# 第3章　设计数据仓库

建造数据仓库有两个主要方面——与操作型系统接口的设计和数据仓库本身的设计。在某种程度上来说，“设计”并不能精确描述在启发式方式下建造数据仓库时发生了什么。首先，载入一部分数据，供DSS分析员使用和查看。然后，根据最终用户的反馈，在数据仓库中修改、增添一些数据。

这种反馈循环贯穿于整个数据仓库的开发过程。那种以为在建造数据仓库时，用过去曾用的设计方法就可以满足需求的想法是错误的。在数据仓库部分载入并且为 DSS分析员使用之前，数据仓库的需求是不可能知道的。因此，在设计数据仓库时，不能采用设计传统“需求-驱动”系统同样的方法。在另一方面，那种认为不预测需求是好思路的想法也是错误的。在实际中，通常是介于两者之间的。

## 3.1　从操作型数据开始

起初，现存系统中存储的是操作型数据。这就难免会让人认为建造数据仓库是一个抽取操作型数据，然后将其输入数据仓库的过程。其他就没有什么要做的了。

图3-1简单描绘了从现有的系统环境中抽取出数据加入数据仓库的过程。可以看到有多个应用程序对数据仓库作出贡献。

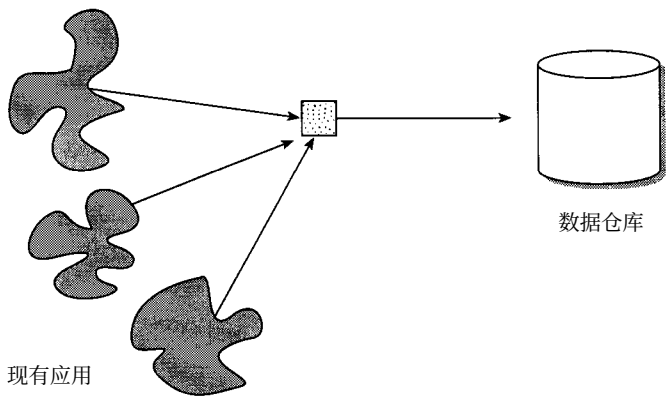


图3-1　把数据从操作型环境移入数据仓库不是简单的抽取

认为图3-1过于简单是有多种理由的。认为建造数据仓库仅仅是数据的抽取过程的观点之所以是错误的，主要是因为操作型环境中的数据是非集成的。图 3-2表明现存系统中缺乏数据集成是很常见的。

在以前构造现有应用时，没有考虑到将来可能还要集成。每个应用系统都有其独立的、特殊的需求，而且在开发过程中不曾考虑到其他的应用。因此毫不奇怪，在不同的地方用不同的名字存放相同的数据；或一些数据在不同的应用中用同一方式标明，但仍然是同样的数

据；或同一应用中某些数据用同样名字却用不同的度量，等等。试图从多处存在数据的地方抽取数据是个非常复杂的问题。

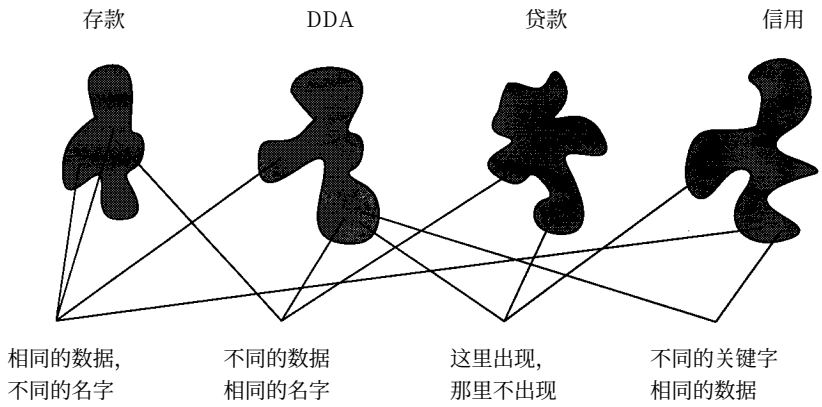


图3-2 跨越不同应用的数据集成性很差

这种缺乏集成的弊病正是程序员的梦魇。正如图 3-3所示，从操作型环境中适当地提取数据的编程过程中有很多细节需要说明。

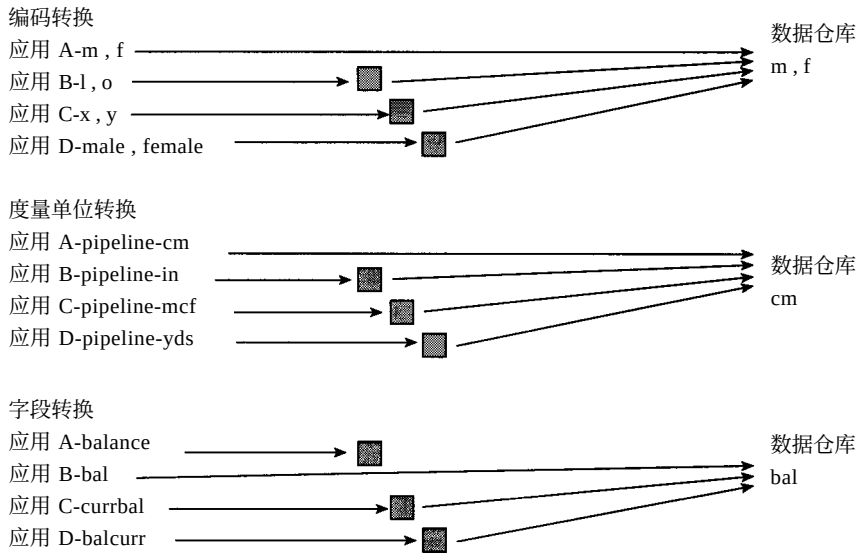


图3-3 为了合理地把数据从现有系统环境移动到数据仓库环境中，必须集成

例如，对“ 性别 ”字段，如果没有一致的数据类型，就是一个典型的缺乏集成性的例子：在一个应用中，性别编码为“ m/f ”，在另一个应用中的编码为“ 0/1 ”。其实，在数据仓库中只要字段的编码方式一致，“ 性别 ”字段到底怎么编码没有什么关系。因为当数据加入到数据仓库时，字段值必须正确地译码并且采用合适的值重新写进记录。

另一个综合集成的例子是针对开发人员的。考虑到四个应用中有同样的字段 PIPELINE，但是每个应用中的度量是不一样的。在应用 A中用的是英寸，在应用 B中用的是厘米，等等。其实，在数据仓库中只要PIPELINE字段的度量单位一致，究竟用什么无关紧要。

字段转换是数据集成的又一个问题。例如同一字段在四个应用中有不同的名字。为了保证转换到数据仓库的数据正确，必须建立不同源字段到数据仓库字段的映射。

这些简单的例子几乎还未涉及到集成的最浅层问题，例子本身也并不复杂。但是当它们在很多的系统和文件中存在时，集成问题就成了十分复杂而又繁重的工作了。

但是，现存系统的集成(或缺乏集成)并不是从操作型的现存系统到数据仓库系统中的数据转换工作的唯一难点。另一主要问题是存取现存系统数据的效率。扫描现存系统的程序如何知道一个文件已经被扫描过？现存系统环境中有大量的数据，每次数据仓库扫描时都试图对这些数据扫描一次，将是极大的浪费，同时也是不现实的。

从操作型环境到数据仓库有三种装载工作要做：

- 装载档案数据；
- 装载在操作型系统中目前已有的数据；
- 将自数据库上次刷新以来在操作型环境中不断发生的变化 (更新)从操作型环境中装载到数据仓库中。

一般来说，装载档案数据的难度不是很大，因为经常不做这项工作。不少企业发现，在很多环境下，使用旧的数据在成本上是不合算的。

从现有的操作型系统中装载数据，由于只需要装载一次，所以难度也不大。通常现存系统环境可以下载到一个顺序文件中，而这个顺序文件可以在不破坏联机环境的前提下下载到数据仓库。

对数据设计者而言，在实时的情况下装载数据 —— 操作型环境正发生变化 —— 是最为困难的。要有效地捕捉到那些变化并对之进行处理并非是一件容易的事。于是，扫描已有的文件成了数据仓库体系结构设计者主要面对的问题。

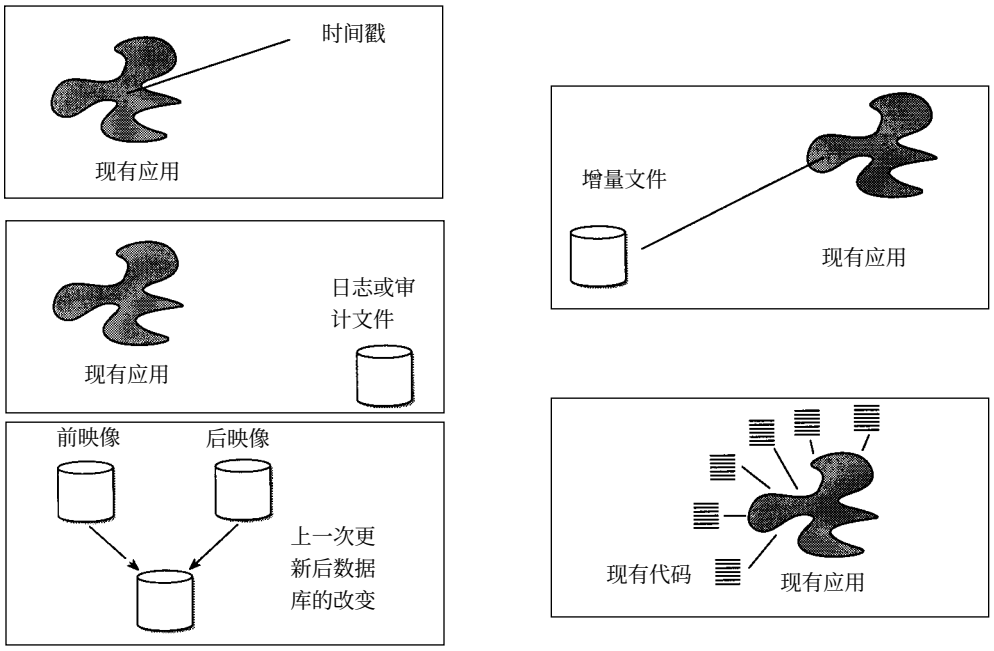


图3-4 你知道要怎样扫描数据？是否你每天、每周扫描每个数据

有五种通用技术用于限制数据的扫描量，正如图 3-4所示。第一种技术是扫描那些被打上时戳的数据。当一个应用对记录的最近一次变化或更改打上时戳时，数据仓库扫描就能够很有效地进行，因为日期不相符的数据就接触不到了。然而，目前的数据被打上时戳的很少。

数据仓库抽取中限制数据扫描量的第二种技术是扫描增量文件。增量文件由应用程序生成，仅仅记录应用中所发生的改变。有了增量文件，扫描的过程就会变得高效，因为不在候选扫描集中的数据永远不会涉及到。但是，许多应用程序并没有创建增量文件。

第三种技术是扫描审计文件或日志文件。审计文件或日志文件记录的内容，本质上同增量文件一样。不过，这里还是有一些重要的区别。由于恢复过程需要日志文件，所以各种操作都要保护日志文件。把日志文件用于其他目的，对计算机的操作也无大碍。利用日志文件的另一个困难是它内部格式是针对系统的用途而构造的，而不是针对应用程序的。这就需要一种技术手段作为日志文件内容的接口。日志文件的另一个缺点是其中所包含的内容超出了数据仓库开发人员所需要的。审计文件有许多与日志文件相同的缺点。

当数据仓库抽取数据时，控制扫描数据量的第四种技术是修改应用程序代码。这并不常用，因为很多应用程序的代码陈旧而且不易修改。

最后一个选择(很多情况下，是一个可怕的选择，其目的是使人们相信一定有更好的办法 )是将一个“ 前 ”映象文件和一个“ 后 ”映象文件进行比较。使用这种方法，一开始抽取就对数据库进行快照(snapshot)。进行另一个抽取时，就进行另一个快照。这两个快照逐次比较，以确定哪个活动发生了。这种方法很麻烦、复杂，还需要各种各样的资源。这只不过是最后的手段。

但是，集成和性能并不是仅有的两个使得简单的抽取过程无法用于构造数据仓库的主要问题。第三个主要困难是时基变化，如图 3-5所示。

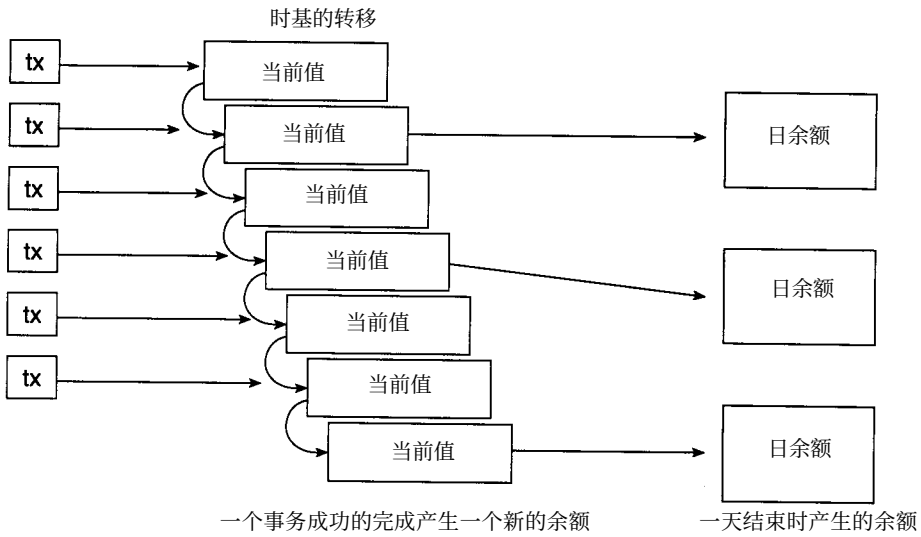


图3-5 当数据从操作型环境移动到数据仓库环境时，需要时基的转移

现存的操作型数据通常是当前值数据。当前值数据在被访问的时刻其精度是有效的，而且是可更新的。但是数据仓库中的数据是不能更新的。这些数据必须附有时间元素。当数据从操作型系统传送到数据仓库时，必需在数据中进行较大范围的改变。

当数据从现存操作型环境传送到数据仓库时，要考虑的另一个问题是需要对数据的量进行管理。数据要浓缩，否则数据仓库的数据量很快就会失控。在数据抽取一开始就要进行数据浓缩。图3-6表示数据仓库数据浓缩的一个简单形式。

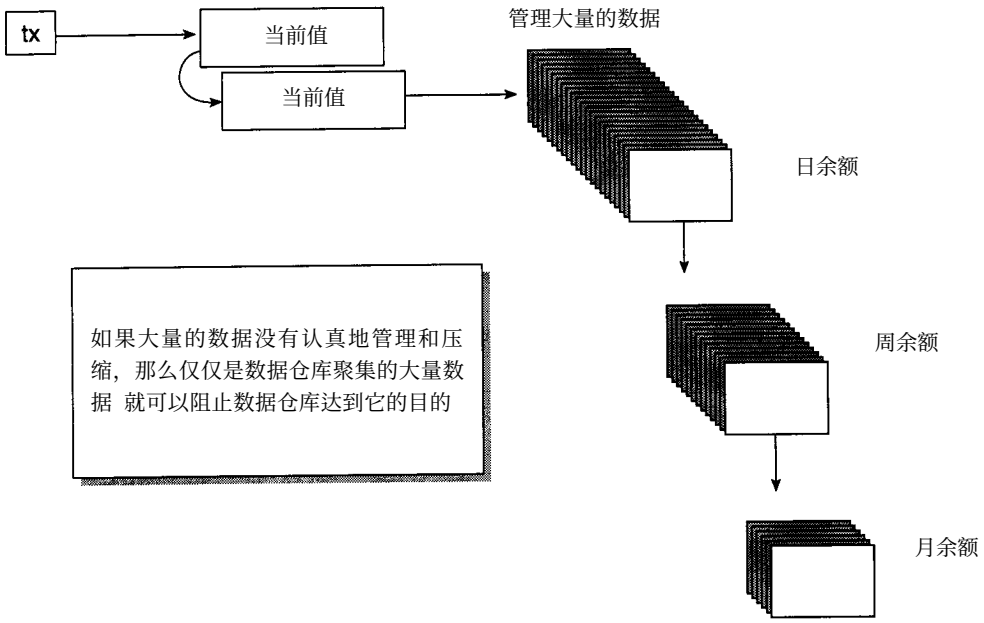


图3-6 数据的压缩对于管理数据仓库至关重要

## 3.2 数据/过程模型和体系结构设计环境

尝试使用传统的设计方法前，设计者必须明白这些方法的适用范围与其局限性。图 3-7说明了体系结构层次与数据建模和过程建模规程之间的关系。过程模型仅仅适用于操作型环境。数据模型既可用于操作型环境，又可用于数据仓库环境。数据模型或过程模型用错了地方，只会带来失败。

