维普资讯 http://www.cqvip.com

在Cache替换策略中的XPath Fragment包含算法

詹 欣 李建中 熊蜀光 王宏志
(哈尔滨工业大学计算机科学与技术学院 哈尔滨150001)

摘 要 在基于查询的XML cache环境中,查询包含算法对于cache替换策略的性能来说是很重要的。XML查询通常用XPath表达式来表达,XPath表达式等价于树模式。本文研究了Xpath查询语言片段的包含问题,实际上我们研究了树模式的一个特殊例子,证明了一个模式包含的声音和完全的PTIME算法。我们也分析了它的时间复杂度,实验结果也证实了我们的分析。

关键词 XML,查询包含,Cache替换策略

Containment Algorithm for XPath Fragment in Cache Replacement Strategy

ZHAN Xin LI Jian-Zhong XIONG Shu-Guang Wang Hong-Zhi
(Department of Computer Science, Harbin Institute of Technology, Harbin 150001)

Abstract In the environment of query-based XML cache, the query containment algorithm is important to the performance of cache replacement strategy. XML queries are usually expressed by means of XPath expressions, which are equivalent to tree patterns. This paper studies the containment problems for a fragment of the XPath query language. In particular, we study a special case of tree patterns, and provide a sound and complete PTIME algorithm for pattern containment. We also analyze its time complexity, and experimental results verify our analysis.

Keywords XML, Query containment, Cache replacement strategy

1 引言

由于XML数据的不断增值,许多分布环境下的应用都需要向远处的XML信息源发查询,以获取所需的数据。为了降低传输延迟,这些应用常常会设置一个局部的基于查询的XML缓存,以保存常用的查询和查询结果,并通过分析与被缓存查询的包含关系,来回答后续查询。也就是说,如果一个新查询能够被某个缓存查询(部分地)包含,那么新查询的(部分的)查询结果可以从本地缓存中获得。查询包含[6~9]已经受到很多研究机构的高度关注。

一种细粒度的替换算法能够更有效地管理缓存空间,因为新查询通常和被缓存查询局部地重叠。这种替换策略会在XML路径结构的粒度上收集用户使用的统计信息。当缓存满时,最少使用的路径所对应的结果文件片段会被替换。在很大程度上,替换策略的效率要依赖包含算法的性能。所以,找到一种正确有效的查询包含算法很有必要。

XPath[1]是一种为部分XML文档定位的语言,可以被XLink[2]和XPointer[3]使用,XQuery[4]也用它绑定变量。XML查询通常使用XPath表达式来表示,所以本文集中研究XPath片断的包含问题。这个片断称为$XP^{([],//,*)}$,包含子轴(/)、后代轴(//)、统配符(*)和分支(或限定词,表示为[…])。它是十分健壮的XPath子集,足够表示XML查询。XPath表达式定义一种遍历XML树的方法。并且,它能够返回从一个(或多个)初始节点起,经过表达式所定义的路径,而到达的节点的集合。由于$XP^{([],//,*)}$表达式与tree patterns(模式树)是等价的,$XP^{([],//,*)}$表达式的包含问题与tree patterns的包含问题就是等价的。

例如,针对图1的XML文档的一条查询"找到至少有一个作者已知的所有书的价格"可以被翻译成下面的XPath表达式:bib/book/[//author]/price。它表示如下的定位过程:从标为bib的根节点开始,通过其标为book的子节点。如果从book节点可以到达一个标为author的节点,那么返回这些book的price。这个表达式和图2所示的模式树是等价的,图中的黑色节点是输出节点。

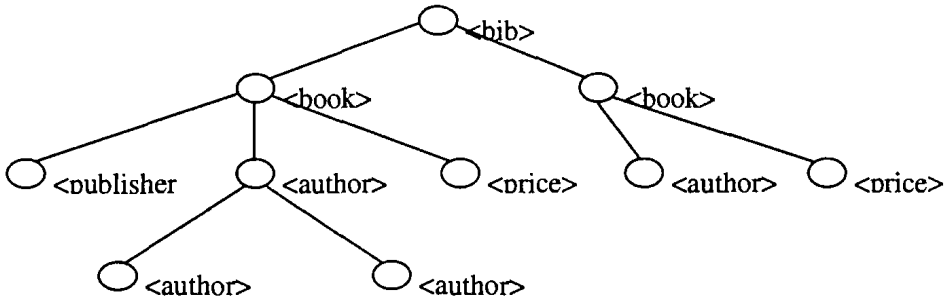

图1 一个XML文件的树型结构

注意,下文中我们用术语path表示从根节点到叶节点的模式树的路径。在本文中,我们的研究对象是模式树的一种特殊情况:一个模式树的每条path上都有输出节点。其中,两条相邻的path既可以有各自的输出节点,也可以共用一个输出

