计算机科学1994Vol. 21. №. 3

知识库查询的固有低效性

朱扬勇
（复旦大学计算机科学系　上海200433）

摘　要　This paper introduces the concepts of knowledge-base system and the scale of knowledge-base's application, analyses the method of knowledge-base query evaluation, and presents several obstacles during this evaluation. All of them cannot be well treated by current hardware techniques and software methods. Therefore inefficiency of knowledge-base queries are inherent today. Finally several research directions of the query optimization of knowledge-base systems are given.

关键词　Knowledge-base, Logic Program, Query Optimization.

1. 引言

知识库研究的重要内容之一是查询优化技术的研究。从七十年代后期开始，经过八十年代，到目前为止，知识库查询优化方面已经取得了大量研究成果，开发了许多有效的查询优化算法。例如：Semi-Naive 求值[1]、Magic_Set 重写技术和 Counting 方法[2,3]等等。这些技术的应用，大大提高了知识库的查询效率。尽管如此，知识库的查询效率仍不能令人满意，这是知识库系统不能实用的主要原因之一。

关于知识库系统，目前有两个典型的定义。一个是 J. D. Ullman 提出的[4]。他认为：一个知识库系统是具有下面两种特征的逻辑程序设计系统。

①有一个既作为查询语言又作为宿主语言的描述性语言。

②支持数据库系统的主要功能，如：大批量数据的高效存取、数据共享、并发控制及错误恢复等。

另一个定义是 D. H. Warren 给出的，他认为：一个知识库系统"能够有效地处理中等规模的知识库（由三千个谓词、三万条规则和三百万个事实组成，总存储量为30M 字节）的逻辑程序设计系统"。

总而言之，我们认为：知识库系统是一个逻辑程序设计系统，并能有效地处理（包括数据库技术的应用）一定规模的知识库。这里我们强调"一定规模"，而不是具体地指出"30MB"的存储量，是指知识库的内容在内存中放不下，必需要放到外存储器由数据库系统或采用数量库技术来管理。我们以下的讨论也是基于这一点的。事实上，如果一个应用问题的知识库可以全部装入内存，则现有的查询优化技术是不重要的，Prolog 也能很好地处理，并不需要数据库

种情形（LDL，DRL，CORAL 都用第二种形式）。

综上所述，我们认为：知识库语言是一种既作为查询语言又作为宿主语言的逻辑语言（基于 Horn 子句的语言）。我们之所以不再提描述性或非过程性，是因为逻辑语言本身就是控制较少的（或非过程）语言。

5. 结论

我们有以下结论：

①描述性程序设计语言只是一个可望而不可及的目标。适当减少一个语言的控制机制可以方便用户、缩短应用程序。但是描述性语言是计算不完备的，不能作为独立的程序设计语言。

②逻辑语言是描述性语言的合适形式。由于逻辑语言隐含了分支和循环结构，以及 Skolem 标准型的顺序无关性，其描述性语言比较容易实现（如 Datalog）。

③知识库语言要想完成宿主语言的工作，描述性条件首先要放弃。由于逻辑语言实现演绎推理方便并且控制较少，所以逻辑程序设计语言作为知识库语言是有益的。

长期以来，描述性一直阻碍了知识库系统的实用化，人们不得不在知识库语言中加入控制机制[6,7,10]。这一现象的发生源于对描述性概念的理解和 Ullman 对知识库的定义。本文在得到上述三个结论的同时，对描述性概念加以澄清，定义了描述性、非过程性和过程性等概念，并对知识库语言进行了切合实际的描述。（参考文献共10篇略）

技术的支持。

本文分析研究了知识库查询的求值方法、其求值过程和现有的查询优化技术，考虑到知识库系统的应用问题规模，我们认为有三个影响查询效率的因素是知识库系统所固有的，是现有方法和技术所无法解决的。这些固有因素是：多关系连接、递归深度和 IDB 谓词数量。

2. 知识库的查询求值及其时间复杂度

不同于一般的逻辑程序设计系统（如 Prolog），知识库系统依靠数据库技术或数据库系统的支持。因此，一般地，知识库系统都采用自低向上"每次一个集合"的求值模型（优于 Prolog 的"每次一个元组"的求值模型）。而具体做法是采用 Naive 或 Semi_Naive 算法。Semi_Naive 是对 Naive 的一个改进，使得自低向上求值时本次循环的求值操作不重复以前循环的工作，从而减少了工作量，但并没有减少循环次数。因此，为了讨论方便（且不会影响分析结果），我们仅对 Naive 算法进行分析。

