小 型 微 型 计 算 机 系 统
Journal of Chinese Computer Systems

2008年3月 第3期
Vol. 29 No. 3 2008

带Order子句的XQuery在XML流上的查询实现

吴晓勇，张 昱，孙东海

(中国科学技术大学 计算机科学技术系，安徽 合肥 230027)
(中国科学院 计算机科学重点实验室，北京 100080)
E-mail: yuzhang@ustc.edu.cn

摘 要：随着XML的广泛应用，使得作为XML文档查询语言的XQuery成为人们研究的热点问题. 将复杂XQuery在XML数据流上的查询应用于服务器/客户端模式来满足高效、实时查询的要求，所实现的XQuery查询原型系统XSIEQ支持嵌套、order子句的多关键字排序等. 一次典型的XQuery查询过程可分为XPath查询、查询后处理两个阶段，本文着重描述查询后处理过程，最后给出了XSIEQ和Qizx在查询后处理时间性能上的对比及分析.

关 键 词：XQuery查询；XPath；XML流；order子句

中图分类号：TP311　　**文献标识码**：A　　**文 章 编 号**：1000-1220(2008)03-0481-06

XQuery Processing on XML Stream with Order Clause

WU Xiao-yong, ZHANG Yu, SUN Dong-hai

(*Department of Computer Science & Technology, University of Science & Technology of China, Hefei* 230027, *China*)
(*Lab. of Computer Science, Chinese Academy of Science, Beijing* 100080, *China*)

Abstract: As XML has been widely used, XQuery, as a favorite language on XML query, has received considerable attention recently. In some applications especially data selecting and distributing, the results are required to be output immediately while parsing XML stream. In this paper, a server/client architecture is applied to resolving the problem of efficient and immediate evaluation of XQuery over streaming XML data. The original XSIEQ(XML Stream Query with Immediate Evaluation) system is extended and tested in order to support complicated XQuery queries including nested queries and order clauses with multiple keys. A typic process of XQuery query is divided into two stages: XPath query and query post-processing. Algorithms on query post-processing are focus of this paper, and the query performance comparison between XSIEQ and Qizx is given in the end.

Key words: XQuery query; XPath; XML stream; order clause

1 引 言

随着XML的迅速发展和广泛应用，XML数据的查询语言XQuery[1]逐步成为最流行的查询语言之一，目前已是W3C的正式建议. 以XQuery为查询条件的XML流查询研究尚处于发展阶段，一些基于XPath[2]的XML流查询引擎能支持简单的XQuery查询，如[9-11]. 其中[11]是本实验室开发的XML流查询引擎XSIEQ，XSIEQ系统支持的XPath特性包括：子孙轴"//"、孩子轴"/"、自身轴"."、属性轴"@"；通配符"*"、"@*"、文本text()等. 此外，还有一些专门的XQuery查询工具[3-6]，这些工具对XQuery查询的支持力度有限，不能满足日益增加的XQuery查询需求.

一次典型的XQuery查询过程可分为XPath查询、查询后处理两个阶段. XPath查询阶段提取XQuery中的XPath式作为条件，查询XML文档；查询后处理阶段将XPath查询得到的中间结果按要求进行过滤、排序和连接，并将最终结果输出. 因此，XQuery流查询更适用于数据选择分发等应用场景，如图1. 服务器负责XPath查询，客户端负责本地的查询

