软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(8):2042−2057 [doi: 10.3724/SP.J.1001.2012.04161]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于时态编码和线序划分的时态 XML 索引*

郭 欢[1,2], 叶小平[1], 汤 庸[1,2+], 陈罗武[2]

[1](华南师范大学 计算机学院,广东 广州 510631)

[2](中山大学 计算机科学系,广东 广州 510006)

Temporal XML Index Based on Temporal Encoding and Linear Order Partition

GUO Huan[1,2], YE Xiao-Ping[1], TANG Yong[1,2+], CHEN Luo-Wu[2]

[1](School of Computer Science, South China Normal University, Guangzhou 510631, China)

[2](Department of Computer Science, Sun Yat-Sen University, Guangzhou 510006, China)

+ Corresponding author: E-mail: ytang@scnu.edu.cn

Guo H, Ye XP, Tang Y, Chen LW. Temporal XML index based on temporal encoding and linear order partition. *Journal of Software*, 2012,23(8):2042−2057 (in Chinese). http://www.jos.org.cn/1000-9825/4161.htm

Abstract: A temporal XML indexing structure based on temporal encoding and linear order partition was studied. First, a temporal encoding method based on extended preorder encoding was proposed, by which the structural relationship between nodes can be determined. Second, based on detail analysis of relationship between time intervals, the concept of linear order partition was proposed, and algorithm to attain a linear order partition was also discussed. Then, a temporal structural summary was introduced which includes both structural and temporal information, and a temporal XML indexing mechanism—TempSumIndex was built based on temporal structural summary, then, both temporal querying and incremental updating algorithms of TempSumIndex were discussed. Finally, experiments were designed to compare the basic performance of TempSumIndex with existing temporal XML indexing methods, and the experimental results show that TempSumIndex has better performance.

Key words: temporal XML index; temporal encoding; linear order partition; temporal query and update; simulation and performance assessment

摘 要: 研究了一种基于时态编码和线序划分的时态 XML 索引机制.首先,提出一种基于扩展先序编码的时态编码方案,通过该编码可确定结点间的结构关系;其次,在深入分析时间区间关系的基础上引入线序划分的概念,并讨论了获取线序划分的算法;然后,建立了整合路径结构信息和时态约束信息的时态结构摘要,并在此基础上建立了时态 XML 索引结构——TempSumIndex,同时研究了基于 TempSumIndex 的时态 XML 查询和增量式更新算法;最后,对 TempSumIndex 和现有时态 XML 索引技术的基本性能进行了详细的实验评估.实验结果表明,TempSumIndex 具有更为优越的性能.

关键词: 时态 XML 索引;时态编码;线序划分;时态查询与更新;仿真与性能评估

中图法分类号: TP311 **文献标识码**: A

* 基金项目: 国家自然科学基金(60673135, 60970044, 60736020); 广东省自然科学基金(7003721, 9151027501000054, S2011010003409); 广东省战略新兴产业项目(2011A010801007, 2011168005)

收稿时间: 2010-06-30; 定稿时间: 2011-11-17

随着网络技术的深入发展和快速普及,XML 数据管理的应用需求日益迫切.数据查询是数据管理的核心,对于 XML 数据库来说,可行的和有效的查询主要依赖于两个基本要素:一是具有足够表达能力的查询语言,如 XPath 和 XQuery;二是具有高效的查询处理器.面对 XML 数据库中的海量数据,开发高效的查询处理器已成为当前急需解决的重要课题,而高效的 XML 索引技术正是提高查询性能的关键.正如时态数据库是常规(非时态)数据库的时态扩充,时态 XML 数据管理也是常规 XML 数据管理的基本拓展,其中许多基本思想和技术都以非时态情形为研究基础.现有的时态 XML 研究工作主要可分为 3 个方面:(1) 时态 XML 建模.主要有 De Capitani 等人提出的授权访问模型[1]、Amagasa 等人提出的基于 XPath 的时态数据模型[2]、Dyreson 等人提出的通过扩展 XPath 的事务时间访问模型[3]、Wang 和 Zaniolo 在 Web 数据仓库研究中提出的历史版本模型[4]以及双时态 XML 数据模型[5].(2) 时态 XML 查询语言.主要有 Gao 等人提出的支持有效时间查询的 τXQuery[6]、Mendelzon 等人建立的 TXPath[7].(3) 时态 XML 索引技术.主要工作有 Mendelzon 等人提出的基于连续路径的时态 XML 索引模型[7].Rizzolo 和 Vaisman 以此为基础,建立了基于图的时态 XML 索引模型 TempIndex,并研究了时态 XML 一致性问题[8].叶小平等人提出了一种基于时态连通和时态包含关系的索引结构 TXIDM[9],并讨论了路径查询和值查询算法,但没有较好地研究索引的更新问题.后来,我们在 TXIDM 的基础上进行了改进,并将其推广至时态数据索引的一般情形[10].

本文工作主要是研究时态 XML 索引技术.不同于常规情形,时态 XML 索引技术需考虑 3 方面的基本要求:

(1) 索引构建.需要考虑 XML 时态查询和路径查询的基本特征,保存数据的时态关系、结构关系和顺序关系.

(2) 索引查询.既能够实现常规的时态“值”查询,又能实现具有 XML 特征的时态“结构”查询.

(3) 动态管理.由于时态 XML 数据量巨大,相应的时态索引结构的更新应当是增量式的,且该增量式更新具有“完备性”,即与完全式更新的结果相同.

相对于常规的 XML 索引技术,国内外现有的对时态 XML 索引方面的研究工作较少,如何在索引构建过程中取得处理结构信息和时态信息之间的协调是一项基本挑战.基于上述分析,我们提出了一种基于时态编码和线序划分的时态 XML 索引机制.本文的贡献主要有 3 个方面:

- 一是考虑数据结点间的时态结构信息,提出了基于扩展先序编码的时态编码方案,通过该编码可确定结点间的时态结构关系.
- 二是在深入分析时间区间关系的基础上引入线序划分的概念,并讨论了获取时间区间集线序划分的算法.
- 三是在时态编码和线序划分的基础上拓展常规 XML 索引中的结构摘要,在保存路径结构信息的基础上引入可以快速执行时态过滤的线序划分,并建立时态结构摘要和相应的索引结构.

1 时态 XML 索引模型

一个时态 XML 文档 TD 是在常规 XML 文档(例如,基于 Xpath 的查询数据模型)中添加了时间标签的有向无环图结构.时间标签是一个时间区间,标记在 TD 中的边上,相应边称为时态边.结点 u 的有效时间区间记为 $VT(u)=[VTs(u),Vte(u)]$,其中,$VTs(u)$和 $Vte(u)$分别表示 $VT(u)$的时间始点和终点,且 $VTs(u)\leqslant VTe(u)$.对于 TD 中任意两个结点 u 和 v,如果 $VTs(u)\geqslant VTs(v)\wedge VTe(u)\leqslant VTe(v)$,则称时间区间 $VT(u)$包含于 $VT(v)$或 $VT(v)$包含 $VT(u)$,记作 $VT(u)\subseteq VT(v)$.如果$\neg(VT(u)\subseteq VT(v)\vee VT(v)\subseteq VT(u))$,则称 $VT(u)$和 $VT(v)$互不相容.为研究时态 XML 数据索引机制,本文引入一种基于 TXPath 的时态 XML 查询数据模型.

定义 1(时态 XML 查询数据模型). 称时态 XML 数据文档 TD 为基于时态 XML 查询数据模型的数据模式,记作 Txqdm,如果 TD 满足下述条件:

(1) 结点 u 可表示为 $u=\langle SL(u),VT(u),ID(u)\rangle$,其中,$SL(u)$和 $VT(u)$分别是结点 u 的语义标签和有效时间标签,$ID(u)$是结点 u 的先序编码,用来唯一标识 u.

(2) 结点所有出边有效时间的并集不超过其所有入边有效时间的并集.

(3) 若结点 u 是结点 v 的父结点,则 $VT(v)\subseteq VT(u)$.

(4) 若结点 u 和 v 是兄弟结点:

- 如果 u 和 v 是值结点,则 $VT(v)\cap VT(u)=\varnothing$;
- 如果 u 和 v 是非值结点(元素结点或者属性结点),且 u 是 v 右兄弟结点,则 $VTs(v)\leqslant VTs(u)$.

例 1:基于 Txqdm 的时态 XML 实例如图 1 所示.为简化图示,图中所有时间区间为[0,*now*]的时间标签都已省略,结点内的数字为该结点的唯一标识码.结点 n_3 和 n_{22} 分别有一个指向结点 n_{12} 和 n_{17} 的引用属性,在图 1 中用虚线表示.职员的的信息会随着时间的变化而改变,如职员 Bob 在时间区间[0,20]内就职于公司 C1,而在时刻 21 转职到公司 C2,并持续到现在,所以结点 n_{12} 的有效时间区间为[0,20]∪[21,*now*]=[0,*now*].

