第25章 对象/关系数据库

25.1 引言

在本书第6版出版后五年左右的时间里，一些厂商已经发布了“对象/关系”DBMS产品（在市场上被称为是通用服务器）。例如DB2的通用数据库版，Informix动态服务器的通用数据选件以及Oracle 8i的通用服务器、数据库服务器或企业服务器（这三个名字都在使用）。在这些产品中广泛的思想是产品能够同时支持对象和关系；换句话说，这些产品试图将这两种技术结合起来。

作者认为这种结合应该以关系模型作为坚实的基础（毕竟如本书第二部分中所谈到的，关系模型是现代大多数数据库技术的基础）。所以，我们希望能够在关系系统发展过程中 ⊖ 加入对象系统的新特性（我们当然不希望完全放弃整个关系系统，也不希望将这两种系统分开处理）。其他许多作者也支持这一观点，包括《第三代数据库系统宣言》[25.34]的作者；他们认为第三代DBMS必须包容第二代DBMS。在这里第一代DBMS是指关系以前的数据库系统，第二代DBMS是指SQL系统，而第三代DBMS是指未来的系统。然而，某些对象系统的作者并不支持这一观点。这里引用一段较为代表性的话：

> 在计算机科学中已经产生了许多代数据管理方式，从开始的索引文件到后来的网状和层次DBMS……，再到最近的关系DBMS……。目前我们正处在新一代数据库系统的开始点……，这种系统提供对象管理并支持更复杂的数据[25.4]。

在这里作者明确地认为：就像关系系统代替更早的层次和网状系统那样，对象系统也将代替关系系统。

我们不同意这一观点的理由在于关系是完全不同的[25.13]，它并不针对特定的环境；而老的层次和网状系统却是基于特定环境的，它们也许能解决某些重要的问题，但却不具备牢固的理论基础。然而不幸的是，早期关系系统的支持者们——包括作者本人在内——对关系和关系以前的系统的优缺点比较存在着很大的分歧；这些争论在当时是必要的，但是实际上却起了不良的后果，使人觉得关系和关系以前的DBMS在本质上是同一事物。这一错误的思想支持了以上引文中的观点——关系系统到对象系统与层次和网状系统到关系系统相类似。

对象系统到底是什么？它们是基于特定环境的吗？《面向对象数据库系统宣言》[24.1]中对此做了很好的说明：“关于系统的详细规范，我们采用一种达尔文方式：我们希望通过构建一系列实验原型，自然而然生成适当的对象模型”。换句话说，作者认为我们不需要预定义模型，而是应该先编写代码并构建系统来看看结果到底如何！

在下文中，我们将始终贯彻我们的观点，即通过在关系系统中加入对象技术的优秀特性

⊖ 注意我们感兴趣的是发展而非完全的变革。相反，在关于ODMG的[24.13]中认为：“对象DBMS是数据库世界的一场革命而非进化式的发展”。我们不认为数据库市场已经为这场革命做好了准备，我们也不认为应该有这样一场革命——这也是第三次宣言[3/3]认为其本质上是进化式发展而非完全变革的原因。

来增强其性能。再次声明，毕竟关系技术已经研究了三十多年，我们绝不希望完全放弃这一技术。

在第24章中我们已经论证——可见[25.23]中的相关叙述——面向对象只包含一种好的思想，那就是适当的数据类型支持（或者说是两种好的思想，如果将类型继承单独算的话）。所以我们遇到的问题是：如何将适当数据类型支持这一特性融入到关系模型中？当然，答案很简单，这种支持已经以域的形式存在了（我们仍倾向于将其称为类型）。换句话说，为了加入对象功能我们并不需要在关系模型中增加任何东西，除了将其完整地实现；而现在大多数系统在实现上都不是很成功[⊖]。

所以，我们认为一个真正支持域的关系系统应该能够处理所有那些所谓的对象系统才能够处理而关系系统处理不了的数据类型：时间序列数据、生物数据、金融数据、工程设计数据、办公自动化数据，等等。同时，我们还认为一个真正的“对象/关系”系统首先应该是一个关系系统——也就是说，支持关系模型及其相关的所有内容。应该鼓励DBMS厂商去开发那些真正需要开发的东西，即在他们的系统中包含对类型和域的适当支持。事实上，可以论证对象系统之所以吸引人的主要原因就在于SQL厂商没能充分支持关系模型；因此这绝不能作为放弃关系系统的原因！

现在让我们来完成第24章中尚未完成的例子，看看对于矩形问题如何提出一个很好的关系解决方案。对此问题的解决首先需要定义一个矩形类型RECT：

```
TYPE RECT POSSREP ( X1 RATIONAL, Y1 RATIONAL,
                    X2 RATIONAL, Y2 RATIONAL ) ... ;
```

我们假设矩形在物理上通过某种能有效支持空间数据的方式存储——如四叉树(quadtree)或R树等[25.27]。

我们同时定义一个用来测试两个给定矩形是否重叠的操作符：

```
OPERATOR OVERLAP ( R1 RECT, R2 RECT )
   RETURNS ( BOOLEAN ) ;
   RETURN ( THE_X1(R1) ≤ THE_X2(R2) AND
            THE_Y1(R1) ≤ THE_Y2(R2) AND
            THE_X2(R1) ≥ THE_X1(R2) AND
            THE_Y2(R1) ≥ THE_Y1(R2) ) ;
END OPERATOR ;
```

此操作符在实现上采用了重叠测试的有效方式（详见第24章中的相关叙述），同时采用了有效的存储结构（四叉树或R树等）。

现在用户可以来创建一个基表，表中包含某个以RECT作为类型的属性：

```
VAR RECTANGLE RELATION { R RECT, ... } KEY { R } ;
```

这样，对于查询“得到所有与单位正方形相重叠的矩形”来说就很简单了：

```
RECTANGLE WHERE OVERLAP ( R, RECT ( 0.0, 0.0, 1.0, 1.0 ) )
```

这一解决方案克服了第24章中讨论的所有缺陷。

本章的其余部分组织如下：25.2节和25.3节分别讨论两个根本性错误（至少其中一个错误出现在市场上几乎每一个对象/关系产品中）。25.4节考虑了对象/关系系统某些实现上的问题，25.5节描述了一个真正对象/关系系统所具有的好处（也就是说，绝不会犯那两个根本性错误）。最后，25.6节给出本章的小结。

⊖ 具体来说，那些系统使大家认为关系系统只能支持有限的简单数据类型。下列这些说法都是很典型的：“关系数据库系统只支持有限的数据类型”[25.25]；“关系DBMS只支持……它内置的类型”[24.38]；“对象/关系数据模型通过提供更丰富的类型系统来扩充关系数据模型”[17.61]；等等。

