法在 Du 和 Elmagarmid [1989] 中给出。

第 19 章　特别的话题

前面的章节中，我们论述了数据库设计和实现的基本原则。本章以及后续章节中，我们简要地论述数据库系统领域中一些特别的话题。在第 20 章里，我们讨论高级事务处理模式。在第 21 章里，我们考虑数据库系统的新应用，以及它们给数据库系统设计带来的挑战。本章末尾的文献注解为各个话题提供了参考文献，可以将这些参考文献作为更详尽地了解这些话题的起点。

19.1　安全性和完整性

数据库中存储的数据需要受到保护，以防止未授权访问、恶意破坏或修改以及意外引入的不一致性。在第 6 章中，我们知道了如何定义完整性约束。在第 7 章中，我们谈到了为便于完整性约束的检查，数据库应怎样设计。在第 13、14 和 15 章里，我们看到了如何在有故障、崩溃和并发处理潜在异常等存在的情况下保持完整性。在第 17 章和第 18 章里，我们看到了如何在并行系统和分布式系统中保持完整性。到目前为止，我们只考虑了如何防止数据完整性意外地遭受破坏。本节要看一看数据误使用或故意使数据不一致的一些方式，然后给出防止这些情况的机制。

19.1.1　安全性和完整性违例

对数据库的误使用可以分为有意（恶意）的或意外的。数据一致性的意外破坏可能是由于以下原因：

- 在事务处理过程中发生崩溃。
- 对数据库的并发访问所导致的异常。
- 在几个计算机上分布数据所导致的异常。
- 违反事务保证数据一致性这一假设的逻辑错误。

防止数据一致性的意外破坏比防止对数据库的恶意访问要容易。下面是一些恶意访问的形式：

- 未经授权读取数据（窃取信息）。
- 未经授权修改数据。
- 未经授权消除数据。

完全杜绝对数据库的恶意滥用是不可能的，但我们可以使那些企图在没有适当授权情况下访问数据库的作恶者付出足够高的代价，以阻止绝大多数（如果不是全部）这样的访问企图。数据库安全性通常指保护数据库不受恶意访问，完整性指避免意外地破坏一致性。实践中，安全性和完整性之间的分界线并不总是很清晰的。当这两个概念间的区别不是非常重要时，我们将用名词安全性同时指代安全性和完整性。

为了保护数据库，必须在几个层次上采取安全性措施：

- 物理层。计算机系统所位于的节点（一个或多个）必须物理上受到保护，以防止入侵者强行闯入或暗中潜入。

• 人际层。对用户的授权必须格外小心，以减少授权用户接受贿赂或其他好处而给入侵者提供访问机会的可能性。

• 操作系统层。不管数据库系统多安全，操作系统安全性方面的弱点总是可能成为对数据库进行未授权访问的一种手段。

• 网络层。由于几乎所有的数据库系统都允许通过终端或网络进行远程访问，网络软件的软件层安全性和物理安全性一样重要，不管在因特网上还是在企业私有的网络内。

• 数据库系统层。数据库系统的某些用户获得的授权可能只允许他访问数据库中有限的部分，而另外一些用户获得的授权可能允许他提出查询，但不允许他修改数据。保证这样的授权限制不被违反是数据库系统的责任。

为了保证数据库安全，我们必须在上述所有层次上进行安全性维护。如果较低层次上（物理层或人际层）安全性存在缺陷，高层安全性措施即使很严格也可能被绕过。

许多应用中，为保持数据库完整性和安全性而付出极大努力是很值得的。包含薪水或其他财务数据的大型数据库对窃贼来说是颇具吸引力的目标。包含公司运作数据的数据库可能是不道德竞争对手的兴趣所在。此外，这些数据的丢失，不管是偶然的还是故意的，都会严重破坏公司的运作能力。

本节剩下内容将在数据库系统层次上讨论安全性。物理层和人际层安全性虽然很重要，但不在我们所要讨论的范围内。操作系统的安全性在几个层次上实现，从访问系统所需口令到在系统中运行的并发进程间的隔离。文件系统也提供了一定程度上的保护。文献注解中给出的有关操作系统的文献覆盖了这里提到的所有话题。最后，随着因特网从学术研究平台发展成为国际电子商务的基础，网络层安全性已经获得了广泛的关注。文献注解中列出了覆盖网络安全性所有基本原则的教科书。尽管本章中的概念对所有数据模型来说都是适用的，但我们对安全性的讨论将按照关系数据模型来进行。

19.1.2　授权

用户在数据库的各个部分上可以有几种形式的授权，其中：

• Read 授权允许读取数据，但不允许修改数据。

• Insert 授权允许插入新数据，但不允许修改已经存在的数据。

• Update 授权允许修改数据，但不允许删除数据。

• Delete 授权允许删除数据。

用户可以获得上面所有授权类型或其中一部分的组合，也可以根本不获得任何授权。

除了以上几种对数据访问的授权外，用户还可以获得修改数据库模式的授权：

• Index 授权允许创建和删除索引。

• Resource 授权允许创建新的关系。

• Alteration 授权允许增加或删除关系中的属性。

• Drop 授权允许删除关系。

drop 授权和 delete 授权的区别在于 delete 授权只允许对元组进行删除。如果用户删除了关系中的所有元组，关系仍然存在，只不过是空的。如果关系被删除，那么关系就不再存在。

创建新关系的能力通过 resource 授权来控制。具有 resource 授权的用户在创建新关系后自动获得该关系上的所有特权。

index 授权看起来似乎不必要，因为索引的创建和删除不会改变关系中的数据。事实上，索引是提高性能的一种结构。但是，索引也会消耗空间，并且所有数据库的修改都需要更新索

引。如果 index 授权被授予所有用户，那么执行更新操作的用户倾向于删除索引，而提出查询的用户倾向于创建大量索引。为使数据库管理员能够管理系统资源的使用，我们有必要将索引的创建作为一种权限来看待。

最大的授权形式是给数据库管理员的。数据库管理员可以给新用户授权，可以重构数据库，等等。这一授权形式类似于操作系统中提供给超级用户或操作员的授权形式。

19.1.3　授权与视图

在第 3 章里，我们介绍了视图的概念，视图是给用户提供个性化数据库模型的一种手段。视图可以隐藏用户不需看见的数据。视图隐藏数据的能力既可以简化系统的使用，又可以实现安全性。由于用户只关注那些感兴趣的数据，系统的使用就得到简化。尽管用户可能不被允许直接访问某个关系，但用户可能被允许通过一个视图访问该关系的一部分。因此，关系级的安全性和视图级的安全性可以结合起来，以限制用户只能访问所需数据。

在银行例子中，假设有一个需要知道在各分支机构有贷款的所有客户姓名的职员。该职员不能看到与客户具体贷款相关的信息。因此，该职员对 *loan* 关系的直接访问必须被禁止。但是，如果他要访问这些信息，就必须能够访问视图 cust-loan，这一视图由所有客户姓名及其贷款分支机构构成。此视图可以用 SQL 定义如下：

create view *cust-loan* **as**
(**select** *branch-name*, *customer-name*
from *borrower*, *loan*
where *borrower*.*loan-number* = *loan*.*loan-number*)

假设该职员提出如下 SQL 查询：

select *
from *cust-loan*

显然，该职员允许看到此查询的结果。但是，当查询处理器将此查询转换为数据库中实际关系上的查询时，我们得到的是 *borrower* 和 *loan* 上的查询。因此，对职员查询授权的检查必须在查询处理开始以前进行。

创建视图并不需要 resource 授权。创建视图的用户不一定能获得该视图上的所有权限。这些用户得到的权限不会为他提供超过原有授权的其他授权。例如，用来定义视图的关系上没有 update 授权的用户不能得到相应视图上的 update 授权。如果用户创建一个视图，而此用户在该视图上不能获得任何授权，这样的视图创建请求将被拒绝。在使用 *cust-loan* 视图的例子中，视图的创建者必须在关系 *borrower* 和 *loan* 上都具有 read 授权。

19.1.4　权限的授予

获得了某种形式的授权的用户可以将此授权传递给其他用户。但是，授权怎样在用户间传递我们必须格外小心，以保证这样的授权在未来的某个时候可以被收回。

作为例子，考虑银行数据库中 *loan* 关系上 update 权限的授予。假设最初数据库管理员将 *loan* 上的 update 权限授给用户 U_1，U_2 和 U_3，他们接下来又可以将这一授权传递给其他用户。授权从一个用户到另一个用户的传递可以表示为授权图，图的结点是用户。如果用户 U_i 将 *loan* 上的 update 权限授给用户 U_j，则图中包含边 $U_i \rightarrow U_j$。图的根是数据库管理员。图 19-1 给出了一个示例图。注意用户 U_1 和 U_2 都给 U_5 授了权，而 U_4 只从 U_1 处获得授权。

用户具有授权当且仅当存在从授权图的根（即代表数据库管理员的结点）到代表该用户的结点的路径。

设数据库管理员决定收回用户 U_1 的授权。由于用户 U_4 从 U_1 处获得授权，因此其权限也应该被收回。由于用户 U_5 既从 U_1 处又从 U_2 处获得授权，且数据库管理员没有从 U_2 处收回 *loan* 上的 update 授权，因此 U_5 继续拥有 *loan* 上的 update 授权。如果 U_2 最终从 U_5 处收回授权，则 U_5 失去授权。

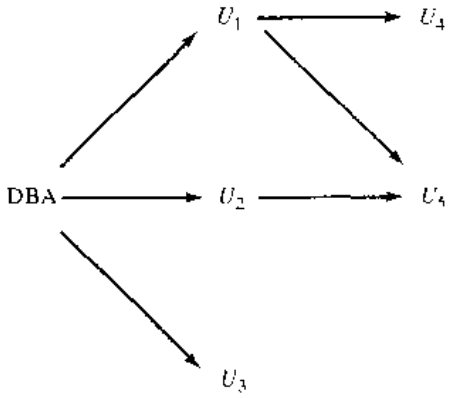


图 19-1　权限授予图

一对狡猾的用户可能企图通过相互授权来破坏权限回收规则，如图 19-2a 所示。如果数据库管理员从 U_2 收回权限，U_2 保留了通过 U_3 获得的授权，如图 19-2b 所示。如果权限接着从 U_3 处收回，U_3 似乎保留了通过 U_2 获得的授权，如图 19-2c 所示。然而，当数据库管理员从 U_3 处收回权限时，从 U_3 到 U_2 的边以及从 U_2 到 U_3 的边已不再是从数据库管理员开始的路径的一部分了。我们要求授权图中的所有边都必须是从数据库管理员开始的某条路径的一部分。这两条边将被删除，产生的授权图如图 19-3 所示。

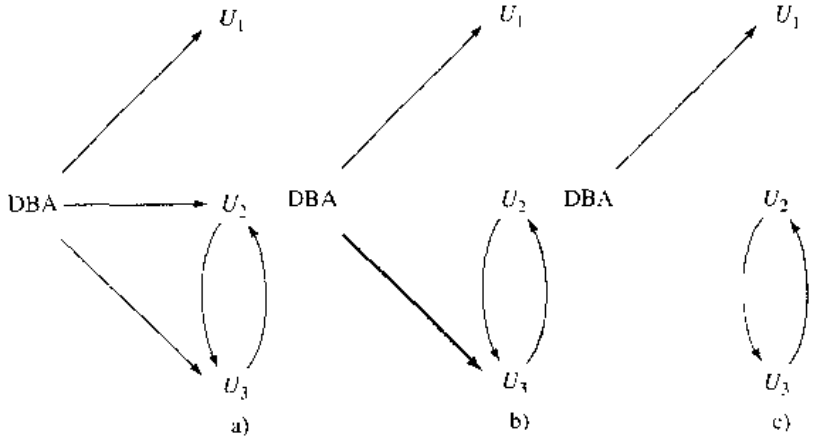


图 19-2　破坏权限回收的企图

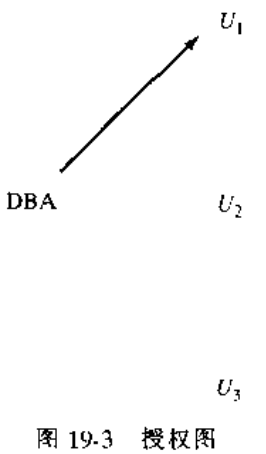


图 19-3　授权图

19.1.5　在 SQL 中进行安全性说明

SQL 数据定义语言中包含了权限授予和回收的命令。SQL 标准包括 delete、insert、select 和 update 权限。select 权限对应于 read 权限。SQL 还包括了 references 权限，用来限制用户在创建关系时定义外码的能力。如果即将创建的关系中包含参照其他关系的属性的外码，那么用户必须在这些属性上具有 references 权限。references 权限之所以会成为一个有用的特征，其原因比较微妙，本小节的后面将作出解释。

grant 语句用来授予权限。这一语句的基本形式如下：

grant ＜权限列表＞ on ＜关系名或视图名＞ to ＜用户列表＞

权限列表使得一个命令可以授予多个权限。下面的 grant 语句授予用户 U_1、U_2 和 U_3 关系 *branch* 上的 select 权限：

grant select on *branch* **to** U_1, U_2, U_3

update 授权既可以在关系的所有属性上进行，又可以只在某些属性上进行。如果 grant 语句中包括 update 授权，则将被授予 update 权限的属性列表可以出现在紧跟关键字 update 的括号中。如果省略属性列表，则关系的所有属性上均被授予 update 权限。下面的 grant 语句授予用

户 U_1，U_2 和 U_3 关系 *loan* 的 *amount* 属性上的 update 权限：

grant update (*amount*) **on** *loan* **to** U_1，U_2，U_3

在 SQL-92 中，insert 权限也可以说明属性列表，对关系做插入时只允许申明这些属性，其余各属性或者赋缺省值（如果该属性有缺省值定义），或者置为空。

SQL 的 references 权限在特定属性上授予，其方式类似于讲述 update 权限时所示方式。下面的 grant 语句允许用户 U_1 创建作为外码参照关系 *branch* 的码 *branch-name* 的关系：

grant references (*branch-name*) **on** *branch* **to** U_1

初看起来似乎完全没有理由防止用户创建参照另一关系的外码。但是，请不要忘记第 6 章中外码约束限制了被参照关系上的删除和更新操作。前面例子中，如果 U_1 在关系 r 中创建了一个参照 *branch* 关系的 *branch-name* 属性的外码，接着在关系 r 中插入与 Perryridge 分支机构相关的一个元组，那么 U_1 就再也不可能从 *branch* 关系中将 Perryridge 分支机构删除，除非同时修改了 r 关系。这样，U_1 所做的外码定义限制了其他用户以后的活动，因此需要有 references 权限。

权限 all privileges 可以用作所有允许权限的缩写形式。同样，用户名 public 指系统所有当前用户和将来的用户。SQL-92 还包括 usage 权限，允许用户使用特定的域（不要忘了域对应于编程语言中类型的概念，并且可以由用户定义）。

缺省情况下，在 SQL 中被授予权限的用户不允许将该权限授予其他用户。如果我们希望去授予权限并允许接受授权者将权限传递给其他用户，我们需要将 with grant option 子句附加在适当的 grant 命令后。例如，如果我们希望 U_1 在 *branch* 上有 select 权限并允许 U_1 将这一权限授予他人，我们写作

grant select on *branch* **to** U_1 **with grant option**

我们使用 revoke 语句来收回授权。这一语句的形式与 grant 几乎完全相同：

revoke ＜权限列表＞ on ＜关系名或视图名＞

from ＜用户列表＞ [restrict|cascade]

因此，要收回前面我们所授予的那些权限，我们写作

revoke select on *branch* **from** U_1，U_2，U_3 **cascade**

revoke update (*amount*) **on** *loan* **from** U_1，U_2，U_3

revoke references (*branch-name*) **on** *branch* **from** U_1

正如我们在 19.1.4 节所看到的那样，从一个用户那里收回权限可能导致其他用户也失去该权限，这一行为称作级联收回。revoke 语句也可以申明 restrict：

revoke select on *branch* **from** U_1，U_2，U_3 **restrict**

这种情况下，如果存在任何级联收回，就返回一个错误，并且不执行收回动作。下面的 revoke 语句仅仅收回 grant option，而并不是真正收回 select 特权。

revoke grant option for select on *branch* **from** U_1

SQL-92 标准定义了数据库模式的基本授权机制：只有模式的所有者才能对模式进行修改。因此，模式的修改（如关系的创建和删除、增加或去掉关系中的属性，以及增加或去掉索引）只能由模式的所有者来执行。一些数据库实现对数据库模式有更加强有力的授权机制，类似于前面讨论的那些机制，但这些机制都不是标准的。

19.1.6　加密

对高度敏感的数据而言，数据库系统的各种授权规则也许不能提供充分的保护。在这种情

况下，数据可以被加密。加密数据是不可能被读出的，除非读数据的人知道如何对加密数据进行解码（解密）。

加密数据的技术数不胜数。简单的加密技术也许不能提供足够的安全性，因为未授权用户破译编码会比较容易。来看一个存在缺陷的加密技术的例子，用字母表中的下一个字母替代每个字母的情况。这样，

Perryridge

就变成了

Qfsszsjehf

如果未授权用户仅仅看到“Qfsszsjehf”，可能还得不到破译编码的充足信息。但是，如果侵入者看到大量加密后的分支机构名称后，就可能使用有关字符的相对频率（如 e 比 f 更常用）的统计数据来猜测所做的替代。

好的加密技术具有如下性质：

- 对授权用户来说，加密数据和解密数据相对简单。
- 加密模式不应依赖于算法的保密，而是依赖于被称作密钥的算法参数。
- 对入侵者来说，确定密钥是极其困难的。

数据加密标准（DES）这一加密方法在密钥的基础上既进行字符替换，又对字符顺序进行重新排列。要使这一模式发挥作用，就必须把密钥通过某种安全机制提供给授权用户。但这一要求又是最大的缺陷，因为该模式不会比传递密钥的安全机制更安全。

公钥加密是另一种模式，它克服了我们在 DES 中所面临的某些问题。此模式基于两个密钥：一个公钥和一个私钥。每个用户 U_i 有一个公钥 E_i 和一个私钥 D_i。所有公钥都被公布，即任何人都可以知道。每个私钥则只能被拥有它的用户知道。如果用户 U_1 想要存储加密数据，就通过公钥 E_1 来对数据加密，这些加密数据的解密需要私钥 D_1。

因为用来加密的密钥对所有用户公开，所以我们有可能利用这一模式安全地交换信息。如果用户 U_1 希望与 U_2 共享数据，那么 U_1 就用 U_2 的公钥 E_2 来加密数据。由于只有用户 U_2 知道如何对数据解密，因此信息的传输是安全的。

要使公钥加密发挥作用，必须有能被公开的、却又不易让人推断出解密模式的加密模式。换言之，在已知公钥时想要推断私钥非常困难。这样的模式确实存在，并且建立在如下基础之上：

- 存在测试一个数字是否为素数的高效算法。
- 不知道求一个数字的素数因子的有效算法。

出于对这一模式的考虑，数据被看作一组整数。我们通过计算两个大素数 P_1 和 P_2 的积来产生公钥。私钥由（P_1，P_2）对构成，并且在只知道乘积 P_1P_2 时使用解密算法不会成功。由于公开的只是乘积 P_1P_2，未授权用户为了窃取数据就需要对 P_1P_2 做因数分解。通过将 P_1 和 P_2 选得足够大（超过 100 位），我们可以使对 P_1P_2 做因数分解的代价高不可攀（即使在最快的计算机上，计算时间也需要用多少年这样的级别来衡量）。

关于公钥加密的细节以及这一技术的性质的数学证明，文献注解中给出了有关的参考书目。

尽管使用上述模式的公钥加密很安全，但是它的计算代价很高。安全通信所采用的一种混合模式如下：DES 密钥通过公钥加密模式被交换，而其后的传输数据用 DES 加密。

密码系统的另一个有趣的应用是在数字签名中。数字签名扮演的是物理签名的电子化角色，用于验证数据的真实性。数字签名的一种模式是逆向使用公钥加密：用私钥来加密数据，

加密后的数据可以公开。所有人都可以用公钥来解码，但没有私钥的人就不能产生编码数据。这样，我们就可以验证数据是否真正由公开宣称产生这些数据的人所产生。

19.1.7　统计数据库

假设在银行例子中，银行希望提供给外人一定的访问权限，即在保证数据只能进行统计性研究（均值、中值等等）并且不会泄露各个客户信息的前提下，允许外人访问银行数据库。本节，我们探讨允许对数据进行统计目的使用的同时保证个人隐私的困难所在。

一种可能的方式是只允许外人访问提供相关聚集数据（如总和或计数）的视图，但是特殊情况是这种统计数据库的薄弱环节。例如，假设一个用户请求获得所有居住在 Smalltown 的客户的帐户余额总和。如果恰好只有一个客户居住在 Smalltown，那么系统就泄露了有关个人的信息。当然，只有在用户知道仅有一个客户居住在 Smalltown 时才会发生这种对安全性的破坏。但是，这一信息很容易通过统计查询“找出居住在 Smalltown 的客户数目”获得。

要处理上述对安全性可能产生的破坏，系统中使用的一种简单方法是拒绝涉及到的个体数目少于某个预定数字的所有查询。假设这个预定数字是 n，一个在银行中有帐户的恶意用户可以用两个查询找出某个人的帐户余额。假设此用户想要找出 Rollo 的帐户余额，该用户于是选择 $n-1$ 个客户并提出两个查询去计算：

- x。这 $n-1$ 个客户及此恶意用户的帐户余额总和。
- y。这 $n-1$ 个客户及 Rollo 的帐户余额总和。

Rollo 的帐户余额为

$$y-x+\text{该恶意用户的余额}$$

这个例子所暴露出的致命弱点在于两个查询引用了很多相同的数据项。两个查询共同的数据项数目称为它们的交。

因此，除了要求查询引用的数据至少 n 个个体以外，我们还要求任意两个查询的交都不大于 m。通过调整 n 和 m，可以增加用户在确定关于个体数据时所面临的困难，但我们并不能使之成为不可能事件。

这两个限制并不能排除特别狡猾的一些查询泄露个体数据的可能性。但是，如果除此之外再加上所有查询只能计算总和、计数或均值这样的限制，并且如果恶意用户只知道自己的数值，我们可以证明恶意用户确定关于别人的数据至少需要 $1+(n-2)/m$ 次查询。这一断言的证明不属本书的范围，文献注解中给出了参考文献。尽管如此，这一事实并不能让我们完全放心。虽然我们可以将一个用户允许的查询数限制在 $1+(n-2)/m$ 次以下，但若两个恶意用户串通一气，还是可以造成数据泄露。

另一种安全措施是数据污染，或者说是对用于回答查询的数据进行随机歪曲。进行这样的歪曲时必须保证不破坏回答的统计意义。与此类似的一种技术是对查询本身进行随机的修改。这两种技术的目标都牵涉到准确性和安全性之间的权衡。

不管对统计数据采用什么安全手段，恶意用户总有可能确定个体数据。但是，好的技术可以使金钱和时间上的耗费足够高而成为制止恶意用户的因素。

19.2　标准化

标准定义软件系统的接口，例如，标准可以定义编程语言的语法和语义、应用程序接口中的函数，甚至数据模型（如面向对象数据库标准）。现在的数据库系统非常复杂，常常由多个

部分组成，这些部分的产生是相互独立的，而彼此之间又需要交互。例如，客户端程序的产生可以不依赖后端系统，但二者必须能彼此交互。具有多个异构数据库系统的公司可能需要在数据库之间交换数据。在这样的情况下，标准发挥着重要的作用。

正式标准由标准化组织或行业组织通过一个公开的程序制定。占统治地位的产品有时成为事实标准，因为它们没有通过任何正规的公认程序而被作为标准广泛接受。一些正式标准，如SQL-92 的许多方面，是引导市场的预见性标准，它们先定义一些特性，然后厂商才实现这些特性。而另外的情况下，标准或标准的一部分是被动标准，因为它们试图把一些已经有厂商实现的、甚至已经变成事实标准的特性标准化。SQL-89 在很多方面都是被动的，因为它对 IBM SAA SQL 标准以及其他数据库中已经出现的某些特征（如完整性检查）进行标准化。

正式标准委员会成员中通常包括厂商代表、来自用户组织的成员、来自标准化组织如国际标准化组织（ISO）和美国国家标准局（ANSI）的成员，以及来自专业群体如电子电气工程师协会（IEEE）的成员。正式标准委员会定期集会，成员对标准中的特性提出增加或修改的建议。在一段（通常是被延长的）时间的讨论、修改建议以及全体审阅后，成员对接受还是拒绝某一特性进行投票。标准制定和实现后，经过一段时间，标准的缺点会变得明显，新得需求变得显著，于是更新标准的过程开始了，并且通常几年后会公布标准的新版本。这样的循环每几年重复一次，直到最后（也许是很多年以后）标准在技术上变得无关紧要或失去其用户基础。

网状数据库标准 DBTG CODASYL 是数据库领域中早期的正式标准之一，其系统表述由 Database Task Group 给出。IBM 数据库产品曾建立了事实标准，因为 IBM 占据了数据库市场的很大份额。随着关系数据库的发展，许多新的竞争者加入到数据库产业中来，因此产生了对标准的需求。

由于 SQL 是使用最广泛的查询语言，因而在对其进行标准化上已经做过很多工作。ANSI 和 ISO 以及其他数据库厂商在这样的工作中担任了领导角色。SQL-86 是最初的版本，IBM 系统应用体系结构（SAA）SQL 标准于 1987 年公布。随着人们意识到需要更多特性，正式 SQL 标准更新后的版本 SQL-89 和 SQL-92 被制定出来。SQL 标准的下一版本（目前暂时称作 SQL-3）现在正在制定中，这一标准将给 SQL 增加多种面向对象的特性。

现在的数据库系统通常由后端服务器和提供图形用户界面的前端客户构成。为了使用户在选择前端和后端时有最大的灵活性，客户-服务器互联标准已经发展起来。个人计算机端市场上使用最广的客户-服务器互联标准是 Microsoft 定义的*开放数据库互联*（ODBC）标准，这是一个用于客户-服务器应用程序接口（API）的标准。

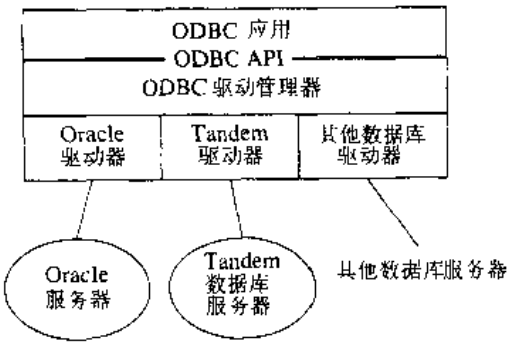


图 19-4　ODBC 体系结构

ODBC 基于 X/Open 行业协会和 SQL Access Group 制定的 SQL 调用层接口（CLI）标准，但已有所扩展。图 19-4 对 ODBC 的作用进行了图解说明。ODBC 主要定义客户程序用以连接到数据库系统和发出 SQL 命令的 API。客户（如图形用户界面、统计包或电子表格）可以利用同一 ODBC API 来连接到任何支持 ODBC 的数据库服务器。每个数据库系统提供一个驱动程序，这一驱动程序受客户端的 ODBC 驱动程序管理器控制，负责与服务器连接和通信以及进行所有必要的数据和查询格式转换。

ODBC API 定义一个 CLI、一个 SQL 语法定义以及关于允许的 CLI 调用序列的规则。该标

准还定义了 CLI 和 SQL 语法的一致级别。例如，CLI 核心级包括连接数据库、准备和执行 SQL 语句、取回结果或状态值以及管理事务的命令。下一个级别（第 1 级）要求对目录信息检索以及其他一些核心级 CLI 之上的特性提供支持，第 2 级要求进一步的特性，如发送和检索参数值数组以及检索更加详细的目录信息。

ODBC 允许一个客户同时连接到多个数据源并在这些数据源之间进行切换，但各个数据源上的事务是独立的，ODBC 不支持两阶段提交。

分布式系统提供一个比客户 - 服务器系统所提供的环境更一般的环境。X/Open 协会也制定了数据库互操作标准，即 X/Open XA 标准。这些标准定义了遵循标准的数据库应提供的事务管理原语（如事务开始、提交、中止和准备提交），事务管理器可以激发这些原语，利用两阶段提交实现分布式事务。XA 标准独立于数据模型以及客户和数据库间交换数据的具体接口。因此，我们可以利用 XA 协议实现分布式事务系统，在这样的系统中，一个事务可以既访问关系数据库又访问面向对象的数据库，而事务管理器通过两阶段提交保证全局一致性。

目前，主要是 OODB 厂商在促进面向对象数据库领域中的标准。Object Database Management Group（ODMG）是由 OODB 厂商组成的一个团体，致力于标准化 OODB 的数据模型和语言接口。ODMG 所定义的 C++ 语言接口已在第 8 章中讨论，ODMG 还定义了 Smalltalk 接口。

Object Management Group（OMG）是由多家公司组成的一个协会，其目标是制定基于面向对象模型的分布式软件应用的标准体系结构。OMG 提出了对象管理体系结构（OMA）参考模型。对象请求代理（ORB）是 OMA 体系机构中的一个部件，它为分布式对象提供透明消息分发，因而对象的物理位置无关紧要。通用对象请求代理体系结构（CORBA）为 ORB 提供了详细的说明，还包括了用于定义数据交换中所用数据类型的接口描述语言（IDL）。IDL 帮助提供对数据转换的支持，当数据在数据表示不同的系统间传递时需要这样的支持。IDL 主要是一种面向对象的数据定义语言，人们正致力于建立 IDL 和 ODMG 对象定义语言（ODL）的统一版本。

本章末尾的文献注解给出了关于本节中所提到的标准的详细描述的参考文献。

19.3 性能基准程序

由于数据库服务器变得越来越标准化，产品性能成为不同厂商产品的主要差异因素。性能基准程序是用于量化软件系统性能的任务集。

19.3.1 任务集

由于大多数软件系统（如数据库）都很复杂，不同厂商的实现中会有许多不同。因此，针对不同的任务它们的性能存在显著差异。某一任务可能是这个系统最有效，而另一任务可能是那个系统最有效。因而仅用一个任务来量化系统性能通常是不够的，而要用一个称作性能基准程序的任务集合来度量。

将来自多个任务的性能值结合起来的工作必须小心进行。假设我们有两个任务，分别为 T_1 和 T_2，并且我们用给定时间如 1 秒内所运行的各类事务的数量来作为系统吞吐量的度量。假设系统 A 运行 T_1 时是每秒 99 个事务，运行 T_2 时是每秒 1 个事务。同样，设系统 B 运行 T_1 和 T_2 时都是每秒 50 个事务。我们还假设工作负载是两类事务的一个等比例混合。

如果我们取这两对数（即 99 和 1、50 和 50）的均值，看起来似乎这两个系统的性能一样。但是，以这种形式取平均是错误的——如果我们每种类型运行 50 个事务，系统 A 要用 50.5

秒来完成，而系统 B 只要 2 秒就可以完成。

上例表明，如果不止一类事务，对性能进行简单的度量可能导致错误发生。综合性能值的正确方法是采用工作负载的完成时间，而不是采用每类事务吞吐量的平均值。这样，对于具体工作负载，我们可以用每秒事务数准确地计算系统性能。因此，系统 A 平均每个事务需要用 50.5/100 即 0.505 秒，而系统 B 平均每个事务需要用 0.02 秒。用吞吐量来描述，系统 A 的平均速度为每秒 1.98 个事务，而系统 B 为每秒 50 个事务。假设每类事务发生的可能性相等，则综合不同事务类型吞吐量的正确方法是求这些吞吐量的调和平均数。n 个吞吐量 t_1，…，t_n 的调和平均数定义如下：

$$\frac{n}{\left(\frac{1}{t_1}+\frac{1}{t_2}+\cdots+\frac{1}{t_n}\right)}$$

本例中，系统 A 吞吐量的调和平均数为 1.98，而系统 B 吞吐量的调和平均数为 50。因此，对于由这两种示例事务类型等量混合而成的工作负载而言，系统 B 大约比系统 A 快 25 倍。

19.3.2　数据库应用类型

联机事务处理（OLTP）和决策支持（包括联机分析处理［OLAP］）是数据库系统所处理的两大类应用。这两类任务有着不同的需求。一方面，支持频繁的事务更新需要有高的并发性和能加速事务提交处理的好的技术。另一方面，决策支持又需要有好的查询求值算法和查询优化。一些数据库系统的体系结构适于事务处理，而另一些数据库系统（如 Teradata DBC 并行数据库系统系列）的体系结构适于决策支持。另外还有一些厂商力争在这两类任务间取得平衡。

通常应用中将事务处理和决策支持的需求混合。因此，对一个应用来说，哪一种数据库系统最好取决于该应用中这两类需求的混合比例。

假设我们分别有这两类应用的吞吐量数值，并且当前的应用是这两类事务的混合。由于事务间的冲突，即使求吞吐量数值的调和平均数时我们也必须小心。例如，一个耗时的决策支持事务可能获得很多锁，这会阻碍所有事务更新的进行。吞吐量的调和平均数只能用在事务互不冲突时。

19.3.3　TPC 基准程序

事务处理性能委员会（TPC）是一个组织，该组织定义了一系列数据库系统基准程序的标准。

该系列中的第一个标准是 TPC-A 基准程序，它于 1989 年制定。TPC-A 基准程序测量更新密集的数据库环境下的性能，这样的环境在联机事务处理（OLTP）应用中很典型。在这种环境中有多个联机终端会话，有大量的磁盘 I/O，并且需要事务的完整性。此基准程序模拟一个典型的银行应用，使用单一类型的事务，该事务模拟银行出纳员办理取款和存款业务。它更新几个关系（例如银行余额、出纳员余额和客户余额），并且在审计追踪关系中增加一条记录。此外，该基准程序还和终端进行通信，以真实地模拟系统的端到端性能。

TPC 基准程序定义得非常详细，它定义了关系的集合和各元组的大小。它对于关系中元组数量的定义并没有使用固定的数值，而是使用了宣称的每秒事务数目的倍数，以反映更高的事务执行速度可能会与更多的帐户相联系。用来度量性能的是吞吐量，用事务数/秒（TPS）表示。在测量性能时，系统必须提供一定范围内的响应时间，这样，高吞吐量的获得就不能以很长的响应时间为代价。进一步，对商业应用来说，代价是非常重要的。因此，TPC 基准程序还

测量用代价/TPS 表示的性能。一个大系统可能有高的事务数/秒值，但可能代价也很高（即代价/TPS 值很高）。此外，公司宣称其系统的 TPC 基准程序测试值时，必须有外部的审查，以确保系统如实地遵循了基准程序的定义，包括对事务 ACID 特性的完全支持。

设计 TPC-B 基准程序是为了测试数据库系统及运行它的操作系统的核心性能。它去除了 TPC-A 基准程序中处理用户、通信和终端的部分，以集中测试后端的数据库服务器。TPC-B 基准程序相对而言使用不太广泛。

设计 TPC-C 基准程序是为了模拟比 TPC-A 基准程序模拟的系统更复杂的系统。TPC-C 基准程序主要考虑订单录入环境中的主要活动，例如输入和分发订单、记录付款、检查订单状态和监控库存量。

设计 TPC-D 基准程序是为了测试数据库系统在决策支持查询中的性能。决策支持系统现在正变得越来越重要。TPC-A、TPC-B 和 TPC-C 基准程序测量事务处理工作负载下的性能，它们不能用来测量决策支持查询中的性能。TPC-D 中的“D”可以理解为代表决策支持。此基准程序由 17 个查询构成的集合组成，以模拟决策支持系统。这些查询必须用 SQL 实现，一些查询使用复杂的 SQL 特性，例如聚集。