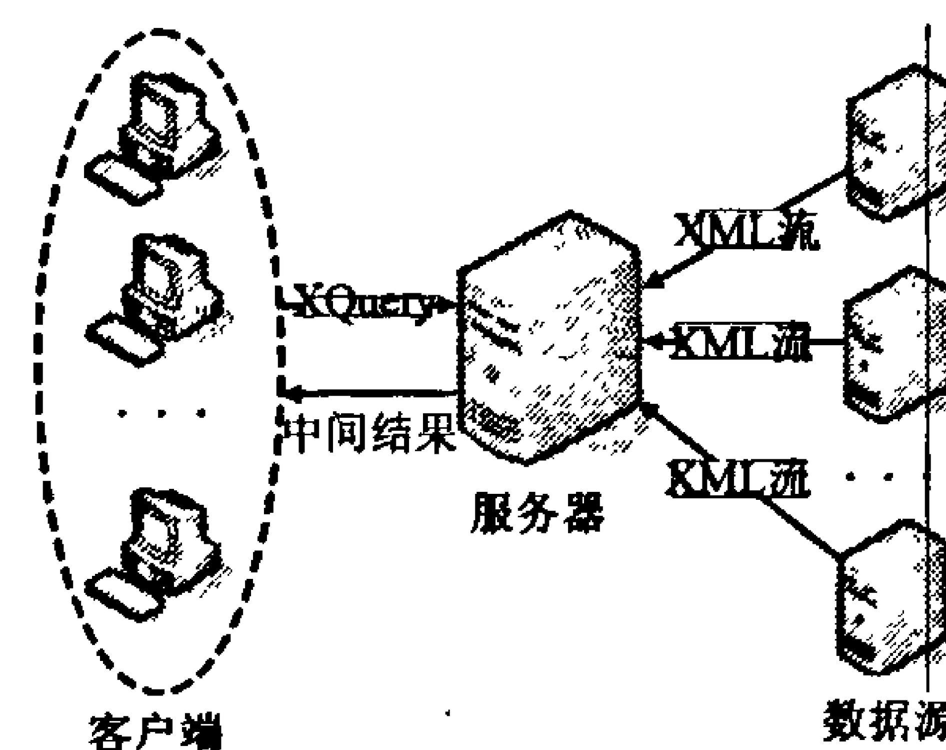

图1 系统应用场景
Fig. 1 Application of system

后处理，从而使两端计算负载平衡，同时服务器可以尽早输出中间结果.

在应用场景中，客户端如何高效地对中间结果进行处理

收稿日期：2006-11-13 基金项目：国家自然科学基金项目(60673126)资助；中国科学院计算机科学重点实验室开放课题基金(SYSKF0502)资助. 作者简介：吴晓勇，男，1977年生，硕士研究生，研究方向为XML数据处理；张 昱，女，1972年生，博士，副教授，研究方向为XML数据处理，程序设计语言理论和实现技术；孙东海，男，1982年生，硕士研究生，研究方向为XML数据处理.

万方数据

和文档重构是本文关注的焦点问题.本文的贡献如下:

(1) 在原有XSIEQ系统的基础上,对查询后处理模块进行了功能扩展,使其支持order子句的多关键字排序.

(2) 提出用原子表来保存中间结果,所提出的原子表连接和排序算法,具有较好的性能.

(3) 通过大量的实验,证明了后处理时间性能上的高效性.

2 预备知识

XQuery语言的核心是XPath式和FLWOR表达式. XPath用于访问XML文档的某一部分. XQuery中FLWOR表达式是由for、let、where、order、return五个关键字定义的子句构成的表达式.

定义1. 对任意作用在单文档上的FLWOR表达式,有且仅有一个根变量,对应文档的根节点.出现在for、let子句中的变量绑定称为变量定义,在where、order、return子句中出现的变量称为对变量的引用,被引用的变量称为结果变量.每个变量的关联XPath式分为两部分:前缀变量和路径. for子句迭代绑定序列中的数据项,依次将变量与每个数据项绑定的过程称为for绑定循环.

定义2. 对于一个FLWOR表达式中的某一变量v_j,如果v_j的定义为for v_j in v_i+p_j或者let $v_j:=v_i+p_j$,其中"+"表示串联,p_j是路径,v_i是变量,则称v_j直接依赖于v_i,记为:$v_i \xrightarrow{p_j} v_j$.依赖关系满足传递性,即:

$$v_i \xrightarrow{p_j} v_j, v_j \xrightarrow{p_k} v_k \Rightarrow v_i \xrightarrow{p_{jk}} v_k (transitivity\ rule)$$

其中p_{jk}是串联p_j和p_k得到的XPath式,称v_k传递依赖于v_i.任意变量v_k传递依赖于根变量,从根变量到v_k的路径称为v_k的导航路径.一般地,将直接依赖和传递依赖统称为变量绑定依赖.

例1. XQuery查询脚本示例*xq*.

```
for $a in doc(abcd.xml)//a, $c in $a/c , $d in $a/d
let $b:= $a/b, $e := $c/e
  order by $c,$d
return <result>
     {$a,$b,$c, $d}
    </result>
```

*xq*中变量*c*、*d*是排序关键字.变量*a*与*b*、*c*、*d*之间以及*c*与*e*之间具有直接依赖关系,*a*、*e*之间具有传递依赖关系,*a*是独立绑定变量,*b*、*c*、*d*、*e*是依赖绑定变量.

3 XQuery查询原型系统

3.1 支持力度

在选择XQuery子集时,忽略了一些次要研究点,这里主要考虑FLWOR查询表达式及其相关特性. XSIEQ支持的FLWOR子句的语法表示如下:

(1) ExprSingle::=FLWORExpr|OrExpr

(2) FLWORExpr ::= (ForClause | LetClause)+WhereClause? OrderClause? "return" ExprSingle

(3) ForClause ::= "for" "$" VarName "in" ExprSingle ("," "$" VarName "in" ExprSingle) *

(4) LetClause ::= "let" "$" VarName ":=" ExprSingle("," "$" VarName ":=" ExprSingle) *

(5) WhereClause ::= "where" ExprSingle

(6) OrderClause ::= "order" "by" OrderSpecList

(7) OrderSpecList ::= OrderSpec ("," OrderSpec) *

(8) OrderSpec ::= OrExpr OrderModifier

(9) OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest" | "least"))?

