专家数据库系统的未来方向

Michael Stonebraker 和 Marti Hearst

摘　要

本文为专家数据库系统应用提出一些可能的体系结构，并断言只有以紧密耦合为基础的体系结构才是普遍适用的一种结构。还探讨了规则系统与DBMS相集成的研究方向，认为目前规则系统的优化技术没有得到足够的重视。最后，论述了专家数据库系统的若干特性。

一、引　言

本文中我们从体系结构设计与规则实现两方面讨论专家数据库系统将来可能的发展方向。第二节以我们所想象的一个应用例子来说明专家数据库管理系统的必要性。本文其它部分的讨论均以该例为基础。

第三节讨论构造如第二节所描述的专家数据库系统可能要采用的各种体系结构。本节首先从规则管理系统与数据管理系统之间的松散耦合开始讨论，在分析了这种体系结构的严重不足之后，讨论了规则管理与数据管理紧密耦合的可能性。

第四节讨论用嵌入式规则系统研究DBMS的两种方法。有些研究者致力于研究递归性的规则系统的有效解决方法，这样就自然要构造可处理递归查询的DBMS查询优化器。另一些研究者则更热衷于研究不出现递归的更一般的规则系统并谋求高效地发现大规则集中规则"点火"的时机。最后讨论了这两个方向的重要性及其可能的商用研究。

第五节讨论判定大规则集中规则"点火"时机的可能技术。我们讨论了基于定理证明器、索引技术与数据库中物理标记的各种方法。结论是没有令人满意的选择方案，我们更热心于其它可能的机制。

最后，第六节探讨了专家数据库系统应用的各种特性。我们说明了如何把嵌入式规则管理系统用于专家系统任务的各个方面，并描述了如何用POSTGRES规则系统实现这些任务。然后我们指出，可以采用紧密耦合在数据管理系统中完整实现简单专家系统，而更复杂的专家系统可以以协同方式编写，使数据管理系统与嵌入式推理机及专家系统外壳协同工作以实现专家数据库应用。我们最后得出的结论是，专家系统外壳与嵌入式规则管理系统之间的关系是一个值得进一步研究的领域。

二、一个专家数据库应用实例

考虑这样一个计算机程序，它旨在给一个人指出从一特定地理区域内的任何起点出发到达某一终点的方向。这种程序对汽车出租公司是很有用的，因为它们必须给出从机场到旅客终点的方向。对那些不熟悉地区的行驶车辆也很有用。事实上，目前商业界正在开发具有我们将要描述的性能的简单系统。

所要的程序将在数据库基础上研制，它包括所涉及的地区街道图。对每一相邻结点对，数据库必须存储：

沿着该路段的旅行时间（可能按天计算）；

沿着该路段的合法门牌号；

是否为单行街道；

有无停车标志；

路面情况。

象旧金山海湾地区这样的地理区域，比如说有五百万人口、两百万合法住址与二十万条路段。虽然这一数据库大达数十兆乃至上百兆字节，很难压缩进主存结构中。而且，如果把地理区域扩大点就不在行驶目标车辆包括在内，则该数据库会变得更大。在后一种情况下，我们不能再存贮街道图而必须存贮地形图。因此，这种应用从本质上讲是一个数据库问题。

此外，我们可以通过编写简单启发式算法（例如，该算法可以首先试着寻找到达高速公路最近立体枢纽的最佳路线）来解决这种应用。文献[Pea84]讨论了这一类算法。然而用规则驱动的求解方法解决这一问题似乎更自然。例如，在海湾地区，所有居民可以定期使用下述规则：