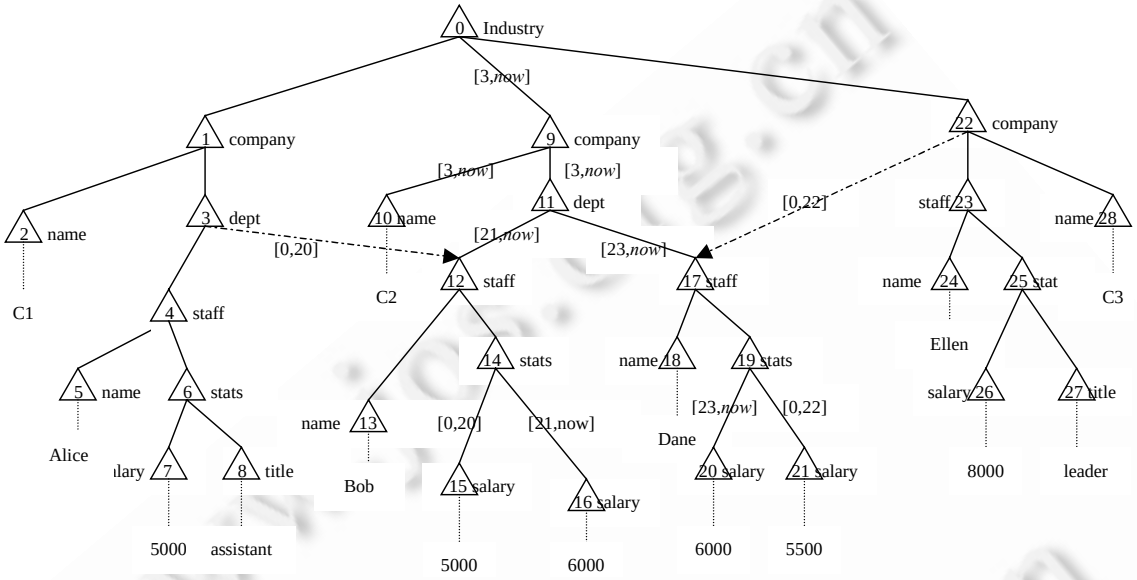

Fig.1 Temporal XML instance based on Txqdm

图 1 基于 Txqdm 的时态 XML 实例

1.1 时态编码

在实际索引结构中,XML 数据结点被划分到不同的索引结点中,丢失了 XML 数据模型中数据结点间的结构关系信息.为保持结点间的时态结构关系,本文提出一种时态编码方案.

定义 2(时态路径). 在基于 Txqdm 的时态 XML 数据文档 TD 中,从根结点 n_1 到结点 n_k,在时间区间 $T(T\neq\varnothing)$ 内的时态路径是一个结点序列$(n_1,\ldots,n_k,T)$,包含时态边 $e_1(n_1,n_2,T_1),e_2(n_2,n_3,T_2),\ldots,e_{k-1}(n_{k-1},n_k,T_{k-1})$,其中,$T=\bigcap_{i=1,\ldots,k-1}T_i$, 记作 $TP(n_1,\ldots,n_k)$或 $TP(n_k)$.

在时态路径 $TP(n_1,\ldots,n_k)$中,称结点 $n_1,\ldots,n_{k-1}$ 为结点 n_k 的时态祖先结点,结点 n_{k-1} 为 n_k 的时态父结点.

定义 3(时态编码). 将基于 Txqdm 的时态 XML 文档 TD 对应的有向无环图中所有引用边看作实边,结点 u 基于扩展先序编码的时态编码表示为 $TCode(u)=\langle Code(u),Gap(u)\rangle$,其中,

(1) $Gap(u)$为结点 u 的时态预留空间.

(2) $Code(u)$为结点 u 的时态扩展先序编码且满足:

- 对于图中的时态兄弟结点 w 和 v,如果 v 是 w 的右兄弟,则 $Code(w)+Gap(w)<Code(v)$;
- 对于图中的结点 v 和其时态父结点 w,有 $Code(w)<Code(v)\wedge Code(v)+Gap(v)\leqslant Code(w)+Gap(w)$;
- 对于图中的结点 w,一定满足 $Gap(w)\geqslant\sum_v Gap(v)$, 其中,$v$ 是 w 的所有时态孩子结点.

为便于表述,下文中直接用时态扩展先序编码 $Code$ 表示时态编码 $TCode$.

定义 4(时态结构关系). 设 v 和 w 是时态 XML 数据模型 TD 的两个结点:

(1) 如果 $TCode(v)<TCode(w)$,则称 w 是 v 的时态后继,v 是 w 的时态前驱;

(2) 如果 w 是 v 的时态后继,且在 v 和 w 之间没有满足 $TCode(v)<TCode(w')$的结点 w',则称 w 是 v 的直接时态后继,v 是 w 的直接时态前驱.

在本文的索引结构中,将根结点 r 的时态编码表示为 $TCode(r)=Code(r)$.如果已知当前时态结点 w 的时态编码为 $TCode(w)$,则其直接时态后继结点 v 的时态编码为 $TCode(v)=Code(w)+Gap(w)+1$.

例 2:对于图 1 所示的时态 XML 实例,相应的时态编码如图 2 所示.

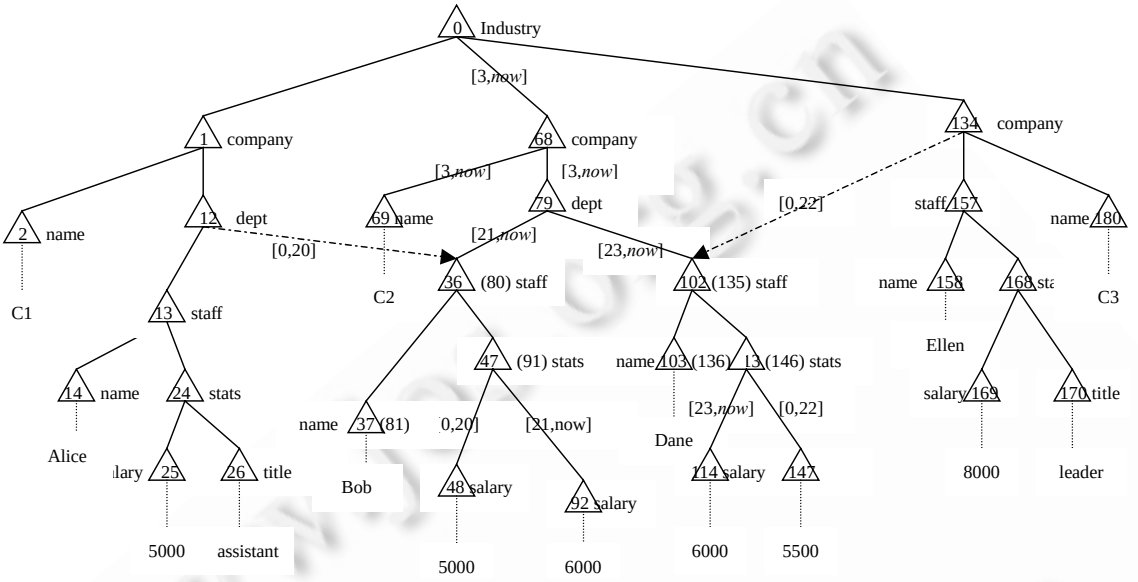

Fig.2 Temporal encoding graph of temporal XML instance

图 2 时态 XML 实例的时态编码图

当存在引用属性时,一个时态结点在不同时间区间内可能存在多条时态路径,在先序遍历过程中会多次访问该结点.因此,一个时态结点可能有多个时态编码,如图 1 中的结点 n_{12} 和 n_{17}.由定义 4 可知,时态编码为 24 的结点是时态编码为 14 的结点的直接时态后继.

定理 1(祖先子孙时态编码关系). 设结点 n 的时态编码为 $TCode(n)$,其具有相同时态路径长度的直接时态后继 n'的时态编码为 $TCode(n')$,则结点 u 是结点 n 的时态子孙结点,当且仅当 $TCode(n)<TCode(u)<TCode(n')$.

证明:充分性:若结点 u 满足 $TCode(n)<TCode(u)<TCode(n')$,由时态结构关系的定义可知,$u$ 是 n 的时态后继,n'是 u 的时态后继.由于 n'是结点 n 的具有相同时态路径长度的直接时态后继结点,因此,u 是一定是 n 的子孙结点,充分性得证;

必要性:根据时态编码的定义可知,结点 n 的子孙结点一定是其时态后继,所以有 $TCode(u)>TCode(n)$成立;同时,结点 n 的子孙结点一定是其具有相同时态路径长度的直接时态后继 n'的时态前驱,因此有 $TCode(u)<TCode(n')$成立,必要性得证. □

推论 1(父子时态编码关系). 结点 n 的所有时态子孙结点中,如果结点 u 的时态路径长度比 n 的时态路径长度大 1,即 $length(TP(u))=length(TP(n))+1$,则 u 是 n 的时态儿子结点.

定理 1 及其推论表明,时态编码可以保持时态 XML 数据模型中结点间时态结构关系,因此通过时态编码可以快速执行结构连接操作.同时,时态预留空间的设定可以有效地降低时态 XML 文档更新过程中的重构代价.