这里OrExpr包含了各种运算表达式:比较运算、算术运算、布尔运算、一元运算等.

3.2 相关数据结构

XQuery查询前构造了两种数据结构:变量依赖树和原子表集合.查询用到的所有变量和XPath式组织在变量依赖树中,XPath查询得到的中间结果组织在原子表集合中.

3.2.1 变量依赖树

在一个FLWOR表达式中定义的各变量根据依赖关系组织形成一棵变量依赖树.在单文档查询中,查询的XQuery集合中所有变量在同一棵树中.

定义3. 变量依赖树$VarTree=(V,E,P)$. V是节点集合,E是边集合,P是边标记集合. $\forall v_i \in V$,v_i表示变量,根节点表示根变量r;$e=(v_i,v_j)\in E$表示直接依赖$v_i \xrightarrow{p_j} v_j$,其中$p_j \in P$表示$v_j$的关联XPath式的路径.

由例1的脚本*xq*构造的VarTree如图2所示.

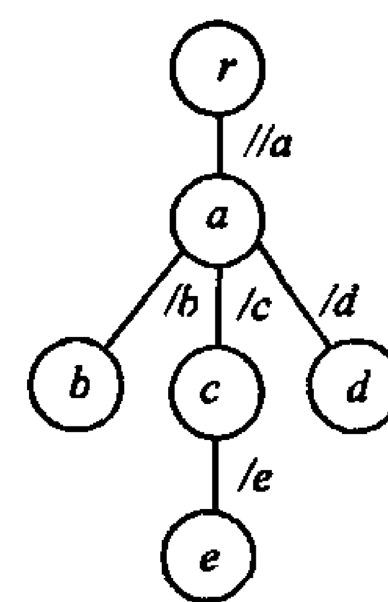

图2 例1的变量依赖树

Fig. 2 The vartree of example 1.

3.2.2 原子表

定义4. 一次查询中变量关联的XPath式匹配的值称为绑定序列.以单个变量作为属性,变量的绑定序列作为元组序列的表结构,称为变量表.对任意两个变量值val_1、val_2,如果val_1、val_2在XML文档中是祖孙关系(含父子关系),则称val_1、val_2之间具有上下文关系,变量表之间根据值的上下文关系进行的连接称上下文连接.根据各个变量表属性间的依赖关系,连接一个FLWOR式中for变量与结果变量的并集中的所有变量表得到用于构造结果的块表.

定义5. 原子表的属性由变量依赖树中的$n(n>0)$个变量组成,每个原子表有且仅有一个for变量属性,作为主码,其它属性列由直接依赖于该for变量的let变量组成,表的元组是由各变量的变量表连接而成.两张原子表之间具有直接依赖关系,当且仅当它们的主码之间具有直接依赖关系.如果原子表T_i、T_j之间满足$T_i \rightarrow T_j$,则对T_i的每个元组tup,T_j中

万方数据

都有一组元组 S_{tup} 与之存在上下文关系,称 S_{tup} 为原子表 T_j 的一个分组. *tup* 和 S_{tup} 中的每一个元组之间存在上下文索引.

在例 1 的脚本 *xq* 中,变量 *a*、*b*、*c*、*d*、*e* 组成的表是 *xq* 的一个块表,变量 *a*、*b* 和变量 *c*、*e* 分别组成原子表 A_{ab}、C_{ce},变量 *d* 组成原子表 *D*. A_{ab}、C_{ce} 之间,A_{ab}、*D* 之间都是直接依赖关系.

3.3　XQuery 查询系统的数据流程图

整个 XQuery 查询系统分为两个模块:XPath 查询处理模块和查询后处理模块,如图 3 所示.

XPath 查询处理模块以自动机对象、变量依赖树和原子表对象作为输入,XPath 查询处理模块按 XPath 式绑定的变量进行查询,将收集到的变量值连接为原子表元组,最后把填充过的原子表输出给查询后处理模块.

