计算机研究与发展 ISSN 1000-1239/CN 11-1777/TP
Journal of Computer Research and Development 47(5): 841-848, 2010

高效的 XML 关键字查询改写和结果生成技术

黄 静 陆嘉恒 孟小峰
(中国人民大学信息学院 北京 100872)
(huangjingruc@ruc.edu.cn)

Efficient XML Keyword Query Refinement with Meaningful Results Generation

Huang Jing, Lu Jiaheng, and Meng Xiaofeng
(*School of Information, Renmin University of China, Beijing* 100872)

Abstract search method provides users with a friendly way to query XML data, but a user's keyword query may often be an imperfect description of their intention. Even when the information need is well described, a search engine may not be able to return the results matching the query as stated. The task of refining the user's original query is first defined to achieve better result quality as the problem of keyword query refinement in XML keyword search, and guidelines are designed to decide whether query refinement is necessary. Four refinement operations are defined, namely term deletion, merging, split and substitution. Since there may be more than one query refinement candidates, the definition of refinement cost is proposed, which is used as a measure of semantic distance between the original query and refined query, and also proposed is a dynamic programming solution to compute the refinement cost. In order to achieve the goal of finding the best refined queries and generate their associated results within a one-time node list scan, a stack-based algorithm is proposed, followed by a generalized partition-based optimization, which improves the efficiency a lot. Finally, extensive experiments have been done to show efficiency and effectiveness of the query refinement approach.

Key words XML; keyword search; query refinement; query rewriting; query suggestion; SLCA

摘 要 用户使用关键字查询时可能不能准确地表达他们的意图,即使用户正确地表达了查询意图,查询引擎也可能不能准确地返回查询结果.针对这一问题,重点研究了在 XML 关键字查询中如何进行有效的查询改写并生成有意义的结果.提出 4 种查询改写操作和查询改写代价的概念,给出了动态规划的方法计算查询改写代价.为了找出最优的查询改写,给出了基于栈的查询改写和结果生成算法,并提出了基于划分的优化算法.最后通过丰富的实验对提出的方法进行了验证.

关键词 XML;关键字查询;查询改写;查询重写;查询推荐;SLCA

中图法分类号 TP391

0 引 言

关键字查询为用户提供了友好便捷的查询方式,如何使用关键字查询从 XML 数据中获取所需信息已经成为学术界近期研究的一个热点问题[1-5].这些工作主要研究如何过滤无关的查询结果来提高查准率.本文关注的是另一个方面:当查询没有结果

收稿日期:2009-06-06;修回日期:2009-10-10
基金项目:国家自然科学基金项目(60903056,60573091);国家"八六三"高技术研究发展计划基金项目(2009AA01Z133,2009AA011904);高等学校博士学科点专项科研基金项目(20090004120002);教育部科学技术研究重点基金项目(109004)
通信作者:陆嘉恒(jiahenglu@ruc.edu.cn)

返回或是结果返回太少时如何通过改写原始查询,使得新的查询获得好的查全率. 这种情况是普遍存在于关键字查询中的,由于用户可能不能准确表达查询意图,输入的查询可能存在拼写错误或不相关的词,这样使得某些关键字在文档中找不到匹配的结点,导致没有结果返回.

即使用户准确地表达了查询意图,搜索引擎也可能不能准确地返回用户想要的结果,例如,对于图1所示的XML文档,用户输入查询"paper, XML"以查找关于XML的文章,文档中包含inproceedings(0.0.1.0), inproceedings(0.1.1.0)和article(0.1.1.2)三个解,但由于关键字"paper"在文档中无法找到匹配的结点,以至于没有结果返回. 根据文献[6]中的调查统计,用户使用关键字搜索时,大概有10%~15%的查询存在错误,有40%~51%的查询要经过修改才能获得所需的信息. 我们将改写用户输入的原始查询来获得更好查询结果的问题定义为查询改写.