1.2 时态关系

为了建立有效的时态索引结构,需要深入讨论时间区间集Γ上的基本时态关系.

定义 5(线序分枝和线序划分). 设 L 为Γ的一个子集,若$\forall P_i,P_j\in L$,满足 $P_i\subseteq P_j\vee P_j\subseteq P_i\vee P_i=P_j$,则称 L 是Γ中的一个线序分枝(linear order branch,简称 LOB).线序分枝 L 可表示为 $L=\langle \max L,\min L,L'\rangle$,其中,$\max L$ 和 $\min L$ 分别表示 L 的最大元和最小元,即$\forall P_i\in L$ 均有 $P_i\subseteq\max L$ 和 $\min L\subseteq P_i$ 成立;L'是 L 中元素根据线序分枝的性质排序后的集合,即 L'中任意相邻的两个元素 P_{i-1}和 P_i有 $P_i\subseteq P_{i-1}$成立.令$\Delta=\{L_i|1\leqslant i\leqslant k\}$为$\Gamma$的线序分枝集,如果 $\bigcup_{L_i\in\Delta}(L_i)=\Gamma$,则称$\Delta$为$\Gamma$的一个线序划分(linear order partition,简称 LOP).

定义 6(极小和极大线序划分). 设Δ_0是Γ上的一个 LOP,如果对于Γ上任意线序划分Δ均满足$|\Delta_0|\leqslant|\Delta|$,则称$\Delta_0$为$\Gamma$上极小线序划分;如果对于$\Gamma$上任意线序划分$\Delta$均满足$|\Delta|\leqslant|\Delta_0|$,则称$\Delta_0$为$\Gamma$上极大线序划分.其中,$|E|$表示集合 E 的基数.

时间区间集Γ上可能存在多个 LOP,对于时态查询效率而言,本文的索引结构需要考虑极小 LOP,在第 2.2 节中的时态查询会给出具体分析.为直观地分析线序划分的计算过程,下面给出时态矩阵的概念.

定义 7(时态矩阵). 设Γ为时间区间集合,时间区间$[i,j]$记为 $P_{ij}(i_1\leqslant i\leqslant i_m,j_1\leqslant j\leqslant j_n)$,其中,$i_1$和 i_m 分别为Γ中所有时间区间的最小和最大时间始点,j_1 和 j_n 分别是最小和最大的时间终点.设平面上横轴和纵轴上的点分别表示时间区间的始点和终点,由$\{P_{ij}|i_1\leqslant i\leqslant i_m,j_1\leqslant j\leqslant j_n\}$确定的所有格点集合称为基于$\Gamma$的时态矩阵(temporal matrix),记作 $TM(\Gamma)$.

对于任意 $P_{xy}\in TM(\Gamma)$,P_{xy} 将 $TM(\Gamma)$分为 4 个区域:$UL(P_{xy})=\{P_{ij}|i\leqslant x,y\leqslant j\}$,$UR(P_{xy})=\{P_{xy}|x\leqslant i,y\leqslant j\}$,$DL(P_{xy})=\{P_{ij}|i\leqslant x,j\leqslant y\}$和 $DR(P_{i0j0})=\{P_{ij}|x\leqslant i,j\leqslant y\}$.上述各式中如果仅是“<”成立,则称相应区域为开区域,分别记为 OUL, OUR,ODL 和 ODR.易知,被 P_{xy} 包含的时间区间一定位于 $DR(P_{xy})$区域内,包含 P_{xy} 的时间区间一定位于 $UL(P_{xy})$区域内,而位于 $OUR(P_{xy})$和 $ODL(P_{xy})$区域内的时间区间则与 P_{xy} 不相容.同理,对于任意线序分枝 L,L 可将时态矩阵划分为 4 个闭区域 $UL(L)=UL(\max L)$, $UR(L)=\bigcup_{P_i\in L}UR(P_i)$, $DL(L)=\bigcup_{P_i\in L}DL(P_i)$ 和 $DR(L)=DR(\min L)$,相应的开区域可类似定义.

获取时间区间集Γ的线序划分的基本思想是:在 $TM(\Gamma)$上进行一次下优先遍历:从最左上角的元素开始,若当前元素 P_i 的 $DR(P_i)$区域内有元素,则获取 $DR(P_i)$区域内最左上角的元素,依此类推;当 $DR(P_i)$区域内没有Γ中元素时,所有扫描到的元素组成一个线序分枝.然后从剩余元素中的最左上角元素开始同样的过程,依此类推,可以获取Γ的一个线序划分.具体过程如算法 1 所示.

算法 1. *getLOP*(Γ). //获取时间区间集合$\Gamma=\{P_1,P_2,\ldots,P_n\}$的一个线序划分

Step 1:将Γ中所有时间区间按照时间始点升序排序;如果时间始点相等,则按照时间终点降序排序.不妨设排序后的区间集仍为为$\Gamma=\{P_1,P_2,\ldots,P_n\}$.

Step 2:令 $i=1,j=1,LOP=\varnothing$.

Step 3:如果$|\Gamma|=0$,则转到 Step 4;否则,

Step 3.1:若$j\leqslant|\Gamma|$且 $P_j\subseteq\min L_i$,则将 P_j 作为最小元添加至 L_i 内,并将 P_j 从Γ中删除;若$j>|\Gamma|$,则转 Step 3.3.

Step 3.2:令 $j=j+1$,转 Step 3.1.

Step 3.3:将 L_i 添加到 LOP 中;令 $i=i+1,j=1$,转 Step 3.

Step 4:输出线序划分 LOP.

算法 1 的时间复杂度:Step 1 中排序的复杂度为 $O(n\log n)$;Step 2 为常数时间复杂度;Step 3 中,获取一个线序分枝需要扫描当前时间区间集中的所有元素,最坏情况下,每个线序分枝只包含 1 个时间区间,因此,扫描的时间区间集的大小是依次递减的,故 Step 3 的复杂度最大为 $n+(n-1)+\ldots+1=n(n+1)/2=O(n^2)$;Step 4 的时间复杂度为 $O(n)$.所以,算法 1 的时间复杂度为 $O(n^2)$.

定理 2(线序划分的最小性). 由算法 1 得到的线序划分是时间区间集Γ的最小线序划分.

证明:设线序划分$\{L_1,L_2,\ldots,L_k\}$由算法 1 获得,且 L_i 按生成顺序排序,则对于任意 $P_{i0}\in L_i(1<i\leqslant k)$,存在 $P_{(i-1)0}\in$

L_{i-1},使得 P_{i0} 和 $P_{(i-1)0}$ 不相容,即 $P_{(i-1)0}$ 位于区域 $ODL(P_{i0})$内.假设 $P_{(i-1)0}$ 不存在,根据算法 1,LOB_{i-1} 中的所有元素一定位于区域 $UL(P_{i0})$内,此时有 $P_{i0}\subseteq \min LOB_{i-1}$ 成立.因此,P_{i0} 应该作为最小元被添加至 LOB_{i-1},与假设矛盾.因此,L_{i-1} 中存在与 P_{i0} 不相容的元素 $P_{(i-1)0}$.由此可得元素 $P_{i0}\in L_i(1\leqslant i\leqslant k)$,且 $P_{10},P_{20},\ldots,P_{k0}$ 互不相容.所以,$\{P_{i0}|1\leqslant i\leqslant k\}$中的任意两个元素不可能位于同一线序分枝中.因此,时间区间集Γ的任何线序划分至少包含 k 个 LOB.

定理得证. □

1.3 时态索引模型

常规 XML 数据索引实现的基本方法是结构摘要技术,通过对 XML 文档的路径结构进行归并,只维护不同的路径信息,使得基于不同路径的查询可以避免对整个文档的遍历访问.本文将 1-Index[11]扩展至时态情形,在保存路径结构信息的基础上引入可以快速执行时态过滤的线序划分,在结构摘要中建立起更为精细和有效的索引结构.

定义 8(基于时态编码和线序划分的时态 XML 数据索引模型). 基于 Txqdm 的时态 XML 数据文档 *TD* 的索引模式 *TempSumIndex*(*TD*)为三元组 *TempSumIndex*(*TD*)=⟨*TSum*,*LTlop*,*TDcode*⟩,其中,

- *TSum* 为 *TD* 基于 1-index 的时态结构摘要树,其作用是快速筛选所需路径结构和时态信息.*TSum* 中的时态摘要结点 *S* 可以表示为五元组 *S*=⟨*Sid*,*TPlen*,*TPlabel*,*DTCodes*,*Tlop*⟩,其中,*Sid* 为结点 *S* 在 *TSum* 中的先序编号;*TPlen* 和 *TPlabel* 分别为结点 *S* 的时态路径长度和时态路径语义标签;*DTCodes* 为结点 *S* 中包含的所有数据结点在 Txqdm 中的时态编码的集合,且编码以升序排序;*Tlop* 为结点 *S* 内所有数据结点的有效时间区间的线序划分.
- *LTlop* 为 *TSum* 中各层时态摘要结点所包含的数据结点的有效时间区间的线序划分,用于文档的快照查询.
- *TDcode* 为 *TD* 中所有结点的时态编码到原始数据编号的映射表,包含两个属性 *TCode* 和 *ID*,表示 Txqdm 中时态编码为 *TCode* 的结点对应原始 XML 实例中编码为 *ID* 的数据结点,主要用于处理引用结点.