25.2 第一个根本性错误

首先引用参考文献[3.3]中的一段话：

在具体考虑如何将对象和关系结合的问题之前，首先需要解决一个关键性的问题，即：

对象世界中的概念“对象类”在关系世界中对等的概念到底是什么？

这个问题之所以非常重要，其原因就在于对象类是对象世界中所有概念的基础——所有其他的对象概念或多或少都依赖于它。对于这个问题通常有两个等式作为其答案的候选：

- 域 = 对象类
- 关系变量 = 对象类

我们现在来证明，第一个等式是正确的，而第二个是错误的。

事实上，既然对象类和域实质上都是类型，第一个等式显然是正确的；而给定关系变量是变量，类是类型，则第二个等式显然是错误的（变量和类型绝非同一事物）。正是基于这一原因，在《第三次宣言》[3.3]中坚持认为关系变量不是域。然而，许多人和一些产品却在事实上支持了第二个等式——我们认为这是根本性错误（或如我们前面所说的，第一个根本性错误）。仔细讨论这一问题是非常必要的。注意：本节中的大部分内容直接引自参考文献[3.3]。

为什么会有人犯这样的错误呢？考虑下列简单的类定义。其中的对象语言是虚构的，并且我们有意不和24.3节中的表达相一致：

```
CREATE OBJECT CLASS EMP
     ( EMP#       CHAR(5),
       ENAME      CHAR(20),
       SAL        NUMERIC,
       HOBBY      CHAR(20),
       WORKS_FOR  CHAR(20) ) ... ;
```

（这里的EMP#、ENAME等都是公共实例变量）。现在来考虑一下SQL语句对“基表”的定义：

```
CREATE TABLE EMP
     ( EMP#       CHAR(5),
       ENAME      CHAR(20),
       SAL        NUMERIC,
       HOBBY      CHAR(20),
       WORKS_FOR  CHAR(20) ) ... ;
```

这两个定义看起来的确很类似，将两者等同起来的思想也的确很诱人。某些系统，包括一些商业产品，事实上就是这么做的。现在让我们进一步思考这一问题，或更准确地说，在以上CREATE TABLE声明的基础上再增加一些扩充以使其更为“对象化”。注意：以下的讨论基于某一特定的商用产品；其实就是基于此产品文档中的一个例子。我们并不想指出这个产品的名称，因为批评或表扬某一产品并不是本书的目的。我们所提出的批评针对所有支持等式“关系变量=对象类”的系统。

第一个扩充为：允许加入复合（即元组值）属性；也就是说，我们允许属性的值可以是其他表中的元组（或就是本表中的元组）。例如，将原先的CREATE TABLE声明替换成如下声明的集合（参见图25-1）：

解释：表EMP中的属性HOBBY被定义为ACTIVITY类型，而ACTIVITY是含有两个属性

NAME和TEAM的表，其中TEAM给出NAME所指定的运动类型中每队运动员的数目；例如，一个可能的元组为（Soccer,11）。这样，每个HOBBY值实际上由一对值（NAME值和TEAM值）构成，而这对值目前以关系变量 ACTIVITY中的元组值形式出现。注意：我们已经触犯了《第三次宣言》中关系变量不是域的声明 —— 属性HOBBY的“域”被定义为关系变量ACTIVITY。在本节中还将讨论这一问题。

```
CREATE TABLE EMP
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBY       ACTIVITY,
       WORKS_FOR   COMPANY ) ;

CREATE TABLE ACTIVITY
     ( NAME        CHAR(20),
       TEAM        INTEGER ) ;

CREATE TABLE COMPANY
     ( NAME        CHAR(20),
       LOCATION    CITYSTATE ) ;

CREATE TABLE CITYSTATE
     ( CITY        CHAR(20),
       STATE       CHAR(2) ) ;
```

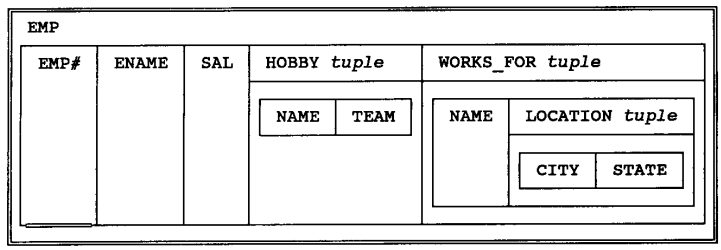

图25-1 属性包含元组（的指针）—— 我们所反对的

类似地，关系变量EMP中的属性WORKS_FOR声明为类型COMPANY，而COMPANY也是一个包含两个属性的关系变量，其中一个属性的类型为 CITYSTATE，而CITYSTATE又是一个包含两个属性的关系变量。换句话说，关系变量 ACTIVITY、COMPANY和CITYSTATE都既被认为是类型（或域），又被认为是关系变量。当然，关系变量EMP本身也是如此。

第一种扩充大致上与允许对象包含对象类似，从而支持了包含层次的概念（见第 24章）。

我们将第一种扩充的特征表述为“属性包含元组”，而这正是由等式“关系变量=对象类”本身所带来的特征（见参考文献[24.33]）。然而，更精确地说，这一特征应该表述为“属性包含元组的指针”—— 我们在随后将分析这一问题（所以在图 25-1中我们应该将三个“元组”替换成“元组的指针”）。

第二个扩充为：允许关系值属性；也就是说，属性值可以是其他表（或就是同一个表）中元组的集合。例如，假设雇员可以有任意个而不只是一个爱好（参见图 25-2）：

```
CREATE TABLE EMP
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBIES     SET OF ( ACTIVITY ),
       WORKS_FOR   COMPANY ) ;
```

解释：现在关系变量 EMP中任意元组的 HOBBY值可以是空值或一系列关系变量ACTIVITY中的元组。第二种扩充大致与允许对象包含“集合”对象类似：包含层次的更为

复杂的版本。注意：在我们所基于的特定产品中，集合对象可以是序列、无序单元组或集合本身。

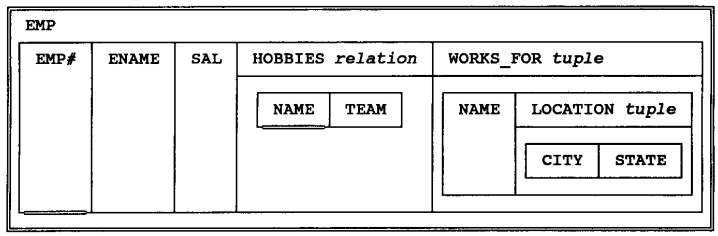

图25-2 属性包含元组（指针）的集合——我们所反对的

第三个扩充为：允许关系变量有相应的方法。例如：

```
CREATE TABLE EMP
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBIES     SET OF ( ACTIVITY ),
       WORKS_FOR   COMPANY )
METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

解释：方法RETIREMENT_BENEFITS将某个EMP元组作为参数，并产生类型为NUMERIC的结果值。

最后一个扩充为：允许子类。例如（参见图25-3）：

```
CREATE TABLE PERSON
     ( SS#         CHAR(9),
       BIRTHDATE   DATE,
       ADDRESS     CHAR(50) ) ;

CREATE TABLE EMP
    AS SUBCLASS OF PERSON
     ( EMP#        CHAR(5),
       ENAME       CHAR(20),
       SAL         NUMERIC,
       HOBBIES     SET OF ( ACTIVITY ),
       WORKS_FOR   COMPANY )
METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

