一种新的XML文档的存储平台SDML的实现技术

洪晓光
(山东大学计算机学院　济南250100)

摘　要　目前,XML文档数据库(NXD—Native XML DBMS)的设计和存储正受到越来越多的关注,这是由于它可以灵活地表示各种数据,尤其是那些关系模式无法表达的复杂的数据。已经有一些NXD产品出现。而对XML文档的存储的好坏直接影响到它的查询效率,基于此我们自主提出了一种高效的XML文档存储平台SDML。详细讨论了它的存储结构和实现细节。特别提出了如何解决具有大量结构相同元素的存储方法,并给出了在其上进行查询、插入、删除和索引维护等操作的解决方案。给出了这种结构I/O费用代价,并进行了相关的实现,为NXD的存储优化提供一种新的途径。

关键词　XML文档,文档存储,路径查询

A New Storage Structure SDML of XML Documents and its Realization

HONG Xiao-Guang
(College of Computer, Shandong University, Jinan 250100)

Abstract　It is becoming increasingly attention for design and store in XML database. XML documents can have a rather complex internal structure; in fact, an XML dcuments can be viewed as an ordered tree. It can express many data types. Storing structure of a XML document influences directly efficency of an XML query. This paper proposes a storage structure SDML, and gives SDML's storage scheme and realization in detail. Especially we propose a scheme how to store a element type with many elements. We research algorithms of query, insert, and delete based-SDML.

Keywords　XML document, Document storage, Path query

1　介绍

目前,XML文档数据库设计(NXD)正成为数据库界关注的热点研究课题[1~4],这其中对XML文档多以平面文件数据库方式存储。很多NXD产品都支持"文档集合"的概念,就像文件系统中的一个目录或RDBMS中的一张表,一个"文档集合"把一类文档聚集在一起,方便用户操作。集合级别上的查询、修改操作都会反映到集合内的每个文档。XML文档多以文本方式表达,NXD提供查询功能,及支持Xpath和XQuery。有关文档的合理存储也已经有很多这方面的研究[5],但他们多考虑对树结构中的节点如何进行编码,如何高效插入和遍历,并且多集中于逻辑层面上的设计,而忽略了其物理存储和实现方面的瓶颈的研究。对于实现中遇到的各种问题基本上没有给出相应的解决方法。尤其是遇到具有大量结构相同元素时效率明显下降。基于此,我们自主提出了一种基于元素架构的XML文档的高效存储方案,它将字符数据与元素基本隔离,元素架构常驻缓存,并且提出了在架构内相同元素类型只保留一个副本并且对应附加映射表的技术,解决了重复元素的存储困难,大大减少了磁盘的I/O代价。在此基础上给出了执行路径查询,插入,删除及索引维护等操作的算法实现。而关于索引和文档加锁的技术讨论我们在另外的文章中给出。

本文第2节给出了XML文档一些基本概念。第3节提出了一种新的XML文档的存储平台SDML,并给出了其详细存储结构,特别给出了具有相同元素类型的XML文档的高效存储方案。第4节给出了在SDML平台上执行路径查询(全路经,基于约束的路径和不完整路径)的方法。第5节给出磁盘I/O的费用估计,探讨了SDML上索引的方法。第6节讨论了该结构的插入和删除操作。最后讨论了带有DTD模式的XML文档的存储方法,并给出了在此平台上正在进行的几个问题的研究说明。

2　基本概念

关于XML文档的基本概念已经有很多介绍,我们这里不再重复,只给出需要的一些概念。一个XML文档一般是由元素(元素名),属性(属性名,取值)和被元素包住的字符数据(可以有超级链接)组成。它是一种近似于层次的逻辑结构。所谓近似是因为在字符数据中仍然有超级链接。整个文档由一个大的元素组成,内部允许嵌套多层元素。结构上类似于广义表。很显然它的结构比关系数据库中关系更有表现力,一个XML元素可以包含其他元素、变化顺序和属性的数量,并且可以包含具有相同元素类型的多个元素,这使得它更容易表示复杂的数据。XML还有一些类似于面向对象的编程语言中对象的特性。例如,元素类型、命名属性和表示层次的能力。但也有本质的不同,对象设计例如方法,继承等,XML就不具备,就封装来说,对象隐藏所有的内部结构,而XML则显式地暴露所有的内部结构,因此对象成为更好的编程基础,而XML成为更好的数据表示的基础。因此XML比对象更容易与数据交换一起使用。