在本章后面会更详细地讨论数据模型。什么是过程模型？一个过程模型 (整个或部分地)一般包括以下内容：

- 功能分解。
- 第零层上下文图表。
- 数据流图。
- 结构图表。
- 状态转换图。
- HIPO图。
- 伪代码。

在许多场合和环境下，过程模型是很有价值的。但在建造数据仓库时，过程模型是个障碍。过程模型是基于需求的，它假设在细节设计开始之前是知道需求的。在处理过程时，是可以这样假设的。但这样的假设在建造数据仓库时是不成立的。其实许多开发工具，如 CASE 工具具有相同的功能定位，为此，它们不适用于数据仓库环境。

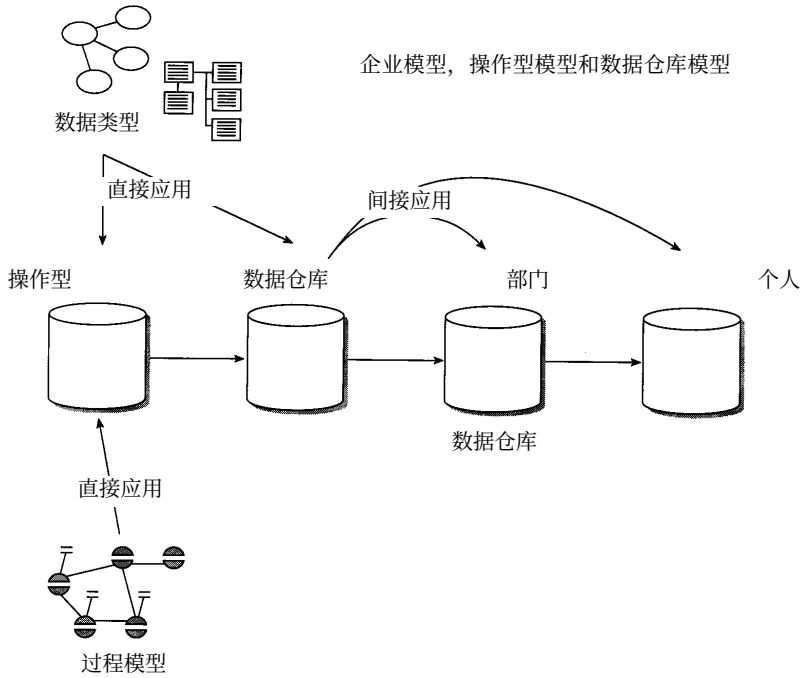


图3-7 不同类型的模型怎样应用于体系结构设计环境

## 3.3 数据仓库和数据模型

数据模型既适用于现存系统环境也适用于数据仓库中环境，图 3-8做了更清楚的解释。

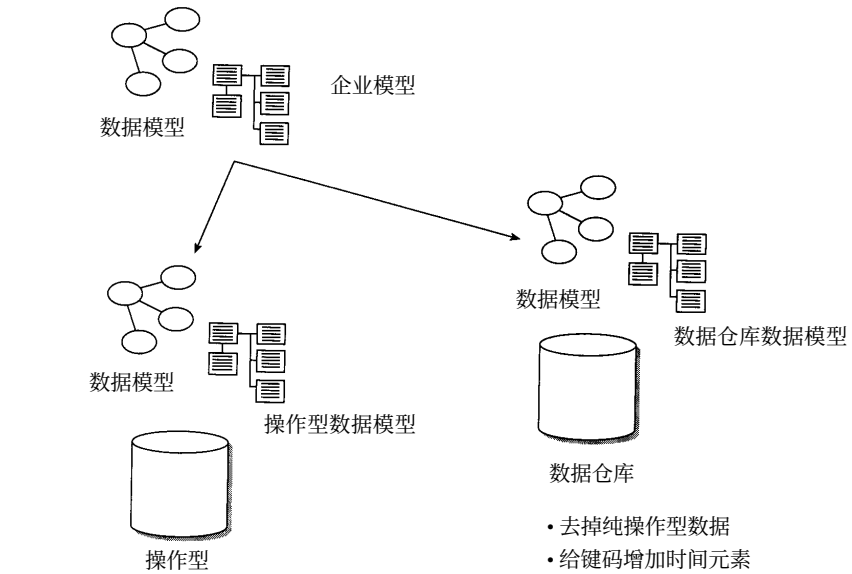


图3-8 不同层次模型间的关系

图3-8所示的是一个企业数据模型，该模型建造时没有考虑现存的、操作型系统与数据仓库之间的差别。企业数据模型只包含原始数据。要建造一个单独的现存数据模型，需要从企业模型开始。然而，当企业数据模型传送到现存系统环境中时，性能因素也应加到该模型中。总之，企业数据模型用于操作型系统时，几乎不用做什么改动。

但是，企业模型用到数据仓库中要做相当多的改动。首先要做的是除去纯粹用于操作型环境的数据。然后，在企业数据模型的键码结构中增加时间元素。导出数据加到企业数据模型中，在那里导出数据作为公用并只计算一次，而不重复计算。最后，操作型系统中的数据关系在数据仓库中就转变为“人工关系”。

设计的最后一项设计工作是企业数据模型到数据仓库数据模型的“稳定性”分析。稳定性分析是根据各个数据属性的变化特性将这些属性分组。图 3-9说明了稳定性分析。

零件表　　很少更改　　有时更改　　频繁更改

图3-9 一个制造业环境中稳定性分析的例子，显示出如何从一个大的、通用的表中根据包含在表中的数据的稳定性需求而产生三个表

在图3-9中，不常变化的数据聚集在一起，时而变化的数据聚集在一起，常变化的数据聚集在一起。稳定性分析的最终结果 (这是物理数据库设计前数据建模的最后一步 )是具有相似特性的数据聚集在一起。

于是，有一个数据模型的共同起源。可以作一个类比，企业数据模型是亚当，操作型数据模型是凯恩，数据仓库的数据模型是亚伯。他们都源于同一个血统，但同时又互不相同。

### 3.3.1 数据模型

(已有一些其他的书籍论述数据建模。有很多的方法可供选择。随便找出几种都可以成功地用来建造数据仓库。本书要讨论的方法是总结性质的，进一步的探讨请参阅《 Information Systems Architecture》, Wiley。)

有三个层次的数据建模：高层建模 (ERD，实体关系层)，中间层建模 (DIS，数据项集)，底层建模(物理层)。

高层建模的特点是实体和关系，如图 3-10所示。实体的名字放在椭圆内。实体间的关系用箭头描述。箭头的方向和数量表示关系的基数，只有直接的关系才标志。这样做以后，关系的传递依赖就可以最小化。

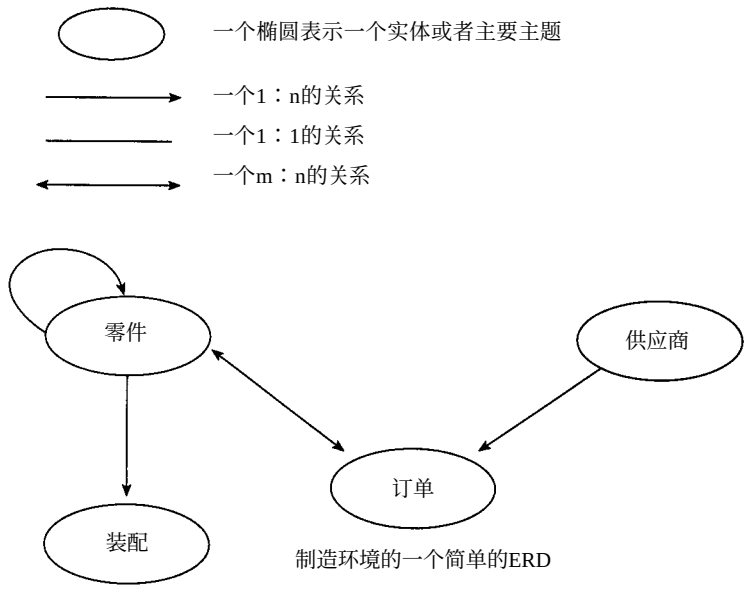


图3-10 表现实体和关系

在ERD层的实体位于最高抽象层。哪些实体属于模型的范围，哪些实体不属于，是由所谓的“ 集成范围 ”来决定的，如图 3-11所示。

集成范围定义了数据模型的边界，而且集成范围需要在建模之前进行定义。这个范围由系统的建模者、管理人员和最终用户共同决定。如果范围没有事先确定，建模过程就会一直持续下去。书写集成范围的定义应该不超过 5页，而且应该使用业务人员懂得的语言。

企业ERD由很多反映了整个企业不同人员的不同观点的单个的ERD合成的，如图3-12所示。已经为企业内不同的群体建立好了各自独立的高层数据模型，这些模型共同组成这个企业ERD。

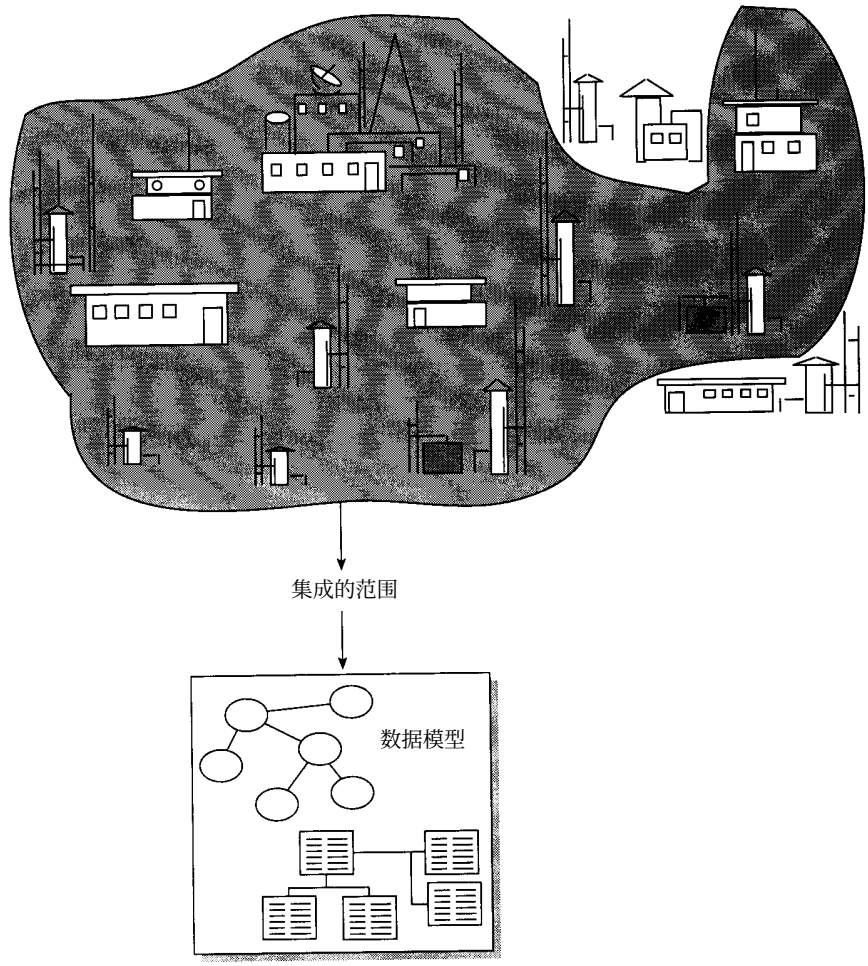


图3-11 集成范围决定了企业的哪些部分反映在数据模型中

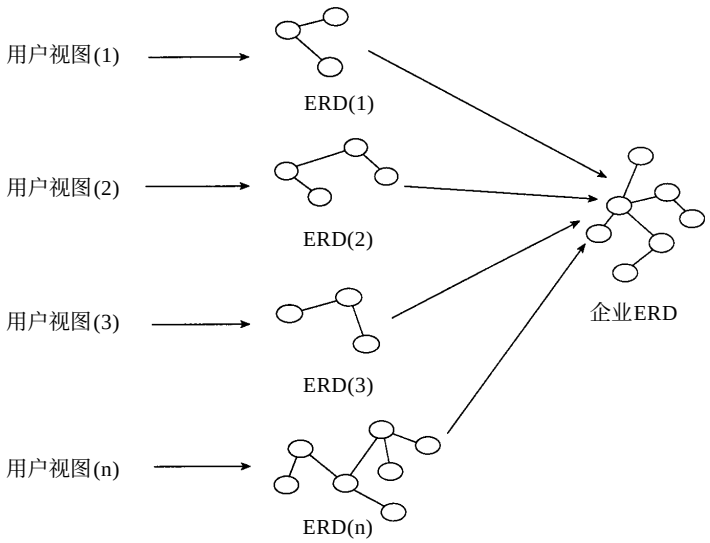


图3-12 企业ERD的构造来自于不同的用户视图ERD

公共的ERD依照用户的观点而建，这些观点来自与不同部门适当的人员的交流。

### 3.3.2 中间层数据模型

高层数据模型建好后，下一步就是建中间层模型 (DIS)。对高层模型中标识的每个主要的主题域，或实体，都要建一个中间层模型，如图 3-13所示。高层数据模型标识了四个实体或主要主题域，每个主题域再扩展成自己的中间层模型。

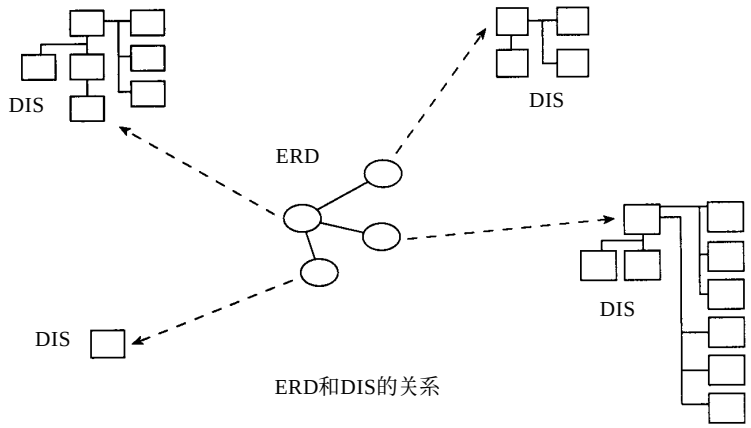


图3-13 ERD中的每一个实体将来都会被它的DIS所定义

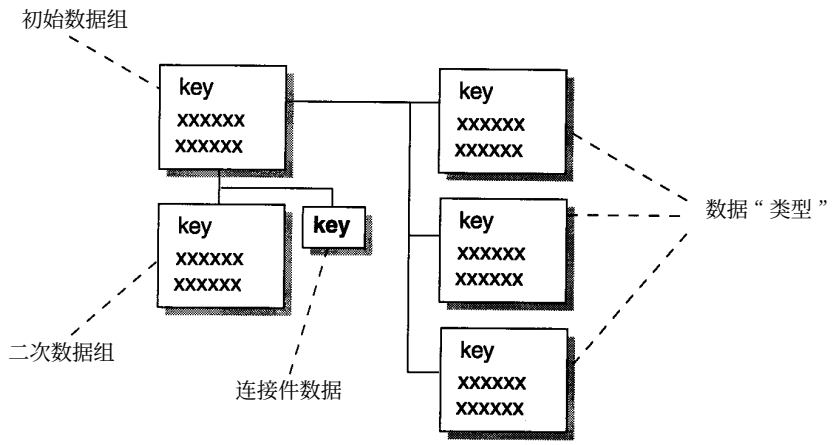


图3-14 中间层数据模型的四个组成部分

有趣的是，只有在很少的情况下，所有的中间层模型能一次全部建好。某个主要主题域的中间层数据模型扩展后，这个中间层模型逐渐增大，而此时模型的其他部分仍然保持不变。图3-14显示了中间层数据模型的构造。这里有四个基本的构造：

- 初始数据组。
- 二次数据组。
- 连接件，表示主要主题域间的数据关系。
- 数据“类型”。

初始数据组对每个主要主题域存在且只存在一次。它有在每个主要主题域只出现一次的属性。同所有的数据组一样，初始数据组有属性和键码。

二次数据组有对每个主要主题域可以存在多次的属性。从初始数据组有一直线段指向二次数据分组。有多少可以出现多次的不同数据组，就含有多少二次数据组。

模型的第三个构造是连接件。它将数据从一个组到另一个组联系起来。一个 ERD层确定的关系导致了DIS层的确认。用来指示连接件的惯例是一个有下划线的外键。

模型的第四个构造是数据的“类型”。数据的“类型”由指向右边数据组的线段指示。左边的数据组是超类型，右边的数据组是子类型。

这四个数据模型构造用来标识数据模型中的数据属性和这些属性间的关系。当一个关系在ERD层标识以后，在DIS层就用一对连接件关系来表现，图 3-15就指出了其中一对。

在ERD，在顾客(CUSTOMER)和帐户(ACCOUNT)之间的关系已经表示出来。对于帐户的DIS层，在帐户下有一个连接件。这说明一个帐户可能附有多个顾客。在顾客的 DIS层，顾客下对应的关系没有表示出来。在顾客的 DIS层，应该有一个到帐户的连接件，说明一个顾客可以有一个或多个帐户。