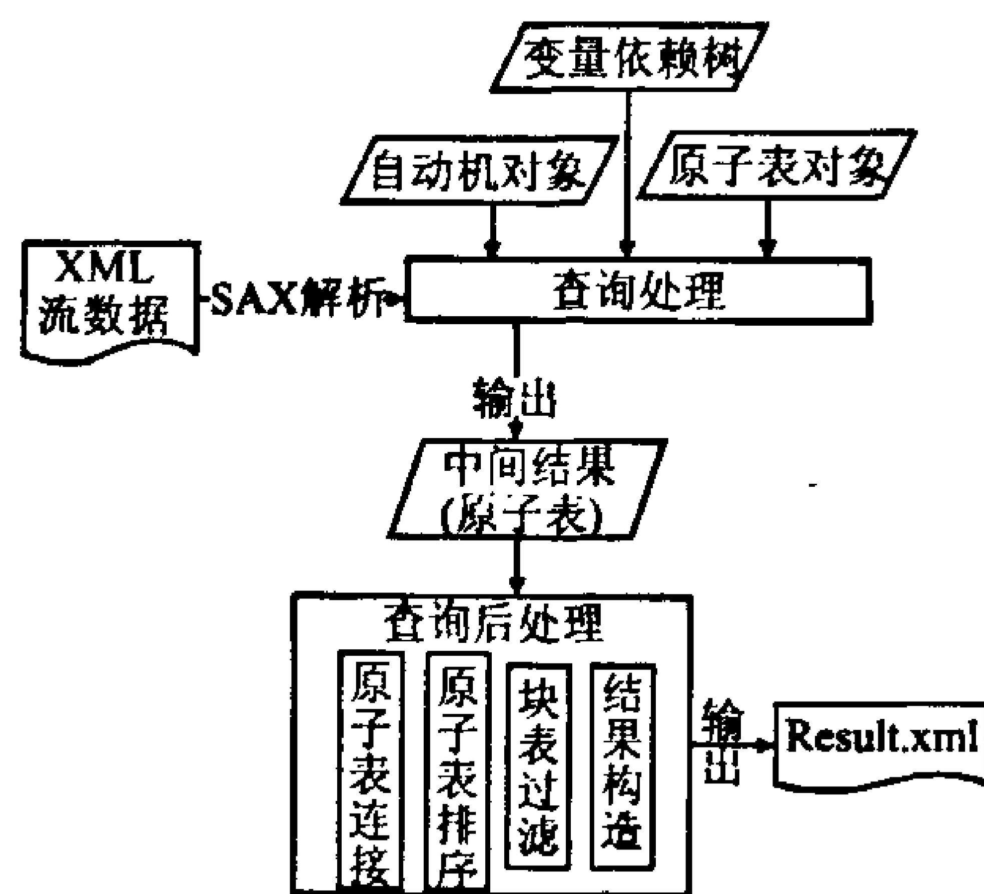

图 3　XQuery 查询系统数据流程图

Fig. 3　The data flow graph of query system

有关 XPath 查询处理模块的实现另文介绍,本文主要介绍查询后处理模块的实现.

4　查询后处理实现

一种朴素的后处理做法是:对原子表元组按上下文关系进行连接,形成块表,再对块表进行过滤、排序.由于原子表连接成块表后的元组规模增大,对元组排序的代价也会随之增加.

本系统提出通过改变排序的时机来减小排序规模的算法—JASIT 算法(Join And Sort In Time).即把朴素做法中原子表连接成块表后排序块表,改为先将第一个排序关键字所在原子表的所有元组排序,以后的排序关键字只要将与前面的排序关键字有上下文关系的元组排序,然后通过上下文索引进行连接,最后形成块表.

4.1　查询后处理框架

查询后处理数据流程如图 3 中查询后处理模块.

算法 1. 查询后处理框架

输入:中间结果,即原子表集合 *S*

输出:结果文档 result.xml

```
(1) S'=adjustTables(S);//对 S 进行预处理
(2) for each T in S' {
(3)   if (T is the first table){
(4)     sortTable(T);
(5)     //blockT 是原子表连接排序后得到的块表
        blockT=add(T) ;}
(6)   else
(7)     blockT=JASIT(blockT,T); }
(8) for each Tup in blockT {
(9)   if (satisfiedWhere(Tup) )
(10)    resultConstruct(Tup); //construct the results
    }
```

算法 1 中用到的函数有:

1. adjustTables(*S*):按主次排序关键字的 for 变量顺序调整对应原子表的顺序,对于非排序关键字的 for 变量对应的原子表,按 for 变量在 XQuery 表达式中出现的次序,排在排序关键字的 for 变量对应的原子表后.

2. sortTable(*T*):对 *T* 中的元组排序.

3. add(*T*):将 *T* 对应的变量作为块表的新的属性,*T* 中的元组作为属性值插入到 *blockT* 中.

4. JASIT(*blockT*,*T*):在 *blockT* 中找到与 *T* 有上下文关系的属性,根据该属性连接和排序 *T*,该函数返回经过连接和排序 *T* 后的块表.

5. satisfiedWhere(*Tup*):布尔函数,用于判断元组 *Tup* 是否满足 where 子句的条件,若满足则返回 true.

6. resultConstruct(*Tup*):对 *Tup* 进行结果构造.

算法第(4)、(5)行是对第一个原子表进行排序,并把该原子表对应的变量作为一个属性,把原子表内的元组作为属性值而组成块表 *blockT*.

算法第(8)～(10)行是对块表的元组按 where 子句的条件过滤后,进行结果构造.

下面给出算法第(7)行的 JASIT 算法.

