文章编号:1001 - 9081(2006)12 - 2977 - 05

纯 XML 数据库语义缓存综述

塔 娜,冯建华,李国良
(清华大学 计算机科学与技术系, 北京 100084)
(dan04@ mails. tsinghua. edu. cn)

摘 要:介绍了语义缓存、XML 查询语言等概念,对现有的多种纯 XML 数据库语义缓存技术及其实现、技术特点等进行了详细的阐述。在讨论了当前语义缓存的研究中遇到的新问题与挑战之后,介绍了一种语义缓存匹配视图的快速查找算法 U-ViewMatch,并总结了语义缓存的技术路线。

关键词:纯 XML 数据库; 缓存技术; 语义缓存
中图分类号: TP311.132 **文献标识码**:A

Survey on semantic cache in native XML databases

TA Na, FENG Jian-hua, LI Guo-liang
(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084, *China*)

Abstract: The concepts of semantic cache and XML inquiry language were introduced, and a thorough survey and detailed analysis of implementations and technical characteristics of the existing semantic caches in native XML databases were presented. A new fast semantic cache look up algorithm was proposed. New problems and challenges arising from current research were discussed and some suggestions were proposed for future work.

Key words: native XML database; caching technique; semantic cache

0 引言

由于 XML 数据具有自描述特点,可以支持用户自定义的标记,符合 Internet 上数据描述和存储的需求,所以近年来 Internet 上信息的表示和存储越来越多地向 XML 格式迁移,XML 也正在逐渐成为 Internet 上数据表示和数据交换的实际标准。随着其规模和复杂性的快速增长,以 XML 格式表示和存储的数据得到了 Internet 领域和数据库领域研究人员的重视。Internet 上的应用对 XML 数据的查询、定位和获取的需求不断增加,也引发了对 XML 数据进行合理存储和快速查询的要求。

考察查询执行的效果,除了准确地返回符合查询条件的所有数据之外,另一个关键的指标就是查询的响应速度。对于同样的查询条件,返回结果的速度越快则查询执行的效率越高。传统的关系数据库(包括基于关系表、关系代数的分布式数据库)为了提高查询响应速度,除了进行查询语句改写、查询计划优化和基于代价的执行顺序选择之外,数据库管理系统(Database Management System,DBMS)还会有选择地对查询结果进行缓存,把经常被查询的数据(即"热数据")保存在缓存中。当新的查询到来时,DBMS 首先检查是否可以由缓存中的数据进行回答,如果可以,则直接返回查询结果,这样就避免了 I/O 开销较大的磁盘数据读取,从而提高了查询响应速度。

随着对纯 XML 数据库技术的不断深入研究,纯 XML 数据库缓存技术正在逐渐受到越来越多的关注。通过对热数据进行缓存,可以有效地提高数据库的处理效率和响应能力。更重要的,XML 技术是起源于 Internet 网络应用大规模发展的大背景下,XML 数据更多地是与 Internet 上的应用相关,网络数据传输代价给纯 XML 数据库的性能带来了更多的挑战,所以对 XML 数据查询响应时间的要求就显得更加重要。在这种情形下,合理高效地缓存数据,一方面可以大规模地提高数据库的响应速度,减少结果数据的返回时间,缓解网络数据传输带来的一系列性能上的问题;另一方面,缓存数据也可以有效地减少网络传输中不必要的负载。

当前纯 XML 数据库系统的缓存技术研究主要集中在语义缓存上,本文主要对语义缓存的技术进行分析和综述,并指出纯 XML 数据库语义缓存研究的发展方向。

1 缓存技术概述

缓存技术最早是在关系数据库中提出的,这是数据库系统的一个重要功能。把一些在一段时间内不发生改变的数据放到缓存中,当以后需要这些数据时,就不必每次从本地磁盘或服务器读取数据,而直接从缓存中取得,这样就加快了查询响应的速度。在关系数据库中,可以利用实体化视图对关系表进行重新组织,根据用户感兴趣的内容,选择若干关系表中相关的列,定义视图,构成一个逻辑上的表格,并"实体化"地保存到缓存中,称之为"实体化视图"。在不涉及到数据更新的情况下,用户看到的视图内容与实际的关系表完全相同。

1.1 缓存技术研究的主要内容

(1) 缓存数据的选择:为使缓存发挥作用,必须恰当地选择并保存经常重用的数据,而且重新计算这些数据的代价是昂贵的或比较昂贵的;不经常变化的数据也是缓存的候选数据。因此"热数据"的选择是缓存技术研究的一个重点。

