XPSort——树形数据多核并行外存排序算法

杨良怀[1],王 靖[1],周为钢[2],边继东[1]
(1.浙江工业大学计算机学院,浙江杭州 310023;2.杭州市公安局交通警察局科研所,浙江杭州 310014)

摘 要: XML数据处理中一个基本问题是树形数据排序.本文针对已有算法的不足提出了一种XML文档多核并行外存排序算法——XPSort.XPSort扫描XML文档产生相互独立的排序任务,利用多核CPU对任务进行并行处理;同时,利用数据压缩、单临时文件以及避免子树匹配等策略,有效地减少磁盘I/O,提高排序性能;它克服了NEXSORT算法没能有效利用内存空间、存在大量随机I/O的问题以及难以处理"右深树"的缺陷,也克服了HERMES的数据冗余、大量磁盘开销等缺点.文章对不同特性的XML文档开展了大量比较实验,结果表明XPSort优于已有算法,所提优化方法是有效可行的.

关键词: XML文档;树形数据;排序算法;并行算法
中图分类号: TP301 **文献标识码:** A **文章编号:** 0372-2112 (2014)02-0292-09
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2014.02.013

XPSort: A Multi-core Parallel Sorting Algorithm for Hierarchical Data in External Memory

YANG Liang-huai[1], WANG Jing[1], ZHOU Wei-gang[2], BIAN Ji-dong[1]
(1. *School of Computer Science and Technology, Zhejiang University of Technology, Hangzhou, Zhejiang* 310023, *China*;
2. *Institute of Traffic Control, Hangzhou Municipal Public Security Bureau Traffic Police Division, Hangzhou, Zhejiang* 310014, *China*)

Abstract: A fundamental problem in XML data handling is hierarchical data sorting. This paper focuses on this problem and proposes an effective sorting algorithm called XPSort for XML document. XPSort exploits multi-core CPU to parallelize the executions of the mutually independent tasks generated by scanning the XML document; and effectively reduces disk I/Os through data compression, single temporary-file storage and avoidance of tree-matching. XPSort overcomes the issues of inefficient space utilization, large number of random I/Os and inability to handle right-deep tree in NEXSORT, and avoids data redundancy and large disk I/O costs in HERMES. Extensive experiments on XML documents with different characteristics show that XPSort outperforms other existing sorting algorithms.

Key words: XML document; hierarchical data; sorting algorithm; parallel algorithm

1 引言

排序是数据库管理系统中的核心操作[1],排序算法一直是研究人员关注的议题[2].随着XML成为因特网数据表示、数据交换的标准,基于XML的应用和系统越来越多,如XML数据管理[3,4]、软件版本控制、数据归档[5,6]、变更检测[7]等.XQuery[8]是W3C的XML数据查询语言推荐标准,它可以实现对XML数据库或文档的灵活查询.对树形数据排序是XQuery提供的重要功能,其ORDER BY子句用以说明对XML数据进行排序,通常只对XML数据某一层子节点排序,而不对XML子节点对应的子树内部进行递归排序;若要实现对XML文档的完整排序,则需要利用DTD,递归地对每个XML片段进行排序.若一棵树的所有节点的子节点集都按照排序字段的某种顺序排列,则称这棵树是有序树.假设每个节点包含排序关键字(有关XML关键字的论述可参看文献[6]),树形数据的一个示意性排序例子如图1所示,图中假定关键字就是节点的标签.左边的树未经过排序,而右边的树则是对关键字进行排序后得到的排序树.

不同于扁平数据(flat data),树形数据的元素和元素间存在结构联系,如父子关系、兄弟关系等.树形数据的顺序指的是同一父节点下的各兄弟子节点之间的序关系,而与父节点无关.因此,对于树形数据,排序系指对每个节点的直系子节点集进行排序.这样,相同规模的树形数据与扁平数据相比,树形数据的内存排序复杂度要降低.然而,由于树形数据节点之间存在层次结构,其存取具有先后次序约束,不能随意分割.因此,对存储于磁盘的XML文档排序必须对文档进行顺序扫描,

收稿日期:2012-10-18;修回日期:2013-04-16;责任编辑:孙瑶
基金项目:国家自然科学基金(No.61070042);浙江省自然科学基金(No.Y1090096,No.Y13F020110)

不能随意对文档分割排序.对于大数据文档,排序过程中需要写出中间数据,排序的主要问题在于如何在保持节点间层次关系的同时减少磁盘 I/O 访问.另外,处理器多核化已是趋势,利用多核来提高树形数据排序算法性能是所期望的.由于树形数据读取有顺序性要求,给排序并行化带来了挑战.

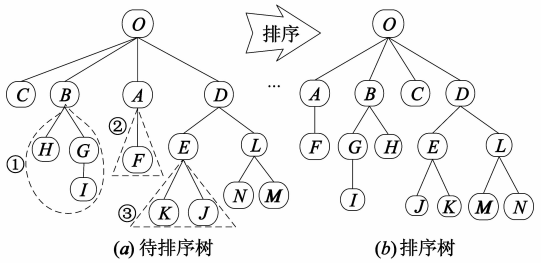


图1　一棵树及其相应排序树

为解决以上问题,本文提出了树形数据排序算法 XPSort(XML Parallel Sort),其贡献是:引入堆栈数据结构,避免子树匹配,降低内存树的维护代价;利用单文件存储中间结果、读/写缓存、数据压缩等策略,减少磁盘 I/O 次数;利用多核并行化树形数据进行排序工作,提高排序性能.所提算法克服了已有 XML 排序算法 NEXSORT[9] 没能有效利用内存空间、存在大量随机 I/O 访问的缺点以及难以处理"右深树"的缺陷,也克服了 HERMES[10] 算法的数据冗余、大量磁盘存取开销等缺点.