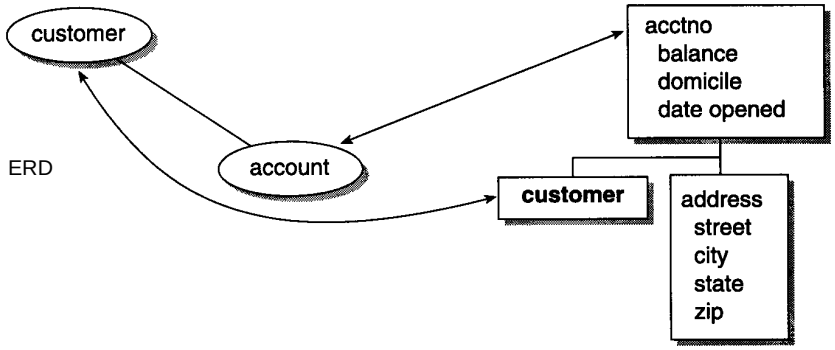


图3-15 ERD标明的关系在DIS中由连接件反映，注意只有一个连接件(从acctno到customer)在图中显示。实际上，另一个连接件从customer到acctno将在customer的DIS中显示

作为一个DIS详细情况的例子，如图 3-16所示。这是一个对于一个金融机构帐户的 DIS，显示了DIS里所有不同的构造。

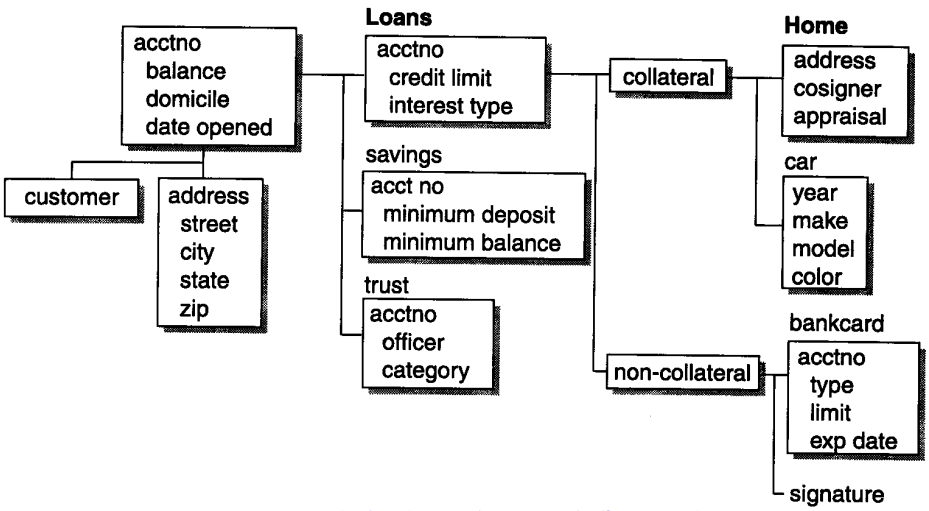


图3-16 一个扩展的表明银行可提供贷款类型的DIS

特别注意一下这样一个情况，就是一个数据组有两个“类型”线引出来，如图 3-17所示。

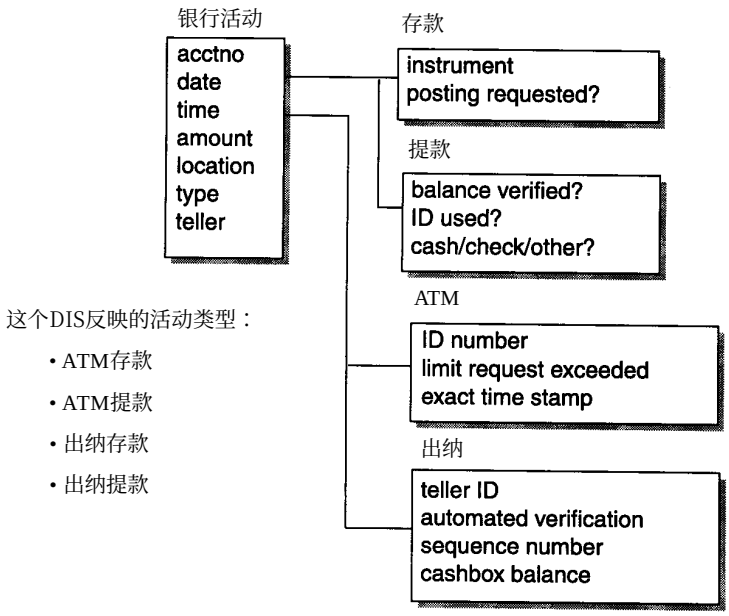


图3-17 显示不同子分类标准的一个DIS

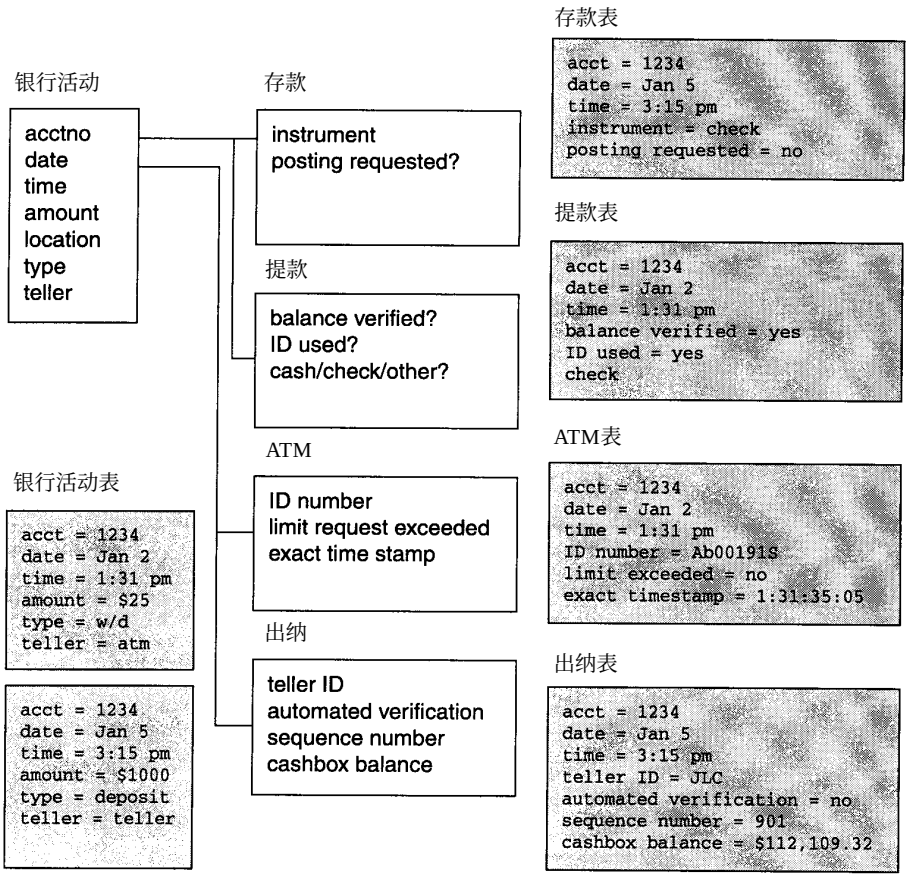


图3-18 两个事务所表现的表的条目

到右边的两条线说明有两个标准“类型”。一条线是活动类型——或者是存款或者是提款。另一条线是活动——或者是ATM活动或者是出纳活动。两种类型的活动都包括下面的交易：

- ATM存款。
- ATM提款。
- 出纳存款。
- 出纳提款。

这个图表的另一个特点是公用数据在左边，所有的独有数据在右边。例如，日期( date)和时间(time)属性对于所有交易是公用的。但是，钱箱 (cashbox)的结算却是出纳独有的活动。

由数据模型产生的和数据模型物理数据表的关系如图 3-18所示。一般来讲，数据模型的每个数据组都产生一个在数据库设计过程中定义的表。假设下面一种情况，两个交易产生一些表条目，如图所示。下面的两个交易产生图中的物理表条目。

- ATM处理提款，在1月2日，下午1:31。
- 出纳处理存款，在1月5日，下午3:15。

两个交易生成5个不同表的6个条目。

与企业ERD是由反映不同用户群体的不同 ERD所建成的一样，企业 DIS由多个DIS建成，如图3-19所示。在进行对个别用户的访问或 JAD(联合应用程序设计 )会议时，就要生成一个DIS和一个ERD。小范围的DIS和其他所有DIS一起形成一个反映企业观点的DIS。

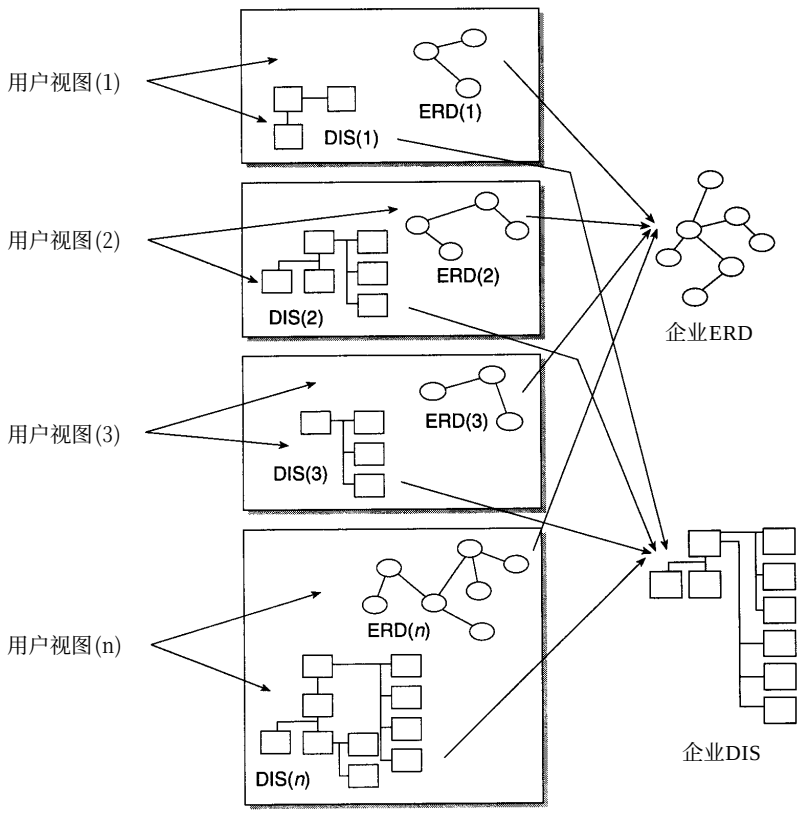


图3-19 企业DIS是由每个用户视图会话结果的DIS组成的

### 3.3.3 物理数据模型

物理数据模型是由中间层数据模型创建的，它只是通过包含键码和模型的物理特性来扩展中间层数据模型而得到的。这时，物理数据模型看上去像一系列表，这些表有时称做关系表。

人们很容易认为这些表是准备用来实现物理数据库设计的。然而，还有设计的最后一步——确定性能特性。

在数据仓库的情况下，确定操作性能特性的第一步意味着决定数据的粒度与分割，必须这样做。(当然，键码结构要做改变，以便能加入与每个数据单元都相关的时间元。 )

做完粒度与分割后，许多其他的物理设计活动就要加进这项设计了。这些其他的物理设计因素的概要如图3-20所示。物理设计因素的中心在于物理 I/O(输入/输出)。I/O就是将数据从存储器上调入计算机，或者将数据从计算机送到存储器，图 3-21就是简单的I/O例子。

数据在计算机和存储器之间的调入调出是按块进行的。对性能来说 I/O事件如此重要是因为存储器和计算机间的数据传输速度比计算机运算速度要慢大约两到三个数量级。计算机内部运算速度以毫微秒计，而数据的传输速度是以毫秒计。因此，物理 I/O是主要影响性能的因素。

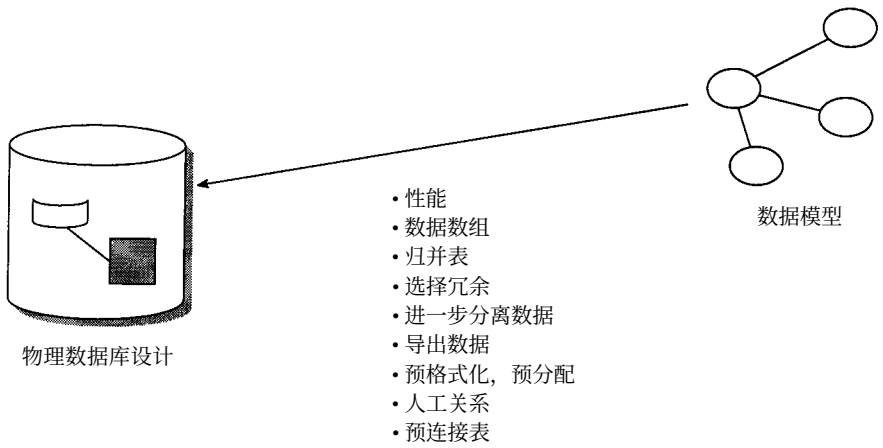


图3-20 从数据仓库环境中获得好的性能


图3-21 取得最高的物理I/O效率

数据仓库设计者的工作是要物理地组织好数据，以便返回最大数量的记录，这些数据是执行物理I/O时产生的(注：这不是盲目地将大量记录从 DASD传到主存上。这是一个很复杂的问题，即传输那些具有高访问概率的大量记录。 )

例如，假定程序员要取5个记录。如果这些记录是在存贮器中不同的数据块上，那么就需要五次I/O操作。但是如果程序员能够预测到这些数据将成组的访问，将其并列地放在同一个物理块中，那么这就只需要一次I/O操作，这样使得程序的运行效率更高。

关于数据仓库中数据存放的问题还有一个缓和因素：数据仓库里的数据一般不更新。这样设

计者就可以自由地采用物理设计技术，这些技术在数据需要经常更新的情况下很可能就不能接受。

## 3.4 数据模型和反复开发

在所有情况下数据仓库最好是反复建造的。下面就是为什么说反复建造是很重要的理由：

- 业界成功的记录强烈地建议这样做。
- 最终用户在第一遍完成以前不能明白地提出需求。
- 只有实际结果切实而且明确时，管理部门才能作出充分的承诺。
- 需要很快看到可视化结果。

这时，可能尚不清楚的是数据模型在反复开发中担当的角色，为了解释数据模型在反复开发期间所起的作用，考虑典型的如图 3-22所示的反复开发过程。首先进行一遍开发，然后另一遍，等等。数据仓库在每遍开发中都起着路标的作用，如图 3-23所示。

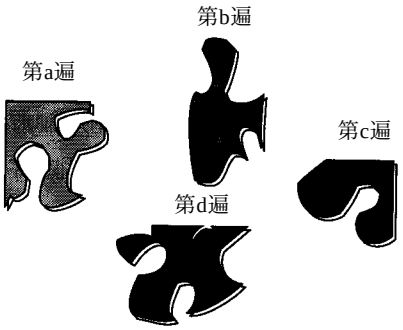


图3-22 数据仓库开发的不同阶段

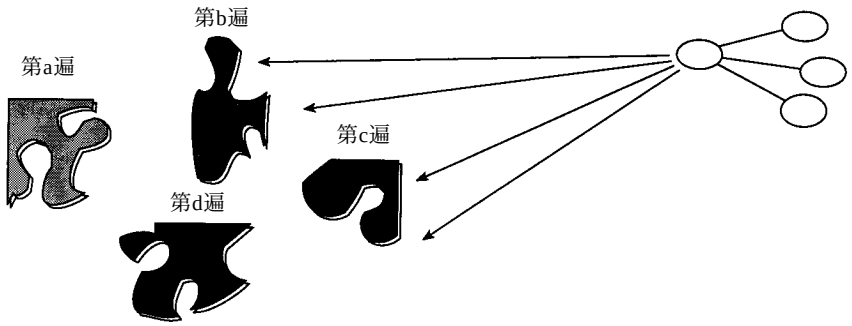


图3-23 数据模型允许开发的不同重复阶段以一种紧密结合的方式进行

当第二遍开发继续接着进行时，开发人员相信，他或她将汇集第一遍开发的成果，因为所有的开发都是数据模型驱动的。每遍后续开发都是建立在前一遍开发的基础上，结果就是都在统一的数据模型上进行不同的开发。正是由于它们都是基于同一个数据模型，各遍开发工作的成果将产生一个内聚的、高度和谐的整体，见图 3-24。