19.3.4　OODB 基准程序

OODB 中应用的性质与典型事务处理应用的性质不同。因此，对 OODB 提出了另外一组基准程序。对象操作基准程序，也即版本 1（通常称作 OO1 基准程序），是一个早期的建议。现在 OO7 基准程序都已被广泛使用。该基准程序遵循的原则与 TPC 基准程序遵循的不同。TPC 基准程序提供一或两个数值，用平均事务数/秒和事务数/（秒 * 美元）表示；OO7 基准程序提供一组数值，包括每一类操作各自的基准程序值。这样做的原因是至今仍不清楚什么是典型的 OODB 事务，清楚的只是这样的事务将执行特定的操作，例如遍历一组相互关联的对象，或检索一个类中的所有对象，但是这些操作的混合比例并不十分清楚。因此，该基准程序为每种操作提供各自的数值，这些数值可以用适当的方式组合，这取决于特定的应用。

19.4　性能调整

系统性能调整涉及对各个参数以及设计中所做选择的调整，以提高系统在特定应用下的性能。数据库系统设计的各方面（范围从高层次的方面如模式和事务设计，到数据库参数如缓冲区大小，直到低层次的硬件问题如磁盘数目）都影响着应用的性能。每个方面都可以进行调整，以提高性能。

19.4.1　瓶颈的位置

大多数系统的性能（至少在调优以前）由于一个或几个部件的性能而受到限制，这样的部件称为瓶颈。例如，一个程序若有 80% 的时间花在代码中的一个小循环上，而其余 20% 的时间分散在剩余的代码上，这个小循环就是一个瓶颈。提高非瓶颈部件的性能对于提高系统总体性能没什么帮助，上述例子中，提高剩余代码的速度不可能使总体速度提高 20% 以上，而提高瓶颈循环的速度在最好的情况下可以将总体速度提高约 80%。

因此，对系统调优时，我们首先必须尽力找出瓶颈是什么，然后通过提高导致瓶颈的部件的性能来消除瓶颈。在一个瓶颈被消除后，可能另一个部件又成为瓶颈。在平衡较好的系统中，任何单个部件都不会是瓶颈。如果系统中存在瓶颈，不构成瓶颈的那些部件就不能被充分使用，因而应该用性能较低、价格较便宜的部件来代替这些部件。

对简单的程序来说，代码各个部分花费的时间决定了总体执行时间。然而，数据库系统要复杂得多，可以用排队系统来建模。事务要向数据库系统请求各种服务，首先需要进入一个服务器进程，接着执行过程中需要读磁盘，需要 CPU 周期，还需要并发控制所需的锁。每个这样的服务都有一个队列与之相联，而小事务可能把它们大部分的时间都花费在队列（尤其是磁盘 I/O 队列）等待上。图 19-5 举例说明了数据库系统中的一些队列。

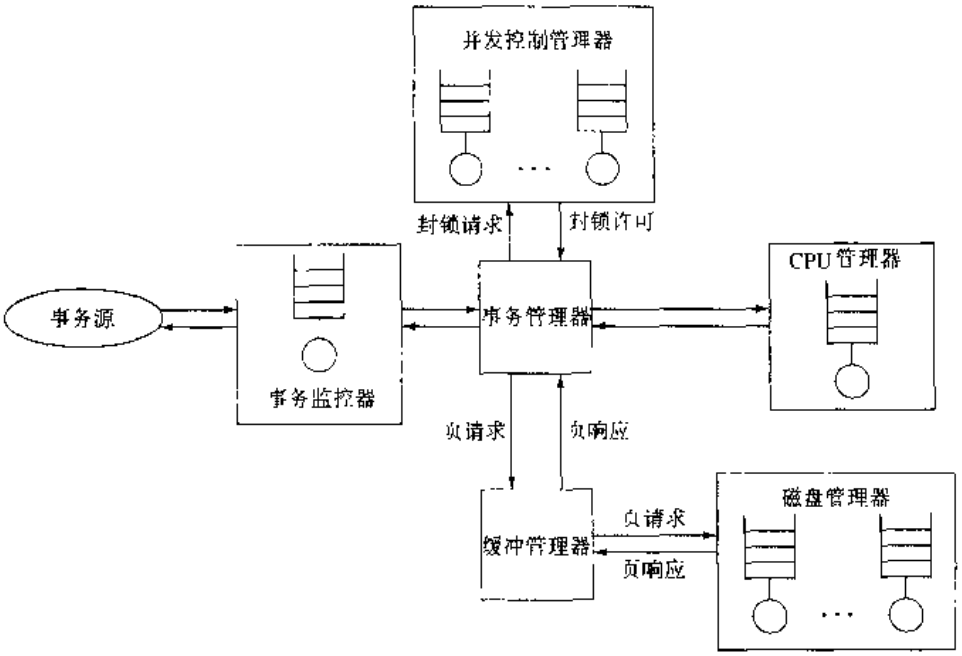


图 19-5　数据库系统中的队列

由于数据库系统中存在大量队列，数据库系统中的瓶颈常常表现为某个服务的队列很长，或者某个服务的利用率很高，这两者是等价的。如果请求的分布很均匀，并且为一个请求服务的时间小于等于下一服务到来的时间，那么每个请求都会发现资源是空闲的，因此可以立即开始执行，而不需等待。不幸的是，数据库系统中请求的到达不那么均匀，而总是随机的。

如果一个资源（如磁盘）的利用率较低，那么当请求提出时，该资源很可能是空闲的，这时请求的等待时间为 0。假设请求的到达均匀地随机分布，队列长度（以及相应的等待时间）随利用率成指数增长，那么当利用率接近 100% 时，队列长度急剧增加，从而导致等待时间过长。由此资源的利用率应保持足够低，以使队列长度很短。经验表明，利用率在大约 70% 时较好，而超过 90% 就太大，因为这会带来显著的迟延。要了解关于排队系统理论（通常称作排队论）的更多知识，请参阅文献注解中的所给书目。

19.4.2　可调参数

数据库管理员可在三个层次上对数据库系统进行调整，最低是在硬件层上，这一层上调整系统的选项包括：如果磁盘 I/O 是瓶颈，则增加磁盘或使用 RAID 系统；如果磁盘缓冲是瓶颈，则增加内存；如果 CPU 使用是瓶颈，则改用更快的处理器。

第二层由数据库系统参数组成，例如缓冲区大小和检查点间隔等。可调数据库系统参数的精确集合同具体的数据库系统相关。大多数数据库系统手册给出了哪些数据库系统参数可调以及用户如何选择参数值方面的信息。设计良好的数据库系统自动进行尽可能多的调整，以减轻用户或数据库管理员的负担。例如，很多数据库系统具有固定大小的缓冲区，而缓冲区的大小

又是可调的。如果系统能够自动根据对页缺失率等指示信息的观察来调整缓冲区大小，那么用户就不必为调整缓冲区大小而烦扰。

第三层是高层设计，包括模式和事务。你可以调整模式的设计、创建的索引以及执行的事务来提高性能。这一层的调整相对而言对系统的依赖较小。以后内容中，我们将讨论高层设计的性能调整。

三层调整相互影响，对系统进行调整时，我们必须将三者结合起来考虑。例如，高层所做的调整可能导致硬件瓶颈从磁盘系统移到 CPU，或从 CUP 移到磁盘系统。

19.4.3 模式的调整

在所采用的范式的约束下，对关系进行垂直划分是可能的。例如，考虑关系 *account*，它的模式为

account（*branch-name*，*account-number*，*balance*）

其中 *account-number* 是码。在所采用的范式（BCNF 和第三范式）的约束下，我们可以将 *account* 关系划分为两个关系，划分如下：

account-branch（*account-number*，*branch-name*）

account-balance（*account-number*，*balance*）

由于 *account-number* 是码，这两种表示在逻辑上等价，但是它们的性能特征不同。

如果大多数对帐户信息的访问都只查看 *account-number* 和 *balance*，那么这些访问可以在 *account-balance* 关系上运行，由于不读取 *branch-name* 属性，这些访问很可能会稍微加快。由于相同的原因，缓冲区中的 *account-balance* 元组将比对应的 *account* 元组多，这也可以提高性能。如果 *branch-name* 属性很大，那么效果尤为明显。因此，在这种情况下，由 *account-branch* 和 *account-balance* 构成的模式比由 *account* 构成的模式更为优越。

另一方面，如果大多数对帐户信息的访问都同时需要 *balance* 和 *branch-name*，则使用 *account* 关系会更优越，因为这样可以避免 *account-balance* 和 *account-branch* 的连接开销。另外，存储开销也会较小，因为这时只有一个关系，而属性 *account-number* 不被重复。

提高性能的另一个技巧是存储非规范化的关系，例如存储 *account* 和 *depositor* 的连接，其中分支机构名称和余额信息会为每个帐户持有者重复一次。这样每当更新发生时，维护关系一致性需要付出更大的代价。但是，读取客户姓名及对应余额的查询速度将会提高，因为 *account* 和 *depositor* 的连接已经预先计算好了。如果这样的查询频繁执行，并且需要尽快完成，那么使用非规范化关系的存储不无裨益。一些数据库系统还支持另一种方式，即将连接中能够匹配的记录聚集存放在同一磁盘页上，从而使连接计算能够高效地进行。

19.4.4 索引的调整

我们可以调整系统中的索引来提高性能。如果查询是瓶颈，那么我们常常可以通过在关系上创建适当的索引来加速查询。如果更新是瓶颈，那么可能是索引太多，这些索引在关系被更新时也必须被更新，删除索引可以加速更新。

索引类型的选择很重要。一些数据库系统支持不同类型的索引，例如散列索引和 B 树索引。如果经常使用范围查询，那么 B 树索引比散列索引更适合。是否使索引成为聚集索引是另一个可调参数。一个关系上只允许一个聚集索引，通常能使尽可能多的查询和更新受益的索引应选为聚集索引。

19.4.5 事务的调整

只读事务和更新事务都可以进行调整。在过去，很多数据库系统的优化器并不是特别好，所以查询的写法对查询执行有很大影响，因此对性能也有很大影响。今天的优化器很先进，甚至能对写得极差的查询进行转换，然后有效地执行查询。但是，优化器在它们能做什么上有其限制。大多数系统提供某种机制来找出查询的最佳执行计划，并用它来调整查询。

在嵌入式 SQL 中，如果某查询经常以不同的参数值执行，那么将多个调用组合成一个面向集合的只执行一次的查询可能会有所帮助。在客户－服务器系统中，SQL 查询的通信开销可能很高，因而在这样的系统中将嵌入式 SQL 调用组合起来更加有利。例如，考虑如下程序，此程序一步步地处理某个列表上的所有部门，对各个部门来说嵌入式 SQL 查询是通过在关系 *expenses*（*date*，*employee*，*department*，*amount*）上使用 group by 结构来找出该部门的总开销。如果 *expenses* 关系在 *department* 上没有聚集索引，那么每个这样的查询都会导致对关系的扫描。我们并不这样做，而是只用一个嵌入式 SQL 查询来找出各个部门的总开销，并把它们存储在一个临时表中，该查询只用一次扫描就能求值。接下来，相关部门就可以在这个（假设小得多的）临时表中查询。

客户－服务器系统中，用于减少通信和 SQL 编译开销的另一个广泛使用的技术是使用可以预先编译的存储过程，这时查询以过程的形式存储在服务器上，客户可以激活这些存储过程，而不需传递完整的查询。

不同类型事务的并发执行有时会由于锁冲突而导致性能低下。例如，让我们考虑银行数据库。一天中，大量小的更新事务几乎是连续不断地执行。假设同时还有一个计算分支机构统计数据的大查询。如果该查询在一个关系上进行扫描，那么在它运行时就可能阻塞所有对该关系的更新，这对系统性能来说将是灾难性的影响。这样的大查询最好在更新很少或不存在时执行，例如在夜间。

长的更新事务会带来系统日志的性能问题和从系统崩溃中恢复所花时间的性能问题。如果事务进行很多更新，那么在事务完成前系统日志就可能已满了，这时事务不得不回滚。如果事务更新运行很长时间（尽管只有少量更新），那么当日志系统设计不好时就可能阻碍日志中旧的部分的删除，这样的阻碍也可能导致日志满。

为了避免这样的问题，很多数据库系统对单个事务所能进行的更新数目做了严格限制。即使系统不做这样的限制，将大的更新事物在可能的情况下分成一组较小的更新事务通常是有益的。例如，给大公司的每个员工加薪的事务可以分为一系列小事务，每个小事务对一个小范围内的员工进行更新，这样的事务称为小型批处理事务。但是，小型批处理事务的使用必须小心。首先，如果在员工集合上存在并发更新，小事务集合的结果可能和单个大事务的结果不等。其次，如果故障发生，就可能有一些员工的工资由于事务的提交得到增长，而其他员工的工资却没有增加。为了避免这样的问题，当系统从故障中恢复后，我们应立即执行批处理中剩余的事务。

19.4.6 性能模拟

为了在数据库系统安装前就能测试其性能，我们可以构造数据库系统的性能模拟模型。图 19-5 中的每个服务，如 CPU、各个磁盘、缓冲和并发控制等，在模型中都将被模拟。模拟模型并不模拟服务的细节，而只是抓住每个服务的某些方面，例如服务时间，即查询处理开始到结束所需要的时间。因此，模型可以仅仅基于平均访问时间来模拟磁盘访问。

由于服务请求通常需要排队等待，在模拟模型中每个服务有一个与之对应的队列。请求到达后就排到队列中去等待，并根据服务策略（如先来先服务）获得服务。不同服务如 CPU 和磁盘的模型在概念上并行地操作，以表示实际系统中这些子系统并行操作的事实。

一旦事务处理的模拟模型建立起来，我们就可以在其上进行很多试验。用到达速率不同的模拟事务做试验，可以发现系统在不同负载情况下表现如何。我们还可以通过改变各服务的服务时间来做试验，以发现性能对各服务的敏感程度。系统参数可以发生变化，这样就可以在模拟模型上进行性能调整。

19.5　数据库中的时间

数据库对其外部现实世界中某些方面的状态建模。通常，数据库只对现实世界的一个状态（当前状态）建模，而不存储有关过去状态的信息，除了有时候为审计追踪而存储以外。当现实世界的状态发生改变时，数据库得到更新，但有关旧状态的信息就丢失了。但是，在很多应用中，存储和检索有关过去状态的信息非常重要。例如，病人数据库中必须存储有关病人病历的信息。工厂的监控系统为了做分析，可能需要存储工厂中传感器当前读数和过去读数的信息。存储了现实世界随时间变化的状态信息的数据库称作*时态数据库*。

在考虑数据库系统中的时间问题时，必须区分系统所测量的时间和现实世界中所观察的时间。事实的*有效时间*是现实世界中使这一事实为真的时间段的集合。事实的*事务时间*是事实处在数据库系统中的时间段的集合。后一个时间基于事务的串行化顺序，由系统自动产生。注意，有效时间段是现实世界的概念，不能由系统自动产生而必须提供给系统。

*时态关系*是其中每个元组都与某一时间相关联的关系，这一时间表明该元组何时为真，可以是有效时间或事务时间。当然，同时存储有效时间和事务时间也是可以的，这时关系被称为*二时态关系*。图 19-6 给出了时态关系的一个例子。为了简化表示，同每个元组相关联的时间段只有一个，因此，对于元组为真的互不相交的时间段中的每一个，元组都有对应的一次表示。这里的时间段是用 *from* 和 *to* 这一对属性来表示的，真正实现时将会有一个包含这两个字段的结构类型，也许叫作 Interval。注意，有的元组在 *to* 时间列上是“*”，这些星号表明这些元组在 *to* 时间列的值改变之前一直为真，因此，当前这些元组为真。尽管时间以正文形式给出，但是时间的存储可以采取更紧凑的形式，例如可以存储从固定的某一天，固定的某个时刻起经过的秒数，而这一秒数可以转换回通常的正文形式。

branch-name	*account-number*	*balance*	*from*		*to*	
Downtown	A-101	500	94/1/1	9:00	94/1/24	11:30
Downtown	A-101	100	94/1/24	11:30	*	
Mianus	A-215	700	95/6/2	15:30	95/8/8	10:00
Mianus	A-215	900	95/8/8	10:00	95/9/5	8:00
Mianus	A-215	700	95/9/5	8:00	*	
Brighton	A-217	750	94/7/5	11:00	95/5/1	16:00

图 19-6　时态关系 *account*

19.5.1　SQL-92 中的时间定义

SQL-92 标准定义了 date、time 和 timestamp 类型。date 类型包含表示年的四位（1～1999），表示月的两位（1～12）和表示日的两位（1～31）。time 类型包含表示小时的两位、表示分钟的两位、表示秒的两位，以及可选的表示秒后小数的一些数位。秒字段可以超过 60 以允许闰

秒的情况——由于地球旋转速度存在微小变化，有些年里需要增加秒数来进行校准，这些增加的秒数就是闰秒。timestamp 类型包含 date 字段和 time 字段，其中秒字段上有六位小数。SQL-92 的时间是在通用协调时间（UTC）中申明的时间，UTC 是申明时间的标准。（标准缩写是 UTC 而不是 UCT，因为它是“Universal Coordinated Time”的法文 universel temps coordonné 的缩写。）SQL-92 还支持 time with time zone 和 timestamp with time zone 两种类型，它们用本地时间和本地时间对 UTC 的偏移来申明时间。例如，可以用美国东部标准时间和偏移 -6:00 来表示时间，因为美国东部标准时间比 UTC 晚六个小时。

SQL-92 支持称作 interval 的一种类型（或者更应该称作 span），这一类型使我们可以引用一段时间如“一天”或“两天零五小时”，而不必指明这一段时间具体从什么时候开始。这一概念和前面使用的时间段概念不同，前面使用的时间段是有具体起点和终点的一段时间。

19.5.2 时态查询语言

正如我们在这本书中曾定义的那样，数据库中的关系有时被称为快照关系，因为它反映的是现实世界一个快照中的状态。因此，时态关系在时间点 t 的快照是关系中在 t 时为真的元组去除时间段属性后构成的集合。时态关系上的快照操作产生关系在指定时刻（或如果不指定就指当前时刻）的快照。

时态选择是涉及时间属性的选择，时态投影是一个投影，它的元组从原始关系中的元组那里继承时间。时态连接结果中元组的时间是产生该元组的两个元组的时间交。如果时间不相交，则该元组从结果中去除。

谓词 precedes，overlaps 和 contains 可以用在时间段上，它们的含义很明显。intersect 操作可以用在两个时间段上，用以得到一个（可能为空的）时间段。然而，两个时间段的并可能是一个时间段，也可能不是。

时态关系中使用函数依赖时必须小心。尽管任一给定时刻帐户号可能用函数确定余额，但显然余额会随时间发生变化。如果对 R 的所有合法实例 r 而言，r 的所有快照都满足函数依赖 $X \rightarrow Y$，则时态函数依赖 $X \xrightarrow{\tau} Y$ 在关系模式 R 上成立。

关于扩展查询语言（如 QUEL 和 SQL）以增加它们支持时态数据的能力的一些建议已经提出。目前能提供最广泛支持的建议是 TSQL2，它是 SQL-92 的扩展。文献注解提供了讲述 TSQL2 的参考文献。

19.6 用户界面

早期数据库唯一的界面是通过过程调用，从那时开始，数据库用户界面的发展走过了一段漫长的历程。今天，数据库系统提供多种工具，可以使一些用于专门目的的用户界面（如定单输入和报表生成这样的任务）快速建造起来。

数据库的用户界面可以分为以下几大类：

- 面向行的界面。
- 表格界面。
- 报表界面。
- 图形用户界面。

不幸的是，关于用户界面没有标准，每个数据库系统通常都有自己的用户界面。本节描述基本概念，而不详细讨论任何具体的用户界面。

面向行的界面是最基本的用户界面。例如，交互的 SQL 系统允许用户键入查询，并以表格形式显示查询结果。尽管面向行的界面对即席查询很有效，但这样的界面不利于重复性工作，如银行出纳员和售货员进行的工作。

表格界面广泛用于将数据加入数据库。联机表格看起来很像书面表格。使用表格的一个常见例子是定单的输入，图 19-7 给出了这样一种表格。从数据库中检索某种类型的信息时表格也很有用。例如，定单输入表格也可用于在已知定单号或客户姓名时检索旧定单。

Customer Name: _____ Date: _____

Customer Number: _____ Order No.: _____

Address: _____

Phone Number: _____

Part Number	Part Description	Unit Price	Quantity	Subtotal
			Total	

图 19-7 定单输入表格

用编程语言如 Pascal 或 C 建造表格界面是一件相当乏味的工作。大多数数据库系统提供表格软件包，允许用户使用表格编辑程序，以一种简单的声明性方式建造表格。使用表格编辑器，用户可以定义表格中字段的类型、大小和格式。系统行为可以和用户行为（如按下键盘上的功能键，或者在某个字段上键入回车）联系起来。例如，登记定单这一事务的执行可以和在标有 Submit 的字段上键入回车联系起来。我们看另一个例子，同客户号字段上输入一个值相联系的动作可能是从数据库中查询客户信息，并且自动填充其他字段如客户姓名和地址的值。

通过在表格的字段上定义约束，我们可以进行简单的错误检查。例如，货物号字段上的约束可以检查用户键入的货物号是否对应于数据库中实际存在的货物。尽管事务执行时会对这样的约束做检查，但是尽早检测错误能使表格系统指明表格中的哪个字段发生了错误。表格软件包还提供诸如菜单这样的特性，菜单指出某一字段上有哪些合法的值。同直接使用编程语言建造表格相比，以声明性方式控制这些特性的能力可以大大减少建造表格的劳动量。

报表书写器是许多数据库系统所支持的另一种用户界面。人们常常是为了发现趋势而查询数据库，例如查出上月销售总额，然后与上年对应月份的销售总额作比较。这种查询的结果需要制成格式恰当的报表，然后才能交给经理。此外，这样的报表需要定期产生，例如在每月月末产生。报表书写器软件包使我们可以很容易地将创建报表所需的正文格式化和产生报表中数据所需的数据库查询结合在一起。

我们用报表书写器提供的正文格式化工具来指明报表的格式。可以用变量来存储像年、月这样的参数，而报表中的字段可以用这些变量来定义。表、图、直方图以及其他图形都可以通

过数据库查询来定义。查询定义可以利用存储在变量中的参数值。一旦用报表书写器工具定义了报表结构，我们就可以把它存储起来，任何时候都可以运行它来产生报表。报表书写器系统提供了多种工具来构建表输出，如定义表首和列首、给出表中各分组的总计、自动将长表分页，以及给出每页总计。

图 19-8 是格式化报表的一个例子。报表中数据可以通过对定单信息做聚集产生。

Period: Jan. 1 to March 31, 1996

Region	Category	Sales	Subtotal
North	Computer Hardware	1,000,000	
	Computer Software	500,000	
	All categories		1,500,000
South	Computer Hardware	200,000	
	Computer Software	400,000	
	All categories		600,000
		Total Sales	2,100,000

图 19-8　格式化报表

数据库系统所提供的应用开发工具如表格软件包和报表书写器的集合常常称为第四代语言(4GL)。此称谓强调这些工具提供的编程范型不同于第三代编程语言如 Pascal 和 C 提供的命令式的编程范型。

近年来，支持图形用户界面的个人计算机在许多数据录入应用中代替了终端的位置。图形用户界面支持基于点击范型的交互，使用了诸如菜单和图标这样的特性。表格系统可以认为是图形用户界面的原始形式。通过使用直方图、饼图、图形以及其他类似这样的手段，图形用户界面提供了很好的数据输出途径。已经有大量用以简化数据库图形用户界面创建的工具被开发出来。Power Builder、Gupta SQL 和 Microsoft Access 数据库系统的用户界面都是这种工具的例子。

数据库图形用户界面家族的最新成员是 World Wide Web，通常简称 Web。（关于 World Wide Web 的更多信息见第 21 章。）在 Web 标准制定后的很短时间内，很多厂商就已宣布他们的数据库中包含了 Web 界面。数据库的 Web 界面主要基于超文本置标语言（HTML）提供的图形-表格工具。Web 是一个飞速发展的领域，在 Web 上很可能再出现新的图形用户界面工具。

19.7　主动数据库

早期数据库系统被动地存储数据，并且只执行用户事务显式要求的动作。与此相反，主动数据库系统不仅存储数据，还对事件如数据修改作出反应，执行一些动作。主动规则定义何时执行动作以及执行什么动作。

主动规则的事件-条件-动作模型使用广泛。这种模型中规则通常具有如下形式：

on 事件
if 条件
then 动作

用事件来模拟数据库的改变（如元组的插入、删除和更新）。在面向对象的数据库中，事件可以是一个动作如创建或删除对象，也可以是对象上方法的执行。事件还包括表示时间的事

件，例如“在 1999 年 12 月 31 日晚上 11:59”或“在每周日上午 12:00”。

当事件发生时，可能会触发一个或多个规则，这一事件称为这些规则的触发事件。一旦规则被触发，规则的条件就被检查。条件的例子如存货量是否低于某个最小值，或者更新是否使数据库处于不一致状态。

如果条件满足，规则的动作将被执行。动作的例子如在存货量低于最小值时订购新的储备，或者回滚事务（当更新使数据库不一致时）。

主动规则可以用于各种目的，如报警，即规则对数据库进行监控并在非常事件发生时告知管理员或用户。主动规则还可以用于检查一致性约束。另外，主动规则可用于维护派生数据，如索引和实体化视图。如果一个视图是实体化的（即被计算并存储），那么当定义它的数据库关系发生变化时，它需要做相应的更新。使视图保持最新的动作可用主动规则编码实现。

假设我们需要体现这样的约束：当员工记录被插入关系 *employee* 时，员工的工资必须小于该员工经理的工资。下面的规则可以体现这一约束：

```
define trigger employee-sal
on insert employee
if employee . salary >
    (select E . salary from employee as E
      where E . name = employee . manager)
then abort
```

define trigger 子句用来赋给触发器一个名字。规则的设计人员可以不将事务中止，而是定义其他某个动作，例如将该员工的工资减少到和他的经理一样。

主动规则的语法尚未标准化，在各个系统中可能有所不同。第 6 章中所讨论的触发器工具是主动规则工具的一个例子。

规则通常像常规数据一样存储在数据库中，因此规则是持久的，并且可以被所有数据库操作所访问。一旦规则进入数据库，数据库系统就必须保证无论何时只要指定事件发生并且对应条件满足，系统就会执行动作。

如果数据库系统不支持触发器，某些应用则可以用轮询来管理。也就是说，一个进程周期性地查询（轮询）数据库，看是否有关心的事件发生，然后执行必要的条件检查和动作。但是，如果数据库不提供适当的支持，检测事件（如插入或删除）是否发生的开销就会很大，因为每一次检测都可能需要对一个很大的关系做扫描。此外，由于执行更新的事务可能在轮询发生以前提交，因些像中止事务这样的动作就不能实现。因此，对很多应用来说轮询是不可行的。使用主动规则可以避免轮询带来的问题。

设计一组主动规则时，必须注意以下几点：

- 终止。规则中动作的执行可能导致新的事件发生，这些事件又可能触发其他规则，而这些规则接下来又可能触发其他规则，由此使得触发动作形成一条链。如果在一条链中某条规则发生两次，这条规则的执行就可能（间接地）触发它自己。这种情况下，触发动作就可能形成一条没有尽头的链，而规则的触发就永远也不会终止。注意，一条链中某条规则发生两次并不总是意味着会发生无限循环，因为规则的触发也可能由不同元组上的更新引起。规则系统的设计者必须保证在数据库一致的状态下任何更新都不会导致规则触发形成无限的链。
- 优先级。如果几条规则被同一事件触发，那么这些规则必须按某种顺序执行。规则系统通常提供一种定义每条规则优先级的方法。如果多条规则被触发，就按照它们的优先级规定的顺序执行。一些系统允许多条规则具有同一优先级。如果多条同一优先级的规则被触发，系

统对这些规则的执行顺序是不确定的（即任意的）。

• 事件－执行绑定。事件－执行绑定定义什么时候执行规则。例如，如果绑定被定义为立即的，那么一旦触发事件（如元组的插入或删除）发生，规则立即执行。规则的执行被看作是执行更新的事务的一部分。如果绑定被定义为推迟的，那么规则在执行更新的事务的末尾执行，仍作为该事务的一部分。如果绑定被定义为分立的，那么规则在执行更新的事务提交后在另一个单独的事务中执行。分立的绑定要求恢复系统提供相应支持，保证动作最终能被执行，即使在触发事务提交后发生了系统故障也应如此。有的规则系统对定义规则何时执行提供更强的支持，例如，允许规则在触发事件前或触发事件后执行。

• 错误处理。如果规则的执行导致运行时的错误，错误恢复就必须被启动。只要事件-执行绑定是立即的或推迟的，规则的执行就是执行更新的事务的一部分。在设计良好的规则系统中，这种错误反映的可能是事务中的错误，而不是规则自身的错误，这时错误恢复可以仅仅是将事务回滚。

如果事件－执行绑定是分立的，错误恢复就会困难得多。这种情况下，当错误被检测到时，想要回滚事务已经太晚了，不执行规则可能导致数据库的某种不一致。因此，到底应该怎样处理分立规则执行中的错误一般说来还并不清楚。为了处理这种错误，可能需要人工干涉。因此，规则系统的设计应保证分立规则执行中不会发生运行时的错误，这一点是至关重要的。

19.8 总结

数据库中存储的数据需要防止未授权访问、恶意破坏或修改、以及意外而引入的不一致性。防止数据一致性的意外破坏比防止对数据库的恶意访问要容易一些。完全杜绝对数据库的恶意滥用是不可能的，但我们可以通过使那些企图在没有适当授权情况下访问数据库的作恶者付出足够高的代价，来阻止绝大多数（如果不是全部）这样的访问企图。

在数据库的不同部分，用户可能具有不同形式的授权。授权是数据库系统用来防止未授权访问和恶意访问的一种手段。获得了某种形式授权的用户可能被允许将此授权传递给其他用户。但是，对于授权可能怎样在用户间传递我们必须很小心，以保证这样的授权在未来的某个时候可以被收回。

对高度敏感的数据而言，数据库系统的各种授权规则或许不能提供充分的保护。在这种情况下，数据可以被加密。只有知道如何对加密数据进行解码（解密）的人才能读出这些数据。

当允许数据用于统计目的时保护个人隐私比较困难。要处理潜在的安全性破坏，对系统来说一种简单方法是拒绝涉及的个体数目少于某个预定数字的所有查询。另一种解决安全问题的方法是数据污染，即对用于回答查询的数据进行随机的歪曲。与此类似的一种技术是对查询本身进行随机的修改。这两种技术的目标都牵涉到准确性和安全性之间的权衡。不管对统计数据采用什么安全手段，恶意用户总有可能确定个体的数据。但是，好的技术可以使金钱和时间上的耗费足够高而成为制止恶意用户的因素。

由于数据库系统的复杂性和对互操作的需要，标准对数据库系统来说非常重要。SQL有其正式标准。事实标准（如ODBC）和被行业组织所采纳的标准（如CORBA），在客户-服务器数据库系统的发展中发挥了重要作用。行业组织正在制定面向对象数据库的标准（如ODMG）。

性能基准程序在对数据库系统进行比较方面扮演了重要角色，尤其在数据库系统变得越来越遵循标准时。TPC基准程序集使用广泛，不同的TPC基准程序可以用于不同工作负载下的数据库系统的性能比较。调整数据库系统参数和高层数据库设计（如模式、索引和事务）对于实现高性能至关重要。调整的最好办法是确定瓶颈所在，然后消除瓶颈。

时间在数据库系统中占有重要位置。数据库是现实世界的模型。大多数数据库对现实世界在某一时刻（当前时刻）的状态建模，而时态数据库对随时间变化的现实世界建模。时态数据库中的事实同指示该事实何时有效的时间联系，这样的时间可以用若干时间段的并来表示。为了简化对时间的建模以及与时间相关的查询，时态查询语言已经被提出。

数据库用户界面有多种类型，从命令行界面到图形用户界面。表格界面广泛用于数据录入和一些打包的查询。表格软件包可以帮助程序员以一种简单的声明性方式建造表格。报表书写器用于产生基于数据库中数据的报表。它们将正文格式化和数据库的数据查询结合在一起。定义表格的语言和报表书写器语言构成人们所说的第四代语言（4GL）。

主动数据库对在数据库中定义和执行规则提供支持。在事件－条件－动作模型中，规则被事件触发，数据库系统检查被触发规则的条件，如果条件满足，数据库系统执行规则定义的动作。规则可用于多种目的，如在非常行为发生时对用户报警、再订购货物或体现完整性约束。如果一条规则的动作所触发的事件会（直接地或间接地）触发同一规则，我们就必须小心地保证规则集合能够终止。

习题

19.1　列出对银行来说需要考虑的安全性问题。对于你列出的每一项，说明它同哪一些安全性考虑有关，即同物理安全性、人际安全性、操作系统安全性或数据库安全性中的哪一些有关。

19.2　使用我们作为示例的银行数据库中的关系，分别写出定义如下视图的 SQL 表达式：

(a) 包含 Deer Park 分支机构所有帐户的帐户号以及客户姓名（但不包含余额）的视图。

(b) 包含在银行有帐户但无贷款的所有客户的姓名和地址的视图。

(c) 包含 Rock Ridge 分支机构每位客户的姓名及其平均余额的视图。

19.3　对于你在习题 19.2 中定义的每个视图，解释更新将如何进行（如果允许更新的话）。

(提示：见第 3 章对视图的讨论。)

19.4　在第 3 章中我们讲到，那些只需要访问数据库的一个部分的用户可以用视图来简化对数据库的访问。本章描述了视图作为安全性机制的使用。视图的这两个目的有冲突吗？解释你的答案。

19.5　索引的授权和资源的授权要用各自的日录，这样做的目的是什么？

19.6　以各自的操作系统文件存储各个关系的数据库系统可以利用操作系统的安全性和授权模式，而不是定义自己特有的模式。讨论这种方式的利弊。

19.7　对存储在数据库中的数据加密有哪两个好处？

19.8　数据加密会给第 11 章中所述的索引模式带来怎样的影响？特别是，它会给试图以排序方式存储数据的模式带来怎样的影响？

19.9　假设作为例子的银行维护了一个包含所有客户平均余额的统计数据库。这一关系的模式为（*customer-name*，*customer-city*，*avg-balance*）。假设出于安全性考虑，对这些数据的查询有如下限制：

- 每个查询至少涉及 10 个客户。
- 任意一对查询的交（即在一对的两个查询中都出现的客户数）至多为 5。

构造一系列查询来找出一个客户的平均余额。

(提示：这可以在七个查询以内做到。)

19.10　任何数据库系统中，最重要的数据项大概都是用来控制数据库访问的口令。为口令的

安全存储设计一个模式，确保你的模式允许系统去检测试图注册到系统中的用户所提供的口令。

19.11　列出预见性标准相对被动标准而言所具有的优点和缺点。

19.12　假设系统上运行三类事务。*A* 类事务的速率是每秒 50 个事务，*B* 类事务的速率是每秒 100 个事务，而 *C* 类事务的速率是每秒 200 个事务。假设事务混合的比例是 *A* 类事务 25%，*B* 类事务 25%，*C* 类事务 50%。

(a) 假设事务间互不干扰，此系统的平均事务吞吐量是多少?

(b) 什么因素会导致不同类型事务间互相干扰，从而使计算出的吞吐量不正确?

19.13　列出使 TPC 基准程序成为现实的、可靠的测量标准的一些特性。

19.14　(a) 数据库系统可以在哪三个大的层次上进行调整，以提高性能?

(b) 对每一层，试举两例说明怎样进行调整。

19.15　将长事务划分为一系列小事务的动机是什么? 这会带来哪些问题? 这些问题又该怎样避免?

19.16　时间有哪两种类型? 它们怎样区别? 为什么将一个元组同时与这两类时间联系起来是有意义的?

19.17　当通过增加一个时间属性来将关系转换为时态关系时，函数依赖能保持吗? 这一问题在时态数据库中是怎样解决的?