(2) 缓存数据的存放:缓存数据实际上是数据库中主数

收稿日期:2006 - 06 - 06;**修订日期**:2006 - 08 - 13 **基金项目**:国家自然科学基金资助项目(60573094);清华大学基础研究基金资助项目(JCqn2005022);浙江自然科学基金资助项目(Y105230);国家重点基础研究发展计划资助项目(2006CB303103)

作者简介:塔娜(1982 -),女,内蒙古呼和浩特人,硕士研究生,主要研究方向:纯 XML 数据库、WWW 下的信息处理; 冯建华(1967 -),男,山西运城人,副教授,博士,主要研究方向:纯 XML 数据库、WWW 下的信息处理; 李国良(1981 -),男,河北唐山人,博士研究生,主要研究方向:纯 XML 数据库、WWW 下的信息处理.

据的拷贝,它可能保存在内存中,也可能以不同的组织形式保存在磁盘上。所以,除了考虑要缓存哪些数据以外,数据缓存的位置也需要考虑。依据数据的存储位置分为两种类型:基于内存的缓存和基于磁盘的缓存。

基于内存的缓存适用于应用程序频繁访问数据的情况。由于对内存的访问开销远小于对磁盘的访问,所以可以减少查询响应时间。磁盘驻留缓存是以磁盘为存储介质的缓存技术,主要面向大文件系统和大型数据库。当面向的应用包含海量数据,或者数据需要离线处理时,以及数据在机器重启的情况下必须保证有效时,磁盘缓存就体现了它的必要性和优势。

(3) 缓存数据的格式:数据在缓存中的组织形式对缓存的性能也有较大的影响。合理优化的缓存数据结构,有助于提高数据的查找、定位以及缓存数据更新的效率。保存在缓存中的数据不单纯是数据库主数据的一个片断(若干相关记录)在缓存中的一份拷贝,缓存中也需要维护一些相关的描述性元数据。比如在关系数据库中,这些描述性数据就可能是某个记录被访问的次数,某组记录是否被修改过且此修改是否需要及时写回磁盘等诸如此类的用来保持数据一致性、完整性以及方便数据维护的信息。

(4) 缓存数据的加载:要想使用被缓存的数据,首先需要把数据加载到缓存中,加载的方式包括"预加载"和"在线加载"两种。前者在应用程序查询之前根据 DBMS 的一些启发性预测规则等指示信息,加载一定数量的数据放入缓存备用;后者则是在应用程序请求数据的过程中,不断调整放入缓存的数据,以便更好地给后续的查询及时提供所要查找的数据。第二种方式也称为"增量式"加载。

(5) 缓存数据的更新:维护一个数据库系统,需要保持事务的原子性、一致性、隔离性和持久性。如果被修改的数据恰好被保存在缓存中,那么缓存的更新,即缓存数据与数据库中相应原始数据的同步,是需要引起注意的。缓存更新的触发事件是数据的更新,对此事件的响应有两种策略:1)先更新缓存,直到相关数据被换出缓存时,再把更新写到磁盘上;2)更新缓存的同时把被更新的数据写回磁盘。这两种方法都各有优劣,适用于不同的应用场合。

(6) 缓存数据的替换:无论是内存型还是磁盘型缓存,其容量总是有限的,当缓存区域没有可用空间时,就需要为新产生的热数据提供空间供其存放,此时必须把一些数据移出缓存。以什么策略、移出哪些数据,移出的数据是否可以被直接丢弃,是否需要与稳定介质中的主数据进行同步等,这些问题不仅影响了数据的一致性,也对缓存的性能和速度造成了相当大的影响。

1.2 缓存的优势

缓存技术的出现对数据库系统的性能有着重要的改善作用,具体表现为:1)利用缓存数据可以有效地减少在进程和机器间传输的数据量;2)在 CS(Client/Server)模式下,通过客户机的缓存处理,当判断本地数据可以回答某一查询时,就不必向服务器提交查询请求,可以减少服务器的查询处理次数和网络负载;3)如果是内存型缓存,则有效的缓存可避免多次重复访问磁盘带来的较大的 I/O 开销。