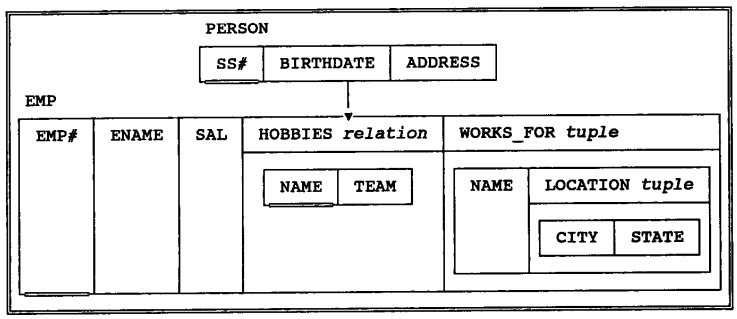

图25-3 将关系变量作为子类或超类——我们所反对的

解释：现在关系变量EMP有三个从关系变量PERSON中继承的附加属性（SS#、BIRTHDATE和ADDRESS）。如果关系变量PERSON有方法，也将继承过来。注意：这里的PERSON和EMP有时分别被称为超表和子表。参考文献[13.12]和附录B中有对这些概念的进一步讨论和批评。

除了以上关于定义的扩展外，当然还需要某些处理上的扩展，例如：

- 路径表达式——例如，EMP.WORKS_FOR.LOCATION.STATE。注意：这一表达式返回的可能是标量、元组或关系。同时还需注意：在上述扩充的后两种情况下，元组或关系的成员也可能是元组或关系；例如，表达式 EMP.HOBBIES.NAME返回的即为关系。此外，这里的路径表达式下降到控制层次，而第 24章中的路径表达式则是上升的。
- 元组和关系的值（可能有嵌套）——例如：

```
( 'E001', 'Smith', $50000,
        ( ( 'Soccer', 11 ), ( 'Baseball', 9 ) ),
                            ( 'IBM', ( 'San Jose', 'CA' ) ) )
```

(并不代表实际语法)。

- 关系比较运算符——例如，SUBSET、SUBSETEQ等（这些具体的操作符来自我们的讨论所基于的某特定产品。在该产品中， SUBSET真正意味着“适当的子集”，而SUBSETEQ只意味着“真子集”！）。
- 遍历类的层次结构的操作符。注意：这里仍需非常小心。在有些情况下，查找 PERSON信息及相关 EMP信息的查询结果不再是关系——这就意味着关系最基本的封闭属性被打破，这将带来一系列很麻烦的问题（关于这一问题，参考文献 [25.31]——其将这种返回结果称为“参差不齐的返回”——仅仅提出：“客户端程序必须准备好处理这种复杂的参差不齐的返回结果”！）。
- 在诸如SELECT和WHERE子句中调用方法的能力。
- 访问属性值为元组或关系的成员的能力。

当我们考虑等式“关系变量=类”如何具体实现时，马上会想到这么一大堆问题。那么这一等式到底存在什么问题呢？

首先我们注意到：关系变量是变量，而类是类型，它们怎么可能相同呢？这一点从逻辑上就足以打消“关系变量=类”的想法。然而，为了更进一步说明问题，我们列出了以下供参考的观点：

- 等式“关系变量=类”意味着进一步的等式“元组=对象”和“属性=（公共）实例变量”。这样的话，一方面一个真正的对象类——至少是标量或“封装的”对象类——必定拥有方法，并且没有公共实例变量，而另一方面关系变量“对象类”却一定有公共实例变量，并且只在某些情况下才有方法（这绝对不满足“封装性”）。还是那句话，这两个概念怎么能够等同呢？
- 属性定义间有着巨大的差别，如“ SAL NUMERIC ”和“ WORKS_FOR COMPANY ”。NUMERIC是真正的数据类型（真正原始的域）；它在属性 SAL的值上设置了与时间无关的约束。相反，COMPANY并不是真正的数据类型，它在属性 WORKS_FOR上设置的约束是时间相关的（其显然依赖于关系变量 COMPANY的当前值）。事实上，正如前面所指出的，关系变量与域的区别——或者使用对象中的术语，集合与类的区别——在这里被混淆了。
- 如我们所看到的，元组“对象”可以包含其他的“对象”；例如， EMP“对象”显然包含COMPANY“对象”，然而，事实上它们包含的是“被包含对象”的指针，用户必须非常清楚这一点。例如，假设用户更新了某条特定的 COMPANY元组（参见图 25-1），所有包含此COMPANY元组的EMP元组对此马上可见。注意：我们并非认为这样不好，只是这样的话，用户就必须了解内情，即了解图 25-1中所示“模型”是不正确的——

EMP元组并不真正包含COMPANY元组，而只包含指向COMPANY元组的指针。

对于这一点还需做以下补充：

a. 我们能不能插入一条 EMP元组、并将其包含的 COMPANY元组值设为目前在COMPANY关系变量中尚未存在的元组？如果答案是能，那么对于 INSERT操作没有任何限制，这样把WORKS_FOR定义成COMPANY就没有任何意义。如果答案是不能，那么对于INSERT操作将会产生不必要的麻烦——用户不得不给出整个公司的信息而不仅仅是公司名称（即外码值）。此外，给出整个公司信息意味着告诉系统一些其本身已经知道的内容，并且在最坏情况下，本该成功完成的 INSERT操作由于用户在给出完整信息时的小错误而失败。

b. 假设我们希望在公司上加入 ON DELETE RESTRICT规则（也就是说，只要公司有一个雇员，公司本身就不能被删除）。此规则必须由过程化代码来实现，如方法 M（注意关系变量EMP并没有可附加此规则声明的外码）。此外，SQL中的标准DELETE操作现在也只能在关系变量COMPANY的方法M中实现，那么如何来实现这一规则呢？其他类似的问题也会被提出，如使用规则 ON DELETE CASCADE，等等。

c. 同时注意到：尽管EMP元组包含COMPANY元组，当删除EMP元组时并不“级联地”删除相应的COMPANY元组。

从以上几点可以看出，我们讨论的不再是关系模型。其基本的数据对象不再是仅包含值的关系，而是包含值和指针的“关系”——就关系模型而言，这已经不再是关系。换句话说，我们损害了关系模型的概念完整性[㊀]。

- 假设将关系变量EMP在属性HOBBIES上的投影定义为视图V。V是由基表导出的关系变量，所以，如果等式“关系变量 =类”成立，V也是一个类。它是什么类？类当然还有方法，在V上定义的方法又是什么？

  “类”EMP只有一个方法RETIREMENT_BENEFITS，显然这一方法并不适合“类” V。事实上，应用于“类”EMP的方法几乎都不能应用于“类” V，所以对于投影的结果没有任何可使用的方法；也就是说，无论投影的结果是什么，它肯定不是类（也许我们认为它是类，但它有公共实例变量而没有方法，我们已观察到一个真正的“封装”类有方法而没有公共实例变量）。