19.18　考虑视图 *branch-cust*，其定义如下：

```
create view branch-cust as
    select branch-name, customer-name
    from depositor, account
    where depositor.account-number = account.account-number
```

假设这一视图被实体化，也就是说，这一视图被计算并存储。书写主动规则来维护这一视图，即在 *depositor* 或 *account* 上进行插入和删除时要保持视图最新。不必考虑更新的情况。

文献注解

计算机系统安全性方面的内容在 Bell 与 LaPadula [1976] 和美国国防部 [1985] 中进行了全面的讨论。SQL 安全性方面的内容在 ANSI [1992] 中进行了详细讨论，Date 与 Darwen [1993] 和 Melton 与 Simon [1993] 对此进行了概括。Stonebraker 与 Wong [1974] 讨论了 Ingres 的安全性方案，其中讲到对用户查询做修改以保证用户不能访问未授权数据。Denning 与 Denning [1979] 是数据库安全性的一个概览。出于对安全性的维护、在必要时能产生错误答案的数据库系统在 Winslett 等 [1994] 和 Tendick 与 Matloff [1994] 中进行了讨论。

数据库安全性著作包括 Lunt 等 [1990]、Stachour 与 Thuraisingham [1990]、Jajodia 与 Sandhu [1990]、Kogan 与 Jajodia [1990]，以及 Qian 与 Lunt [1996]。Rabitti 等 [1988, 1991] 和 Thomas 与 Sandhu [1996] 讨论了面向对象数据库的安全性。Lunt [1995] 给出了有关授权问题的综述。

操作系统安全性问题在多数操作系统书本中进行了讨论，其中包括 Silberschatz 与 Galvin [1994]。

Chin 与 Ozsoyoglu [1981] 讨论统计数据库的设计。一些论文提供了在各种假设下数据库

中获取私人信息所需查询数的数学分析，其中包括 Kam 与 Ullman [1977]、Chin [1978]、DeMillo 等 [1978]、Dobkin 等 [1979]、Yao [1979a]、Denning [1980]，以及 Leiss [1982b]。

另一个从数学角度来研究的安全性方面的内容是数据加密。数据加密标准由美国贸易部 [1977] 给出。公钥加密在 Rivest 等 [1978] 中讨论。其他对密码的讨论包括 Diffie 与 Hellman [1979]、Lempel [1979]、Simmons [1979]、Davies [1980]、Fernandez 等 [1981]，Denning [1982]、Leiss [1982a]，以及 Akl [1983]。

美国国家标准 SQL-86 在 ANSI [1986] 中描述。IBM 系统应用体系结构 SQL 定义由 IBM [1987] 定义。SQL-89 和 SQL-92 标准可分别由 ANSI [1989] 和 ANSI [1992] 获得。SQL-3 标准的制定工作目前正在进行之中，本书的主页（见前言）中可以找到最新进展。X/Open SQL 调用层接口在 X/Open [1993] 中定义，ODBC API 在 Microsoft [1992] 中描述。ODMG-93 标准在 Catell [1994] 中定义。

数据库系统基准程序的一个早期建议（威斯康星 基准程序）由 Bitton 等 [1983] 提出。TPC A，B，和 C 基准程序在 Gray [1993] 中描述。在万维网上，通过 URL http://www.tpc.org 可以获得联机版本的所有 TPC 基准程序描述以及基准程序结果，这一站点还包含了关于最新基准程序如 TPC-E 的信息，TPC-E 是对 TPC-C 的一种建议性扩展。用于 OODB 的 OO1 基准程序在 Catell 与 Skeen [1992] 中描述，OO7 基准程序在 Carey 等 [1993] 中描述。与 TPC 基准程序不同，OODB 基准程序由个人设计，而不是由委员会设计。

Kleinrock 的两卷书 [1975，1976] 是很受欢迎的排队论教科书。Shasha [1992] 在数据库调整方面提供了一个很好的综述。O'Neil 的教科书 [1994] 描述了性能的度量和调整。Brown 等 [1994] 描述了一种自动的调整方案。

讨论将时间引入关系数据模型这一问题的有 Snodgrass 与 Ahn [1985]、Clifford 与 Tansel [1985]、Gadia [1986，1988]、Snodgrass [1987]、Tnasel 等 [1993]、Snodgrass 等 [1994]，以及 Tuzhilin 与 Clifford [1990]。Stam 与 Snodgrass [1988] 和 Soo [1991] 提供了关于时态数据管理的概览。Jensen 等 [1994] 给出了时态数据库概念的一个词汇表，旨在统一术语。Snodgrass 等 [1994] 提出了 TSQL2 建议。Tansel 等 [1993] 是时态数据库不同方面文章的一个汇集。Chomicki [1995] 给出了管理时态数据一致性的技术。Clifford 等 [1994] 给出了关于时态查询语言完备性的概念，类似于关系语言完备性（等价于关系代数）。

商业数据库系统的用户界面（特别是表格和报表书写器界面）在各自的用户手册中进行了描述。Nguyen 与 Srinivasan [1996] 描述了 IBM DB2 数据库系统的 World Wide Web 界面。

系统 R 的触发器子系统由 Eswaran 与 Chamberlin [1975] 描述。McCarthy 与 Dayal [1989] 讨论了基于事件-条件-动作形式的主动数据库的体系结构。Widom 与 Finkelstein [1990] 讨论了基于面向集合规则的规则系统的体系结构。基于 Starburst 可扩展数据库系统的规则系统的实现在 Widom 等 [1991] 中给出。考虑一种允许非确定性选择的执行机制。如果不管规则如何选择最终状态总是一样的，一个规则系统则被认为是收敛的，关于终止、非确定性以及收敛规则系统等的问题在 Aiken 等 [1995] 中讨论。模块化是管理大型规则库的一种手段，在 Baralis 等 [1996] 中进行了讨论。

第 20 章　高级事务处理

在第 13、14、15 章中，我们介绍了事务的概念，事务是程序的一个单位，它对各种数据项进行访问，也可能进行修改，它的执行确保具有 ACID 特性。在那 3 章中，我们讨论了在可能发生故障，也可能多个事务并发运行的环境中，为保证 ACID 特性所采用的各种模式。本章我们比以前讨论过的基本模式更进一步，介绍一些更高级的事务处理概念，包括远程备份系统、事务处理监控器、高性能事务系统、长事务、嵌套事务、实时事务系统、以及一致性级别比可串行性弱的事务系统。

20.1　远程备份系统

传统的事务处理系统是集中式系统或客户－服务器系统。这样的系统不具备抵御火灾、水灾、地震等环境灾害的能力。人们逐渐提出了对具有高可用性，能够在环境灾害的情况下继续工作的事务处理系统的需求。

通过在分布式数据库系统中进行数据复制，对数据的所有副本进行更新，事务可以达到高可用性。可以采用两阶段提交来在节点间进行同步。如果一个节点发生故障，事务处理可使用其他节点继续进行。然而，像并发控制这样原先在单个节点上实施的动作，现在需要在多个节点间进行同步，从而会造成延迟。而且，两阶段提交本身代价很高。因此，采用这样的模式使事务吞吐量受到很大影响。另一种可采用的方法是在一个称作主节点的节点上进行事务处理，而另有一个远程备份节点，主节点的所有数据都复制到远程备份节点上。远程备份节点有时也称作辅助节点。当在主节点上进行更新时，远程节点必须保持与主节点同步。我们通过将所有日志记录从主节点传送到远程备份节点来达到同步。远程备份节点必须是物理地与主节点分离的，例如，可以将它设置在另一个州，这样发生在主节点的灾害就不会毁坏远程备份节点。图 20-1 显示了远程备份系统的体系结构。

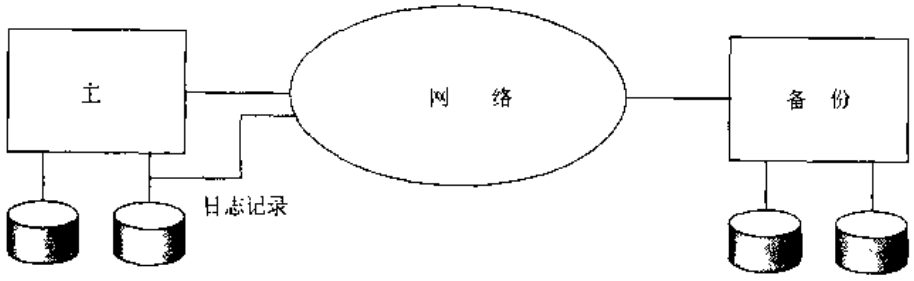


图 20-1　远程备份系统的体系结构

当主节点发生故障时，由远程备份节点取而代之，进行处理。然而，远程备份节点首先使用它自己的（也许是过时的）来自主节点的数据副本，以及它从主节点接受到的日志记录进行恢复。事实上，远程备份节点执行的恢复动作就是主节点恢复时需执行的动作。对标准的恢复算法稍加修改，就可用于远程备份节点的恢复。一旦完成了恢复过程，远程备份节点就可以开始处理事务。

与单节点系统相比，远程备份系统的可用性大大增强了，因为即使主节点上的所有数据都丢

失了，系统仍然可以恢复。远程备份系统的性能比采用两阶段提交的分布式系统的性能要好。

下面是当你设计一个远程备份系统时应注意的几个问题：

• 故障检测。对于远程备份系统来说，重要的一件事是检测主节点何时发生故障，这一点与分布式系统中的故障处理协议一样。通信线路故障会造成假象，使远程备份节点认为主节点发生了故障。为避免这一问题，可以在主节点和远程节点之间维持几条通信链路，它们各有自己的故障方式。例如，除了网络连接外，还可有一个通过电话线路的调制解调器连接，电话线路服务由另外的通信公司提供。这些连接可以通过接线员的人工干预做后盾，接线员能通过电话系统进行通信。

• 控制的转移。当主节点发生故障时，备份节点接替它进行处理，成为新的主节点。当原来的主节点恢复正常时，它可以充当远程备份的角色，也可以重新担任主节点的角色。不管是哪种情况，原来的主节点都必须接收当它发生故障不能工作时由远程备份节点所做的所有更新。

转移控制最简单的方法是，原来的主节点从原来的备份节点接受 redo 日志，在本地执行 redo，以实现这些已做的更新。原来的主节点可以充当远程备份节点。如果必须将控制转移回来，可以假装原来的备份节点发生故障，于是原来的主节点就取而代之。

• 恢复的时间。如果远程备份节点的日志增大到很大，恢复就要花很长时间。远程备份节点可以定期地处理它所收到的 redo 日志，还可以执行检查点，于是日志中较早的部分就可以删除掉，从而大大缩短远程备份节点接替主节点工作之前的这段延迟。

采用热备份配置可以使远程备份节点接替主节点的动作几乎是即时的。在这种配置中，远程备份节点在 redo 日志记录到达时不断进行处理，在本地进行更新。一旦检测到主节点发生故障，备份节点就将未完成的事务回滚，也即完成了恢复过程，然后它就可以进行新的事务了。

• 提交的时间。为了保证已提交事务的更新是永久的，一个事务在它的日志记录未到达备份节点之前不能认为是已提交的。这一延迟可能造成提交事务时较长的等待，因此，有些系统允许较低程度的持久性。持久性的程度可以分类如下：

  - 一方保险。一旦事务的提交日志记录被写到了主节点的稳定存储器中，该事务就算提交了。这种模式的问题在于，当备份节点接替主节点进行处理时，已提交事务的更新可能还没有进入到备份节点中。于是，更新看起来就像是丢失了。当主节点恢复工作时，丢失的更新不能直接归并进去，因为这些更新可能与后来在备份节点上进行的更新冲突。于是，需要人工干预来使数据库达到一致的状态。
  - 两方强保险。一旦事务的提交日志记录写入到了主节点的稳定存储器和备份节点的稳定存储器中，该事务就提交了。这种模式的问题在于，如果主节点或备份节点发生故障，事务处理就不能进行了。于是，该模式虽然丢失数据的可能性小多了，但可用性实际上却比单节点系统低。
  - 两方保险。如果主节点和备份节点都能正常工作，这种模式与两方强保险一样。如果只有主节点能工作，那么一旦事务的提交日志记录写入到主节点的稳定存储器中，就允许事务提交。这种模式比两方强保险的可用性高，同时也避免了一方保险模式面临的丢失事务的问题。虽然它的提交比一方保险模式慢，但总的来说好处高于代价。

市场上有一些共享磁盘系统提供了一个介于集中式系统和远程备份系统之间的容错层次。在这些系统中，CPU 故障不导致系统故障，其他的 CPU 可以接替该 CPU 的工作，并且由它们进行恢复。恢复工作包括运行在出故障的 CPU 上的事务的回滚，以及这些事务所持有的锁的

恢复。由于数据是在共享磁盘上的，因此不需要日志记录转移。然而我们应该采取一些措施，例如 RAID 磁盘结构，来保护数据，以免磁盘故障的影响。

20.2 事务处理监控器

传统上将数据库系统看作一个整体式系统。然而，在现实世界中，数据常常分割在多个数据库系统、文件系统和应用系统中，它们可以运行在不同的机器上。作为单个事务的组成部分，用户可能需要在几个这样的系统中进行更新。因此，事务的一致性特性和故障时的恢复是至关重要的。两阶段提交协议是重要成分以支持跨多个数据库的事务。然而，各个层次的软件以及对应的软件体系结构需要来协调这种任务的执行。事务处理（TP）监控器正是提供这种支持的系统。

70 年代和 80 年代开始开发 TP 监控器，那时是针对一台计算机支持大量的远程终端（例如机票预定终端）的需求。术语 TP 监控器原是代表远程处理监控器。这些系统后来演化成提供对分布式事务处理的核心支持，于是术语 TP 监控器具有了它现在的含义。市场上的 TP 监控器（大致按推出的时间顺序）有 IBM 公司的 IMS 系统及 IBM CICS（70 年代推出）、Tandem 公司的 Pathway（1979 年）、Digital Equipment 的 ACMS 和 RTR（1981 年和 1987 年）、NCR 的 Tuxedo 和 Top End（1985 年和 1991 年）以及 Transarc 的 Encina（1993 年）。

20.2.1 TP 监控器体系结构

在当今典型的操作系统中，用户注册进入系统并执行一个或多个进程。考虑一个飞机订票系统，它有数千个订票终端同时在活动，所有的终端都连接到一台中央计算机。对于每一个注册对话期，操作系统都有相当大的存储开销，如果一个终端的每个用户都要注册到计算机系统中，那么存储器开销就会更大。再者，出于安全性考虑，也不应该让太多远程用户通过注册对话而具有对操作系统的全面访问。

不让每一个终端都有一个注册对话期的一种方法是用一个服务器进程来与所有终端通信，并进行鉴别和执行终端所要求的动作。图 20-2a 描述了这种每个客户一个进程的模式。这种模式在存储器的利用和处理速度方面存在几个问题：

- 每个进程的内存需要很大。即使程序代码所用的内存是所有进程共享的，每个进程还要占用内存用于局部数据和打开文件描述，以及支持虚拟存储的页表等操作系统开销。
- 操作系统通过在进程之间切换来划分可利用的 CPU 时间，这种技术称作多任务调度。

一个进程和下一个进程之间的上下文切换需要相当大的 CPU 开销，即使在今天相当快的系统上，一次上下文切换也要花上数百微秒的时间。

在 70 年代和 80 年代有限的内存容量和处理器能力条件下，采用这种模式来同时为数千个用户服务是不可能的，这会用尽内存和 CPU 资源。

为避免这些问题，一些系统，例如 IBM CICS 推出了远程处理监控器的概念。这样的系统中有单一的一个服务进程，所有的远程终端都与它连接，这种模式称作单服务器模式，如图 20-2b 所示。远程终端将请求发送到服务器进程，然后由服务器进程执行这些请求。这种模式也可以用在客户 - 服务器环境中。客户将请求发送到单一的服务器进程，服务器进程处理如用户鉴别等任务，如果没有服务器进程，这些任务通常是由操作系统执行的。为了避免在处理一个客户的长时间请求时阻塞其他客户，服务器进程是多线索的：服务器进程对每个客户有一个线索，并且事实上是在实现它自己的低开销多任务调度。它为一个客户执行一段时间，然后保存其上下文，接着切换去为另一个客户执行代码。在线索之间切换的代价是很小的（通常仅有

几个微秒），比完全的多任务调度代价要小得多。

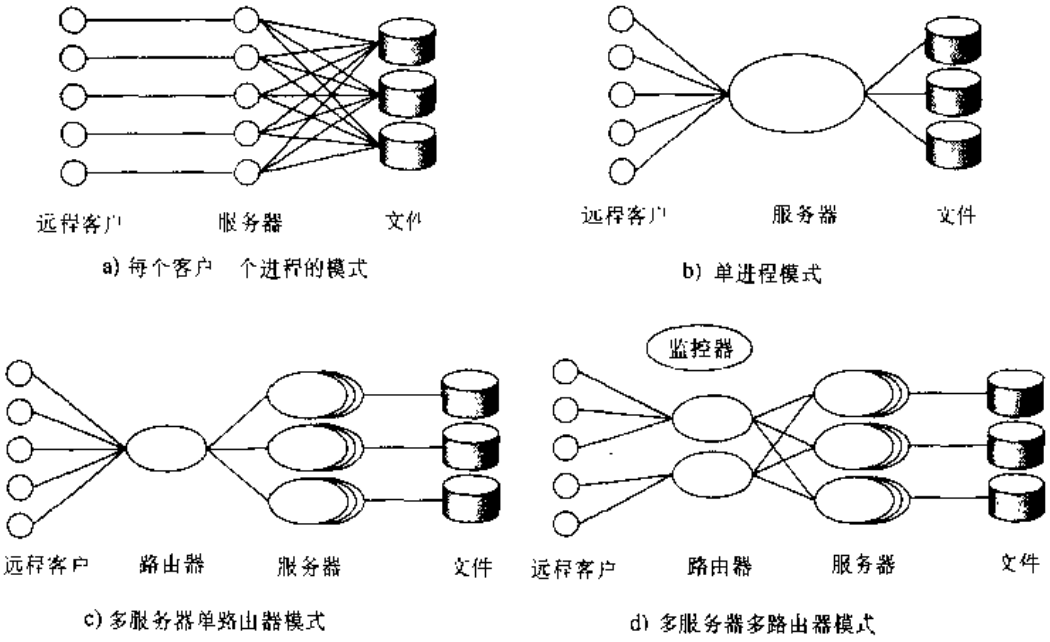


图 20 2　TP 监控器体系结构

基于单服务模式的系统，例如 IBM CICS TP 监控器的早期版本，以及像 Novel 的 Netware 这样的文件服务器，在资源有限的情况下都成功地提供了较高的事务率。然而，它们也存在一些问题，尤其当多个应用访问同一数据库时：

- 由于所有的应用作为单一的一个进程运行，因而它们之间没有保护。一个应用系统中的错误会影响到所有其他的应用系统。最好是每一个应用作为一个独立的进程执行。
- 这样的系统不适合于并行的或分布式的数据库，因为一个服务器进程不能同时在多台计算机上执行。（然而，在共享内存多处理器系统中能够支持一个进程中多个并发的线索。）在大型组织机构中，数据的分布正变得越来越普遍。

解决这些问题的一个办法是运行多个应用服务器进程，它们访问一个公共的数据库，并且让客户通过单一的一个通信进程去与多个应用服务器进程通信，这些通信进程管理请求的发送。这种模式称作多服务器单路由器模式，如图 20-2c 所示。这种模式支持用于多个应用的独立服务器进程，而且，每一个应用可以有一组服务器进程，这些服务器进程中的任何一个都可以处理客户对话。例如，请求可以安排发送到这一组服务器进程中负载最轻的那一个去。如前所述，每一个服务器进程自身可以是多线索的，因此它可以并发地处理多个客户。进一步发展，多个应用服务器可以在并行的或分布式数据库的多个不同节点上运行，通信进程可以在这些进程间进行协调。

一种更通用的体系结构是有多个进程，而不是一个进程，来与客户通信。客户通信进程与一个或多个路由器交互，路由器进程将请求发送给适当的服务器。因此，更新一代的 TP 监控器具有一种不同的体系结构，称作多服务器多路由器模式，如图 20-2d 所示。一个控制进程启动其他的进程，并监控它们的工作。Tandem Pathway 是采用这种体系结构的新一代 TP 监控器的实例。

TP 监控器的详细结构如图 20-3 所示。TP 监控器不是仅仅简单地将消息传送给应用服务器。当消息到达时，必须将它们放入到队列中，因此，需有一个队列管理器来管理到达的消息。此外消息还可以，存到稳定存储器中，这样，消息一旦收到了，就会最终被执行，哪怕系

统发生故障。TP 监控器的进一步功能是权限控制和应用服务器管理（例如服务器启动和消息分发到各服务器）。TP 监控器通常还提供日志、恢复和并发控制功能。这样，如果需要的话，应用服务器就能直接实现事务的 ACID 特性。

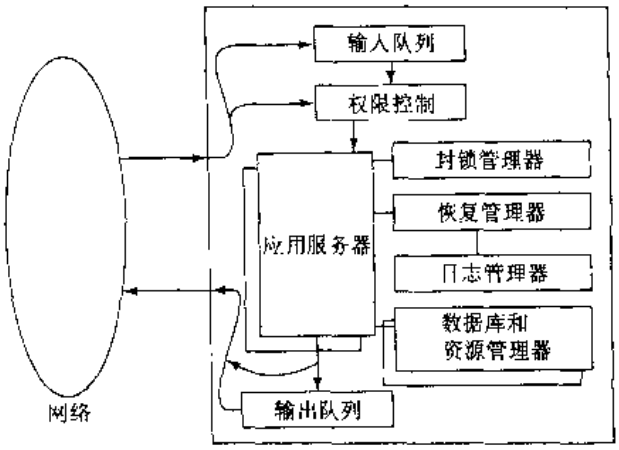


图 20-3　TP 监控器构成成分

TP 监控器还为输出消息提供永久性队列服务。因此，服务器可以把消息输出作为事务的一部分。TP 服务器保证，只要（而且仅当）事务提交了，消息就能发出去，这样 TP 服务器不仅对数据库内部的动作，而且对数据库之外的消息传送都提供了 ACID 特性。

除了以上设施之外，许多 TP 监控器为终端这样的哑客户提供了*表示功能*，以帮助建立菜单和表格界面。

20.2.2　使用 TP 监控器进行应用协调

当前的应用系统经常要与多个数据库打交道。它们还可能与遗产系统[⊖]，例如直接建立在文件系统上的数据存储系统，进行交互。此外，它们可能要与远程节点上的用户或其他应用系统通信，因此，它们还必须与通信子系统交互。对于跨越在这样系统上的事务，重要一点是能对数据访问进行协调以实现 ACID 特性。

先进的 TP 监控器提供支持，去构造和管理这种由多个子系统（例如数据库、遗产系统、通信系统）构成的大型应用系统。TP 监控器将每个子系统作为一个资源管理器，资源管理器提供对某些资源的事务性访问。TP 监控器与资源管理器之间的接口由一组事务原语（例如 *begin _transaction*、*commit _transaction*、*abort _transaction*，以及用于两阶段提交的 *prepare _to _commit _transaction*）来定义。当然，资源管理器还必须提供其他服务，例如为应用系统提供数据。资源管理器接口由 X/Open 分布式事务处理标准定义。许多数据库系统支持 X/Open 标准，可以用来充当资源管理器。TP 监控器（以及其他产品，例如支持 X/Open 的标准的 SQL 系统）可以与资源管理器连接。而且，TP 监控器还提供如持久队列管理和消息传送等服务，这些服务充当支持事务的资源管理器。TP 监控器可以对这些服务系统和数据库系统中的许多事务进行协调。例如，当队列中的一个更新事务被执行时，产生一条输出消息，然后一个请求事务从请求队列中删除。TP 监控器即使在系统故障情况下也能保证这些动作或者都发生，或者都不

⊖　遗产系统是指与当前的标准和系统不兼容的老系统。这样的系统中仍然可能包含有价值的数据，仍然支持关键的应用。把这些应用移植到更现代的环境中通常既费时间又费财力。因此，支持这些老系统或遗产系统，使它们能与较新的系统互操作是很重要的。

发生。

在客户－服务器系统中，客户通常通过远程过程调用（RPC）机制与服务器交互，由客户启动一个对已存在于服务器中的过程的调用，再将结果传回给客户。只要涉及 RPC 的客户代码被启动，它看起来就和本地过程调用一样。像 Encina 这样的 TP 监控器系统对它们的服务提供一个事务 RPC 接口。在这样的接口中，RPC 机制提供一些调用，这些调用可用来将一系列的 RPC 调用包含在一个事务中。于是，由一个 RPC 执行的所有更新都在该事务的范围内执行，如果发生了故障，事务就能够回滚。

我们还可以使用 TP 监控器来管理包含多个服务器和大量客户的复杂的客户－服务器系统。每一个服务器起一个资源管理器的作用，并注册到 TP 监控器中。TP 监控器对系统检查点和系统关闭等各种活动进行协调。它提供安全性和客户鉴别，管理服务器缓冲池，使系统不用中断就能增加和删掉服务器。此外，它还能控制故障范围。如果一个服务器发生了故障，TP 监控器能探测到这一故障，然后中止正在进行的事务，再重新启动事务。如果一个节点发生了故障，TP 监控器能够将事务移到其他节点的服务器上去，并且回滚未完成的事务。当故障节点重新启动时，TP 监控器能够控制节点上的资源管理器的恢复。

现代数据库系统支持数据库的复制，主节点上进行的更新也自动地在复制节点上执行。远程备份系统是这种复制的一个例子。如果主节点发生故障，辅助节点中的一个就接替它进行处理（参见第 20.1 节）。在这种复制系统中，可以用 TP 监控器来隐藏数据库故障，事务的请求发送到 TP 监控器，TP 监控器再将消息发送给某一个数据库副本。如果某一个节点发生故障，TP 监控器可以先给这个节点做上故障标记，然后不动声色地将消息发送到备份节点上去。

20.3　高性能事务系统

要达到高事务处理率（每秒种数百或数千个事务），我们必须使用高性能硬件，同时还必须充分利用并行性。然而，单靠某一种技术，还不足以达到高事务处理性能，原因如下：

- 硬盘 I/O 仍然是个瓶颈——每一次 I/O 大约需要 10 毫秒，而且随着处理器速度的增长，这一数字并不能以相对应的比率减小。磁盘 I/O 通常是读的瓶颈，也是事务提交的瓶颈。
- 并行运行的事务可能去读或写同一数据项，造成数据冲突，从而减少有效的并行。长时间运行的事务在这一问题上更加严重。

下两节讨论减少上述问题影响的技术。

20.3.1　主存数据库

通常，数据库系统的性能受数据从磁盘读出和写入磁盘的速度的限制。我们可以通过增大数据库缓冲区来减小数据库系统受磁盘限制的程度。主存储器技术的进步使得我们能够以相对较低的代价建造相当大的主存储器。现在，市场上有能够支持 14GB 的主存储器的 64 位系统。

长时间的磁盘等待（平均约 10 毫秒）不仅增加了存取一个数据项所需的时间，而且限制了每秒钟的存取数目。因此，对于某些应用，例如实时控制应用，必须将数据存储在主存储器中，以满足性能要求。虽然总是有几个应用系统需要将几个 GB 的数据保留在主存中，但大多数这类系统对于主存大小的要求并不是极端地大。随着主存大小的增长，会有更多的应用系统能够将数据保存到主存储器中。

大的主存储器能使数据驻留在主存中，从而使事务处理更快。然而，仍然存在与磁盘有关的限制：

- 在事务提交前必须将日志记录写到稳定存储器中。由于大的主存储器所带来的性能增

长可能导致日志处理成为瓶颈，我们可以通过使用电池支持的存储器（也称作永久性 RAM）在主存储器中建立稳定的日志缓冲区来缩减提交时间。记日志而带来的开销还可以通过下节中讨论的成组提交技术来减小。吞吐量（每秒钟的事务数目）仍然受日志磁盘的数据传输率的限制。

• 为了减小在恢复时必须重新执行的日志量，仍然有必要写出由已提交事务修改过的缓冲块。如果更新的比率非常高，那么磁盘数据传输率可能变成瓶颈。

• 如果系统崩溃了，整个主存储器的内容都会丢失。恢复后，系统的数据库缓冲还是空的，当要对数据项进行访问时必须重新从磁盘输入。因此，即使恢复已经完成了，仍然需要一些时间等待数据库完全装入到主存储器中，并恢复对于事务的高速处理。

另一方面，主存数据库为优化提供了机会：

• 由于主存储器比磁盘空间价格高，因此主存数据库的内部数据结构设计必须注意减小空间需求量。但是，主存数据结构中可以有跨不同页面的指针，这一点与磁盘数据库不同，磁盘数据库中跨越多个页面的 I/O 代价非常高。例如，主存数据库中的树结构与 B^+ 树不同，前者深度可以相对大一些，但要尽量少占空间。

• 没有必要在对数据进行存取之前在存储器中占住缓冲区页面，因为缓冲区页面不会被替换。

• 查询处理技术应该设计成尽量减小空间开销，从而在执行查询时不会超出主存储器的界限，一旦发生这种情况会导致分页至交换区中，从而减慢查询处理。

• 一旦消除了磁盘 I/O 瓶颈，封锁等操作可能会变成瓶颈。必须通过对这些操作实现的改进来消除瓶颈。

• 可以对恢复算法进行优化，因为很少需要将页面写出以给其他页面腾出空间。

文献注解中的参考文献给出了有关主存数据库的更多的信息。

20.3.2　成组提交

事务 T 的提交过程需要将以下内容写到稳定存储器中：

• 尚未输出到稳定存储器中的与事务 T 有关的所有日志记录。

• $<T$ Commit$>$ 日志记录。

这些输出操作经常需要将没有完全写满的块输出去。为了保证输出接近写满的块，我们采用成组提交技术。系统不是在 T 完成时就提交 T，而是等到有几个事务都完成了，然后将这一组事务一起提交。写到稳定存储器中的块可能包含几个事务的记录。通过仔细选择组的大小，系统可以保证当块写到稳定存储器时，块是满的。平均说来，这个技术减少了每个提交事务的输出操作。虽然成组提交减少了日志所引起的开销，但由于要等到足够大的一组事务完成了一起提交，因而它导致了事务提交的延迟。在高性能系统中这种延迟是可以接受的，因为等到一组事务都完成再提交花不了多少时间。

20.4　长事务

事务概念原本是在数据处理应用中提出的，在该应用中大多数事务不是交互式的，并且持续时间很短。虽然本章，以及前面的第 13、14、15 章中讨论的技术能够很好地支持那些应用，但当事务的概念应用到涉及人机交互的数据库系统时，就出现了严重的问题。这些事务具有如下重要特性：

• 持续时间长。一旦人介入到活动事务中，从计算机的观点看，事务的持续时间就会很

长了，因为人的响应时间比计算机的速度慢。而且，在设计类应用中，被模拟的人的活动可能会长达数小时、数天，甚至更长。因此，从人的观点和机器的观点看，事务的持续时间都很长。

- 未提交数据的曝光。长事务所产生并显示给用户的数据是未提交的数据，因为事务可能中止。因此，用户和由此而导致的其他事务，可能被迫去读未提交的数据。如果几个用户合作进行一项设计，那么用户可能需要在事务提交之前交换数据。
- 子任务。一个交互式的事务可能由用户启动的一组子任务构成。用户可能希望中止其中一个子任务，而不是中止整个事务。
- 可恢复性。由于系统崩溃而使一个交互式的长事务中止，这是不可接受的。活动事务必须被恢复到系统崩溃之前不久的某一个状态，这样丢失掉的人的工作较少。
- 性能。交互式事务系统中性能好定义为响应时间快。这一定义与非交互式系统中的定义不同。在非交互式的事务系统中，目标是高吞吐量（每秒钟的事务数目）。然而，在交互式事务的情况下，代价最高的资源是用户。如果要提高用户的工作效率和满意程度，响应时间就应该快（从人的观点看）。在一个任务持续很长时间的情况下，响应时间应该是可预知的（即响应时间中发生的变化应该不大），于是用户就可以很好地安排自己的时间。

从下面五小节会看到为什么这五个特性与前面所讨论的技术不相容，我们还将讨论如何对那些技术进行修改以适应交互式长事务。

20.4.1　不可串行化的执行

我们所讨论的这些特性使得前面章节中所要求的只允许可串行化调度变得不切实际。第 14 章中讨论的每一种并发控制协议对于长事务都有负面影响。

- 两阶段封锁。当一个事务请求封锁而得不到时，它必须被迫等待所要求的数据项上的锁释放。等待时间的长短与持有锁的事务的持续时间成比例。如果数据项被一个短事务封锁了，我们估计等待时间会很短（除非发生了死锁或系统负载极重）。然而，如果数据项被一个长事务封锁了，则会发生长时间的等待。长时间等待导致响应时间长，并且发生死锁的可能性增大。
- 基于图的协议。基于图的协议使得锁释放比两阶段封锁协议早，并且能避免死锁。然而，它对数据项强加了一个顺序，事务对于数据项的封锁必须与该顺序一致，其结果可能是事务封锁的数据比它实际需要的多。而且，事务确信某个锁不会再被使用之前，始终会保持该封锁，这样，很容易发生长时间的锁等待。
- 基于时间戳的协议。时间戳协议不需要事务等待。然而，在某些情况下，它需要事务中止。如果一个长事务被中止了，就会丢失大量的工作。对于非交互式事务来说，这种工作的丢失属于性能问题；对于交互式事务来说，这是个用户满意度的问题，用户很不希望看到他几个小时的工作都作废了。
- 有效性确认。与基于时间戳的协议一样，有效性确认协议通过中止事务来保证可串行性。综上所述，可串行化的强制要求看来会导致长时间的等待、长事务的中止、或者两者同时发生。对此结论有理论上的研究结果，参见文献注解。

当考虑恢复问题时，会发现强制要求可串行化所带来的进一步的困难。前面讨论过级联的回滚问题，即一个事务的中止可能导致其他事务的中止。这一现象并不是我们所希望的，尤其对长事务而言。如果采用封锁机制，又想避免级联回滚的发生，就必须保持排他锁直至事务结束。然而，这样保持排他锁就增加了事务等待时间的长度。

由此看来，强制要求事务的原子性或者导致长时间等待的可能性增大，或者导致发生级联回滚。

以上所讲述的为我们以下的讨论打下基础，下面我们对于并发执行和事务恢复的正确性概念重新加以考虑。

20.4.2　并发控制

数据库并发控制的基本目标是确保事务的并发执行不破坏数据库的一致性，可以利用可串行性概念来达到这一目标，因为所有可串行化调度都能保持数据库的一致性。然而，并非所有保持数据库一致性的调度都是可串行化的。为证明这一断言，让我们再来考虑有两个帐户 A 和 B 的银行数据库的一致性，其一致性要求是 A 与 B 之和保持不变。例如，尽管图 20-4 中的调度不是冲突可串行化的，但它却保持了 $A+B$ 的和不变。该例子还说明了有关非可串行化的正确性概念的两个重要观点。

- 正确性依赖于数据库的特定一致性约束。
- 正确性依赖于每个事务所执行操作的特性。

对事务所执行的低级操作以及这些操作对指定数据库一致性约束的可能影响进行自动分析，其计算代价实在是太高了。然而，另有一些比较简单、代价比较小的技术。其中之一是利用数据库一致性约束作为基础，对数据库进行分割，将数据库分成子数据库，每个子数据库中的并发可以分别管理。另一种技术是将 read 和 write 以及另外一些操作都看成是基本的低级操作，并且对并发控制进行扩展，将对它们的处理也包括在内。