当不同遍的开发是在不同的数据模型之上进行时，就会产生很多重复的工作和很多不连贯的开发，图3-25就说明了这个不协调的结果。

在数据仓库的增量式开发和反复式开发的过程中，在数据模型与达到长期集成和和谐工作的能力之间，存在一个间接的但很重要的相互关系。

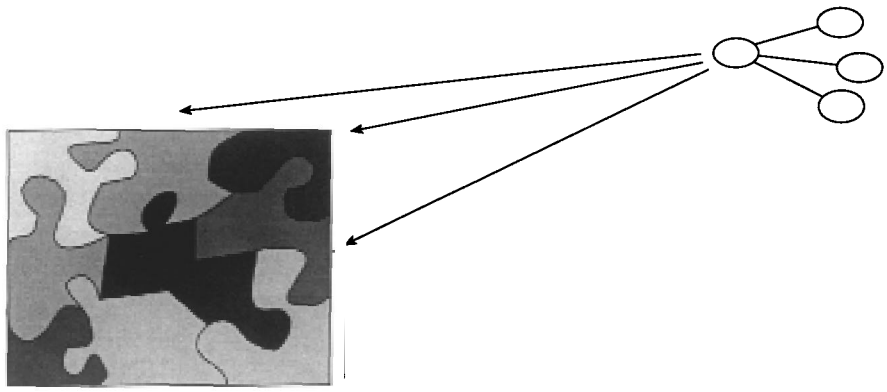

图3-24 在开发工作结束时，所有遍的结果融合在一起

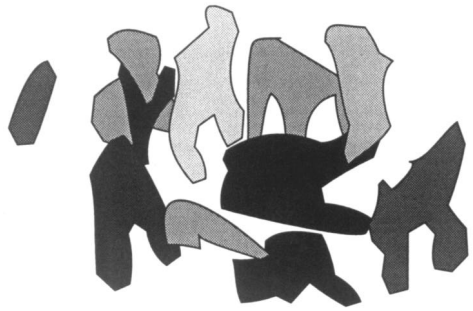

图3-25 如果没有数据模型，重复的开发不能构成一个内聚的模式有许多重叠和缺乏一致性

## 3.5 规范化/反规范化

数据模型处理过程的输出是一系列表，每个表包含键码和属性。常规的输出是很多表，其中每个表只包含少量数据。

虽然输出大量小表本身没有什么不对，但从性能上看是一个问题。请考虑程序为使表之间的动态互连而必须做的工作，如图 3-26所示。

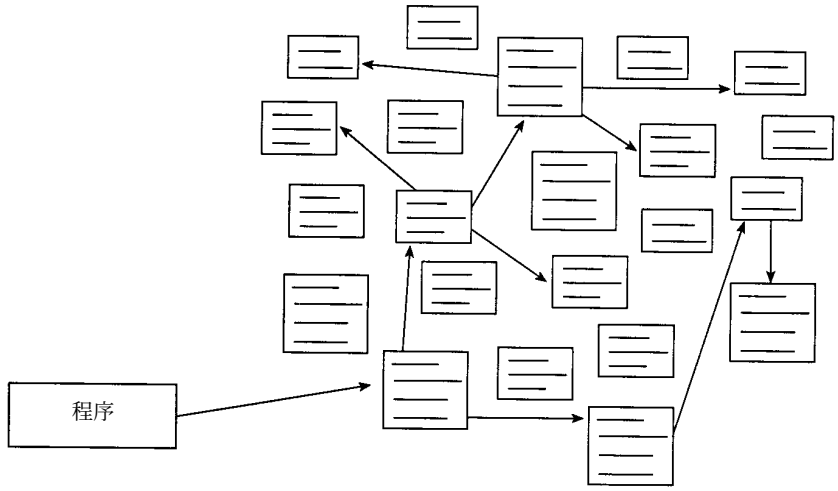


图3-26 当有许多表时，动态互连能力要求大量的I/O

在图3-26中，一个程序开始执行。首先，一个表被访问，然后另一个表被访问。为了成功运行，程序必须在很多表中跳转。每次程序从一个表跳到另一个，就要进行 I/O,既要存取数据，又要存取索引以找到数据。如果只有一两个程序需要进行 I/O，那是不成问题的。但是如果当所有的程序需要进行大量的 I/O时，性能就会受到影响。当物理设计生成的是很多小表，而且每个小表都只有有限量的数据时，所发生的就正好是这种情况。

一个较为合理的方法是将这些表合并，使得只需进行最低限度的 I/O，如图3-27所示，其中很多小表都已经在物理上合并在一起。

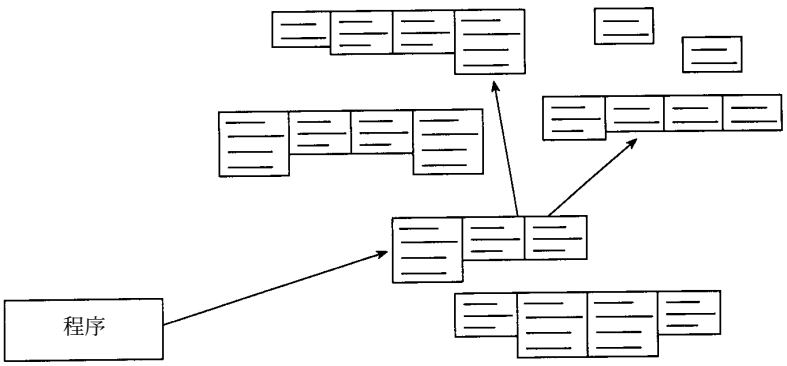


图3-27 当表在物理上被归并，就需要较少的I/O

这样做以后，同样的程序跟以前一样运行，只是现在只需要少得多的I/O去完成同样的工作。

于是，问题就转化，采取一种什么策略将表合并起来以获得最大的收益才是明智的？要回答这个问题，就是数据库物理设计人员需要做的事了。

但是合并表只是能够节省 I/O的设计技术之一。另一个非常有用的技术是生成一个数据数组。在图3-28中，数据是规范化的，使得每产生一个数据序列值都存在一个不同的物理位置。每次检索每一个值， n,n+1,n+2,...,都需要进行一次物理 I/O才能得到数据。如果数据存放在一个数组的单一行中，那么一次 I/O就足以检索到了，如图中下部所示。

当然，并不是在所有情况下创建一个数据数组都是有意义的。仅当数据序列产生的数量是稳定的，数据是按序列存取的，且数据的创建与修改在统计上是以很规律的方式进行的时候，创建一个数据数组才是有意义的。

有趣的是，在数据仓库中，由于数据具有基于时间特性，这些情况是经常发生的。数据仓库中的数据经常与某个时刻相关，而且时间部分以很有规律的形式出现。在数据仓库中，例如，每月创建一个数组，是很容易而且是很自然的事情。

另一个重要的，特别是和数据仓库相关的设计技术是引入冗余数据。图 3-29就给出了一个引入冗余数据而带来好处的例子。在图的上部，字段(描述信息)是规格化的、无冗余的。这样，所有需要查看一个部件描述的过程都必须访问基本部件表。尽管数据的更新是优化的，但对这些数据的访问开销是很大的。

在图3-29的下部，数据元素(描述信息)有意地放在了可能要用到它的许多表中。这样做使访问数据效率更高，而数据的更新是不优化的。然而，对于那些广泛使用的数据(如描述信息)，和稳定的数据(也如描述信息)，几乎没有理由担心更新问题。尤其是在数据仓库中不用考虑更新。

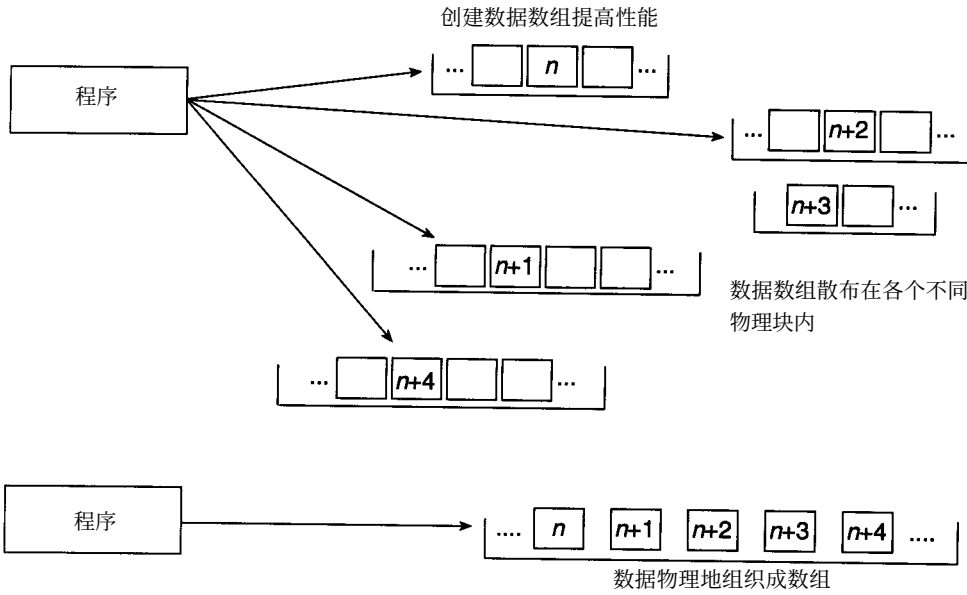


图3-28 在合适的情况下，创建数据数组可以节约可观的资源

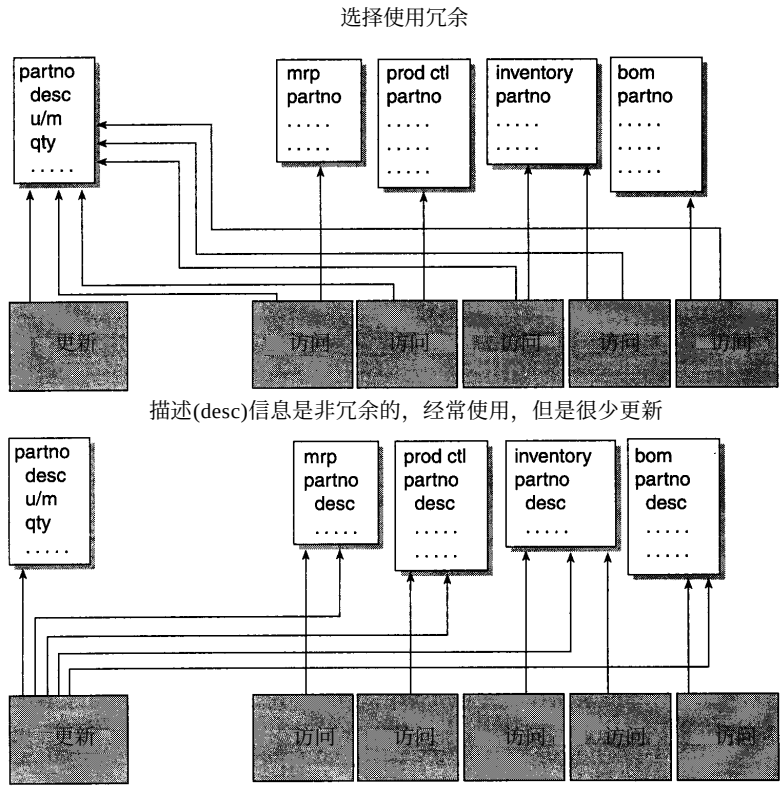


图3-29 描述是冗余的，散布在使用它的许多地方，当它被改变的时候每一处都应改变，但是它很少改变

另一个有用的技术是：当访问概率有很大悬殊时，要对数据做进一步分离。图 3-30给出

了这种情况。

在图3-30，考虑一个银行账户，账户住址(domicile)、开户数据(data opened)和余额(balance)是规范化的。但是，余额与其他两项数据的访问概率差别很大。余额经常用到，而其他数据则很少用到。为了使I/O效率高一些，并且使数据存放得更紧凑一些，可以将规范化的表分成两个独立的表，如图3-30所示。

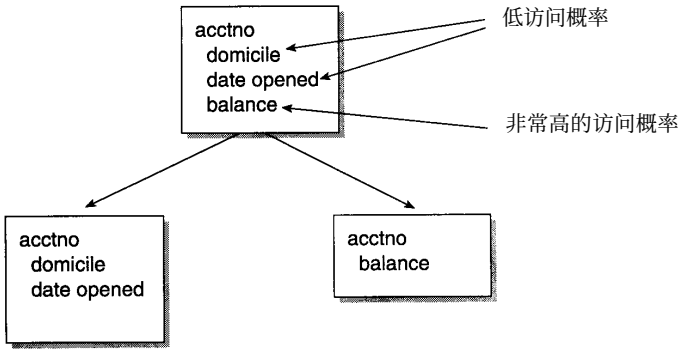


图3-30 根据访问概率的很大差异进一步分离数据

有时，在物理数据库的设计中引入导出 (即计算出的)数据可以减少I/O，图3-31给出了这样一个例子。一个程序为了计算出年薪和已付的税金，要定期访问工资清单。如果该程序在每年年底运行一次，那么生成一个字段来存储计算出的数据就很有意义了。这些数据只要计算一次，将来需要时只要访问那个字段就可以了。这个方法的另一个优点在于那个域的数据只用计算一次而不必重复计算，减少了由不正确计值导致错误算法的可能。

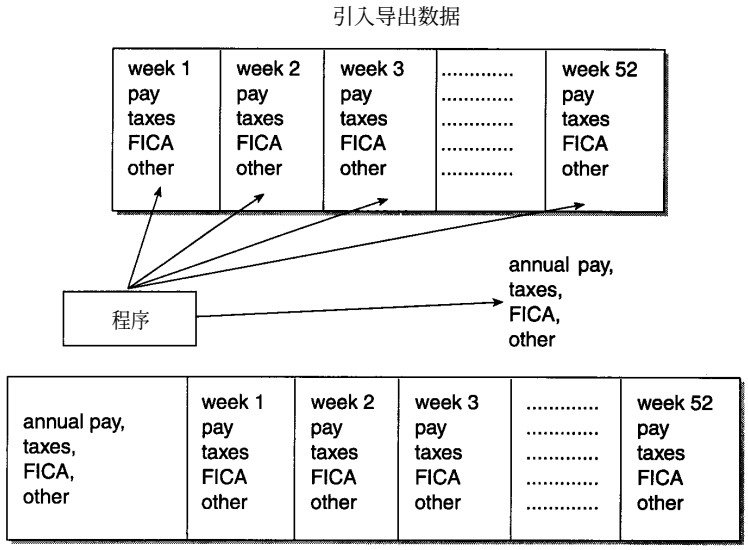


图3-31 导出数据，计算一次就可以永久使用了

建造数据仓库的一个最具革新意义的技术是建立所谓的“创造的”索引或创造的简要记录(由莫尔杜撰的术语)，图3-32给出了一个创造的索引的例子。创造的索引是当数据由操作型环境转移到数据仓库环境时建立起来的。由于每个数据单元必须能在任何情况下处理，所以

在这时计算或建立索引只要很少的开销。

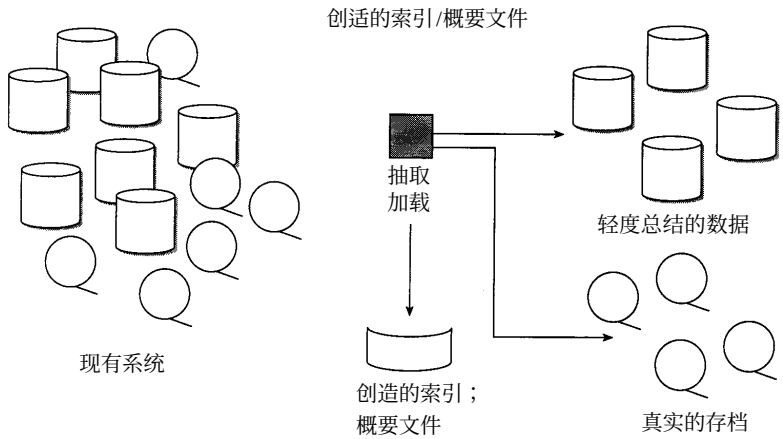


图3-32 创造的索引的例子：
- 卷中的前十名顾客是_____
- 这个抽取的平均的交易值是$nnn.nn
- 最大的交易值是$nnn.nn
- 查看交易活动而没有购买的顾客数nn

创造的索引为终端用户感兴趣的项目建立一个概要文件，例如“最大宗的购买”，最不活跃的帐户，最近的发货，等等。如果对管理有意义的需求能够预测(当然不是所有情况都如此)，那么在数据传送给数据仓库时建立“创造的索引”就有意义了。