  非常明显，当人们将关系变量与类等同时，他们只考虑到基表而忘记了由基表所导出的表（我们在上面讨论的指针是指向基表元组的指针，而非导出表）。然而，以这种方式区分基表和导出表是错误的，因为关系变量是基表还是导出表这一问题的答案从某种意义上来说是任意的（想一想第19章中关于可交换性原则的讨论）。

- 最后，能够支持什么域？那些支持等式“关系变量 =类”的人对域不会有很大的认识，因为他们没能看到域对于整个模式的适应性。然而，从我们的大量讨论（例如第 3章中的讨论）可以看出，域是必要的。

㊀ “概念完整性”来自Fred Brooks，他在[25.1]中写到：“概念完整性是系统设计中最重要的考虑因素。在系统中删去某些不协调的特性以反映整体设计思想，要比包含某些优秀却彼此独立、不能互相协调的特性要好”(原文为*意大利语*)。二十年后，他又补充到：“一个简洁、优秀的软件产品必须表现出……一致的思维模型……。概念完整性……是用户能方便使用的最重要的因素……。现在我更为坚信这一点。概念完整性是产品质量的核心”。

以上所有内容可以概括如下：显然，系统可以在错误的等式“关系变量 =类”上进行构建；事实上也确实有这样的系统存在。然而，那些系统（就像汽车在引擎中没有油或建筑在沙滩上的房子）也许能提供一些有用的服务，但最终注定是要失败的。

第一个根本性错误从何而来

推敲第一个根本性错误的来源是一件有趣的事。我们认为其根源来自对象世界中术语的意义缺乏一致性。具体来说，对于术语“对象”本身就没有一个统一的理解，这也是我们为什么倾向于不使用这一术语的原因。

尽管这样，至少在对象编程语言领域，术语“对象”通常指更为传统的值或变量。不幸的是，此术语还用在其他领域；如作为“对象分析与设计”或“对象模型”技术的一部分使用在某些语义模型领域（见[13.3]）。显然，在那些领域中，“对象”不表示值或变量，而是数据库中通常所称的实体（与编程语言中的对象不同，这里的对象并不是封装的）。换句话说，“对象模型”实际上就是“实体/关系模型”；在参考文献[13.3]中或多或少承认了这一点。结果，在某些系统中标识为“对象”的事物被映射为关系变量中的元组而非域中的值。

25.3 第二个根本性错误

在本节中我们将讨论第二个根本性错误；正如我们将要看到的，第二个错误可以说是第一个错误在逻辑上的延续，但就其本身而言也非常关键（实际上，在避免第一个错误的情况下，仍有可能犯第二个错误）。第二个错误为：把指针与关系混为一谈。为了说明这个问题，首先回顾一下采用等式“关系变量=类”后的主要特性。某些读者可能对上一节的内容有些疑惑，因为某些我们似乎是反对的特性在本书的前面部分曾经支持过（元组和关系值属性即为一个例子）。所以在这里就这一问题做进一步说明：

- 元组和关系值属性：事实上，我们并不反对这种属性。我们所反对的是（a）必须有这种属性的思想，尤其当其值已出现在其他(基本)关系变量中时；（b）属性的值并非元组或关系本身，而是指向元组或关系的指针的思想——这意味着，我们讨论的已根本不是元组或关系值。注意：通过指针指向元组或关系值的思想毫无意义，关于这一点下面我们还将详细讨论。
- 关系变量的相关操作符（“方法”）：我们也不反对这一思想——实质上这就是存储或触发过程的另一种表述方式。我们所反对的是将操作符与关系变量（且只与关系变量）联系而不与域或类型相联系的思想。我们也反对将操作符与特定关系变量联系起来的思想（这其实是目标运算对象的另一种表现形式）。
- 子类和超类：对此我们明确反对……。在一个将关系变量与类等同的系统中，子类和超类变成了子表和超表——对于这一思想非常值得怀疑[13.12]。我们希望拥有第19章所描述的适当的继承性支持。
- 路径表达式：我们并不反对那些在参照中用于语法简写的路径表达式——例如[25.11]中提到的从外部码到相关候选码的表达形式。然而，25.2节中讨论的路径表达式却是某些指针链的简写，对此我们是坚决反对的（因为我们首先反对指针）。
- 元组和关系名称：这是必要的——虽然需要一般化为元组和关系选择器[3.3]。
- 关系比较操作符：同样是必要的（但需要遵循正确的操作方式）。
- 遍历类层次结构的操作符：如果“类的层次”即意味着“关系变量的层次”，那么我们对此是坚决反对的，因为这样打破了关系封闭的属性（见[25.31]）。如果“类的层次”

意味着第19章中所提到的“类型层次”，我们则表示赞成（但实际并非如此）。

- 在诸如SELECT或WHERE等子句中调用方法：当然不反对。
- 对包含元组或关系的属性值中单个成员的访问：当然不反对。

现在重点考虑将指针与关系相混淆的问题。问题的关键是显然的，从定义来看，指针指向的是变量而不是值（因为只有变量才有地址而值没有）；如果关系变量 R1中的属性值为指向关系变量R2的指针，这些指针指向的一定是元组变量而非元组值。在关系模型中并不存在元组变量。关系模型处理关系值，关系值（大致上）是元组值的集合，而元组值又（大致上）是标量值的集合。关系模型也处理关系变量，其值为关系；然而，它并不处理元组变量或标量变量。总之，在关系模型中唯一的变量（也是关系数据库中唯一允许的变量）为关系变量。所以，将指针与关系相混淆的思想实质上是在关系模型中引进了某一全新的变量，从而构成了对关系模型的背离。正如上节中指出的，这严重损害了关系模型的概念完整性。

对这一观点更详细的论证可以参见参考文献 [24.21]和[25.11]。在参考文献[25.8-25.10]和[25.13]中讨论了在数据模型中数据构造的基本思想。

通过以上讨论，我们很不幸地看到，目前绝大多数对象 /关系产品 —— 甚至那些避免了第一种根本性错误的产品 —— 就是因为采用了上述方式而混淆了指针和关系的区别。当 Codd定义关系模型时，他有意排除了指针。他曾在 [5.2]中写到：

> 我们可以放心地假设所有用户（包括最终用户）理解值之间的比较，但能够理解指针复杂性的用户却实在不多。关系模型是建立在这一原则基础上的……。即使用户能够理解指针的复杂性，对指针的处理也比值之间的比较更容易出错。

具体来说，指针的使用将会导致指针冲突，而这正是系统容易出错的一大原因。如第 24章中所提到的，对象系统的这一特性有时被贬低为“就像是 CODASYL系统的重新使用”。

第二个根本性错误从何而来

想为第二个根本性错误找出任何辩护之词是非常困难的（主要是指技术上的合理性说明 —— 有些辩护只从策略上考虑而根本不涉及技术）。当然，由于对象系统和编程语言都包含指针（以对象ID的形式表示），将指针与关系混淆的思想很可能是希望使关系系统更为“对象化”，但这种“辩护”只是将问题推向了另一个层次；我们已经很清楚地表明：对象系统将指针暴露给用户是因为它们不能很好地区分模型与实现的差别。

