面向对象的数据库系统宣言

Malcolm Atkinson, Francois Bancilhon, David Dewitt,
Klaus Dittrich, David Maier, Stanley Zdonik

摘要 这篇文章试图定义面向对象的数据库系统。它描述了作为一个合格的面向对象的数据库系统所应具备的主要性质和特征。

我们把这些特性分成了三组：

·必备的：被称作面向对象的数据库系统所必须满足的特性。它们是复杂对象，对象标识，封装性．类型或类，继承性，结合滞后联编的复载，可扩充性，计算完备性，持久性，辅存管理，并发性，恢复和即席查询功能。

·可选的：为了使系统更完善可添加的而非必备的特性。它们是多重继承性，类型检查和推理，分布．设计事务处理和版本。

·开放的：设计人员可以选择的特性。它们是程序设计范例，表示系统，类型系统及一致性。

我们的立场是，不祇望上述看法成为一个里程碑，以便为进一步的讨论指明方向。

1. 引 言

目前，面向对象的数据库系统(OODBS)从实验到理论均受到了广泛的关注，并且对此系统的定义一直有着相当大的争论。

现阶段可用三个特征来刻划该领域：(i)缺乏通用数据模型，(ii)缺乏形式基础，(iii)强有力的实验活动。

对关系数据库系统(数据模型和查询语言)，Codd的开创性论文[Codd70]给出了清晰的规范说明，而对面向对象的数据库系统尚不存在这样的规范说明[Maier89]。这里我们并不是说没有完全面向对象的数据模型存在，其实很多论文提出过这一思想(见[Albano等1986]，[Lecluse和Richard 89]，[Carey等88]等)，但是各有各的看法。有些看法逐渐集中到面向对象的系统族的几个重要特性，但是目前对什么是面向对象的系统还没有一个明确的一致性意见，更不用说面向对象的数据库系统了。

该领域的第二个特点是缺乏强有力的理论框架。面向对象的程序设计不能与逻辑程序设计相提并论[Van Emdem和Kowalski 76]。需要坚实的理论基础是明显的：象类型或程序这样的概念，其语义常被瞎定义。没有坚实的理论框架，就几乎不可能得到一致的数据模型。

最后一点，许多实验工作正在进行：人们正在实际中建立系统．有些系统还只是原型[Bancilhon等88]，[Nixon等87]，[Banerjee等87]，[Skarra等86]，[Fishman等87]，[Carey等86]，但有些已成为商业产品[Atwood85]，[Maier等84]，[Caruso和Sciore 87]，[G-Base 88]。人们对面向对象的数据库的兴趣好象出于设计支持系统的需要(如CAD，CASE，办公信息系统)。这些应用要求数据库能处理非常复杂的数据，能适度地演化，并能提供由交互式系统支配的高性能。

实现的情况与70年代中期关系数据库系统相似(尽管在面向对象方面有更多的实现)。对关系型系统，尽管在某些方面有分歧意见，如查询语言的形式，关系应为集合还是包(bags)。但绝大多数分歧都是表面的，它们的基本模型是共同的。人们主要在开发实现技术。而现在，我们同时要选择

系统规范说明和开发支持其实现的技术。

这样，关于系统的规范说明，我们采用达尔文的方法：我们希望从正在建立的实验原型中出现一个合适的模型。我们还希望同时得到一个对那个模型可行的实现技术。

可惜，随着仓促的实验，我们急于推出一个系统当作所需的系统，并非因为它最合适，而是因为它最先提供了市场所需的重要的功能子集。早期产品成为事实上的标准并一直存在下去是计算机界一个传统的和可悲的模式。这种模式至少对语言和操作系统的情况是如此的（Fortran，Lisp，Cobol和SQL是这种情形的很好示例）。然而要注意，我们这里的目标并非对语言进行标准化，而是对术语进行求精。