下面我们给出 Naive 算法[4]：

输入：一组具有 EDB 谓词 $r_1, r_2, \cdots, r_k$ 和 IDB 谓词 $p_1, p_2, \cdots, p_m$ 的 DATALOG 规则，还有一组关系 $R_1, R_2, \cdots, R_k$，它们是 EDB 谓词的值。

输出：从这些规则得到的方程的最小不动点解。

方法：首先建立每条规则的方程 $P_i = EVAL(p_i, R_1, R_2, \cdots, R_k, P_1, P_2, \cdots, P_k)$。然后，将每个 P_i 初始值置为空集，并且反复使用 EVAL 获得 P_i 的新值。当没有更多的元组添加到任何 IDB 关系中时，得到的 IDB 谓词的内容为输出。具体步骤如下：

```
for i:=1 to m do
    Pi:=;
repeat
 for i:=1 to m do
      Qi:=Pi;
        for j:=1 to m do
      Pi:=EVAL(pi,R1,…,Rk,P1,…,Pm);
until Pi=Qi for all i
output Pi;
```

其中，EVAL 是 P_i 的每条规则所对应的 EVAL_RULE 的集合（或关系）的并，而 EVAL_RULE 是根据子目标关系，计算出规则体关系，然后再投影到 P_i 定义的域上，得到 P_i 关系。如对于规则

$r(X,Y):-p(X,X_1),p(Y,Y_1),s(X_1,Y_1)$，

$EVAL\_RULE(r,P,S)=\Pi_{[X,Y]}(P(X,X_1)\infty P(Y,Y_1)\infty S(X_1,Y_1))$。

从 Naive 算法的描述，我们不难看出，Naive 算法的时间复杂度由循环次数（repeat_until 和 for i:=1 to m）和 EVAL 运算所决定的。repeat_until 的循环次数虽然和规则的排列次序（或 EVAL 作用于 IDB 谓词的次序）有关，但它主要是由逻辑程序 LP 的递归深度决定的，LP 的递归深度取决于具体的规则和 EDB 关系的性态。第二个循环 for i:=1 to m 的循环次数就是 IDB 谓词的数目。最后，我们来看 EVAL，EVAL 的执行时间取决于规则的子目标数目和每个子目标关系的大小。这就是 Naive 算法的（或广义地说是自低向上求值或知识库求值）的三个主要时间因素。我们分别称为：递归深度因素（用 Tr 表示）、IDB 谓词数量因素（用 Ti 表示）和多关系连接因素（用 Tm 表示）。当然，Naive 算法的时间复杂度为 O(Tr * Ti * Tm)。

3. 知识库查询的固有低效性

现在，我们回顾 D. H. Warren 对知识库系统规模的描述：三千个谓词、三万条规则和三百万个事实。且不说递归深度，仅 Ti * Tm 就已经非常可观了。何况递归深度可能会达到事实的数目——三百万个。在第一节，我们强调了"一定规模"，是想放弃这些庞大的数字，仅从最简单情形来讨论。"内存放不下"（这是知识库系统规模的基本考虑）简单地可看作为每个 EDB 关系在内存放不下。一般地，这还将导致每个 IDB 关系也在内存放不下。由于"内存放不下"，才需要数据库技术来管理外存数据。这样，多关系连接操作必需要借助于外存才能完成。这就是说，执行一次 EVAL 操作至少要8次访问外存*)。这样总的求值时间（或者说总的访问外存的次数）为 Tr * Ti * 8。设一个应用问题有10个 IDB 谓词、递归深度为100，并设一次访问外存的时间为10毫秒（当前较好的硬件速度）。于是完成一次知识库查询求值至少需要 10 * 100 * 8 * 0.01 = 80秒钟的时间。可见，对于这样一个小问题，其查询响应时间就已经太长了，更不用说"中等规模的三千个谓词、三万条规则和三百万个事实"的应用了。

另一方面，现有的知识库查询优化方法虽然提高了查询求值的效率，但远没有解决问题。现行的查询优化手段可分为三类：程序优化、规则优化和运时优化。

程序优化是指寻找一个与原来的逻辑程序 LP

*)设 R∞P，由于 R 和 P 都在内存放不下，就要对它们分块处理，要作 $(R_1 \cup R_2)\infty(P_1 \cup P_2)$，所以至少要4次读和4次写操作。