为了从伯克利某地到旧金山某地，可以先从当前位置到海湾大桥，再从海湾大桥到达目的地。

这一规则规定从伯克利到旧金山的任何好路线都必须经过海湾大桥。我们大多数人会用成百甚至上千个这种规则来通过该局部地理区域。另外，一般规则也常有例外，例如：

为了从伯克利北部到达南部，除星期六足球赛外，由山麓往南（山麓经过体育场）。在星期六足球赛时由Hearst往西到shattuck，再由shattuck往南。

假定我们希望能使用数千个规则及其例外求解这样的应用，就可以控制象旧金山海湾地区那样的一个区域内的航向。

我们将把专家数据库应用定义为上述必须与大型数据库及大型规则库交互的应用。下一节我们将讨论可以支持这类应用的各种可能的体系结构。

三、专家数据库应用的体系结构

对专家数据库应用我们可能采用的第一种体系结构叫做松散耦合。这里我们可以用诸如OPS5、KEE、ART等专家系统外壳来编写这种应用程序。应用逻辑、服务表示以及航向控制规则都由这种子系统支撑。为此，专家系统外壳得管理该规则库。由于专家系统外壳是程序设计环境而不是数据库系统，所以它们不能存贮我们的应用所需的地图与合法地址。因此，许多专家系统外壳都进行了扩充以支撑外部数据管理系统的调用。KEE连接器[AbW86]就是支撑这种外部DBMS访问的一个产品例子。因此，该应用的数据库部分由另一类软件系统管理。这两种子系统是松散耦合的，因为外壳对数据库服务的调用方式与其它DBMS用户相同。

这种体系结构有一些严重的缺点。首先，规则库要驻留主存。因此，如果外壳所占据的地址空间释放掉，则规则库也不复存在。如果应用使规则发生了变化，那么这些变化不会被保存，除非手工将它们保存起来。尽管这一特性在本应用实例中没有什么问题，但在许多情况下规则必须进行动态变化。此外，规则库也不是共享的，如果一个用户改变了某个规则，那么其他同时作业的用户是不会得到有关变化的消息的。

另一个严重的缺点与动态数据有关。假定外壳从数据库中抽取事实而后事实的值又发生了变化。这可能发生在以下两种情况：1)在外壳执行了一个需要相当长时间的复杂推理后；2)外壳为求得更快的性能在推理期间把事实快速缓存在主存中。在这两种情况中的任何一个情况下，外壳都维护了一个数据库对象的高速缓存器，高速缓存器的一致性必须得到保证。唯一可用的技术是，在事务处理中运行所有数据库抽取程序，然后只有当所要作的推理完成后才结束该事务处理。这将保证在推理期间把所抽取的数据库对象封锁起来。这种途径在推理机完成之前不允许更新所抽取的对象，它对性能有很大的影响。另一方面，为得到新抽取值外壳可能定期询问数据管理系统。这一途径要求即使当所要数据项没有变化时也要让外壳没有必要地运行DBMS查询子系统。这对性能也有很大的影响。因此，对于外壳必须处理动态变化事实的应用来说松散耦合结构肯定有问题。

最后一个问题是所谓的不可分割的应用。假定在用户所希望的推理完成之前外壳必须进行查询以取得整个事实库。在此情况下，外壳高速缓存可能很大而应用产生的性能却低劣，因为这需要整个地址空间来作高速缓存。

由于这些缺点，许多DBMS研究者一直在研究把规则系统与数据管理系统集成起来的方法。我们称这种途径为紧密耦合。用这种体系结构，DBMS不仅要管理数据库也要管理规则库。在这种情况下，规则库被自动地共享并且保持不变。此外，DBMS还可以很容易处理动态数据，仅当所需数据有实际变化时才唤醒规则。最后，由于DBMS适合于管理大批量数据，自然对不可分割数据也没有什么特别的问题。因此，在理论上松散耦合的缺点可以通过把规则系统放在DBMS中得以克服。下一节将讨论有关这方面已有的两种途径。

四、各种紧密耦合方法

对DBMS规则系统有两种主要方法。我们用例子说明第一种方法。试考虑有关祖父母的标准PROLOG程序，例如：

```
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
parent(Joe,Sue).
parent(Sam,Bill).
parent(Sam,Joe).
```