2　相关工作

扁平数据排序算法已有深入研究,文献[1,11]详细介绍了外部归并排序在关系数据库中的应用和实现.随着 XML 的广泛应用,树形数据的排序问题成为 XML 数据处理中的一个基本问题,研究高效树形数据排序算法具有重要应用价值.下面着重介绍树形数据排序的相关研究.

科学数据大多都具有良好的结构,在科学数据管理中,随着新数据的产生,科学数据要不断地更新、发布新版本.针对版本管理,Buneman 等[5] 提出了一种结构数据存储方案,通过嵌套归并多个版本的数据,并在合并后的新节点中记录该节点存在的版本号,不仅能够有效存储多版本的结构数据,而且还可以有效压缩多个版本数据的存储空间.在未排序的情况下归档某个版本,其每一个节点都可能需要遍历一遍原版本中的所有节点;而排序后则可以避免这个缺陷,加速归档.对结构数据排序也可加速变更检测,但目前 XML 文档变更检测方案大多未考虑排序带来的性能优化,停留在内存建树方式[7,12,13].

XMLTK[15] 描述了 XML 排序工具 XSort,它只对指定路径的元素按标准的外部排序算法进行排序,而不对其他层次的子树排序.XML 文档排序算法 NEXSORT[9] 由两个阶段构成:排序阶段和输出阶段.排序阶段通过深度优先遍历 XML 文档,每次获取一棵一定规模的子树进行排序;当所有子树完成排序后,按深度优先的方式从根节点开始输出子树,期间需要不断地访问外存得到相应排好序的子树.NEXSORT 在输出阶段需要随机存取磁盘中的子树,存在较为严重的随机 I/O,影响了排序的整体性能;对于超出内存限制的"右深"树,因无法构建一棵自然子树而不能排序,或由于虚拟内存交换而严重影响性能.

为减轻 NEXSORT 中严重随机 I/O,优化内存使用,Koltsidas 等[10] 提出了一种改进的排序算法 HERMES.不同于 NEXSORT,HERMES 排序时为每个内存树节点建立一个优先队列用于保存其子节点,采用替换选择法对子节点进行排序;最后,采用标准的外部归并法将得到的子树进行合并.HERMES 与其他树形排序算法相比,引入读/写缓存减少随机 I/O,改善了磁盘访问.但还存在以下不足:①排序阶段一次性只对一棵子树进行排序,不适合在多核环境下并行排序;②排序阶段产生的子树序列存在数据冗余,所有子树有相同的根节点,两棵子树间可能存在多个相同的节点;做为唯一标识节点,HERMES 需要为每个节点附加一属性,如该节点在 XML 文档中的位置,冗余数据不仅占用内存、磁盘空间,而且增加了 I/O 访问次数;③将排序阶段的子树序列分别保存在大量文件,增加了 I/O 随机性;④在归并阶段,HERMES 需要对每个子树序列在内存中重新建树,增加了排序时间.

综上所述,对于树形数据排序存在的多种需求,不同应用有不同的排序需要.本文讨论的树形数据排序问题与 NEXSORT 和 HERMES 中的相同,即对 XML 文档排序,排序递归地从根到叶节点逐层进行.现有树形数据排序方法存在大量随机 I/O,存储空间需要优化,没有考虑多核计算[16].所提 XPSort 算法用于解决以上不足.

3　XPSort

本节介绍树形数据排序算法 XPSort.如前所述,树形数据排序的基本单元是一个节点的直系子节点集.这些排序任务之间相互独立,是本文并行化算法设计的基本出发点.另一种可能的任务单位是一些相互独立的子树,但该方案存在确定这些独立子树规模的问题,分割子树会引起额外开销;不分割则引起负载失衡甚至内存不足,如一棵右深树.

3.1　XPSort 主要思想

XPSort 通过扫描 XML 文档,获取相互独立的排序任务(节点集),形成任务队列进行并行排序;在内存不

足时,输出已排序的节点集或子树到外存,在内存树相应节点记录节点集或子树在外存的位置,并释放相应内存,继续上述排序过程.在完成文档扫描、任务队列为空时,作最后的归并,输出一个有序的 XML 文档.

考虑到物理核负载均衡问题,XPSort 采用动态任务分配的方式将任务分配给线程.为避免线程动态创建和销毁的开销,XPSort 采用线程池机制:创建一组线程,等待接受任务;XPSort 循环使用线程池中的线程执行任务.在排序过程中,XPSort 扫描 XML 文档形成排序任务并传递给线程池,线程池等待空闲线程并负责分配任务给空闲线程.

3.2 XPSort 算法

3.2.1 基本概念

XPSort 扫描 XML 文档,按先序方式建立内存树.对应 XML 文档的每个元素,XPSort 在内存中建立 Node {String label; Attribute attrbutes[]; String text; Posi_info posi[]; Node children[]},其中 label 是 XML 元素的标签名;数组 attributes[]记录 XML 元素的属性;text 是 XML 元素的排序字段;数组 posi[]记录该节点已排序的部分子节点集在外存中的位置信息;数组 children[]记录该节点的子节点(元素).下面给出 XPSort 中用到的几个概念:

(1)直系子节点:称节点 C 是 N 的直系子节点,若 $C\in N.\text{children}[]$.

(2)完全扫描树:若子树 T 所有节点都已被扫描,且 T 在内存的节点数大于 1,则称 T 是一棵完全扫描树.例如,对于图 1(a)中的子树①$B\{H, G\{I\}\}$,若其所有节点已经扫描到内存,则称其为一棵完全扫描树.