2 纯 XML 数据库的语义缓存技术

2.1 语义缓存

语义缓存是近年来数据库缓存领域出现的一个新方向。所谓"语义缓存"是指把用户的查询与结果数据对应起来,一起保存到缓存中。这样,缓存中保存的就不只是一些单纯的数据,由于加入了对数据的描述性语义信息,因此可以此为依据提高缓存数据的利用率,进一步提高缓存对数据库性能的改善作用。

文献[1]介绍了纯 XML 数据库的一种语义缓存技术。这种语义缓存技术基于 XQuery 查询的包含和重写。通过查询改写,重用已缓存的查询结果,用以回答新查询。文献[1]重点解决两个问题:XQuery 的包含和重写,缓存实现及替换策略研究。文献[1]指出,支持关联访问、语义区替换以及缓存一致性的缓存管理子系统是一个语义缓存系统必要的组件。在查询包含和重写方面,文献[1,2]研究了两个主要问题:1)如何判断两个查询是否存在包含关系,2)如何根据已有查询重写新提交的查询。在语义缓存的实现策略上,文献[1]针对 XQuery 查询中的 XPath 条件,构造了查询区和 XPath 表,用于组织缓存中的数据。

与文献[1]不同,文献[3]针对 XPath 查询语句进行缓存。基于 XPath 的一种匹配算法,利用匹配矩阵,判断新查询 Q 的 XPath 是否包含缓存中的某个 XPath 视图 V,再根据二者查询条件的差,构造补偿查询 CQ(Compensation Query),以便从 V 中求得 Q 的结果。由于是基于 V 的条件对 Q 进行改写,所以 CQ 的复杂度比 Q 低,而且只需 V 的结果即可给出 Q 查询的结果。整个查询执行过程中开销比较大的地方在于判断查询的包含关系以及补偿查询的构造,但是这些运算性的 CPU 开销相比从磁盘读取数据的 I/O 开销而言仍然很小,这便是语义缓存中语义查询和实体化视图为回答用户提交的新查询而带来的优势。

2.2 语义缓存与查询语言

数据自身的结构特点、查询语言的特性等都会对语义缓存的组织结构造成影响。在 XML 查询语言方面,目前比较广泛使用的查询语言有 XQuery 和 XPath。其中作为主流查询语言的 XQuery 是程序化语言与描述化语言结合的产物,其最新版本是由 W3C 在 2005 年 11 月发布的 1.0 版工作草案[4]。XQuery 以 FLWR (For-Let-Where-Return)表达式为基本构架,以复杂的长路径表达式为核心语句。查询表达式中不但规定了查询语义,而且限制了查询的执行顺序。XPath 的最新版本是由 W3C 在 2005 年 11 月发布的 2.0 版工作草案[5],它是 XQuery 的一个关键组件。单独的 XPath 路径本身就是完全有效的 XQuery 语句,两种语言有相同的公共数据模型。

复杂的 XPath 或者 XQuery 查询语句对缓存结构造成了影响。比如,在现有的 XML 语义缓存技术中,缓存中可能保存若干查询语句及其对应的结果节点集,如果新的查询只有一部分条件与缓存的查询语句匹配,就产生了所谓的"交叉"问题,是绕过缓存直接到数据库中对此新查询进行求解,还是设计算法利用缓存中的数据对新查询进行求解,在两种方法中做出选择并对二者的查询代价及效率进行评价,都是需要深入探讨的问题。

目前的纯 XML 缓存结构,针对不同的数据储存粒度、查询语言及其子集,有不同的实现方法。既有针对 XQuery 中 XPath 查询条件的 XPath Table[1],也有直接匹配 XPath 查询语句的匹配矩阵[3],还有把 XPath 表达式改写为字符串再进行缓存匹配的处理方法[6]。

2.3 语义缓存的组织方法

2.3.1 语义区:利用单个已缓存查询的结果

在文献[1]中语义缓存是以 XQuery 查询及其对应结果的 XML 视图为基本单位进行组织的。在缓存中为每个查询及其结果数据集构造一个语义区,并在每个语义区中为其对应的查询构造一个查询描述符,描述查询的语义信息,包括查

询语句等。根据查询描述符,语义缓存系统就可以确定某个查询是否可以全部或部分(与 2.2 节中提到的“交叉”情况对应)地利用缓存中的视图来回答。查询描述符之间的逻辑包含关系就意味着查询结果集的包含关系。