A 现在商定一个面向对象的数据库系统的定义是至为重要的。作为达到这个目标的第一步，本文提出了这样的系统应该具备的特性。我们期待本文将被当作稻草人，也期待其他人来否定或证实这里陈述的观点。注意，本文并非OODBS技术现状的概述，也不打算评价当前的技术状态，它只不过提出一系列定义。

我们把面向对象的数据库系统的特性分成了三类：必备的（为得到此标签，系统必须满足的特性），可选的（为了使系统更完善可添加的而非必备的特性）和开放的（设计人员可以从众多等价的可接受的解中进行选择的特性）。此外，如何更好地系统地阐述每一特性（必备的或可选的）还有一些余地。

本文其余部分组织如下：第2节描述了OODBS的必备特性，第3节描述了可选特性，第4节给出了留给系统设计员的自由度。

2. 必备特性：金科玉律

一个面向对象的数据库系统必须满足二个标准：它应该是一个DBMS，并且是一个面向对象的系统。即在一个可能的范围，它与当前的一批面向对象的程序设计语言一致。第一个标准转化成五个特性：持久性，辅存管理，并发性，恢复和即席查询功能。第二个标准转化为八个特性：复杂对象，对象标识，封装性，类型或类，继承性，结合滞后联编的复载，可扩充性及计算完备性。

2.1 复杂对象 对简单对象运用各种构造符即可得到复杂对象。简单对象有整数，字符，任意长的字符串，布尔量和浮点数（可以添加其它原子类型）。复杂对象构造符有多种，如元组、集合、包、表和数组都是例子。系统应该具备的最小构造符集是集合、表和元组。集合之所以必需是因为它自然地表示了现实世界中的一类事物。而元组之所以必需是因为它自然地表示了实体的性质。当然，集合和元组的重要还因为它们作为对象构造符通过关系模型得到了广泛的接受。表和数组的重要在于它们拥有序，这是现实生活中所具有的，也出现在人们需要矩阵或时间序列数据的许多科学应用中。

对象构造符必须是不相关的：任何构造符都应该能用于任何对象。关系模型的构造符不是这样，因为集合构造符仅能用于元组而元组构造符仅能用于原子值。还有例如非第一范式关系模型，其顶层构造必须是一个关系。

注意，支持复杂对象也需要提供一些适当的操作符，以便把这些对象作为一个整体（不管它们的成分）来处理。即对一个复杂对象的操作必须传及它的所有成分。例如对整个复杂对象的检索或删除，或生成一个“深”拷贝（相对于“浅”拷贝，“浅”拷贝不复制其成分，而仅拷贝对象的指针）。另外，当然系统的用户还可定义复杂对象的其它操作（见下面的可扩充性规则）。但是，这一能力需要系统提供诸如二个可区分的引用类型，（“部分（is-part-of）”和“概括（general）”）。

2.2 对象标识 对象标识在程序设计语言中由来已久。但在数据库中还比较新，如〔Hall等76〕，〔Maier和Price 84〕，〔Khoshafian和Copeland 86〕。其含义如下：在带对象标识的模型中，对象独立于它的值而存在。这样，便有了二个对象等价的概念：二个对象可能相同（它们是同一个对象），也可能相等（它们有同样的值）。这又蕴含了二个含义：一个是对象共享，另一个是对象更新。

对象共享：在基于标识的模型中，二个对象可能共享一个成分。这样，复杂对象的图形表示是一个图，而在没有对象标识的系统中只是一棵树。考虑下例：一个人有一个名字，年龄及若干个孩子。假定彼得和苏珊都有一个名叫约翰的15岁的孩子。在实际生活中，可能有二种情况发生：彼得和苏珊是同一个孩子的父母，或者，涉及到的是二个孩子。在一个无标识的系统中，彼得被描述成：