从存储角度上来说,由于它是文档的形式,因此存储一个文档有多种粒度可以选择,可以存储完整的文档,也可以被分割为更小的部分保存为片段,或被分解为独立的元素,这里SDML平台像其他NXD一样只考虑整个XML文档的存储,也就是说是一种大粒度的存储。当然这种方式对并发支持的力度将有所下降。下面我们首先给出数据块的定义。

数据块(block):数据在二级存储(一般为磁盘)和一级存

储(缓存)之间一次 I/O 的单位,在内存中它也称为页面(page)。大小在4k～56k 之间,这里我们以4k 为单位。在磁盘上它一般由多个扇区(512字节)组成。

对于关系存储由于记录大小比较固定而且相同,记录中不含有指针,关系模式单独存放。因此,一个数据块内一般有多条记录。块首存有模式信息和记录的偏移量,见图1。而对于对象存储难度将会大一些,因为对象的属性可能会指向另外的对象,就是说对象记录中有指针存在,当然还会有很多相关的方法被邦定在此对象上。这都需要用指针来完成。XML 文档的结构模式既不同于关系也不同于对象。它的元素和对应的字符数据之间既不像关系模式和记录那样完全分离也不像对象那样嵌套。它的每一对元素和它们包含的字符数据都有对应关系。而元素之间又存在一种层次架构。XED(XML-Enable DBMS)是在原有数据库基础上扩展了 XML 支持模块,完成 XML 数据和数据库之间的格式转换和传输。从存储粒度上,可以把整个 XML 文档作为 RDBMS 表中一行,或把 XML 文档进行解析后,存储到相应的表格中。为了支持 W3C 的一些 XML 操作标准,如 XPath,XED 提供一些新的原语(如 Oracle9iR2增加了一些数据包来操作 XML 数据等),并优化了 XML 处理模块。这种转换人为地将数据的层次关系与平面关系建立映射,建立复杂的映射关系。虽然存储上借用了现成的产品,但代价是降低了数据关系的反映能力,同时降低了查询效率。

我们根据这一特点给出了 XML 文档的 NXD 存储模式,采用元素和字符数据分离的原则,构造的元素架构是以数据块为单位结点的一种树结构,字符数据则以聚簇方式存放。每个结点表示一个不同的元素。下一节我们给出详细的设计。

图1　一个典型的存储关系记录的数据块

3　存储平台

我们将文档的存储分为两个部分:元素架构和字符数据。

3.1　元素架构

原则上文档中每个元素名有一个独立的 ID 号,对应一个数据块,块内由三部分信息组成,第一部分是块首部,它存放该元素在整个文档层次架构中的信息,例如该元素在文档中出现的次数。除此之外,还有一种编码信息,以方便遍历。编码方式这里不是我们讨论的重点,另外,还要存放块内各个其他指针的偏移量,如果该文档有 DTD,还要存放相关的 DTD 信息。第二部分存放所有子结点元素块的名称和地址,若文档中有相同元素类型的其他结点(元素名相同,属性不同,或者元素名相同,字符数据不同)则在元素架构中只存放一个结点元素副本,所有指向该元素类型的指针都指向这一副本。其余相同元素的数据块不被纳入元素架构,在该类结点附加有一个映射表。第三部分存放指向字符数据的指针,如果需要还可以保存下一个兄弟结点的地址或者父结点的地址,在这里我们一般只保留父结元素的地址。如果元素含有映射表,应当有指针指向映射表。图2为一个元素数据块的示意图。