令 Q 表示一个查询,V 表示缓存中的一个视图。在缓存命中即 V 可回答 Q 时,直接返回 V 中满足 Q 条件的数据即可;若 V 可部分回答 Q,则重写 Q,分为两部分,一部分称为“探测查询”,用于从 V 的语义区中取得满足 Q 查询的结果数据;另一部分称为“余查询”,用于从服务器或本地数据库磁盘上取得不在 V 结果数据集中但是满足 Q 条件的数据记录。通过分解原始查询 Q,余查询的结果规模较之 Q 会有所减少,这样就有效地减少了网络负载和服务器的处理负担。

这种分解查询的处理方法在关系数据库中已有类似的研究。文献[7]就是针对关系数据库缓存技术的研究,根据输入的查询条件生成一个探测查询(注意,这里的“探测查询”与文献[1]中的不同),通过分析查询,生成两个查询执行计划:本地计划和远端计划,前者只需要利用本地的缓存表格即可求解查询,而后者则需要完全利用远端服务器的数据。利用这种方法求解一个查询,数据要么全部来自缓存,要么全部来自远端服务器,本地计划和远端计划是两个独立的查询执行计划。当缓存没有命中时,必须从数据库中取得全部结果。相比之下,文献[1]提出的方法,可以进一步提高缓存数据的利用率,同时减少不必要的网络传输开销。

语义缓存的好处在于,通过维护与查询相关的语义信息,可以有效地跟踪用户的访问习惯,而用户查询的信息往往是前后相关的,某些数据可能被频繁地访问,通过语义信息就可以把这些数据合理地组织到一起,快速响应用户的查询。而利用关系数据库的缓存技术,其中缓存子系统在处理一个新查询的时候,通常只涉及这些数据的物理信息,如元组号、页号等,不能更好地利用用户的访问特征对缓存进行改善。

语义区的提出是 XML 语义缓存与关系数据库缓存组织方式的一个重要区别,在 XML 数据库中,语义区用来保存 XML 查询语句及其结果,这是 XML 语义缓存的一种基本的组织形式。在文献[1]中,以 XQuery 查询为单位进行缓存,好处是缓存语义区的结果集较小而且视图/查询对的匹配更加精确,但是相应的缓存查找过程则较难优化,维护缓存的代价也较高。

语义缓存组织中的另一个问题是,由于 XML 数据自身的树状层次结构特征,缓存的查询结果集如果仅包含 XQuery 或 XPath 的结果节点元素,那么对于结果已经包含在缓存中的某些新查询,由于缺失了树根节点到结果节点的路径等元信息,缓存的查询结果就有可能无法用来回答新的查询。

XML 文档片段:

```
<Pathway name = "PA1" >
  <Reaction name = "RE1" >
    <Enzymes >
      <Protein name = "PR1" EC# = "1.0.0.1"/ >
      <RNA name = "RN1"/ >
    </Enzymes >
  </Reaction >
  <Reaction name = "RE2" >
    <RNA name = "RN2" >
  </Reaction >
</Pathway >
```

假设缓存中已经保存了 XPath 查询 Q1 = “//RNA”及其结果,即两个 RNA 元素节点 < RNA name = “RN1”/ > 和 < RNA name = “RN2” > ,但是却无法利用 Q1 的结果来回答新查询“/Reaction/RNA”,因为无法判断 Q1 结果的 RN1 和 RN2 的父亲节点类型是否为“Reaction”,所以文献[9]提出存储一些附加信息,比如从根到结果节点的路径信息,用于判断缓存数据的上下文关系,决定其可用性,这样就可以进一步提高缓存数据的利用率。

对这种方法还有进一步的改善,文献[3]也指出,缓存的视图中仅仅保存查询结果对利用缓存匹配新查询是不够的。文献[3]提出了缓存中可用的 4 种有用的辅助信息:Copy、Data、Path 和 Ref。其中,Copy 是指实际的 XML 数据片断,可以理解为查询结果,保存在缓存中,用来回答与缓存查询相同的新查询;Data 是指“有类型的值”,是 XML 文档中某种类型值的实例,用于辅助判断包含比较条件的查询语句;Path 是指从树根节点到某个节点的全路径,包含此路径上所有节点的标签,用于快速判断包含后代轴(//)以及通配符(*)的 XPath 查询是否可被缓存数据回答;Ref 是指节点之间的引用关系,可以用于进行节点的导航。