（彼得，40，{（约翰，15，{ }）}）

且苏珊被描述成：

（苏珊，41，{约翰，15，{ }）}）

这样，无法说明彼得和苏珊是否是同一孩子的父母。在基于标识的模型中，这二个结构可以共享其共同部分（约翰，15，{ }）也可以不共享，这样便处理了上述两种情形。

对象更新：假定彼得和苏珊确实为名叫约翰的孩子的父母。在这种情况下，对苏珊儿子的所有更新将作用于对象约翰，理所当然对彼得的儿子也是这样。在基于值的系统中，二个子对象必须分别更新。对象标识也是一个有力的数据操纵原语。可以成为集合、元组和递归复杂对象操作的基础〔Abiteboul和Kanellakis 89〕。

支持对象标识意味着提供诸如对象赋值、对象拷贝（深拷贝和浅拷贝）及对对象标识和对象相等（深相等和浅相等）进行测试的操作。

当然，可以在一个基于值的系统中通过引入显式的对象标识符来模拟对象标识。但是，这个方法给用户加上了保证对象标识符唯一性和维持参照完整性的负担（并且对诸如无用单元收集这样的操作这是个很重的负担）。

注意，基于标识的模型是强制式程序设计语言的一个标准：程序中操纵的每个对象有一个标识并能更新。这个标识可以来自变量名或存储器中的物理位置。但是这个概念在基于值的纯关系系统中还是挺新的。

2.3 封装性 封装性的概念来自（i）需要明确区分操作的说明和实现及（ii）模块化的需要。对于由一组程序员来设计和实现复杂的应用软件，模块化是必需的。作为保护和授权的工具它也是必需的。

对封装性有二种观点：程序设计语言观点（这是原观点，因为此概念源于它）和数据库改用的观点。

程序设计语言中封装性的概念出自抽象数据类型。按照这种观点，对象有一个接口部分和一个实现部分。接口部分是作用于对象的操作集的说明。这是对象的唯一可见部分。实现部分含有数据和过程二部分。数据部分是对象的描述或状态，而过程部分在一些程序设计语言中描述了每一操作的实现。

这个主张在数据库中变成对象既封装程序又封装数据。在数据库领域，并不清楚类型的结构部分是否是接口部分（这依赖于系统）。而在程序设计语言领域，数据结构明显是属于实现部分而非接口部分。

例如，考虑一个雇员的情况。在关系系统中，一个雇员由某个元组来表示。之后，可用关系语言进行查询，应用程序员可编程序更新此记录，如提高这个雇员工资或开除他。这些程序通常用嵌入有DML语句的强制式程序设计语言或第四代语言写，并被存放在一个传统的文件系统而非数据库中。可见在此方法中，程序和数据之间以及查询语言（指即席查询）和程序设计语言（指应用程序）之间有显著的区别。

在面向对象的系统中，我们把雇员定义成一个具有数据部分（可能与关系系统中所定义的记录非常类似）和操作部分的对象，操作部分包括提升（raise）和开除（fire）操作及其它存取雇员数据的操作。当我们存储一组雇员时，数据和操作均被存到数据库中。

这样，对数据和操作有了一个统一的模型，并且信息能够被隐藏。没有在接口中说明的操作不能被执行。这个限制对更新操作

和检索操作同样适用。

封装提供了一种逻辑数据独立性的形式：我们能够在不改变使用类型的程序的情况下改变类型的实现。这样应用程序就免于随系统更低层上的实现改变而改变。

我们相信，如果让操作可见而将数据和操作的实现隐藏在对象中，就可以获得适当的封装。

但是在有些情况下并不需要封装，在一定条件下如果允许系统打破封装，系统的使用就会非常简单。例如，对即席查询，封装的需要可以降低，因为这时可维护性的问题并不重要。所以，OODBS 必须提供封装机制，但是情况表明也有不太妥当的地方。

2.4 类型和类 这个问题很棘手：有二大类面向对象的系统：支持类概念的系统和支持类型概念的系统。属第一类的有诸如 Smalltalk〔Goldberg和Robson 83〕，Gemstone〔Maier等84〕，Vision〔Caruso和Sciore87〕以及更一般的 Smalltalk 族的系统，Orion〔Banerjee等87〕，Flavors〔Bobrow和 Steifik 81〕，G-Base〔G-Base88〕，Lore〔Caseau89〕和其它更一般的从Lisp中衍生出的系统等。属第二类的有诸如 C++〔Stroustrup86〕，Simula〔Simula67〕，Trellis/Owl〔Shaffert等86〕，Vbase〔Atwood85〕及 02〔Bancilhon等88〕等系统。