数据仓库设计者要知道的最后一个设计技术就是数据仓库环境中参考完整性的管理。如图3-33所示，在数据仓库环境中，参考完整性表现为“人工关系”。

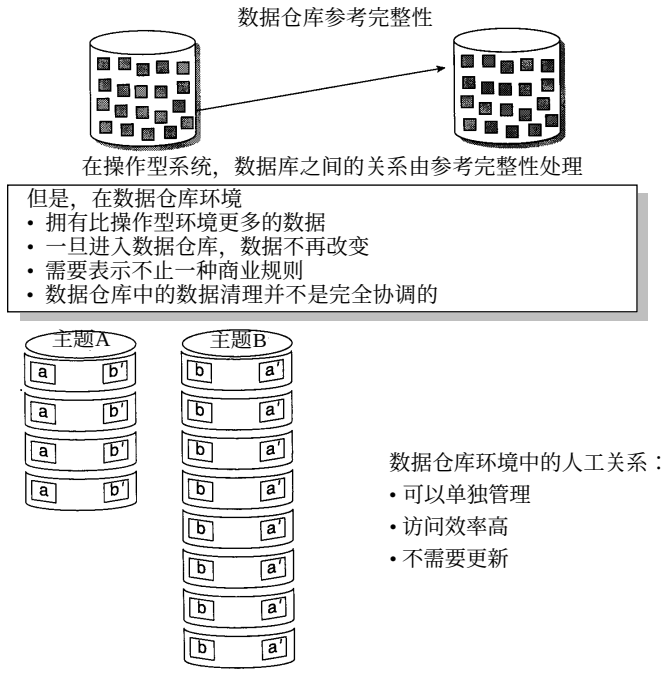


图3-33 数据仓库环境中的参考完整性

在操作型环境中，参考完整性表现为数据表之间的动态连接，但在数据仓库环境中有如下事实：数据量很大、数据是不更新的、数据按时间表示、关系不是静态的，因此应采取不同的方法表示参考完整性。总之，数据的关系在数据仓库环境中需人为地加以表示。这意味着有些数据要复制，有些要删除，同时其他数据仍然保留在数据仓库中。无论如何，试图在数据仓库环境中重复参考完整性，显然是不正确的方法。

## 3.6 数据仓库中的快照

数据仓库是应各种各样的应用和用户而建，如顾客系统，市场系统，销售系统和质量控制系统。不管数据仓库有什么非常不同的应用和类型，还是有一条共同的线索贯穿其中。在其内部，每个数据仓库都围绕着一个称之为“快照”的一种数据结构。

图3-34说明了数据仓库快照的基本组成形式。

快照是因为一些事件的发生而产生的。能够触发快照的事件有很多种。一类事件是对离散活动的信息的记录，例如填写支票，打电话，收到货物，完成订单，购买保险单等。在发生离散活动时，将会带来一些商业活动，并且需要记录下来。总之，离散活动是随机发生的。

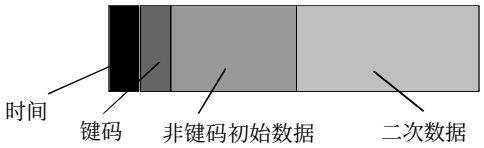


图3-34 数据仓库中的数据记录是一个时刻的快照，包括不同类型的数据

触发快照的另一类常见的事件是规定的时间点。在一个特定的时刻，快照就会触发。典型的时间点包括日末，周末，月末。当然，这些时间点是事先可预测的，并不是随机的。

由事件触发的快照有四个基本部分：

- 键码(KEY)。
- 时间单元。
- 只和键码相关联的初始数据。
- 作为快照过程的一部分所捕获的二次数据，和初始数据或键码无直接的关系。

在数据仓库的这些基本部分中，只有二次数据是可选的。

键码可以是唯一的也可以不唯一。键码可以是单一的数据元素。但在数据仓库中，更多的键码是由用来识别初始数据的很多数据元素组成的混合物。键码用来识别记录和初始数据。

时间单元根据时间元素生成，例如年、月、日、时和刻。时间单元通常是 (但并不总是)指快照所描述事件已经发生的时刻。有时时间单元指的是捕获数据的时刻。 (在一些情况下，事件发生时刻和捕获信息的时刻是不同的，而在另一些情况下是没有差别的。 )在由固定时间触发事件的情况下，时间元素可以暗含而不是直接附于快照中。

初始数据是与记录的键码直接相关的非键码数据。例如，假设键码表示产品销售，时间元素描述的是销售活动终结的时刻，初始数据描述的是销售什么产品以及销售的价格、条件、地点和代理。

二次数据，如果存在的话，表示快照生成时捕获的外来信息。如与销售相关的二次数据是一些关于产品出售的附带的信息 (例如成交时的股市价格 )。关于销售的另一个二次信息是销售时银行对优惠顾客的流行利率。有很多种伴随信息可以加到数据仓库记录中去，如果将来这

些信息会在DSS处理过程中使用到。注意这些加到快照中的伴随信息可以是也可以不是外键码。

一旦二次信息加到快照中——如果全部加入——初始信息和二次信息间的关系就可以推断出来，如图3-35所示。快照意味着在二次信息和初始信息之间存在关系。除此之外，没有什么别的了。这种关系是快照的即时反映。不过，产生快照时，从快照记录中初始数据和二次数据的并列，就能推出数据之间的关系。有时，这种导出关系叫做“人工关系”。快照记录是数据仓库中最为一般和最容易发现的记录的例子。

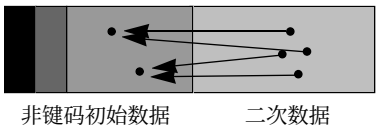


图3-35 由于用驻留于同一快照中的二次数据可能构成如像初始数据那样暗含的关系，人工关系被捕获

## 3.7 元数据

数据仓库环境中一个重要方面是元数据。元数据是关于数据的数据。只要有程序和数据，元数据就是信息处理环境的一部分。但是在数据仓库中，元数据扮演一个新的重要角色。也正因为有了元数据，可以最有效地利用数据仓库。元数据使得最终用户 /DSS分析员能够探索各种可能性。

元数据在数据仓库的上层，并且记录数据仓库中对象的位置。典型地，元数据记录：

- 程序员所知的数据结构。
- DSS分析员所知的数据结构。
- 数据仓库的源数据。
- 数据加入数据仓库时的转换。
- 数据模型。
- 数据模型和数据仓库的关系。
- 抽取数据的历史记录。

## 3.8 数据仓库中的管理参照表

大多数人想到数据仓库技术，就经常会想到不停地用来运行公司日常事务的一般大型数据库，例如顾客档案，销售记录，等等。事实上，这些一般文件构成了数据仓库工作的框架。然而，数据仓库中还有常被忽略的一类数据：参考数据。

参照表常常被认为是用来创建一个专门的问题。在很多情况下，参照表不是数据仓库的一部分。例如，假设一个公司有1995年的一些参照表，并且从1995年开始建立数据仓库。随着时间的推移，大量数据装载到数据仓库，同时，参照表用于操作上，并可能改变。在 1999年，需要将数据仓库与参照表进行比较。 1995年的数据与参照表作过一个比较，但参照表没有保持历史准确性， 1995年数据仓库数据与参照表数据相比结果是准确的，而 1999年数据相比就得到很不准确的结果。正因为这个原因，数据仓库既需要一般数据也需要参考数据。

参考数据特别适用于数据仓库环境，因为使用参考数据可以明显地减少数据仓库的数据

量。数据仓库环境中管理参考数据有很多的设计技术。这里要讨论两个，这两个技术恰恰是处于这些技术的两个对立端的技术。(在给出的设计选项中有很多选择和变型。)

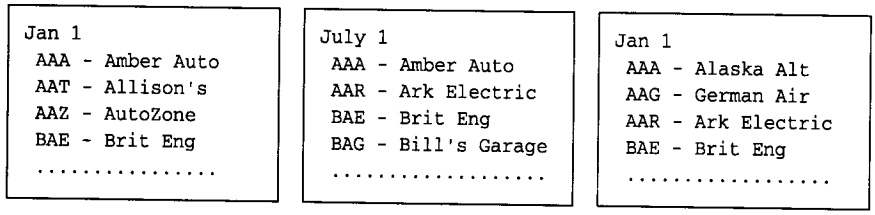


图3-36 一种管理数据仓库环境中的参照表的方法——每六个月产生一个参照表的快照

图3-36显示的是第一个设计选择，每6个月做完整参照表的一个快照。示例中一个快照是在1月1日生成，另一个是7月1日，如此下去。这个方法好在非常简单。每6个月生成一个小表的快照没有什么难度。但是这个方法逻辑上是不完善的。例如，假设参照表的一些活动发生在3月15日。如果加入一个新的条目——ddw，然后5月10日该条目被删除。每6个月做个快照就没法捕获3月15日到5月10日发生的活动。所以第一种管理参照表的方法是最简单的，但不完备。

在数据仓库环境中，对参照表的第二个管理方法如图 3-37所示。该图表示，在某一时间起点上，一个快照就是由一个参照表构成的。一年中参照表的活动都会被收集起来。为了确定某一时刻参照表某个给定条目的状态，该活动将按参照表进行重组。在这种方法中，表的逻辑完整性将可以在任何时刻被重新建立起来。然而，这种重建是一件复杂的事情，可能是一个巨大的而且复杂的任务。这种方法在逻辑上是完整的，但实现起来却是复杂的。

这里给出的两种方法在想法上是相反的。第一种方法较简单。但是逻辑上并不完善。第二种方法很复杂，但是却有逻辑上的完整性。在我们所讨论的这两种极端之间有很多可供选择的设计方案。但是在设计和实施的过程中，有必要把参照表作为数据仓库中正式的一个部分进行管理。

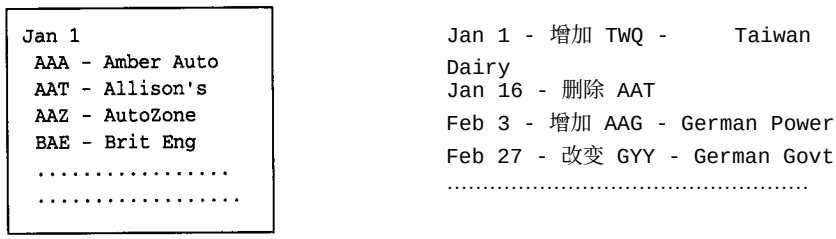


图3-37 另一个管理参照表的方法

## 3.9 数据周期

数据仓库设计中的一个引人注目的问题是数据周期。所谓数据周期是指从操作型环境数据发生改变起，到这个变化反映到数据仓库中所用的时间。考虑如图 3-38中所示的数据。

图中给出了关于Judy Jones的当前信息，数据仓库包含有Judy的历史信息。现在假设发现

Judy已经改变了她的地址。图3-39表明这个变化一旦被发现，就马上被反映到操作型环境中。

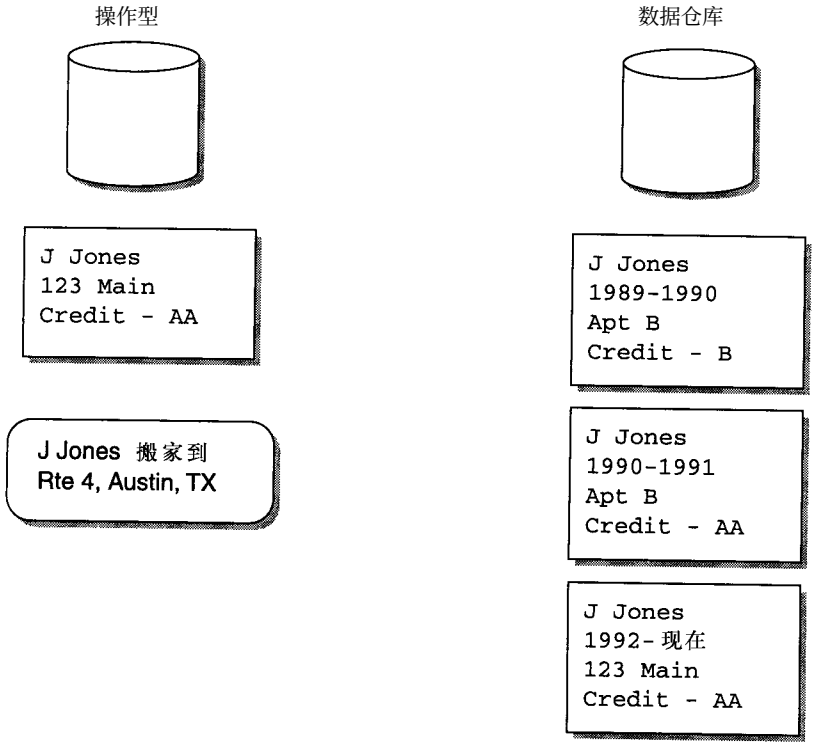


图3-38 当公司发现J Jones搬家后会发生什么？

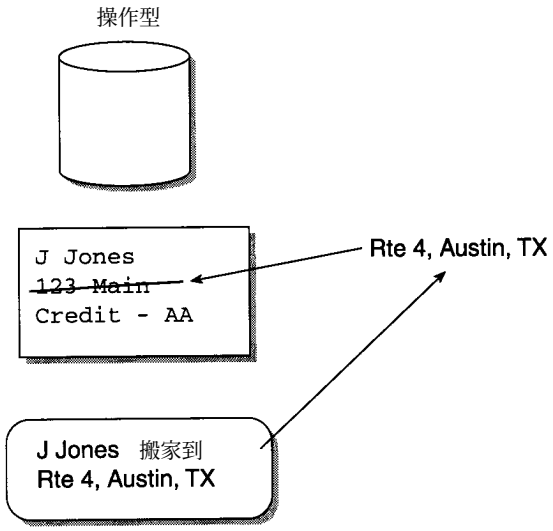


图3-39 第一步是在操作型环境中改变J Jones的地址

一旦数据反映到操作型环境中，这个变化就必须被转入数据仓库中。图 3-40表示数据仓库对最当前记录的终结日期进行了更正。并且插入了一条新的记录，这条记录反映了对有关Judy Jones的记录所做的修改。

问题是对数据仓库数据的这种调整，应该多长时间进行一次呢？原则上从操作型环境知道数据的改变到这个变化反映到数据仓库中至少应该经历 24小时(见图3-41)。没有必要急于把这个变化转入信息仓库中去，有几个需要把“时间间隔”放到数据中的原因：

首先，如果操作型环境与数据仓库相互之间结合得越紧密，那么所需的费用就越昂贵，技术也越复杂。24小时的时间间隔以现有技术来说将很容易被实现。

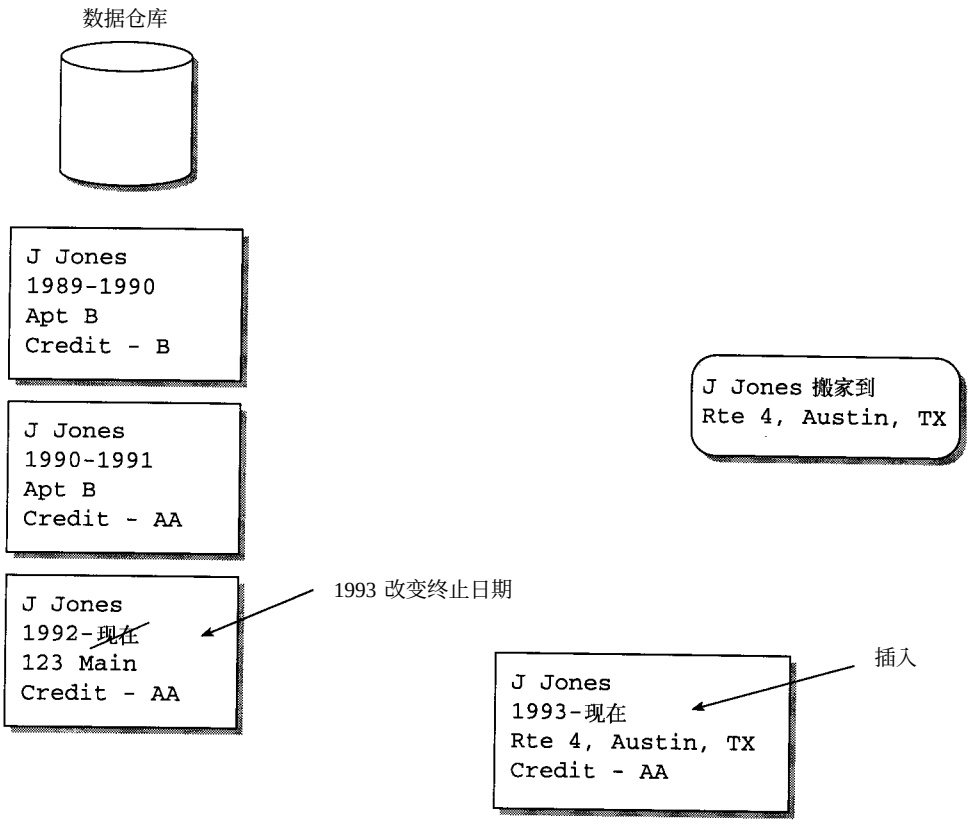


图3-40 改变地址引起的在数据仓库中出现的活动

但是，更有说服力的一个原因是，时间间隔给环境附加了一个特殊的限制。间隔 24小时，使得在数据仓库中不必做操作型处理；在操作型环境中不必做数据仓库处理。时间间隔的另一个好处是在转入数据仓库之前，数据能达到稳定。