显然，这些父母亲事实可以放到传统关系DBMS的PARENT关系中：

```
PARENT(older,younger).
```

用于定义祖父母的规则可以用关系视图描述，即：

```
range of P,P1 is PARENT
define view grandparent(P.older,P1.younger)
  where P.younger=P1.older
```

然后，有关祖父母的查询

```
retrieve(grandparent.older)
where grandparent.younger="Sue"
```

可以用传统系统中的视图算法处理成：

```
range of P, P1 is PARENT
retrieve(P.older)
  where P.younger=P1.older and P1.younger
    ="Sue"
```

所产生的查询程序可以按传统的方法进行优化。因此，关系视图已支撑了若干种规则。然而，第一种方法拓广了DBMS提供的支撑，可以表示更复杂的规则。例如，考虑规则：

```
ancestor(X,Y):-ancestor(X,Z),parent(Z,Y)
```

该规则是线性递归的，因为同一子句在规则的两边各出现了一次。在这种情况下，对应于该规则的视图定义为：

```
define view ancestor(ancestor.older,parent.
    younger)
  where ancestor.younger=parent.older
```

因此，该视图定义包含了对自身的引用因而是递归的。对该视图的一个查询

```
retrieve(ancestor.older)where ancestor.you-
    nger="Sue"
```

必须翻译成如下语句：

```
retrieve into temp(parent.older)
  where parent.younger="Sue"
append*to temp(parent.older)
  where parent.younger=temp.older
```

第二个语句是一个传递闭包查询的例子，这种查询是以带一个“*”的单个查询语言命令来表达的，它表示在逻辑上该命令应反复执行直至不再有效。在优化传递闭包查询时要作大量的工作，可参见[IoW86, Ros86]。

更一般地，我们可以表达涉及一般递归的规则。例如，下一规则用于定义共同祖先：

```
common-ancestor(X,Y):-ancestor(Z,
  Y),ancestor(Z, Y)
```

其中，定义的右边不止一个递归子句，有关共同祖先的查询显然要比涉及传递闭包的查询难得多。文献[BaR86,HeN84,SaZ87, Ull85]中对有效地解决涉及一般递归的查询作了研究。

因此，对于DBMS中规则管理的第一种方法，其主要研究工作是放在以有效求解上述各种规则为目标的线性递归与一般递归

上。第二种方法针对非递归规则，下面以标准EMP关系：

EMP(name,salary,manager, age, desk)

来进行说明。在该关系中，前四个域是传统的数据，我们着力讨论最后一个域desk(办公桌)。许多公司都详细建立了一套允许谁使用哪一种办公桌的规则。因此，该域基本上由一组规则确定，这些规则一般写在公司的职员手册中，有的甚至记在主要雇员的头脑中。如果我们只把办公桌一栏作为普遍数据存贮，那么我们就得不到限制许可值的规则。相反，我们要存贮一组能确定办公桌值的规则。

我们现在讨论POSTGRES规则系统，它能自然地支撑这种规则定义。POSTGRES提供的查询语言叫POSTQUEL，它主要借用了其前驱QUEL。它的主要扩充是在用户定义运算符与函数方面，并提供了处理时间与继承性的设施。该规则系统完整地使用了这一查询语言。特别地，所有POSTQUEL命令都可以通过前置一个关键字转换成规则。目前，我们提供的关键字有always、never与onetime。例如，下一命令将Mike的办公桌置为Joe的办公桌：

```
replace EMP(desk=E.desk)
from E in EMP
where EMP.name="Mike" and E.name
="Joe"
```

在该命令运行时，会对Mike的办公桌作适当的更新。为了把该命令转换成规则，我们可以在该命令之前附上关键字always，即：

```
always replace EMP(desk=E.desk)
from E in EMP
where EMP.name="Mike"and E.na-
me="Joe"
```