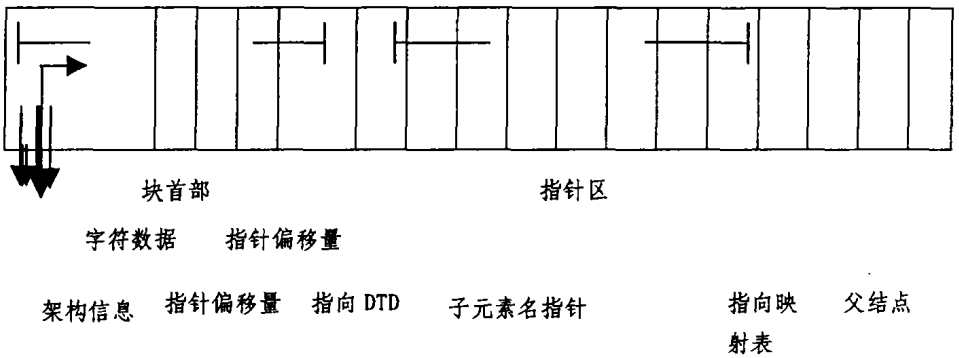

图2　一个元素块示意图

如上所述,由于在元素架构中只存放了相同元素类型的一个数据块,因此我们还必须有一张映射表来处理这一类元素。处理方式如下:

对于每一个元素架构中的这一类元素,保持一张映射表,表内纪录了其他同名元素的块地址,ID 号,属性值(如果有属性值),该元素的父结点元素名。形式如图3所示。

元素名	元素 ID	属性值	父结点元素名和 ID	元素块地址

图3　映射表组成

映射表表示文档中所有与该元素同名的其它元素,它们的 ID,属性值,各自父结点的元素名和 ID,这一项可以让我们很容易找到其父结点。另外还有该元素在外存的块地址。

我们给出一个 XML 文档实例,对其元素架构加以说明,图4表示文档的树层次,图5是该文档的架构树。

```
〈?xml version="1.0"?〉
〈DB〉
  〈A〉
〈B id='1'〉
  〈T1〉……〈/T1〉
    〈T2〉
      〈F1〉……〈/F1〉
      〈F2〉……〈/F2〉
      〈F3〉……〈/F3〉
      〈/T2〉
〈/B〉
  〈C〉……〈/C〉
  〈D〉
    〈F2〉……〈/F2〉
    〈T1〉……〈/T1〉
  〈/D〉
〈B id='2'〉
    〈T1〉……〈/T1〉
    〈T2〉
    〈F1〉……〈/F1〉
    〈F2〉……〈/F2〉
    〈F3〉……〈/F3〉
    〈/T2〉
〈/B〉
```

```
〈B〉
    〈T1〉……〈/T1〉
    〈T2〉
    〈F1〉……〈/F1〉
    〈F2〉……〈/F2〉
    〈F3〉……〈/F3〉
    〈/T2〉
〈/B〉
```