(3)直系完全扫描树:若完全扫描树 T 的根节点是节点 N 的直系子节点,则称树 T 是 N 的一棵直系完全扫描树.例如,图 1(a)中的完全扫描树①$B\{H, G\{I\}\}$是节点 O 的直系完全扫描树.

(4)完全节点:若一个节点的所有直系子节点都已被扫描且全部位于内存,则称该节点是一个完全节点.规定叶节点不属于完全节点.

(5)结束标签对应子树:在 XML 文档中,由结束标签 < /label > 和与它对应的开始标签 < label > 之间的所有节点组成的子树,称为该结束标签对应子树.

(6)完全扫描树集合(TC):TC 是当前内存中的所有互不包含的完全扫描树集合,满足:$\forall T_1, T_2\in TC$,若 $T_1\subseteq T_2$(或 $T_2\subseteq T_1$),则只添加 T_2(或 T_1)到 TC 中.TC 中维护子树互不包含的目的是在将子树保存到外存时避免不必要的重复存储.因此,在将新获得的完全扫描树 T 添加到 TC 前,必须检查 TC 中已有子树与 T 的包含关系:$\forall T'\in TC$,若满足 $T'\subseteq T$,则从 TC 中剔除T',称这一过程为完全扫描树集合(TC)的相容性检查.

推论 1 若节点 N 是一棵完全扫描树的根节点,且其 posi 为空(所有子节点已扫描且在内存),则节点 N 必是一个完全节点.

推论 2 若内存树 T 不含完全扫描(子)树,那么 T 的任意层的节点集中最多只有一个非叶子节点,且位于该层节点集的最后一个位置.

证明 由于 T 是按照先序方式建立的内存树,当向 T 的任意层插入一个新节点时,该层的其余节点必定是叶节点或是一棵完全扫描树的根节点.新插入的节点总是被放置在该层节点集的最后一个位置.因 T 不含完全扫描树,所以对于 T 的任意层节点集,除了最后一个节点可能是非叶子节点外,其余节点均为叶子节点.

例子 待排序 XML 文档树如图 2(a)所示.假设内存可以容纳 8 个节点,当读到 XML 元素 G 节点子树的叶节点 I 后,$G\{I\}$子树作为第一棵完全扫描树加入队列进行排序;继续扫描,得到完全扫描树 $B\{H, G\{I\}\}$,

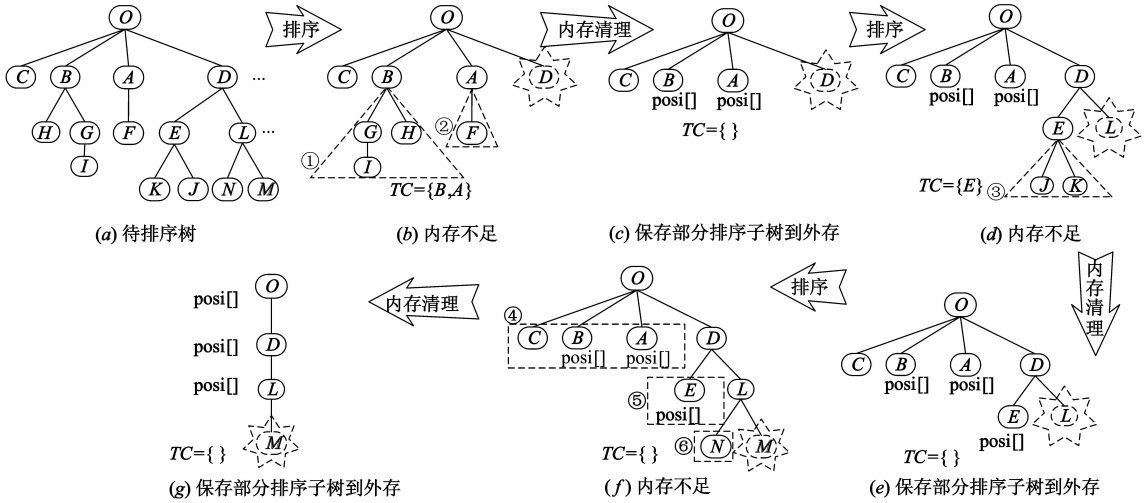


图2 XML文档树排序的一个例子

加入队列;当要读取 D 时,内存超出限额,如图 2(b)所示.此时,要对内存中有序子树进行清理,保存到外存,并在相应的内存节点记录相应子树在外存的地址,如图 2(c)所示.重复此过程,直到再次内存不足(如图 2(f));此时,不存在完全扫描树,需对各层叶子节点(除最右待扫描节点)进行排序,保存到外存,结果如图 2(g)所示.

3.2.2 XPSort 算法设计

XPSort 算法如算法 14 所示,基本流程是:若 XPSort 扫描到的元素是一个 XML 开始标签或者是一串纯文本,创建新节点并在内存树中插入;若内存空间不足,则先将部分内存树节点保存到外存,腾出空间,然后创建新节点并在内存树中插入(第 2~7 行),其中有关内存清理细节(第 5 行)将在稍后进行详细说明.