所以我们只能猜测，指针与关系混淆的思想之所以被广泛采用是因为很少有人理解为什么当初要将指针排除在关系之外。正如 Santayana所说：那些不能牢记过去的人必将重复过去所犯的错误。在这一点上我们非常同意 Maurice Wilkes在[25.35]中的说法：

> 我很高兴看到计算机科学的教学建立在历史框架的基础上……学生们应该了解目前这一领域的发展从何而来，做过哪些尝试，哪些可行哪些不可行以及硬件的改进对其发展的贡献。如果在教学中缺少这些内容，人们往往会在一些原则性问题上犯错误。这样我们不但不能站在前人的肩膀上继续前进，反而还会热衷于一些已被证明是不可行的方法。

25.4 实现上的问题

适当的数据类型支持所带来的一大好处就是允许第三方厂商（包括 DBMS厂商本身）构

建并出售可方便地插入到DBMS中的独立的“数据类型”包。例如对复杂文本处理、金融时间序列处理、地理空间数据分析等的支持。对这些包的描述各种各样，如Informix的“数据刀片”(data blades)、“数据盒”(data cartridges)、IBM的“关系扩展器”(relational extenders)[⊖]，等等。在下文中，我们仍将使用术语“类型包”(type package)。

然而，在系统中加入新的类型包并非简单的工作；要想具备此能力，首先需要DBMS本身在设计和结构方面做相应变动。考虑查询中包含引用用户定义类型数据和调用用户定义操作符的情况：

- 首先，查询语言编译器要进行语法分析和类型检查，所以编译器必须知道用户定义的类型和操作符。
- 其次，优化器必须确定适当的查询计划，所以它也要了解用户定义的类型和操作符的某些特性。具体来说，它必须知道数据在物理上是如何存储的。
- 最后，管理物理存储的部分也必须支持新的存储结构（“四叉树、R树等”），它甚至还应该支持有经验的用户引入新的存储结构，并访问自己定义的方法。

总之，系统需具有可扩展性——事实上在每一层上都要有可扩展性。下面我们将对每一层做简要讨论。

1. 语法分析和类型检查

在传统系统中，所有可用的类型和操作符都是内置的，有关它们的信息可以“硬捆绑”到查询语言编译器中。相反，在用户可自定义类型与操作符的系统中，这种“硬捆绑”的方法显然是行不通的。所要改进的地方包括：

1) 用户定义类型和操作符信息——包括内置类型和操作符信息——保存在系统字典中。这意味着字典本身需要重新设计（至少需要扩展）；同时，引入新的类型包意味着需对字典进行大量更新（在Tutorial D中，更新是在执行TYPE和OPERATOR定义声明时完成的）。
2) 需要重写编译器来访问字典以获取必要的类型和操作符信息。通过这些信息可以实现第5、8和19章中所描述的类型检查等内容。

2. 优化

关于优化设计有许多问题，在本书中我们只触及到问题的表面。这些问题包括：

- 表达式转换（“查询重写”）：如第17章中所看到的，传统的优化器通过某些转换规则来重写查询。然而，这些转换规则一般都“硬捆绑”在优化器上（因为所有数据类型和操作符都是内置的）。相反，在对象/关系系统中相关信息（至少有关用户自定义的类型与操作符）需要保存在目录中——这就意味着目录需进一步扩充，而优化器本身也需要重写。下面是一些示例：
  a. 给定表达式NOT(QTY>500)，传统优化器将会把它变成QTY≤500（采用第二种表达形式能够利用QTY上的索引而第一种却不行）。同样，如果用户定义的操作符是某种形式的相反关系，则也需要通过一定的方式通知优化器。
  b. 传统的优化器能够知道：表达式QTY>500与表达式500<QTY在逻辑上是一致的。如果用户定义的操作符中存在这种情况，也需要通过一定的方式通知优化器。

⊖ 我们认为这是一个很不恰当的术语。

c. 传统的优化器还能知道：操作符“ + ”与“ - ”相互抵消（即互为反操作）；例如，表达式QTY+500-500可简化为QTY。当用户定义的两个操作符也互为反操作时，需要通知优化器。

- 选择率：给定某一布尔表达式，如QTY>500，典型的优化器会估算一下此表达式的选择率（即使表达式成立的元组的百分比）。对于内置的数据类型和操作符，选择率信息可以“硬捆绑”到优化器中；而对于用户定义的类型和操作符，则需要向优化器提供一些用户定义的代码来进行选择率的估算。
- 代价公式：优化器需要知道执行某个用户定义操作符的代价。例如，给定表达式 *p* AND *q*，其中*p*为某个复杂多边形的AREA操作符，而*q*为简单的比较如QTY>500。我们当然希望系统先执行*q*，只有那些满足*q*的元组才需要执行*p*做进一步判断。事实上，一些经典的表达式转换启发规则，如在连接前先做选择等，对于用户定义的类型和操作符来说都不一定有效[25.7，25.18]。
- 存储结构和访问方法：显然优化器实际上需要知道存储结构和访问方法（见下一小节）。

3. 存储结构

显然对象/关系系统比SQL系统在物理层上需要更多的存储和访问数据的方式。下面是一些相关的考虑：

- 新的存储结构：如前所述，系统需要支持新的“硬捆绑”的存储结构（如R树等），甚至还需要提供方式，让有经验的用户自己来引入新的存储结构和访问方法。
- 用户自定义类型数据上的索引：传统的索引基于内置类型的数据，并能够理解内置操作符的意义。在对象/关系系统中，要想在用户定义类型的数据上构建索引，则需要了解用户定义操作符的语义（假设此操作符在一开始已被定义了）。
- 操作符结果集上的索引：直接在类型POLYGON的数据集合上构建索引比较简单，最可能的方式是按照多边形的内部字符串来对索引进行排序[⊖]。然而，基于多边形面积来构建索引将会有用得多。注意：我们在第21章中将此索引称作函数索引。

25.5 真正融合的好处

在参考文献[25.31]中，Stonebraker列出了DBMS的“分类矩阵”（见图25-4）。矩阵的第一象限表示那些只处理简单数据并且不支持特定查询的应用（传统的文字处理器是其中的代表）。这类应用从经典的定义来看并非真正的数据库应用；所谓的“ DBMS”只是服务于特定需求并构建在操作系统之上的内置文件系统。

第二象限表示那些支持特定查询但只处理简单数据的应用。目前绝大多数应用都属于这一象限，这也是传统的关系DBMS最为适合的领域。

第三象限表示那些不支持特定查询但能够处理复杂数据的应用。例如， CAD/CAM应用即属于这一象限。目前的对象DBMS其目标就是这部分的市场（传统的SQL产品在第三象限中的应用效果不理想）。

最后，第四象限表示那些既支持特定查询又能够处理复杂数据的应用。 Stonebraker曾给出一个例子：某数据库中包含许多数字幻灯片，拥有典型查询如“给出所有在加州萨克拉门

[⊖] 在第24章中我们曾经提到，系统对于“二进制的大型对象”一般通过数据类型 BLOB来处理。在对象/关系系统中，某些用户定义类型的数据值在物理上也可以存储成 BLOB的形式。