维普资讯 http://www.cqvip.com

节点。例如，图3(a)中，每条 path 自己有一个独立的输出节点，分别标为 B 和 C。图3(b)中，两条 path 共用一个标为 * 的输出节点。

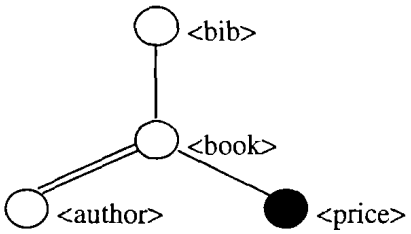

图2　一个模式树的例子

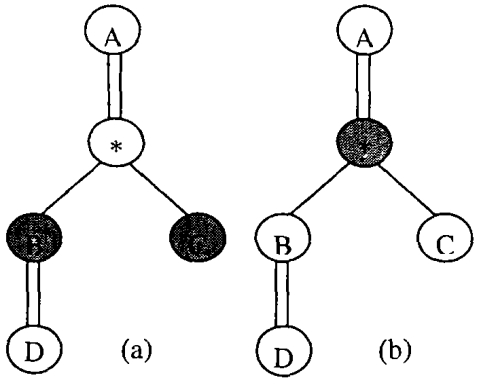

图3　两类模式树

对于这种特殊的模式树，我们提出一个正确的且有效的 PTIME 算法，来检查两个特殊的模式树间的包含关系。我们还分析算法的时间复杂性，并通过实验数据证明我们的分析。

2　相关工作

XPath 子集的包含问题已经受到研究机构的大量关注。早期的实验结果表明，对于 *，// 和[…]中任意两者的组合，包含问题的时间复杂性是 PTIME。在没有后代边($XP^{\{[],*\}}$)的情况下，一个 PTIME 包含算法[6]遵循无环连接查询的经典结论。不包含通配符时，文[7]中找到一个 $XP^{\{[],//\}}$ 的多项式时间的包含算法。对于 $XP^{\{//,*\}}$，文[8]的一个研究结果表明这个子集的包含问题同样是多项式时间的。然而，对于 $XP^{\{[],//,*\}}$ 的包含问题却是 NP 完全问题[9]。

3　预备知识

在本文中，我们将一个 XML 文档看作一棵无序树，节点的标签来自无穷字母表 Σ。我们只研究非递归结构的 XML 树，也就是一条 path 上的节点要互不相同。我们设节点集合为 N，节点标签集合为 Σ。

3.1　树和模式树

一棵树 t 是一个四元组(r_t, N_t, E_t, l_t)，其中

(1) $N_t \subseteq N$，是节点集合，

(2) $r_t \in N_t$，是 t 的根节点，

(3) $l_t: N_t \to \Sigma$，是节点标注函数，

(4) $E_t \subseteq N_t \times N_t$，是连通无环边的集合。

在节点标签集合 Σ 上定义的所有树的集合记为 T_Σ。

一个模式树 p 是一个二元组$\langle t_p, o_p\rangle$，其中

(1) $t_p = (r_p, N_p, E_p, l_p)$是一棵树，

(2) E_p 分为两个不相交的子集 C_p 和 D_p，分别表示孩子边(|)的集合和后代边(||)的集合，

(3) $o_p \subseteq N_p$，是输出节点的集合。

对于我们研究的特殊的模式树，每条 path 上都有一个输出节点。如果 p 的最长路径上有 m 个节点，那么 p 的深度为 $m-1$。例如，图2中模式树的深度是2。

已知一棵树 t 和一个模式树 p，从 p 到 t 的嵌入 e 是一个函数 $e: N_p \to N_t$，使得

(1) $e(r_p) = r_t$，

(2) $\forall (x, y) \in C_p$，在 t 中，$e(y)$是 $e(x)$的子节点，

(3) $\forall (x, y) \in D_p$，在 t 中，$e(y)$是 $e(x)$的后代节点，

(4) $\forall x \in N_p$，如果 $l_p(x) = a (a \neq *)$，那么 $l_t(e(x)) = a$。

模式树 p 的 *models* 是 T_Σ 中能被 p 嵌入的树，p 的 *models* 的集合记为 $Mod(p) = \{t \in T_\Sigma \mid p(t) \neq \varnothing\}$。$p(t) = \{X \subseteq N_t \mid \exists$ 一个从 p 到 t 的嵌入 e，且 $e(o_p) = X\}$。