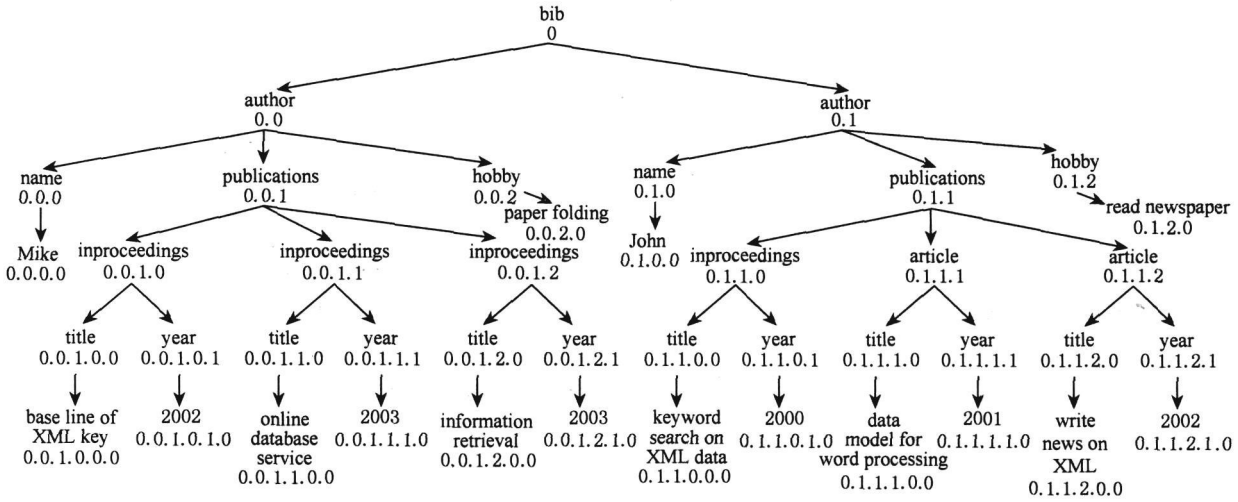

Fig. 1 Example XML document.

图1 带Dewey编码的XML文档

就我们所知,目前的研究工作还没有涉及XML关键字查询改写问题,本文首次提出了在XML关键字查询中,当用户输入的查询没有结果返回或者只有整个文档作为结果返回时,自动地将原始查询改写成语义相同或相近的查询,并返回有意义的结果. 本文采用SLCA[1]作为关键字查询语义计算结果. 例如,对于图1所示的XML文档,提交查询"proceedings, XML"和"XML, John, 2003",前者没有任何结果返回,后者将返回根结点,如果将这两个查询分别改写成"inproceedings, XML"和"XML, John"重新进行查询,前者的查询结果是inproceedings(0.0.1.0)和inproceedings(0.1.1.0);后者的查询结果是author(0.1),明显可以看出改写后的查询获得了更好的查询结果. 因此,查询改写不仅可以提高查询质量,而且可以作为查询推荐,帮助用户更快地找到所需的信息.

本文将查询改写和结果生成作为一个完整的问题进行研究,贡献可以总结为以下几点:

1. 形式化地定义了XML关键字查询改写问题,并提出了是否需要查询改写的判断标准;

2. 提出了查询改写的4种操作,并设计了动态规划的方法来计算改写原始查询的代价;

3. 提出了基于栈的查询改写算法,计算最小代价的查询改写,同时生成相应的查询结果,只需一次扫描关键字倒排索引,并能处理不需要查询改写的情况;

4. 进一步设计了基于划分的优化算法,与现有的语义和查询算法具有良好的正交性;

5. 进行了全面的实验,结果表明了本文查询改写方法的有效性和算法的高效性.

1 查询改写

本节给出XML关键字查询改写问题的定义,并介绍4种查询改写操作和查询改写代价计算方法.

1.1 XML关键字查询改写

定义1. 对XML文档D提交查询Q,如果没有结果返回或者基于SLCA语义返回整个文档作为结果,那么查询Q需要被改写.

例如,对于图1所示的XML文档,表1左栏所

示的原始查询都需要改写. 分析这些查询可以发现主要包括以下 4 种情况: 查询中包含拼写错误的词或文档中不存在的词; 查询中错误地将多个词合并; 查询中错误地将一个词拆分; 查询中包含太多关键字以至限制过于严格或包含与查询意图无关的词.

针对这 4 种情况, 我们提出 4 种相应的查询改写操作: 词替换、词拆分、词合并和词删除. 表 1 给出了一些查询改写实例, 原始查询在图 1 所示的 XML 文档中找不到解或只返回根结点, 经过这 4 种操作之后得到改写后的查询, 这些查询能够在图 1 所示的文档中得到非根结点的 SLCA.