在面向对象系统中，类型概括了具有相同特性的一组对象的共同特征。它对应于抽象数据类型的概念。它有二部分：接口和实现(或多个实现)。只有接口部分对类型的用户是可见的，对象的实现部分只有类型的设计人员才可见。接口部分包括一组操作及用法说明（即输入参数的类型和结果类型）。

类型的实现包括数据和操作二部分。数据部分描述了对象数据的内部结构。基于系统的能力，数据部分的结构的复杂程度不一。操作部分包括实现接口部分中操作的过程。

在程序设计语言中，类型作为保证程序正确性而提高程序员工作效率的工具。由于强迫用户说明他/她所操纵的变量和表达式的类型，系统可以依据类型信息来推导程序的正确性。如果类型系统设计得仔细，系统能够在编译时作类型检查，否则某些检查在编译时要推迟。所以类型主要用在编译时检查程序的正确性。通常，在基于类型的系统中，类型不属最高阶层且有一个特殊状态，它不能在运行时修改。

类的概念不同于类型。它的说明与类型相同，但是它更属于运行时刻的概念。它包括二个方面：对象工厂和对象仓库。对象工厂通过对类实行new操作，或繁殖类的某原型对象代表，可创建新的对象。对象仓库表示与类相关的疆域，即该类的一组实例对象。用户可以通过对类的所有元素进行操作来操纵仓库。类不用于检查程序的正确性而用于创建或操纵对象，在大多数用到类机制的系统中，类属于最高阶层，并且可在运行时刻操纵，即可作为参数进行更新或传送。在大多数情况下，每当给系统提供更大的灵活性和一致性，就会使得在编译时的类型检查成为不可能。

当然，类和类型有很大的相似性。每个概念都是按二种意义使用的，在某些系统中差别是很细微的。

我们觉得不应该只选其中之一，而是考虑留给系统设计员去选择（见 4.3节）。但是，我们要求系统提供某种构造数据的机制，即类或类型。这样，经典的数据库模式的概念将由一组类或类型的概念替代。

但是，我们觉得并不一定要系统自动维护类型的疆域（即数据库中一组已知类型的对象）。或即使维护一个类型的疆域，系统可以让用户自行访问。考虑一个例子矩形(rectangle)类型可由多个用户用在多个数据库中。谈论由系统维护所有矩形的集合对它们进行操作是没有意义的。我们认为要求每个用户维护和操纵他自己的矩形集才更加实际。另一方面，在诸如雇员类型的例子中，

系统自动维护雇员的内容可能很得当。

2.5 类或类型的层次结构 继承性有二个好处：它是有力的建模工具，因为它提供了对世界简明而精确的描述，并且它还有助于进行共享说明和应用的实现。

下例有助于显示使系统提供继承机制的好处。假定我们有雇员（Employee)和学生(Student)。每个雇员有一个姓名，大于18的年龄，工资，他/她可能死亡、结婚及获薪。每个学生有年龄、姓名和一组分数。他/她可能死亡，结婚及计算他/她的GPA值。

在关系系统中，数据库设计员为雇员和学生各定义一个关系，并为雇员关系上的die，marry和pay操作、为学生关系上的die，marry和GPA计算写代码。这样，应用程序员要写六个程序。

在面向对象的系统中，用继承性这一性质，我们知道雇员和学生都是人；这样，他们有共同之处（都是人），也有特殊之处。我们引入类型人（Person），它带有属性name和age，并且我们可以为这个类型写die和marry操作。然后，我们说明雇员是特殊类型的人，他继承了人（Person）的属性和操作，并且有一个特殊的属性salary及一个特殊的操作pay。同样，我们说明学生也是一个特殊类型的人，它带有一个特殊的属性：分数集（set-of-grade）和一个特殊的操作：GPA计算。在这种情况下，模式的描述更加结构化、也更加简明（我们有了规范说明）并且我们只要写四个程序（我们又有了实现）。继承性有助于代码重用，因为每个程序都处于受大量对象共享的地位。