3.2　模式树的包含

已知两个模式树 p_1和 p_2，当且仅当$\forall$ *tree* t，$p_1(t) \subseteq p_2(t)$时，我们称 p_1被 p_2包含($p_1 \subseteq p_2$)。文[9]已经证明，对任意的模式树 p_1和 p_2，$p_1(t) \subseteq p_2(t)$当且仅当 $Mod(p_1) \subseteq Mod(p_2)$。

3.3　path 的包含

由于 path 是模式树的特例，那么 path 的嵌入和 model 的定义与模式树的相关定义类似。已知一个 path q 和一棵树 t，$q(t) = \{x \in N_t \mid \exists$ 一个从 q 到 t 的嵌入 e，且 $e(o_p) = x\}$。q 的 *models* 是 T_Σ 中能被 q 嵌入的树，它的集合记为 $Mod(q)$。根据文[9]的研究结果，我们定义 path 的包含：已知两个 path q_1和 q_2，$q_1(t) \subseteq q_2(t)$当且仅当 $Mod(q_1) \subseteq Mod(q_2)$。

4　包含算法

到目前为止，我们已经介绍了模式包含的相关定义。在这一章，我们详细研究特殊模式树的包含问题。首先，定理1证明模式树的包含和它的 path 包含的等价性。在定理2中，我们指出如何证明两个 paths 的包含关系。根据以上两个定理，我们提供两个算法。

4.1　定理描述

定理1　已知模式树 p_1，它包含的 paths 是 $q_{11}, q_{12}, \cdots, q_{1i}, \cdots, q_{1n}$，模式树 p_2，它包含的 paths 是 $q_{21}, q_{22}, \cdots, q_{2j}, \cdots, q_{2m}$。那么，$p_1 \subseteq p_2$当且仅当$\forall\ i (1 \leqslant i \leqslant n)$，$\exists\ j (1 \leqslant j \leqslant m)$使得 $q_{1i} \subseteq q_{2j}$。

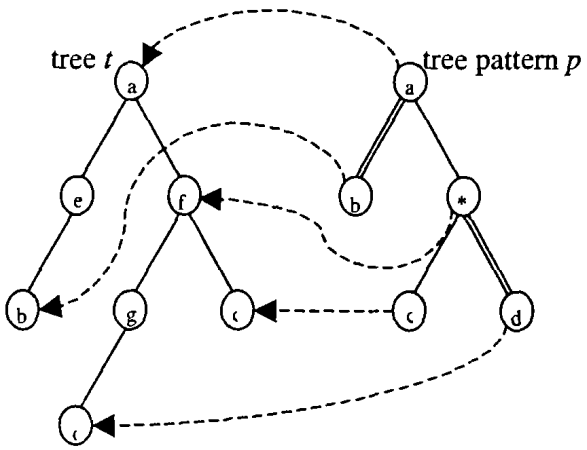

图4　从 p 到 t 的一个嵌入

定理2　已知两个 paths q_1和 q_2，$Mod(q_1) \subseteq Mod(q_2)$当且仅当以下条件都成立：

(1) 除了通配符 * 外，q_2的标签集合被 q_1的标签集合包

维普资讯 http://www.cqvip.com

含，即 $l_{q2}(N_{q2})\backslash\{*\}\subseteq l_{q1}(N_{q1})\backslash\{*\}$。

(2) 对于 q_2 中除 * 外的任意标签 a 和 b，如果在 q_2 中，标为 a 的节点是标为 b 的节点的后代，那么在 q_1 中，标为 a 的节点也是标为 b 的节点的后代，即 $\forall\ a,\ b\in l_{q2}(N_{q2})\setminus\{*\}$，如果 $l_{q2}^{-1}(a)$ 是 $l_{q2}^{-1}(b)$ 的后代，那么 $l_{q1}^{-1}(a)$ 是 $l_{q1}^{-1}(b)$ 的后代。