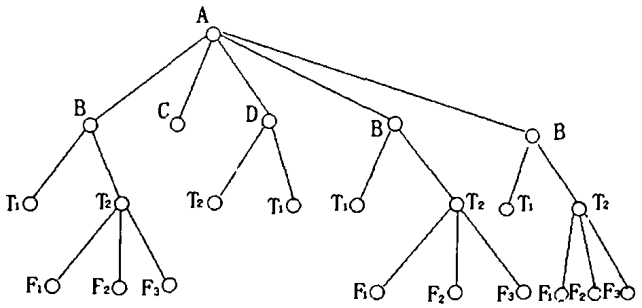

图4

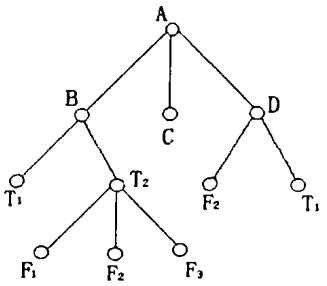

图5 元素架构

由图5可以看出元素架构中只保留了文档中每个元素的一个副本，这里B，T2，F1，F2，F3均带有映射表。元素架构只保留了那些必需的数据块，大量的重复块被排除在架构之外，这大大减少了查询中涉及的元素块的个数，一个元素结点以4k计，1000个结点只占据4M空间，而一个文档元素架构加上附加映射表远小于这个数字，使得文档的元素架构可以长期驻留在缓存中，由此可以大大减少磁盘的I/O费用，第4节我们将详细说明查询操作。

3.2 字符数据

我们以聚簇方式存放字符数据，相同元素名的字符数据被存放在一起。每一个聚簇保有一个表，表内存放该聚簇内所有超级链接的地址。该表随字符数据一起被调入内存。每一个字符数据前端计有该元素的Id号表示它属于哪一个元素。

由于结构中有大量的指针，一个无法回避的问题就是指针混写问题，所谓指针混写就是当指针作为数据被调入内存后到底是继续采用外存地址还是用内存地址加以替换。这里涉及到的指针有元素架构中每个结点指向字符数据的指针，映射表中各个元素的块地址，这里我们采用按需混写的方法，具体方案见另外的文章，这里不再赘述。

4 基于SDML存储的查询

查询一般有四种拓扑结构：结点，路径，树，图。查询可能从唯一标识符检索单一的元素结点；从“XML路径规范”定义的路径检索一个或多个结点；从树或图模式检索一个结点列表。查询结果的数据类型可能取决于查询拓扑。依靠唯一标识符的独立结点的查询可能返回一个字符串或者一个元素。我们这里描述的是基于路径的查询，它将结果规范为一个结点的集合。

路径查询：路径查询提供一种检索结点的机制，这一结点出现在相对于开始结点的某一位置。典型地，这一查询引擎总是搜索一条从开始结点到终止结点的路径。在上例中一种路径查询可以表示为A/B/T2/F2，这种查询一般分为如下四步：

step1：从起始结点开始，例如从A开始。

step2：检索起始结点的子元素，使其元素种类名与路径中的第一个种类名相同。

step3：对每个子元素，用路径的其余部分，重复上一步骤。

step4：如果完成了路径，将完成所有分支时返回的子元素搜集在一个列表中。

理论上这种查询是在一个文档树中相应于路径的部分，进行深度优先搜索。对于这种查询，我们注意到如果文档中存在大量相同元素类型，这种搜索方式将会带来大量的递归和多次的磁盘I/O，查询效率非常低下。而在SDML存储平台之上，我们将会看到由于采用元素架构使得递归和磁盘I/O会大大地减少。其查询步骤如下：

step1：根据查询提交的路径，首先从元素架构的根结点块开始，检索它的子结点使其与路径中的元素匹配。

step2：对每个子元素，用路径的其余部分，重复上一步骤，由于在元素架构中没有重复，因此这样不断重复可到达终止结点。

step3：如果终止结点不带有映射表，表示它没有重复元素。这时，只需要将它的字符数据指针所指的字符数据块调入内存就完成查询操作。

step4：如果终止结点带有映射表则表示文档中有该元素的重复项，将映射表中所有元素进行过滤，即采用自底向上的策略，找出哪些元素符合查询路径。

过滤操作并不困难，因为不论是架构中的元素还是映射表中的元素都有一个父结点元素名和ID的属性。若父结点元素名在路径表达之中，则元素从架构中一个元素沿父结点向上进入上一层，在该层中若父结点也带有映射表则根据ID号找到该元素，重复这一过程，直到根结点。