托方圆20英里内拍摄的有关日落的照片”。通过对这一示例的分析，他提出了自己的观点：(a) 对象/关系DBMS适合第四象限的应用；（b）若干年后，大部分应用将会在此象限中。例如，甚至简单的人事资料也会包括雇员的相片、声音记录等。

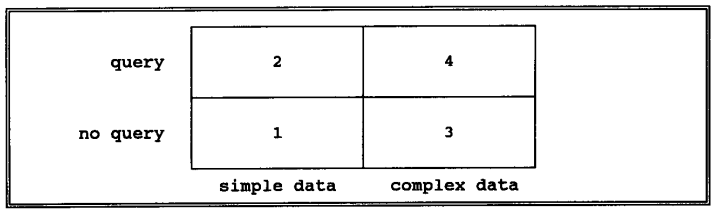

图25-4 Stonebraker的DBMS分类矩阵

总之，Stonebraker论证（我们同意这一点）“对象/关系系统是未来的发展方向”；它们并不会昙花一现并被其他思想所取代。然而，应该提醒大家的是，就我们了解，一个真正的对象/关系系统首先是一个真正的关系系统。具体来说，这一系统不会犯那两个根本性错误！在这一点上Stonebraker可能不怎么同意我们的观点——至少在文献[25.31]中他并没有提到这两个错误，事实上，文中甚至认为将指针与关系结合是可接受也是必要的。

尽管如此，我们仍然认为真正的对象/关系系统能够解决单纯的对象系统所遇到的所有问题。具体来说，此系统可方便地支持下列内容：

- 特定查询、视图定义及声明完整性约束；
- 跨越整个类的方法（这样就不需要特别的“目标”运算对象）；
- 动态定义类（对于特定查询的结果）；
- 双重模式访问（在第24章中我们并未强调这一点，但典型的对象系统不支持双重模式访问——相反，它们使用不同的语言来访问数据库）；
- 延迟（提交时）完整性检查；
- 事务约束；
- 语义优化；
- 度（属性数目）在2以上的关系；
- 外码规则（ON DELETE CASCADE等）；
- 可优化性；

等等。此外还支持：

- 屏蔽OID和指针冲突，使其对用户不可见；
- “麻烦”对象的问题（例如，连接两个对象的结果是什么？）不再存在；
- 仍保持封装的优点，但针对的是关系中的标量值而非关系本身；
- 关系系统能处理“复杂的”应用，如CAD/CAM等。

最后，这一方法在概念上是清晰的。

25.6 小结

在本章中我们首先简要讨论了对象/关系系统。这一系统实际上就是（也应该是）能很好支持关系域（即类型）概念的关系系统——这就意味着用户能够定义自己的类型。为了获得对象方面的功能，我们并不需要对关系模型做任何改动（除了去实现它们）。

我们还讨论了两个根本性错误。第一个错误是将对象类与关系变量等同（不幸的是，这一等式在表面上非常吸引人）。我们猜测这一错误来自对术语“对象”两种不同理解的混淆。我们通过一个示例详细讨论了犯第一个错误的系统将会怎样，并且解释了这一错误的部分后果。其中一个重要的后果就是将直接导致犯第二个根本性错误！——即将指针与关系相混淆（当然不犯第一个错误的系统也可能犯第二个错误，并且目前市场上几乎每一个产品都犯有这一错误）。我们认为，第二个根本性错误在许多方面削弱了关系模型的概念完整性。具体来说，它破坏了基本关系与导出关系之间的可交换性原则。

接下来，我们又简要讨论了实现上存在的问题。其基本考虑在于加入新的“类型包”将至少影响系统中编译器、优化器和存储管理的设计。结论是，对象/关系系统不能通过在关系系统之上包装一层的方法来实现；系统需要从底层开始重建，以使每一部分具备必要的可扩展性。

最后，我们介绍了Stonebraker的DBMS分类矩阵，并简要讨论了将对象与关系技术真正融合所能产生的好处（这里“真正的”一词是指系统不犯任何一个根本性错误）。

参考文献和简介

在过去几年中已经构建了一些对象/关系原型。其中有两个最为知名，分别是加州大学伯克利分校的Postgres系统[25.26，25.30，25.32]和IBM研究中心的Starburst系统[25.14，25.17，25.21~25.22]。我们认为这两种系统（至少其原始形式）都没有采用“明显正确的”等式“域=类”。

提醒一句，在SQL3中包含了一些特意为支持对象/关系系统而加入的新特性（参见附录B）。

25.1 Frederick P. Brooks, Jr.: *The MYthical Mcm-Month* (20th anniversary edition). Reading, Mass.: Addison-Wesley (1995).

25.2 Michael J. Carey. Nelson M. Mattos, and Anil K. Nori: “Object/Relational Database Systems: Principles, Products, and Challenges,” Proc. 1997 ACM SIGNIOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

文中写到：“抽象数据类型、用户定义函数、元组类型、参照、继承、子表、集合、触发器——所有这些到底是什么？”好问题！可以看到，这里列出了8个特性（默认假设它们都是SQL3中的特性）。在这8个特性中，至少有4个是我们不希望看到的，有两个几乎是同一事物，而另两个与系统是否是对象/关系系统无关。详见附录B。

25.3 Michael J. Carey *et al.*: “The BUCKY Object/Relational Benchmark,” Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

在其摘要中提到：“BUCKY（通用或复杂的Kwery Ynterfaces标准测试）是一个面向查询的标准测试，测试了许多对象/关系系统的主要特性，包括行类型与继承、参照与路径表达式、原子值集合与参照集合、方法与延迟绑定以及用户定义抽象数据类型与用户定义方法。”

25.4 R. G. G. Cattell: “What Are Next-Generation DB Systems?”, *CACM 34*, No. 10 (October 1991).

25.5 Donald D. Chamberlin: “Relations and References-Another Point of View,” *InfoDB 10*, No. 6 (April 1997).

见[25.11]中的注释。

25.6 SUrajit Chaudhuri and Luis Gravano: “Optimizing Queries over Multi-Media Repositories,” Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

对象/关系数据库可以充当“多媒体仓库”。多媒体数据查询的结果一般不仅仅是结果对象集合，还包括对象与查询条件的匹配程度（例如某一图片红的程度）。在查询中既可以设定匹配度的最低要求，也可以设定结果的限额[6.4]。论文中考虑了对于这类查询的优化。

25.7 Surajit Chaudhuri and Kyuseok Shim: “Optimization of Queries with User-Defined Predicates,” Proc. 22nd Int. Conf. on Very Large Data Bases, Mumbai (Bombay), India (September 1996).

25.8 E. F. Codd and C. J. Date: “ Interactive Support for Nonprogrammers: The Relational and Network Approaches,” in C. J. Date, *Relational Database: Selected Writings*. Reading, Mass.: AddisonWesley (1986).