(3) 用 q_2 中除 * 外的节点作为分界点，分别把 q_1 和 q_2 分割成等量个子段(subpath)，对于由相同分界点分割(不包括分界点在内)的 q_1 的子段 sq_{1i} 和 q_2 的子段 sq_{2i}，设 sq_{1i} 中非分界点的节点个数为 n_1，sq_{2i} 中以通配符 * 为标签的节点个数是 n_2。如果 sq_{2i} 中有后代边(∥)，那么 $n_1\geqslant n_2$。如果 sq_{2i} 中没有后代边，那么 sq_{2i} 中也没有后代边，并且 $n_1=n_2$。

4.2 包含算法

算法1：检查两个 paths 的包含

```
input: paths q1, q2
output: true, if q1⊆q2; false, otherwise
begin
    for subpath sq2i {the iteration proceeds from root to leaf of q2}
        To the non-*-labeled node in sq2i,
        if a node in sq1j matches with it
        then
        Case 1: sq2i has at least one descendant edge
            if n1<n2, return true;
        Case 2: sq2i has no descendant edges
            if n1=n2 and sq1j has no descendant edges, return true;
end
```

算法1的时间复杂性分析：已知两个 paths q_1 和 q_2，它们的深度分别是 d_1 和 d_2，那么算法1的时间复杂性是 $O(\max\{d_1,d_2\})$，因为此算法同步扫描两个 paths。

算法2：检查两个模式树的包含

```
input: tree patterns p1, p2
output: q1i and q2j, if q1i⊆q2j, q1i is a path of p1, q2j is a path of p2
begin
Store each leaf node of p1 in array A1, and its corresponding path in B1;
Store each leaf node of p2 in array A2, and its corresponding path in B2;
for a leaf node ni in A2
    for a leaf node nj in A1
      if ni and nj match
        then check containment between paths qi and qj corresponding to ni, nj;
        if qi⊆qj,
        then output qi and qj;
end
```

算法2的时间复杂性分析：设 p_1 有 n 个叶节点，p_2 有 m 个叶节点，且二者的最大深度是 k，那么 $TC=O(n)+O(m)+O(m\times n)\times O(k)=O(m\times n)\times O(k)=O(mnk)$。

当一个模式树中的叶节点标签互不相同且标签不是通配符时，我们可以提高算法的效率。首先，我们按照字母表顺序对 A_1 和 A_2 中的叶节点标签排序，并相应地对 B_1 和 B_2 作相同的排序。例如，按照字母表顺序，标签 book 应该排在 title 前面，我们称 book 小于(is smaller than)title。然后，我们在 A_1，A_2 中分别使用指针 c_1，c_2，指针初始指向数组的头部，并且随着对叶节点的比较而向前移动。若 c 是一个指针，我们用 $c\rightarrow$label 表示指针当前指向的标签，$c\rightarrow$pos 表示指针当前指向的数组下标值。

算法2的优化：

```
begin
for each leaf node in p1
    Store it in array A1 and its corresponding path in array B1 respectively; (1)
for each leaf node in p2
    Store it in array A2 and its corresponding path in array B2 respectively; (2)
Sort A1, B1 by leaf labels; (3)
Sort A2, B2 by leaf labels; (4)
for the c2→label {the iteration proceeds from beginning to end of A2} (5)
    if c1→label is bigger than c2→label break;
    else if c1→label is smaller than c2→label
        c1 moves ahead to the next element;
    else check containment between two paths in B1, B2 at the position of  c1→pos, c2→pos;
        c1 moves ahead to the next element;
end
```

优化算法的时间复杂性分析：设 p_1 有 n 个叶节点，p_2 有 m 个叶节点，且二者的最大深度是 k。步骤(1)和(2)的时间是线性的。在步骤(3)(4)中，我们使用 *quicksort*[10]，它的运行时间是 $O(n\times\log n)$。核心步骤(5)比较两个 paths 的包含关系，至多循环 $\max\{m,n\}$ 次。已知 paths 包含的时间复杂性是 $O(k)$，那么优化算法的运行时间是：

$TC=O(n)+O(m)+O(n\times\log n)+O(m\times\log m)+O(k)\times O(\max\{m,n\})=O(k\times\max\{m,n\})$。