从语义上看，修改后的命令似乎在不停地运行。该命令的实现遵从两种策略之一：提早求值或推迟求值。

如果采用第一种求值策略，在Joe收到一张新办公桌时，POSTGRES将唤醒该命令并把该变化传送给Mike。我们称之为提早求值。因此，只要它所读的任一数据项被修改时，该命令就被唤醒。规则求值的第二种策略是在Joe收到新办公桌时什么也不做，而是处于等待状态，直到某个人需要Mike的办公桌。此时，POSTGRES不是给该人一个已存贮的值，而是运行规则的修改版，从Joe的记录中取得实际数据项。我们称这种策略为推迟求值。POSTGRES的做法是针对不同的规则在提早求值与推迟求值两种策略之间自动选择。

当然，各个规则集可能相互制约。例如，假定有第二个规则把Joe的办公桌置为Sam的办公桌：

```
always replace EMP(desk=E.desk)
from E in EMP
where EMP.name="Joe"and E.name="Sam"
```

如果这两个规则都提早求值，那么将会产生正向链接控制流，因为当Sam得到一张新办公桌时，第二个规则将点火，而后第一个规则再点火。另一方面，推迟求值对应于反向链接，因为对Mike办公桌的请求会引起对Joe办公桌的请求，再引起对Sam办公桌的请求。因此，POSTGRES自动在反向链接与正向链接控制流之间选择。

对于该规则系统值得注意两点：首先，可以允许规则相冲突。例如，我们可以有一个关于所有35岁以上雇员都得到一张木制办公桌的规则，即：

```
always replace EMP(desk="wood")
where EMP.age>=35
```

在Mike或Joe已过35岁而Sam还不到的情况下，这一规则给这两个雇员的办公桌与前面规则给的不同。POSTGRES处理这种情况的方法是，让用户为每一规则指定一优先权，当冲突发生时使用具有高优先权的规则。

最后，假定用户指定了会引起规则系统循环的规则，例如：

```
always replace EMP(salary=1.1*E.salary)
from E in EMP
where EMP.name="Mike"andE.name="Joe"
always replace EMP(salary=1.1*E.salary)
```

```
from E in EMP
where EMP.name="Joe"and E.name="Mike"
```

其中Joe的薪水比Mike多百分之十，而后者又比Joe多百分之十。如果选择提早或推迟求值，POSTGRES通常会进入无限循环。但是，POSTGRES只保存了以前规则启动情况的线，它检查其中规则启动情况并检查前面调用过的规则是否在同一状态下被唤醒。如果有，就中止当前事务处理从而中断无限循环。

这一策略不仅中断了无限循环，也中断了某些非循环情况。例如，如果在上述每一规则中增加一个形如

```
and EMP.salary<5000
```

的子句，那么规则将在有限次的迭代之后终止。然而，POSTGRES不能识别这种事实并中止薪水的请求者(推迟求值)或薪水的更新者(提早求值)。

用POSTGRES可以写出一组完整的办公桌分配规则，如下所示：

```
always replace EMP(desk="steel")
where EMP.age>=35
always replace EMP(desk="wood")
where EMP.age<35
```

这是一般规则的例子，它指明35岁以下的雇员得到一张木制办公桌而35岁以上的雇员得到一张钢制办公桌。然而，一般规则总有些例外。假定Bill是一年青的雇员，他应得到一张木制办公桌，而Sam是一个年长的雇员，他拒绝使用木制办公桌。最后，假定Joe已被允许使用与Sam相同的办公桌，Mike已被允许使用与Joe相同的办公桌。这些例外可表示如下：

```
always replace EMP(desk="wood")
where EMP.name="Bill"
always replace EMP(desk="steel")
where EMP.name="Sam"
always replace EMP(desk=E.desk)
from E in EMP
where EMP.name="Joe"
and E.name="Sam"
always replace EMP(desk=E.desk)
from E in EMP
where EMP.name="Mike"
and E.name="Joe"
```