至少有四种继承：替代继承，包含继承，限制继承和特化继承。

在替代继承中，如果我们能够对类型t中对象比类型t'的对象实施更多的操作，我们说类型t继承类型t'。这样，对任何具有类型t'的对象的地方，我们能够用一个类型t的对象来替代。这种继承是基于行为而并非值。

包含继承对应于分类的概念。如果类型t的每个对象也是类型t'的对象，则说t是t'的子类型。这种继承基于结构而非操作。例如带有get，set（大小）方法的square类型和带有get，set（大小），fill（颜色）方法的filled-square类型。

限制继承是包含继承的特殊情形。如果类型t'包括满足某种已知限定条件的类型t的所有对象，则类型t是t'的一个子类型。例如少年（teenager）是人（person)的一个子类型：少年（teenager）有的域或操作人（person）都有，但是他们服从更特殊的限制条件（他们的年龄限于13到19岁之间）。

对特化继承，如果类型t'的对象是类型t的对象而t带有更多特殊信息，则类型t是t'的子类型。例如人和雇员，其中雇员的信息是人的信息再加上某些额外的字段。

现有的系统和原型在不同程度上给出了这四种类型的继承，我们不想再提出更特殊的继承风格。

2.6 复载，过载和滞后联编 与前面的例子不同，有时需要同一名字用于不同的操作。例如，考虑display操作：它以一个对象为输入，并把它显示在屏幕上。按照对象不同的类型，我们要用不同的显示机制。如果对象是一张图片，我们要在屏幕上显示它。如果对象是一个人，我们要打印某种格式的元组。最后，如果对象是一个图形，我们需要表示它的图形。现在考虑这个问题：要显示一个集合，而集合中元素的类型在编译时尚不知道。

在传统系统的应用程序中：我们有三个操作：display-person，display-bitmap及display-graph。程序员检测集合中每个对象的类型，再用相应的显示操作。这就迫使程序员要知道集合中对象的所有可能的类型，还要知道相关的显示操作并且要对应的使用它。

```
for x in x do
```

```
begin
  case of type(x)
    person: display(x);
    bitmap: display-bitmap(x);
    graph: display-graph (x) ;
end
end
```

在面向对象的系统中，我们在对象类型这一层次上定义显示操作（系统中最普通的类型）。这样，display仅有一个名字，且可以无区别地用在图形，人员和图片上。但是，要按照类型的不同重新定义每个类型的操作的实现（这个叫做复载（overriding））。结果导致了一个名字（display）表示三个不同的程序（这叫做过载(overloading)）。为了显示集合元素，我们对每一元素简单地运用 display 操作，让系统在运行时选择适当的实现。

```
for x in x do display(x)
```

这里，我们得到另一好处：类型实现者仍要写同样数目的程序，但是应用程序员将不必对这三个程序费心。另外，由于没有关于类型的case语句，代码也更加简单。最后，当要引入一个新类型作为这个类型的新实例时，display程序不必修改即可工作（只要我们为这个新类型复载该显示方法）。因此，代码的可维护性更好。

为了提供这个新功能，系统不能在编译时就把操作名联编到程序上。所以，操作名必须在运行时解释（转换成程序地址）。这个推迟的转换叫做滞后联编(late binding)。

注意，尽管滞后联编使类型检查更加困难（并且在某些情况下不可能），但并没有完全排除它。

2.7 计算完备性 从程序设计语言的观点，这个性质很明显：它意味着能用数据库系统的 DML 表达任何可计算的功能。以数据库角度看这还很新，因为诸如SQL是不完备的。