由于 *TD* 中引用边的存在,被引用的结点可能会存在多个的时态编码,所以在 *TempSumIndex*(*TD*)中的时态编码与原始数据结点没有一一对应关系.

因此,为了建立由时态编码到原始数据的快速映射机制,*TempSumIndex*(*TD*)中引入了编码映射表 *TDcode*.

例 3:与图 2 中时态 XML 数据模型的时态编码图相对应的时态结构摘要树如图 3 所示,相应的时态摘要结点及各层线序划分的基本信息见表 1 和表 2.其中,时间区间的下标数字表示有效时间为该时间区间的数据结点的时态编码.由图 1 和图 2 可以直接获取结点的时态编码和数据结点标识 *ID* 的映射关系,在此没有给出映射表 *TDcode* 的详细信息.

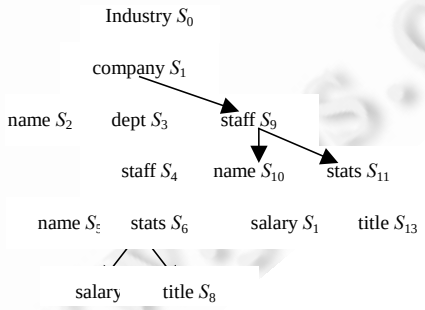

Fig.3 Temporal structural summary tree *Tsum*

图 3 时态结构摘要树 *Tsum*

Table 1 Information of nodes in temporal structural summary tree *Tsum*

表 1 时态结构摘要树 *TSum* 中的结点信息

Sid	*TPlen*	*TPlabel*	*DTCodes*	*Tlop*
S_0	0	*Industry*	0	$\{\langle[0,now]_0\rangle\}$
S_1	1	*Industry.company*	1, 68, 134	$\{\langle[0,now]_1,[0,now]_{138},[3,now]_{64}\rangle\}$
S_2	2	*Industry.company.name*	2, 69, 180	$\{\langle[0,now]_2,[0,now]_{180},[3,now]_{69}\rangle\}$
S_3	2	*Industry.company.dept*	12, 79	$\{\langle[0,now]_{12},[3,now]_{79}\rangle\}$
S_4	3	*Industry.company.dept.staff*	13, 36, 80, 102	$\{\langle[0,now]_{13},[0,20]_{36}\rangle,\langle[21,now]_{80},[23,now]_{102}\rangle\}$
S_5	4	*Industry.company.dept.staff.name*	14, 37, 81, 103	$\{\langle[0,now]_{14},[0,20]_{37}\rangle,\langle[21,now]_{81},[23,now]_{103}\rangle\}$
S_6	4	*Industry.company.dept.staff.stats*	24, 47, 91, 113	$\{\langle[0,now]_{24},[0,20]_{47}\rangle,\langle[21,now]_{91},[23,now]_{113}\rangle\}$
S_7	5	*Industry.company.dept.staff.stats.salary*	25, 48, 92, 114	$\{\langle[0,now]_{25},[0,20]_{48}\rangle,\langle[21,now]_{92},[23,now]_{114}\rangle\}$
S_8	5	*Industry.company.dept.staff.stats.title*	26	$\{\langle[0,now]_{26}\rangle\}$
S_9	2	*Industry.company.staff*	135, 157	$\{\langle[0,now]_{157},[0,22]_{135}\rangle\}$
S_{10}	3	*Industry.company.staff.name*	136, 158	$\{\langle[0,now]_{158},[0,22]_{136}\rangle\}$
S_{11}	3	*Industry.company.staff.stats*	146, 168	$\{\langle[0,now]_{168},[0,22]_{146}\rangle\}$
S_{12}	4	*Industry.company.staff.stats.salary*	147, 169	$\{\langle[0,now]_{169},[0,22]_{147}\rangle\}$
S_{13}	4	*Industry.company.staff.stats.title*	170	$\{\langle[0,now]_{170}\rangle\}$

Table 2 *LTlop*: LOP of data nodes in each level

表 2 *LTlop*:各层对应数据结点的 LOP

Level	*Tlop*
1	$\{\langle[0,now]_0\rangle\}$
2	$\{\langle[0,now]_{1,134},[3,now]_{68}\rangle\}$
3	$\{\langle[0,now]_{2,12,157,180},[0,22]_{135}\rangle,\langle[3,now]_{69,79}\rangle\}$
4	$\{\langle[0,now]_{13,158,168},[0,22]_{136,146},[0,20]_{36}\rangle,\langle[21,now]_{80},[23,now]_{102}\rangle\}$
5	$\{\langle[0,now]_{14,24,169,170},[0,22]_{147},[0,20]_{37,47}\rangle,\langle[21,now]_{81,91},[23,now]_{103,113}\rangle\}$
6	$\{\langle[0,now]_{25,26},[0,20]_{48}\rangle,\langle[21,now]_{92},[23,now]_{114}\rangle\}$

2 时态查询

基于结构摘要的时态 XML 索引,对 XPath 的查询主要是基于索引的结构连接和时态过滤,通过快速的结构连接和时态查询获取查询目标.下面分别介绍 TempSumIndex 中两个最主要的查询算法:结构连接算法和时态查询算法.

2.1 结构查询

结构连接操作包括祖先/子孙结构连接和父亲/儿子结构连接.根据时态编码和时态结构关系的定义以及定理 1 可以得到如下两个基本结构连接操作:

算法 2. *getDescendants*(*inQue*,*label*). //对于当前队列 *inQue* 中每个结点 *n*(*TCode*,*TPlabel*),查找以 *label* 为语义标签的子孙结点

Step 1:对于 *inQue* 中的每个结点 *n*(*TCode*,*TPlabel*):

Step 1.1:在 *TSum* 中找到路径标签为 *TPlabel* 的时态摘要结点,并读取对应的 $DTCodes_i$.

Step 1.2:对 $DTCodes_i$ 执行二分查找,获取时态编码为 *TCode* 的元素 $DTCodes_i[i]$.

Step 1.3:获取 *n* 的具有相同语义标签的同层直接时态后继结点的时态编码 $sTCode=DTCodes_i[i+1]$.

Step 1.4:对于 *TSum* 中的每个满足 *TPlen*>*length*(*TPlabel*)时态摘要结点 S_i:

Step 1.4.1:若 *TPlabel*(S_i)是以 *TPlabel* 为前缀且以 *label* 结束;

Step 1.4.2:对 *DTCodes*(S_i)执行二分查找获取满足 *TCode*<*DTCodes*(S_i)[*i*]<*sTCode* 的编码;

Step 1.4.3:将(*DTCodes*(S_i)[*i*],*TPlabel*(S_i))放入 *outQue* 中.

Step 2:返回 *outQue*.

由时态结构关系的定义可知,算法 2 中,Step 1.2 和 Step 1.3 获取结点 *n* 的具有相同语义标签的同层直接时态后继结点的时态编码,Step 1.4.1 查找结点 *n* 的以 *label* 为语义标签的子孙结点所在的时态摘要结点,Step 1.4.2 则是在 Step 1.4.1 的基础上查找结点 *n* 的子孙结点.当 Step 1.4 中的路径长度条件为 *TPlen*=*length*(*TPlabel*)+1

时,算法获取的是满足查询条件的孩子结点,相应的算法记为 *getChildren*(*inQue*,*label*).

算法 2 的时间复杂度:设 *inQue* 中的结点个数为 *k*,*n*(*TCode*,*TPlabel*)对应的时态摘要结点中的 $DTCodes_i$ 的元素个数最大为 *n*,则 Step 1.2 的复杂度为 *O*(log*n*),设 *TSum* 中共有 *m* 个时态摘要结点,则 Step 1.4 的复杂度为 *mO*(log*n*),所以算法 2 的复杂度为 *kmO*(log*n*).

算法 3. *getAncestors*(*inQue*,*label*). //对于当前队列 *inQue* 中的每个结点 *n*(*TCode*,*TPlabel*),查找以 *label* 为语义标签的祖先结点

Step 1:对于 *inQue* 中的每个结点 *n*(*TCode*,*TPlabel*):

Step 1.1:对于 *TSum* 中每个满足 *TPlen*<*length*(*TPlabel*)的时态摘要结点 S_i:

Step 1.1.1:若 *TPlabel* 是以 *TPlabel*(S_i)为前缀,且 *TPlabel*(S_i)以语义标签 *label* 结束;

Step 1.1.2:对 *DTCodes*(S_i)执行二分查找,查找符合 *DTCodes*(S_i)[*i*]<*TCode*<*DTCodes*(S_i)[*i*+1]的元素;

Step 1.1.3:将(*DTCodes*(S_i)[*i*],*TPlabel*(S_i))加入到 *outQue*.

Step 2:返回 *outQue*.