5 实验

我们为实验输入设计了模式树生成器。它产生的模式树的节点数最大值是5000。它能够控制叶节点的数量和最大深度。在实验中，节点的标签至多包含4个字符。

5.1 不同最大深度时的优化算法的运行时间比较

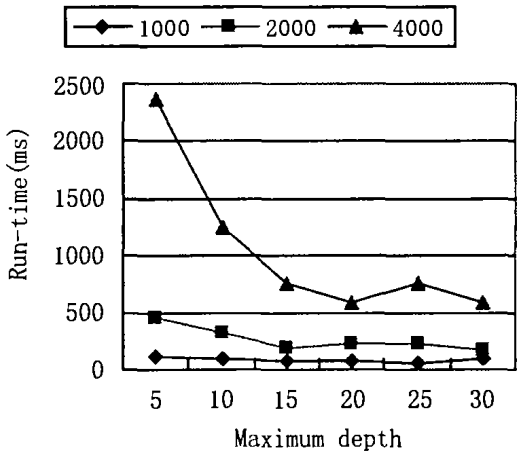

图5 不同最大深度时的运行时间

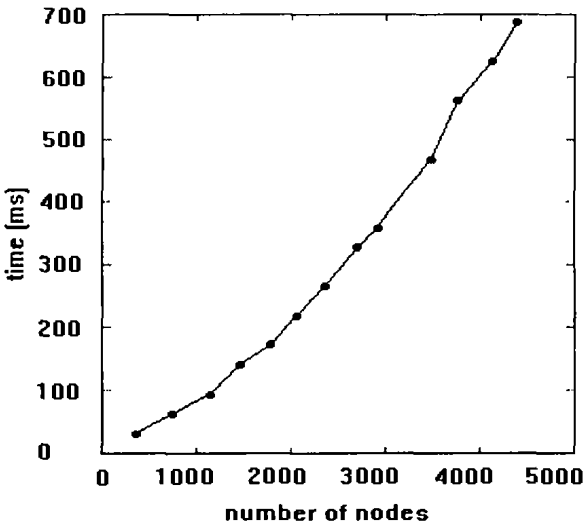

图6 不同节点数时的运行时间

我们使用3个输入数据集合，每个集合由100个模式树组成。各集合中每个模式树的节点数分别是1000，2000和4000。当输入模式树的最大深度由5变到30时，我们比较优化算法的运行时间。图5中，横轴代表输入模式树的最大深度，纵轴代表

维普资讯 http://www.cqvip.com

对于每组输入数据的算法平均运行时间。从图6中可以看出，随着最大深度的增加，平均运行时间逐渐降低。而且，输入的模式树的节点数越多，运行时间降低速度越大。

5.2 不同节点数时的优化算法的运行时间比较

我们令输入的模式树最大深度恒为15。那么，节点个数与叶节点数大体上成比例，节点个数也就与 $\max\{m,n\}$ 成比例。由于 k 是常数，优化算法的运行时间 $O(k\times\max\{m,n\}) = O(\max\{m,n\})$。在图6中，横轴代表输入模式树的节点个数，纵轴代表算法运行时间。随着节点数的增加，运行时间线性增长。图中的实验结果验证了我们的分析。

结束语 在基于查询的 XML 缓存替换策略的应用背景下，本文研究了 $XP^{\{[],//,*\}}$ 的包含问题。特别地，我们的研究对象是一类特殊的模式树。我们提出一个正确完整的 PTIME 算法，以判定这类模式树的包含问题。而且，从理论和实验的角度分析了算法的性能。实验结果表明，即使输入数据量很大时，算法也是有效的。目前，我们在研究当 XML 树有递归结构时的包含问题。这种情况下，path 中的节点可以重复，那么 paths 的包含算法运行时间达到 O(n ^ 2)。

参 考 文 献

1 W3C. XPath 1.0: XML Path Language. http://www.w3.org/TR/xpath, Nov. 1999
2 W3C. XLink 1.0: XML Linking Language. http://www.w3.org/TR/xlink/, June 2001
3 W3C. XPointer: XML Pointer Language. http://www.w3.org/TR/xptr/, Aug. 2002
4 W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, Nov. 2003
5 Chen L, Rundensteiner E A. A Fine-Grained Replacement Strategy for XML Query Cache. In WIDW, MeLean, Virginia, 2002
6 Yannakakis M. Algorithm for acyclic database scheme. In: Morgan Kaufman pubs. (Los Altos CA), Zaniolo and Delovel eds. Proc. of the 7th Conf. on Very Large Databases, 1981
7 Amer-Yahia S, choo S, Lakshmanan L V S, Srivastava D. Minimization of tree pattern queries. SIGMOD, 2001
8 Milo T, Suciu D. Index structures for path expressions. In ICDT, 1999. 277～295
9 Miklau G, Suciu D. Containment and Equivalence for an XPath Fragment. In: Proc. of the 21st ACM SIGMOD-SIGART Symp. on Principles of Database Systems(PODS), Madison, Wisconsin, USA, June 2002
10 Cormen T, Leiserson C, Rivest R, Stein C. Introduction to Algorithm. Second Edition, 2001