4.2　JASIT 算法

JASIT 算法可以充分利用元组间已有的上下文索引,减少每次排序的元组规模,从而减少排序的代价.

4.2.1　JASIT 算法描述

算法 2. JASIT(*blockT*,*B*)算法

输入:块表 *blockT*,变量 *b* 对应的原子表 *B*.

输出:连接排序原子表 *B* 后的块表 *newBlockT*.

```
(1) //tempTable 是与 B 有上下文关系的临时表
    tempTable=getContextAttri(blockT,B);
(2) a=getRelatedVar(tempTable);
(3) for each tup_a in tempTable {
(4)   tup_t=getBlockTup(tup_a);
(5)   if(a、b is not brothers)
(6)     //取得 tup_a 在表 B 中的上下文分组
        S_tup=getContextTups(t_tup_a,B);
(7)   else{//a、b is brothers
(8)     C=getNearestAncestor(a,b);
(9)     //取得 tup_a 在表 C 中的上下文元组
        tup_c=getContextTups(tup_a,C);
(10)    //取得 tup_c 在表 B 中的上下文分组
        S_tup=getContextTups(tup_c,B);
      }
```

万方数据

```
(11)    sortTable(S_tup);
(12)    for each tup_b in S_tup{
(13)       blockTup=join(tup_t,tup_b);
(14)       //newBlockT 是连接排序 B 后得到的块表
           newBlockT=appendTup(blockTup);
           }
      }
(15) return newBlockT;
```

算法 2 中用到的函数有：

1. getContextAttri(*blockT*, *B*)：在 *blockT* 中找到与当前原子表 *B* 有上下文关系的属性和属性值，形成临时表并返回.

2. getRelatedVar(*tempTable*)：取得临时表 *tempTable* 对应的属性.

3. getBlockTup(tup_a)：取得临时表元组 tup_a 所在的 *blockT* 的元组.

4. getContextTups(*tup*, *X*)：取得元组 *tup* 在表 *X* 中的上下文分组.

5. getNearestAncestor(*a*,*b*)：取得变量 *a*,*b* 的最近公共祖先变量所对应的原子表.

6. sortTable(S_{tup})：对原子表的分组 S_{tup} 内的所有元组进行排序.

7. join(tup_t,tup_b)：将原子表元组 tup_b 连接到 *BlockT* 的元组 tup_t 的后面，成为 *newBlockT* 的一个元组.

8. appendTup(*blockTup*)：将连接排序后得到的块表元组 *blockTup* 插入到 *newBlockT* 中.

对于变量依赖树中的两个变量 *a* 和 *b*，*a*、*b* 之间的关系可以是：祖孙关系（包括父子关系）、子孙与祖先关系（包括孩子与父亲的关系）、兄弟关系.

当变量 *a*、*b* 的关系为祖孙关系或子孙与祖先的关系时，如算法 2 第 5)行，对于临时表 *tempTable* 中的当前元组 tup_a，找到 tup_a 在表 *B* 中的对应的分组 S_{tup}，对 S_{tup} 分组内的元组排序后（当变量 *a*、*b* 是子孙与祖先关系时，分组内只有一个元组，所以不用排序操作），将 tup_a 与 S_{tup} 分组内的每个元组进行连接，将连接后的元组作为 *newBlockT* 的一个元组.

当变量 *a*、*b* 之间的关系为兄弟关系时，如算法 2 第 7)行，对于临时表 *tempTable* 中的当前元组 tup_a，通过变量 *c* 绑定到的元组 tup_c 找到 tup_a 在表 *B* 中的对应的分组 S_{tup}，对分组 S_{tup} 内的元组进行排序后，将 tup_a 与 S_{tup} 分组内的每个元组进行连接，得到 *newBlockT* 的一个元组.

4.2.2 JASIT 算法的时间代价分析

本节讨论对两个存在有不同关系的原子表进行连接和排序的时间代价. 为便于描述，我们做以下假设：

(1) 假设变量 *a*、*b* 对应的原子表 *A*、*B* 的元组数分别为 *m*,*n*；当 *a*、*b* 为兄弟关系时，在变量依赖树中，*a*、*b* 的最近公共祖先为 *c*，对应原子表 *C* 的元组数为 *k*.

(2) 当原子表元组个数为 *x* 时，对这些元组排序的时间复杂度假设为 $O(x\log x)$.

(3) 两个元组之间连接的时间代价设为 O(1).

当变量 *a*、*b* 为祖孙关系时，不妨设 *A* 表中每个元组对应 *B* 表中分组的元组数相同，均为 (n/m)，则对 *A* 表所有元组排序的时间代价为 $O(m\log m)$；*A* 表连接 *B* 表的时间代价为 $O(n)$；对 *A* 表中每个元组在 *B* 表中的分组元组排序的时间代价为 $O((n/m)\log(n/m))$；*B* 表中共有 *m* 个分组，所以对 *B* 表中的所有分组排序的时间代价为 $O(n\log(n/m))$. 则 *AB* 总的排序连接的时间代价为 $O(m\log m+n+n\log(n/m))$.