有了这 4 种辅助信息之后,通过对 XPath 查询进行重写,即可根据匹配算法判断某个新查询 Q 是否可以与缓存的某个视图 V 匹配了。在视图匹配判断的过程中,如果 Q 的条件等于 V 的条件,或者比 V 的条件更加严格,则结果为匹配,否则结果为不匹配。匹配判断的过程是遍历 Q 和 V 的比较过程,递归地判断 Q 对应的查询树的每个分支、每个节点是否可以在 V 中找到与之对应的映射分支或节点,并返回最终匹配结果。

如果视图匹配成功,即可构造 Q 的补偿查询 CQ,构造过程分为两步:1)放松条件,也就是把 Q 中 V 和 Q 都具备的公共条件去掉,因为 V 的结果数据是完全满足这种条件的,所以这种条件对 V 的结果数据而言完全是冗余的;2) 查询树优化,利用一些启发式规则,如去掉过滤条件、去掉查询的某些“步”等,对 CQ 进行优化。生成 CQ 之后,将其作用于 V 的结果,就可得到 Q 的查询结果。

文献[3]中方法的不足在于,它没有考虑当缓存中存在大量的视图时的视图查找算法,导致缓存查找成为系统性能的瓶颈。同时视图与查询匹配的条件要求视图完全严格包含查询,这样也使缓存的利用率不高。

2.3.2　模式集:组合利用多个已缓存查询的结果

对于如何组织缓存进而高效地回答新查询这一问题的研究,文献[1~3]的主要思想都是利用单个已缓存查询的结果来回答新的查询。若缓存的查询可部分或全部回答新查询,就可以达到利用缓存数据提高性能的目的。文献[9]为了解决已缓存的若干查询不能组合起来回答新查询的问题,对存入缓存的查询进行了一定的限制和修改,要求放入缓存的 XML 数据必须能够用一个模式集抽象地表示。此模式集要满足两个条件:1)缓存的 XML 数据片断必须是导出此片断的 XML 数据库的一个有根子树;2)缓存的 XML 数据片断必须能够完全回答模式集中的所有模式(此处一个“模式”可以理解为一个 XPath 查询语句)。其中,第一个条件与文献[3]中的“Path”数据的作用有着异曲同工之妙,其好处都在于可以保留文档树中从根到叶子节点的完整路径信息,用于判断新查询与已缓存查询的包含关系。第二个条件用来判断已缓存查询的结果是否可以组合起来回答新查询。对于缓存中的模式集而言,它的生成是由多个查询的查询条件不断累加而成,也就是通过“增量式”的维护,把新查询的模式归并到已缓存查询的模式集中。

假设有两个 XPath 查询 Q1 = “a[c]//b”和 Q2 = “a[c]/d/b”,如果已经缓存了 Q1 的结果及其模式集,则缓存 Q2 时,

因为其结果集是 Q1 结果集的子集,所以就把 Q2 的模式加入 Q1 的模式集,同时把 Q2 的数据加入到 Q1 的结果集中(由于 Q2 的结果包含于 Q1 的结果中,而 Q1 的结果已经被缓存,所以不必将 Q2 的结果放入缓存)。这样,放入缓存的每个“查询”(实际上是多个查询不断组合而成的一个广义“查询”的子树)的结果,都满足它所代表的模式集包含的所有模式。假设一个新查询 Q′的模式集 P 中的任意模式 p_i 都是组合模式集 Q 的有根子树,那么 Q 的结果就可以用来回答 Q′。

模式集的方法可以有效地利用已求值的多个视图的结果,但是由于其对放入缓存的查询要求的条件较多,故应用的范围不如上一节中提到的几种方法广泛。

2.3.3 字符串匹配:利用单个已缓存查询的结果

在前面介绍的语义缓存组织中,视图的表示基本上都遵循 XML 的格式,语义区或者模式集、结果集的表示都可以归类为 XML 数据片断。另一方面,关系数据库的技术经过了几十年的发展,有很多成熟的技术可以利用,文献[6]就是基于关系数据库,利用关系表实现 XML 数据的语义缓存和 XPath 的实体化视图。通过定义“查询/视图可回答性”,把复杂的树操作简化为字符串操作,进行视图与查询的匹配,提出了一种性能良好的缓存查找算法。