算法 3 中的 Step 1.1.1 是查找结点 *n* 的以语义标签 *label* 结束的祖先结点所在的时态摘要结点,Step 1.1.2 则是在 Step 1.1.1 的基础上查找结点 *n* 的祖先结点.当 Step 1.1 中的路径长度条件为 *TPlen*=*length*(*TPlabel*)−1 时,算法获取的是满足查询条件的父结点,相应算法记为 *getParents*(*inQue*,*label*).

算法 3 的复杂度:设 *inQue* 中的元组个数为 *k*,*TSum* 中时态摘要结点的总数为 *m*,*DTCodes*(S_i)的元素个数最大为 *n*,则 Step 1.1.2 的复杂度为 *O*(log*n*),则算法 3 的复杂度为 *kmO*(log*n*).

2.2 时态查询

时态查询分为基于 TXPath 的时态查询和时态 XML 文档快照查询两大类.在 TempSumIndex 中,时态查询的核心思想是:对线序分枝的查询,根据线序分枝的特点可以快速地得到满足时态要求的结点;时态 XML 文档快照查询则主要基于索引结构中的 *LTlop* 表.

算法 4. *tempFilter*(*inQue*,*VT*(*Q*)). //在 *inQue* 中查找有效时间区间包含 *VT*(*Q*)的结点

Step 1:对于 *inQue* 中的每个结点 *n*(*TCode*,*TPlabel*):

Step 1.1:获取语义标签为 *TPlabel* 的时态摘要结点 S_i,若 S_i 已被处理过,则转 Step 1;

Step 1.2:获取 S_i 中所有数据结点的有效时间区间的线序划分 *Tlop*(S_i);

Step 1.3:对于 *Tlop*(S_i)中的每个线性分枝 L_i:

Step 1.3.1:如果 $VT(Q)\not\subseteq \max L_i$,则 L_i 内的所有元素都不满足时态查询条件;

Step 1.3.2:如果 $VT(Q)\subseteq \min L_i$,则 L_i 内的所有元素都满足时态查询条件,将相应的时态编码放入队列 *tempCodeQue* 内;

Step 1.3.3:否则,读取整个线性分枝 $L_i=\{P_1,P_2,\ldots,P_n\}$,对 L_i 中所有元素执行二分查找,选取满足 $VT(Q)\subseteq P_j \wedge VT(Q)\not\subseteq P_{j+1}$ 的时间区间 P_j,此时,线序分枝片段$\{P_1,P_2,\ldots,P_j\}$中所有元素都满足时态查询条件,将相应的时态编码放入队列 *tempCodeQue* 内.

Step 1.4:对 *tempCodeQue* 内的每个时态编码 *tempCode*,若 *tempCode* 同时包含在 *inQue* 中,则将(*tempCode*,*TPlabel*)放入 *outQue*.

Step 2:返回 *outQue*.

由于不同的数据结点会被包含在同一时态摘要结点内,算法 4 中的 Step 1.1 确保同一时态摘要结点内的线序划分不会被重复处理,Step 1.3 是在线序分枝中查找满足符合时态约束的结点,Step 1.4 是确保结果集中所有满足时态约束的结点为查询队列 *inQue* 中的结点.由 Step 1.3 中的线序分枝查询过程可知,给定时间区间集 Γ,当线性分枝越少时,单个线性分枝包含的时间区间越多,通过 Step 1.3 可以确定或者排除的满足时态约束的数据结点就越多,相应的时态查询效率也就越高.这就是在第 1.2 节中研究极小线序划分的原因.

算法 4 的复杂度:假设 Step 1.1 确定的需要处理的时态摘要结点共有 m 个,而每一个时态摘要结点的线序划分包含 b 个线序分枝,则 Step 1.3 的时间复杂度为 $O(\log n)$,其中,n 是 *LOB* 中的元素个数.所以,算法 4 的复杂度为 $mbO(\log n)$.

时态 XML 文档的主要目的是为了保存历史的 XML 文档的情况,对过往历史任一时刻的 XML 文档的信息,可以通过时态 XML 文档进行查询,这就是时态 XML 文档查询的另一种查询——时态 XML 文档快照查询.算法 5 是相应的时态 XML 文档快照查询算法.

算法 5. *SNQuery*(*t*). //查询时态 XML 数据文档 *TD* 在给定时刻历史 *t* 的快照

Step 1:令 *level*=1.

Step 2:取索引结构第 *level* 层的线序划分 *LTlop*(*level*),对于 *LTlop*(*level*)中的每个线序分枝 L_i:

Step 2.1:若 $t<VTs(\max L_i)$,转 Step 4;

Step 2.2:将 *t* 看作一个时间区间[*t*,*t*],调用时态查询算法 4 中的 Step 1.3 在线序分枝 L_i 中执行时态过滤,并将满足时态约束的结点的时态编码放入 *tempCodeQue* 队列内.

Step 3:令 *level*=*level*+1,若 *level*≤*treeHeigtht*(索引树的高度),则转 Step 2.

Step 4:调用映射表 *TDcode* 获取 *tempCodeQue* 中的时态编码对应的数据结点并放入 *outQue* 中.

Step 5:输出 *outQue*.

算法 5 的时间复杂度为 $treeHeigtht\cdot b\cdot O(\log n)$,其中,$b$ 为 *LTlop*(*level*)中的元素个数,n 为 *LOB* 中的元素个数.

例 4:给定查询条件“从时刻 21 至今一直就职于公司 C2 的职员的名字”,相应的 TXPath 语句为

//company[name='C2']//staff[VT='[21,*now*]']/name.

由于所有的查询都是从根结点开始的,所以首先将时态 XML 文档的根结点加入到队列 *inQue* 中,作为后续查询的输入:*inQue*.*put*(*root*.*getTempCode*(·),*root*);然后解析 TXPath 查询语句,并对解析产生的操作算子执行结构连接和时态查询;最后,利用结点编码映射表 *TDcode* 将当前获取的时态编码转为时态 XML 数据模型的数据结点(*getResults*(*TDcode*,*inQue*)).对上述 TXPath 查询语句解析后产生的操作算子及相关操作如下:

(1) *inQue*=*getDescendants*(*inQue*,“company”);

(2) *inQue*=*getChildren*(*inQue*,“name”);

(3) *inQue*=*valFilter*(*inQue*,“C2”);

(4) *inQue*=*getParents*(*inQue*,“company”);

(5) *inQue*=*getDescendants*(*inQue*,“staff”);

(6) *inQue*=*tempFilter*(*inQue*,“[21,*now*]”);

(7) *inQue*=*getChildren*(*inQue*,“name”);

(8) *results*=*getResults*(*TDcode*,*inQue*).

其中,函数 *valFilter*(*inQue*,*value*)的作用是查找 *inQue* 中值为 *value* 的结点.对于值过滤,涉及到值索引查询,本文不作深入介绍,只做简单的处理,直接对结点值依次匹配获取目标结点.

最终得到的是 *ID* 为 13 的数据结点,时态 XML 文档片段:⟨name ID=“13” VT=“[21,*now*]”⟩Bob⟨/name⟩.

3 时态更新

时态 XML 文档会随着时间的推进不断更新,相应的索引也需要进行动态管理和维护.由于时态 XML 的数据量较大,相应的索引结构通常采用增量式更新.TempSumIndex 的更新可以细化为 3 个过程:由待更新结点的语义标签确定其所在的时态结构摘要结点、调整相应的时态编码和线序划分.时态编码的调整只需满足时态编码的约束条件即可,可参照相关编码的调整方法[8,12,13],在此不再详述.下面主要讨论对线序划分的调整过程.

3.1 插入更新

结点 u_0 的插入过程可能会引起线序划分中相关线序分枝的分裂和新线序分枝的建立,基本思想为:从 L_1 开始查找时间区间 $VT(u_0)$应插入的线序分枝 L_i,将 $VT(u_0)$插入的同时将 L_i 中属于 $OUR(VT(u_0))$的分枝片段删除,

并放入 *subL* 中;如果 L_i 不存在,则 $VT(u_0)$单独构成一个线序分枝.如果 *subL* 不为空,则从 L_{i+1} 开始对 *subL* 做类似插入操作,直至 *subL* 为空.由于线序分枝片段 *subL* 一定位于 L_{i+1} 的左下方,插入 *subL* 的基本思想为:如果 L_{i+1} 中存在位于 $UL(subL)$或 $DR(subL)$中的元素,则将 *subL* 插入 L_{i+1} 中,然后删除位于 $OUR(subL)$的分枝片段并递归插入.具体插入过程如算法 6 所示.