考虑下面对EMP关系的查询：

```
retrieve(EMP.desk)whereEMP.name="Joe"
```

尽管所有六个规则都可产生这一查询的答案，但我们只需要运用第五个规则。这将产生一个新的查询：

```
retrieve(EMP.desk) where EMP.name="Sam"
```

这里所有六个规则也都适用，但我们要求使用第四个规则，它产生"steel"作为该查询所要的答案。

这一规则集可以刻画为三个性质：

(1)由于存在例外，该规则基本上是不一致的。因此，必须有处理这种情况的机制可供使用。

(2)规则有许多，但没有一个是递归的。

(3)度量与雇员办公桌有关查询的主要性能是有效地判定哪个(或哪些)规则必须"点火"。尽管许多规则都适用，但没有几个能实际使用。

我们称具有这三个性质的规则系统为平凡规则(mundane rule)。

注意，平凡规则(如上面讨论过的办公桌分配规则)可以是专家系统应用的一部分，也可以是用于加强标准数据库维护的一组语义约束。因此，平凡规则对专家系统与标准数据库用户同样是很有用的。而且，许多应用都有线性递归规则(如像祖先的定义)。这方面的例子有用于递归确定构成某个对象的所有成分的查询(部分展开查询)以及用于启发式搜索树结构的AI程序。然而，我们还没有遇到需要作更复杂的一般递归查询的数据库应用。由于我们感到研究工作应与实际用户需要成比例，所以希望看到对平凡规则集做更多的工作，而对一般递归的工作要少些。

五、规则点火技术

我们认为，支撑平凡规则系统的DBMS效率的关键是只对那些实际运用的规则点火。有三种途径似乎可加以探索：1)定理证明；2)索引技术；3)标记技术，下面我们逐个进行讨论。

当进入诸如

```
retrieve(EMP·desk)
    where EMP .name="Joe"
```

的查询时，DBMS可以提供一种定理证明器，后者能提供如下服务。对每一规则R，使用限定Q(R)，定理证明器将判定

(EMP.name="Joe")intersect Q(R)

是否为空。任何这样的规则都可以立即被丢弃，不能丢弃的规则则被点火。因此，重点可以放在怎样有效地建造这种定理证明器上，文献[Bot86]中讨论了这方面的研究。我们对这种途径是否可行并不乐观。首先，定理证明器将所有涉及办公桌的规则都按顺序排好以处理上面的查询。如果有1000个这种规则，那么对它们的相继求值将是一个大瓶颈。其次，它很可能正像能胜任"点火"规则一样，查看该规则是否适用而不试图证明它没有用。而且，就办公桌规则而言，为了断定哪个一般规则适用，也有必要检索Joe的年龄。于是，定理证明器还需要数据库中额外信息来作这一工作，从而降低了性能。

第二种途径是建立规则中的谓词索引。基本思想是建立一个数据结构来存贮每一规则R的限定Q(R)。然后，当交给系统一个用户查询时，限定便进入到索引中，并且在这一过程中会找到其限定与用户的限定相重叠的规则。这种索引本质上是多维的，也是R树的推广[Gut84]。此外，RETE网[For81]就是一种索引系统，它长期用在正向链接专家系统外壳中。这一途径的变种或许可以适用于数据库。

最后一种选择是基于标记的途径。其基本思想是，在DBMS的执行系统中实际运行规则的限定，并把所有存取或要更新的数据项做上标记。这种标记以后可以辅助确定哪些规则要点火。就办公桌这一例子而论，各种规则要给雇员的办公桌数据项设置标记，因为规则打算对该数据项规定一个值；同时也给雇员的姓名或年龄设置标记，因为这些是规则要读取的数据项。