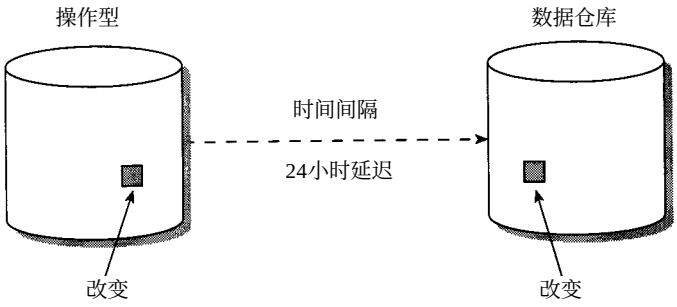


图3-41 在操作型环境知道改变到改变被反映在数据仓库之间的时间，至少需要24小时延迟

## 3.10 转换和集成的复杂性

粗略一看，当数据从传统环境转入数据仓库时，除了简单地从一个地方抽取数据再放入另一处，并没有做别的什么。由于表面上看起来很简单，很多组织开始手工建立他们的数据仓库。程序员只看到了数据从旧的操作型环境到新的数据仓库环境中的简单流动就轻率地说："我可以做到！"于是，在数据仓库设计、开发伊始，程序员往往就着手编写代码。

然而，第一印象通常是非常靠不住的。开始时认为仅仅是数据在不同环境中的简单传送，很快就会变成一个巨大的复杂的任务——比程序员所考虑的要大得多复杂得多。

准确地说，数据从操作型环境到数据仓库环境的传递要完成什么功能呢？下面就是所要完成的某些功能。

- 从原始操作型环境到数据仓库环境的数据抽取需要实现技术上的变化。一般包括，从操作型系统获取数据的数据库管理系统 (DBMS)技术，如IMS，以及将数据写入更新的数据仓库的DBMS技术，如INFORMIX。在数据传递过程中需要实现技术的转移。
- 从操作型环境中选择数据是非常复杂的。为了判定一个记录是否可进行抽取处理，往往需要完成对多个文件中其他记录的多种协调查询，需要进行键码读取，连接逻辑等。
- 操作型环境中的输入键码在输出到数据仓库之前往往需要重新建立。在操作型环境中读出和写入数据仓库系统时，输入键码很少能够保持不变。在简单情况下，在输出键码结构中加入时间成分。在复杂情况下，整个输入键码必须被重新散列或者重新构造。
- 数据被重新格式化。举一个简单例子：有关日期的输入数据格式是 YY/MM/DD，当它被写入输出文件时，需要转化为 DD/MM/YY的格式。(操作型数据进入数据仓库之前的格式转换往往比这要复杂得多。)
- 数据将被清理。在某些情况下，为了保证输入数据的正确性，需要一个简单的算法。在复杂情况下，需要调用人工智能的一些子程序把输入数据清理为可接受的输出形式。
- 存在多个输入数据源。在某些情况下数据仓库中数据项的来源是一个文件，而在另外一些情况下，则为另外一个文件。逻辑上必须分清楚，以便由适当的数据源提供正确条件下的数据。
- 当存在多个输入文件时，进行文件合并之前要首先进行键码解析。这意味着如果不同的输入文件使用不同的键码结构。那么，完成文件合并的程序必须提供键码解析功能。
- 当存在多个输入文件时，这些文件的顺序可能不相同甚至互不相容。在这种情况下这些输入文件需要进行重新排序。当有许多记录需要进行重新排序时可能有些困难，但可惜的是，通常都是这种情况。
- 可能会产生多个输出结果。同一个数据仓库的创建程序会产生不同概括层次之上的结果。
- 需要提供缺省值。有时候，数据仓库的一个输出值没有对应的输入源。这时，必须提供缺省值。
- 对抽取选择输入的数据，其效率通常是一个问题。我们考虑一个情况，在刷新的时候，我们没有办法将需要抽取的操作型数据和不需要抽取的操作型数据区别开来。这时，必须读取整个文件。而读取整个文件的方法效率很低，因为在一个文件中，只有一小部分的记录是用得上的。这将导致在线环境一直处于活动状态，进而挤掉了其他的处

理活动。

- 经常需要进行数据的汇总。多个操作型输入记录被连接成单个的“简要”数据仓库记录。为了完成汇总，那些需要汇总的详细的输入记录必须被正确排序。当把不同类型的记录汇总为一个数据仓库记录时，这些不同输入记录类型的到达次序必须进行协调，以便产生一个单一记录。
- 在数据元素从操作型环境到数据仓库的数据转移过程中，对数据元素的重命名应该进行跟踪。当一个数据项移动时，往往被改变名字。这样就必须生成记录这些变化的文档。
- 必须被读的输入的记录有着不常见的或不标准的格式。下面列出了必须读的各种输入类型：
  - 固定长度的记录
  - 不定长记录
  - 出现不定
  - 重复出现
- 必须进行转换。但是必须指定转换逻辑，转换机制（正如BEFORE，AFTER)会非常复杂。有时转换逻辑变得非常曲折。
- 这一点也许是最糟糕的：建立在旧的传统程序逻辑中的数据之间的关系必须被理解、被解开，这样，这些文件才能被用来作为输入。而这些关系常常是深奥难懂的，没有可供参考的文档资料。
- 必须进行数据格式的转换。EBCDIC到ASCII的转换(或反过来)必须进行。
- 必须考虑到进行大容量输入的问题。当只有少量的数据作为输入时，可以有很多供选择的方案。但是，当有大量的记录作为输入时，就必须引入一些特殊的设计方案（如并行装载和并行读出)。
- 数据仓库的设计必须符合企业数据模型。这样数据仓库的设计和建立就有一定的规则和限制。数据仓库的输入是以很久以前的一个应用程序设计说明书为准的。然而自从组织编写这个应用程序以来，它所依赖的业务条件可能已经经过了多次大的变化。已经针对程序代码做过多次维护，但没有相关文档。另外，这个应用可能没有与其他应用集成的需求。所有的这些脱节之处，在设计和建立数据仓库的时候必须予以考虑。
- 数据仓库反映的是对信息的历史需求，而操作型环境是体现对信息目前的需求。
- 数据仓库着眼于企业的信息化需求，而操作型环境则着眼于精确到秒的企业日常事务需求。
- 必须考虑到新创建的将要传入数据仓库的输出文件的传输。有些情况下这是很容易做到的；在另一些情况下就不那么容易了，尤其是跨操作系统的情况。

还有更多需要考虑的；以上列出的仅仅是当一个程序员着手装载一个数据仓库时所面对的复杂性的开始。

## 3.11 触发数据仓库记录

引起数据仓库的数据载入的基本的业务活动可以称为“事件→快照”事务。在一个事件→快照事务中，某个事件触发了数据的一个快照 (一般是在操作型环境中)，然后被转移到数据

仓库环境中。图3-42象征性地图示了一个“ 事件→快照 ”事务。

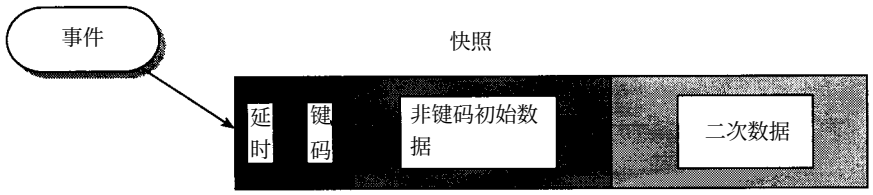


图3-42 数据仓库中的每个快照都是由事件触发的

### 3.11.1 事件

引发快照的业务事件可能是一些重要活动的发生，比如：进行一次销售，货物入库，通一次电话，或者发送一次货物。这类业务事件称作“ 活动－发生 ”事件。在数据仓库中另一类可能触发快照的业务事件是日常的时间推移标志。如一天的结束，一个星期的结束，或一个月的结束。这种业务事件称为“ 时间－发生 ”事件。

业务活动引起的事件是随机的。而时间推移所触发的事件则不是随机的。与时间有关的快照的建立是有规律的并且是可以预知的。

### 3.11.2 快照的构成

产生的且要放入数据仓库中的快照一般包括几个构成部件。一个部件是标志事件发生的时间单元。一般来讲(并不是必然的)，这个时间单元标记了快照产生的时间。另一个部件是用来标识快照的键码。数据仓库快照的第三个部件是与键码相联系的初始的、非键码数据。另外，有一个可选部件，是在形成快照时偶然捕获并被置入快照的二次数据。这些数据往往被称作关系的“ 人工因素 ”。

在最简单的情况下，公司每一个重要的运作活动都将在数据仓库触发一次快照。在这种情况下，公司内已经发生的一些业务活动与被置入数据仓库的快照数目之间是一一对应的。当这种一一对应关系存在的时候，数据仓库就能跟踪与某一主题域有关的所有历史活动。

### 3.11.3 一些例子

每当发生操作型业务活动就产生快照的例子可以在客户文件中找到。每次当客户搬迁、更改电话号码或改变工作的时候，数据仓库就会相应改变，而且有关这个客户历史的一连串记录就会写入数据仓库。一个记录跟踪客户从 1989到1991年的活动。另一个记录跟踪他从1991到1993年的活动。还有一个记录跟踪这个客户从 1993年到现在的活动。这个客户的每次活动都以一个快照的形式载入数据仓库。

另一个说明每次业务活动都导致在数据仓库中产生一个快照的例子是在保险公司中保险金的支付。假设保险金是每半年支付一次的，那么每隔 6个月就会在数据仓库中创建一个快照记录，用来描述保险金的支付情况，包括所支付的时间、金额等。

当数据量不是太大，数据稳定(不是经常变化)，并且需要详细历史记录时，数据仓库可以通过存储已发生的每次活动的详细情况来跟踪业务事件。

## 3.12 简要记录

在很多情况下，数据仓库中的数据并不满足这些标准。有时数据量是巨大的。有时数据的内容经常发生变化，而且有时商业上并不要求特别详细的历史记录。当上述一种或多种情况出现时，可以建立另一种不同的数据仓库记录。这种记录可以被称为聚集记录或简要记录。它是把各个不同操作型数据的详细信息聚集在一个记录中而形成的。一个简要记录的创建以聚集的形式代替了许多条操作型记录。

简要记录可以象单个活动记录那样用来表示数据快照。二者之间的区别是：数据仓库中的单个活动记录代表了一个单一的事件，而简要记录则代表的是多个事件。

如同单个活动记录一样，简要记录也是由某些已经发生的事件触发时，这些事件或者是业务活动，或者是正常时间推移的标记。图 3-43说明了一个事件是怎样导致一个简要记录的创建的。

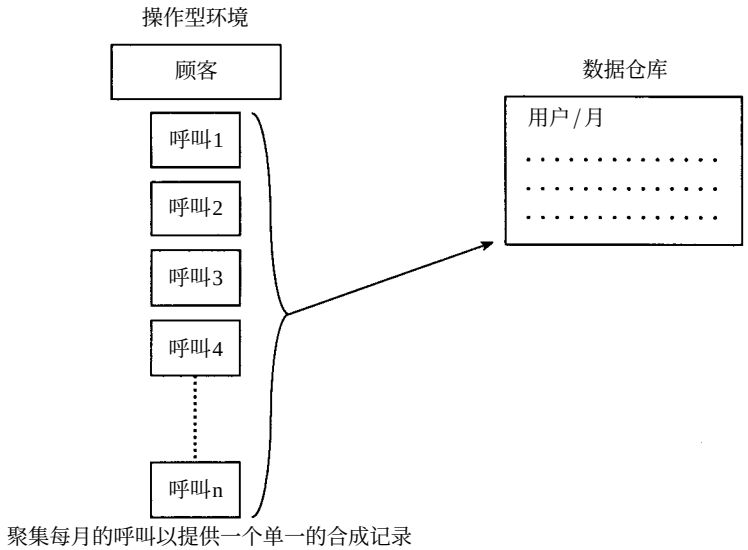


图3-43 从一系列详细记录创建一个简要记录

简要记录是由许多详细的记录聚集而创建的。如电话公司可能在月底整理用户在本月所有的电话活动情况，把这些活动聚集在数据仓库中的一个客户记录中。这样，一个单一的记录就被创建了，它所反映的是客户在一个月之内的所有活动情况。或者银行把一个顾客全月的活动收集起来创建一条聚集的数据仓库记录，这条记录记载了客户这个月内的所有银行活动。

操作型数据聚集形成一条数据仓库记录的过程可以采取多种形式，如：

- 可以对操作数据的取值进行概括。
- 可以对操作数据单元进行计数，以便获得单元的总数。
- 可以对数据单元进行一些处理，以找出最高值、最低值、平均值等。
- 可以捕获第一个和最后一个事件。
- 落入几个参数限界之内的某些类型的数据可以被量度。
- 在一段时间内有效的数据可以被捕获。

■ 最老的数据和最新的数据可以被捕获。

有多少种算法和想象力，就有多少种方式可以表现操作型数据的聚集。

建立简要记录的另一个值得注意的好处是为用户的访问和分析提供了一种紧凑的方便的数据组织形式。采用这种方式，终端用户会感到很方便。因为通过把许多记录聚集为一个记录，用户仅仅一次就可以找到他所需要的数据。这种数据组织方式通过在数据仓库中的数据聚集把用户从大量的劳动和繁重的处理中解放出来。

## 3.13 管理大量数据

有时候数据仓库中需要进行管理的大量数据是一个重要问题。建立简要记录是大量数据管理的一种有效技术。在把操作型环境中的详细记录转入数据仓库中简要记录的过程中，数据量的降低是显著的。一般通过建立简要记录可以使数据量降低 2～3个数量级。由于这种可能性，创建简要记录是每一个数据体系结构设计人员手中很强有力的一种技术。事实上，与其他设计或数据管理技术相比较，要想在数据仓库中有效地管理大量数据，那么建立简要记录应该是数据仓库体系结构设计者应该考虑的首选技术和最强有力的技术。

然而，采用这种方式也有其不足之处。当采用简要记录方式的时候，必须清楚的是这样将会失去数据仓库的一些能力或功能。首先，只要实现数据的聚集，信息的详细程度就会降低。但有时，详细程度的降低不一定是坏事。这时的设计者必须能够保证详细程度的降低对于利用该数据仓库进行决策支持的分析人员来讲是无关紧要的。数据仓库构造者保证所丢失的细节并不特别重要的第一道防线(最简单有效的)就是重复建立简要记录。这样设计人员就有了很好地控制改变的灵活性。简要记录内容设计的第一遍为第二遍提供依据，依此类推。只要数据仓库开发过程中每一遍走得很小，很快。就不至于在简要记录中忽略对终端用户来讲是重要的某种要求。但是当简要记录的创立和开发的第一遍走得非常大，设计者可能会把他们自己带入危险的境地。此时，由于数据仓库相当大，它的内容不能被仔细改动而导致重要细节的忽略，使设计人员可能使自己陷于难堪的境地。

可以保证重要细节在简要记录的创建过程中不被丢失的第二种方法 (可以和第一种共同使用)是在建立简要记录的同时建立历史细节的备用层，如图 3-44所示。

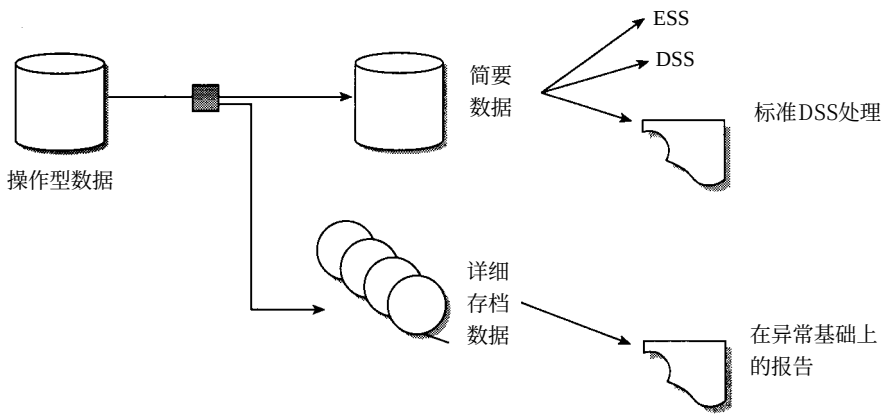


图3-44 另一种典型数据仓库体系结构——所有的细节数据在需要的时候都可以得到，而对于DSS处理的最好性能是常规性能

这种备用的细节并不会被经常用到。它被存储在较慢的便宜的顺序读取的介质上。在任何情况下都不容易访问到，使用起来相当麻烦。但是一旦需要的话，细节确实是存在的。当管理部门确实需要这些信息的时候，它们总可以被找到，尽管需要花费一些时间和金钱。

## 3.14 创建多个简要记录

一个细节可以用来创建多个简要记录。如电话公司的单个电话记录可以用来创建客户简要记录，地区通信量简要记录，线路分析简要记录，等等。

简要记录可以被置入数据仓库或者置入数据仓库所支持的数据集市。当简要记录进入数据仓库时是面向普通应用的，而进入数据集市时则是为了适应部门应用的。

操作型记录聚集成一条简要记录的过程通常是在操作型服务器上完成的。这是因为操作型服务器足以管理大量的数据，而且这些数据任何情况下都驻留于服务器上。通常创建简要记录的过程就是给数据排序和进行合并的过程。一旦建立快照的过程变得十分复杂冗长时，就应该怀疑是否有必要建立快照。