论文中介绍了关于本质性的思想，这一概念对于正确理解数据模型（包括术语的两种意义，见第1章的1.3节）是非常重要的。关系模型只有唯一的本质数据结构，即关系本身。相反，对象模型却有许多本质数据结构，如集合、无序单元组、列表、数组，等等（尚未提到对象ID）。参考文献[25.9~25.10]和[25.13]对此有进一步的解释。

25.9 C. J. Date: “Support for the Conceptual Schema: The Relational and Network approaches,” in *Relational Database Writings* 1985-1989. Reading, Mass.: Addison-Wesley (1990).

这篇论文反对将指针与关系相混淆[25.11]的一个论据是指针的复杂性。文中通过一个示例清楚地表明了这一观点（见图25-5与图25-6）。

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P4	4
P5	P3	1
P3	P6	3
P6	P1	9
P5	P6	8
P2	P4	3

图25-5 材料单关系

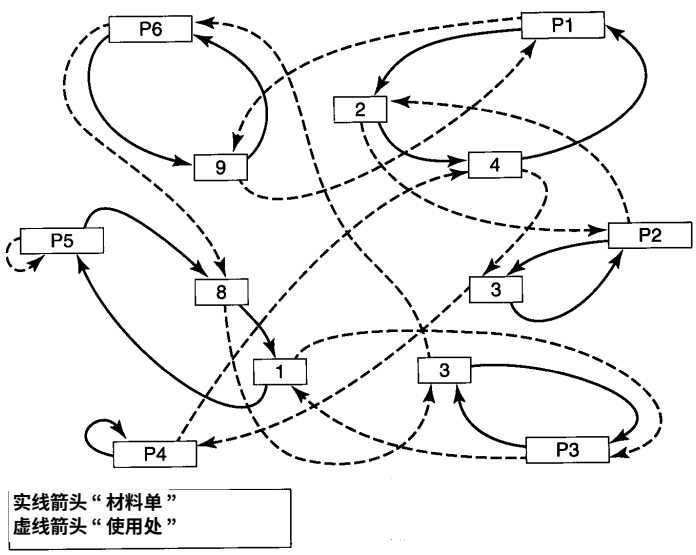

图25-6 图25-5中内容基于指针后的情形

25.10 C. J. Date: “Essentiality,” in *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).

25.11 C. J. Date: “ Don’t Mix Pointers and Relations!” and “ Don’t Mix Pointers and Relations—*Please*!’, both in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).

这两篇论文的第一篇强烈批评了第二个根本性错误。在 [25.5]中，Chamberlin驳斥了第一篇论文的观点，第二篇论文是对 Chamberlin所驳斥内容的直接回应。

25.12 C. J. Date: “Objects and Relations: Forty-Seven Points of Light,” in C. J. Date, Hugh Darwen, aiid David McGoveran, *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).

这篇文章对文献 [25.19] 做了极为详细的答复。

25.13 C. J. Date: “Relational Really Is Different,” installment no. 10 of reference [5.9]. *www.intelligententerprise.com.*

25.14 Linda G. DeMichiel, Donald D. Chamberlin, Bruce G. Lindsay, Rakesh Agrawal, and Manish Arya: “Polyflot: Extensions to Relational Databases for Sharable Types and Functions in a MultiLanua Environment,” IBM Research Report RJ8888 (July 1992).

在其摘要中提到：“ Polyglot系统是一个可扩充关系数据库类型的系统，它支持继承、封装和动态方法调度 ”（‘ 动态方法调度 ” 为 “ 运行中绑定 ” 的另一个术语）。“ Polyglot系统允许使用不同的应用语言，同时还允许对象在跨越数据库与应用程序的边界时保持其自身的行为。这篇论文描述了 Polyglot系统的设计，为了支持 Polyglot中的类型与方法而对SQL语言所做的扩充以及 Polyglot系统在Starburst关系原型中的实现。”

Polyglot系统涉及了本章中（包括第 5、19、24章）所提到的所有问题。然而，还有两点需要提出。第一，系统中没有提到关系术语 “ 域 ”（令人奇怪）。第二，Polyglot系统为类型生成器（在 Polyglot中称为元类型）提供了基类型、元组类型、重命名类型、数组类系和语言类型，唯独没有关系类型（同样令人奇怪）。当然，系统允许引入新的类型生成器。

25.15 David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu: “Client-Server Paradise,” Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

Paradise系统 —— “ 并行数据信息系统 ” —— 是威斯康星大学设计的对象/关系（原先称为 “ 扩展关系 ”）原型，其处理对象为 GIS（地理信息系统）方面的应用。文中描述了 Paradise系统的设计与实现。

25.16 Michael Godfrey, Tobias Mayr, Praveen Seshadri, and Thorsten von Eichen: “Secure and Portable Database Extensibility,” Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

“ 自从不安全的客户端支持了用户定义运算符后， DBMS必须明白这些操作符可能破坏系统，直接修改文件和存储器中的内容（避开权限管理机制），独占CPU、内存或磁盘资源 ”。控制显然是必要的。这篇论文通过 Java和对象/关系原型PREDATOR[25.24]探讨了这一问题。其结论是乐观的，认为数据库系统：“ 通过使用 Java，在不大规模牺牲性能的前提下能够支持安全通用的可扩展性 ”。

25.17 Laura M. Haas, J. C. Freytag, G. M. Lohman, and Hamid Pirahesh: "Extensible Query Processing in Starburst," Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (June 1989).

在论文[25.21]中写道："Starburst系统提供以下支持：在表中加入新的存储方法，提供对新类型的访问方法和完整性约束，提供新的数据类型与函数，在表中提供新的操作符"。而在这之后，Starburst系统的目标又有所扩充。系统被分为两大部分，Core和Corona，分别对应于系统R中的RSS和RDS（[4.2]中对系统R的两个部分做了说明）。Core支持文献[25.21]中所描述的可扩展性函数；而Corona支持Starburst系统的查询语言Hydrogen，这一查询语言是SQL语言的变种，它（a）取消了系统R的SQL语言中大部分实现上的限制；（b）比系统R的SQL语言更具有独立性；（c）支持递归查询；（d）对于用户可扩展。这篇论文还包括了对"查询重写"——即表达式转换规则——问题的讨论。参见参考文献[17.50]。

25.18 Joseph M. Hellerstein and Jeffrey F. Naughton: "Query Execution Techniques for Caching Expensive Methods," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

25.19 Won Kim: " On Marrying Relations and Objects: Relation-Centric and Object-Centric Perspectives. " *Data Base Newsletter 22*, No. 6 (November/December 1994).