文献[6]中的“视图”由 XPath 表达式定义,“查询”则包括 XPath 表达式以及一些有限制的 XQuery 语句。如果存在某个查询 C,在视图 V 上求值,返回的结果与查询 Q 在数据库中求值的结果完全相同,则称 V 可回答 Q,即 V 对 Q 具有“可回答性”。这里,把查询 C 称作“构成查询”。

假设缓存中已经有若干视图,查询树与树形结构的视图匹配是利用缓存回答查询的开销所在。为了提高匹配效率,便于建立索引,文献[6]根据几个规范化规则,把树形结构的模式改写成字符串:对查询轴上的每个节点,依次调用规范化节点的子过程,直到整个查询树都被改写为字符串为止。“查询轴”是指在查询模式树中从根到结果节点的路径;在此路径上的节点称为“轴节点”,不在此路径上的其他节点称为“谓词节点”;轴节点的个数表示了“查询深度”。这样,视图匹配过程就简化为字符串匹配过程,加快了利用缓存数据回答查询的速度。

视图的字符串改写算法能够保证同样的查询模式树会被改写为同样的字符串,这对缓存匹配是必要的。对于树形结构的缓存匹配而言,视图可回答性的判断要检查两个条件:1)V 的 k 前缀与 Q 的 k 前缀是同构树。其中 k 是 V 的查询深度,“k 前缀”是指去掉查询轴上第 k 个节点之后的所有节点后剩下的查询,例如一个查询/a[p/q/r][b]/x/y//z,其“2 前缀”就是/a[p/q/r][b]/x;2)对 V 的第 k 个轴节点处的任意谓词 view-pred,都可找到 Q 的第 k 个轴节点处的一个谓词 qry-pred,满足 view-pred qry-pred。

将视图和查询模式改写为字符串之后,视图可回答性的判断就简化为:1)V 的长度为 k 的前缀等于 Q 的长度为 k 的前缀,其中 k 为 V 的查询深度;2)V 的深度为 k 的谓词包含在 Q 的深度为 k 的谓词集合中。

文献[6]的巧妙之处在于利用字符串简化查询的匹配过程:对于一个深度为 n 的查询 Q,从 n 到 1 变化参数 k,从缓存中选择具有 k 前缀的视图,穷举生成包含这些 k 前缀的所有谓词字符串,如果这些字符串包含在 Q 的 k 前缀中,则缓存命中,查找过程结束;否则,继续查找,直至找到一个匹配的视图或者匹配过程结束,匹配失败。

在上述缓存匹配过程中,可能出现问题的地方在于穷举这一步。文献[6]认为穷举不会造成太大的开销,对于有 n 个节点的树,生成所有包含它的树的时间复杂度为 2.62n。但是这种分析建立在查询树的规模不大、深度不深的基础上(文献[6]中作为例子的查询主要是类似于“/a/b[z]/c”和“/a/b[w][@y="str"]”这样比较简单的查询)。对于目前受到越来越多重视的分支模式这种分支条件比较复杂的查询,穷举所有可能包含某个分支模式的树,其时间复杂度将会大大提高。另外,随着查询深度的增加,由于穷举带来的时间开销使得利用缓存数据来回答查询带来的优势荡然无存。

2.3.4 不完全树:利用单个已缓存查询的结果

针对语义缓存的结构,文献[10]提出利用树型结构表示每个“前缀选择查询”,这种树型结构称为不完全树,它包括两部分:1)数据树,即查询对应的结果 XML 文档片断;2)类型树,即缺失信息,也就是此不完全树的数据树不满足的条件的集合。经过若干次连续的查询回答和不完全树的构造之后,一个不完全树将包含已缓存查询的结果数据以及这些数据不满足的条件。与之相反,文献[9]的方法则是保存数据及它们满足的条件。文献[10]中的“前缀选择查询”是指从树根开始遍历 XML 文档树,读入一定深度的元素及数据节点等判断条件的一类查询,是对查询的一种抽象。查询的匹配基于模式匹配。

文献[10]是较早研究 XML 数据缓存的,它基于以下几个假设条件:1)数据树是无序的;2)树中节点的编号是永久的;3)使用简单的树形查询语言,不考虑递归路径以及数据链接。在这样的条件下使用不完全树,可以在多项式时间复杂度内判断已有的缓存数据是否可以完全回答一个新查询。但是不完全树存在“指数爆炸”问题,也就是说,尽管可以在多项式时间复杂度下增量地维护不完全树,但其占用的空间却与“查询—回答”对的序列大小成指数增长。