算法 6. *InsUpdateLOP*(Δ,u_0). //返回插入结点 u_0 后的线序划分Δ

Step 1:令$\Delta=\{L_1,L_2,\ldots,L_k\}$,$i=1$;

Step 2:如果 $i>k$,则 $VT(u_0)$单独构成一个线序分枝 L_{k+1},转 Step 4;否则,

Step 2.1:如果 max$L_i\in UL(VT(u_0))$,通过二分法找到 L_i 中属于 $UL(VT(u_0))$的最后一个元素 P_j:

Step 2.1.1:如果 P_j 是 L_i 的最小元 minL_i,则 $L_i=\{P_1,P_2,\ldots,P_m,VT(u_0)\}$,转 Step 4;

Step 2.1.2:如果 $P_{j+1}\in ODL(VT(u_0))$,则 $i=i+1$,转 Step 2;

Step 2.1.3:如果 $P_{j+1}\in DR(VT(u_0))$,则 $L_i=\{P_1,\ldots,P_j,VT(u_0),P_{j+1},\ldots,P_m\}$,转 Step 4;

Step 2.1.4:如果 $P_{j+1}\in OUR(VT(u_0))$:

Step 2.1.4.1:如果 min$L_i\in OUR(VT(u_0))$,则 $L_i=\{P_1,\ldots,P_j,VT(u_0)\}$,$subL=\{P_{j+1},\ldots,P_m\}$,$i=i+1$,转 Step 3;

Step 2.1.4.2:如果 min$L_i\in DR(VT(u_0))$,通过二分法查找 L_i 中位于 $DR(VT(u_0))$的第 1 个元素 P_k,$L_i=\{P_1,\ldots,P_j,VT(u_0),P_k,\ldots,P_m\}$,$subL=\{P_{j+1},\ldots,P_{k-1}\}$,$i=i+1$,转 Step 3;

Step 2.2:如果 max$L_i\in ODL(VT(u_0))$,则 $i=i+1$,转 Step 2;

Step 2.3:如果 max$L_i\in DR(VT(u_0))$,则 $L_i=\{VT(u_0),P_1,\ldots,P_m\}$,转 Step 4;

Step 2.4:如果 max$L_i\in OUR(VT(u_0))$:

Step 2.4.1:如果 min$L_i\in OUR(VT(u_0))$,则 $VT(u_0)$单独构成一个线序分枝 L_{k+1},转 Step 4;

Step 2.4.2:如果 min$L_i\in DR(VT(u_0))$,则通过二分法找到 L_i 中位于 $DR(VT(u_0))$的第 1 个元素 P_k,$L_i=\{VT(u_0),P_k,\ldots,P_m\}$,$subL=\{P_1,\ldots,P_{k-1}\}$,$i=i+1$,转 Step 3;

Step 3:如果 $i=k$,则 *subL* 单独构成一个线序分枝 L_{k+1},转 Step 4;否则,令 $subL_{UL}=subL_{DR}=\varnothing$:

Step 3.1:如果 max$L_{i+1}\in UL$(max*subL*),则通过二分法查找 L_{i+1} 中位于 UL(max*subL*)的线序分枝片段,并放入 $subL_{UL}$ 中;

Step 3.2:如果 min$L_{i+1}\in DR$(min*subL*),则通过二分法查找 L_{i+1} 中位于 DR(min*subL*)的线序分枝片段,并放入 $subL_{DR}$ 中;

Step 3.3:如果 $subL_{UL}=\varnothing\wedge subL_{DR}=\varnothing$,则 *subL* 单独构成一个线序分枝 L_{k+1},转 Step 4;

Step 3.4:令 $subL=L_{i+1}-subL_{UL}-subL_{DR}$,$L_{i+1}=subL_{UL}\cup subL\cup subL_{DR}$,如果 $subL=\varnothing$,转 Step 4;否则,$i=i+1$,转 Step 3.

Step 4:算法结束.

算法 6 的时间复杂度:Step 1 和 Step 4 为常数时间复杂度;Step 2 和 Step 3 的时间消耗主要体现在对各线序分枝的依次判定和对线序分枝中特殊元素的二分查找,最坏情况下需要扫描所有的线序分枝.假设共有 k 个线序分枝,每个线序分枝平均包含 m 个元素,Step 2 和 Step 3 的时间复杂度均为 $kO(\log m)$.所以,算法 6 的时间复杂度为 $kO(\log m)$.

3.2 删除更新

结点 u_0 的删除过程可能会引起线序划分中相关线序分枝的合并和删除,假设 $VT(u_0)$所在的线序分枝为 $L_i=\{P_1,\ldots,P_j,VT(u_0),P_{j+1},\ldots,P_m\}$,该算法的基本思想是:首先在 L_i 中直接删除 $VT(u_0)$,同时,根据线序分枝的性质将 L_{i+1} 中位于 $DR(P_j)\cap UL(P_{j+1})$区域内的线序分枝片段 *subL* 插入到 L_i 中 $VT(u_0)$所在的位置.如果 *subL* 不为空且 L_i 不是最后一个线序分枝,则采用同样的处理方式将线序分枝片段 *subL* 整体从 L_{i+1} 中删除,依此递归处理.具体删除过程如算法 7 所示.

算法 7. *DelUpdateLOP*(Δ,u_0). //返回删除结点 u_0 后的线序划分Δ

Step 1:令$\Delta=\{L_1,L_2,\ldots,L_k\}$,$i=1$;$subL=\{VT(u_0)\}$.

Step 2:如果 $subL$ 不在 L_i 内,则 $i=i+1$,转 Step 2.

Step 3:设 $L_i=\{P_1,\ldots,P_j,subL,P_{j+1},\ldots,P_m\}$,将 $subL$ 从 L_i 中删除.

Step 4:如果 $subL=L_i$,则删除线序分枝 L_i,转 Step 6.

Step 5:如果 $i<k$,令 $subL_{DR}=subL_{UL}=L_{i+1}$:

Step 5.1:如果 P_j 存在,则通过二分查找 L_{i+1} 中位于 $DR(P_j)$的分枝片段,并替换 $subL_{DR}$ 中的元素;

Step 5.2:如果 P_{j+1} 存在,则通过二分查找 L_{i+1} 中位于 $UL(P_{j+1})$的分枝片段,并替换 $subL_{UL}$ 中的元素;

Step 5.3:令 $subL=subL_{DR}\cap subL_{UL}$:如果 $subL=\varnothing$,转 Step 6;否则,将线序分枝片段 $subL$ 插入到 L_i 中 P_j 和 P_{j+1} 之间的位置,$i=i+1$,转 Step 3.

Step 6:算法结束.

算法 7 的时间复杂度:Step 1,Step 2,Step 3,Step 4 和 Step 6 为常数时间复杂度;Step 5 的时间消耗主要体现在对各线序分枝的依次判定和对线序分枝中特殊元素的二分查找,最坏情况下需要扫描所有的线序分枝.假设共有 k 个线序分枝,每个线序分枝平均包含 m 个元素,Step 5 的时间复杂度均为 $kO(\log m)$.所以,算法 7 的时间复杂度为 $kO(\log m)$.

4 仿真和评估

实验环境为:奔 4CPU,主频 2.0GHz;主存 1GB;操作系统 Windows XP,开发环境 Eclipse 3.2.实验使用美国 NBA 球队球员资料时态 XML 仿真数据,其中,有效时间区间端点取非负整数,内部结点和叶结点平均时间跨度分别为 500 和 200.基本性能比较对象为 DOM 和 TempIndex.

由于 DOM 和 TempIndex 对索引数据的时间跨度不敏感,而 TXIDM[10]是基于结点时间区间的联通等价类的时态 XML 索引结构,因此在测试时间跨度对 TempSumIndex 查询性能的影响时,采用 TXIDM 作为比较对象.按照 XPath 规范,XML 查询可分为绝对路径查询(//A)和相对路径查询(A//B)两类,本文根据时态约束设计了 8 种时态 XML 查询类型:Q1,//A;Q2,A//B;Q3,//A[VT];Q4,A//B[VT];Q5,A[VT]//B;Q6,A[VT]//B[VT];Q7,TXPath 快照查询;Q8,时态 XML 文档快照查询.其中,Q1 和 Q2 形式上不涉及时态约束,但最终返回结果是⟨结点,有效期间⟩,所以从语义上同属于 TXPath;Q3~Q6 显式包含了时态约束的不同组合形式;Q7(TXPathSN)可看作 Q6 的特殊情形;Q8 为时态 XML 文档快照查询模版.对每种查询类型设计 100 条查询语句,统计结果取平均查询时间.

4.1 空间开销

本节主要测试索引结构 TempSumIndex 的空间开销,实验结果如图 4 所示.

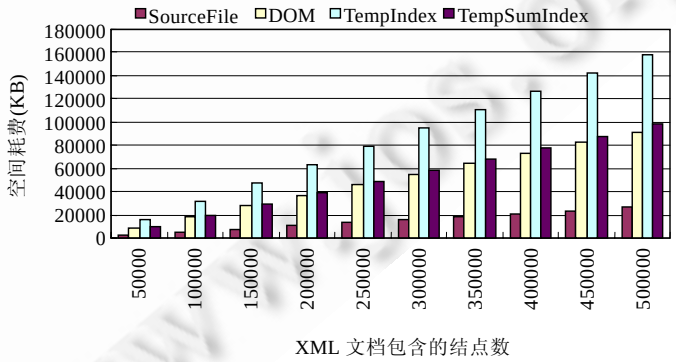

Fig.4 Space cost of index

图 4 索引空间开销

由图 4 可知,DOM,TempSumIndex 和 TempIndex 耗费的空间分别是 XML 数据源的 3.5 倍、3.8 倍和 6.1 倍.