文献注解中引进了不要求可串行性而保证一致性的另外一些技术。许多这些技术都利用了某种类型的多版本并发控制（参见第 15.6 节）。对于较早的只需要一个版本的数据处理应用来说，多版本协议导致了较高的空间开销，因为要存储额外的版本。由于许多新的数据库应用系统需要维护数据的多个版本，所以利用多版本的并发控制技术是符合实际的。

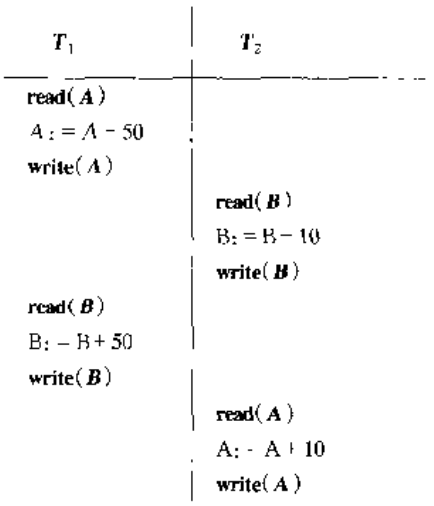


图 20-4　一个非冲突可串行化的调度

20.4.3　嵌套事务和多级事务

可以将一个长事务看成一组相关的子任务或子事务。通过将一个事务组织成一组子事务，可以提高并行度，因为有可能并行地运行其中的几个子事务。此外，还可以不用回滚整个长事务就能处理由于事务中止、系统崩溃等所造成的子事务失败。

一个嵌套的或多级的事务 T 包括一个子事务集合 $T=\{t_1, t_2, \cdots, t_n\}$ 和 T 上的一个偏序 P。T 中的一个子事务 t_i 可以中止，但不必强制地让 T 也中止。反过来，T 可以重新启动 t_i，也可以决定不运行 t_i。如果 t_i 提交了，这一动作并不使 t_i 成为永久的（这与第 15 章中所谈论的情况不同）。相反，如果 T 中止，则 t_i 提交给 T，同时 t_i 可能中止（或者会像下节中所讲，t_i 需要补偿）。T 的执行不能违反偏序 P。也就是说，如果在前驱图中出现边 $t_i \rightarrow t_j$，那么 $t_j \rightarrow t_i$ 一定不能在 P 的传递闭包中。

嵌套可能有好几层深，表示将事务划分为子任务，子任务又划分为子任务，等等。在嵌套的最底层，是我们前面用到过的标准数据库操作 read 和 write。

如果允许 T 的子任务在完成的时候释放锁，那么 T 就称作多级事务，当一个多级事务代表一个长时间活动时，有时将该事务称作 saga。而如果当 T 的子事务 t_i 完成时，它所持有的锁自动地赋给 T，那么 T 就称作嵌套事务。

虽然多级事务的主要实用价值在于复杂的长事务，但我们用图 20-4 的简单例子来说明嵌套如何能够建立较高级的操作，从而提高并发度。我们使用子事务 $T_{1,1}$和 $T_{1,2}$重写事务 T_1，这两个子事务执行增加或减少操作：

- T_1 包括
  - $T_{1,1}$，它从 A 中减掉 10。
  - $T_{1,2}$，它往 B 中加上 50。

同样，我们使用子事务 $T_{2,1}$和 $T_{2,2}$重写事务 T_2，这两个子事务执行增加或减少操作：

- T_2 包括
  - $T_{2,1}$，它从 B 中减掉 10。
  - $T_{2,2}$，它往 A 中加上 10。

对于 $T_{1,1}$，$T_{1,2}$，$T_{2,1}$，$T_{2,2}$没有说明顺序，这些子事务的任何顺序的执行都会产生正确的结果。图 20-4 中的调度对应于调度$< T_{1,1}, T_{2,1}, T_{1,2}, T_{2,2}>$。

20.4.4　补偿事务

为减少长时间等待的发生频率，我们安排将未提交的更新暴露给其他并发执行的事务。事实上，多级事务可能允许这种曝光。然而，将未提交数据暴露出去就会导致级联回滚的可能性。补偿事务的概念帮助我们处理这一问题。

令事务 T 分为子事务 $t_1, t_2, \cdots, t_n$，当一个子事务 t_i 提交之后，它释放了它的封锁。现在，如果中止外层事务 T，那么它的子事务所造成的影响必须清除。假设子事务 $t_1, \cdots, t_k$ 已经提交了，且做出中止决定时子事务 t_{k+1}正在执行。我们可以通过中止子事务 t_{k+1}来消除它的影响。然而，不可能中止子事务 $t_1, \cdots, t_k$，因为它们已经提交了。

我们通过执行一个新的、称作补偿事务的子事务 ct_i 去消除子事务 t_i 的影响，每一个子事务 t_i 都需要一个补偿事务 ct_i。补偿事务必须以相反的顺序 $ct_k, \cdots, ct_1$ 执行。下面我们给出几个补偿的例子：

- 考虑图 20-4 的调度，我们已经说明了它是正确的，尽管它不是冲突可串行化的。每个子事务在完成时就释放它的封锁。假设在 T_2 即将结束时且 $T_{2,2}$已经释放了它的封锁之后，T_2 失败了，那么我们就运行一个对 $t_{2,2}$的补偿事务，它从 A 中减掉 10，同时运行一个对 $T_{2,1}$的补偿事务，它往 B 中加上 10。

- 考虑事务 T_i 所做的一个数据库插入操作，它的副作用造成了 B^+ 树索引的更新。该插入操作可能修改了 B^+ 树索引的几个节点，其他事务在访问数据（不是 T_i 新插入的记录）时可能已经读过这些结点。正如第 15.9 节所讨论的，我们可以通过删除 T_i 插入的记录来消除这个插入的影响。其结果是一棵正确的、一致的 B^+ 树，但不一定与 T_i 开始之前的结构完全相同，因此删除是插入的补偿动作。

- 考虑一个长事务 T_i，它代表一次旅游订座。事务 T_i 有 3 个子事务：$T_{i,1}$预定飞机票；$T_{i,2}$预定出租汽车，$T_{i,3}$预定旅馆房间。假设旅馆取消了该订单，可不用消除整个 T_i 的影响，

我们对 $T_{1,3}$失败的补偿是删除原来的旅馆订单，并且建立一个新的订单。

如果系统在一次外层事务执行中崩溃了，那么当进行恢复时它的子事务必须回滚。第15.9节所描述的技术可以用于这一目的。

对于事务失败的补偿需要利用失败了的事务的语义。对于某些操作，例如增加，或往 B^+ 树中插入，很容易定义对应的补偿。对于更复杂的事务，可能需要应用程序在对事务进行编码时定义补偿的正确形式。对于复杂的交互式事务，可能需要系统去与用户进行交互，以确定补偿的正确形式。

20.4.5　实现问题

本节讨论的事务概念为实现带来了很大的困难。我们在这里列出了其中的几个，并且讨论我们可以如何对待这些问题。

在系统崩溃时长事务必须想办法解脱。我们能够确保这一点，办法是对已提交了事务执行一个 redo，对于系统崩溃时正在进行的短的子事务或者执行一个 undo，或者执行一个补偿。然而，这些动作仅解决了问题的一部分。在典型的数据库系统中，像封锁表、事务时间戳等内部系统数据是存放在易失性存储器中的。为使长事务在系统崩溃后能够恢复，这些数据必须是能够恢复的。因此，不仅必须对数据库的改变记日志，而且还要对与长事务有关的内部系统数据记日志。

由于存在于数据库中的数据项的类型繁多，这使得更新记日志变得更加复杂。数据项可能是一个 CAD 设计、文档的正文，或者组合设计的另一种形式。这些数据项物理上很大，因此我们不希望在日志记录中既存储数据项的旧值，又存储它的新值。

有两种方法可以减少保证大数据项可恢复性的开销：

- 记操作日志。只将在数据项上执行的操作和数据项的名字存放在日志中。操作日志也称作逻辑日志。我们用逆向操作执行 undo，用操作本身执行 redo。通过操作日志进行恢复是一件更加困难的事情，因为 redo 和 undo 不是幂等的。而且，对更新多个页面的操作使用逻辑日志是非常复杂的，因为更新过的一些页面，但不是所有页面可能已经被写到磁盘中去了，在恢复的时候很难在磁盘映象上进行该操作的 redo 和 undo。
- 记日志和影子页。日志用于对小数据项的更新，而大数据项的恢复通过影子页技术来实现（见第 15.5 节）。当使用影子页时，只有那些实际上更新了的页面需要被重复存储。

尽管采用了这些技术，由于长事务和大数据项面导致的复杂性仍然使恢复过程变得很复杂。因此，我们希望能够对某些非关键性数据不记日志，而依赖于脱机的备份和人的干预。

20.5　实时事务系统

迄今为止所讨论的约束都与存储在数据库中的值有关。在某些应用系统中，约束条件还包括任务必须完成的截止时间。工厂管理、交通控制、调度等都是这类应用系统的例子。当把截止时间考虑在内时，执行的正确性就不再仅仅是数据库一致性的问题了。我们还关心有多少次延误了截止时间，延误了多少截止时间。截止时间的特性如下：

- 硬的。如果任务在截止时间之后完成，它的价值为零。
- 软的。如果任务在截止时间之后完成，它的价值减少了，并随着延误时间的增长其价值趋近于零。

具有截止时间的系统称作实时系统。

实时系统中的事务管理必须将截止时间考虑在内。如果并发控制协议确定让事务 T_i 等待，

那么它可能导致 T_i 延误截止时间。在这种情况下，并发控制协议可能希望强行中止持有锁的进程，而让 T_i 进行下去。然而，抢占的办法必须慎重使用，因为被强行中止的事务所失去的时间（由于回滚和重新启动）可能导致该事务延误它的截止时间。不幸的是，在给定的情况下，很难确定是回滚好还是等待好。

支持实时约束的一个主要困难来自事务执行时间的差异。最好的情况下，要访问的所有数据都在数据库缓冲区中。最坏的情况下，每一次访问都导致将缓冲区页面写到磁盘中（还要先写必不可少的日志记录），然后再从磁盘读入包含要访问的数据的页面。由于最坏情况下所需要的两次或更多次的磁盘访问比最好情况下主存储器访问所花的时间要高好几个数量级，所以如果数据驻留在磁盘上，那么事务执行时间的估算就非常粗略。因此，如果需要满足实时约束，那么常常采用主存数据库。然而，即使数据驻留在主存储器中，还会由于等待封锁、事务中止等而导致事务执行时间的差异。

在高性能事务系统中（参见第 20.3 节），速度是一个关键问题。在实时系统中，不是绝对速度，而是截止时间是最重要的问题。设计一个实时系统需要保证在不要求很大硬件资源的前提下，有足够的处理能力来满足截止时间要求。尽管由于事务管理而导致执行时间的差异，仍然要达到这一目标，这是一个挑战性问题。文献注解给出了有关实时数据库领域中的研究工作的参考文献。

20.6　较弱的一致性级别

在第 14 章中，给出了可串行性概念，并且定义了协议以保证只允许可串行化的调度。可串行性是一个有用的概念，因为它使得程序员在对事务进行编码时不必考虑与并发有关的问题。如果每一个事务在它单独执行时能保持数据库的一致性，那么可串行性就保证了并发执行时数据库的一致性。然而，对于某些应用系统来说保证可串行性的协议可能使得并发度很低，在这些情况下，可以采用较弱的一致性级别。采用较弱的一致性级别给程序员增加了一些保证数据库正确性的负担。

20.6.1　二级一致性

二级一致性的目的是在不必保证可串行性的前提下避免级联中止。二级一致性封锁协议采用与两阶段封锁协议相同的两种封锁模式：共享锁（S）和排他锁（X）。当一个事务对数据项进行访问时，它必须持有适当的锁模式。与两阶段封锁情形不同的是，S 锁可以在任何时候释放，且封锁可以在任何时候申请。排他锁只有在事务提交或中止时才能释放。这种协议不能保证可串行性。事实上，一个事务可能两次读相同的数据项而得到不同的结果。在图 20-5 中，事务 T_3 在 T_4 写 Q 值之前和之后两次读了 Q 的值。

由于二级一致性的不可串行化调度所导致的不一致性的可能性，所以很多应用系统不采用这一方法。

T_3	T_4
lockS(Q)	
read(Q)	
unlock(Q)	
	lockX(Q)
	read(Q)
	write(Q)
	unlock(Q)
lockS(Q)	
read(Q)	
unlock(Q)	

图 20-5　具有二级一致性的非可串行化调度

20.6.2　游标稳定

游标稳定是二级一致性的一种形式，该形式用 Pascal、C、COBOL、PL/I 或 Fortran 等通

用的，面向对象的语言编写，用于程序设计。这样的程序常常在一个关系的各个元组中迭代。游标稳定不是封锁整个关系，而是保证：

- 迭代当前所处理的元组以共享方式封锁。
- 被更新过的元组都以排他方式封锁，直至事务提交。

上述规则保证达到二级一致性，并不需要两阶段封锁，但这些规则不能保证可串行性。实践中，在频繁访问的关系上采用游标稳定的方法可以提高并发度，改进系统性能。尽管调度可能是不可串行化的，但采用游标稳定的应用系统必须用一种保证数据库一致性的方法编码，因此游标稳定的使用仅限于具有简单一致性约束条件的情况。

20.7　事务工作流

工作流是一个活动，它涉及由不同的处理实体所执行的多个任务的相互协同的执行。任务定义要做的某项工作且可以用多种不同的方式来说明，包括文件中的正文描述、电子邮件信息、表格、消息，或计算机程序。执行任务的处理实体可以是人或软件系统（例如：邮件管理器、应用程序、或数据库管理系统）。

图 20-6 显示了这种工作流的一些例子。工作流的一个简单例子存在于一个电子邮件系统中。一条邮件信息的发送涉及几个邮件管理器系统，它们接收和转送邮件信息，直至消息达到它的目的地，并存放在那里。每个邮件管理器执行一个任务，即将邮件转送给下一个邮件管理器。多个邮件管理器上的多个任务可能需要用来为邮件指明路线，将邮件从始发地送到目的地。在数据库和相关文献中用来指工作流的术语包括任务流和多系统应用，有时也将任务称作步骤。

工作流应用	典型的应用	典型的处理实体
电子邮件按路线发送	电子邮件消息	邮件管理器
借贷处理	表格处理	人、应用软件
订单处理	表格处理	人、应用软件、DBMS

图 20-6　工作流的例子

一般说来，工作流可能涉及一个或多个人。例如，有关借贷处理的工作流如图 20-7 所示。想要贷款的人填一个表格，然后借贷工作人员查看这个表格，处理借贷应用的职员通过信用咨询处等机构来查证表格中的数据。当收集到所需的所有信息后，借贷工作人员可能决定同意这笔贷款，这一决定也许还需要经一个或多个高级工作人员批准，然后就可以进行贷款了。这中间的每一个人执行一个任务。在借贷处理任务尚未自动化的银行中，任务间的协同一般是通过从一个工作人员向下一个工作人员传送借贷申请表来达到，在这个过程中附上意见和其他信息。工作流的其他例子包括开支凭单处理、订单处理、信用卡事务处理等。

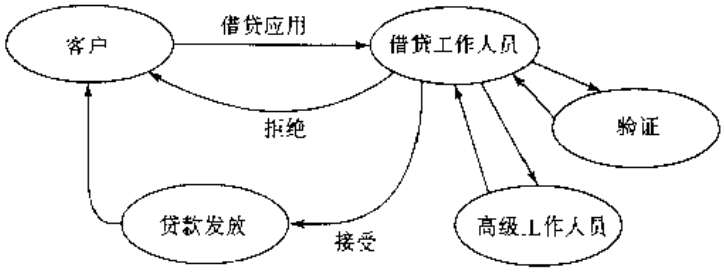


图 20-7　借贷处理中的工作流

如今，有关一个工作流的所有信息一般都以数字化形式存储在一个或多个计算机中，而且，随着网络通信的发展，信息可以很容易地从一台计算机传送到另一台计算机。因此，企业可以自动化地运行他们的工作流。例如，为了使借贷处理所涉及的各项任务自动化，我们可以把借贷应用和有关的信息存储在数据库中。然后，工作流自己将责任从一个人转到下一个人，甚至可能自动地去取所需的信息。人员之间可以通过电子邮件之类的途径来协同他们的活动。

关于工作流自动化，一般我们需要强调两个问题。第一个问题是工作流的说明：详细描述必须执行的任务，并定义执行需求。第二个问题是工作流的执行，在执行工作流时必须提供传统的数据库系统安全性保证，如计算的正确性、数据完整性和持久性。例如，绝不允许在发生系统故障时将借贷申请或凭单丢失，或对它们处理多次。事务工作流的思想是：使用事务的概念并将它扩充到工作流环境中。

这两个问题都很复杂，因为许多企业都使用多个相互独立管理的信息处理系统，在大多数情况下，这些系统是分别开发的，自动地执行不同的功能。工作流活动可能需要几个这样的系统之间的交互，每个系统执行一个任务，并能同人交互。

近年来已开发了多个工作流系统。这里我们在一个相对抽象的层次上研究工作流系统的特性，而不深入到任何一个特定系统的细节。

20.7.1　工作流说明

工作流思想可以追溯到批操作系统的作业控制语言（JCL）。使用 JCL，用户可以将一个作业说明为一组步骤，每个步骤启动一个程序，这些步骤串行地执行。某些步骤可能在一定的条件下执行，例如，仅当前一个步骤成功或失败时执行。因此，可以认为 JCL 说明是工作流说明的一个例子。

为实现工作流管理不必模拟任务的内部。从抽象的角度来看，任务可以使用存放在它的输入变量中的参数，可以检索和更新本地系统中的数据，可以将它的结果存放到输出变量中，对任务的执行状态可以查询。在执行过程中的任何时刻，工作流状态包括构成该工作流的各个任务的状态集合，以及工作流说明中的所有变量的状态（值）。

任务之间的协同可以静态地说明，也可以动态地说明。在静态说明情况下，任务以及任务之间的依赖关系在工作流开始执行前定义。例如，开支凭单工作流中的各个任务可能顺序地包括秘书、经理、会计对凭单的认可，最后才签发一张支票。任务之间的依赖关系可能很简单，如必须先完成前一个任务，然后再开始下一个任务。

如果对这一策略推而广之，即工作流中每一个任务的执行都有一个前提条件，那么工作流中所有可能的任务以及它们之间的依赖关系都是预先知道的，但是只有那些前提条件满足的任务才会被执行。前提条件可以通过如下依赖关系来定义：

- 其他任务的执行状态——例如，“任务 t_j 结束了，任务 t_i 才能开始，”或“如果任务 t_j 提交了，则任务 t_i 必须中止”。
- 其他任务的输出值——例如，“如果任务 t_j 返回一个大于 25 的值，则任务 t_i 可以开始，”或“如果秘书审批任务返回 OK 值，则经理审批任务才能开始”。
- 被外部事件修改的外部变量——例如，“格林威治时间上午 9 点以后任务 t_i 才能开始，”或“在任务 t_j 完成后的 24 小时内必须开始事件 t_i”。

我们可以用通常的逻辑连接符（or、and、not）来将多个依赖关系组合在一起，形成复杂

的调度条件。

任务动态调度的一个例子是电子邮件按路线发送系统。给定邮件消息的下一个任务依赖于消息的目的地址是什么，以及哪些中间路由器正在运行中。

20.7.2　工作流的故障原子性需求

利用工作流语义，工作流设计人员可以说明工作流的故障原子性需求。传统的故障原子性概念会要求当任何任务发生故障时就导致工作流失败。但是，许多情况下，当其中的一个任务发生故障时工作流不一定失败。例如，可以通过在另一个节点执行一个功能上等价的任务来使工作流完成。因此，应该允许工作流设计人员定义工作流的故障原子性需求。系统应该保证工作流的每一次执行都能以一个满足设计人员定义的故障原子性需求的状态终止，我们将这些状态称作工作流的可接受终止状态。工作流的所有其他执行状态构成了一个不可接受终止状态集合，这些状态可能违背故障原子性需求。

可以指定一个可接受终止状态为提交或中止。提交的可接受终止状态是工作流的目标全部达到了的执行状态。与之相反，中止的可接受终止状态是一个合法的终止状态，但工作流未能达到它的目标。如果达到了一个中止的可接受终止状态，那么工作流的部分执行所造成的所有不符合需要的影响都应该按照该工作流的故障原子性需求予以撤消。

即使在发生系统故障的情况下，工作流也必须达到一个可接受终止状态。因此，如果当故障发生时工作流处于一个不可接受终止状态，那么当系统恢复时，必须使它进入一个可接受终止状态（中止或提交皆可）。

例如，借贷处理工作流中，在最终状态，要么告诉贷款申请人他不能获得贷款，要么将贷款发出。在发生故障的情况下，例如验证系统的长时间故障，可以将贷款申请退回给贷款申请人，并附上适当的解释，这种结局构成一个中止的可接受终止。提交的可接受终止应该是或者批准了贷款，或者拒绝了贷款。

一般说来，在工作流达到终止状态之前，一个任务可以提交并释放资源。然而，如果后来这个多任务事务中止了，它的故障原子性可能要求通过执行补偿任务（作为子事务）来撤消已完成的任务（例如，已提交的子事务）所造成的影响。补偿语义要求补偿事务最终成功地完成它的执行，这也许需要若干次重新交付执行。

例如，在开支凭单处理工作流中，由于开始时经理批准了一张凭单，所以部门预算的余额减少了。如果后来由于故障或由于其他原因这张凭单又被否定了，那就需要通过补偿事务来将预算恢复到原来的值。

20.7.3　工作流的执行

多个任务的执行可以由一个协调者来控制，或者由一个称作工作流管理系统的软件系统来控制。工作流管理系统包括一个调度程序、多个任务代理，和一个查询工作流系统状态的机制。一个任务代理控制一个处理实体对一个任务的执行。调度程序是一个对工作流进行处理的程序，它提交不同的任务去执行、监控各种事件、计算与任务间的依赖关系有关的条件。调度程序可以将一个任务提交（给任务代理）执行，也可以要求一个先前提交执行的任务中止。在多数据库事务的情况下，这里所说的任务是子事务，处理实体是局部 DBMS。调度程序依据工作流说明，按依赖关系进行调度，并保证任务的执行能够到达可接受的终止状态。

并发工作流管理系统体系结构有三种方式。集中式方式用单一的一个调度程序去对并发执行的所有工作流的任务进行调度。部分分布式方式对于每一个工作流都有一个调度程序（实例），当并发执行问题可以从调度功能中分离出来时，后一种方式是很自然的选择。完全分布式方式没有调度程序，而由各个任务代理通过相互之间的通信来协同它们的执行，以满足任务间的依赖关系和其他工作流的执行需求。

最简单的工作流执行系统基于电子邮件，采用上述的完全分布式方式。每个节点有一个任务代理，它执行通过电子邮件接收到的任务，执行中也可能会将电子邮件交给人，由人去执行某些动作。当任务在一个节点上完成了，需要到另一个节点去处理时，任务代理发电子邮件消息给下一个节点，消息中包括关于被执行的任务的所有信息。这样的基于电子邮件的工作流系统特别适合于部分时间里断开连接的网络，例如拨号网络。

集中式方式适合于数据存储在集中式数据库中的工作流系统。调度程序通知各种代理（例如人或计算机程序）有任务需要执行，并且跟踪任务的执行情况。集中式方式下跟踪工作流的执行状态比完全分布式方式下要容易。

调度程序必须保证工作流终止于一个可接受终止状态。理想情况是，在打算执行一个工作流之前，调度程序检查该工作流，看它是否可能终止于一个不可接受状态。如果调度程序不能保证一个工作流将终止于一个可接受状态，它应该拒绝这样的工作流说明，而不试图去执行它。作为一个例子，让我们考虑一个工作流，它包括两个任务，表示为子事务 S_1 和 S_2，其故障原子性需求指明要么两个子事务都提交，要么一个也不提交。如果 S_1 和 S_2 没有提供用于两阶段提交的准备提交状态，而且也没有补偿事务，那么就很有可能得到一个子事务提交，而另一个中止的状态，而且没有办法使两个子事务到达同一状态。因此，这样的工作流说明是不安全的，应该拒绝它。

由调度程序进行像上述的安全性检查可能是办不到的或者是不现实的，于是保证工作流的安全性变成了设计工作流说明人员的责任。

20.7.4　工作流的恢复

工作流管理中故障恢复的目标是保证工作流的故障原子性。恢复过程必须保证工作流处理成分（包括调度程序）的任何一个在发生故障的情况下，工作流都能最终到达一个可接受终止状态（提交或中止）。例如，在故障和恢复后，调度程序可以继续处理，就像什么事也没发生一样，从而提供向前的可恢复性。另外，调度程序可以撤消整个工作流（即达到一个全局中止状态）。在上述两种情况下，都可能需要提交某些子事务，或者将某些子事务提交执行（例如，补偿子事务）。

假设工作流中涉及到的处理实体有它们自己本地的恢复系统，能处理本地的故障。为了恢复执行环境上下文，故障恢复例程需要恢复调度程序在发生故障时的状态信息，包括关于每一个任务的执行状态的信息。因此，必须将适当的状态信息作为日志记录到稳定存储器中。

我们还需要考虑消息队列的内容。当一个代理将任务交给另一个代理时，必须保证恰好只交了一次。如果交了两次，那么一个任务可能会被执行两次；如果没有交，那么任务就丢失了。

如果使用持久的消息机制，那么消息存放在稳定存储器中，在故障的情况下也不会丢失。这样由持久的消息系统负责保证即使在发生系统故障时消息也肯定会送到。持久的消息支持可

能由事务处理监控器提供，或直接由操作系统提供，例如由 DEC VMS 操作系统提供的邮箱。代理在它提交之前，先将它要发送的消息写到持久消息队列中，接收端的消息系统也必须将关于请求的日志写到稳定存储器中，这样它就能够发现是否对一个请求接收了多次。

20.8 总结

远程备份系统提供高可用性，使得即使主节点由于火灾、水灾或地震而被摧毁了，事务处理仍能继续进行。远程备份系统的事务处理速度比复制的分布式系统要高。

开发事务处理监控器最初开发时是作为多线程服务器，由单独的一个进程为大量的终端服务。自那时起事务处理监控器演变为建立和管理有大量客户和多个服务器的复杂事务处理系统提供基础设施。它提供各种服务，例如使用户界面应用系统的编写更简化的表示工具、客户请求和服务器响应的持久队列机制、客户消息到服务器的按路线传送、当事务访问多个服务器时的两阶段提交协调等。

在某些系统中提供了大量的主存储器，以提高系统吞吐量。在这样的系统中，日志成为瓶颈。依据成组提交的概念，可以减少向稳定存储器的输出，从而缓和这个瓶颈。

持续时间长的交互式事务的高效管理是一个更加复杂的问题，因为有长时间的等待，还因为有中止的可能性。由于第 14 章中采用的并发控制技术使用等待、或中止、或两者都采用，因此在这里必须考虑其他技术，这些技术必须在不要求可串行性的条件下保证正确性。持续时间长的事务表示为嵌套事务，其最底层是数据库的原子操作。如果一个事务失败了，那么只让活动着的短事务中止。一旦任何短事务恢复了，活动着的长事务就继续进行。我们可以采用第 15 章中讨论的 undo 机制来做恢复。但是，通常可以使用补偿事务，而不是做 undo。补偿事务对事务的失败进行改正、或者补偿，而不导致级联回滚。

在具有实时约束的系统中，执行的正确性不仅包括数据库的一致性，而且还包括对截止时间的满足。不同读/写操作的执行时间的巨大差异使得具有时间约束的系统中事务管理的问题更加复杂化了。

有时候我们可以使用关于一致性的其他概念，它们不保证可串行性，但可以提高性能。游标稳定性和二级一致性保证任何事务都不会读由未提交事务写出的数据。但是，可能发生非可串行化执行。

工作流是一个活动，它包含由不同的处理实体执行的多个任务的相互协同的执行。工作流不仅存在于计算机应用系统中，也存在于几乎所有的企业组织活动中。随着网络通信的发展，以及多个自治的数据库系统的存在，工作流提供了执行涉及多个系统的任务的一个简便的方式。虽然对于这样的工作流应用，通常的 ACID 事务性要求太强了，或不可能实现，但是工作流必须满足事务特性的一个有限集合，保证进程不会终止于一个不一致的状态。

习题

20.1　解释系统崩溃与“灾害”之间的区别。

20.2　解释分布式系统中的数据复制与维持一个远程备份节点之间的区别。

20.3　对于下面的每一个需求，给出持久性程度的最佳选择：

(a) 必须避免数据丢失，但可用性方面的某些损失是可以容忍的。

(b) 事务提交必须快速完成，即使以发生灾害时丢失一些已提交事务为代价也在所不

惜。

(c) 需要高可用性和持久性，但事务提交协议运行较长的时间是可以接受的。

20.4　解释 TP 监控器在管理内存和处理器资源方面如何比通常的操作系统更加有效。

20.5　如果整个数据库能放在主存储器中，我们还需要数据库系统来管理数据码？对你的答案做出解释。

20.6　考虑系统崩溃后的主存数据库系统的恢复。解释下列方法的相对优越性：

- 在重新开始事务处理之前将整个数据库装回到主存储器中。
- 当事务请求数据时将该数据装入主存储器。

20.7　采用成组提交技术时应该在一个组中包括多少个事务？解释你的答案。

20.8　令 T_1 和 T_2 是事务，它们各修改两个数据项。假设一个块中可容纳 100 个日志记录。如果 T_1 和 T_2 分别提交，那么需要多少次块的写出？如果 T_1 和 T_2 作为一个组提交，那么又需要多少次块的写出？

20.9　解释为什么长事务可串行化要求可能是不现实的。

20.10　讨论如果我们允许嵌套事务，那么第 15 章中介绍的每一种恢复模式需要做什么修改？另外，解释如果我们允许多级事务，那会带来什么不同。

20.11　补偿事务的目的是什么？给出使用补偿事务的两个例子。

20.12　高性能事务系统一定是实时系统码？为什么？

20.13　在采用先写日志原则的数据库系统中，最坏情况下读一个数据项需要多少次磁盘存取？解释为什么这对于实时数据库系统的设计人员是一个问题。

20.14　解释采用二级一致性的原因。这种方法有什么缺点？

20.15　工作流系统与数据库系统一样需要并发和恢复管理。为什么我们不能简单地利用一个采用 2PL、WAL 和 2PC 的关系型 DBMS，给出三条原因。

文献注解

Gray 和 Edwards [1995] 给出了关于 TP 监控器体系结构的综述，Gray 和 Reuter [1993] 给出了详细的（并且极好的）关于事务处理系统的教科书式的描述，其中包括几章关于 TP 监控器的描述。我们关于 TP 监控器的描述源于上述两个文献。X/Open [1991] 定义了 X/Open XA 接口。Huffman [1993] 描述了 Tuxedo 中的事务处理。关于用 CICS 进行应用开发有几篇文献，Wipfler [1987] 是其中之一。

关于高性能事务系统已经有许多研究工作。Garcia-Molina 和 Salem [1992] 给出了关于主存数据库的综述。Jagadish 等 [1993] 描述了为主存数据库设计的一个恢复算法。Jagadish 等 [1994] 描述了主存数据库的存储管理。Gray 等 [1975] 提出了二级一致性。

Lynch [1983]、Moss [1982, 1985]、Lynch 和 Merritt [1986]、Fekete 等 [1990a, 1990b]、Korth 和 Speegle [1994]、Pu 等 [1988] 介绍了嵌套事务和多级事务。有几个关于扩展事务模型的定义，包括 Sagas (Garcia-Molina 和 Salem [1987])、ACTA (Chrysanthis 和 Ramamritham [1994])、ConTract 模型 (Wachter 和 Reuter [1992])、ARIES (Mohan 等 [1992])，以及 NT/PV 模型 (Korth 和 Speegle [1994])。Shasha 等 [1995] 讨论了对事务进行分割以达到更高的性能。Beeri 等 [1989] 讨论了嵌套事务系统的并发控制模型。Garcia-Molina [1983] 和 Sha 等 [1988] 讨论了放松可串行性要求的问题。Moss [1987]、Haerder 和 Rothermel [1987]、Rother-

mel 和 Mohan [1989] 讨论了嵌套事务系统的恢复。Weikum [1991] 讨论了多级事务管理。Gray [1981]、Skarra 和 Zdonik [1989]、Korth 和 Speegle [1988, 1990] 讨论了长事务。关于长事务的事务处理见 Weikum 和 Schek [1984]、Haerder 和 Rothermel [1987]、Weikum 等 [1990]、和 Korth 等 [1990a]。Salem 等 [1994] 给出了 2PL 对于长事务处理的扩展，方法是在某些情况下提前释放锁。

关于实时数据库中事务处理的讨论见 Abbot 和 Garcia-Molina [1992]、Dayal 等 [1990]。Barclay 等 [1982] 描述了一个用于电信交换系统的实时数据库系统。实时数据库的复杂性和正确性问题在 Korth 等 [1990b]、Soparkar 等 [1995] 中进行了讨论。实时数据库中的并发控制和调度见 Haritsa 等 [1990]、Hong 等 [1993]、Pang 等 [1995]。Ozsoyoglu 和 Snodgrass [1995] 是对实时数据库和时态数据库研究的综述。

事务处理中的安全性考虑在 Smith 等 [1996] 中进行了讨论。

Berenson 等 [1995] 对 SQL-92 中提供的一致性（或孤立性）级别进行了解释和评论。

我们对工作流的描述沿用了 Rusinkiewicz 和 Sheth [1995] 的模型，该模型讨论了事务工作流的说明和执行。Reuter [1998] 给出了将多个事务组合在一起成为多事务活动的一个方法，称作 ConTracts。Dayal 等 [1990, 1991] 中关于长时间运行的活动的研究中讨论了与工作流有关的一些问题。他们提出将事件 条件 活动规则作为对工作流进行说明的一种技术。Jin 等 [1993] 描述了电信应用系统中的工作流问题。Hollinsworth [1994] 介绍了工作流管理联盟提出的一个工作流参考模型。

第21章 新的应用

我们使用关系数据库已经有20多年了。关系数据库应用中有很大一部分都用于商务领域，支持诸如银行和证券交易所的事务处理、各种业务的销售和预约，以及几乎所有公司都需要的财产目录和工资单管理。我们下面要研究几个新的应用，近几年来它们变得越来越重要。

- 决策支持系统。由于越来越多的数据可联机获得，企业已开始利用这些可获得的数据来对自己的行动作出更好的决策，比如进什么货，以及如何最好地吸引客户以提高销售额。我们可以通过使用简单的SQL查询语句提取大量用于决策支持的信息。但是，最近人们感到需要使用多种数据源的数据，以便在数据分析和数据挖掘（或知识发现）的基础上，更好地来作决策支持。21.1节描述了决策支持系统的概貌，21.2节介绍了数据分析，21.3节介绍了数据挖掘，21.4节介绍了数据仓库工程。
- 空间数据库。空间数据库包括地理数据库和计算机辅助设计数据库，其中前者存储地图及其相关信息，后者存储诸如集成电路或建筑设计之类的信息。空间数据的应用起初是将数据作为文件存储在文件系统中，像早期商务应用一样。但是随着数据的复杂程度、数据量和用户数的增加，利用文件系统存储并检索数据的即兴方法已无法满足使用空间数据应用的需求。空间数据应用需要数据库系统提供的工具，特别是其高效地存储并查询大量数据的能力。某些应用可能还需要其他的数据库特性，比如对部分存储数据的原子性更新、持久性和并发控制。在21.5节中，我们将研究需要对传统数据库系统做什么扩展以支持空间数据。
- 多媒体数据库。在21.6节中，我们研究存储图像、视频和声音之类数据的数据库系统所要求的特性。视频和声音数据最突出的特点是，数据的显示要求读取数据必须以稳定的、预先可确定的速度进行。因此，这种数据被称为连续介质数据。
- 移动数据库。在21.7节中，我们研究新一代的移动计算系统对数据库的要求，比如笔记本电脑和掌上计算设备，它们通过无线数字通信网络连接到基站。这些计算机要求能够在与网络断开连接时仍能操作，而不像第18章所讨论的分布式数据库系统那样。此外由于这种计算系统的存储能力有限，所以需要特殊的内存管理技术。
- 信息检索。伴随着数据库系统领域的发展，信息系统领域近几十年也有了很大的发展。信息系统与数据库系统有许多共性，特别是在二级存储器上存储和检索数据方面。但是，信息系统领域的着重点与数据库系统是不同的，前者着重解决的问题诸如如何基于关键词进行查询、文档与查询的相关性以及文档的分析、分类和索引。在21.8节中，我们研究信息检索系统中出现的问题。
- 分布式信息检索。近几年，计算机网络发展迅猛，现在有上千万的计算机接入了全球互联网。许多用户（包括个人或机构）在这些计算机中存储了信息，希望能与他人共享。分布式信息系统解决的问题是如何表示信息的统一接口，以及如何使用户从浩瀚的互联网信息中查找到感兴趣的信息。我们在21.9节中将讨论分布式信息系统的问题与挑战。