算法 1　XPSort 主过程

```
Algorithm. XPSort.
Input:XML 文档;
Output:有序的 XML 文档;
变量:TC:完全扫描树的集合;TQ:排序任务队列.
1.  扫描 XML 文档直到完成:
2.  读取一个 XML token(开始标签、结束标签或纯文本);
3.  if token∈{开始标签,纯文本}
4.    if out of memory
5.      OutputNodes(TC);//输出部分内存树节点
6.    endif
7.    为 token 创建节点并插入到内存树;
8.  else/* token 是一个结束标签 */
9.    n←T. root;//T. root 是 token 对应的子树 T 的根节点
10.   if n 是完全节点
11.     TQ←n;//等待空闲线程进行排序
12.     TC←T;//进行相容性检查
13.   else/* n 不是完全节点 */
14.     doMergeSort(n 的内存子节点);/* 归并排序 */
15.     保存排好序的内存子节点(及其子树)到外存;
16.     n 的 posi[]之中记录相应外存位置信息;
17.     释放 n 那些已经保存在外存的节点所占用的内存空间;
18.   endif
19.   if n 是内存树的根节点/* XML 文档扫描完毕 */
20.     等待 TQ 的所有任务排序完毕;
21.     LastMerge(n);
22.     Return;
23.   endif
24. endif
```

算法 2　输出中间结果,内存清理过程

```
Procedure. OutputNodes(TC)./* 将内存中的节点保存到外存 */
Input:TC:完全扫描树的集合;
Output:输出到同一外存文件.
符号:LN_level:第 level 层的所有叶子节点集合;
    TreeHeight:当前内存树的高度.
1. if TC≠∅
2.   等待 TQ 的所有任务排序完毕;
3.   for each tree T∈TC do/* 输出有序子树 */
4.     保存树 T 到外存;
5.     记录树 T 在外存的位置信息到 T. root 的 posi[];
6.     释放除 T. root 外子树 T 占用的内存空间;
7.   endfor
8.   清空 TC;
9. else /* TC = ∅ */
10.   for level = 2 to TreeHeight do
11.     LN_level←{level 层除最右待扫描节点之外的所有节点};
12.     doMergeSort(LN_level);
13.     保存 LN_level到外存;
14.     保存 LN_level的位置信息到 LN_level的父节点 posi[];
15.     释放 LN_level占用的内存空间;
16.   endfor
17. endif
```

算法 3　最终输出归并后的 XML 文档

```
Procedure.  LastMerge(Node sr).
Input:sr:子树的根节点;
Output:有序的 XML 文档.
1. 输出 sr 节点信息;
2. if sr. posi[]≠∅/* 有待归并外部子节点 */
3.   doMergeSort(sr. children);
4. endif
5. if sr. children[]≠∅
6.   for each node sn in sr. children;
7.     LastMerge(sn);
8.   endfor
9. endif
10. 释放 sr;
```

算法 4　归并排序

```
Procedure.  doMergeSort(Node nodes[]).
Input:待排序节点子集 nodes[];
Output:有序节点子集 nodes[];
全局常量:SCALE:节点集规模的阈值-常量.
1. if  nodes[]的节点个数 < SCALE
2.   MergeSort(nodes);
3. else
/* 并行归约,使用最小堆维护 nodes */
4.   ParallelMergeSort(nodes);
5. endif
```

若扫描遇到一个 XML 结束标签,则该结束标签对应子树 T 的所有节点都已被扫描.若 T 的节点数大于 1,则 T 是一棵完全扫描树.对每棵完全扫描树 T,其根

节点记为 $T.\text{root}$,令 $n \leftarrow T.\text{root}$,进行以下操作:

(1)若 n 是一个完全节点,则进行如下处理(第 9~12 行):①将 n 放入任务队列 TQ,表示 n 的子节点集是一个待排序的任务.只要存在空闲线程,就从 TQ 中获取任务进行排序;②将完全扫描树 T 添加到完全扫描树集 TC,检查 TC 的相容性.为了简化算法描述,TC 的相容性检查过程没有在算法 1~4 中详细列出,而是使用 $TC \leftarrow T$ 代替(第 12 行).相容性检查优化策略在 4.4 节叙述.

否则,n 不是一个完全节点,对其进行如下处理(13~18 行):①对 n 的内存子节点进行归并排序 doMergeSort;②保存到外存;③把所保存节点位置信息记录到 n 的数组 posi[]中.

(2)若 n 是内存树的根节点,此时,XML 文档扫描完毕,现在开始归并内存和外存上的节点,产生一个完整、有序的 XML 文档.具体操作如下(第 19~23 行):①等待 TQ 中的所有任务排序完成,保证最终保存到外存的所有子树都是有序的;②保存 n 到最终 XML 文档;③若 n 有子节点保存在外存,即 $n.\text{posi}[] \neq \varnothing$,则对当前根节点的子节点进行归并排序 doMergeSort,对排序后的子节点集逐个按深度优先次序扫描子节点,递归执行步骤②,直到所有节点输出完成.

为了防止并行排序中多线程运行开销大于直接排序的开销,具体实现中 doMergeSort 待排序节点数小于给定规模阈值 SCALE(在实验部分说明)时,采用普通归并排序算法 MergeSort 直接对节点进行排序;否则,使用并行归约方法 ParallelMergeSort 对节点集进行排序.并行归约根据 CPU 物理核个数 P,将节点集划分成 P 块节点集,为每块节点集分配一个线程执行排序操作,同时使用最小堆维护 P 块(已排序)节点集.

如前所述,随着扫描工作的进行,内存树中的节点不断增多,内存空间不断减少.当内存空间不足时,为了能在内存中继续保存扫描到的节点,必须将内存树中的部分节点保存到外存,以腾出内存空间,这个过程称为内存清理(XPSort 第 5 行,其实现过程 OutputNodes).

(1)若内存中存在完全扫描树,即 $TC \neq \varnothing$,则

①等待 TQ 中的所有任务排序完成;

②保存每棵完全扫描树 T 到外存,并将树 T 在外存的位置信息记录在 $T.\text{root}$ 的 posi[]之中;

③释放除 $T.\text{root}$ 外 T 所占用的内存空间;

④待 TC 中的所有完全扫描树输出完毕后,清空 TC.例如,图 2(b)的内存树经过内存清理后,完全扫描树 A、B 被保存到外存,TC 被清空,结果如图 2(c)所示.