我们并非在此倡导面向对象的数据库系统的设计人员设计新的程序设计语言：计算完备性可以通过与现有程序设计语言的合理连接来实现。大多数系统实际上都使用了现有程序设计语言〔Banerjee等87〕，〔Fishman等87〕，〔Atwood85〕，〔Bancilhon 等 88〕；对这个问题的讨论参见〔Bancilhon 和 Maier 88〕。

注意，这不同于资源完备，即能通过语言存取所有系统资源（例如，屏幕和远程通讯）。所以，即使系统是计算完备的，也许不能表示一个完全的应用。但是它比仅仅存储和检索数据、对原子值进行简单计算的数据库系统的能力要强。

2.8 可扩充性 数据库系统带有一组预先定义的类型。程序员可随意用这些类型写它们的应用程序。这组类型必须在如下的意义下可扩充：具有可定义新类型的手段并且系统定义和用户定义的类型在使用上没有区别。当然，系统支持系统定义类型和用户定义类型的方法可能有很大的不同，但是这对应用程序和应用程序员来说是不可见的。记住：类型的定义包括类型上的操作的定义。注意封装性需求意味着要有一个定义新类型的方法，并强调这个方法应具有这样的能力，使得新产生的类型必须具备和已有类型相同的状态。

但是，我们并不要求类型构造符(元组，集合，表等）的集合是可扩充的。

2.9 持久性 这个需求从数据库观点看是很明显的，但从程序设计语言的角度却很新颖，〔Atkinson等83〕。持久性是指程序员保证他/她的数据在进程的执行中存在的能力。持久性应该是互不相关的，即每个对象，不管它的类型是什么，都允许永久地保留下来（不必显式的转换）。这也意味着用户不必转换或拷贝数据来使它持久保存下来。

2.10 辅存管理 辅存管理是数据库管理系统的一个经典特征。它通常有一组方法支持，包括索引管理，数据聚集，数据缓冲，存取路径选择及查询优化。

这些对用户均不可见：它们只是性能特征。但是，从性能上说，它们是十分关键的以至一旦缺少将使系统不能完成某些任务（因为它们花费太长时间）。重要的是它们不可见。应用程序员不必为维护索引、分配盘空间或在盘和主存间传送数据而写代码。系统的逻辑级和物理级之间应有清晰的独立性。

2.11 并发性 关于和系统并发交互的多个用户的管理，系统应该提供和目前数据库系统同样级别的服务。所以它应保证同时在数据库上工作的用户之间的和谐共存。系统还应支持操作序列原子性和受控共享等标准概念。尽管提供的方法可能不够严格，但至少应该提供操作的可串行性。

2.12 恢复 这里再说一遍，系统应该提供和当前数据库系统同样级别的服务。所以，在硬件或软件发生故障时，系统应该恢复，即，使它返回到数据的某相关状态。硬件故障包括处理器和盘故障。

2.13 即席查询功能 这里的主要问题是提供即席查询语言的功能。我们不需要它按照查询语言的形式来做，只要它提供这种服务。例如，图形阅读器就足以完成这个功能。服务包括允许用户仅对数据库作简单的询问。关系系统显然是参照的标准。这样，可以用一些代表性的关系查询来测试系统1是否能做同样的工作。注意，这个功能可由数据操纵语言或它的子集支持。

我们认为查询功能应该满足下列三个标准：（i）它应该是高级的，即应能简明地表示（用一些词或按几个鼠标）重要的查询。这意味着它应该是合理的陈述，即它应该强调"什么"而不是"怎样"。（ii）它应该是有效的。即查询的形式应该是进行过查询优化的。（iii）它应该是与应用无关的。即它应能适用于任何数据库。最后这个要求排除了与应用相关的特殊的查询功能，或需要对用户定义的类型写额外操作。

2.14 小结 这里得出了一组必备的特征。这样传统的和面向对象的数据库系统之间的区别应该很明显了。关系数据库系统不满足规则1到8。CODASYL数据库系统部分满足规则1和2。有人争辩说，面向对象的数据库系统无非是CODASYL系统。应该注意（i）CODASYL并不完全满足这二条规则（对象构造符不是不相关的并且因为关系限于1:n，对象标识的处理是不一致的），及（ii）它们不满足规则3，5，6，8和13。