21.1 决策支持系统

数据库应用从广义上可分为事务处理和决策支持两类（19.3.2节）。事务处理系统现在正

被广泛使用，并且公司已经积累了大量由这类系统产生的信息。

例如，公司数据库常常包含了大量关于客户和交易的信息。信息存储的规模可能达到GB级，大型零售连锁店甚至达到TB级。零售商的交易信息可能包括客户的姓名或号码（如身份证号）、购买的商品，所付的价格，以及购买日期。所购商品的信息可能包括商品的种类、生产商、型号、颜色及号码。客户信息可能包括信贷历史，年收入、住所、年龄、甚至教育背景。

这样大的数据库可以作为制定商业决策的信息宝库，譬如决定要进什么商品，或者应打多少折扣。例如，一家零售公司发现西北太平洋地区突然盛行购买法兰绒衬衫，于是意识到这是一种趋势，故而在该地区的商店中开始大量购进这种衬衫。另一个例子是，一个汽车公司通过查询数据库发现购买小型运动汽车的大部分人是年收入超过$50 000的年轻妇女，于是该公司瞄准市场以吸引更多这类妇女来购买它的小型运动汽车，从而避免由于试图吸引其他类型的人购买这种车而浪费资金。在这两个例子中，公司认识到了顾客的行为模式，并利用这种模式制定商业决策。

用于决策支持的数据的存储和检索有几个问题：

- 尽管许多决策支持查询可以用SQL书写，但另一些则无法用SQL表示或简便地表示，因此人们提出了一些SQL的扩展。在21.2节中，我们研究一些这样的扩展，它们可以简化生成汇总数据的工作。
- 数据库查询语言不适合做数据的详细统计分析。有一些程序包，如S++，可以帮助进行统计分析。这些包与数据库有接口，允许大量数据存储在数据库中，并有效地进行检索以便分析。统计分析领域本身就是一门庞大的学科，我们在这里不准备讨论它。有关更多信息可参见文献注解中的参考资料。
- 由人工智能领域发展出来的知识发现技术试图从数据中自动发现统计规则和模式。数据挖掘领域将知识发现的思想与高效实现技术相结合，使之可以用于规模极大的数据库。数据挖掘在21.3节讨论。
- 大公司需要使用各种各样的数据源来制定商业决策。各数据源可能以不同的模式存储数据。由于性能的原因（以及机构控制的原因），这些数据源通常不允许公司的其他部门随便检索数据。

为在各种各样的数据上高效地执行查询，公司已开始构造数据仓库。数据仓库以统一的模式从多数据源收集数据，并放在一个单独的地方。这样，它们就提供给用户一个统一的数据接口。我们在21.4节研究构造和维护数据仓库的问题。

21.2　数据分析

尽管复杂的统计分析最好交由统计程序包来做，但数据库应支持简单常用的数据分析形式。由于数据库中所存储的数据量通常很大，因此如果我们要导出用户能使用的信息，库中数据就需要以某种方式进行汇总。这个工作常常要使用聚集函数。

由于SQL的聚集功能有限，所以不同的数据库各自实现了一些扩展。例如，虽然SQL只定义了几个聚集函数，但许多数据库系统提供了更为丰富的函数集，包括方差、中值等等。某些系统还允许用户增加新的聚集函数。

数据分析中常使用直方图。直方图将一个属性的值划分成区间，并计算聚集，如将每个区间的值求和。例如，工资额的直方图可能要计算工资额在0～20 000、20 001～40 000、40 001～60 000，以及超过60 000的分别有多少人。使用SQL来有效地构造这样的直方图会很

麻烦。我们把它留作习题，供读者验证我们的判断。现已有了对 SQL 语法的扩充，允许在 groupby 子句中使用函数来简化该工作。例如，Red Brick 数据库系统支持的 *N_tile* 函数可把值按百分比划分。考虑如下查询：

select *percentile*, **avg** (*balance*)
from *account*
groupby *N_tile* (*balance*, 10) **as** *percentile*

这里，*N_tile* (*balance*, 10) 将 *balance* 的值分成 10 个连续区间，每个区间有相同多个数值，并且不去掉重复。也就是说，第 1 个区间是最底下值的 10%，而第 10 个区间是最高值的 10%。查询的其余部分在这些区间的基础上执行 groupby，并返回每个区间的平均余额。

统计分析常要求在多个属性上做聚集。例如如下应用，一个商店想要知道什么样的服装流行。假设服装按颜色和尺码做分类，并且有模式为 *Sales* (*color*, *size*, *number*) 的关系 *sales*。要想通过颜色（淡色或深色）和尺码（小号、中号、大号）分析销售情况，经理就需要参阅像图 21-1 中表格那样表示出来的数据。

图 21-1 中的表是交叉表格（交叉表）的例子，许多报告书写器都提供对这种表格的支持。本例中，数据是二维的，因为它们基于两个属性：*size* 和 *color*。一般地，数据可表示成多维数组，数组的每个元素有一个值。这种数据称为多维数据。

	小号	中号	大号	总计
浅色	8	35	10	53
深色	20	10	5	35
总计	28	45	15	88

图 21-1 有关 *size* 和 *color* 的 *sales* 的交叉表格

交叉表格中的数据不能由单独一条 SQL 查询语句生成，因为要从不同的级别进行合计。另外，我们容易发现交叉表格不同于关系表。通过引入特殊值 all 表示分项合计，可以用关系形式表示数据，如图 21-2 所示。

考虑元组 (Light, all, 53) 和 (Dark, all, 35)。这两个元组是通过消除在 *size* 上具有不同值的各个元组，并将 *number* 的值替换成以 sum 做聚集的值得到的。值 all 可以被看作是代表 *size* 的值的集合。通过聚集从较细粒度数据变到较粗粒度数据称作 rollup。本例中，我们对属

颜色	大小	数目
Light	Small	8
Light	Medium	35
Light	Large	10
Light	**all**	53
Dark	Small	20
Dark	Medium	10
Dark	Large	5
Dark	**all**	35
all	Small	28
all	Medium	45
all	Large	15
all	**all**	88

图 21-2 图 21-1 中数据的关系表示

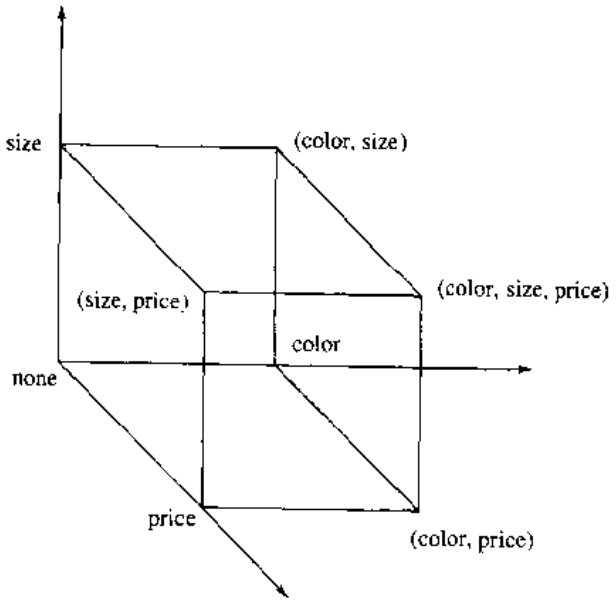


图 21-3 三维数据立方体

性 size 做了 rollup。相反的操作，即从较粗粒度数据变到较细粒度数据，称作 drill down。显然，较细粒度数据不能从较粗粒度数据生成，它们必须从原始数据生成，或从粒度足够细的汇总数据生成。

对元组分组做聚集的方式很多，这可以很容易地从图 21-2 中的表得到验证。事实上，对一个 n 维的表，rollup 可以对 n 维的 2^n 个子集中的任一个进行。考虑关系 *sales* 的三维情况，三个维分别是属性 size、color 和 price。图 21-3 将该关系的属性的子集表示成三维立方体的顶角，rollup 可以在这些属性的任一个子集上进行。一般地，n 维关系的属性的子集可被表示成相应 n 维立方体的顶角。

尽管我们可以用 SQL 生成图 21-2 中那样的表，但这样做是很麻烦的。查询语句要使用 union 操作，并且语句可能很长。从一个包含 all 以外其他元组的表生成包含 all 的元组，我们把这个留作习题。

已经有人提出用 cube 操作符扩展 SQL 语法。例如，如下的扩展 SQL 查询语句可生成图 21-2 所示的表：

select *color*, *size*, **sum** (*number*)
from *sales*
groupby *color*, *size* **with cube**

21.3　数据挖掘

数据挖掘这个概念广义上讲是指从大量数据中发现有关信息，或“发现知识”。与人工智能中的知识发现类似，数据挖掘试图自动从数据中发现统计规则和模式。但是，数据挖掘与机器学习的不同在于它处理的是大量数据，它们主要存储在磁盘上。

从数据库中发现的知识可以用一个规则集表示。下面是规则的一个例子，此规则非形式地表示为：“年收入超过 $50 000 的年轻妇女是最有可能购买小型运动汽车的人。”

我们首先在下节描述用来表示知识规则的结构，然后在 21.3.2 节考虑几类数据挖掘应用，以解释数据挖掘的需要。我们用如下两个模型之一从数据库中发现规则：

- 在第一个模型中，用户直接参与知识发现的过程。该模型在 21.3.3 节中讨论。
- 在第二个模型中，系统通过检测数据的模式和相互关系，自动从数据库中发现知识。该模型在 21.3.4 节中有详细的阐述。

知识发现系统可能包含这两个模型的特点，它自动发现某些规则，并且由用户制导规则发现的过程。

21.3.1　用规则表示知识

规则提供一个表示各种知识的公共框架。规则具有如下一般形式

$$\forall \overline{X} \text{ 前件} \Rightarrow \text{后件}$$

其中 $\overline{X}$ 是相应取值范围上的一个或多个变量的列表。设数据库包含一个关系 *buys*，它指明在每笔交易中购买了什么。下面是一个规则的例子：

$$\forall \textit{transactions}\ T,\ \textit{buys}\ (T,\ \textit{bread}) \Rightarrow \textit{buys}\ (T,\ \textit{milk})$$

这里，T 是一个变量，其取值范围是所有交易的集合。该规则指出，如果在关系 *buys* 中有元组 $(t_i,\ bread)$，则关系 *buys* 中也一定有元组 $(t_i,\ milk)$。

变量的取值范围是指该变量可以取的值的集合。规则中各变量的取值范围的叉乘构成了个体总数。许多数据挖掘系统限制规则只能有一个变量。

规则有相应的支持度和置信度。

• 支持度是同时满足规则前件和后件的情况占个体总数的百分比的测度。

例如，如果所有交易中0.001%的交易涉及同时购买牛奶和面包，则规则

$$\forall \textit{transactions}\ T,\ \textit{buys}\ (T,\ \textit{bread}) \Rightarrow \textit{buys}\ (T,\ \textit{milk})$$

的支持度就很低。该规则在统计意义上不重要，因为可能只有一笔交易既买了面包又买了牛奶。商家往往对支持度低的规则不感兴趣，因为它只涉及了少量客户，不值得花力气处理。另一方面，如果所有交易中50%的交易涉及同时购买牛奶和面包，则支持度相对较高，规则就值得注意。具体支持度最低多少为好依赖于应用。

• 置信度是在前件为真时，后件有多大可能为真的测度。例如，规则

$$\forall \textit{transactions}\ T,\ \textit{buys}\ (T,\ \textit{bread}) \Rightarrow \textit{buys}\ (T,\ \textit{milk})$$

的置信度为80%是指购买面包的80%的交易也购买牛奶。置信度低的规则是没有意义的。在商业应用中，规则的置信度往往大大低于100%，而在其他领域如物理中，规则却可能有较高的置信度。

21.3.2 数据挖掘问题的类型

分类和关联规则是两类重要的数据挖掘问题。

分类是指找出规则将数据分到互不相交的组。例如，假设一个信用卡公司决定是否给某人信用卡。该公司有这个人的各种信息，例如她的年龄、教育背景、年收入、当前债务情况，以及居住地点，这些信息可以用来帮助做决定。

这些信息中有的可能与申请人的信用有关，有的则可能无关。为作出决定，公司给现有客户的抽样集中的每个客户一个信用级别，分为优秀、良好、一般或差。信用的评定依据该客户的支付历史。然后，公司再试图根据个人信息，而不是根据实际支付历史（这对于新客户来说是无法得到的）将现有客户分成优秀、良好、一般或差。让我们只考虑两个属性：教育程度（所获得的最高学历）和收入。规则可能有如下形式：

$$\forall \textit{person}\ P,\ P.\textit{degree} = \textit{Masters}\ \textbf{and}\ P.\ \textit{income} \geqslant 75000 \Rightarrow P.\textit{credit} = \textit{excellent}$$

$$\forall \textit{person}\ \boldsymbol{P},\ P.\textit{degree} = \textit{Bachelors}\ \textbf{or}\ P.\textit{income} \geqslant 25000\ \textbf{and}\ P.\textit{income} < 75000) \Rightarrow P.\textit{credit} = \textit{good}$$

其他信用级别（一般和差）也可用类似的规则表示。

其他分类的重要应用包括贷款审批、计算保费、根据关于顾客的统计信息决定商店是否进某种商品，等等。

零售商店常常对人们购买的不同商品之间的关联感兴趣。例如，买面包的人很可能也买牛奶，这是我们前面看到的规则所表示的关联：

$$\forall \textit{transactions}\ T,\ \textit{buys}\ (T,\ \textit{bread}) \Rightarrow \textit{buys}\ (T,\ \textit{milk})$$

它具有相应的置信度和支持度。

关联信息可以几种方式来使用。商店可能决定将面包和牛奶摆放得很近，帮助购买者更快地购买。商店也可能将它们放在货架的两头，而将其他相关商品放在中间以吸引顾客在从货架

一头走到另一头时购买。商店可能对相关联的商品中的一种打折，但对另一个不打折，因为顾客可能不管怎样都会购买它。

另一类重要的数据挖掘应用是序列相关性。时间序列数据，如在连续几天内的股票价格是序列数据的一个例子。股市分析试图找出股市价格序列的相关性。这种相关性的一个例子是如下规则："每当债券利率上涨，股票价格就在两日内下跌。"发现序列间的这种关系能够帮助我们做出明智的投资决策。

21.3.3　用户制导的数据挖掘

在用户制导的数据挖掘模型中，用户对发现规则负有主要责任，数据库系统起到一个支持的作用。一般地说，用户提出一个猜想，然后在数据库上运行一些测试以验证或反驳该猜想。例如，用户可能猜想有硕士学历的人信用级别为优秀的可能性最大，然后用户用数据库验证该猜想。数据可能真的表明具有硕士学历的人比其他人更有可能具有优秀的信用级别。

用户可能还会有一个新的猜想，或者修正现有的猜想，并且用数据库验证。也就是说，可能存在多次循环，这是由于用户在验证修正后的猜想的同时，还在不断地修正猜想。

例如，由于已经验证出具有硕士学历的人与其他学历的人相比，具有优秀信用级别的可能性要稍高一些，同时这类人的某个子类更有可能具有优秀的信用级别，于是用户就会猜想具有硕士学历并且年收入为 $75 000 或更高的客户可能具有优秀的信用级别。该规则的置信度和支持度可由数据库导出。这时，置信度可能会很高（接近 1），用户到这里就可以停止了，并导出了如下形式的规则

$$\forall \textit{people } P,\ P.\textit{degree} = \textit{Masters } \textbf{and } C.\textit{income} \geqslant 75000$$
$$\Rightarrow C.\textit{credit} = \textit{excellent}$$

它具有适当的置信度和支持度。

数据可视化系统帮助用户检查大量数据，并通过视觉发现模式。数据的可视化显示，如地图、图表及其他图形表示，使得数据简洁地呈现给用户。一个图形化的屏幕所能表示的信息可以相当于很多个文本形式的屏幕所表示的信息。

例如，也许用户想要知道工厂的生产问题是否与工厂的位置有关，那么用户可将有问题的地区在地图上用特殊的颜色表示，譬如红色。这样用户就可以很快地找到发生问题的地区。然后用户猜想为什么在那些地区发生问题，并通过数据库定量地验证这些猜想。

另一个例子是，值的信息可以用颜色表示，并且显示时可以小到屏幕区域上的一个像素。要寻找两样商品之间的关联，我们可以使用二维像素矩阵中的每个行和列表示一件商品。同时买两件商品的交易的百分比可以表示为像素的色彩强度。关联强的商品在屏幕上表示为明亮的像素，在较暗的背景下可以很容易被发现。

21.3.4　规则的自动发现

有关自动发现规则的研究很大程度上是受人工智能领域在知识学习方面研究的影响。其主要的区别在于数据库中处理的数据量，以及是否需要访问磁盘。已经有一些具体的数据挖掘算法用于高效地处理存放在磁盘上的大量数据。

规则发现的方式依赖于数据挖掘应用的类型。我们用两类应用阐述规则发现：分类和关联。

1. 分类规则的发现

发现分类规则的过程开始于一个数据样本集（称为训练集）。训练集中的每个元组分到哪个组是已知的。例如，信用卡应用的训练集可能是现有的客户，他们的信用级别由其支付历史决定。实际的数据，或称个体总数，可能包括所有人，即也包括那些不属于现有客户的人。

上节关于找出哪类人具有优秀的信用级别的例子不仅阐明了人如何发现规律，而且提示了数据挖掘系统如何发现规则。

最初的数据挖掘系统只包含单个属性上的一个简单条件。属性 *degree* 上可能的条件是 *P*.*degree* = *None*，*P*.*degree* = *Bachelors*，*P*.*degree* = *Masters* 及 *P*.*degree* = *Doctorate*，如果以上这些是所有可能的学历。对于数字型值的属性，数据挖掘系统将可能的值分为若干个区间。*salary* 属性值可以分为区间 0～25 000、25 000～50 000、50 000～75 000，以及超过 75 000。也就是说，如果我们考虑某个属性，则该属性上的条件将元组的集合分成互不相交的组。

如果对某个属性，每个组中的所有或大多数元组都具有相同的分类值（这里的例子指信用级别），就可以输出一组基于该属性的规则，数据挖掘过程终止。但这种情况多半不会发生，因为单是属性 *degree* 或 *income* 上的规则不足以将数据适当地分类。

这种情况下要选择一个属性，通常选择最可用来划分数据的属性。数据根据这个属性上的条件进行分组，然后在各自的组中再根据其他属性分类。假设 *degree* 被选为分类所依据的属性。考虑对应于 *Masters* 的分类。下一层的划分就要根据其他属性中的一个，这里我们用属性 *income*。*Masters* 组中规则的前件除 *income* 上的条件外，还有 *degree* = *Masters* 这样的条件。其他分组，如 *degree* = *None* 等等，各自以同样的方式进行划分。

这种方法自顶向下地生成分类树。本例所对应的部分分类树如图 21-4 所示。树中的每个结点根据一个属性将数据划分成组。当属性已能适当地划分数据或所有属性均已被考虑进去时树中路径的构造停止。本例中，根据属性 *degree* 划分的每个组中，由 *income* 定义的分组足以划分元组。对每个基于 *degree* 的分组，树的构造过程在这一级终止。通常，树的不同分枝可能生成不同的层次。

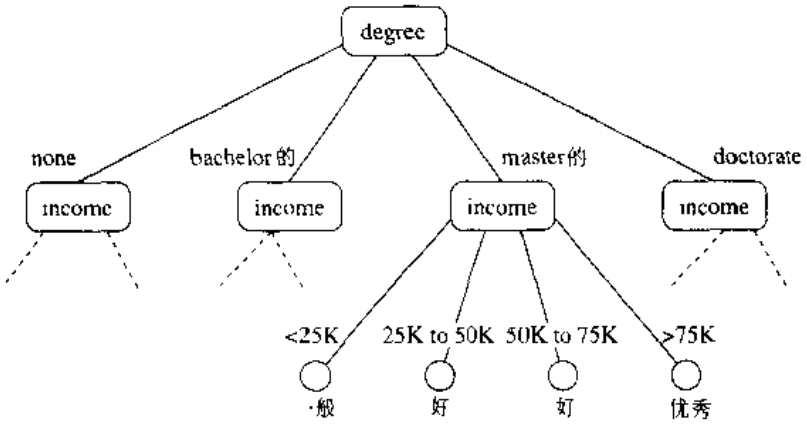


图 21-4　分类树

人们已经对这个基本技术提出了许多改进。可以通过合并规则减少生成规则的数目，例如当相邻的区间属于相同的分类组时。另外可以通过一次扫描元组生成树的多个分枝的技术来减少 I/O 开销。更多的信息可参考文献注解。

2. 关联规则的发现

考虑前面关联规则的例子：

$$\forall \textit{transactions}\ T,\ \textit{buys}\ (T,\ \textit{bread}) \Rightarrow \textit{buys}\ (T,\ \textit{milk})$$

我们可以通过用一个位图代表一笔交易，高效地导出类似这样的规则，位图中的一位代表商店中一件令人感兴趣的商品，这种方法在所有令人感兴趣的商品数目不很大时是合理的。即使原来的商品集很大，用户通常也只需要具有强支持度的那些规则，它们只涉及在足够大百分比的交易中都被购买的商品。因此，相关商品的集合能够很快求出来，而且该集合可能很小。

现在，为找出如下形式的关联规则

$$\forall \textit{transactions}T,\ \textit{buys}\ (T,\ i_1)\ \textbf{and}\ldots\textbf{and}\ \textit{buys}\ (T,\ i_n) \Rightarrow \textit{buys}\ (T,\ i_0)$$

必须考虑相关商品集的所有子集，并且对于每个集合，必须检查在足够多的交易中，该集合中的所有商品是否都被购买了。

如果集合的数目较少，对交易做一遍扫描就足以检测所有集合的支持度。可以为每个集合维护一个计数器，初值设成0。每当取出一笔交易，交易位图中的所有相应位都置位的商品集的计数器加1。一遍扫描结束时计数器值足够高的集合对应于具有高关联度的商品。

如果集合的数目较多，则处理每笔交易的开销就会变得非常大。现已提出的一些技术采用多遍数据库扫描，每遍扫描只考虑部分集合。一旦某个集合在一小部分交易中出现时就被消去，则其所有超集都不必加以考虑。

21.4　数据仓库工程

大型公司在多个地点都有机构，每个机构都可能生成大量数据。例如，大型零售连锁店在成百上千个地方都有商店，而保险公司可能有来自数千个地区公司的数据。另外，大型机构有复杂的内部组织结构，因此不同数据会放在不同地点，或不同的操作系统上，并且模式也可能不同。例如，制造问题的数据和客户投诉数据可能存储在不同的数据库系统中。公司决策制定者需要访问来自所有这些数据源的信息，对各个数据源做查询既麻烦，且效率又低。另外，数据源可能只存储当前数据，而决策制定者说不定还需要访问过去的数据。例如，有关过去一年中购买模式如何变化的信息就很重要。数据仓库提供了解决这些问题的方案。

数据仓库是从多数据源收集来的信息的仓储（或转储），它在一个地点以统一的模式存储。一旦收集来，数据将长期保存，以支持对历史数据的访问。因此，数据仓库提供给用户一个统一的数据接口，使得决策支持查询的书写更为容易。另外，通过从数据仓库访问用于决策支持的信息，决策制定者保证了联机事务处理系统不会受决策支持工作负载的影响。

图21-5给出了一个典型的数据仓库的体系结构，并且表示出了数据的收集、数据的存储以及查询和数据分析支持。构造数据仓库面临的问题如下：

- 何时以及如何收集数据。在源驱动数据收集体系结构中，数据源要么连续地在事务处理发生时传送新信息；要么阶段性地，譬如每天晚上传送新信息。在目标驱动体系结构中，数据仓库阶段性地向源发送对新数据的请求。

除非对源的更新通过两阶段提交在数据仓库中做了复制，否则数据仓库不可能总是与源同步。两阶段提交通常因开销太大而不被采用，所以数据仓库常会保留稍微有点儿过时的数据。但这对于决策支持系统来说通常不是问题。

- 采用什么模式。各自独立构造的数据源可能具有不同的模式。事实上，它们甚至可能

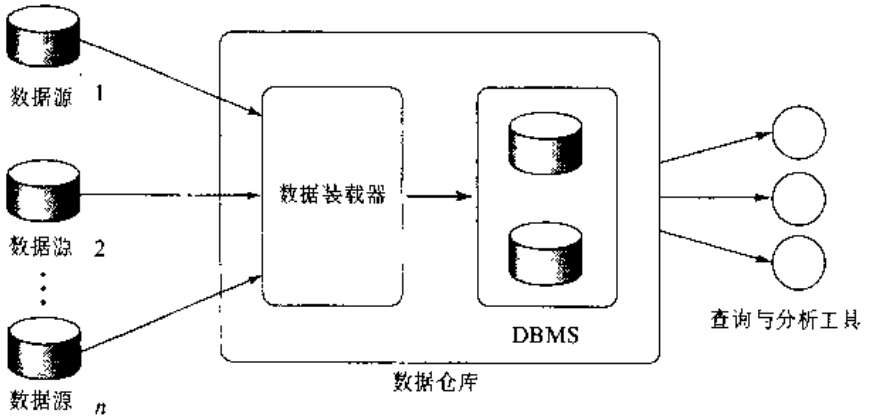


图 21-5 数据仓库的体系结构

使用不同的数据模型。数据仓库的部分任务就是做模式集成，并且在数据存储前将数据按集成的模式转化。因此，存储在数据仓库中的数据不仅仅是源中数据的拷贝，同时它们也可被认为是源中数据所存储的视图（或实体化的视图）。

• 如何传播更新。数据源中关系的更新必须被传至数据仓库。如果数据仓库中的关系与数据源中的一模一样，传播过程则是直接了当的。否则，传播更新的问题基本上是视图维护问题，这已在 3.7.1 节中做了简要讨论。

• 汇总什么数据。由事务处理系统产生的原始数据可能太大以致于不能联机存储。但是，我们可以通过维护对关系做聚集而得到的汇总数据回答很多查询，而不必维护整个关系。例如，我们不是存储每件服装的销售数据，而是按类存储服装的销售总额。

假设关系 r 已由汇总关系 s 代替，用户可能仍被允许做一些查询，就像关系 r 还联机存在一样。如果该查询只需要汇总数据，就有可能将它转化成对 s 的等价查询。这个转化可按如下方式完成。存储的汇总关系 s 定义为原关系上的一个视图，但是，s 是实体化的视图，而不是随时需要计算的视图，随后查询优化器必须根据对给定查询的分析以及 s 的视图定义，决定是否可以在做相应改动后，用 s 代替给定查询中的 r。更多的信息参见文献注解。

21.5 空间和地理数据库

空间数据库存储有关空间位置的信息，并且对高效查询和基于空间位置的索引提供支持。例如，假设需要在数据库中存储一组多边形，然后查询该数据库以找出与给定多边形相交的所有多边形。我们不能使用标准的索引结构（如 B 树或散列索引）高效地回答这样一个查询。针对这项工作，空间数据库会使用专用的索引结构，如 R 树（我们将在后面研究）。

有两种空间数据库特别重要：

• 设计数据库或计算机辅助设计（CAD）数据库是用于存储设计信息的空间数据库，这些信息主要是关于物体（如建筑、汽车或飞机）是如何构造的。另一个计算机辅助设计数据库的重要例子是集成电路和电子设备设计图。

• 地理数据库是用于存储地理信息（如地图）的空间数据库。地理数据库常称为地理信息系统。

21.5.1　几何信息的表示

图 21-6 表示了各种几何结构如何以规范化的形式在数据库中表示。我们要强调的几何信息的表示可以有多种不同方式，这里只描述其中几种。

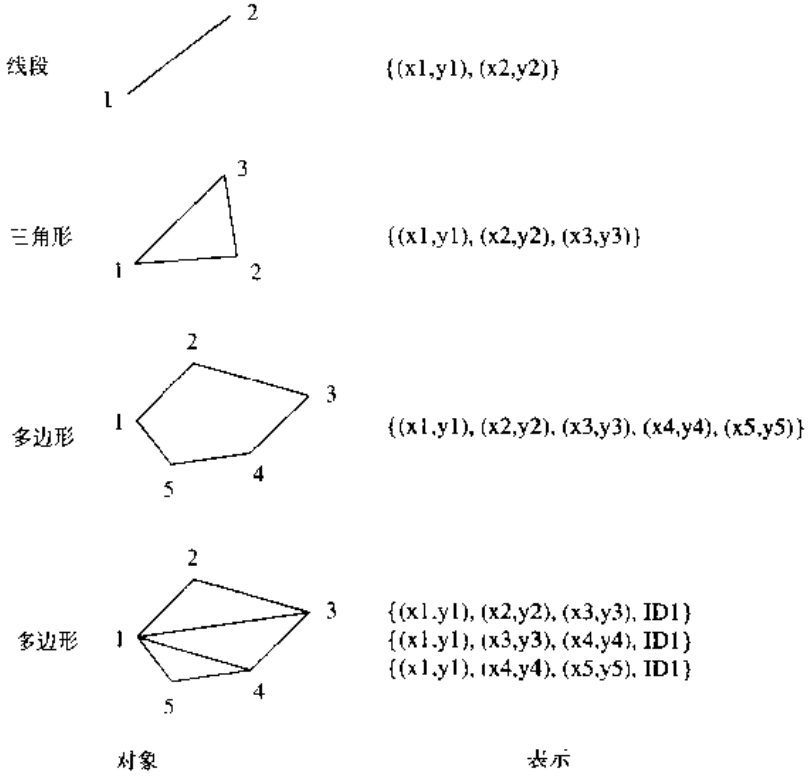


图 21-6　几何结构的表示

线段可以由其端点的坐标表示。我们可以将任意曲线近似地用划分后的线段序列表示。为使用固定大小的元组，我们可以给曲线一个标识，然后用单个元组表示每个线段，并且在每个元组中放入该曲线的标识。这种表示对二维特征如道路是有用的，这里，道路的宽度相对于整张地图的规模来说，小到足以将其认为是二维的。对于地图数据库，一个点的两个坐标是它的纬度和经度。

我们可以通过顺序列出多边形的顶点表示一个多边形，如图 21-6 所示。顶点的序列实际上指出了多边形区域的边界。为简化起见，我们只考虑闭合多边形，也就是序列中的起始顶点与序列中的结束顶点相同的那些多边形。边界的内部形成了多边形区域。

在另一种表示方法中，闭合多边形可以被分割成一组三角形，如图 21-6 所示。该过程称为三角化，任何闭合多边形都可以被三角化。与曲线的表示方式类似，复杂的多边形被赋予一个标识，它所分割出来的每个三角形都具有该多边形的标识。由曲线（如圆）包围起来的复杂二维特征可以用多边形近似表示。

三角化表示是为存储在第一范式的关系格式中而设计的。非第一范式表示，如用顶点或边的序列表示多边形，在实际中也有使用，它由底层数据库支持。这种基于序列的表示对查询处理往往是方便的。

三维空间中点和线段的表示类似于其在二维空间中的表示，唯一的区别是点多了 z 坐标。

同样，从二维到三维，平面图形如三角形、矩形及其他多边形的表示没有太大变化；四面体和长方体可以用类似三角形和矩形的方法表示。可以通过将多面体分割成若干四面体来表示任意多面体，这类似于对多边形的三角化。也可以通过列出多面体所有的面来表示多面体，每个面本身是一个多边形，使用这种表示我们还需要指出该面的哪一侧是多面体的内侧。

21.5.2 设计数据库

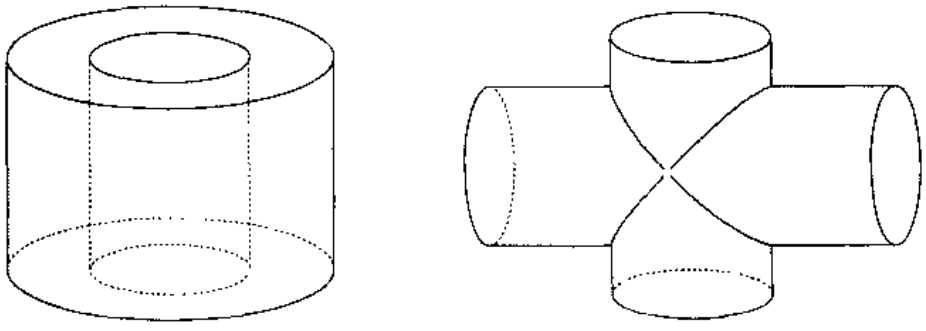
计算机辅助设计（CAD）迄今已使用了许多年。传统上，CAD系统在编辑或做其他处理时将数据存在内存中，并且在一次编辑结束时将数据写回文件。这种方式的缺点包括将数据由一种形式转化为另一种形式的开销(编程的复杂性和时间开销)，以及即使只需要其中一部分也得读入整个文件的需要。对于大型设计，譬如大规模集成电路的设计，或整个飞机的设计，也许根本不可能将整个设计放到内存中。面向对象数据库的设计者很大程度上是受CAD系统的数据库需求的激发。设计中的组件被表示成对象，并且对象间的联系指明了设计是如何构造的。

设计数据库中存储的对象通常是几何对象。简单的二维几何对象包括点、线、三角形、矩形和一般多边形。复杂的二维对象可以由简单对象通过并、交、差操作得到。同样复杂的三维对象可以由简单对象如球、圆柱和立方体的并、交、差操作得到，如图21-7所示。三维表面也可以用线框模型表示，它主要是将表面模拟成一组简单对象，如线段、三角形和矩形。

设计数据库也存储有关对象的非空间信息，如构造对象的材料。通常可以用标准数据建模技术表示这些信息。这里我们只考虑空间的那部分。

我们必须对设计执行多种空间操作。例如，我们的任务可能是取出设计中的某部分，它对应于我们感兴趣的特定区域。21.5.5节将讨论的空间索引结构对这类任务很有用。这种空间索引结构是多维的，它处理的是二维或三维的数据，而不是处理B^+树所提供的那种简单的一维排序。

空间完整性约束，如“两个水管不应在同一地点”，对设计数据库是很重要的。如果设计是手工制做的，这种错误就常会发生，并且只有当原型制造出来时才会被发现，因此这些错误修改起来代价很高。支持空间完整性约束的数据库可以帮助人们避免设计错误，以保证设计的一致性。这种完整性检查的实现也要依赖于高效多维索引结构的可用性。



a) 圆柱的差　　b) 圆柱的并

图21-7 复杂三维对象

21.5.3 地理数据

地理数据本质上是空间的，但与设计数据相比在几个方面有所不同。地图和卫星图像是地

理数据的典型例子。地图不仅可提供位置信息，如边界、河流和道路，而且还可提供许多与位置相关的详细信息，如海拔、土壤类型、土地使用和年降雨量。

地理数据可以分为两类：

• 光栅数据。这种数据由二维或更高维的位图或像素图组成。二维光栅图像的典型例子是云层的卫星图像，其中每个像素存储了特定地区的云层的可见度。这种数据可以是三维的，例如，同样在卫星的帮助下测得的不同地区在不同高度上的温度。时间也可以是一维的，例如，不同时间上不同点的表面温度。设计数据库通常不存储光栅数据。