(2)若 $TC = \varnothing$,则内存树中必定存在相当多的叶节点,且每个非叶节点都可能包含大规模子节点集.此时,进行内存清理,把各层叶节点排序后保存到外存.在节点集达到一定规模时,使用并行规约的排序方式对子节点集进行排序,加速内存清理的进度.算法执行如下操作:

①对第 level 层的所有叶节点 LN_{level},执行归并排序算法 doMergeSort,按序保存 LN_{level}的叶节点到外存并释放其内存空间;

②将 LN_{level}在外存的位置信息保存到它的父节点中(由推论可知,该父节点必为 level-1 层的节点集的最后一个节点,同时也是 level-1 层的唯一非叶节点).

例如图 2(f)的树不存在完全扫描树,除节点 O,D,L 外,其余节点均为叶节点.经过内存清理后,每一层的叶节点均被保存到外存,而内存中只剩下由每层非叶节点组成的一棵树.

3.3 IO 分析

对于外存排序,IO 开销是所有开销占主导地位的,可以借鉴 HERMES[10]中的 IO 开销分析方法.假设 XML 文档占 B 内存页,可用内存 M 页,其中一页做输出缓存.XPSort 需要扫描 XML 文档一次,中间写出部分排完序的节点,最后读入中间结果进行归并.归并过程中,对于那些子节点规模较大的节点,涉及外部归并,归并次数上限为 $\log_{M-1}\lceil B/(M-1)\rceil$.因此,总的 IO 开销是 $O(2B(1+\log_{M-1}\lceil B/(M-1)\rceil))$.

4 XPSort 的优化策略

4.1 XML 数据压缩策略

为了保持子树节点之间的结构关系,压缩策略在节点之间引入三个结构字符:“,”表示当前节点是上一个节点的子节点;“/”表示当前节点是上一个节点的兄弟节点;“.”表示当前节点是上个节点的父节点的兄弟节点.子树序列不再包含结束标签.

4.2 读写缓存策略

I/O 是影响 XPSort 性能的主要问题之一.为减少磁盘访问,XPSort 引入读写缓存,主要过程如下:(1)在内存中开辟读/写缓存;(2)在发生磁盘读操作时,先检查所需数据是否已经在读缓存中,若是,则从读缓存中取得;否则从磁盘中读取一块一定规模的连续数据到读缓存,再从读缓存中获得所需数据;(3)在发生磁盘写操作时,先将数据保存到写缓存,待写缓存达到一定规模时,再将数据一起写入磁盘.引入读写缓存可以减少排序过程中对磁盘的频繁访问,减少随机 I/O,从而减少 CPU 等待 I/O 的时间.

4.3 单临时文件策略

NEXSORT 和 HERMES 在清理内存时将每棵子树保存到各自临时文件.不同于此,XPSort 将所有子树保存

到单个临时文件中.使用单临时文件可以增加数据存取的有效性,还可省去维护多个临时文件的开销,如文件多次开启和关闭的开销.

在单临时文件策略中,每次向外存保存子树 T 时,记录 T 在临时文件中的偏移和包含的节点个数.当需要从临时文件中获取子树 T 时,根据偏移进行定位,然后根据节点个数读取节点.

4.4 相容性检查优化策略

在扫描 XML 文档的过程中,若内存中形成一棵完全扫描树 T,则将 T 添加到完全扫描树集合 TC 之前要进行相容性检查,剔除 TC 中满足如下条件的子树:$\forall T' \in TC$,若满足 $T' \subseteq T$.若通过遍历 T 的子节点来剔除 TC 中满足条件子树,子树之间包含关系的直接检查代价是 $O(nm)$,其中 n,m 是两棵子树的节点个数,开销较大.为了减小 TC 的维护代价,算法在建立内存树的过程中,记录从 T 根节点到当前节点的路径上各层节点包含的直系完全扫描树个数.在将 T 添加到 TC 前,由于内存树是按照先序的方式建立,所以只需要根据 T 包含的直系完全扫描树个数剔除 TC 尾端相应个数的子树即可,而不用对 TC 中的每棵子树 T' 判断是否满足 $T' \subseteq T$,检查代价是 $O(m)$.该 TC 维护策略,在添加 T 到 TC 时,可以避免判断 TC 中的每棵子树 T' 是否满足 $T' \subseteq T$ 的过程.

例如,在图 3 中,节点按先序方式插入到树 O,G 是待插入的节点,因此树 A、B、C、E、F、H 属于完全扫描树,而树 O、G 不属于完全扫描树.由于 $H \subseteq E$,所以此时完整扫描树集合 $TC=\{A,B,C,E,F\}$,记录当前路径包含的直系完全扫描树个数的堆栈 currPathStack = $\{O\ 3, D\ 2\}$,表示当前路径上节点 O 包含 3 棵直系完全扫描树(A、B、C),节点 D 包含 2 棵直系完全扫描树(E、F).若插入节点 H 后,树 E 成为一棵完全扫描树,则在添加 E 到 TC 前,需要先对 TC 进行相容性检查.因为 G 是叶子节点,所以插入 G 后节点 D 包含的直系完全扫描树个数保持不变.优化后的 TC 相容性检查的过程如下:①根据节点 D 包含的直系完全扫描树个数剔除 TC 中的子树,$TC=\{A,B,C\}$;②将树 D 添加到 TC,$TC=\{A,B,C,D\}$;③修改 currPathStack = $\{O\ 4\}$,表示此时节点 O 包含 4 棵直系完全扫描树.优化后的 TC 相容性检查复杂度为 $O(M)$,M 是内存可容纳节点数.