当变量 *a*、*b* 为子孙与祖先的关系时，*A* 表中每个元组只能找到 *B* 表中的一个祖先元组，所以只有对 *A* 表排序的时间代价 $O(m\log m)$，而没有对 *B* 表中分组的排序代价；*A* 表连接 *B* 表的时间代价为 $O(m)$；所以 *AB* 总的排序连接的时间代价为 $O(m\log m+m)$.

当 *a*、*b* 为兄弟关系时，*A* 表的一个分组通过 *C* 表中的一个元组对应表 *B* 的一个分组，不妨设表 *A* 的一个分组有 (m/k) 个元组，表 *B* 的一个分组有 (n/k) 个元组. 对 *A* 表所有元组进行排序的时间代价为 $O(m\log m)$；对表 *B* 的一个分组排序的时间代价为 $O((n/k)\log(n/k))$；表 *A*、*B* 的一个分组之间连接的时间代价为 $O(mn/k^2)$；而表 *A*、*B* 中分别有 *k* 个分组，则 *AB* 总的排序连接时间代价为 $O(m\log m+mn/k+n\log(n/k))$.

当有多个表进行连接时，只需对第一个表进行全排序；在连接后面的表时，前面的临时表已经有序，所以而后的连接只是对与临时表中的元组有上下文关系的分组进行排序和连接.

4.2.3 JASIT 算法示例

下面举例说明 JASIT 算法.

现假设在例 1 的脚本 *xq* 中，原子表 A_{ab}、C_{ce}、*D* 的查询结果分别有 1、3、3 个分组，如图 4. 图中省略了表 *D* 的元组到表 A_{ab} 元组的反向索引和表 C_{ce} 中与排序无关的变量 *e* 的值.