• 矢量数据。矢量由基本几何对象构成，如点、线段、三角形和其他二维多边形，以及圆柱、球、长方体和其他三维多面体。

地图数据常以矢量形式表示。河流和道路可以表示成多个线段的并；州和国家可以表示成多边形；拓扑信息如高度，可以通过将表面分成覆盖等高区域的多边形来表示，每个多边形对应一个高度值。

1. 地理数据的表示

地理特征如州和大型湖泊，可表示成复杂多边形。某些特征如河流，可以表示成复杂曲线或复杂多边形，后者取决于其宽度是否重要。

关于地区的地理信息如年降雨量，可以表示成一个数组，也就是以光栅的形式。为提高空间利用率，该数组可以压缩的形式存储。在 21.5.5 节中，我们将研究这种数组的另一种表示，它使用称为四分树的数据结构。

如前所述，我们可以矢量的形式用多边形表示区域信息，每个多边形表示一个区域，区域内部的数组值是相同的。某些应用中矢量表示比光栅表示更简洁，而且对于某些任务它也更精确，如表示道路，将区域分成像素（可能非常大）会导致位置信息精确度的损失。但是，矢量表示不适于数据本来就是基于光栅的那类应用，如卫星图像。

2. 地理数据的应用

地理数据库有多种用途，包括车辆导航系统、公共服务设施的分布网络信息，如电话、电源和供水系统，以及为生态学家和规划者提供的土地使用信息。

车辆导航系统存储关于道路和服务的信息供司机使用。这种信息包括道路图、道路上的速度限制、道路状况、道路间的连接及单行限制。最简单地，这些系统可以用来生成联机地图供人们使用。联机地图一个重要的好处是它可以简便地缩放至需要的大小，也就是放大或缩小以定位重要的特征。有了这些有关道路的附加信息，地图可用来自动产生旅行计划。用户可以查询有关服务的联机信息来找出提供所需服务和价格范围的酒店、加油站或饭馆等等。

车辆导航系统通常是移动的，并且装在车辆的仪表板上。移动地理信息系统的一个有用的附件是全球定位系统部件，它使用 GPS 卫星的广播信息，以几十米范围的精度确定当前位置。拥有这样的系统，司机永远也不会迷路，GPS 部件以{纬度，经度，海拔} 的形式确定位置，并且通过查询地理数据库找出车辆当前在哪儿以及在哪条道路上。

用于公共设施信息的地理数据库随着地下电缆和管道网络的增长变得越来越重要。由于缺少详细地图，一个设施的工程可能毁坏另一个设施的电缆，导致服务的大规模破坏。地理数据库加上定位系统，可以保证避免这种灾难。

到现在为止，我们已经解释了人们为什么使用空间数据库。以下内容将研究技术细节，如空间信息的表示和索引。

21.5.4 空间查询

有几种查询都涉及空间位置。

• 临近查询请求找出特定位置附近的对象。找出给定地点给定距离内的所有饭馆的查询是临近查询的例子。最近邻居查询请求找出离特定点最近的对象。例如，我们可能想要找出最近的加油站。注意这个查询不必指定距离的限制，我们即使不知道最近的加油站有多远，也可以做这个查询。

• 区域查询处理的是空间区域。这种查询请求找出部分或全部位于指定区域内的对象。例如找出给定城镇的地理边界内的所有零售商店的查询。

• 查询可能也要求区域的交和并。例如，给出区域信息，如年降雨量及人口密度，查询可能要求找出所有年降雨量低并且人口密度高的区域。

计算区域的交的查询可以被看作是计算两个空间关系的空间连接，一个关系代表降雨量，另一个代表人口密度，而位置作为连接属性。一般地说，如果给出两个关系，每个都包含空间对象，那么两个关系的空间连接生成若干对相交的对象，或这些对象的相交区域。

现在已经提出了几种对矢量数据高效计算空间连接的连接算法。尽管可以采用嵌套循环连接和索引嵌套循环连接（空间索引），但对空间数据不能采用散列连接和排序归并连接。基于两个关系上空间索引结构的坐标遍历的连接技术也已被提出。更多信息参见文献注解。

一般地，空间数据的查询可能是空间和非空间请求的结合。例如，我们可能想要找出可以提供素食的，且一次就餐收费少于$10的最近的饭馆。

由于空间数据本来是图形化的，因而我们通常使用图形化的查询语言对它们做查询。这种查询的结果也以图形显示，而不是以表的形式。用户可以调用界面上的各种操作，例如通过指向或点击曼哈顿的西郊查看这个地区、放大或缩小、选择条件指定显示内容（例如，在低犯罪率地区的多于三个卧室的房子），等等。图形界面构成了前端，现已提出用 SQL 扩展作为后端，允许关系数据库存储和检索空间信息，并且也允许混合空间和非空间条件的查询。扩展包括对抽象数据类型的支持，如线、多边形、位图，还包括对空间条件的支持，如包含或重叠。

21.5.5 空间数据的索引

高效访问空间数据需要索引。传统的索引结构，如散列索引和 B 树，不适合空间数据，因为这些索引结构只处理一维数据，而空间数据通常是二维或更高维的。

1.k-d 树

要理解如何对由二维或更高维构成的空间数据加索引，先考虑一维数据中点的索引。树结构，如二叉树和 B 树，是不断地将空间分成更小的部分。例如，二叉树的每个内部结点将一维区间分成两个。位于左边区域的点进入左子树，位于右边区域的点进入右子树。在平衡二叉树中，划分的选择应使得落入每个区间的点的数目大约是存储在子树中的点的数目的一半。同样，B 树的每一层将一个一维区间分裂成多个部分。

我们可以用直觉为二维空间及更高维空间创建树结构。一种称为 k-d 树的树结构是用于多维索引的一种早期结构。k-d 树的每一层将空间分成两个。树的顶层结点按一维进行划分，下一层结点按另一维进行划分，以此类推，各个维循环往复。划分要使得在每个结点存储在子树中的点大约有一半落入一侧，而另一半落入另一侧。当一个结点中的点数少于给定的最大点数

时，划分结束。图 21-8 表示了二维空间中的一组点，以及这些点的 k-d 树表示。每条线对应于树中的一个结点，叶结点的最大点数设为 1。图中的每条线（除了外面的方框）对应于 k-d 树的一个结点，图中线的标号表示了相应结点在树中的层数。

k-d-B 树扩展了 k-d 树，允许每个内部结点有多个子结点，就像 B 树扩展了二叉树以减小树的高度。k-d-B 树比 k-d 树更适合于辅助存储器。

2．四分树

二维数据的另一种表示是四分树。用四分树划分空间的例子如图 21-9 所示。点的集合与图 21-8 中的一样。四分树中的每个结点对应于空间中的一个矩形区域，顶层结点对应于整个目标空间。四分树中的每个非叶结点将其区域分成四个相同大小的象限，相应地，每个结点有四个子结点对应于四个象限。叶结点有零至某固定最大值那么多个点。如果一个结点的区域有多于最大值的点数，该结点就需要创建子结点。在图 21-9 所示的例子中，叶结点的最大点数设为 1。

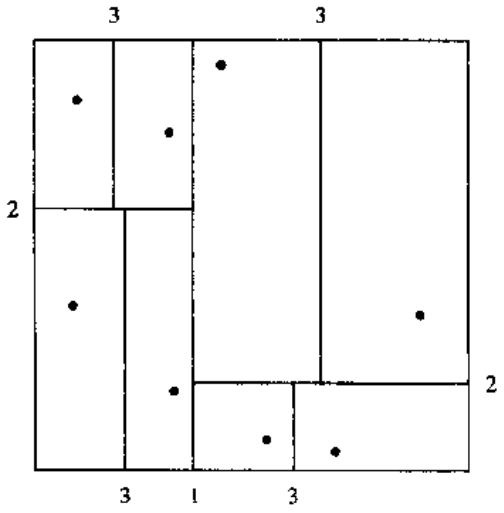


图 21-8　用 k-d 树划分空间

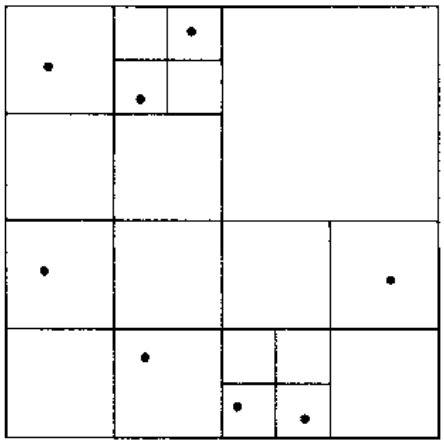


图 21-9　用四分树划分空间

上面描述的四分树类型称为 *PR* 四分树，表示它存储的是点，并且空间根据区域而不是实际存储的点集做划分。可以用区域四分树存储数组（光栅）信息。如果结点所覆盖的区域中的所有数组值都相同，那么区域四分树的这一结点是叶结点。否则，它进一步划分出四个相等的子区域，而它则是内部结点。区域四分树中的每个结点对应于一个子数组。对应于叶结点的子数组只有一个数组元素，或具有有多个相同值的数组元素。

线段和多边形的索引产生了一些新问题，可以对 k-d 树和四分树做扩展来加以解决。但是，一个线段或多边形可能跨越分界线，如果这样的话，就必须将它们分割开，然后在其各个部分出现的相应子树中加以表示。一个线段或多边形的多次出现会导致存储和查询效率不高。

3.R 树

一种称为 R 树的存储结构对矩形和其他多边形的索引是有很用的。R 树是平衡树的结构，非常像 B 树。R 树的每个结点不是存放取值范围，而是对应一个矩形边界框。多边形只存在叶结点上。叶结点的边界框是包含叶结点中所有对象的最小矩形（平行于坐标轴）。类似地内部结点的边界框是包含其子结点的边界框的最小矩形（平行于坐标轴）。

图 21-10 中的例子表示了一组矩形（用实线画出）及对应于这组矩形的 R 树中结点的边界

框（用虚线画出）。R树本身画在右边。注意，画边界框的时候我们在内部留了一点儿空，以使它们醒目。实际中，这些框应该更小一些，每条边至少都应该接触到下一层的某些对象或边界框。

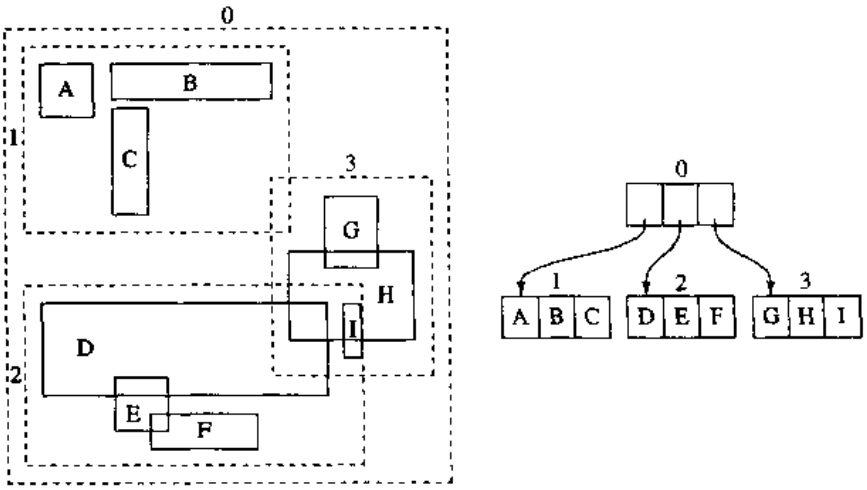


图21-10 一棵R树

兄弟结点所对应的边界框可以重叠，这与k-d树和四分树不同，后两者的区间是不重叠的。于是，搜索包含一点的多边形必须遍历所有其边界框包含该点的子结点，这样就必须搜索多条路径。同样，找出所有与给定多边形相交的多边形的查询必须向下遍历所有其对应矩形与该多边形相交的结点。

当我们向R树中插入一个多边形时，我们选择一个叶结点来放该多边形。如果该叶结点已经满了，就要进行结点分裂（必要时要向上传播），这类似于B^+树的插入。该插入算法保证树能保持平衡，并且叶结点和内部结点的边界框保持不变。分裂满结点有多种方法。有关R树以及R树的各种变形如R^*树或R^+树的更多详细信息参见文献注解。

R树的存储效率比k-d树或四分树要高，因为多边形只被存储一次，并且我们可以很容易地保证每个结点至少是半满的。但是，查询可能会慢一些，因为要搜索多条路径。用四分树作空间连接比用R树更简单，因为一个区域的所有四分树都以同一种方式划分。但是，由于R树及其变形的存储效率更高，并且类似于B树，使得它们在支持空间数据的数据库系统中更为流行。

21.6 多媒体数据库

最近，有关存储多媒体数据（如图像、声音和视频）的数据库的研究很热门。现在，多媒体数据通常存储在数据库以外的文件系统中。当多媒体对象的数目相对较少时，数据库所提供的特点往往不那么重要。

但是当存储的多媒体对象数目很多时，数据库的功能就变得重要起来。随之，事务更新、查询机制和索引也开始变得很重要。多媒体对象常常有描述属性，如指明它们是何时创建的、谁创建的，以及它们属于哪一类。构造这种多媒体对象的数据库的方法之一是用数据库存储描述属性，并且跟踪存储这些多媒体对象的文件。

但是，将多媒体数据存在数据库以外，使得难于提供数据库的功能，譬如基于实际多媒体数据内容的索引。此外，这种情况还会造成不一致，譬如一个文件在数据库中做了记录，但其

内容却丢失了；或其相反情况。因此我们更希望将数据本身存储在数据库中。

如果将多媒体数据存储在数据库中，就必须解决几个问题。

• 数据库必须支持大对象，因为像视频这样的多媒体数据可能会占据几个 GB 空间。许多关系系统不支持这种大对象，尽管对象存储系统如 Exodus 可以提供对大对象的支持。

• 许多多媒体数据库应用都需要基于相似性的检索。例如，在存储指纹图像的数据库中要查询一个指纹图像，这时数据库中与该指纹相似的指纹都必须被检索出来。像 B^+ 树和 R 树这样的索引结构不能用于这个目的。这里需要创建特殊索引结构。我们将在下节讨论基于相似性的检索。

• 某些数据类型的检索，如声频和视频，要求数据的传输必须在一个可以保证的平稳速率上。这种数据有时称为等时数据，或连续介质数据。例如，如果声频数据没有及时提供，声音中就会有间断。如果数据提供得太快，系统缓冲区就可能溢出，造成数据的丢失。我们将在 21.6.2 节讨论连续介质数据。

21.6.1　基于相似性的检索

在许多多媒体应用中，数据只是近似地在数据库中描述。前面我们讲了指纹数据的例子，其他例子还有：

• 图片数据。在数据库中的表示略微有些不同的两张图片或图像，用户可能认为是相同的。例如，一个存储商标的数据库。当一个新商标要注册时，系统首先需要识别以前注册的所有商标中是否有类似的商标。

• 声频数据。现已开发出基于语音的用户界面，允许用户通过说话发指令或指明数据项。必须检测用户输入与系统中存储的命令或数据项的相似性。

• 手写体数据。手写体输入用于指明数据库中存储的手写体数据项或命令。这里又一次需要相似性检测。

相似的概念常常是主观的和针对用户的。但是，相似性检测往往比语音或手写体识别更成功，因为输入可以与已经在系统中的数据比较，因而系统可选的集合很有限。

现在已经有一些算法利用相似性检测，找出与给定输入最匹配的数据。一些系统，包括用姓名拨号、由声音驱动的电话系统，已开始在商业上应用。这些算法的细节不在本书的范围之内，相关参考书目可以参见文献注解。

21.6.2　连续介质数据

最重要的连续介质数据类型是视频和声频数据（例如，一个电影数据库）。连续介质系统的特点是数据量大且要求实时的信息传输。

• 数据传输必须足够快，使得声频或视频输出没有间断。

• 数据传输的速度不会造成系统缓冲区的溢出。

• 必须保证不同数据流之间的同步。这个要求在某些情况下尤其重要，例如，一个人说话时嘴唇动作的视频数据必须与他说话的声频数据同步。

1. 多媒体数据格式

由于表示多媒体数据需要大量字节数，因此以压缩的格式存储和传输多媒体数据就很重要。对图像数据来说，最广泛使用的格式就是 JPEG，它是以制定它的标准组织 Joint Picture

Experts Group（JPEG）命名的。我们可以通过将视频的每一帧用JPEG格式编码来存储视频数据，但这样编码会有浪费，因为视频中连续的帧往往几乎是一样的。Motion Picture Experts Group（MPEG）提出了对视频和声音数据进行编码的标准，利用帧序列之间的共性达到更高程度的压缩。MPEG-1标准存储1分钟每秒30帧的视频和声频数据约要12.5MB（比较而言，只用JPEG压缩视频数据时约要75 MB）。但是，MPEG-1编码要损失一些视频质量，其效果近似于VHS录像带。MPEG-2标准是针对数字广播系统和数字视频盘（DVD）而设计的，它带来的视频质量损失是可忽略的。MPEG-2标准压缩1分钟的视频和声频数据约要17MB。

2. 视频服务器

许多厂商都开发出了视频点播服务器。因为现有的数据库系统不能满足应用的实时响应需求，因而当前的视频点播系统都基于文件系统。视频点播系统的基本体系结构包括如下：

• 视频服务器。多媒体数据存储在多个磁盘上（通常是以RAID的方式）。包含大量数据的系统可使用三级存储器存储不常访问的数据。

• 终端。人们通过各种设备观看多媒体数据，这些设备统称为终端。这些设备包括个人计算机或连接到一个机顶盒的电视机，机顶盒是一台小的廉价计算机。

• 网络。从服务器向多个终端传送多媒体数据需要高传输率的网络。异步传输模式（ATM）网络常用于该用途。

我们预计视频点播服务终将会无所不在，就像现在的有线电视和广播电视那么普及。目前，视频服务器技术的主要应用是在办公室（用于培训、观看录制好的谈话和演讲，及其他类似的应用）、酒店、以及视频生产厂家。

21.7 移动性和个人数据库

大型商用数据库传统上是存储在中央计算设备上的。在分布式数据库应用中，通常有强大的中央数据库和网络管理。然而以下这两个技术趋势的结合产生了一些应用，这些应用使中央控制和管理不再完全正确：

1）个人计算机越来越广泛的使用，其中更重要的是便携式或“笔记本”计算机的使用。

2）基于无线局域网、蜂窝数字包网络，及其他技术的成本相对低廉的无线数字通信基础设施的发展。

移动计算在许多应用中都证明是有用的。许多商务旅客使用便携式计算机以便于在途中可以工作和访问数据。邮递服务使用移动计算机辅助邮件的跟踪。紧急响应服务使用移动计算机在灾难、医疗急救及类似情况的现场访问信息，并且输入与当时情况有关的数据。移动计算机的新应用还在继续增多。

无线计算使得计算机不必有固定的位置和网络地址，这使得查询处理更加复杂，因为它难于决定实体化查询结果的最佳位置。某些情况下，用户的位置是一个查询参数。例如，一个旅客信息系统提供关于酒店、路边服务的信息及类似信息给乘车的旅客。有关当前道路前方服务的查询必须根据用户的位置、移动的方向及速度进行处理。

能源（电池电源）对移动计算机来说是有限的资源，这一限制影响了系统设计的许多方面。能源效率需求最有趣的结果之一是使用计划的数据广播来减少传输查询中移动系统的需求。

越来越多的数据会放在由用户管理、而不是由数据库管理员管理的计算机上，并且这些计

算机有时可能与网络断开连接。许多情况下，用户在断开连接时仍能继续工作的需求与保持全局数据一致性的需求是相互矛盾的。

以下四节我们讨论正在使用和开发的用于解决移动及个人计算问题的技术。

21.7.1 移动计算模型

移动计算环境包括移动计算机（称为*移动主机*），和有线计算机网络。移动主机通过称为*移动支持站点*的计算机与有线网络通信。每个移动支持站点管理其*蜂窝*（即它所覆盖的地理区域）内的移动主机。移动主机可以在蜂窝间移动，也就是说需要有从一个移动支持站点到另一个移动支持站点的控制交接。由于移动主机有时可能会关闭电源，因而一个主机可能会离开一个蜂窝而随后在某个很远的蜂窝内重新启动。因此，蜂窝间的移动不一定是在相邻蜂窝间进行的。在一个小区域内部，如一个建筑内，移动主机可能通过一个无线局域网相互连接，这与蜂窝广域网相比，前者连接成本要低，并且减少了交接的开销。

移动主机之间不通过移动支持站点而直接通信是可能的。但是，这种通信只能是在临近主机之间。

移动计算机的大小和电源限制导致了另一种内存层次结构。它要包括快闪存储器以取代磁盘存储器或附加在磁盘存储器之上，我们在第 10 章对快闪存储器做了讨论。如果移动主机包含了一个硬盘，该硬盘允许在不用的时候卸下，以节省能源。

21.7.2 路由和查询处理

移动计算模型的一个结果是一对主机之间的路由可能随着其中一台主机的移动而发生改变。这个简单的事实在网络层上有巨大的影响，因为在系统内基于位置的网络地址将不再是恒定不变的了。这些网络问题不在本书的讨论范围之内。

移动计算模型的另一个结果直接影响了数据库查询处理。正如我们在第 18 章所看到的，我们在选择分布式查询处理策略时必须考虑通信成本。移动性导致了通信成本的动态改变，于是使优化处理更为复杂。另外，还需要考虑如下一些同样重要的问题：

- 用户时间在许多商务应用中是十分宝贵的财富。
- 连接时间在某些蜂窝系统中是决定收费标准的指标。
- 传输的字节数或包数在数字蜂窝系统中是计算收费的单位。
- 基于每日不同时段的收费随着通信发生在高潮时段还是非高潮时段而不同。
- 能源是有限的。电池电源是珍贵的资源，必须优化其使用。无线电通信的一个基本原理是接收无线电信号需要的能量比发送无线电信号要少。因此，移动主机发送和接收数据的能源要求是不同的。

21.7.3 广播数据

我们常常希望被频繁请求的数据由移动支持站点不断周期性地广播，而不是在需要的时候才发送到移动主机中。广播数据的典型应用是股市价格信息。使用广播数据有两个原因。首先，移动主机避免了发送数据请求的能源开销。其次，广播数据可以立刻被大量的移动主机接收到，而没有额外的开销。因此，可获得的传输带宽得到了更有效的使用。

移动主机于是可以在那些数据被发送过来时接收数据，而不会因发送请求耗费能源。移动

主机可以有局部非易失性存储器，在数据接收到时能缓存广播数据，以便将来使用。当给定一个查询时，移动主机通过判断是否只使用缓存数据就可以处理该查询来优化能源开销。如果缓存数据还不够，可以有两个选择：等待数据被广播，或发送数据请求。要想做出决定，移动主机必须知道相关数据什么时候会广播。

广播数据可能按照固定的或变化的时间表发送。对于前者，移动主机使用已知的固定时间表决定什么时候相关数据会被发送。对于后者，广播时间表本身必须在一个众所周知的无线电频率上，以众所周知的时间间隔进行广播。

实际上，广播介质可以用看作延迟时间很长的磁盘来建模。对数据的请求可以被认为在所请求的数据被广播时得到了服务。发送时间表的作用类似于磁盘上的索引。将广播数据作为移动信息系统环境的一部分的使用仍在研究中。文献注解中列出了最新的研究论文。

21.7.4 连接断开与一致性

由于无线通信可以根据连接时间付费，所以某些移动主机就强烈需要能在某些时间内断开连接。在断开连接的时间段内，移动主机仍可以继续操作。移动主机的用户可以对位于本地的或缓存在本地的数据发查询和更新命令。这种情况引起了许多问题，其中最值得注意的有：

- 可恢复性。如果断开连接的移动主机上发生灾难性故障，在其上进行的信息更新可能会丢失。由于移动主机代表的是故障中的孤立点，所以不可能很好地模拟稳定存储器。
- 一致性。移动主机只有在重新连接后才能发现在本地缓存的数据可能已经过时。与此类似，在移动主机上进行的更新只有在其重新连接后才能传播给其他主机。在第 18 章讨论网络分割时，我们曾探讨了这个问题，这里我们进一步做详细的讨论。

在有线分布式系统中，分割被认为是一种故障模式。而在移动计算中，由于断开连接而造成的分割是常规操作模式的一部分。所以即使存在分割，甚至有损失部分一致性的危险，数据的访问仍应当被允许。

对于只被移动主机更新的数据，一个简单的方式是在移动主机重新连接时传播其所做的那些更新。但是，如果移动主机缓存了可能被其他计算机更新的数据的只读拷贝，则缓存的数据就可能变得不一致了。当移动主机连接时，可以发给它失效报告通知它有过期的缓存数据项。但是，当移动主机断开连接时，它就会错过失效报告。解决这个问题的简单方法是在重新连接时将整个缓存都作失效处理，但这种极端的解决方法开销太大。文献注解中列出了几种缓存方案。

如果更新可以发生在移动主机和其他地方，检测冲突的更新就会更为困难。现已提出了一些基于版本编号的方案，允许断开连接的主机对共享文件做更新。这些方案不能保证更新是一致的，但是，它们保证如果两个主机相互独立地更新了一个文档的同一个版本，那么当主机直接或通过一公共主机交换信息时这种不一致最终会被发现。基本想法是，每台主机 i 连带其每个文档 d 的拷贝都存储一个版本矢量，即版本号的集合 $\{V_{d,i,j}\}$，对每一个有可能更新该文档的其他主机 j 都有一项。当主机 i 更新文档 d 时，它将版本号 $V_{d,i,j}$ 加 1。

每当两台主机 i 和 j 相互连接时，它们交换更新的文档，使得两台主机都可获得文档的新版本。但是，在交换文档前，主机必须检查拷贝是否一致：

1）如果两台主机的版本矢量相同，即，对每个 k，$V_{d,i,k}=V_{d,j,k}$，则文档 d 的拷贝是相同的。

2）如果对每个 k，$V_{d,i,k}\leqslant V_{d,j,k}$，并且版本矢量不完全相同，则文档 d 在主机 i 上的拷贝比在主机 j 上的拷贝旧。也就是说，文档 d 在主机 j 上的拷贝位于主机 i 上的拷贝之后又做

了一次或多次修改。这时，主机 i 用主机 j 上的拷贝替换其 d 的拷贝和版本矢量。

3）如果存在一对主机 k 和 l，使得 $V_{d,i,k} < V_{d,j,k}$ 且 $V_{d,i,l} > V_{d,j,l}$，则拷贝是不一致的，也就是说，主机 i 的拷贝包含主机 k 所做的更新，而这些更新还没有传播到主机 j 上。同样，d 在主机 j 上的拷贝包含主机 l 所做的更新，而这些更新还没有传播到主机 i 上。于是，d 的拷贝就是不一致的，因为 d 上分别做了两次或多次更新，这时可能需要手工介入来更新合并。

上面的版本矢量方案最初用来解决分布式文件系统的故障。这个方案的重要性在于移动计算机常常要存储也在服务器系统上存放的文件的拷贝，实际上这就相当于一个经常会断开连接的分布式文件系统。这一方案的另一个应用是用于群件系统，其主机周期性地而非连续地相互连接，并且必须交换更新后的文档。版本矢量方案在有副本的数据库中也有应用。

但是，版本矢量方案并不能很好地解决更新共享数据中最困难、最重要的问题：不一致数据拷贝的一致化。许多应用可以通过在每台计算机上执行在连接断开期间在远程计算机上已执行的更新操作，自动做一致化。该方案在更新操作可交换时有效，也就是不论这些操作执行的顺序怎样，其结果是相同的。某些应用中可有其他技术，但是，在最差的情况下，必须由用户来解决不一致性。如何自动处理这种不一致性，以及如何辅助用户解决不能自动处理的不一致性仍然是一个研究课题。

版本矢量方案的另一个问题是它要求重新连接的移动主机和该主机的移动支持站点之间有大量的通信。一致性检查可以延迟到需要数据时再做，尽管这种延迟可能增加数据库整体的不一致性。

断开连接的可能性和无线通信的成本限制了第 18 章所讨论的分布式系统的事务处理技术的实用性。通常最好由用户在移动主机上准备事务，然后将事务提交给服务器执行，而不是在本地执行。涉及多台计算机并且包含移动主机的事务在事务提交过程中会而临长时间的堵塞，除非断开连接的情况极少发生或者是可以预料的。

21.8　信息检索系统

信息检索领域与数据库领域是同步发展的。信息检索领域中所用的传统模型如下：信息被组织成文档，并且一般认为会有大量的文档。信息检索的过程就是根据用户的输入，例如关键词或示例文档，来查找相关文档的过程。

信息检索系统的典型例子是联机图书目录和联机文档管理系统，譬如存储报纸上文章的文档管理系统。在这种系统中，数据被组织成一系列文档，例如报纸上面的文章或图书馆目录中的条目。系统使用者可以检索特定的一个或一类文档。所要查找的文档通常由一组关键词描述，例如，关键词“数据库系统”可能用来标定关于数据库系统的图书，而关键词“股票”和“谣传”可能用来标定有关股市中的谣传的文章。文档与一组关键词相关联，并且其关键词包含用户输入的关键词的文档将被检索出来。

这一模型与传统数据库系统所使用的模型有以下几点不同：

- 数据库系统中要处理的几个操作在信息检索系统中并不强调。例如，数据库系统要处理更新，以及相应事务的并发控制和持久性要求。这些事情在信息系统中就不很重要。同样，数据库系统处理的是在相对较复杂的数据模型（例如关系模型或面向对象的数据模型）基础上组织起来的结构化信息，而信息检索系统一直使用的是一种简单得多的模型，在这种模型中，数据库中的信息被简单地组织成一系列非结构化的文档。

• 信息检索系统处理的某些问题在数据库系统中也未得到充分重视。例如，信息检索领域中要处理管理非结构化文档的问题，比如用关键词进行模糊查询，还要处理基于查询文档的相关程度检索文档的问题。

21.8.1 查询

基于关键词的信息检索不仅普遍应用于存储文本数据的信息系统中，而且也应用于存储其他类型数据的系统中，例如多媒体数据（声频或视频片段，或静止图像）。可以为多媒体数据创建描述性关键词。与一部电影关联的关键词可以是它的片名、导演、演员、类型等等。

基于关键词的信息检索系统一般允许使用由关键词组成的表达式来查询。用户可以要求找出包含关键词“摩托车 *and* 维护”的所有文档，或者找出包含关键词“计算机 *or* 微处理器”的所有文档，甚至包含关键词“计算机 *but not* 数据库”的所有文档。

考虑用关键词“摩托车”和“维护”查找关于摩托车维护的图书的目录条目。假定每个图书条目的关键词是书名中的词和作者名，那么书名是“摩托车维修”的图书条目将不会被找出，因为“维护”这个词并没有在它的书名中出现。

我们可以采取使用同义词的方法解决这个问题。对每个词都定义一组同义词，每个词出现时都被它的所有同义词（包括它自己）的“*or*”操作所代替。于是，查询“摩托车 *and* 维修”被替换为“摩托车 *and*（维修 *or* 维护）”，这样就可以找出所要的条目了。

信息检索中的另一个重要问题是文档的相关度。假定除了书名和作者之外，我们还找出了书中内容涉及的一系列主题，并用它作为目录关键词的一部分。也许有许多关于赛车的书包含有关汽车和摩托车的章节，同时又包含有关维护的章节。关键词“摩托车”和“维护”也应该与这些书的条目相关联，在查询摩托车维护时，所有这些书都应当被检索出来，而不仅仅检索有关摩托车维护的书。但是，这些书只是与该查询有一点儿相关。关键词“摩托车”和“维护”出现得较近的书比这两个关键词出现得较远的书更有可能与查询相关。有些信息检索系统允许查询指明两个关键词在原文中一个先出现，另一个后出现，或者在同一个句子中出现。通常，指定关键词紧挨着出现的文档比分开出现的文档更有可能与查询相关。这样，包含所有指定关键词的文档可以按关键词出现的远近给出可能的相关度的评分，并按相关度顺序排序后呈现给用户。如果返回的结果过多，系统可以只呈现给用户最相关的那几个文档。

某些信息检索系统允许基于相似性的检索。这时，用户可以给系统一个文档 A，然后要求系统找出与 A“相似”的文档。两个文档的相似性可以定义，例如，根据共同的关键词。如果与 A 相似的文档非常多，系统可以只呈现给用户其中几个，并允许用户从中选择最相关的那些文档，然后根据选出文档和文档 A 的相似性开始一个新的检索。

21.8.2 文档的索引

一个高效的索引结构对于信息检索系统查询的高效处理是十分重要的。这种系统可以采用倒排索引定位包含给定关键词的文档。倒排索引将每个关键词 K_i 映射到包含 K_i 的文档集合（文档标识的集合）S_i 上。为了支持基于关键词相似性的相关度排序，索引可能不只提供文档标识，还提供关键词在文档中出现位置的列表。由于这些索引都必须存储在磁盘上，索引的组织还应尽量减少查找包含给定关键词的文档集合时 I/O 操作的次数。所以，系统可能需要将包含同一关键词的一组文档保存在连续的磁盘页上。

每个关键词都可能被许多文档包含，所以紧凑的表示是减少索引占用空间的关键。因此，一个关键词对应的一组文档应以压缩的形式加以保存。虽然这节省了空间，但有的时候索引的存储使得检索只是近似的，一些相关的文档可能没有被检索出来（称为误舍弃），或者一些无关的文档被检索了出来（称为误选中）。一个好的索引结构不应当存在任何误舍弃，但可以有一些误选中，因为系统随后可以通过查看文档中实际包含的关键词来滤除误选中的文档。

一个信息检索系统是否能够很好地回答查询可以用两个度量来表示。第一个是*准确率*，度量检索出的文档与查询真正相关的百分比。第二个是*查全率*，度量百分之多少与查询相关的文档被检索了出来。

and 操作用来找出包含给定一组关键词 K_1，K_2，…，K_n 中所有关键词的文档。实现 *and* 操作时，我们首先检索出分别包含关键词 K_1，K_2，…，K_n 的文档标识集 S_1，S_2，…，S_n。其交集 $S_1 \cap S_2 \cap \cdots \cap S_n$ 就是所求的文档标识集合。*or* 操作用来找出至少包含关键词 K_1, K_2，…，K_n 中的一个关键词的所有文档。我们通过计算并集 $S_1 \cup S_2 \cup \cdots \cup S_n$ 来实现 *or* 操作。*but not* 操作用来找出不包含给定关键词 K_i 的文档。给定文档标识集合 S，我们可以通过求差 $S\text{-}S_i$ 来去掉那些包含给定关键词 K_i 的文档，其中 S_i 是包含关键词 K_i 的文档标识的集合。

文档的*全文索引*用文档中的每个词作为关键词，这种索引可以用于较短的文档。但是，几乎所有的文本文档都包含“and”、“or”、“a”之类的词，所以这些词没有索引价值。信息检索系统定义了一组词称为*标点词*，它包括大约 100 个最常用的词，建索引时要从文档中去掉这些词，并且这些词也不允许作为查询中的关键词。

21.8.3　浏览与超文本

信息系统最初仅仅被认为是目录系统，比如图书馆的目录系统，实际的文档存储在这些系统之外。但是，随着磁盘容量的大幅度提高，且越来越多的文档被创建和联机存储，信息检索系统现在也用来存储实际的文档。因此，如今的信息系统允许用户浏览联机存储的文档。

通常，图书馆的用户使用目录来查找所要的图书，但当他从书架上拿到那本书时，他也可能浏览放在附近的其他图书。由于图书馆组织图书时会把相关图书摆放在一起，所以与用户想要的书摆放得近的图书也可能使用户感兴趣，故而用户浏览一下这些书是值得的。