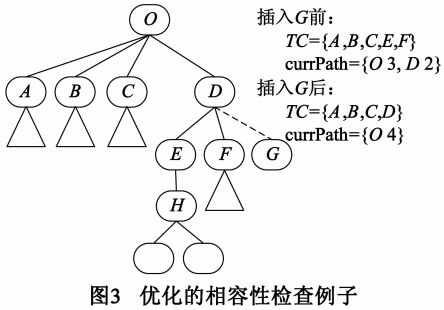


图3 优化的相容性检查例子

5 性能评价

本文利用 Xerces-C++ XML Parser,在 Windows 7 环境用 C++ 实现了 XPSort、HERMES 与 NEXSORT.实验中使用了 Intel 酷睿 2 四核 Q8200 处理器,2.33GHz 主频,4GB 内存,硬盘 WD 7200RPM.实验将根据不同特性的 XML 文档以及不同物理核数对 3 个算法进行比较,同时考察不同优化策略对 XPSort 的影响.

在对不含完全扫描树的内存树进行内存清理时,为防止并行排序时并行规约的开销大于直接排序的开销,待排序集需具有一定的规模,用 SCALE 来表示.我们通过直接排序和并行规约排序的比较实验来确定 SCALE.实验显示,节点集规模在 1000 时,直接排序时间是并行规约排序时间的 0.93 倍;在 2000、10000、100000 时,两者比值分别是 1.11、1.25、1.57 倍.可知,当排序数据集超过一定规模时,并行规约排序的优势随着节点集规模的增大而增大.因此,在后续实验中,SCALE 取 2000.

表 1 XML 生成器可调参数

参数	参数说明
MAX_NUM-元素个数上限	XML 文档最多包含的元素个数
MAX_DEPTH-深度上限	XML 文档的最大深度
DWN_RATE-深度增长率	XML 元素深度继续增长的概率
TXT_LEN-平均文本长度	XML 元素文本的平均长度
MULTI_VAL-扇出度增倍值	扇出度增长 MULTI_VAL 倍

5.1 数据生成

XML 文档的元素个数、树高、扇出度、文本长度等特性对排序算法会有不同的影响.为生成不同特性的 XML 文档,我们编写了 XML 生成器.XML 生成器采用先序方式生成 XML 元素,并为每一个 XML 元素随机生成标签名、属性名、属性值或者纯文本,XML 生成器的可调参数如表 1 所示,其中节点的扇出度在[20,40]之间均匀分布,为模仿现实 XML 文档中存在部分较大扇出度的节点,节点扇出度以 0.001 的概率增长 MULTI_VAL 倍,所以在生成的 XML 文档中,节点的最大扇出度为 40 * MULTI_VAL.XML 文档规模上限由 MAX_NUM 控制.生成的 XML 文档如表 2 所示.

5.2 XML 文档规模的影响

本实验中,XPSort、HERMES、NEXSORT 在排序期间的可用内存大小为 100MB,IO 缓存大小为 4KB,分别对表 2(a)~(b)两组不同规模($10 \sim 120 \times 10^6$)、不同平均扇出度(60,600)的 XML 文档考察其性能,相应结果如

图 4(*a*)~(*b*)所示.可知,随着文档规模的增加,XPSort 的性能总是优于 HERMES 和 NEXSORT.在图 4(*a*)中,对 10×10^6 个元素的 XML 文档,HERMES、NEXSORT 的时间是 XPSort 的 1.5~3.5 倍;当文档规模增加到 80×10^6 时,HERMES、NEXSORT 的时间是 XPSort 的 1.6~2.9 倍.

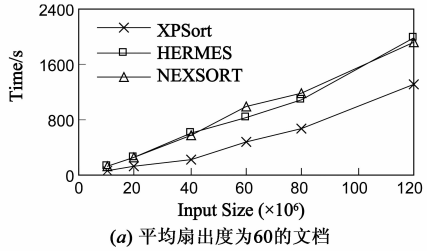
表 2 XML 文档及文档特征

(*a*)MAX_DEPTH = 6, DWN_RATE = 0.999, TXT_LEN = 20, MULTI_VAL = 1000

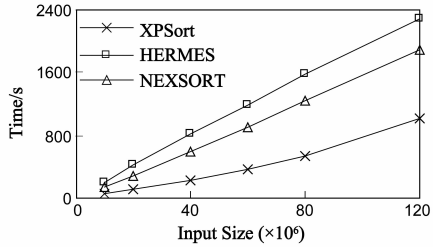
元素个数(million)	文件大小(MB)	平均扇出度
10	243	56
20	487	59
40	974	60
60	1454	61
80	1945	59
120	2918	60

(*b*)MAX_DEPTH = 6, DWN_RATE = 0.999, TXT_LEN = 20, MULTI_VAL = 20000

元素个数(million)	文件大小(MB)	平均扇出度
10	242	533
20	483	576
40	967	671
60	1443	639
80	1935	669
120	2908	548



(*a*) 平均扇出度为60的文档



(*b*) 平均扇出度为600的文档

图4 XML文档的规模对排序性能的影响

不同于 NEXSORT,XPSort 将读入的 XML 元素直接构建内存树,而不是放在数据堆栈中,省去了数据堆栈的维护开销以及对数据堆栈中的 XML 节点建树的开销;另外,XPSort 在内存空间不足时才清理内存,当内存空间足够大时,可以使保存到外存上的子树规模更大,从而减少排序期间对磁盘的访问次数.