还有一些特征，它们应该是必备的还是任选的，尚未达成一致的意见。这些特征是：

- 视图定义和派生数据；
- 数据库管理功能；
- 完整性约束条件；
- 模式演化功能。

3. 可选特性：引人入胜

放在这个栏目下的内容能明显地改进系统，但是要使系统成为面向对象的数据库系统它不是必备的。

这里某些特性具有面向对象的性质（如多重继承性）。把它们归属于此类是因为它们不属于核心需求，尽管它们使系统更加面向对象。

另一些只是数据库特征（如设计事务管理）。这些特性通常改进了数据库系统的功能，但是它们不属于数据库系统的核心需求并且与面向对象概念无关。事实上它们大部分目标在于为新的应用领域（CAD/CAM，CASE，办公室自动化等）服务，它们更加面向应用而非面向技术。由于许多面向对象的数据库系统当前目标在于这些新的应用，所以这些特性和系统面向对象的性质之间有些混淆。

3.1 多重继承性 系统是否提供多重继承性是可选的。由于在面向对象界对多重继承性尚未达成一致意见，所以我们认为提供这个特性是任选的。注意一旦决定支持多重继承性，对处理冲突分解的问题有很多可能的解决办法。

3.2 类型检查和类型推理 系统在编译时实施类型检查的程度是随意的，但是越

多越好。最佳情形是编译程序接受的程序不会产生任何运行时的类型错误。类型推理的多寡对系统设计员也是随意的：越多越好，理想的情况是只须说明基本类型，临时类型由系统推理而成。

3.3 分布 很明显这与系统的面向对象的性质无关。这样，数据库系统可以是分布式的也可以不是。

3.4 设计事务处理 在很多新的应用中，面向经典商务的数据库系统的事务模型不能令人满意：事务往往很长，而通常的可串行性标准不再适合。这样，许多OODBS支持设计事务处理（长事务或嵌套事务）。

3.5 版本 大多数新的应用（CAD/CAM和CASE）涉及到设计活动并需要某种形式的版本控制。这样，许多OODBS支持版本。再说明一遍，提供版本控制机制并非系统的核心需求的一部分。

4. 开放的选择

每个满足规则1到13的系统都是OODBS。当设计这样的系统时，仍要作大量的设计选择。这给OODBS设计人员提供了自由度。这些特性不同于那些必备的特性，它们在科技界尚未达成一致意见。它们也不同于可选特性，因为我们不知道其中哪些是更加面向对象的。

4.1 程序设计范例 我们没有理由把一个程序设计范例强加于人：逻辑程序设计风格〔Bancilhon 86〕，〔Zaniolo 86〕，函数式程序设计风格〔Albano等86〕，〔Banerjee等87〕，或者强制式程序设计风格〔Stroustrup 86〕，〔Eiffel 87〕，〔Atwood 85〕都能选作程序设计范例。另一个解决办法就是系统独立于程序设计风格而支持多种程序设计范例〔Skarra等 86〕，〔Bancilhon等 88〕。

当然，语法选择也是随意的，而人们总是争辩应该写"john hire"，还是"john. hire"，"hire john"，或"hire(john)"。

4.2 表示系统 表示系统由一组原子类型和一组构造符定义。尽管我们给出了对描述对象的表示可用的最低数量的原子类型和构造符（程序设计语言的基本类型，及集合、元组和表构造符），这仍可用多种不同方法进行扩充。

4.3 类型系统 对于类型构成也是自由的。我们唯一需要的类型形成机制是封装。可以有其它诸如类属类型或类型生成器（如set〔T〕，T可以是任意的类型），限制，联合和标记（函数）等类型构成。