为了解决指数爆炸问题,以便在多项式空间复杂度下维护不完全树,文献[10]提出了两个方法:1)增加一些附加查询,以减小不完全树的不确定程度;2)放松不完全树的条件,牺牲一定的精确性以换取空间性能的改善。通过这样的限制,不完全树的维护就可以维持在多项式的空间复杂度之内了。在满足文献[10]提出的假设条件的情况下,以不完全树对缓存进行组织是有优势的。但是,当查询树具有分支、存在可选子树、节点编号可变以及原文档树有序时,指数爆炸的问题就会出现,就不可能在多项式的时间和空间复杂度内增量地维护缓存了。

3 纯 XML 数据库缓存技术的研究方向

缓存技术对于数据库的性能改善有着非常重要的作用,对纯 XML 数据库而言更是如此。如果对纯 XML 缓存技术的关键问题进行更加深入的研究,则有可能在纯 XML 语义缓存技术上取得突破性的进展。

3.1 缓存的组织结构问题

缓存组织是缓存技术的基础问题,由此引发的问题包括:采用什么样的单位对缓存进行组织,不同粒度的缓存单位对缓存性能有何影响,如何根据已提交查询之间的相似度对类似的查询进行合理的归并后再放入缓存以提高数据的集中程度、避免多余的空间开销等。这些都是在缓存组织结构方面可以深入探讨的问题。

缓存的替换策略是缓存组织中需要注意的另一个问题。

传统的 LRU、LFU 等替换算法已经得到了广泛的应用,但它们对 XML 数据的某种缓存组织结构是否可以进一步优化,也是值得研究的问题。

3.2 热数据的选择

在缓存中放入什么数据是研究缓存技术要特别关注的一个问题。可以考虑采用数据挖掘的方法,针对设计好的缓存组织结构来挖掘频繁的查询模式。这里的一个关键问题是 XML 数据的表示方法,也就是说以什么样的形式来表示 XML 数据,才能做到既保留其树形结构的祖先—后代关系,又便于运用数据挖掘的技术来挖掘频繁的查询模式。

3.3 缓存匹配算法

对于新提交的用户查询,如何快速地确定已缓存的查询结果是否可以回答此查询,是要重点研究的问题。对 XQuery 和 XPath 的实体化视图、字符串表示等多种数据表现形式,可能需要有不同的缓存匹配算法来回答用户的查询。

总之,XML 语义缓存技术可以在三个方面展开研究:1)语义缓存的组织结构;2)热数据的合理判断和选择算法;3)查询与缓存的快速匹配算法。

4 一种快速的匹配视图查找算法

本节介绍一种快速的缓存匹配视图查找算法 U-ViewMatch[14]。U-ViewMatch 基于 XPath 查询主路径进行匹配视图的查找。在这里"主路径"是指从 XPath 查询的根节点到结果节点的所有节点构成的路径(不包含路径上各节点的谓词节点)。在语义缓存中以 XPath 查询及其结果数据集为单位构成每个视图,缓存的组织结构如表 1 所示。其中 No. 为每个视图的序号,BP 为视图对应的 XPath 查询的主路径,UDFTS-out 为 XPath 查询的唯一序列化表示,RS 为查询结果集。

表 1　语义缓存组织结构

No.	BP	UDFS-out	RS
1	PRT	P(0) D(1) T(2) R(1) T(4, $)	<T>1... </T> <T>2... </T> ...
2	PDT	P(0) D(2) D(1) T(3, $) U(3)	<U>1... </U> <U>2... </U> ...
3	P * T	P(0) R(1) M(2) * (1) T(4, $)	<T>1... </T> <T>2... </T> ...
4	SPF	S(0) K(1) P(1) F(3, $) M(4) Q(1)	<F>1... </F> <F>2... </F> ...
...			