HERMES 采用替换选择的方式排序,整个排序阶段都需要维护每个兄弟节点集,区分当前可输出和不可输出的节点,保证每次从节点集中输出的节点不会破坏当前输出的子树序列的有序性.HERMES 每次向外存保存一棵从 XML 根节点开始衍生的子树序列,这样许多节点会被多次重复保存到不同的序列文件中,导致外存存在较多的冗余节点,不仅增加了磁盘空间,而且增加了 I/O 开销;此外,为了判断外存中具有相同文本的两个节点是否是同一个节点,HERMES 算法必须为每个 XML 元素添加一个唯一标识符(如在文档中的位置信息),增加了磁盘开销,也导致 I/O 增多.不同于 HERMES,XPSort 毋须维护兄弟节点集,不需要为每个 XML 元素分配唯一标识符,在内存空间不足时才保存完全扫描树到外存,最大程度地减少了数据冗余.所以,XPSort 比 HERMES 具有更好的性能.

另外,多线程和单临时文件等策略也使 XPSort 性能优于 NEXSORT 和 HERMES.

5.3 物理核数的影响

通过调整线程池工作线程数量 n(≤物理核数)来考察物理核数对 XPSort 性能的影响.由于 IO 时间是一样的,且占统治地位,为了更清楚显示结果,实验中将这部分时间去除,只计算 CPU 部分排序时间.为方便,XPSort(n)表示线程数为 n.当 $n=1$ 时,XPSort 成为串行排序算法.实验中,可用内存大小为 100MB,排序对象是表 2(*a*)的 XML 文档,结果如图 5 所示.对于 80×10^6 个元素的文档,XPSort(1)的排序时间是 XPSort(4)的 3.1 倍.

随着文档规模的增加,多线程 XPSort(n)的排序性能优于单线程 XPSort(1).其原因是多线程 XPSort(n)能够边扫描文档边排序,扫描与排序存在较多重叠的时间片,从而缩短了 XPSort 单独的排序时间.而单线程 XPSort(1)的文档扫描和节点集排序是串行进行的,两者没有重叠的时间片,从而其排序时间多.

另一方面,由于 I/O 限制,文档扫描速度(排序任务产生)远小于 CPU 对节点集的排序速度,无法为更多工

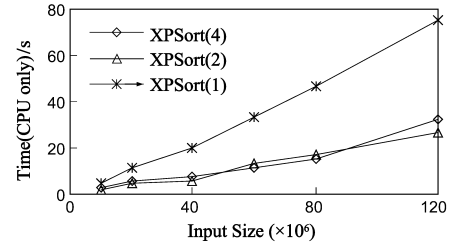


图5 物理核数对排序性能的影响

作线程提供排序任务.大部分时间里,XPSort 只有两个线程处于工作状态:一个线程负责文档扫描,产生排序任务;另一线程进行排序操作.因此,在图 5 中,XPSort(2)和 XPSort(4)的排序时间无明显差异.

5.4 优化策略的影响

为考察四种优化策略对 XPSort 的影响,对表 2(a)中 XML 文档组,我们比较了 XPSort 使用优化策略前后在空间和时间上的变化.结果如图 6~8 所示.

(1)数据压缩的影响

图 6 表明 XPSort 采用压缩策略后在排序中产生的临时数据占磁盘空间更少;相应地,其排序时间更少,见图 7.压缩策略使用 1 个结构字符代替占用多个字节的结束标签,减少了每棵子树在外存中的存储空间,提升了排序性能.对 80×10^6 个元素的 XML 文档(1.9GB)排序时,不采用压缩策略的 XPSort 产生的临时文件大小为 2.6GB,排序时间为 802s;采用压缩策略后,临时文件大小为 1.8GB,排序时间为 687s,前后减少了 30% 的空间和提升了 14% 性能.

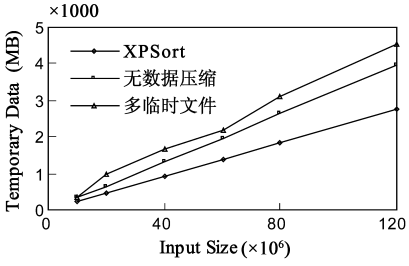


图6　优化策略对空间的优化

(2)单临时文件策略的影响

XPSort 采用单临时文件策略后产生更小的临时文件,见图 6;由此获得的性能改进如图 7 所示.与采用多临时文件策略相比,将所有子树保存到一个临时文件可使数据在磁盘上的存储变得更加紧凑,磁盘空间的占用更少;另外,单临时文件策略还可避免对多个临时文件的维护开销,如多次开启和关闭文件.实验中,对 80×10^6 个元素的 XML 文档排序时,会向外存写出 40 万棵子树,使用单临时文件策略,磁盘中只会增加 1 个临时文件,占用磁盘空间为 1.8GB,排序时间为 687s;若使用多临时文件,磁盘中会增加 40 万个临时文件,占磁盘空间 3GB,是单临时文件策略的 1.66 倍,排序时间为 5154s,是单临时文件策略的 7.5 倍.

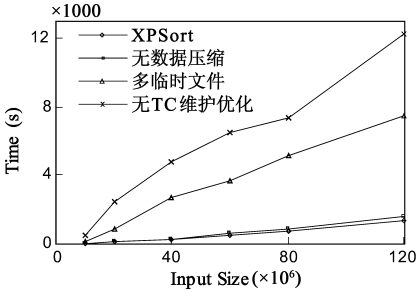


图7　优化策略对时间的优化

(3)相容性检查优化策略的影响

相容性检查优化策略使 XPSort 性能得到了较大提升,对 80×10^6 个元素的 XML 文档实验结果见图 7,使用相容性检查优化策略前后的排序时间相差 10.7 倍.