简要记录中写入的元数据记录非常类似于单一活动快照中写入的元数据记录。不同的是，聚集记录的过程成为一条重要的元数据。(从技术上讲，聚集过程产生的记录是“元过程”信息，而不是“元数据”信息。)

## 3.15 从数据仓库环境到操作型环境

操作型环境与数据仓库环境的不同与任何两个环境可能的不同是一样的。从内容、技术、用途、所服务的群体等许多方面来讲都是不同的。二者之间的接口被很好地说明。数据从操作型环境到数据仓库环境要经过一次基本的转换。从操作型环境到数据仓库通常的数据流向如图3-45所示。

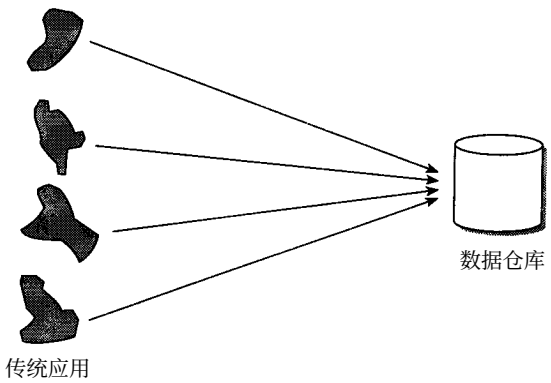


图3-45 在传统应用/数据仓库体系结构设计环境中的数据正常流动

有时会有这样的问题，从数据仓库环境到操作型环境可以传送数据吗？换言之，数据是否可以反向传送？从技术角度讲，当然可以。这种数据传送在技术上是可行的，然而使数据“回流”本身是不正常的。

## 3.16 正常处理

在正常情况下，数据不会从数据仓库流向操作型环境。由于各种原因，如业务活动的顺序，对操作型处理的高性能的需求，数据寿命，操作型处理的较强的面向应用的特性，等等，

使得从操作型环境到数据仓库的数据流动是很自然的，正常的。但是，在一些特别情况下，的确需要数据的“回流”。

## 3.17 数据仓库数据的直接访问

图3-46说明了在那些最简单的动态的数据回流，即由操作型环境对数据仓库环境进行直接的数据访问。在操作型环境中向属于数据仓库的数据提出了访问请求。这个请求被传送到数据仓库中，然后找到所需要的数据，接着再传输到操作型环境中。很明显，从动态的角度来看，传送过程的实现不会是简单的。

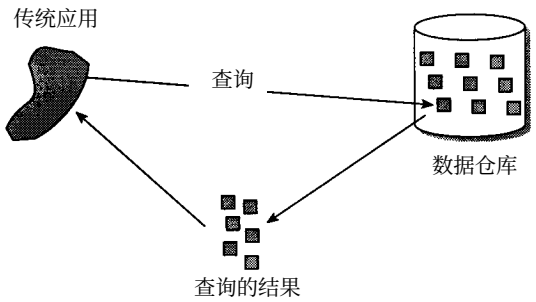


图3-46 一个从传统应用环境对数据仓库的直接查询

在直接访问数据仓库数据的过程中，有一些严格的、不能让步的限制。下面列出了一些这类限制。

- 从响应时间的角度来讲，这个请求必须能够忍受冗长的响应时间。它可能在经过 24个小时后才被响应，这意味着请求数据仓库数据的操作处理并不具有在线特性。
- 所请求的数据量必须是最小量的。数据的传输是以字节计的，而不是兆字节或千兆字节。
- 管理数据仓库所用到的技术必须与管理操作型环境所用到的技术一致，如容量、协议等。
- 从数据仓库取得的准备传输到操作型环境的数据必须不做或做最小的格式化。

这些条件限制了数据从数据仓库到操作型环境的数据传送。很容易明白在数据的直接访问时为什么仅仅有少量的数据回流。

## 3.18 数据仓库数据的间接访问

由于严峻的不可妥协的传输条件，由操作型环境到数据仓库数据的直接访问很少发生。但是，对数据仓库数据的间接访问则是另一回事。事实上，数据仓库的一个有效的应用就是操作型环境对数据仓库的间接数据访问。一些间接访问数据仓库数据的例子可以充分说明这点。

### 3.18.1 航空公司的佣金计算系统

间接使用数据仓库数据的一个例子是在航空公司。为了能够理解这个例子，需要一些航空公司预定和售票方面的知识。

考虑如图3-47所示的航空公司定票事务。旅行社代表客户与航空公司机票预定服务人员交涉。这个客户想购买一张飞机票而旅行社需要知道以下问题：

- 还有座位吗？

- 座位票价是多少？
- 旅行社能获得多少佣金？

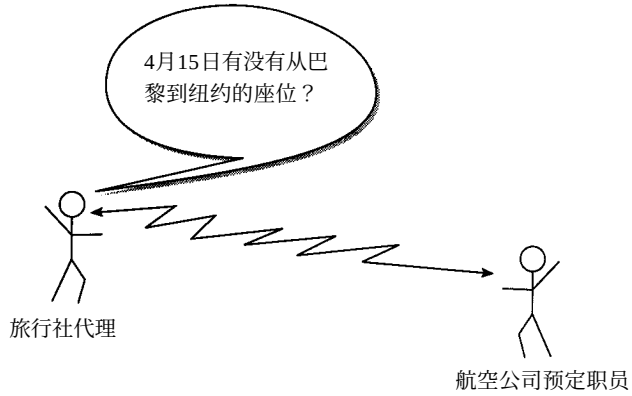


图3-47 一个典型的商业交易

如果航空公司支付太多的佣金，他们将获得旅行社的这笔交易，但是会损失一部分钱。如果佣金太少，这家航空公司可能会失去这笔交易，旅行社将会寻找另外一家支付佣金较多的航空公司完成这笔交易。航空公司有必要在其最佳利益范围内非常仔细地计算所应支付的佣金，因为这直接关系到它的底线。

旅行社代理和航空公司职员之间的交易必须在很短的时间内完成，比如 2～3分钟之内。在这么短的时间内，航空公司职员必须输入完成一系列事务处理，如：

- 是否有剩余座位？
- 座位是否可优先使用？
- 联系哪些航班？
- 是否能联系上？
- 票价多少？
- 佣金多少？

如果航空公司职员(与旅行社代理交流时运行多个事务)的响应时间太长，那么航空公司将会因此而失去交易。因此，出于航空公司的最高利益必须尽量缩短与旅行社对话过程中的响应时间。

佣金的计算成为交易中的重要组成部分。最佳佣金的计算将考虑两个因素 —— 当前的预定情况和历史情况。当前的预定情况提供了目前飞机票的预定情况。而历史情况则提供从前一段时间的定票情况。二者之间可以计算出一个合适的佣金。图 3-48说明了参与计算的元素。

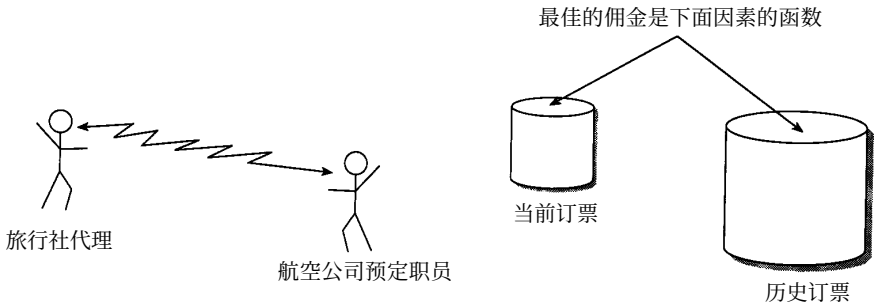


图3-48 最佳的佣金通过比较当前订票和历史订票计算得来

人们希望通过联机方式完成预定和航班历史情况的计算。但是所需处理的数据量很大，如果采用联机方式，势必会影响到响应时间。相反，在有足够机器资源的地方，采用脱机方式完成佣金计算和航班历史分析。图 3-49说明了采用动态的脱机佣金计算方法。

脱机计算和分析定期进行，且需要建立一个小的易于访问的航班状态表。当航空公司职员与旅行社代理交互时，很容易查阅现有定票情况和航班状态表。结果，二者之间的对话很迅速，且很好地利用了数据仓库的数据。

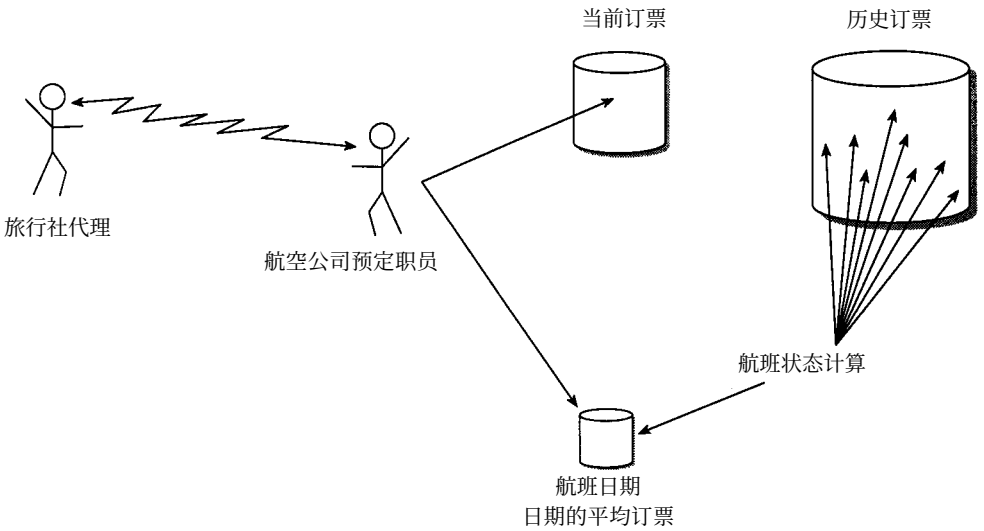


图3-49 航班状态文件通过读取历史数据周期地创建，它是航空公司代理快速获得当前订票以及与历史平均订票情况比较的手段

### 3.18.2 零售个性化系统

在操作型环境中，间接使用数据仓库数据的另一个例子是零售个性化系统。在这样的系统中，客户阅读到由零售商编制的目录或宣传广告后促使他有了购买的念头，或者至少想查询一下目录，结果是给零售商打电话。图 3-50表明了零售商和客户之间的对话。

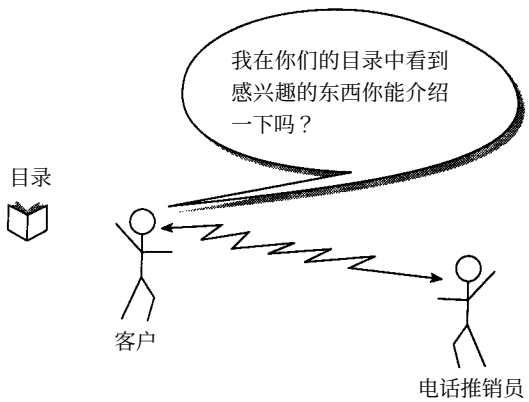


图3-50 顾客看了邮件中的目录，希望得到更多信息

这种对话可能持续5～8分钟。这段时间内零售商的代表需要做一系列的事情 —— 确定客户，记下所需的定货信息等。响应时间必须简短，否则客户将失去兴趣。

当客户定货或咨询情况时，零售商代表查出一些与此有关的其他信息，如：

■ 客户上次购物的时间。

■ 上次购物的类型。

■ 客户所属的市场地段。

与客户对话的过程中，销售代表说出以下一些事情：

“ 我记得我们曾在二月份通过话 ”

“ 你购买的兰色运动衫怎么样？ ”

“ 你的那条裤子的问题解决了吗？ ”

一句话，销售代表有必要使交谈进行得很有人情味。这样，将会更加激起客户的购买欲望。

另外，销售服务人员应该拥有市场地段信息，如：

■ 男/女

■ 专业/其他用品市场

■ 城市/乡村市场

■ 儿童用品市场

- 年龄
- 性别

■ 体育用品市场

- 鱼具
- 猎具
- 沙滩用具

因为对话可以进行得很个性化。而且有可用的客户所属的市场地段信息。因此，当客户打入电话时，销售代表能够进行针对性的提问，如：

“ 你知道我们在泳装方面还有未公布的产品吗？ ”

“ 我们刚刚进一批意大利太阳镜，我想你可能有兴趣。”

“ 天气预报这是打野鸭的寒冬，我们有一种特制的长筒靴。”

客户已经完全投入了电话对话中，个性化的电话和关于客户对什么商品感兴趣的知识使得销售商在不增加资本投入、不增加广告量的情况下增加收入。

这种个性化的电话对话正是通过对数据仓库的间接访问而完成的。图 3-51表明如何达到这种个性化的动态过程。

后台(即数据仓库环境中)有一个分析程序在不断读入和分析客户的记录。这个分析程序通过一种复杂的方法扫描，分析客户的历史记录。它定时地提供给操作型环境一个包括下面内容的文件：

■ 上次购物的日期。

■ 上次购物的类型。

■ 市场分析/市场地段信息。

当客户打入电话时，联机准备好的文件就等待由零售销售代表使用。

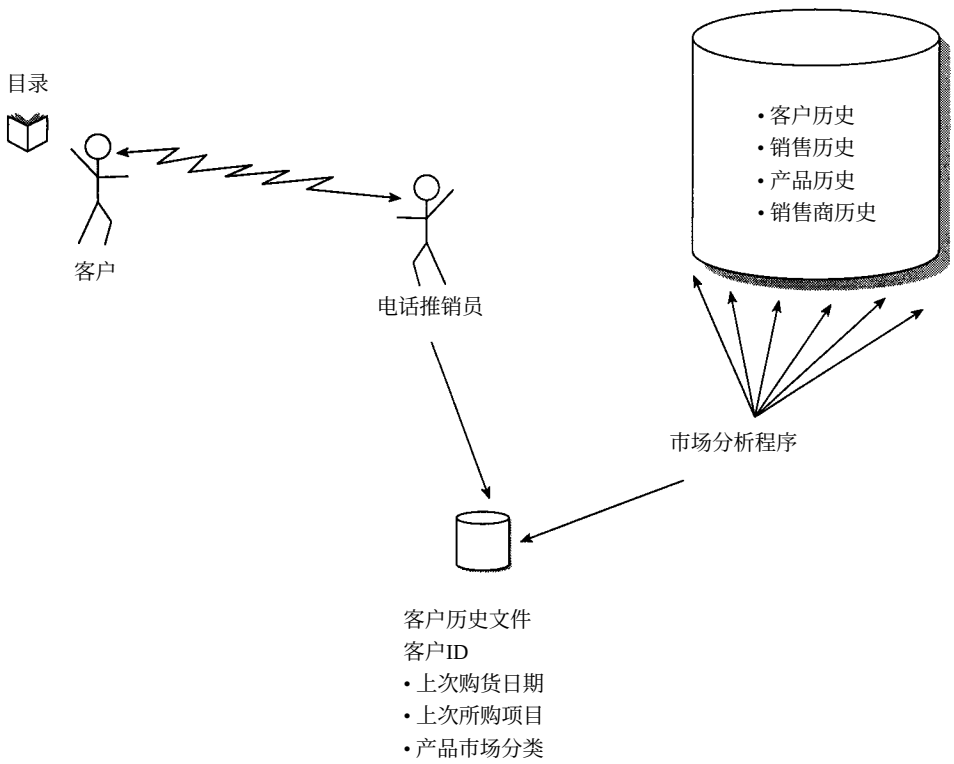


图3-51 客户历史被电话推销员立即利用

### 3.18.3 信用审核

间接利用数据仓库的另一个例子是银行金融领域的信用审核过程。它是用来确定一个客户是否有资格获得贷款的一种审核过程。图 3-52说明了一个以交互模式进行的信用审核案例。