这篇论文论证了将关系变量与类相等同观点是一个严重的错误。参考文献[25.12]是对此的一个回答。

25.20 Won Kim: "Bringing Object/Relational Down to Earth," *DBP&D 10*, No. 7 (July 1997).

在这篇文章中，Kim称在对象/关系产品市场上"混乱占了主导"，因为：首先，"将考虑重点过分放在数据类型扩展上"；第二，"对对象/关系完整性的评价……处于非常混乱的地步"。他进而提出"一套可用来判断对象/关系产品是否具备完整性的实用化度量。"在他的评价模式中涉及下列标准：

1) 数据模型
2) 查询语言
3) 关键任务服务
4) 可计算模型
5) 性能与可伸缩性
6) 数据库工具
7) 能力的发挥

关于第一条标准（最重要的标准！），Kim的观点为——与《第三次宣言》[3.3]中的观点完全不同——数据模型必须是"对象管理组（OMG）定义的核心对象模型"，由"关系数据模型和面向对象编程语言中核心的面向对象模型两者的概念构成"。根据Kim的观点，数据模型包含下列概念：类（Kim加入了"或类型"）、实例、属性、完整性约束、对象ID、封装、（多）类继承、（多）ADT继承、类型参照的数据、集合值属性、类属性、类方法，等等。注意：关系——我们认为这是最关键也是最基本的概念——并没有被明确提出；Kim称OMG的核心对象模型除上面列出的概念外还包含有整个关系模型，但事实上并不是这样。

25.21 Bruce Lindsay. John McPherson, and Hamid Pirahesh: "A Data Management Extension Architecture," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

描述了Starburst原型系统的整个体系结构。Starburst系统“使关系数据库系统在数据管理方面的扩展实现起来更为方便”。在论文中描述了两类扩展：用户定义的存储结构和访问方法；用户定义的完整性约束（但所有的完整性约束不都是用户定义的吗？）和触发过程。然而，“还有其它一些能够扩展DBMS的重要方面，包括用户定义……数据类型和查询评价技术。”

25.22 Guy M. Lohman *et al.*: “Extensions to Starburst: Objects, Types, Functions, and Rules, ” CACM 34, No. 10 (October 1991).

25.23 David Maier: “Comments on the Third- Generation Database System Manifesto,” Tech. Report No. CS/E 91-012, Oregon Graduate Center, Beaverton, Ore. (April 1991).

Maier几乎对文献[25.34]中的每项内容都持反对观点。我们赞成其中的一些批评意见，也反对其中的另一些。然而，我们认为下述评论（其证实了我们所提出的，对象系统只涉及一个优秀思想，即适当的数据类型支持）是非常有趣的：“在面向对象数据库领域，大部分人都试图提取出数据库‘面向对象’的本质……。我个人对OODB最重要特性到底是什么的认识也在随着时间的变化而不断变化。一开始我认为是继承和消息模型，后来我有认为对象标识，对复杂声明的支持及行为上的封装更为重要。如今，在听取了OODBMS用户关于他们对系统中哪部分最为满意的意见后，我认为类型的可扩展才是关键。标识、复合态和封装也很重要，但这些只有在系统能够支持新数据类型的创建之后才是重要的”。

25.24 Jignesh Patel et al.: “Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation,” Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

在其摘要中提到：“这篇论文在地理空间数据库的并行化方面提出了一些新的技术，同时讨论了这些技术在Paradise对象/关系数据库系统中的实现”。[25.15]

25.25 Raghu Ramakrishnan: Database Management Systems. Boston, Mass.: McGraw-Hill (1998).

25.26 Lawrence A. Rowe and Michael R. Stonebraker: “ The Postgres Data Model,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

25.27 Hanan Samet: *The Design andAnalysis of Spatial Data Structures*. Reading, Mass.: AddisonWesley (1990).

25.28 Cynthia Maro Saracco: *Universal Database Management: A Guide to ObjectlRelational Technology.* San Francisco, Calif.: Morgan Kaufmann (1999).

一本易读的高层次概论书籍。然而，我们注意到Saracco支持（与Stonebraker在文献[25.31]中的观点正好相似）一种非常不可靠的继承形式，此形式涉及子表与超表思想的某种版本——我们在[13.12]一开始就对此持怀疑态度，但又与SQL3中的版本完全不同。具体来说，假设表PGMR（“程序员”）被定义为表EMP（“雇员”）的子表。对此Saracco和Stonebraker认为EMP只包含那些不是程序员的雇员信息的元组，而SQL3认为EMP包含所有雇员信息的元组（参见附录B）。

25.29 Praveen Seshadri and Mark Paskin: “ PREDATOR: An OR-DBMS with Enhanced Data Types.” Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

“PREDATOR系统的基本思想是为每个数据类型提供机制以确定其相关方法的语义；

这些语义在查询优化中将会用到。”

25.30 Michael Stonebraker: “The Design of the Postgres Storage System,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

25.31 Michael Stonebraker and Paul Brown (with Dorothy Moore): *Object/Relational DBMSs*: *Trackbig the Next Great Wave* (2nd edition). San Francisco, Calif.: Morgan Kaufmann (1999).

本书是对象/关系系统的指南。它主要 —— 事实上，几乎毫无例外 —— 基于Informix的动态服务产品的通用数据选件。通用数据选件基于早期的 Illustra系统（一个商业化产品，Stonebraker本人推动了这个产品的发展）。在参考文献[3.3]中有对此书的分析和批评；还可以参见文献[25.28]中的评论。

25.32 Michael Stonebraker and Greg Kemnitz: “The Postgres Next Generation Database Managenient System.” *CA CM 34*, No. 10 (October 1991).

25.33 Michael Stonebraker and Lawrence A. Rowe: “The Design of Postgres,” Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (June 1986).

Postgres系统声明的目标有：

1) 为复杂对象提供更好的支持；
2) 提供数据类型、操作符和访问方法的用户可扩展性；
3) 提供活动数据库工具（警告器和触发器）和接口支持；
4) 简化DBMS关于系统恢复的代码；
5) 能够利用光盘、多处理器工作站和定制的 VLSI芯片的良好特性；
6) 对关系模型做尽可能少的改动（最好不改动）。

25.34 Michael Stonebraker *et al*.: “ Third-Generation Database System Manifesto, ” *ACM SIGMOD Record 19*, No.3 (September 1990).

这篇文章部分是为了回答《面向对象数据库系统宣言》 [24.1]中的内容，认为它在本质上忽略了整个关系模型。文中称：“ 第二代系统在两个方面做出了重大贡献；一是非过程化数据访问，另一是数据独立性，这两个优点绝不应该在第三代系统中受到损害 ”。下列特性被认为是作为第三代 DBMS 的本质要求：

1) 提供传统数据库服务，并加入更丰富的对象结构和规则
   - 丰富的类型系统
   - 继承
   - 函数和封装
   - 可选择的由系统分配的元组 ID
   - 不针对特定对象的各项规则（如完整性规则）
2) 包含第二代 DBMS
   - 只在万不得已的情况下才使用导航
   - 集合的内涵和外延定义（即由系统自动维护的集合与由用户手工维护的集合）
   - 可更新的视图
   - 聚集、索引等对用户不可见
3) 支持开放系统
   - 多语言支持

- 类型的持续无关性
- SQL （作为通用的数据语言）
- 查询及相应结果必须处于客户/服务器通信的最底层

文献[3.3]和[25.23]对这篇文章进行了分析和批评。注意：现在我们可以解释为什么《第三次宣言》被称为“第三次”……。它是建立在将文献[24.1]和[25.34]作为前两次宣言的基础上的。

25.35 Maurice V. Wilkes: “Software and the Programmer,” *CACM 34*, No.5 (May 1991).