(4)读写缓存策略的影响

通过调整 IO 缓存的大小(0K~12K)考察对 XPSort 的影响,实验结果如图 8 所示.在对元素个数是 80×10^6 的 XML 文档排序时,在不使用读写缓存的情况下,每次读、写外存数据都需要直接访问磁盘,加上大量 I/O 随机访问,严重影响排序性能,此时的 XPSort 时间是使用 4KB 缓存策略的 3.77 倍.但在使用读写缓存的情况下,缓存大小的变化对 XPSort 的性能影响不大,其原因是虽然随着缓存的增加,XPSort 对磁盘的寻址次数会减少,由于 I/O 访问总量保持不变,排序的整体时间变化不大.

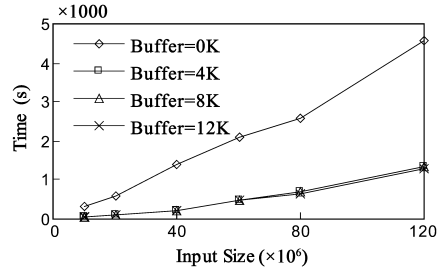


图8　读写缓存大小对XPSort的影响

此外,还进行了其他实验,限于篇幅仅对实验结果作扼要叙述.XML 文档平均扇出度对 XPSort 性能有影响,XPSort 性能随文档中要写出的子树数增多而降低;XML 元素的平均文本长度越长,XPSort 的排序开销越大;XML 深度的变化对 XPSort 的性能影响不大.

6 结束语

本文提出的 XML 文档外存并行排序算法 XPSort 是对 NEXSORT 与 HERMES 的改进,它克服了 NEXSORT、HERMES 存在的缺点.实验结果表明 XPSort 性能上优于已有算法,所提优化策略有效可行.

进一步的挑战性工作是如何克服 XML 文档的顺序扫描约束,从而增加排序并行度.

参考文献

[1] G Graefe. Implementing sorting in database systems[J]. ACM Computing Survey, 2006, 38(3): 1 – 37.

[2] M-C Albutiu, A Kemper, T Neumann. Massively parallel sort-merge joins in main memory multi-core database systems[J].

PVLDB,2012,5(10):1064 – 1075.
[3] 徐海渊,吴泉源,王怀民,贾焰.基于相容关系的 XML 索引机制[J].电子学报,2003,31(8):1155 – 1159.
H Y Xu,Q Y Wu,H M Wang,Y Jia.Containment based XML indexing[J]. Acta Electronica Sinica, 2003, 31(8): 1155 – 1159.(in Chinese)
[4] 衡星辰,覃征,邵利平,曹玉辉,高洪江.基于两阶段查询重写的 XML 近似查询算法[J].电子学报,2007,35(7):1271 – 1278.
X C Heng,et al.Two phase query rewriting based approximate XML query algorithm[J]. Acta Electronica Sinica, 2007, 35(7):1271 – 1278.(in Chinese)
[5] P Buneman,et al.Archiving scientific data[J].ACM Transactions on Database Systems,2004,29(1):2 – 42.
[6] P Buneman, et al. Keys for XML[J]. Computer Networks, 2002,39 (5):473 – 487.
[7] S Chawathe,et al.Change detection in hierarchically structured information[A].Proceedings of the ACM SIGMOD Conference[C].New York:ACM,1996.493 – 504.
[8] XQuery 1.0:An XML Query Language[OL].http://www.w3.org/TR/xquery.
[9] A Silberstein, J Yang. NEXSORT: Sorting XML in external memory[A].Proceedings of the ICDE Conference[C].Piscataway,NJ:IEEE,2004.695 – 707.
[10] I Koltsidas, H Müller, S Viglas. Sorting hierarchical data in external memory for archiving[A].Proceedings of the VLDB Conference[C].New York:ACM,2008,1(1):1205 – 1216.
[11] G Graefe.Query evaluation techniques for large databases[J]. ACM Computing Survey,1993,25(2):73 – 169.
[12] G Cobena,S Abiteboul,A Marian.Detecting changes in XML documents[A].Proceedings of the ICDE Conference[C].Piscataway,NJ:IEEE,2002.41 – 52.
[13] Y Wang,D J DeWitt,J Y Cai.X – Diff:An effective change detection algorithm for XML documents[A].Proceedings of the ICDE Conference[C].Piscataway,NJ:IEEE,2003.519 – 530.
[14] L Zheng,PÅLarson.Speeding up external mergesort[J].IEEE Transactions on Knowledge and Data Engineering, 1996, 8(2):322 – 332.
[15] I A Campillo, et al. XMLTK: An XML toolkit for scalable XML stream processing[A].Proceedings of PLAN-X: Programming Language Technologies for XML[C].Pittsburgh, PA,2002.1 – 10.
[16] 李文石,姚宗宝.基于阿姆达尔定律和兰特法则计算多核架构的加速比[J].电子学报,2012,40(2):230 – 234.
W S Li,Z B Yao.Multicore architecture speedup computation based on Amdahl's law and Rent's rule[J].Acta Electronica Sinica,2012,40(2):230 – 234.(in Chinese)

作者简介

杨良怀 男,1967 年生于浙江新昌.毕业于北京大学,获博士学位,浙江工业大学计算机学院教授,主要研究方向为数据库系统、数据挖掘.在 VLDB J,J of Info and Soft Tech,J of Sys and Soft,《计算机研究与发展》、《软件学报》、《中国工程科学》等期刊以及 VLDB、SIGKDD、CIKM、DASFAA 等数据库领域重要会议上发表学术论文 50 余篇.E-mail:yanglh@zjut.edu.cn

王 靖 男,1987 年出生于浙江温州.浙江工业大学硕士研究生,研究方向为数据库系统.E-mail:wuxianwangjing@gmail.com