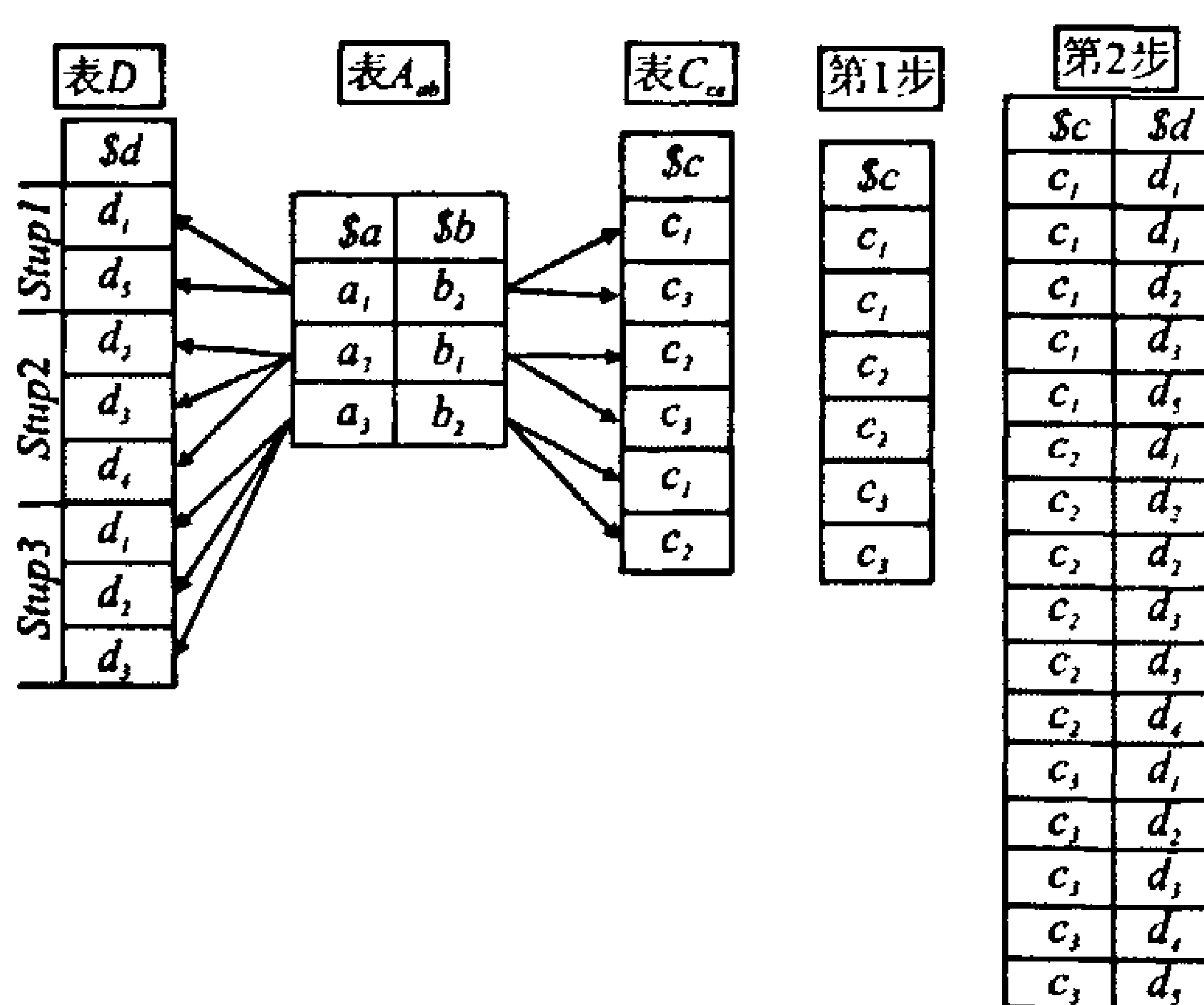

图 4 JASIT 算法示例图

Fig. 4 The example for JASIT

图 4 给出了连接排序算法的示意过程：

(1) 首先对表 C_{ce} 排序，排序结果如图 4 第 1 步；

(2) 在表 C_{ce} 中取等值的两个 c_1，在表 *D* 中找到与其有上下文关系的两个分组 S_{tup1}、S_{tup3}，对分组 S_{tup1} 的元组 d_1、d_5 和分组 S_{tup3} 的元组 d_1、d_2、d_3 一起排序，如图 4 第 2 步的前 5 个

万方数据

元组.

(3) 在表 C_{α} 中再取两个等值的 c_2,在表 D 中找到与其有上下文关系的两个分组 S_{tup2}、S_{tup3},对分组 S_{tup2} 的元组 d_2、d_3、d_4 和分组 S_{tup3} 的元组 d_1、d_2、d_3 一起排序,排序后的结果如图第 2 步中灰色部分所示. 如此对 c_3 的上下文分组排序;排序后连接表 C_{α} 和表 D,结果如图 4 第 2 步.

(4)最后连接表 C_{α}、表 D 和未排序的表 A_{ab},形成最终块表.

5　实验结果与分析

笔者用 Java 实现了 XSIEQ1.02 系统,并与 Qizx[7] 在查询后处理的时间性能上进行了对比测试,Qizx 是实现 XQuery 规范的开源 Java 软件,它支持带多关键字排序的 order 子句等复杂的功能. 实验考核的是影响后处理时间性能的因素:经连接后得到的块表元组的规模、排序关键字个数、return 子句中变量个数以及 where 子句的存在等. 实验平台为 Windows XP 操作系统,CPU 为 P3,主频为 800M,内存为 320M;实验用到的 XML 文件使用 XMark[8] 生成,XML 文档变化范围为 5～7M;XQuery 脚本为非嵌套查询. 图 5 为实验部分结果.

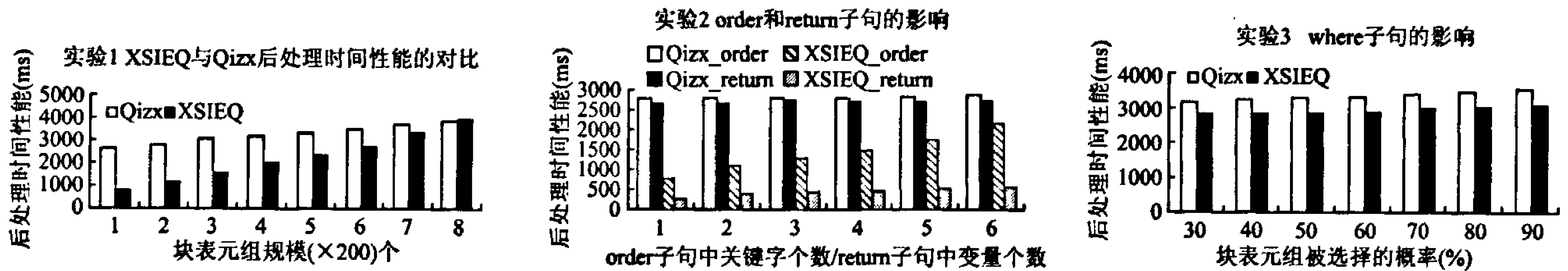

图 5　查询后处理时间性能比较

Fig. 5　The time cost in query post-processing

实验 1 是在没有 where、order 子句且 return 子句返回的变量相同,XQuery 脚本中定义了 8 个 for 变量的情况下,块表元组规模不同时的测试结果. 实验 2 是在块表元组规模为 200 个时,return 子句中变量个数和 order 子句中排序关键字个数不同时的测试结果. 实验 3 是没有 order 子句且 return 子句返回的变量相同,块表元组规模为 1200 个时,where 子句选择概率不同时的测试结果.