因此，如果某个用户要读取某一特定雇员的办公桌数据项，那么运行时系统可以发现在该数据项设置的标记并只唤醒那些特定的规则。这为规则启动提供了很好的粒度判定。文献[SHH87]中提出了这方面的一个建议，而在[SSH86]中给出了一个比较标记系统与索引系统性能的一个分析模型。

在POSTGRES中实现标记的经验告诉我们，这一工作是非常复杂的。这种复杂性加重了设计者与实现者的负担，我们担心这一途径最终能否生存下去。另一方面，我们还未看到过这样的谓词索引模式，它的一般性足以适合一个规则系统所需要的全部谓词。例如，在我们看来，把规则中的谓词限制成仅仅涉及一个关系是不合理的。最后，定理证明技术也没有显示出大有希望。因此，我们认为一种新思想可能会作出很大贡献，我们鼓励研究者在这方面作些尝试。

六、专家数据库系统特性

从研制诸如POSTGRES等专家数据库系统的经验可以深入地了解超大型专家系统的构造。下面第一小节，描述怎样用POSTGRES规则实施专家系统的任务。第二小节说明基于规则的数据管理系统所起的另一种作用，我们称之为协同式体系结构。在协同式体系结构中，系统起高级应用程序的智能数据操纵设施的作用。

6.1 POSTGRES作为专家系统 在众所周知的专家系统外壳中，OPS5是与POSTGRES规则系统最相像的一个。OPS5用一个"工作存贮器"来存贮其插入和删除会触发规则库中规则的数据项。如前所述，如果工作存贮器中所需的数据量超出可用空间，那么就必须把工作存贮器放到磁盘中。在POST-

GRES中，必须通过对某关系的更新来模拟工作存贮器的概念，因为仅当查询触及由规则对关系设置的标记时规则才能启动。这个限制很重要，因为差不多对所有反向链接任务的可变汇集的维护都要求在一个规则调用到下一规则调用期间将该汇集保存住。为了在POSTGRES中保存汇集，必须把它们写到中间关系中。

举一个列车时刻表的例子。当两列或多列列车都在争夺一公共资源(如铁路中同一路段)时，就需要用优先权规则来决定将该资源给哪一个。确定列车优先权有如下一些因素：列车种类(客车、易腐品列车、军车)、列车是正常调度还是"加班"、旅行方向等。这可以有许多方法：到旁轨、到同一铁道的另一组铁轨、整个地走另一铁道或路线、停在铁路交叉口。另外，由于解决两列列车的冲突可能会造成与第三列列车的又一次冲突。可以定义用于确定何时发生冲突以及怎样解决冲突的规则。例如，假定我们有如下启发式(带前缀"$"号的名字是变量)：

```
(1)if both $train1 and $train2 have $arc1 listed in their next-move attribute and
  if $arc1 is of single track type and
  if $train1 is heading toward $train2
  then $train1 conflicts with $train2
(2)if $train1 is first class and $train2 is second class
  then $train1 is superior to $train2
(3)if $train2 is in conflict with $train1 and
  if $train1 is superior to $train2
  then $train2 yields right-of-way to $train1
(4)if $train2 yield right-of-way to $train1 and
  if $train2 is heading toward $train1 and
  if $train2 has $arc1 listed as its next-move attribute and
  if $siding1 extends from $arc1 and
  if $siding1 is not occupied
  then $train2's next-move is to pull into $siding1
```

比如说现在要确定34次列车的下一次运行。规则系统试图满足规则(4)的前件，这意味着必须满足规则(2)与(3)的前件，这又要求要在数据库中找到规则(1)前件的条件。在POSTGRES中，如果规则(1)得到满足，其作用是在中间关系CONFLICTS中添加与$train1及$tlain2相对应的列车车次。规则(3)在CONFLICTS与RANKINGS关系中探查以找到两组相匹配的列车车次。如果这能成功，那么规则(3)的后件为真，其结果写在YIELDS-TO关系中。最后，如果规则(4)的其余条件得到满足，那么原始目标就达到了，也就确定了34次列车的下一次运行。于是，中间关系(或基本关系中的特定域)必须用于保存可变汇集。这使得POSTGRES中专家系统的规格说明比在某些专家系统外壳中更麻烦。