为了将相关的图书摆放在一起，图书馆采用了分类层次。关于科学的图书被分在一类，在这类图书中还可以细分，比如计算机科学图书被分在一起，数学书被分在一起等等。由于数学和计算机科学存在一些联系，因而这两类图书也被摆放得比较近。在分类层次的另一层上，计算机科学图书又被分成许多子类，例如操作系统、语言或算法等。图 21-11 表示的是图书馆使用的一个分类层次。由于图书只能摆放在一个地方，所以图书馆里的每本书在分类层次中只被分到一个结点中。

在信息检索系统中，将相关文档存储在一起并不十分必要，这是因为磁盘访问速度已经足够快，磁盘臂的移动通常在这类系统中不是主要的性能问题。但是，这类系统需要逻辑地组织文档，以利于浏览。因此，这类系统可以采用和图书馆中使用的相似的分类层次，并且当某个文档显示出来时，它可以显示在分类层次中与该文档相近的图书的简要描述。

在信息检索系统中，没有必要一个文档只对应分类层次中的一个结点。一本面向计算机科学家的数学书可以同时分在数学类和计算科学类中。保存在每个结点中的都是文档标识（即指向该文档的指针），这样就可以很方便地通过标识获得文档的内容。

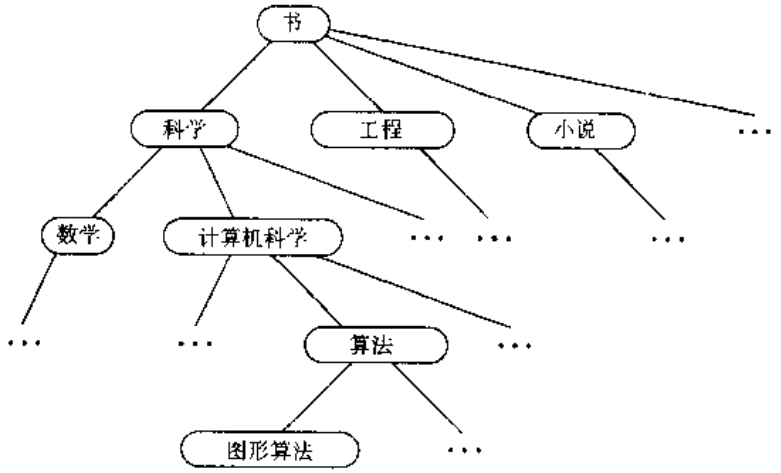


图 21-11　图书馆系统的分类层次

这种灵活性不仅使一本图书可以同时被分在两个类中，而且分类层次中的子类本身也可以同时出现在两个类中。例如，“图形算法”类的图书可以同时分在数学类和计算机科学类中。于是，现在的分类层次就是一个有向无环图（DAG），如图 21-12 所示。一本关于图形算法的书在有向无环图中可能只出现在一个位置上，但却可以从多条路径达到该位置。

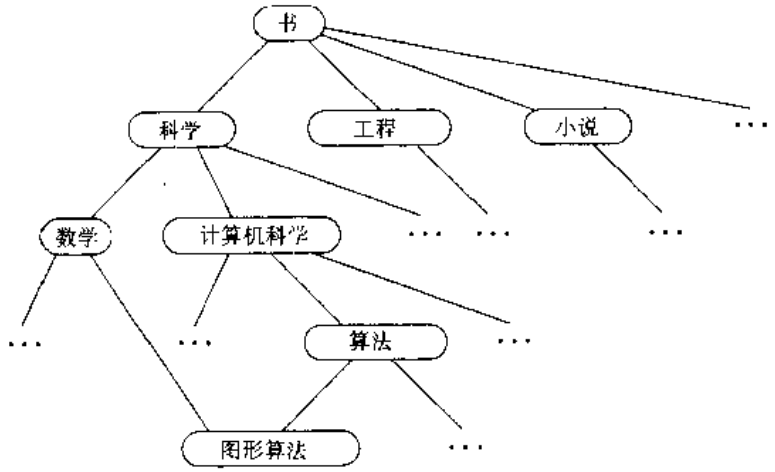


图 21-12　图书馆信息检索系统的分类 DAG

对于一个查询，信息检索系统不仅可以显示相关文档，甚至还可以显示在分类层次中相关的类，然后用户可以浏览该类中的所有图书（或其子类）。

课本或研究报告之类的典型文档有一个参考文献清单，列出提供相关信息的其他文档。它提供文档的名称、作者和出版商，系统可以使用这些关键词来查找对应的文档。但是，如果该文档已经在系统中，用户就可以提供与其对应的文档标识，避免再使用关键词进行查询。

超文本系统基于保存其他文档标识的思想。它提供功能强大的机制，使用户可以方便地从第一个文档切换到其他文档。通常，用户通过点击界面完成这项功能，在显示屏上简单地用鼠标点击一下引用的文档，就可以取出并显示该文档。超文本系统还允许对本文档或其他文档中特定位置的引用。在这样的链接上点击可以让用户转移到文档中想要的位置上。超文本系统提供了一种极为方便的浏览文档的方法。

一个早期的超文本系统是 Apple Macintosh 计算机系统引入的 HyperCard。超媒体系统不仅提供文本，而且还提供其他媒体，例如图像、视频和声频片段等等。当前流行的联机百科全书

提供这些支持和基于关键词的查询。传统的百科全书只提供文字和图像，这些系统与之不同，它们还包括视频和声频片段。我们也用超文本来指超媒体系统。

目前，分布式超文本系统甚至允许引用保存在分布式系统中其他位置的文档。在 21.10 节中我们将研究这类系统中最流行的系统—— World Wide Web。

21.9　分布式信息系统

近几年中计算机科学领域中一个最重要的发展是远距离计算机网络的兴起，尤其是全球因特网。现在，位于某一地区（如印度）的用户可以很容易地连接到遥远地区（如美国）的一台计算机上，并且可以获取存储在远距离外的计算机上的信息。远距离计算机网络的最初用途主要是发送电子邮件、在电子公告板上发布新闻、传输数据文件（ftp），以及连接远程计算机（远程登录或 telnet）。

人们很快认识到计算机网络可以用来获取大量有用的信息，例如，人名地址簿和研究论文之类的联机文档，另外还有许多其他信息可以联机存储。但是简便、标准的访问数据的方法和标准的图形用户界面还很缺乏。为此，人们现已开发出一些分布式信息系统，并取得了显著的成果。

现在使用最广泛的分布式信息系统是 World Wide Web（万维网）。由于这一系统十分重要，我们将在下节对其做详细讨论。

明尼苏达大学的 Internet Gopher 和广域信息系统（WAIS）是两个在 WWW 之前的早期分布式信息系统。它们代表了在分布式系统中组织和检索信息的两种不同方法。

Gopher 系统由服务器和客户机组成。Gopher 服务器将数据组织成目录，非常类似于分类层次。Gopher 客户机首先与服务器通信，并以菜单形式显示服务器层次中的顶级目录，如图 21-13 所示。用户可以从菜单中选择一项，它可能是层次中的另一个目录、或文档、或指向另一个服务器上的目录的链接。如果所选菜单项是层次中的一个目录，则目录中的数据就会以菜单的形式显示。如果所选菜单项是一个文档，则该文档就会被显示出来。如果所选菜单项是指向另一个 Gopher 服务器上的目录的链接，则客户机就会连接到这个新 Gopher 服务器上，并以菜单的形式显示目录。也就是说，Gopher 系统允许与远程服务器进行无缝连接。本地服务器和远程服务器从逻辑上讲没有区别。

```
                Internet Gopher Information Client 2.0 pl8

              University of XYZ Computer Science Department Gopher

-->  1.  WELCOME
     2.  Information About Gopher/
     3.  About the University of XYZ CS Department
     4.  University of XYZ Campus Information/
     5.  Search Gopher Titles at the Univ. of Minnesota <?>
     6.  University of Minnesota Gopher (Gopher Central)/
     7.  Libraries/
     8.  Phone Books/
     9.  Other Gopher and Information Servers/
     10. Internet file server (ftp) sites/

Press ? for Help, q to Quit, u to go up a menu      Page: 1/1
```

图 21-13　Gopher 客户机显示的例子

Gopher系统中信息检索的模型基于目录层次的浏览和导航。然而，Gopher目录中的一个项目可以是到其他类型服务器的链接，比如可以提供基于关键词索引能力的索引服务器。图21-13表示了存储在Gopher服务器上的典型主题，它还表示了与文档对应的菜单项、与索引服务器对应的菜单项（以“<？>”作为后缀），以及与目录对应的菜单项（以“/”作为后缀）。

广域信息系统（WAIS）采用了另一种信息检索的方法。WAIS中的基本模型是靠服务器系统进行信息的索引。每个服务器存储关于一个或多个主题的信息，并支持基于关键词的功能强大的针对本地数据的索引机制。除此之外，每个服务器维护其他一个服务器的目录或描述。该目录概括了存储在服务器上的信息类型以及如何访问这些信息。如果对WAIS服务器做查询，服务器会使用其目录信息告诉用户哪台服务器上可能有相关的信息。在因特网上的每台WAIS服务器上运行查询是不可行的，因为这种服务器的数目太多。目录提供了一个缩小搜索范围的方法，也使用户不必非要记住哪台服务器包含与查询相关的信息。

21.10 World Wide Web

World Wide Web（WWW，或Web）是一个基于超文本的分布式信息系统。在Web上存储的文档可以是几种类型之一。最常见的类型是超文本标记语言（HTML）格式的超文本文档，而HTML本身又是基于标准通用标记语言（SGML）的。HTML文档包含文本、字体描述和其他格式指令。指向（本地的或远程的）其他文档的链接可以与文本域结合在一起。图像也可以用适当的格式指令来引用（例如指向图像文件的链接）。

Web系统的用户看到的是格式化的文本和图像，而不是有格式指令的纯文本。含有图像的格式化文本从视觉上比纯文本更吸引人。而且，它所显示的文档是超文本文档，只要使用适当的浏览器，用户就可以点击与链接关联的区域，链接指向的文档就会显示出来。这样，Web上的超文本界面就成为一个功能强大的且视觉上吸引人的浏览界面。现在，Web浏览器已开始支持称为Java的编程语言，它允许文档包含可以在用户端执行的程序。所以，现在文档可以是主动的，而不仅仅是被动的。

下面五小节将研究支持Web的基本技术，并考察Web会如何影响数据库系统。

21.10.1 统一资源定位器

前面曾提到，一个超文本系统必须能够存储指向文档的指针。在Web上，指针的功能用统一资源定位器（URL）来实现。下面就是一个URL的例子：

http：//www.bell-labs.com/topic/book/db-book

URL的第一个部分表明文档如何被访问：“http”表明文档应当用超文本传输协议（HTTP）访问，这是一个传输HTML文档的协议。第二个部分给出在因特网上唯一的机器名。URL的其余部分是这个文件在这台机器上的路径名。

这样，URL给出了从Web系统可访问的每个文档全球唯一的名字。由于URL是用户可读的，所以用户可以直接使用它来访问所需的文档，而不必从一个预先定义的位置一步步地浏览过去。由于URL允许使用因特网机器名，所以它们是全球范围内的，并且用户可以使用URL创建跨机器的链接。

与Gopher系统不同，Gopher系统显示的要么是文档要么是目录，而HTML文档可以同时显示文档和目录。另一个重要的区别在于，由于URL是用户可读的，并且可以指明系统中的

文件，所以任何用户在与因特网连接的机器上都可以容易地创建文档，而其他用户可以通过Web对它进行访问。Gopher系统需要由系统管理员建立并控制，而Web则开放得多：任何用户都可以在Web上创建文档，并且可以将URL告诉别人。实际上，这也导致Web几乎处于无政府状态，Web上没有任何中央权威机构进行管理。现在许多因特网用户在Web上都有自己的主页，这些主页常常包含用户的个人信息和工作生活信息。

URL可以指明如何访问文档，这一特性也很重要。除了HTTP，Web客户机也支持传输文档的其他协议，例如应用广泛的文件传输协议（ftp）。Web客户机甚至提供针对ftp服务的基于超文本的前端。

21.10.2　Web服务器

除了简单的文档传输之外，HTTP还提供功能强大的特性，例如出于安全性考虑的数据加密。URL中的文档名可以指定一个可执行程序，该程序运行后生成一个HTML文档。当一个HTTP服务器接收到对该文档的请求后，它执行相应的程序，并将生成的HTML文档返回。并且，Web客户机可以在传输文档名的同时传递其他参数，这样它们可以在程序执行时作为参数传递。所以，生成的文档可以根据传入参数的不同而不同。

这些特性使得Web服务器可以很容易地作为多种信息服务的前端服务器。要在Web上提供新的服务，用户只需创建并安装一个提供该服务的可执行程序。Web上支持的HTML语言用来实现信息服务图形用户界面。

21.10.3　显示语言

如今的计算机系统不仅提供简单的字符界面，还支持图形、多种字体和颜色等等，这些选项极大地提高了信息显示的质量。但是，图形显示有许多标准，并且不同类型的计算机有不同的接口，所以很难编写一个程序可以直接支持多种显示接口。如PostScript之类的页面描述语言和如nroff或$\TeX$/$\LaTeX$之类的标准格式化文本语言提供了标准化接口，但这些语言有大量的特殊功能，使得它们的速度很慢，而不适于交互使用。

标准通用标记语言（SGML）之类的文本标记语言已被提出，以填补纯文本和页面描述或格式化文本语言之间的空白。标记语言提供简单的格式命令，例如段分隔符、编号或项目列表，以及一定程度的显示字体控制。因此，这些语言为程序提供标准的输出接口。

文本标记语言对通用分布式信息系统尤其重要，因为要显示的信息应当以一种好的方式格式化，而不仅仅是纯文本。实现这种格式化的另一种方法是在客户端为每一个信息服务运行一个专用程序，这个方案对想访问多种信息服务的用户显然是不方便的。

1. 超文本标记语言

SGML提供了根据标准标记注释定义文档格式的语法。HTML是基于SGML的通用超文本显示语言，它允许使用格式化文本命令、超文本链接命令和图像显示命令来定义文档格式。图21-14给出了一个HTML文档的源代码的例子，图21-15给出了与该文档相对应的显示图像。

HTML还提供了少量的输入特性。例如，一个HTML文档可以指定显示一个表格。HTML显示程序（即Web客户机）允许用户填写表项。HTML还提供了菜单和其他图形输入机制，用户可以选择一个菜单项、在一个范围内移动滚动条来选择一个数值、点击图像（比如

```
<title>WWW Home Page</title>
<img src="lucent_logo.gif"> <img src="iitb-logo.gif">
<hr>
<h3> <a href=
   "http://www.bell-labs.com/topic/books/db-book">
   Database System Concepts:  Home Page </a>
</h3>
<h3> Information Services </h3>
<dl>
   <dd> <a href="http://www.altavista.digital.com">
           The Altavista Information
           Gathering Service</a>
   <dd> <a href=
           "http://www.informatik.uni-trier.de/~ley/db">
           Database and Logic Programming
           Bibliography</a>
   <dd> <a href="http://cs.indiana.edu/cstr/search">
           Indiana University's Unified
           CS Tech. Report Service</a>
   <dd> <a href="http://www.ncstrl.org">
           Networked Computer Science
           Tech. Report Library</a>
</dl>
```

图 21-14　HTML 源代码的一部分

图 21-15　与图 21-14 的 HTML 文档相对应的图像显示

一张地图）中的某个位置，等等。用户完成数值输入时，可以按“提交”按钮。

这时，用户输入的值被收集起来，Web 客户机发送对文档的请求，用户输入的值作为参数也发送给服务器。该文档是一个可执行程序，Web 服务器在调用该程序时使用传递来的参数。与前面描述的一样，这个程序生成一个 HTML 文档，返回给客户机显示。

屏幕的实际布局、要填写的具体表格以及待选的菜单项都由 HTML 文档控制。所以，HTML 可以提供几乎能在所有计算机上运行的图形用户接口。但是，客户机和服务器之间没有持续的连接，服务器不保存与客户机之间交互的历史记录，所有的状态信息必须存储在客户端，并在每次通信时传给服务器。尽管有上面这些限制（可能在将来会被去除），但是 HTML 对分布式信息系统的重要性将会越来越显著。

2. Java

一个更新的发展是使文档变得主动的 Java 语言的发布，它通过在本地机器上运行程序执行像动画之类的动作，而不仅仅是显示被动的文本和图像。由 Sun 公司开发的 Java 语言是一种与 C++ 类似的解释语言。Java 语言还提供标准的图形用户接口函数库。Java 程序被编译后，编译器不是生成机器码，而是生成可在解释器上执行的字节码。现在几乎每个具体平台（硬件和操作系统的结合）上的 Java 解释器都已编写好了，因此同一个 Java 程序可以运行在几乎所有平台上。

Java 环境的统一性使得 Java 程序可以像 HTML 文档一样存储在服务器端，并且可以由任何客户机下载和运行，而不管客户端使用什么平台。所以，Java 提供了一个发布（小）程序的简便途径。这些程序的主要用途是与用户做灵活交互，这比 HTML 和 HTML 表格提供的有限的交互能力要强得多。另外，与每次交互都要传给服务器端处理的方式相比，在客户机端执行程序可以大大加速交互的速度。

只需按一下按钮就可以下载和执行程序的方便也带来了下载的程序会导致破坏的危险，例如删除客户端的文件。与系统程序不同，下载的程序没有权威机构，如广商，来检查它们，以确保没有恶意破坏行为。因此，只允许 Java 程序在那些不会造成永久性破坏的机制如屏幕上显示数据是十分重要的，Java 程序不允许直接进行系统调用。所以，Java 解释器内置了一个安全系统，保证 Java 代码不直接进行任何系统调用。有潜在危险的操作，如写文件，将被通知给用户，且用户可以选择是中止该程序还是继续执行。

21.10.4　数据库的 Web 接口

数据库与 World Wide Web 接口的重要性在于以下两点。第一，随着 Web 上电子商务的发展，用于事务处理的数据库必须连接到 Web 上。HTML 表格界面对事务处理来说十分简便。用户可以在订单表格中填写详细信息，然后按提交按钮传给服务器一条包含他所填写信息的消息。与该订单表格相应的程序会在服务器端执行，然后在服务器端的数据库上执行一个事务。事务执行的结果可以表示成 HTML 的格式并显示给用户。

数据库与 Web 接口的第二个原因在于用来向用户显示的固定 HTML 代码有一些局限：

- 使用固定 Web 文档，显示不能依用户的不同而不同。例如，一个公司可能希望根据用户的兴趣显示广告信息。
- 当公司数据更新时，Web 文档如果不同步则会过时。当多个 Web 文档复制同一重要数据，并且所有文档都必须更新时，这一问题就变得更加尖锐。

我们可以通过动态生成 Web 文档解决这些问题。当请求一个文档时，可以有一个程序在服务器端运行，它查询数据库，并根据查询结果生成文档。这样就可以根据保存在数据库中的用户信息，针对不同的用户显示 Web 文档。Web 文档中的数据可以由数据库查询来定义，这样，不论什么时候数据库中的相关数据更新了，Web 文档就会随之被更新。

将 HTML 表格中定义的变量转化成 SQL 会涉及格式的转换。将数据库查询生成的表和其他数据转化成 HTML 就更麻烦了，因为程序员必须知道许多如域的格式和大小以及生成的元组个数之类的细节。连到数据库的 Web 接口可以简化这些工作。这些接口与报告书写器（参见 19.6 节）很相似，实际上，两者的主要区别在于 Web 接口生成 HTML 文档而不是格式化文本。程序员可以用嵌入 SQL 查询的宏语言来定义一个 HTML 文档。HTML 表格中定义的变量可以直接在嵌入的 SQL 查询中使用。当用户请求该文档时，宏处理器执行 SQL 查询并生成传给用户的实际的 HTML 文档。用宏语言编写的 HTML 文档可以有两个部分：一个表格输入接口和一个结果显示接口。交互发生在两个阶段。在第一个阶段中，服务器生成 HTML 文档的表格部分并传回给客户机。用户在提交表格后，服务器生成文档的结果部分并传回给客户机。

21.10.5 在 Web 上查找信息

Web 提供一种简便的途径去访问因特网上的信息源，并与之交互。但是，Web 长期面临的问题是信息爆炸，且几乎没有什么指导帮助用户查找感兴趣的信息。

用户找到感兴趣的信息的一个简单途径是去问他的朋友！实际上，即使现在这仍然是大多数有趣 Web 站点上信息传播的方式。第二个办法稍微复杂一些，用处也可能小一些，它是用户在他的主页上发布他看过的最有趣的 Web 站点的列表，其他人可以按照这些链接查找有趣的站点。这两种办法对查找信息来说都不是特别有效。

实际地去一个地方旅游和访问 Web 上的一个站点、查询一个文档或者使用一个信息服务有一定的相似性，所以让我们来看一看上面的问题在旅游中是怎么解决的。时间和金钱是旅游的限制因素，而访问 Web 站点的限制通常只有时间。你如何决定去哪个地方旅行呢？一个信息来源是曾经旅游过的朋友。导游手册也有你要去的地方的介绍。图书馆将各种图书进行分类以帮助你方便地浏览相关的旅游图书。最后，图书目录可以帮助你找到具体想要的图书。

Web 上有与上面这些资源大致相似的信息资源。有些 Web 站点提供关于其他 Web 站点的信息，其方式类似于导游手册。有些这类站点的功能甚至像一个联机购物商店，它在一个站点上综合了各种服务，使用户可以通过 Web 购买各种各样的商品。有些站点维护了关于网上站点信息的分类层次，用户可以按照这个分类层次找到感兴趣的 Web 站点。

类似于图书目录的信息资源在 Web 上是最难实现的。在图书馆里，每本图书都是由图书馆管理员购买的，并且由他相应地在目录中添加一个条目。而在 Web 上没有这样的管理员，那么谁来找出 Web 上有什么站点，以及它们包含什么内容呢？其他信息系统，比如 Gopher，也存在类似的问题。

分布式信息收集系统执行找出 Web 上存在什么信息的任务。尽管 WAIS 系统在用户请求信息时才将查询请求传给远程系统，但分布式信息收集系统却是定期地从远程系统获取信息，并在本地创建一个索引。这样，系统可以把多个信息源的信息综合在一个站点上。客户端程序可以访问分布式信息收集系统，并提出查询请求，而系统会查询本地存储的索引并给出结果。由于磁盘空间的限制，文档通常并不存储在信息收集系统中，返回的查询结果只是指向位于远

程机器上的文档的指针。

这类系统的一个早期的例子是“计算机科学技术报告索引”，它可以收集联机存储在各种ftp站点上的计算机科学技术报告信息。Archie系统在这类系统中更具竞争力，它可以自动沿着Gopher链接查找信息，并创建从各站点所获信息的集中式索引。

各种信息收集系统统称为网络爬虫，已被开发出来用于Web信息的查询和收集。他们沿着已知文档中存在的超文本链接递归地找到其他文档。文档取出后，从文档中获得的信息被加到组合索引中去，而文档并不被存储。一个后台运行的守护进程执行这些服务，它沿着链接查找新站点，从站点中获得最新信息，并丢弃已不存在的站点，这样索引中的信息可以得到合理的更新。

随着当前Web的迅猛发展，网络爬虫的工作变得愈发繁重，这是因为有更多的站点需要被索引。另外，由于存在大量站点，一个查询可能产生大量与之对应的结果，这就需要将相关站点和不相关站点区分开来的良好技术。然而，网络爬虫具有不需要中央机构进行文档注册的巨大优势，这使得用户可以很容易地创建新文档并让别的用户根据索引找到它。Web上的数据索引是一个正在发展的研究领域。

与数据库系统提供的查询功能相比，基于关键词查找文档是一个简单的查询处理形式。支持更复杂的查询所面临的问题在于Web上的数据一般都是非结构化文本格式，而不像关系型数据库中存储的高度结构化的数据。*半结构化数据*是指数据基本上是非结构化的，只存在一定程度的结构。例如，用属性author:、title:、subject:、和keywords:中的几个属性值或全部属性值标注的文档可以被看作是半结构化的数据。对半结构化的数据的查询处理是另一个正在研究的课题。

21.11　总结

随着企业认识到联机事务处理系统收集的联机数据的价值，决策支持系统也越发变得重要了。现已提出的**SQL**扩展，如**cube**操作，能帮助系统生成汇总数据。数据挖掘致力于从大数据库中自动发现统计规律和模式等知识。数据可视化系统帮助人们从视觉上发现这些知识。数据仓库是从多数据源收集的信息的仓储（或转储），它以统一的模式在单一地点存储。这些数据一旦被收集起来，就会存储很长时间，并允许访问历史数据以便决策支持的需要。

目前，空间数据库正在越来越多地被应用于存储计算机辅助设计数据和地理数据。设计数据基本上以矢量数据的形式存储，而地理数据则包含矢量数据和光栅数据。空间完整性约束对设计数据来讲十分重要。矢量数据可以被编码成第一范式数据，也可以用非第一范式结构存储，如链表。专用索引结构对访问空间数据和处理空间查询尤为重要。R树是B树的二维扩展，它和它的变形如R^+树和R^*树在空间数据库中得到了广泛的应用。将空间以某种规则方式划分的索引结构（如四分树）可以帮助处理空间连接查询。

多媒体数据库正变得越来越重要。基于相似性的查询以及按可以确保的速率传送数据等问题是当前研究的重要课题。

移动计算系统的普及使人们对在这类系统上运行的数据库系统产生了浓厚的兴趣。在这类系统上的查询处理可能会涉及在服务器端数据库上的查找。查询成本模型必须包括通信成本，其中包括金钱上的成本和电池电源的成本，它们对于移动系统来说是相对较高的。对每个接收者而言，广播方式比点对点通信便宜得多，像股市数据之类的数据广播使得移动系统可以低成

本地接收数据。连接断开操作、广播数据的使用和数据缓存是当前移动系统正致力解决的三个重要问题。

信息检索系统使用比数据库系统更为简单的数据模型，但在这个受限制的模型上可以提供更强大的查询能力。查询试图通过指定关键词这样方法查找感兴趣的文档。用户头脑中的查询往往不能很精确地表达出来，所以，信息检索系统会按可能的相关度将查询结果排序。

在因特网上运行的分布式信息系统最近几年得到了飞速发展。World Wide Web 系统支持用超文本范型浏览这些信息。把 HTML 和 SQL 集成在一起的工具减轻了为数据库创建 HTML 前端应用的工作。分布式信息收集系统帮助人们在 Web 上查找信息。

习题

21.1 考虑关系 *account* 中的 *balance* 属性。编写一个 SQL 查询来计算 *balance* 值的直方图，将从 0 到当前最大帐户余额之间的区域平均划分为三个区间。

21.2 考虑 21.2 节中的关系 *sales*，编写一个 SQL 查询来计算该关系上的 *cube* 操作，关系如图 21-2 所示，不要使用 with cube 结构。

21.3 假定有两个分类规则，一个认为工资在 10 000～20 000 之间的人信用级别为 *good*，另一个认为工资在 20 000～30 000 之间的人信用级别为 *good*。在什么条件下，这两条规则可以被一条规则（认为工资在 10 000～30 000 之间的人信用级别为 *good*）代替，且不损失任何信息？

21.4 假设一个服装店中 1/2 的交易是购买牛仔裤，1/3 的交易是购买 T 恤衫。并假设购买牛仔裤的交易中有 1/2 同时还购买了 T 恤衫。请你写出从上面信息推出的所有（非平凡的）关联规则，给出每条规则的支持度和置信度。

21.5 考虑找出频繁项目集的问题，即经常一起购买的项目的集合。一个项目集的支持度是指购买了项目集中每一种项目的交易数量。一个项目集是频繁项目集，如果其支持度大于某个预先定义的 k。

(a)描述如何用一次数据扫描得到给定一组项目集的支持度。假定这些项目集及其相关信息，如计数，可以放在内存中。

(b)假设一个项目集的支持度小于 j，证明这个项目集的任何超集的支持度都不会大于或等于 j。

(c)由于内存的限制，在一次数据扫描中只能得到少数几个项目集（内存中可放得下的数目）的支持度。设计一种算法，通过计算最少数量的项目集的支持度来减少计算频繁项目集所需的数据扫描次数（从而也就减少了所需的 I/O 次数）。

21.6 阐述在数据仓库中收集数据时，源驱动体系结构和目标驱动体系结构相比的优缺点。

21.7 假设有一个关系包含 x、y 坐标和餐馆名，并假设查询只能是如下形式：查询指定一个点，问是否恰好在这个点上有一家餐馆。R 树或 B 树中哪一种索引更好？为什么？

21.8 考虑二维矢量数据，且每个数据项都不重叠。是否可以把这些矢量数据转换为光栅数据？如果可以，存储转换得到的光栅数据而非存储原始矢量数据有什么缺点？

21.9 假设有一个空间数据库支持区域查询（圆形区域），但不支持最近邻查询。描述一个算法，通过使用多条区域查询找出最近邻。

21.10 假设要在 R 树中存储线段。如果线段与坐标轴不平行，它的边界框会很大，并包含大

量的空白区域。

- 请说明在查询与一个给定区域相交的线段时，大边界框对性能的影响。
- 简要描述一种技巧来提高这类查询的性能，并给出体现其优点的例子。

(提示：可以把线段分为更小的部分。)

21.11　在连续介质系统中，数据传送过慢或过快会导致什么问题?

21.12　阐述 RAID 组织（10.3 节）中的思想方法如何应用于数据广播环境，其中偶尔会有噪声破坏部分传输数据的接收。

21.13　列出与传统分布式系统不同的无线网络上的移动计算的三个主要特性。

21.14　列出在传统查询优化器中不予考虑，而在移动计算的查询优化中要考虑的三个因素。

21.15　定义一个广播数据模型，其广播媒介是一个虚盘。描述这个虚盘的寻址时间、等待时间和数据传输率与一个普通硬盘的对应参数的差异。

21.16　考虑一个将所有文档保存在中央数据库的文档数据库。移动计算机上有部分文档的拷贝。假设移动计算机 A 在它断开连接时更新了文档 1 的拷贝，同时，移动计算机 B 在它断开连接时更新了文档 2 的拷贝。请说明：在移动计算机重新连接时，版本向量机制如何保证对中央数据库和移动计算机的正确更新。

21.17　给出版本向量机制无法保证可串行性的例子（提示：使用习题 21.16 中的例子，假设文档 1 和文档 2 在移动计算机 A 和 B 上都可用，并考虑文档读出时没有被更新的可能性）。

21.18　误选中和误舍弃的区别是什么？如果信息检索查询时不漏掉任何相关信息这一点至关重要，应当允许误选中还是误舍弃？为什么?