TempSumIndex 和 TempIndex 空间耗费大体上呈线性变化趋势,两者都具较好的空间性能.TempSumIndex 空间消耗低于 TempIndex,这主要因为 TempSumIndex 中引入了时态编码和线序划分,使得索引结构相对简单,只包含时态结构摘要树、线序划分表和时态编码映射表;而 TempIndex 为了提高时态查询效率引入了复杂的存储结构,包括众多数据表,如 XML 数据源 DOM 结点哈希表、时态连续路径表和时态深度表(Delta 表)等.

4.2 查询效率

下面从不同方面测试 TempSumIndex 的查询性能.

4.2.1 基于不同的结点数

本节主要测试 TempSumIndex 基于不同的源数据结点数对查询类型 Q1~Q8 的性能,实验结果如图 5~图 12 所示.

由图 5 可知,TempSumIndex 和 TempIndex 基于类型 Q1 的查询性能相差不多,特别在小数据量的时候.这是因为 TempSumIndex 和 TempIndex 中维护路径结构信息的基本思想都是基于 1-Index 的结构摘要树.但是在结构连接处理方面,TempIndex 搜索范围是所有的连续路径,而 TempSumIndex 则将搜索限制在时态摘要结点范围内,从而缩小了搜索范围并提高了查询效率.类型 Q2 的查询处理方法和 Q1 类似,但涉及结构连接操作.由图 6 可知, TempSumIndex 对 Q2 的查询性能优于 TempIndex,主要是由于 TempSumIndex 中引入了可以实现高效结构连接的时态编码.TempSumIndex 和 TempIndex 基于类型 Q1 和 Q2 的查询性能均优于无索引的 DOM,这是因为无索引的 DOM 查询要遍历 DOM 树中的所有结点.

基于类型 Q3~Q6 的查询过程均分为两个操作:结构连接操作和时态查询操作,其中,结构连接操作的处理方式与查询类型 Q1(或 Q2)类似.由图 7~图 10 可知,TempSumIndex 的查询性能优于 TempIndex.这是因为在时态查询操作中,TempSumIndex 利用时态摘要结点中的线序划分进行时态过滤,对于线序划分中的每个线序分枝,可以通过二分查找获取满足时态约束的元素;而 TempIndex 则要扫描整个 Delta 表来判定结点是否在时态深度表对应的时态区间有效列表中.另外,TempSumIndex 中还引入了基于时态编码的高效结构连接操作.由于无索引 DOM 的结构连接和时态查询只能通过遍历 DOM 文档树,所以其查询性能较差.

类型 Q7(TXPath 快照查询)是 TXPath 查询的一种,也是 Q6(A[VT]//B[VT])的一种特殊情形.其中,A 和 B 的时态约束相同,且时态约束 VT 的时间始点与终点相同.Q7 最终的查询结果既满足 TXPath 中的结构约束又要满足 TXPath 的时态约束,所以最终的实验结果和 Q6 的查询性能相似,如图 11 所示.由图 12 可知,TempSumIndex 的时态 XML 文档快照查询性能比 TempIndex 稍差;但随数据结点增加,两者差异会逐步缩小.这主要是由于 TempIndex 通过其 Delta 表将同一深度的所有结点的时态区间进行分割,得到一个不相交时态区间集,相应的数据结点构成 TempIndex 中的 valid 表.TempIndex 获得时刻 t 的 valid 表的时间复杂度为 $treeHeigtht \cdot O(\log n)$;而基于 TempSumIndex 的时态 XML 快照查询算法复杂度为 $treeHeigtht \cdot b \cdot O(\log n)$.但是,随着数据源结点数的增加,两者的性能差异越来越小.而 DOM 的时态 XML 文档快照查询则要扫描整个时态 XML 文档,因此表现出最差的查询性能.

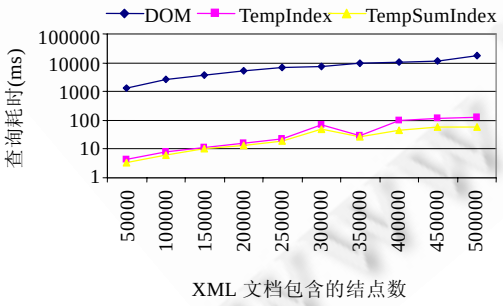

Fig.5 Query performance of Q1 (//A)

图 5 Q1(//A)的查询性能

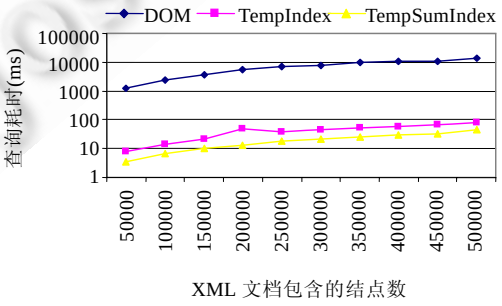

Fig.6 Query performance of Q2 (A//B)

图 6 Q2(A//B)的查询性能

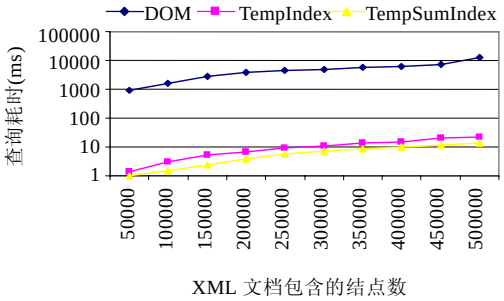

Fig.7 Query performance of Q3 (//A[VT])

图 7 Q3(//A[VT])的查询性能

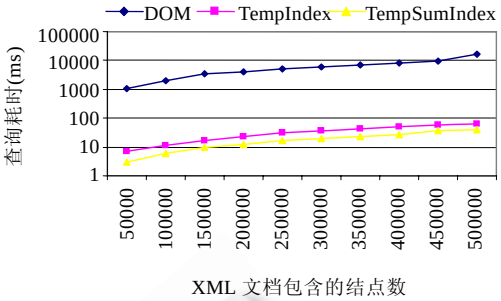

Fig.8 Query performance of Q4 (A//B[VT])

图 8 Q4(A//B[VT])的查询性能

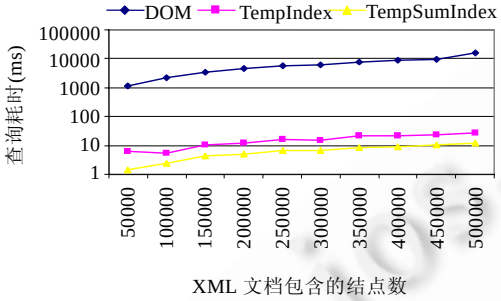

Fig.9 Query performance of Q5 (A[VT]//B)

图 9 Q5(A[VT]//B)的查询性能

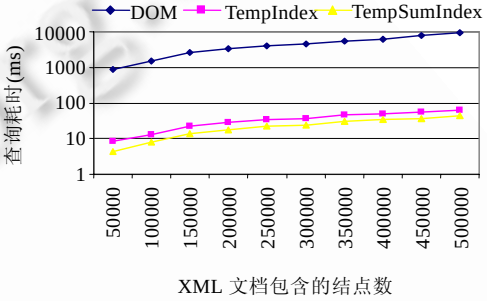

Fig.10 Query performance of Q6 (A[VT]//B[VT])

图 10 Q6(A[VT]//B[VT])的查询性能

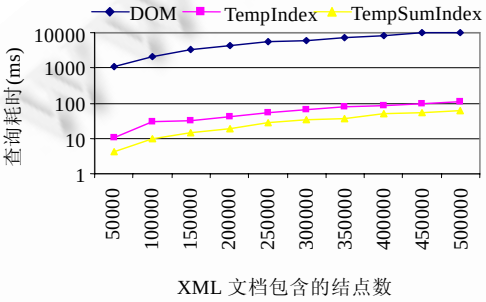

Fig.11 Query performance of Q7 (TXPath snapshot)

图 11 Q7(TXPath 快照)的查询性能

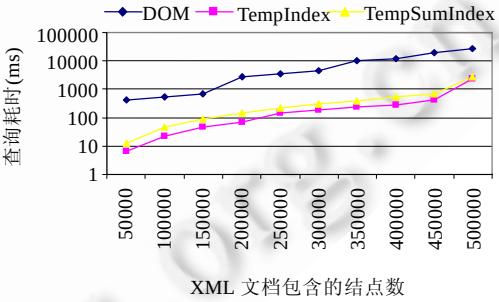

Fig.12 Temporal XML document snapshot query of Q8

图 12 Q8 时态 XML 文档的快照查询

4.2.2 基于不同的时态约束个数

由于时态索引结构的查询性能与查询条件中的时态约束有着密切联系,因此,本节设计了对查询语句“/A/B/C/D/E”时态约束递增变化的 6 类查询,在具有 300 000 个结点的时态 XML 数据源上进行查询,实验结果如图 13 所示.