step5：将过滤出的元素从映射表中直接就可以得到它们的块地址，将它们调入内存，按照它们的字符数据指针即可得到所要的查询结果。

可以看出该算法经过对搜索路径的一次折返，充分利用SDML的元素架构和映射表结构，避免了大量的递归操作和磁盘I/O。我们给出一个实例，考虑上面给出例子，查询路径为A/B/T2/F2，按照我们的算法，首先匹配路径A/B/T2/F2，第一次匹配我们不考虑元素的映射表，由于对于重复元素我们在其元素架构中只保留了一个副本，不必递归，很容易就可以到达F2，由于F2带有映射表，也就是说文档中有很多F2元素，我们必须过滤掉不是该路径上的F2，这时进行路径的自底向上二次匹配，从F2的映射表父结点属性可以得到所有F2的父结点元素名和ID，可以看出，有三个父节点满足路径的表达，因此从F2到T2，T2也带有映射表，重复这一过程直到根结点，从上例中很显然只有一个父节点为D的F2被过滤掉，剩下的F2就是满足条件的元素。再向磁盘读取它的字符数据。

除了搜索全部指定的路径之外，在一个XML路径中的任一结点也可能是一个通配符，或者被一个结点约束所限制，例如上例中A/*/F2，或者A/B id='1'/T2/F2。这种在路径之上加约束的查询对于我们上面给出的算法无需做大的改动，只要在二次匹配时将过滤算法扩大，判定条件除了符合路

径条件还要符合属性要求，若不符合，过滤过程就终止，不需要回到根结点，进行下一个元素的过滤。

但对于含有通配符的路径查询我们将把上面算法中两次匹配合并进行，这需要借助于索引，首先找到路径中的两个端结点，索引我们采用散列方式，它建立在元素架构之上，每个元素类型对应一个散列桶，桶内存有该元素在元素架构中的块地址，得到端点之后，从底端开始进行匹配，只要发现路径元素，一条路经就匹配成功。

综上所述，我们可以看出，该存储平台非常适合于基于路径的查询。

5 查询代价

现在我们来看基于SDML平台的查询费用。首先，由于元素架构是可以常驻内存的，与元素相对应的映射表也可以常驻内存，这一点前面我们已经分析过，因此这一部分的磁盘I/O可以忽略。当进行查询时，只有找到匹配的元素，从磁盘读进映射表中该元素块，这要求一次I/O，然后读进相应的字符数据又需要一次I/O，总共两次磁盘读写完成查询，在内存算法中搜索深度相当于查询路径长度相对于普通XML文档的存储和查询，我们的算法首先避免了大量的深度优先递归，而且避免了相同元素被同时存入占用大量内存的缺点。但对于重复的元素，我们需要进行过滤运算，这需要对映射表中的元素一一操作，我们这里假设重复元素的映射表不大。这种过滤是线性的，而且过滤深度最大就是元素架构的深度。如果在元素架构之上建有散列索引，在全路经查询中也可以利用索引进行，可以直接到达查询路径最后元素在元素架构中的位置，无需自上而下得路径匹配，直接进行过滤，但这样仍然需要对索引有一次I/O操作，改进并不大，因此只有在路径存在通配符的情况下使用索引才更有效。如果某个元素的映射表很大，无法常驻内存，我们将借助于B^+树作为映射表的索引，元素的属性，ID号作为索引项。一般情况，映射表大时，查询多为路径上有属性约束。任何约束都将使我们减少元素过滤时路径匹配的长度。

现在假设B表示文档元素数据块的个数，$B_1,B_2,\cdots,B_m$分别表示不同元素类型元素的块数，$B=|B_1|+|B_2|+\cdots|B_m|$，$B_i$可以为0。这样元素架构中最多就只有$m$个不同的元素类型。

B表示文档中元素的数据块数，影响B的变量有m和$|B_i|$。随着m的增加我们在图中可以看出架构变大。另外一种情况是m不变，但有若干B_i增加，B增加，但架构不变。