专家系统任务通常需要灵活的控制结构以适应各种求解方法。许多规则任务最好分成子目标组，其中有些必须在另一些子目标之前完成。例如，在本模拟例子中，"find-conflict"、"determine-superiority"与"find-yield-action"就是"next-move"的子目标(实际上，这些子目标中每一个都关联着许多规则)。OPS5没有显式地提供面向目标的控制结构；建立在OPS5基础上的R1系统[McD82]为了分离各子目标任务在其规则中放置了"上下文标记"。类似地，POSTGRES规则管理系统没有子目标达到的概念，故必须模拟子目标的编排工作。

子目标编排涉及在STATUS关系的域中设置"当前目标"与"当前状态"标记。作用于某个子目标的每一规则在其限定(前件)的某个子句中都包含有一个子目标状态指示器必须设定的约定，以便规则点火。当某个系统状态表明子目标已达到或有关工作已挂起时，就会在适当的域中放置一个指示"完成"或"挂起"的标记。顶层目标确定整个控制流。

下面这个POSTGRES例子使用了前一例中(3)的启发式。第一个规则是，在当前子目标为"find-yield-action"时把让路列车的名字放到YIELDS-TO关系中。第二个规则

是，当对这一规则无事可做时把当前子目标的状态更新为"achieved"，第三个规则把当前子目标的状态由"find-yield-action"改为"next-move"。

```
always append YIELDS-LTO (yielder=C1.
                train2, yieldee=C1.train1)
from C1 in CONFLICTS, R1 in RANKINGS
where STATUS.current-goal="find-yield-
                                action"
and  C1.train1=R1.train1
and C1.train2=R1.train2
and R1.status="superior-to"
priority=8
always replace STATUS(current-stats="ach-
                                ieved")
where STATUS. current-goal= "find-yield-
                                action"
priority=1
always replace(current-goal="next-move",
            current-status="in-progress")
where STATUS.current-status="achieved"
    and STATUS.current-goal="find-yield-
                                action"
```

注意，指示子目标是否已完成的规则的优先权比还在实际运行子目标的规则要低。于是，只有在所有与该子目标有关的其它规则都已点火后它才能点火。另一方面，这需要相当数量规则的同步，从而使求解复杂化。

我们认为，将来如果要建造集成式规则管理系统，那么应使这种同步更容易表示。

6·2 **协同式体系结构** 许多困难的数据密集型问题并没有定义很好的规则集，因而不能用上述专家系统技术来求解。通常，这一类问题(如环境影响分析与天气预报)是通过充分运用数据库的大型单个程序来解决的。基于规则的数据库是一种有效的工具，可以把这种应用中的数据密集型运算与其余部分的分析分离开来。我们称紧密耦合式规则管理系统与应用程序之间的关系为协同式体系结构。

例如，我们不可能建立一些达到"判定兴修水利是否对环境有害"这一目标的显式规则，因为我们不能很好地理解确定这一查询结果的因素。而规则管理统系可以执行一系列局部规则，标志数据库中某一部分出现的变化。如，假定对某些大型水文数据集(地下水深度、降雨量、河流数据等)定义三个不同的规则集。其中一个规则集可以设计为每隔一段时间标记某一地区地下水位的降落情况，另一个规则集监控河流水量的变化，而第三个规则集则报告降雨趋势。这种经过预处理的信息被送给分析程序，随后试图从中找出相关性，从而避开了低级数据分析任务。把数据管理任务从这样的程序中分离开来可以使分析任务更成功。此外，以这种方式来分割计算可以达到实际的并行性。