进行缓存查找时,算法 U-ViewMatch 首先进行查询主路径与视图主路径的匹配检查,返回主路径与待求解查询主路径匹配的视图,这样就可以快速缩小搜索空间,提高缓存查找速度和效率。然后再根据 UDFTS-out 序列的匹配进行补偿查询的构造(所谓"补偿查询"是指将其作用于匹配视图求解的结果与原查询求解结果完全相同的这种查询),回答用户查询。

判断主路径匹配的复杂性远小于判断整个查询匹配的复杂性,此外在主路径查找过程中 U-ViewMatch 算法还采取了索引技术用于加快查找过程,所以其查找复杂度远小于基于树形结构的查找方法以及文献[6]中的视图查找算法。

以基于树遍历的缓存查找[3]为例,假设缓存中存放了 N_v 个视图,每个视图对应的 XPath 查询的平均大小(即查询模式树的节点个数)为 S_v。令 r 表示查询树中主路径节点个数(设为 N_{BP})与所有节点个数 S_v 之比,$r = N_{BP}/S_v$,则 $r \leqslant 1$,且大小为 S_v 的查询树中的主路径节点个数为 $N_{BP} = S_v \cdot r$,r 会随着主路径节点占查询树节点总数比率的减小而减小。

因为基于树查找的算法要遍历整个缓存区,所以其平均时间复杂度为 $O(0.5 \times N_v \cdot S_v)$。而算法 U-ViewMatch 的索引查找过程的时间复杂度为 $O(1)$。若索引查找成功,假设索引项指向 k 个视图($k \ll N_v$),根据上面的假设,每个视图主路径的平均长度为 N_{BP},则缓存区查找过程的时间复杂度为 $O(k \cdot N_{BP}) = O(k \cdot S_v \cdot r)$。由于 $k \ll N$,故 $k \ll 0.5 \times N$,而 r 是一个小于等于 1 的参数,所以算法 U-ViewMatch 的时间复杂度远小于基于树遍历的缓存查找算法。

算法 U-ViewMatch 存在可以改进之处:当根据主路径匹配关系返回多个候选视图时,可以对这些视图进行排序,主路径长的优先用于构造补偿查询,这样就可以降低补偿查询的复杂度,从而降低整个查询的执行代价。

5 结语

本文主要对纯 XML 数据库的语义缓存问题进行了讨论和分析,介绍了语义缓存、XML 查询语言等基本概念,对当前纯 XML 数据库的语义缓存组织的四种方法进行了详细的阐述,并分析了这些方法的优劣和可能存在的问题。给出了纯 XML 数据库的语义缓存必须要深入研究的几个关键问题,并介绍了作者研究的一种语义缓存中视图的快速查找算法 U-ViewMatch,其性能优于文献[6]中提出的目前缓存查找性能最好的字符串方式匹配的查找算法。缓存查找之后补偿查询代价的判断以及如何构造执行代价最小的补偿查询也是今后可以进一步研究的方向。

参考文献:

[1] LI CHEN. Semantic caching for XML queries[EB/OL]. http://www.wpi.edu/Pubs/ETD/Available/etd-0129104-174457/, 2006.

[2] LI CHEN, RUNDENSTEINER EA. ACE-XQ: A cache-aware XQuery answering system[A]. WebDB 2002[C]. 2002.

[3] BALMIN A, OZCAN FÄ, BEYER KS, *et al*. A framework for using materialized XPath views in XML query processing[A]. VLDB 2004[C]. 2004.

[4] XQuery 1.0[S]. http://www.w3.org/TR/xquery/.

[5] XPath2.0[S]. http://www.w3.org/TR/xpath20/.

[6] MANDHANI B, SUCIU D. Query caching and view selection for XML databases[A]. VLDB 2005[C]. 2005.

[7] ALTMEL M, BORNHOVD C, KRISHNAMURTHY S, *et al*. Cache tables: paving the way for an adaptive database cache[A]. VLDB 2003[C]. 2003.

[8] TAIJIMA K, FUKUI Y. Answering XPath queries over networks by sending minimal views[A]. VLDB 2004[C]. 2004.

[9] XU WH. The framework of an XML semantic caching system[A]. WebDB 2005[C]. 2005.

[10] ABITEBOUL S, SEGOUFIN L, VIANU V. Representing and querying XML with incomplete information[A]. PODS 2001[C]. 2001.

[11] LIANG HUAI YANG, MONG LI LEE, WYNNE HSU. Efficient mining of XML query patterns for caching[A]. VLDB 2003[C]. 2003.

[12] LIN GUO, FENG SHAO, BOTEV C, *et al*. XRANK: ranked keyword search over XML documents[A]. SIGMOD 2003[C]. 2003.

[13] HE BS, LUO Q, CHOI B. Cache-conscious automata for XML filtering[A]. ICDE 2005[C]. 2005.

[14] 塔娜,冯建华,李国良,等. 纯 XML 数据库语义缓存中视图的快速查找算法[A]. NDBC 2006[C]. 2006.