更具一般性，假设可用内存缓冲区大小为M，$m \ll M$，因此，元素架构可以常驻内存。我们假设查找的元素就是相同元素名最多的B_i。B_i越大，普通存储下需要的I/O就越多，而SDML中元素架构并不受影响，只是具有B_i元素的映射表加大，过滤过程增加。对映射表这里要分三种情况：假设B_i的映射表占用K_i个块，而且是紧缩存放。很显然，$B_i \gg K_i$。一般情况$K_i=1$，$B_i=240$，即一个映射表块可以表示240个元素块。映射表类似于一个该元素的索引表。

a）$K_i \ll M$，这是最好情况，即映射表可以常驻内存。不改变I/O的次数。

b）$K_i < M$，这时K_i已经不能常驻内存，但仍然可以读入内存。对该表中元素的过滤需要K_i次I/O的读取。但查询过程中无需I/O操作。

c）$K_i > M$则在查询过程中至少需要有K_i/M次I/O操作。每次读进M大小的映射表子块。最坏情况是路径中多个元素（h个）都有映射表并且大小都是K_i，需要$h * K_i$次表扫描，h是元素架构的深度。那么在查询过程中需要$h * K_i * K_i/M$次I/O操作。

普通存储下通过深度优先搜索查找符合的路径本身虽然比在解空间中查找最优路径相对简单但依然非常困难，即使利用回溯法，通过剪枝操作减少递归路径也很难从本质上提高算法的时间。依然为B_i指数级操作，这还没有算它的I/O操作。可以看出K_i/M越小，SDML存储平台的优点就越明显。

6 插入和删除

对于插入，如果是插入字符数据，我们可以利用查询算法找到所要插入的元素，并由它的字符数据指针调入字符数据块，进行插入。这种插入相对简单，因为这没有改动元素架构。若插入新的元素，则分为两种情况，一种是元素类型已经存在，则首先在磁盘为其分配新的数据块，通过索引搜索找到元素架构中的该元素，修改它的映射表将插入元素的信息填入。同时在其父结点中引一条边到该元素。另一种情况是要插入的元素在元素架构中不存在，这时，找到该元素要插入的父结点，将该元素作为其子结点直接插入，同时，同步完成对索引的维护，由于索引只是对元素架构建立的，维护本身属于对散列表的操作，我们另文介绍。一般情况下，在元素架构中每个元素占用一个数据块，以此有足够的空间允许子结点的插入。如果数据块确实已经存满没有空间容纳新的元素，我们将为该结点增加一个溢出块。在极端情况下，也就是该结点的子结点如果太多，而且又没有重复元素。我们将实现对该文档的分裂使原来的文档一分为二。在原始架构中保留新产生文档的元素架构根结点的地址。

对于删除，我们对元素架构不作改动，只是对其指针和映射表的相应内容作删除标记，在查询时对有标记的元素不予考虑。等以后有新元素插入时将占用该位置。

我们再次提到指针混写，这里多处需要内外存之间的指针混写，我们一般采用按需混写的模式来处理此类问题，详细实现这里不再赘述。

结论 可以看出，不论插入还是删除，对元素架构和建立在架构之上的索引的维护相对都很简单，因此SDML平台针对以文档为存储单元的XML存储是一种非常有效的方法，但该结构对于含有递归的查询我们未进行讨论。我们将在下一步详细讨论该结构对于查询所适应的范围。在此基础上我们将继续完成元素架构的拆分、合并操作，使之能够对多文档环境下高效查询成为一种有效的平台。另外，我们将给出在该平台之上适合于查询的种类和模式。

参考文献

1 Chien S-Y, Vagena Z, Zhang D-H. Efficient Structural Joins on Indexed XML Documents. In: Proc. of the 28th VLDB Conf. Hong Kong, China, 2002

2 Cattell R G. Object Data Management. Addison-Wesley, Reading MA, 1994

3 Wiederhold G. File Organization for Database Design. McGraw-Hill, New York, 1987

4 Li Q, Moon B. Indexing and Querying XML Data for Regular Path Expressions. In: Proc. of VLDB, 2001

5 Garcia-Molina H, Ullman J D, Widom J. Database System Implementation. Prentic Hall, 2000