从测试结果可以看出:

1. XSIEQ 和 Qizx 的查询后处理时间均随元组规模的增加而线性增长,当元组规模小于 1600 个,XSIEQ 的后处理时间比 Qizx 要快.

2. 在 XQuery 脚本中定义的变量相同时,后处理时间分别随 return 子句中变量个数和 order 子句中排序关键字个数的增加而增加. 这是因为 return 子句中变量个数增加,使得构造的文档规模增大,后处理时间也随之缓增长. order 子句中排序关键字增加,多了一级排序的时间开销,导致后处理时间增加.

3. where 子句的存在对后处理时间性能没有负作用,如果 where 子句过滤的查询结果较多,还会减少结果构造的时间,这是因为 where 子句筛选掉部分元组,缩短了结果构造的时间.

4. 从图中也可以看出,XSIEQ 的后处理时间对元组规模和 order 子句中排序关键字个数的变化敏感,这是因为 XSIEQ 的后处理主要是对原子表元组的排序和连接操作,当原子表数目较多时,原子表元组规模的较小增加都会导致连接次数的急剧增加,排序关键字的增加也会导致频繁地操作原子表.

6　相关工作

流查询引擎 Raindrop[3]、TurboXPath[6] 采用自动机和代数相结合的技术,它们对 XPath 的支持力度较完整. FluXquery[4] 和文献[5]利用的都是查询重写技术. FluXQuery 的关注点在于缓冲内容的及时输出,而不关心查询力度的完整性,所以支持力度有限,不支持嵌套 XPath 式、路径中的通配符"*"和子孙轴"//". 目前,[3-6]都没有实现 order 子句,其中 FluXQuery 不提供对 let 语句的支持. 文献[5]通过查询重写支持嵌套和聚集,包括用户定义的递归函数. Qizx 虽实现了 XQuery 的大部分功能,但它是把 XML 文档读入内存再处理,它以空间的代价来获得稳定的性能,不适合较大文件的查询和连续查询,也不支持模式的导入与验证.

本文用服务器/客户端模式来解决复杂 XQuery 在 XML 数据流上的高效实时查询,系统实现的客户端构造器根据自己的需求,能够快速构造并输出结果文档,并支持 order 子句的多关键字排序特性.

7　结　论

本文介绍了一个以 XQuery 为条件的流查询系统,并着重给出了系统的查询后处理实现. XSIEQ 使用了高效的排序算法,保证了客户端结果构造比较稳定的性能. 同时也要注意到,随着元组规模和排序关键字个数的增加,XSIEQ 的后处理时间比 Qizx 增加得更快. 如何获得更稳定的查询后处理性能,减少元组规模和排序关键字个数变化的影响是下一步工作的重点.

万方数据

References:

[1] Scott Boag, et al. XQuery: a query language for XML[EB/OL]. W3C Recommendation, Jan. 23, 2007,http://www.w3.org/TR/xquery.

[2] Anders Berglund, et al. XML path language (XPath) 2.0. W3C recommendation[EB/OL]. Jan. 23, 2007, http://www.w3.org/TR/xpath20/

[3] Hong Su, Elke A. Rundensteiner and murali mani semantic query optimization for XQuery over XML streams [C]. In: VLDB, Trondheim, August 2005.

[4] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, et al. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams[C]. In: Proc. 30th VLDB 2004,228-239.

[5] Li Xiao-gang, Gagan Agrawal. Efficient evaluation of XQuery over streaming data[C]. In: VLDB Conf., 2005.

[6] Vanja Josifovski, Marcus Fontoura, Attila Barta. Querying XML streams[C]. In: VLDB, April 2005, 14(2):197-210.

[7] Qizx 1.1p2 released, September 3, 2006[EB/OL]. http://www.axyana.com/qizxopen/.

[8] Albrecht Schmidt, Florian Waas, Martin Kersten, et al. XMark: a benchmark for XML data management[C]. In: Proc. VLDB: 974-985, 2002.

[9] Yanlei Diao, Mehmet Altinel, Hao Zhang,et al. Path Sharing and predicate evaluation for high-performance XML filtering [J]. TODS, Dec. 2003. http://yfilter.cs.berkeley.edu/code_release.html, 28(4):67-516.

[10] Zachary G, Ives, et al. An XML query engine for network-bound data[C/OL]. In:VLDB J. http://data.cs.washington.edu/x-scan.html,2002, 11(4): 380-402.

[11] Zhang Yu, Wu Nian. An XML stream query automaton with promoting layered buffer[J]. Journal of Chinese Comuter Systems,2007, 28(3):456-461.

附中文参考文献:

[11]张 昱,吴 年. 一种逐层提升缓冲的XML流查询自动机[J]. 小型微型计算机系统, 2007,28(3):456-461.

万方数据