21.19　假设要查找“至少包含给定的 n 个关键词中的 k 个关键词”的文档，并且假设有一个关键词索引，它给出包含某一关键词的所有文档标识的（排序）列表。给出用来查找所求文档集合的有效算法。

21.20　阐述 HTML 得以取得巨大成功的三个主要特性。

21.21　阐述 Java 提供的而 HTML 没有的主要特性，以及该特性的几个优点。

21.22　阐述根据数据库中存储的数据动态生成网页的几个优点。

21.23　阐述用户从 Web 上查找信息的三种不同方法。

文献注解

Gray 等［1995］描述了数据立方体操作符。Harinarayan 等［1996］阐述了数据立方体操作符的高效拖延实现技术，它在需要时才生成所需的聚集。

Fayyad 等［1995］有大量关于知识发现和数据挖掘的文章。Agrawal 等［1993］是在数据库中作数据挖掘的早期综述。Agrawal 等［1992］阐述了用于分类的数据挖掘技术。Ng 和 Han［1994］阐述了基于空间聚类的数据挖掘技术。Srikant 和 Agrawal［1996］介绍了推导定量关联规则的技术。

Poe［1995］和 Mattison［1996］是讲述数据仓库的教科书。Zhuge 等［1995］描述了在数据仓库环境中的视图维护。Larson 和 Yang［1985］以及 Dar 等［1996］描述了如何使用实体化的关系回答查询。

空间索引结构方面的著作有很多，Samet［1995b］是在这一领域的较新的综述。Samet

[1990] 是讲述空间数据结构的教科书。Finkel 和 Bentley [1974] 提供了关于四分树的一个早期描述。Samet [1990, 1995b] 描述了四分树的各种变形。Bentley [1975] 描述了 k-d 树，Robinson [1981] 描述了 k-d-B 树。Guttman [1984] 最先提出了 R 树。R 树的各种扩展由 Sellis 等 [1987] 做了阐述，其中描述了 R^+ 树。Beckmann 等 [1990] 描述了 R^* 树。Kamel 和 Faloutsos [1992] 描述了 R 树的并行版本。

Brinkhoff 等 [1993] 讨论了使用 R 树实现空间连接。Lo 和 Ravishankar [1996] 以及 Patel 和 DeWitt [1996] 介绍了计算空间连接的基于分区的方法。Samet 和 Aref [1995] 综述了空间数据模型、空间操作以及空间与非空间数据的集成。Aref 等 [1995a, 1995b] 以及 Lopresti 和 Tomkins [1993] 讨论了手写体文档的索引。Barbará 等 [1992] 中讨论了近似数据的连接。Evangelidis 等 [1995] 介绍了并发访问空间数据索引的技术。

Samet [1995a] 阐述了多媒体数据库中的研究问题。Faloutsos 和 Lin [1995] 讨论了多媒体数据的索引。Anderson 等 [1992]、Rangan 等 [1992]、Özden 等 [1994]、Freedman 和 DeWitt [1995] 以及 Özden 等 [1996a] 讨论了视频服务器。Berson 等 [1995] 和 Özden 等 [1996b] 讨论了容错性。Reason 等 [1996] 提出了另一种无线网络上视频传输的压缩方案。Chen 等 [1995]、Chervenak 等 [1995]、Özden 等 [1995a]、以及 Özden 等 [1995b] 阐述了视频数据的磁盘存储管理技术。

Alonso 和 Korth [1993] 以及 Imielinski 和 Badrinath [1994] 研究了包括移动计算机的系统中的信息管理。Imielinski 和 Korth [1996] 介绍了移动计算及与其相关的一系列研究论文。Imielinski 等 [1995] 讨论了无线媒介上的广播数据的索引。Barbará 和 Imielinski [1994] 以及 Acharya 等 [1995] 讨论了移动环境中的数据缓存。Douglis 等 [1994] 讲述了移动计算机的磁盘管理。

Popek 等 [1981] 以及 Parker 等 [1983] 阐述了用来检测分布式文件系统中的不一致性的版本向量方案。

Salton 的书 [1989] 是一本关于信息检索系统的很好的教科书。其他关于文本数据库的著作还包括 Gravano 等 [1994] 以及 Tomasic 等 [1994]。关于因特网的书有很多，包括本章中描述的关于分布式信息系统的资料。Krol [1994] 描述了一些分布式信息系统，并提供了因特网上信息资源的一个目录。大多数流行的分布式信息系统都可以从因特网上免费得到。学习这些系统的最好方法是使用它们。Nguyen 和 Srinivasan [1996] 描述了基于在 HTML 文档中嵌入 SQL 查询的 HTML 和 SQL 之间的接口。在 Web 上有一些分布式信息收集和索引系统，其中一个比较流行的是 Digital Altavista 系统，其 URL 是 http://www.altavista.digital.com。URL 为 http://www.yahoo.com 的 Yahoo 系统提供了一个庞大的 Web 站点的分类层次。

附录A 网状模型

在关系模型中，数据及数据之间的联系是用一组表来表示的。网状模型与关系模型不同，网状模型中的数据是用记录的集合来表示的，而数据之间的联系是用链接来表示的。

A.1 基本概念

网状数据库由一组通过链接相互关联的记录组成。记录与实体－联系（E-R）模型中的实体在多方面有相似之处。一个记录由多个字段（属性）构成，每个字段只包含一项数据值。一个链接恰好是两个记录之间的关联。因此从 E-R 模型的意义上来说，链接可以看作是关系的一个受限（二元）形式。

为了说明以上概念，我们考虑银行系统中一个表示联系 *customer-account* 的数据库。其中有两个记录型：*customer* 与 *account*。如前所述，可以用类 *Pascal* 表示法定义 *customer* 如下：

```
type customer = record
                    customer-name: string;
                    customer-street: string;
                    customer-city: string;
                end
```

account 记录型可以定义如下：

```
type account = record
                    account-number: string;
                    balance: integer;
                end
```

图 A-1 中的示例数据库表明 Hayes 有帐户 A-102、Johnson 有帐户 A-101 和 A-201、Turner 有帐户 A-305。

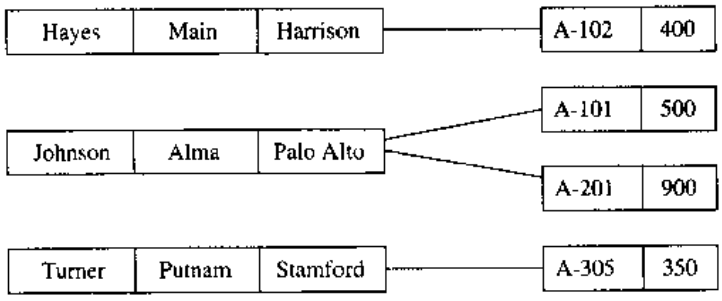
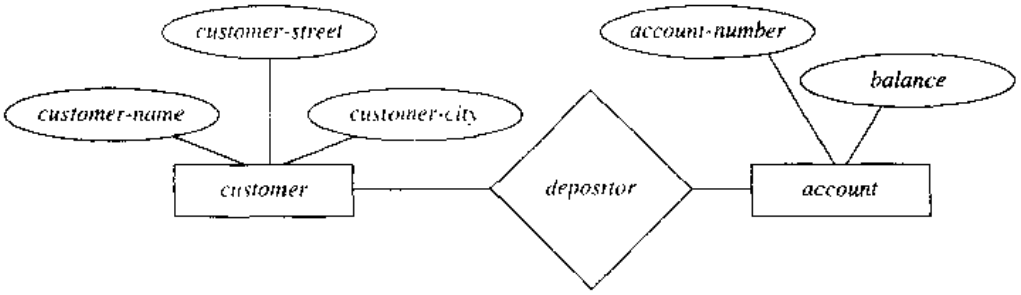


图 A-1 示例数据库

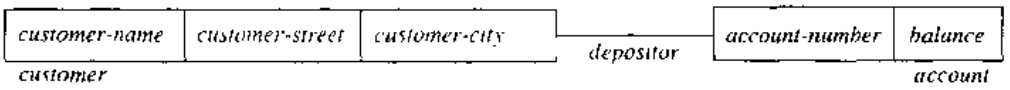
A.2 数据结构图

数据结构图是表示网状数据库设计的一种模式。这种图由两个基本成分组成：代表记录类型的方框和代表链接的线条。数据结构图和 E-R 图的目的一样，即说明数据库的总体逻辑结构。E-R 图可以转换成相应的数据结构图。

举例来说，请考虑如图 A-2a 所示的 E-R 图。它由两个实体集 *customer* 和 *account* 组成，这两个实体集通过多对多且无描述属性的二元联系 *depositor* 联系起来。这个图说明，一个客户可以有多个帐户，一个帐户可以属于多个客户。相应的数据结构图如图 A-2b 所示。记录型 *customer* 对应于实体集 *customer*，正如前面定义的那样，它包含三个字段：*customer-name*、*customer-street* 和 *customer-city*。同样，*account* 是对应于实体集 *account* 的记录型，它包含两个字段：*account-number* 和 *balance*。最后，联系 *depositor* 被替换成链接 *depositor*。如果联系 *depositor* 从 *customer* 到 *account* 是一对多的，则链接 *depositor* 将有一个指向 *customer* 记录型的箭头。同样，如果联系 *depositor* 是一对一的，则链接 *depositor* 将有两个箭头：一个指向 *account* 记录型，一个指向 *customer* 记录型。

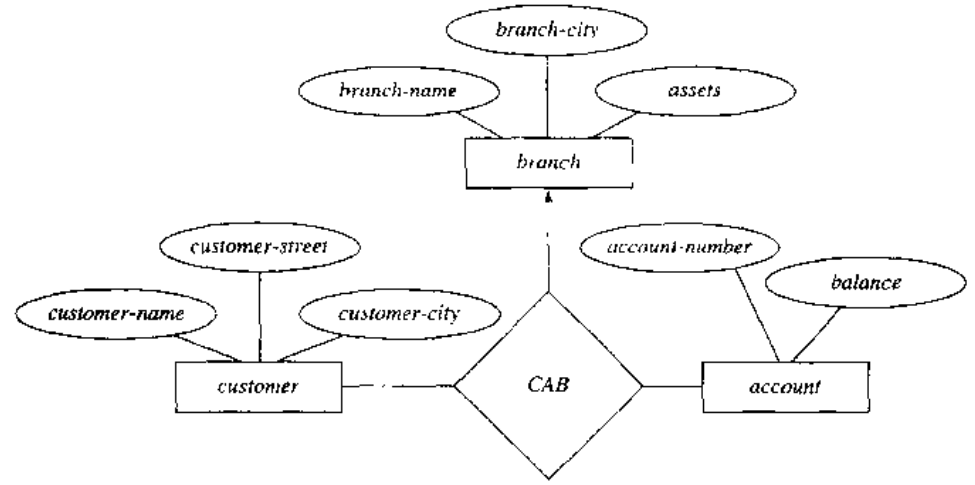


a) E-R 图

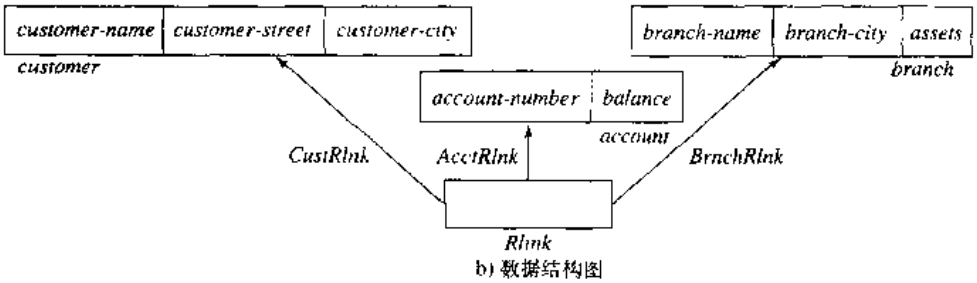


b) 数据结构图

图 A-2 E-R 图及其相应的数据结构图



a) E-R 图



b) 数据结构图

图 A-3 E-R 图及其相应的数据结构图

考虑如图 A-3 所示的 E-R 图，它由三个实体集 *account*、*customer* 与 *branch* 组成，这三个实体集通过不带描述属性的概括联系 *CAB* 相联。该图说明一个客户可以有多个帐户，每个帐户位于特定的银行分支机构，且一个帐户可属于多个不同的客户。

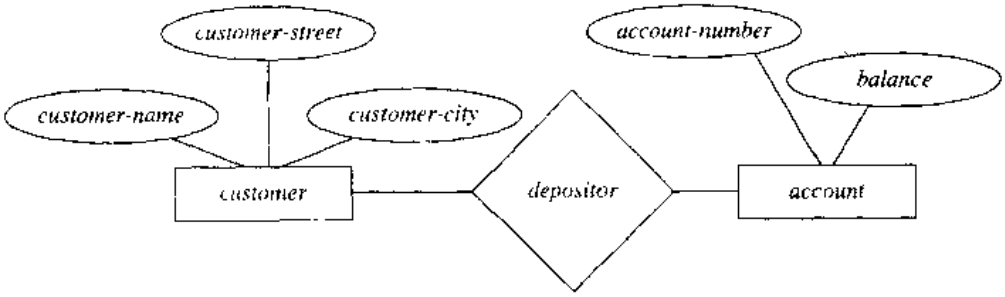
由于一个链接正好可以连接两个不同的记录型，因此需要一个新的记录型来将这三个记录型连接起来，这个新记录型直接与三个记录型相链接。

要把图 A-3a 中的 E-R 图转换成网状数据结构图，必须创建一个新的记录型 *Rlink*，*Rlink* 可以没有字段或只有一个包含唯一标识的字段。该标识由系统提供，应用程序不直接使用该标识。这一新记录型有时称为虚记录型（或链接记录型、连接记录型）。我们必须创建三个多对一链接：*CustRlnk*、*AcctRlnk*、*BrncRlnk*，如图 A-3b 所示。如果联系 *CAB* 有描述属性，它们将成为记录型 *Rlink* 的字段。

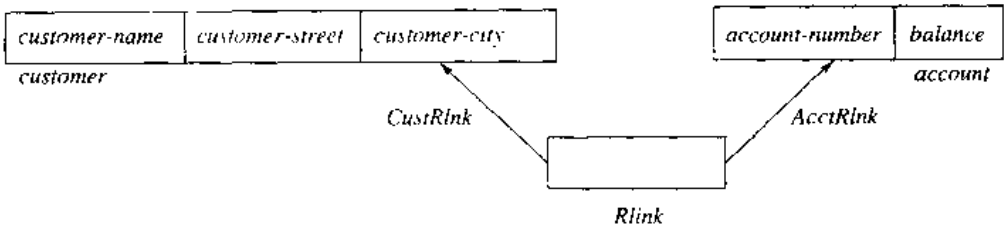
A.3　DBTG CODASYL 模型

第一个数据库标准规范是由数据库任务组在 60 年代后期写成的，称为 CODASYL DBTG 1971 报告。在 DBTG 模型中只允许使用多对一链接。一对一链接表示成多对一链接。为简化实现，多对多的链接是不允许的。

如果联系 *depositor* 是多对多的，则转换算法必须按如下方式修改。考虑如图 A-4a 所示的联系。为转换该联系，必须创建一个新的虚记录型 *Rlink*，它没有字段或只有一个包含外部定义的唯一标识的字段，并且还必须创建两个多对一的链接：*CustRlnk* 与 *AcctRlnk*，如图 A-4b 所示。



a) E-R 图



b) 数据结构图

图 A-4　E-R 图及其相应的数据结构图

由于 DBTG 模型中只可用多对一链接，因此链接在一起的两个记录型组成的数据结构图通常具有图 A-5 所示的形式。该结构在 DBTG 模型中被称为 DBTG 系。系的名字通常就使用连接两记录型的链接的名字。在每个这样的 DBTG 系中，记录型 *A* 称为该系的首记录型（或该系的父记录型），记录型 *B* 称为该系的属记录型（或该系的子记录型）。每个 DBTG 系可以有任

意多的系值——链接记录的实际实例。例如图 A-6 中，对应于图 A-5 所示的 DBTG 系我们有三个系值。

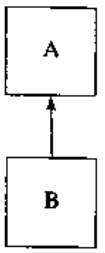


图 A-5 DBTG-系

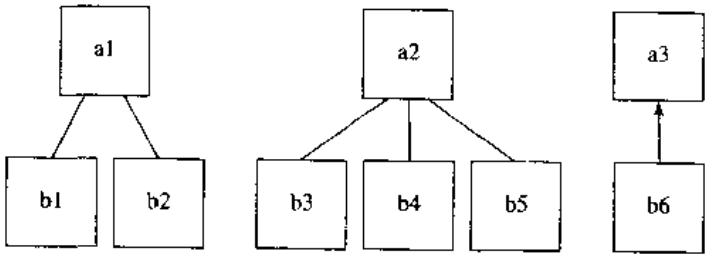


图 A-6 三个系值

由于多对多的链接是不允许的，因此每个系值正好有一个首记录，有零个或多个属记录。此外任何时刻，一个系的属记录不能参与该系的多于一个的系值，但一个属记录可以同时参与不同 DBTG 系的多个系值。

DBTG 支持的数据操纵语言由嵌入宿主语言的命令组成。通过使用命令，程序员可以根据给定字段的值从数据库中选取记录，并通过重复取下一记录的命令遍历选中的记录。程序员还可以通过使用所提供的命令找出某个记录所在系的首记录，以及遍历该系的属记录。当然，更新数据库的命令是必然会有的。

A.4 实现技术

在 DBTG 模型中链接是通过在由链接相互关联的记录中增加指针字段来实现的。为说明该实现，假设联系 *depositor* 是从 *account* 到 *customer* 的多对一联系。一个 *account* 记录只能与一个 *customer* 记录关联。因此在 *account* 记录中只需要一个指针来表示联系 *depositor*。一个 *custmer* 记录可以与多个 *account* 记录关联，但我们并不在 *customer* 记录中使用多个指针，而是用环结构表示整个 DBTG 系的 *depositor* 的值。在环结构中，系值的首记录型与属记录型的记录被组织成循环链表，每个系值（即首记录型的每个记录）有一个循环链表。

图 A-7 所示的是图 A-1 所示例子的环结构。我们来检查属于“Johnson”记录的 DBTG 系值，它有两个属记录型（*account*）的记录。首记录中并非对每个属记录都有一个指针，例如客户为 Johnson 的记录只包含一个指向帐户为 A-101 的属记录指针，该属记录包含指向下一帐户 A-201 的属记录指针。由于帐户 A-201 记录是最后的属记录因而，因而它包含一个指向首记录的指针。

使用指针实现多对多的链接是十分困难的，因此，DBTG 模型限制链接必须是多对一的。DBTG 模型的实现策略也为 DBTG 数据检索功能提供了基础。

A.5 讨论

从上面的讨论可以清楚地看到，网状模型同其实现紧密相关。如前所述，数据库设计者为了实现简单的多对多联系就不得不创建人工记录型。关系模型中的查询可以用简单的、声明性的方式进行，网状模型则不同于关系模型，其查询要复杂得多。程序员被迫按链接的方式思考，考虑如何遍历链接以获取所需的信息；网状模型中的数据操纵因此被称为导航式的。

与关系模型相比，网状模型在数据库设计以及数据操纵方面大大加重了程序员的负担。由于关系模型一开始实现效率较低，网状模型阻碍关系模型发展长达数年之久。现在，由于关系数据库模型的许多优秀的功能，网状模型已失去其重要性，因此本书也不再对此详细讲述。

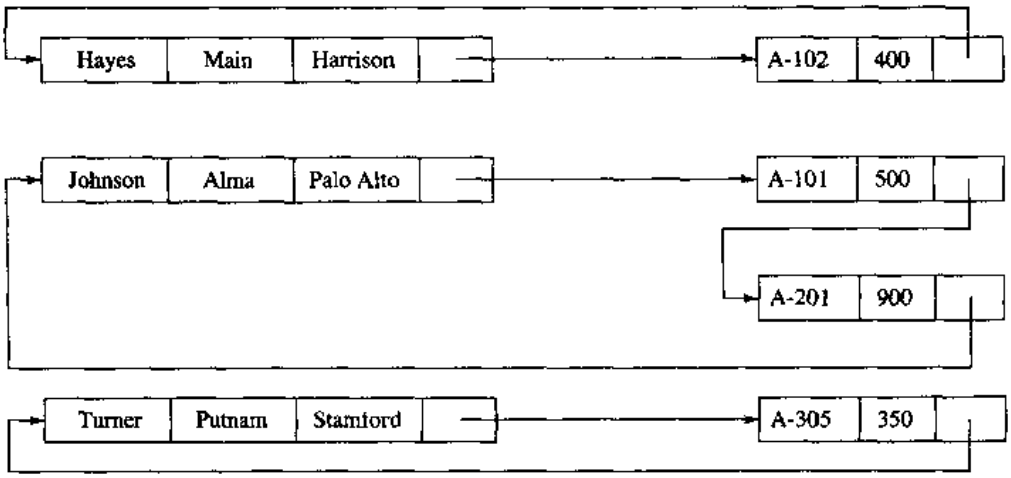


图 A-7 针对图 A-1 所示例子的环结构

然而为了方便读者，网状模型的详细描述可在下述 Web 站点得到，其 URL 为：

http：//www.bell-labs.com/topic/books/db-book

或者通过匿名 FTP 从 ftp.research.bell-labs.com 的子目录 dist/db-book 下得到。

附录B 层次模型

在网状模型中，数据用记录的集合表示，数据间的联系用链接表示。层次模型中也采用这样的结构。它们唯一的区别在于：层次模型中记录组织成树的集合，而不是任意图。

B.1 基本概念

层次模型是由一组通过链接互相联系在一起的记录组成的。这里的记录和网状模型中的记录十分相似。每个记录都是字段（属性）的集合，其中每个字段只包含一个数据值。链接是两个记录之间的关联，因此，这里的链接类似于网状模型中的链接。

来看一个表示银行系统 *customer-account* 联系的数据库。其中有两种记录型：*customer* 和 *account*。*customer* 记录型可用与附录 A 中相同的方法定义，它包含二个字段：*customer-name*、*customer-street*、*customer-city*。同样，*account* 记录由两个字段组成：*account-number* 和 *balance*。

图 B-1 所示为一个示例数据库。它表明客户 Hayes 有帐户 A-102、客户 Johnson 有帐户 A-101 和 A-201、客户 Turner 有帐户 A-305。

不难发现，所有客户和帐户记录被组织成带根树的形式，并且该树的根是一个空结点。正如我们将看到的：层次数据库是这种带根树的集合，因而数据库是一个树林。我们把每个这样的带根树称为*数据库树*。

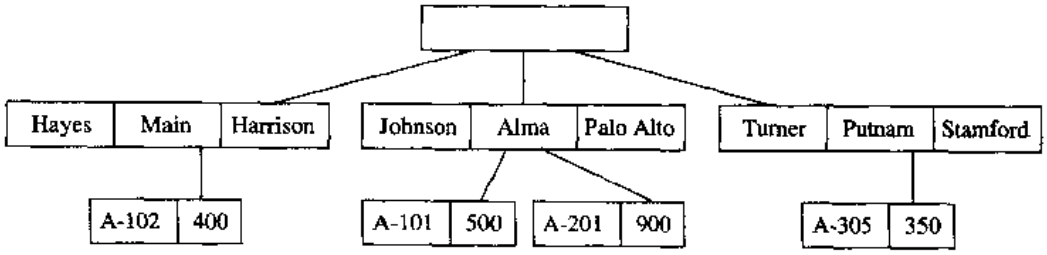


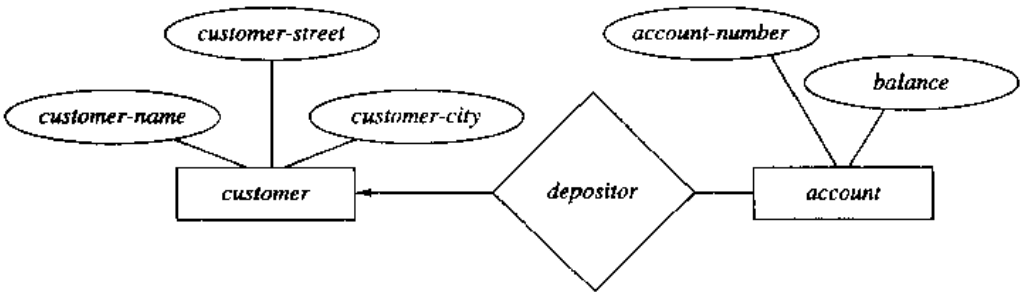
图 B-1 示例数据库

B.2 树结构图

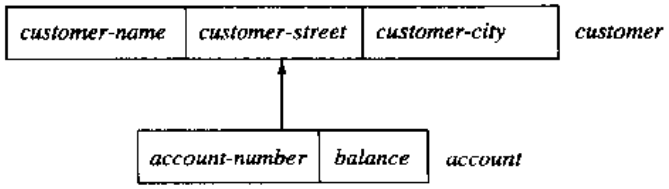
树结构图是层次数据库的模式。这样的图由两个基本成分组成：代表记录型的方框和代表链接的线条。树结构图和 E-R 图的目的一样，也就是说，它说明数据库的总体逻辑结构，树结构图类似于网状模型中的数据结构图。它们的主要区别在于：前者的记录型组织成带根树的形式，而后者的记录型组织成任意图的形式。树结构图不能有环。同时，我们仍可以把 E-R 图转换成相应的树结构图。

举例来说，请考虑图 B-2a 中的 E-R 图，它由实体集 *customer* 和 *account* 以及将它们联系起来的联系 *depositor* 组成，这一联系是无描述属性的二元一对多联系。这个图说明，一个客户可以有多个帐户，但一个帐户只能属于一个客户。图 B-2b 描述的是相应的树结构图。记录型 *customer* 对应于实体集 *customer*，它包含三个字段：*customer-name*、*customer-street* 和 *customer-city*。同样，*account* 是对应于实体集 *account* 的记录型，它包含两个字段：*account-number* 和

balance。最后，联系 *depositor* 被替换成箭头指向记录型 *customer* 的链接 *depositor*。



a) E-R 图



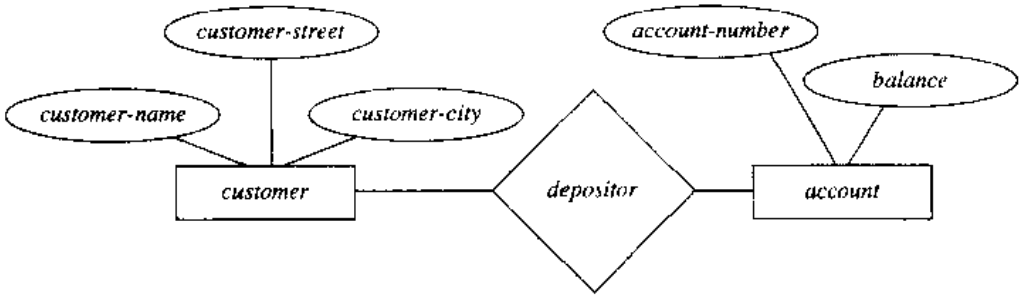
b) 树结构图

图 B-2　E-R 图及其相应的树结构图

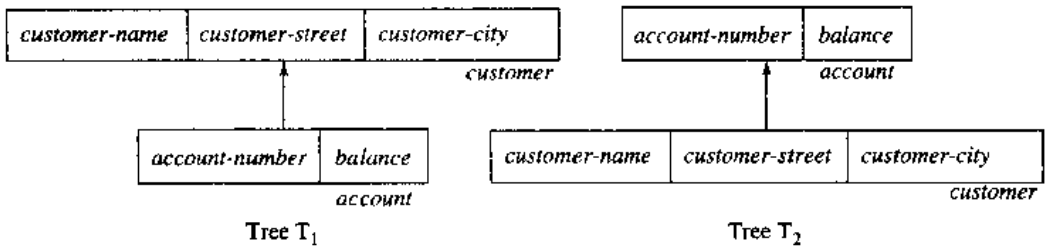
因此，如图 B-1 所示，一个对应于所描述模式的数据库实例可能包含许多 *customer* 记录，它们与许多 *account* 记录链接在一起。由于 *customer* 到 *account* 是一对多的联系，所以一个客户可以有多个帐户，例如 Johnson 有帐户 A-101 和 A-201。但是，一个帐户不能属于多个客户，在示例数据库中就没有哪个帐户属于多个客户。

如果联系 *depositor* 是多对多的（参阅图 B-3a），那么从 E-R 图到树结构图的转换就复杂得多。在层次模型中，只有一对多和一对一联系可以直接表示。

要把图 B-3a 中的 E-R 图转换成树结构图，我们可按如下步骤操作：



a) E-R 图



b) 树结构图

图 B-3　E-R 图及其相应的树结构图

1）创建两个独立的树结构图 T_1 和 T_2，每个树结构图都有 *customer* 和 *account* 记录型。在树 T_1 中，*customer* 是根；在树 T_2 中，*account* 是根。

2）创建下面两种链接：

- *depositor* —— T_1 中从 *account* 记录型到 *customer* 记录型的多对一链接。
- *account- customer* —— T_2 中从 *customer* 记录型到 *account* 记录型的多对一链接。

由此得到的树结构图如图 B-3b 所示。

数据库模式用一组树结构图表示。对每一个图，都只存在数据库树的一个实例，且该树的根是空结点。结点的子女是对应记录型的实例，每一个子实例同样又可以有在相应树结构图中定义的各种记录类型的多个实例。

对应图 B-3b 中树结构图的示例数据库如图 B-4 所示，其中有两棵数据库树。第一棵树（图 B-4a）对应树结构图 T_1；第二棵树（图 B-4b）对应树结构图 T_2。我们可以看到，所有 *customer* 和 *account* 的记录都重复出现在两棵数据库树里。此外，*account* 记录 A-201 在第一棵树里出现两次，而 *customer* 记录 Johnson 和 Smith 在第二棵树中现两次。

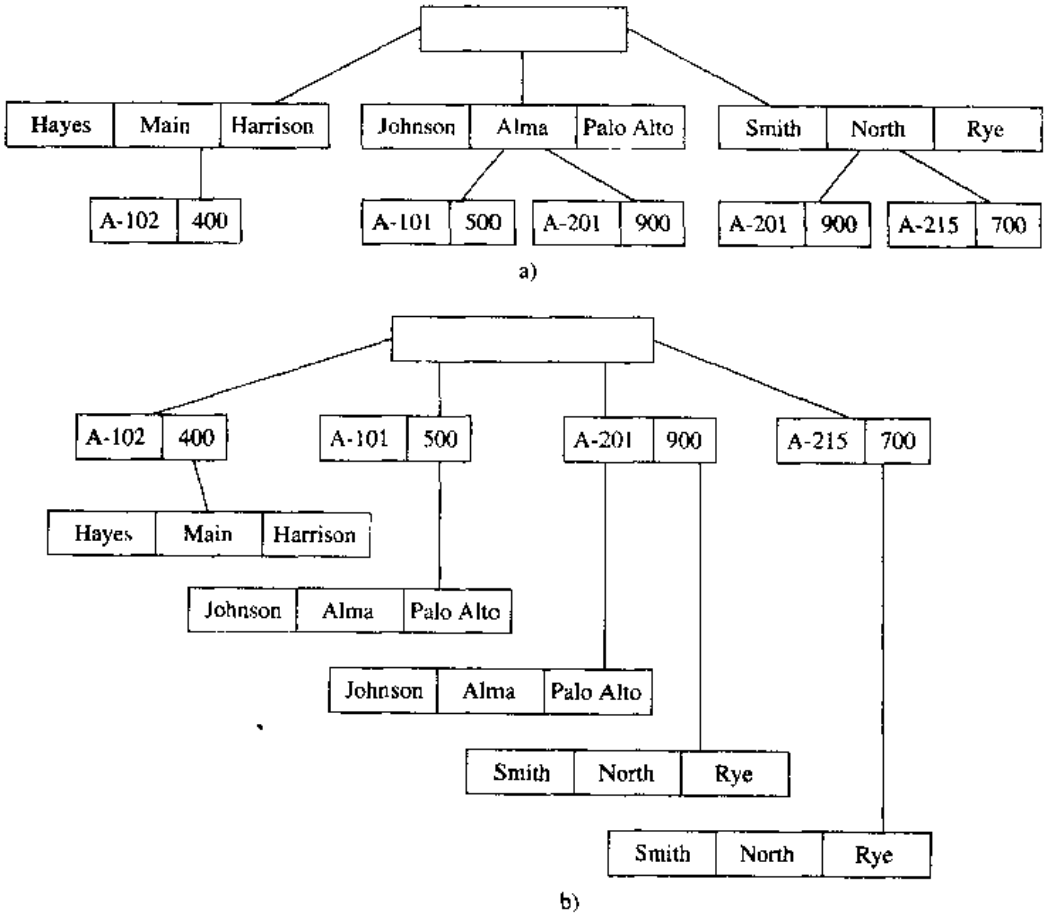


图 B-4 对应图 B-3b 的示例数据库

考虑如图 B-5a 所示的 E-R 图。通过将前面描述过的转换算法应用到 *account-branch* 和 *depositor* 联系，得到如图 B-5b 所示的图。这个图不是一个带根树，因为这棵树唯一可能的根是记录型 *account*，但这个记录型和它的子女有多对一的关系，违反了我们对带根树的定义。为了把这个图转换成带根树的形式，我们复制 *account* 记录型以创建出两棵分离的树，如图 B-6 所示。注意每棵树确实是一棵带根树。因此，一般而言，我们总可以把一个图分解成多个图，并

且使其中每个图都是带根树。

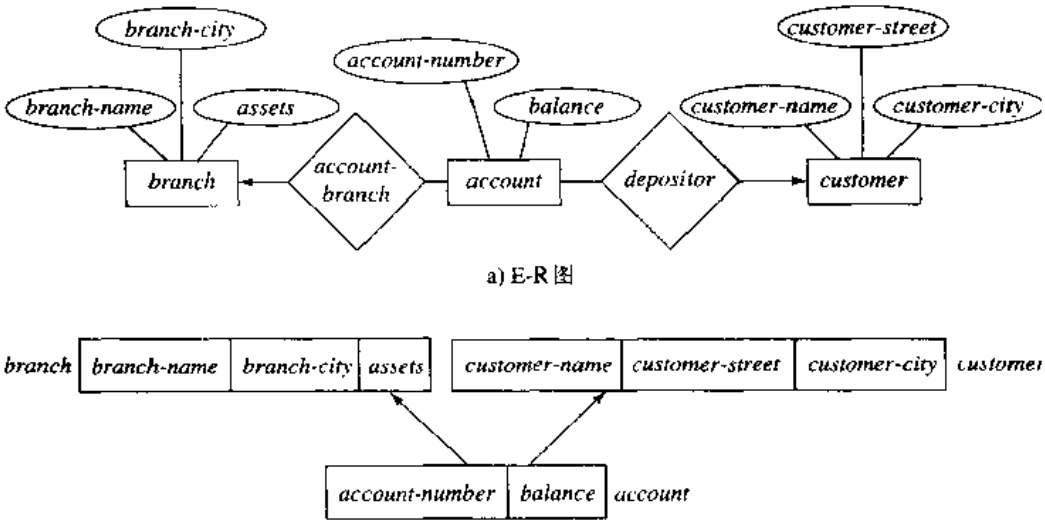


图 B-5　E-R 图及其变换

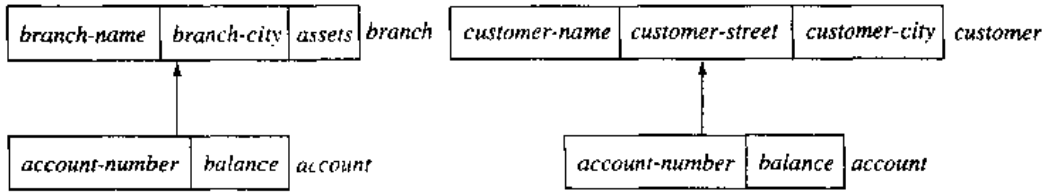


图 B-6　对应于图 B-5a 的树结构图

B.3　实现技术

记录复制有两个主要的缺点：

1）更新的时候可能出现数据不一致。

2）不可避免的空间浪费。

解决这些问题的方法是引入虚记录的概念。虚记录不含数据，但它包含指向特定物理记录的一个逻辑指针。当一个记录被复制到几个不同的数据库树里时，我们只保留这个记录在某棵树中的单一拷贝，而用一个含有指向物理记录的指针的虚记录来代替其他记录。

作为例子，考虑图 B-3a 的 E-R 图和相应树结构图，该树结构图由两棵分离的树组成，每一棵树都由 *customer* 和 *account* 记录型组成（图 B-3b）。

为了消除数据重复，我们创建了两个虚记录型：*virtual-customer* 和 *virtual-account*。然后我们在第一棵树中用记录型 *virtual-customer* 替换 *customer*，在第二棵树中用记录型 *virtual-account* 替换 *account*。我们还在记录 *virtual-customer* 和 *customer* 之间以及 *virtual-account* 和 *account* 之间分别加上一条虚线来表明虚记录与其相应物理记录之间的关联。所得树结构图如图 B-7 所示。

树结构图的实例可用指向最左子女和临近兄弟的指针来实现。一个记录只有两个指针：最左子女指针指向它的一个子女，临近兄弟指针指向其同一双亲的下一个子女。图 B-8 给出了图 B-1 所示数据库树的这样一种结构。在这种结构中，每个记录正好有两个指针。这样，当我们

加入所需指针的时候，定长记录保持了固定的长度。

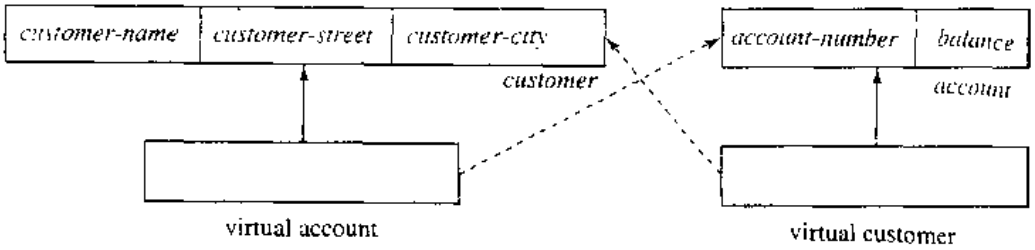


图 B-7 带虚记录的树结构图

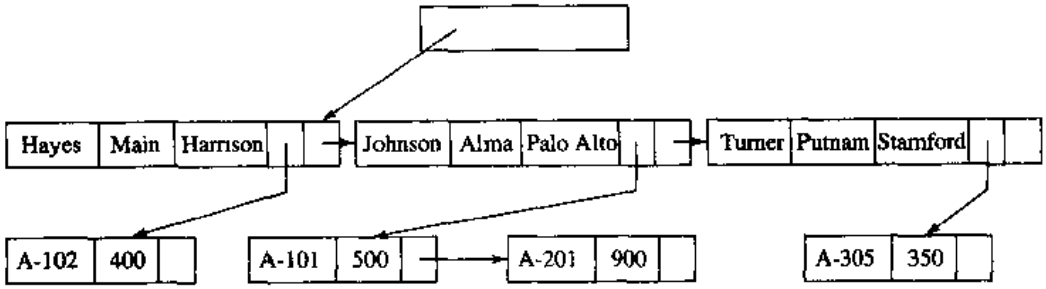


图 B-8 使用最左子女指针与临近兄弟指针的一个实现

B.4 IMS 数据库系统

层次模型之所以重要，其主要原因在于 IBM 公司的 IMS 系统的重要。IBM 的信息管理系统（IMS）是最早和使用最广的几个数据库之一。由于 IMS 系统历史上曾是最大的数据库之一，因而它的开发者是最早开始处理并发、恢复、完整性和高效查询处理这些问题的人。

高性能事务处理的需要导致了 *IMS Fast Path* 的引入。Fast Path 使用一种物理数据组织方法，让数据库中最活跃的部分驻留在主存中。此类方法是主存数据库系统方面所展开的工作的一个先驱。

层次数据库的数据操纵语言由嵌入宿主语言的命令组成。风格上它类似于网状数据库的数据操纵语言，但所提供的功能有所不同。IMS 数据库系统的数据操纵语言称为 DL/I。

B.5 讨论

层次模型具有和网状模型类似的不足之处。树结构图的复杂性和多对一链接的限制使数据库的设计工作非常复杂。此外，缺乏声明的查询功能、存取信息需利用指针进行导航，这更使查询变得复杂。

和网状模型一样，数年之久，由于层次模型（如 IMS）的实现技术比关系模型优越，层次模型阻碍了关系模型的发展。现在，由于关系数据库模型的许多优秀功能，层次模型已失去其重要性，因此本书也不再对此详细讲述。

然而为了方便读者，层次模型的详细描述可在下述 Web 站点得到，其 URL 为：

http://www.bell-labs.com/topic/books/db-book

或者通过匿名 FTP 从 ftp.research.bell-labs.com 的子目录 dist/db-book 下得到。