(上接第103页)

② $t[TC]\leqslant c$。

这里我们可以发现删除操作仅针对那些主体可以看到所有属性的元组。

(6)调用操作处理策略 调用操作的语句格式如下所示：

EXECUTE funname([*para*1[, *para*2]…])

这里 funname 是存储过程/函数的名字，*para*1、*para*2为存储过程/函数的参数。设 cf 是存储过程/函数的安全级，c 为执行调用操作的主体的安全级，当且仅当 $cf=c$ 时，调用操作被接受。

(7)特权情况操作处理策略 上面讨论的六种处理策略均为无特权情况下的处理策略，我们这里还要讨论各种操作在有特权情况下的处理策略。特权可以用一个五元组表示 $P = (Subject, Object, OPType, BTime, Etime)$，其中 *Subject*、*Object*、*OPType*、*BTime*、*Etime* 分别表示主体、客体、操作类型、起始时间、结束时间，这里客体的粒度最细为表/视图或者存储函数/过程。特权情况处理策略可以理解为当主体有相应特权的情况下，无论主体对特权中指定的客体的操作是否满足对应操作处理策略，该操作都被允许。

4 NDMAC 模型分析

NDMAC 模型是在 BLP 模型和 JS 模型的基础上发展起来的，它是为增强型安全数据库而设计的。在增强方式下，自主访问控制由底层数据库负责，因此 NDMAC 模型中没有自主访问控制相关的描述。这样模型考虑了底层数据库的安全功能，使得 TCB 尽可能小。粒度选择也是 NDMAC 模型的一个重点。模型中明确了客体的最细粒度是属性/元组级。其原因是：我们研究目标是 B2安全级安全，这决定了客体粒度要至少达到属性/元组级；而在增强方式下，元素级控制粒度下的资源消耗和效率损失远远大于属性/元组级粒度下的资源消耗和效率损失，这样为了提高系统性能，控制粒度要尽可能粗。

很多多级关系安全模型通过引入多实例来处理隐蔽信道问题。多实例在带来更高的安全性的同时，也会引起数据的不一致，这样可能会对主体产生误导作用。多实例虽然是现实世界的某些现象的反映，但是总体而言它还是弊大于利，因此 NDMAC 模型未采用多实例。但是模型考虑到推理控制引起的隐蔽信道问题，因此模型中通过引入推理完整性规则来局部消除函数依赖导致的隐蔽信道。由于推理完整性并非强制实现的，模型在查询操作处理策略中通过自动生成的推理约束来进一步消除函数依赖导致的隐蔽信道。

NDMAC 模型扩充了现有安全模型的客体类，明确了该模型中的六种客体：数据库、表、视图、存储过程/函数、属性和元组。此外 NDMAC 模型通过引入隶属完整性规则，使得关联客体的安全级满足一定偏序关系。模型扩充了现有安全模型讨论的操作的范围，模型中讨论了查询、插入、修改、删除、连接、调用这六种操作的处理策略。模型借鉴 JS 模型中的实体完整性，定义了自己的实体完整性规则，使得元组可见时，关键字一定可见。此外模型引入了特权机制以处理特殊情况的需要，这样增加了模型的灵活性和实用性。

结束语 论文的研究成果被应用于国家863计划信息安全领域的研究课题中，实现了一个基于 ORACLE 数据库的安全增强器，其强制访问控制功能已基本达到了 B2级安全要求。如何达到更细的客体粒度而不严重影响系统效率、数据库系统的入侵检测和对抗将成为论文进一步的研究重点。

参 考 文 献

1 Bell D E, Lapadula L J. Secure Computer Systems: Unified Exposition and Multics Interpretation, MTR-2997, MITRE, March 1976
2 Denning. The SeaView Security Model. IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, 1988. 218～233
3 Jajodia S, Sandhu R. Toward a Multilevel Secure Relational Data Model, Information Security: An Integrated Collection of Essays: IEEE Computer Society Press, 1995. 461～491
4 Sandhu R, Chen F. The Multilevel Relational (MLR) Data Mode. ACM Transactions on Information and System Security, 1998, 1(1)