另一个选择是类型系统可否为二阶的。最后，变量类型系统可能比对象类型系统更丰富。

4.4 一致性 对这类型系统应具有的一致性程度有着激烈的争论：类型是否为对象？方法是否为对象？这三个概念是否应区分对待？我们可以在三个不同层次上考察这个问题：实现层，程序设计语言层和接口层。

在实现层，必须决定类型信息是否应作为对象存储，或是否必须实现一个即席系统。这跟关系型数据库系统设计员所面临的是同一问题，他们必须决定是否要存贮模式，是否有某种即席形式。作出此决定应依据性能的需要和实现的难易。但是，不管作出什么决定，都独立于上一层所作的决定。

在程序设计语言层，问题如下：类型是否为语言语义的第一级属性？大部分争论集中于这个问题。可能有不同形式的一致性（语法的或语义的）。这一层上的完全一致性也和静态类型检查不相容。

最后，在接口层，必须作另一个独立的决定。可以提供用户关于类型，对象和方法的一致的视图，尽管在程序设计语言的语义方面它们是不同性质的概念。反之，可以把它们作为不同属性呈给用户，尽管程序设计语言把它们视作相同。必须以人的因素为标准来作此决定。

演绎数据库系统的研究与发展

余金山 （华侨大学）

摘 要

Deductive database systems are now one of the most important directions in the development of database technologies because of their many attractive properties. They have received widespread intensive interest and very great attention. In this paper We first give a brief overview of the development of deductive database systems. Then We motivate as extensively and thoroughly as possible the directions and significant research topics in this field. The status, importance, trends, research methods of these topics, and the relationship among these topics and other disciplines are also discussed.

一般认为，演绎数据库(DDB—Deductive Database) "诞生"于1978年，至今仅十一年的历史。但是由于其突出的特点，演绎数据库自面世以来就引起了人们的极大重视和广泛兴趣。目前DDB已构成了数据库学科中的一门独立分支，成为数据库技术中最受重视的前沿研究领域之一。这些可见诸于近期以来出版的大量有关文献及其国际上举行的各种有关学术会议，包括近几届的VLDB国际会议[1,10,12,18,21]。我国对DDB的研究大约只有六年的时间。但不管在理论方面还是实践方面也都取得了不少鼓舞人心的成果[14,15,16,19]，由此可见演绎数据库的重要性。用著名学者Ullman教授在"数据库

5. 结 论

有些作者，[Kim 88]和[Dittrich 86]争辩说OODBS是一个以面向对象的数据模型为基础的DBMS。如果在更广的意义上对待数据模型的概念，尤其包括面向记录以外的其它方面，则此观点无疑相应于我们的方法。[Dittrih 1986]和[Dittrich 1988]引入了面向对象的数据模型（及随后的OODBS）的分类：若一个模型支持复杂对象，则称其是结构上面向对象的；若提供可扩充性，则称其是行为上面向对象的，一个完全面向对象的模型必须同时提供这二个特性。这个定义也需要持久性、磁盘管理、并发性及恢复；它至少蕴含地假定了大部分其它特性（是否合适，要依不同的类而定）；总的说来，它比我们的方法略为宽松。但是，不管怎样大多数当前的系统和原型没有达到我们的定义规定的所有需求。所以这个分类对比较已有系统和现行工作提供了一个有用的框架。

我们已经给出了一组关于面向对象的数据库系统的特性的定义。就我们所知，本文中给出的金科玉律是目前关于面向对象的数据库系统最详细的定义。特性的选择和对它们的解释出自于对当前系统的分析和实现的经验。对面向对象的数据库设计、实现和形式化的进一步体验将无疑会修改和完善我们的观点（换言之，如果将来你听到我们中的某位作者赞扬当今的定义请勿吃惊）。我们的目标仅提出一个具体的意见以引起科学界的争论，批评和分析。这样我们最后的规则是，你应该怀疑金科玉律。

参考书目（略）

〔王宗斌 王 珊 译校自 Proceedings of the first deductive and object-oriented database, Japan, Dec. 1989, pp40-pp57〕