Table 1　Examples of Query Refinement

表 1　查询改写实例

Original Query	Refined Query
Q_1: IR, 2003, Mike	RQ_1: information retrieval, 2003, Mike
Q_2: Mike, publication	RQ_2: Mike, publications
Q_3: database, paper	RQ_3: database, inproceedings
Q_4: XML, John, 2003	RQ_4: XML, John
Q_5: hobby, news, paper	RQ_5: hobby, newspaper
Q_6: on, line, data, base	RQ_6: online, database

对于一个查询可能有多种改写方法, 例如, 对于表 1 中查询 Q_1 来说有两种改写方法: 一种是使用替换操作, 用"information retrieval"替换"IR"; 另一种是使用删除操作, 直接删除"IR", 使得查询变成"2003, Mike". 很明显应该采用第 1 种改写, 因为它的语义更接近原始查询. 对于多种查询改写方案, 如何确定最优的方案, 我们提出查询改写代价 $refineCost(Q, RQ)$, 其中 Q 是原始查询, RQ 是改写后的新查询, 它在一定程度上表示了新查询与原查询在语义上差异.

定义 **2.** 查询改写代价. 假设查询 Q 通过一系列改写规则 $R_1, R_2, \ldots, R_r$ 变成新查询 RQ, 记作: $Q \xrightarrow{R_1, R_2, \ldots, R_r} RQ$, 其中 $R_i\ (1 \leqslant i \leqslant r)$ 称为改写规则, 有如下形式: $k_1, k_2, \ldots, k_n \xrightarrow{op} k'_1, k'_2, \ldots, k'_m$, 其中 op 为 4 种操作之一, $k_i \in Q$ 或 $k_i = \varnothing (1 \leqslant i \leqslant n)$, $k'_j \in RQ$ 或 $k'_j = \varnothing (1 \leqslant j \leqslant m)$. 我们定义查询改写代价 $refineCost(Q, RQ) = \sum_{i=1}^{r} cost(R_i)$, 其中 $cost(R_i)$ 为应用规则 R_i 的代价.

因此, 查询改写代价就是改写查询时应用的所有改写规则的代价之和. 改写规则的生成不是本文的研究重点, 可以使用已有的字典(如 WordNet)生成. 规则的代价可以由系统定义, 也可以由用户定义. 本文假设使用替换和删除操作的规则代价定义为 2, 其他为 1, 对表 1 来说, 我们可以得到 $refineCost(Q_4, RQ_4) = 2$; $refineCost(Q_5, RQ_5) = 1$.

查询改写的目的是获得有意义的解, 根据定义 1, 在本文就是非根结点的 SLCA. 因此好的查询改写不仅是代价最小的, 也要使得新查询有非根结点的解.

定义 **3.** 给定 XML 文档 D 和查询 Q, RQ 称为 Q 的最优改写, 如果不存在 Q 的查询改写 RQ', $refineCost(Q, RQ') < refineCost(Q, RQ)$, RQ' 在 D 上有非根结点的 SLCA; 且 RQ 在 D 上有非根结点的 SLCA.

因此, XML 关键字查询改写问题就是计算最优查询改写和相应的查询结果.

1.2　查询改写代价的动态规划算法

应用不同的改写规则, 一个查询可能有多种改写结果, 因此需要一个高效的方法计算查询改写代价.

这个问题可以定义为: 给定两个关键字序列 $S = \{k_1, k_2, \ldots, k_s\}$, $T = \{k'_1, k'_2, \ldots, k'_t\}$ 和改写规则集合 R, 返回将 S 改写成 T 的代价, 即 $refineCost(S, T)$. 为了解决这个问题首先要解决子问题: 对于任意 $1 \leqslant i \leqslant s$, 计算将 S 的子序列 $S[1, i] = \{k_1, k_2, \ldots, k_i\}$ 改写成 T 的代价. 我们提出了一个动态规划方法, 如下面方程所示:

$$C[i] = \min \begin{cases} C[i-1], & \text{if } k_i \in T; \\ C[i-1] + \text{cost of deletion } k_i, & \text{if } k_i \notin T; \\ C[i-|l(r_i)|] + \text{cost}, & \text{if } k_i \notin T \text{ and of applying rule } r_i \\ & \text{for } \forall r_i \in R(k_i). \end{cases}$$

在这个方程中, $C[i] = refineCost(S[1, i], T)$, 我们的目标是计算 $C[s]$, 它等于 $refineCost(S, T)$. 对于每个关键字 $k_i \in S$, 对应一个规则集合 $R(k_i) = \{r \mid r = <* k_i \rightarrow k'_m, \ldots, k'_n>$ 其中, $* k_i$ 是以关键字 k_i 结尾的 S 的子序列, $k'_m, \ldots, k'_n \in T\}$. 对于每个规则 r, 用 $|l(r)|$ 表示规则左边包含关键字的个数.