规则管理系统辅助分析程序可以用两种技术：数据"筛选"与数据"监理"。在应用筛选式时，规则从大量数据中抽取出最重要的信息并"忽略"无关的数据。例如，考虑由某一特定地区许多点的地下水深度构成的水文数据集。有时，这种数据是由不同机构用不同测量标准或技术收集的。假定已知下述启发式：

Agency1的数据比Agency2的数据更精确；

Ageicy1在时间区间T没有进行测量；

在四个特定子地区，Agency2比Agency1所作的测量要多。

完成该数据计算的应用可能要求只使用该地区各点的最精确的数据。为在数据库一级实施这些启发式约束而定义的规则可筛掉不希望的数据点，从而降低从DBMS传送到应用程序的信息量。

数据监理可用于标记动态数据集的长期趋势或偏差。一个应用程序如果使用了当出现变化时就要报警的标准数据管理系统，那么就必须不停地查询数据库的相关变化。如果使用的是具有触发能力的数据管理统系，那么就不必轮询。应用程序可以在一个关系的某个域中存贮一个过程并定义当有关条件出现时执行该过程的规则。

现在再回过头来讨论列车时刻表问题。

复杂对象数据库研究：历史和现状

石桥林 （吉林大学计算机系）

摘 要

本文简要回顾了复杂对象数据库的发展历史，比较全面地介绍了国际上当前正在开展的研究工作，尤其是对复杂对象模型的形式化理论研究工作做了重点介绍。本文最后所列的文献可作为读者进一步了解这方面工作的指南。

本文将复杂对象数据库的研究分为两个阶段：1986年以前为早期，1987年以后为近期。这样划分的主要依据是复杂对象数据库研究的发展水平，同时也是为了便于叙述。对每一阶段的工作，都将从复杂对象模型的形式化理论研究及复杂对象数据库系统的建造这两个主要方面进行介绍。

1. 早期研究工作(1986年以前)

数据库技术中，关系模型的产生和发展大大促进了计算机信息管理在很多领域中的应用，但是，随着应用的深入和范围的扩大，传统关系模型的局限性也日益突出，主要是以下三个方面：

(1)基于于第一范式(1NF)的关系模型，即平关系模型(flat relation model)的数据抽象能力不足，在表达复杂对象数据时，语义不直观，语义信息有丢失，而且结构不灵活。

(2) 在诸如工程数据管理、CAD／CAM、办公室信息系统(OIS)等许多领域中，需要增加数据库的语义内容，以使更多的信息能以可用的形式包容在数据库中，而不是分散在访问和操作数据库的各种程序中。但是在平关系模型数据库中要实现这一点是不可能的。

(3) 面向对象的程序设计也对数据库提出了面向对象管理数据的要求，以消除程序设计语言(包括非面向对象的程序设计语言，如 Pascal、C、等)与数据库管理语言及数据库数据结构之间的不匹配。

复杂对象模型正是为克服平关系模型的这些缺陷而产生的，最初是彼此独立地出现在一些应用领域中，在术语上并不一致，例如语义数据库模型(SDM)[65]、非第一范式关系(¬1NF)[48,63]、格式模型(format mod-

---

系统可以"通告"铁路某部分何时成为瓶颈，并可以建议调度机构减少该地区的车辆。一个简单的方法是定义一些规则，统计在一段时间有多少列车通过一特定区域或车站等。还可以定义其它一些规则，标记何时一个区域的统计数比其余区域高得多。

这些例子说明了运用规则管理系统作为应用程序的智能数据操纵工具的技术。所讨论的应用程序可以作为专家系统外壳。这样，有些规则可以对DBMS规定，而另一些规则可由外壳实施。在这两个系统之间研制最好最小的接口是一个值得进一步研究的有趣课题。

参考文献(略)

〔徐宝文译自《proceedings from the second International Conference on Expert Database Systems》, The Benjamin/ Cummings publishing Company, 1989 声钟仲源校〕