由于 DOM 只能通过遍历 DOM 文档树,通过比较时态谓词进行时态过滤,所以 DOM 的查询性能随着时态约束个数增加保持稳定;而 TempIndex 和 TempSumIndex 查询性能随着时态约束个数的增加不断提升.这主要是由于通过时态过滤可减少结构连接操作中的结点数,提高了结构连接操作的性能,并减少了最后执行结果重构的结点数.

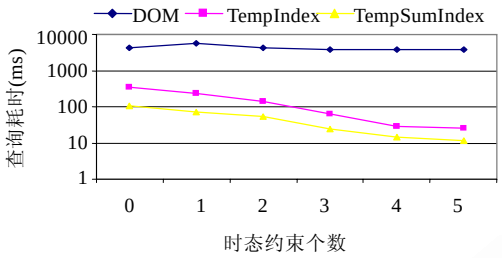

Fig.13 Relationship between query performance and temporal constrain

图 13 查询性能与时态约束的关系

4.2.3 基于不同的平均时间跨度

有效时间区间跨度(区间长度)是时间标签的基本特性,因此,有必要研究时间跨度对时态索引查询性能的影响.

本节以 TXIDM 为比较对象,并采用与文献[9]相同的实验数据.其中,叶结点平均时间跨度以 50 为间隔在 100~500 之间变化,内结点时间平均跨度以 100 为间隔在 200~1 200 之间变化.对每个跨度生成以 5 000 递增的 5 000~500 000 共 10 个数据源,对每个数据源分别查询 100 次,最后取查询过程中访问结点数的平均值,实验结果如图 14 所示.

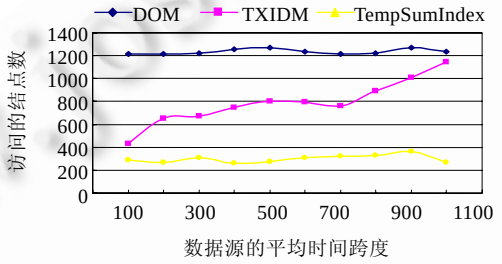

Fig.14 Relationship between query performance and average time span

图 14 查询性能与平均时间跨度的关系

由图 14 可知,随着时间区间跨度的增大,DOM 和 TempSumIndex 的查询性能相对稳定,而 TXIDM 查询性能则会有所下降.主要原因在于 TempSumIndex 是通过线序划分执行时态过滤,影响其时态查询性能的主要因素是线序划分中的线序分枝数,而不是数据源的时间跨度;而 TXIDM 则是基于时态连通关系执行时态过滤,当数据源时间跨度较大时,会形成较大的时态等价类,导致时态查询过程中访问较多的时间区间,降低了索引结构的查询性能.

4.3 更新效率

本节将索引结构 TempSumIndex 的增量式更新与索引重建(IndexCreation)进行比较,其中,插入和删除增量式更新分别标识为 Insertion 和 Deletion,实验结果如图 15 所示.

由于增量式更新只需调整部分结点,相对于索引重建,TempSumIndex 表现出较好的更新(插入和删除)性能;且随着数据结点的增加,其更新性能较稳定.

由图 15 可知,删除更新比插入更新具有更高的效率.这主要是因为结点的插入既需要对索引结构进行调整,又需要对相关结点的时态编码进行调整;而删除则只需调整索引结构.

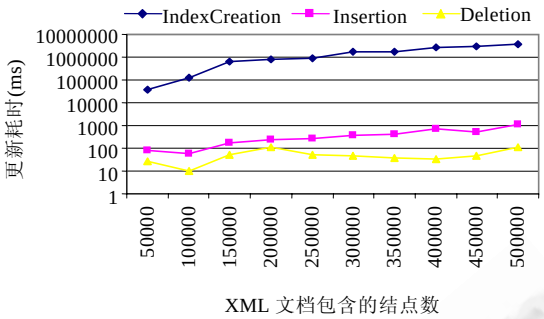

Fig.15 Performance comparison between incremental update and index reconstruction

图 15 增量更新与重建索引性能对比

5 结 语

随着网络技术的深入发展和快速推进,作为 XML 数据库和时态数据库技术相互交叉学科领域,时态 XML 数据管理技术正日益得到人们关注和重视.索引机制是提高时态 XML 数据查询性能的一项关键技术.相对于常规的 XML 索引技术,当前针对时态 XML 数据索引技术的研究工作较少.如何实现数据结点结构信息和时态信息的相互参照和有机整合,是一项具有挑战性的研究课题.本文在深入分析时态 XML 数据的时态结构信息和时态约束信息的基础上,引入一种具有高效查询性能的时态 XML 索引技术,并研究了具有“完备”性的索引增量式更新方法.实验结果表明,本文提出的时态 XML 索引结构具有高效的查询和更新性能.

References:

[1] De Capitani S. An authorization model for temporal XML documents. In: Panda D, ed. Proc. of the 2002 ACM Symp. on Applied Computing. New York: ACM Press, 2002. 1088−1093. [doi: 10.1145/508791.509006]

[2] Amagasa T, Yoshikawa M, Uemura S. A data model for temporal XML documents. In: Ibrahim MT, Küng J, Revell N, eds. Proc. of the 11th Int’l Conf. on Database and Expert Systems Applications. Berlin: Springer-Verlag, 2000. 334−344. [doi: 10.1007/3-540-44469-6_31]

[3] Dyreson CE, Bolen MH, Jensen CS. Capturing and querying multiple aspects of semistructured data. In: Atkinson MP, Orlowska ME, Valduriez P, Zdonik SB, Brodie ML, eds. Proc. of the 25th VLDB Conf. San Francisco: Morgan Kaufmann Publishers, 1999. 290−301.

[4] Wang F, Zaniolo C. Temporal queries in XML document archives and Web warehouses. In: Reynolds M, Scattar A, eds. Proc. of the 10th Int’l Symp. on Temporal Representation and Reasoning. Washington: IEEE Computer Society, 2003. 47−55.

[5] Wang F. XML-Based support for database histories and document versions [Ph.D. Thesis]. Los Angeles: University of California, 2004.

[6] Gao DF, Snodgrass RT. Temporal slicing in the evaluation of XML queries. In: Freytag JC, Lockemann PC, Abiteboul S, Carey MJ, Selinger PG, Heuer A, eds. Proc. of the 29th Int’l VLDB Conf. Toronto: VLDB Endowment, 2003. 632−643.

[7] Mendelzon AO, Rizzolo F, Vaisman A. Indexing temporal XML documents. In: Nascimento MA, Özsu MT, Kossmann D, Miller RJ, Blakeley JA, Schiefer KB, eds. Proc. of the 30th VLDB Conf. Toronto: VLDB Endowment, 2004. 216−227.

[8] Rizzolo F, Vaisman A. Temporal XML: Modeling, indexing, and query processing. The VLDB Journal, 2008,17(5):1179−1212. [doi: 10.1007/s00778-007-0058-x]

[9] Ye XP, Chen KY, Tang Y, Tang N, Hu S. Technology on temporal XML indexing. Chinese Journal of Computers, 2007,30(7): 1074−1085 (in Chinese with English abstract).

[10] Ye XP, Tang Y, Guo H, Chen LW, Zhu J, Chen KY. Study and application of temporal index technology. Science in China Series F: Information Science, 2009,39(12):1258−1270 (in Chinese with English abstract).

[11] Goldman R, Widom J. Dataguides: Enabling query formulation and optimization in semistructured databases. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA, eds. Proc. of the 23th VLDB Conf. San Francisco: Morgan Kaufmann Publishers, 1997. 436−445.

[12] Li Q, Moon B. Indexing and querying XML data for regular path expressions. In: Apers PMG, Atzeni P, Ceri S, Paraboschi S, Ramamohanarao K, Snodgrass RT, eds. Proc. of the 27th VLDB Conf. San Francisco: Morgan Kaufmann Publishers, 2001. 361−370.

[13] Luo DF, Meng XF, Jiang Y. Updating of extended preorder numbering scheme on XML. Journal of Software, 2005,16(5):810−818 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/16/810.htm [doi: 10.1360/jos160810]

附中文参考文献:

[9] 叶小平,陈铠原,汤庸,汤娜,胡苏.时态 XML 索引技术.计算机学报,2007,30(7):1074−1085.

[10] 叶小平,汤庸,郭欢,陈罗武,朱君,陈铠原.时态索引技术研究及其应用.中国科学(F 辑:信息科学),2009,39(12):1258−1270.

[13] 罗道锋,孟小峰,蒋瑜.XML 数据扩展前序编码的更新方法.软件学报,2005,16(5):810−818. http://www.jos.org.cn/1000-9825/16/810.htm [doi: 10.1360/jos160810]

郭欢(1984－),女,安徽砀山人,博士,主要研究领域为时态数据库技术及其实现.

汤庸(1964－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为时态数据库技术,协同工作软件.

叶小平(1955－),男,博士,教授,CCF 高级会员,主要研究领域为时空数据库,XML 数据库技术.

陈罗武(1983－),男,工程师,主要研究领域为时态 XML 数据库技术.