等价且能较快求值的逻辑程序 LP'。已经证明即使是 DATALOG 程序，等价性也是不可判定的[6]。相关的工作还有逻辑程序的递归有界性和线性化研究。有界性研究是指：对于一个有界的 LP，找一个非递归的等价 LP'来取代它；线性化研究是指：对于一个可线性化的 LP，找一个线性的等价 LP'来取代它。同样，有界性和可线性化性也已证明是不可判定的[5,6,9,10]。即现有的研究结果表明这方面的研究是困难的，或者说很难减小一个 LP 的 Tr。

规则重写方面的研究成果是最丰富也是最富有成效的。其典型的代表是 Magic_Set 重写技术、Counting 方法和左(右)线性变换等等[2,3,4]。Magic_Set 方法的关键内容是减少无关事实的产生，从而减小了参予连接的关系，即减小了 Tm。另外一方面，Magic_Set 方法引入辅助关系而减少了规则的子目标数目，但增加了程序的 IDB 谓词数目。而 Ti * Tm 可能会变大也可能会变小，但对 Tr 仍无影响。而 Counting 方法和左(右)线性变换的情形也是一样的。

运时优化是在执行查询求值时尽可能减少重复工作，减少无关元组和冗余元组的产生。如 Semi_Naive、Not_So_Naive 等都是这方面的优秀工作[1,7]。但是这些算法也只是对 Tm 产生影响，而基本上不改变 Tr 和 Ti。

综上所述，到目前为止，查询优化工作的实际应用效果，主要是减小了多关系连接因素 Tm。这一因素是 Naive 算法的核心部分。对多关系连接(即对算子 EVAL)的优化确实能提高查询效率。但是，仅仅减小 Tm 无法使知识库的查询效率达到令人满意的程度。在上面的分析中，我们已经看到，即使将 Tm 降到最低限(仅考虑访问外存的机械操作，而没有考虑每次访问外存时的数据传输量和任何数据之间的操作)，其查询效率也仍然低。所以，象 Magic_Set、Counting 方法和左(右)线性变换等非常优秀的查询优化算法也无助于使知识库查询响应时间降到可以接受的程度。这说明了知识库查询的固有低效性。

4. 结论及今后的研究

由以上讨论，我们得到如下结论：知识库查询的低效性是固有的，是由知识库本身的应用规模所决定的，是现有的硬件技术和查询优化方法所无法解决的。因此，采用当前技术实现知识库系统，其执行效率是低的。

从硬件上看，多处理机系统能较好地执行并行算法，提高查询效率。另外，高速存取的外存储器及大容量内存的开发应用是很有效的。事实上，从第三节的分析，我们看到，硬件技术的革命将是提高知识库查询效率的重要因素。

从软件方面来看，虽然目前已经取得很多成果，并且我们还讨论了知识库查询的固有低效性，但是仍然有很多工作可以做。下面分别加以讨论：

(1)减小 Tr。这是程序优化的研究内容。虽然一般逻辑程序的等价性是不可判定的，并且由此而导致逻辑程序的有界性和可线性化性也是不可判定的，但是我们仍可通过适当排列规则的执行次序，或是找一个递归深度较浅的等价程序 LP 来减小 Tr。理论上，找一个对上述问题可判定的子类也是有意义的。

(2)减小 Ti。硬性减少 IDB 谓词数目会导致与原来的程序不等价。但是，对于某个查询目标，适当去掉一些谓词是可以的。只是这样往往会导致规则的子目标数目的增加，从而降低了效果。由于谓词在查询求值时(计算 EVAL)具有相对独立性，所以开发并行求值算法是可行的。这将使 Ti 减小(甚至为 1)。

(3)减小 Tm，到目前为止的大部分查询优化研究都是在做这一工作，这方面仍然有很多工作可以做，如尽可能减少无关元组产生和访问外存的次数，使得每次执行 EVAL 所需的访问外存次数和时间尽量少。

(4)减小 Ti * Tm，我们已经注意到减小 IDB 谓词数目会增加规则的子目标数目，反之也一样。因此，适当调整二者的数目，可以减小 Ti * Tm。当然，这样做一般会导致程序不等价，但只要查询等价就是可行的。

通过本文的分析，我们知道：即使是当今实现的最好的知识库系统，其查询效率也是低的。这正是知识库系统难以实用推广的主要原因之一。(参考文献共10篇略)