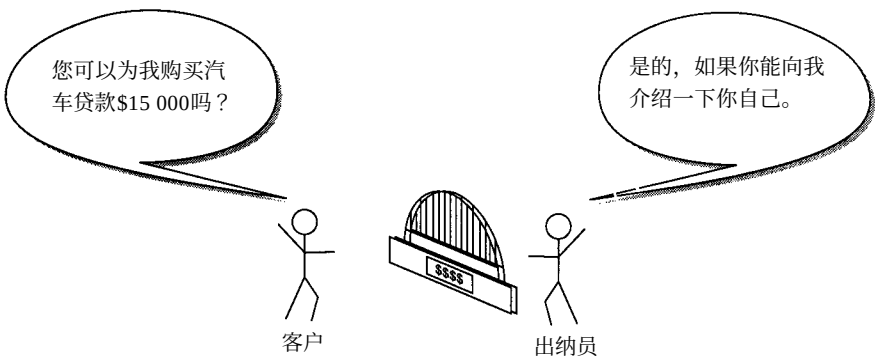


图3-52 联机贷款过程

在图3-52中，客户来到出纳员的窗口要求贷款。出纳员询问用户的一些基本信息，然后

决定是否提供贷款。这种交互过程也发生在一个很短的时间内——大概5～10分钟。

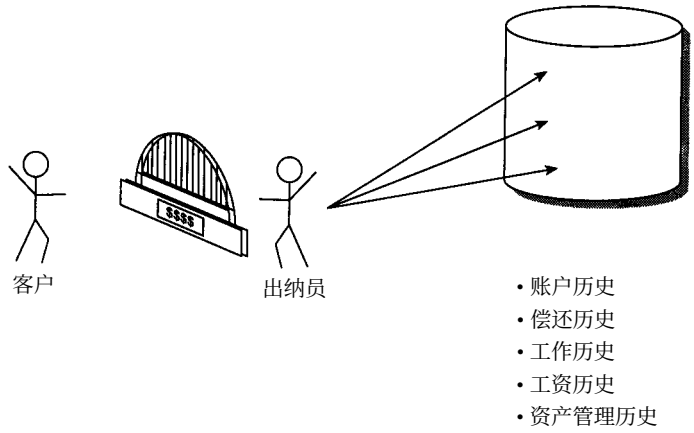


图3-53 同意贷款前先检查客户历史

为了确定是否应该提供贷款，需要进行一些处理，如图 3-53所示。

贷款请求首先经过一个简单审核处理。如果贷款金额较小，而且贷款人有一个稳定的经济背景，那么就可以决定给他提供贷款，而不必再加以审核。然而，如果贷款金额较大，或者贷款人没有稳定的经济来源，那么就需要继续审查。

后台审核程序依赖数据仓库。事实上，这种审核是综合的，需要对客户的各个方面进行调查。例如：

- 偿还历史。
- 私有财产。
- 财务管理。
- 净值。
- 全部收入。
- 全部开销。
- 其他的无形资产。

这种大范围的背景检测过程需要大量的历史数据。完成贷款审核中的这个处理过程需要花几分钟的时间。

为了在最短的时间内满足尽可能多的客户要求，需要编写一个分析程序。图 3-54说明了这个分析程序是如何与信用审核过程中其他部件协调工作的。分析程序定期启动，它为操作型环境提供了一个可用文件。除了其他数据以外，这个文件应包括以下信息：

- 客户识别信息。
- 核准信贷限制。
- 特殊的核准限制。

这样，当客户想申请获得贷款时，出纳员利用高性能的联机方式就可以给予 (或不给予)客户贷款请求。仅当客户贷款金额超过预先核准的限额时，才需要经过贷款官员的进一步审核。

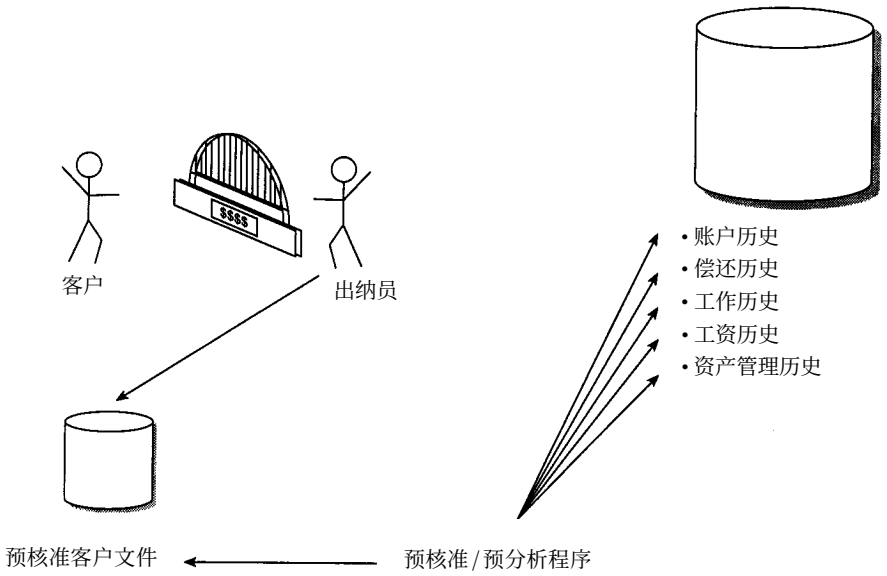


图3-54 预核准客房信用文件立即被银行出纳员访问

## 3.19 数据仓库数据的间接利用

对于间接利用数据仓库来讲，还有一种正在出现的模式，如图 3-55所示。

由一个程序对数据仓库进行定期的分析，以检验相关的特征和标准。这种分析过程将在联机环境中产生一个小文件，其内容包括了有关企业业务方面的简明信息。这个文件被有效快速地利用，以满足操作型环境中其他处理的需要。

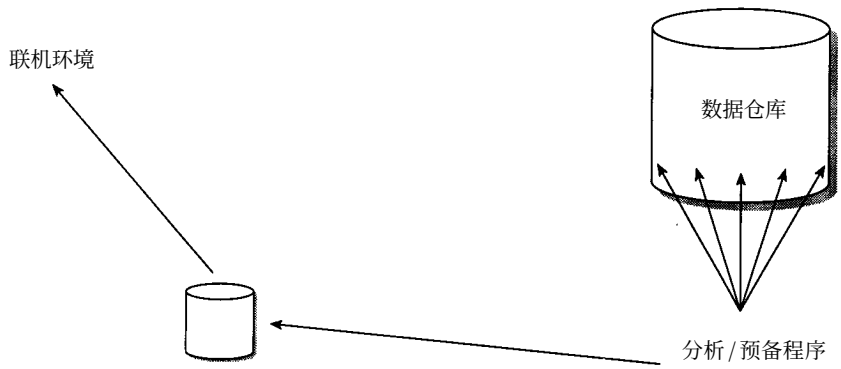


图3-55 数据仓库数据被联机操作型环境间接应用的方法

下面是间接使用数据仓库数据时应考虑的因素。

■ 分析程序：

- 拥有许多人工智能的特征。
- 可以运行在任何可用的数据仓库中。
- 在后台运行，这样处理时间就不是一个问题 (至少不是一个大问题)。

- 程序的运行与数据仓库发生变化的速度一致。

■ 周期性刷新
- 不是经常进行。
- 采用一种替代模式操作。
- 从支持数据仓库的技术传送数据到支持操作型环境的技术。

■ 联机预分析数据文件
- 每个数据单元仅仅包括少量的数据。
- 总体上可以包含大量的数据(因为可以有很多的数据单元)。
- 准确地包含了联机处理人员所需要的东西。
- 不被修改，但是定期全部刷新。
- 是联机高性能环境的一部分。
- 访问效率高。
- 可以访问单个数据单元，而不是以块的形式访问数据。

## 3.20 星型连接

数据模型作为一种数据仓库的设计基础，在实际应用中还存在许多缺点。考虑图 3-56所示的简单数据模型。

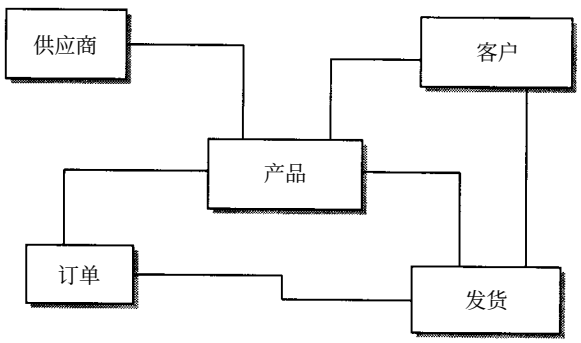


图3-56 一个简单的二维模型给人的印象是所有的实体都是等同的

图3-56中所示的数据模型中有四个相互关联的简单实体。如果数据库设计只需要考虑数据模型的话，可以推断所有的实体都是平等关系。换言之，从数据模型的设计角度来看，所有的实体之间的关系是对等的。仅仅从数据模型的角度来着手设计数据仓库会产生一种“平面”效应。实际上，由于种种原因，数据仓库的实体绝不会是相互对等的。一些实体，要求有它们自己的特别处理。为了明确为什么从数据模型的角度看一个组织中的数据和关系会发生失真，根据在数据仓库中建立实体时将载入数据实体的数据量，我们来考虑数据仓库中数据的一种三维透视。图 3-57表明了这种三维透视。代表供应商、客户、产品、发货的实体被稀疏地载入，而代表订单的实体则大量地载入。将会有大量的数据载入代表订单实体的表中，而在代表别的实体的表中载入的数据量则相对较少。由于大量的数据要载入订单实体，因此需要一种不同的设计处理方式。

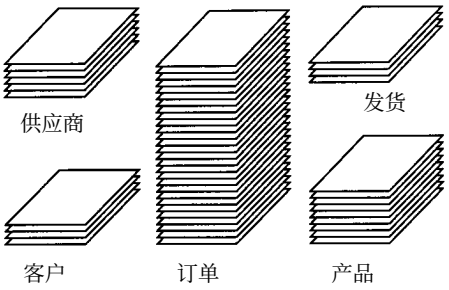


图3-57 实体的三维透视表现出实体并不是平等的，有些实体包含的数据远远超过其他实体

用来管理数据仓库中载入某个实体的大量数据的设计结构被称为“星型连接”。图3-58给出星型连接的一个简单例子。“订单”位于星型连接的中央。它是被大量载入数据的实体。在其周围分别是“产品”、“客户”、“供应商”和“发货”实体。这些实体仅仅会产生不大的数据量。星型连接中央的“订单”被称作是“事实表”，而其周围的其他实体——“产品”、“客户”、“供应商”和“发货”则被称为“维表”。事实表包含了“订单”独有的标识数据，也包含了订单本身的独有数据。事实表还包含了指向其周围的表——维表的外键。如果非外键的信息经常被事实表使用，那么星型连接内的非外键信息将会伴随外键的关系共同存在。例如，如果“产品”的描述将被“订单”处理过程经常用到的话，那么这个描述将会与产品号一起存储在事实表中。

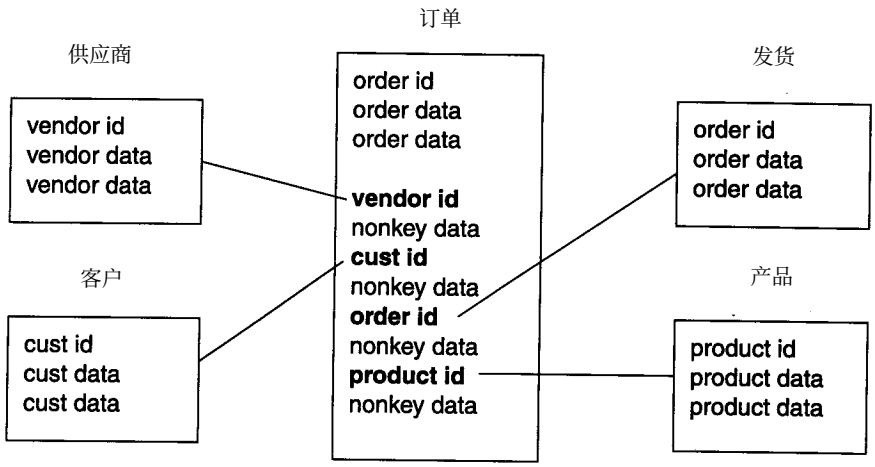


图3-58 一个简单的星型连接，其中“订单”实体会由许多数据载入，其他实体与数据预连接

可以有任意多个外键与维表相关。当有必要检查外键数据与事实表中的数据时，就创建一个外键关系。

创建和使用星型连接的一个有趣的方面是，在很多情况下，文本数据与数值数据是分离开的。考虑图3-59所示的图表。文本数据常出现在维表中，数值数据常出现在事实表中，这种划分似乎在所有情况都会发生。

创建和使用星型连接的好处是可以为决策支持系统的处理优化数据。通过数据预连接和建立有选择的数据冗余，设计者为访问和分析过程大大简化了数据，这正是数据仓库所需要的。应该

注意，如果不是在决策支持系统数据仓库环境中使用星型连接，则会有很多的缺点。在决策支持系统数据仓库环境以外，常有数据更新，而且数据关系的管理要在秒的一级上进行。在这种情况下星型连接在创建和维护上就是很麻烦的数据结构。但是由于数据仓库是一个装载——访问环境，它包括很多历史数据，且有大量的数据要管理，因此，星型连接的数据结构是十分理想的。

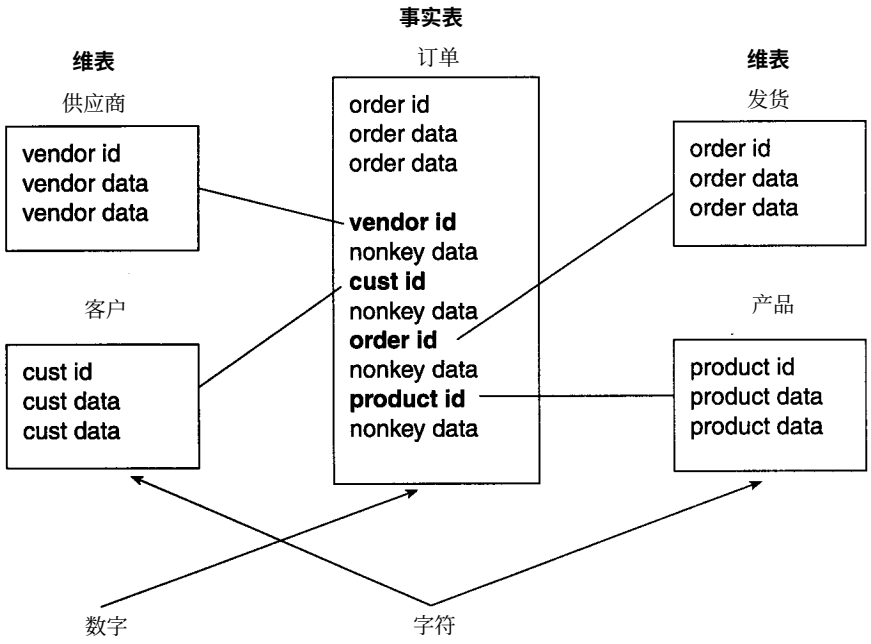


图3-59 通常事实表格放置数字数据和外键，而维表则放置字符数据

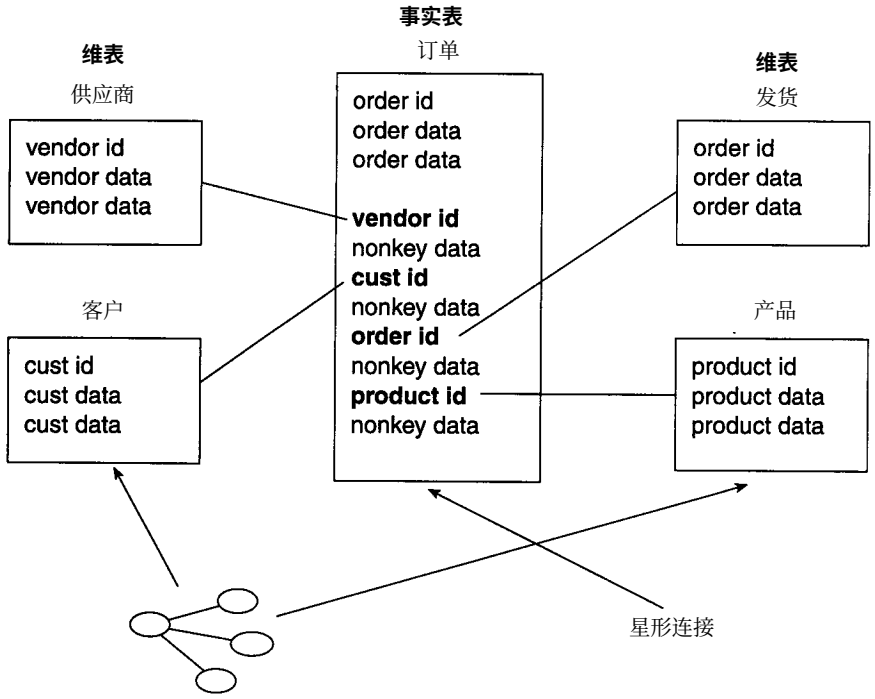


图3-60 传统数据模型应用于维表即数据不多的实体，星形连接应用于事实表(即数据量大的实体)

是不是星型连接结构的存在意味着数据模型不是设计数据仓库的基础了呢？完全不是！数据模型对于大多数数据仓库环境的设计来讲，仍然是非常有用的一种结构。然而，星型连接有它本身的恰当位置。图 3-60说明了数据仓库决策支持系统的设计中星型连接和数据模型是怎样配合起来使用的。星型连接应用于设计数据仓库中很大的实体，而数据模型则应用于数据仓库中较小的实体。

## 3.21 小结

数据仓库的设计始于数据模型。企业数据模型用于操作型环境的设计。企业数据模型的一种变型用于数据仓库的设计。数据仓库以反复开发的形式建立。对于数据仓库的需求是不可能预先知道的。数据仓库的构造是在与传统操作型系统完全不同的开发生命周期中进行的。

数据仓库开发者面临的基本问题是管理大量数据。为此，数据的粒度和分割是数据库设计的两个最重要问题。然而，也存在许多其他物理设计问题，其中大多数都与数据访问的效率有关。