我们初始化 $C[0] = 0$, 也就是改写一个空查询的代价为 0. 在迭代计算 $C[i]\ (1 \leqslant i \leqslant s)$ 时, 考虑以下 3 种情况, 并将得到的最小值赋给 $C[i]$:

1. 当关键字 k_i 同时出现在 S 和 T 中时不需要任何改写代价;

2. 当关键字 k_i 不出现在 T 中时增加删除代价;

3. 当关键字 k_i 不出现在 T 中但是有规则 r 可以使用, 时追溯至 $C[i-|l(r)|]$, 重新计算 $C[i]$ 的值. 如果有多个规则可以使用, 取使得 $C[i]$ 值最小的那个.

2 查询改写和结果生成算法

给定查询 Q, 如果根据定义 1 需要进行改写, 查询引擎最好能够自动地进行查询改写并计算新查询的结果. 我们的目标是只扫描关键字倒排索引一次, 在执行原始查询的过程中有效地计算最优的查询改写和结果. 如果不需要改写则返回原始查询的结果.

2.1 基于栈的算法

我们修改了文献[1]中基于栈的算法来解决本文中的问题. 主要数据结构栈的设计如下: 栈中每个元素由(*id*, *keywords*)对组成. 假设一个从栈底到栈元素 *en* 的 *id* 分别为 id_1, id_2, …, id_m, 那么元素 *en* 代表了 Dewey 编码为 id_1, id_2, …, id_m 的结点. *keywords* 是布尔值数组. 栈中元素的 *keywords*[*i*]= T 表示以该元素表示的结点包含关键字 k_i, 否则为 F. 根据改写规则, 将互相替换的关键字共享同一列; 将拆分操作后涉及的多个关键字共享一列, 这列中只有当所有的关键字都被包含时 *keywords* 才赋值 T.

由于在扫描完关键字索引之前不确定是否需要改写和最优改写, 为了只处理结点一次, 我们需要保存当前最优解, 成为查询改写候选, 当发现更优候选时进行替换, 最后得到整体最优解. 具体算法如算法 1 所示, 算法中的 *result* 在运行过程中保留了当前最优解.

算法 1. 基于栈的查询改写和结果生成算法.

输入: 查询 $Q=\{k_1, k_2, \ldots, k_n\}$ 和 XML 文档 D;

输出: 结果集 RQ.

```
①设置RQ 和 堆栈stack 为空;
②RQ. cost= deletion. cost* size(Q);
③getTerms()= {k1, k2, …, km};
④getNodeLists= {S1, S2, …, Sm};
⑤while(Si ≠ ∅) or (stack ≠ ∅) do
⑥   vs= getSmallestNode();
⑦   p= lca(stack, vs);
⑧   while(stack. size> p. length) do
⑨      entry= stack. pop();
⑩      if(entry! = root) then
⑪         for( i= 1 to m) do
⑫            if(entry. keyword[i]= = true)
              then
⑬               k[i]= tempR;
⑭               stack. top. keywords[i]= true;
⑮            endif
⑯         endfor
⑰      tempR. cost= refineCost(Q. tempR);
⑱      if(tempR. cost ≤ result. cost) then
⑲         add{tempR, entry} into result;
⑳         RQ. cost= tempR. cost;
㉑      endif
㉒   endif
㉓ end while
㉔ for(j= p, length to vs. length) do
㉕    stack. push (vs[p], vs[length]);
㉖ endfor
㉗ stack. top. kewyords[k]= true;
㉘ end while
㉙return RQ.
```

算法 1 与文献[1]中的基于栈的算法有如下明显的区别: 首先, 我们需要根据当前查询和可以使用的规则, 获取涉及的所有关键字, 对于没有倒排索引的关键字进行删除(行 ③~ ④); 其次, 根据改写条件, 我们摒弃了以根结点为解的情况(行 ⑩); 第三, 对于每次出栈的非根结点元素 *entry* 都检验其包含的关键字, 形成改写候选 *tempR*, 并计算此时的改写代价(行 ⑪~ ⑰); 第四, 如果改写候选的代价是当前最小的, 就将 *tempR* 和 *entry* 放入 *result* 中, 并修改最小代价阈值(行 ⑱~ ㉑), 这里在行 ⑲需要检验 *entry* 是否是 *tempR* 的 SLCA, 只有 SLCA 才放入 *result* 中. 因为当算法找到一个当前解时, 由于不能确定这个解是否是整体最优解, 并没有将栈中剩下的元素的 *keywords* 置为 F. 最后, *result* 中的结果就是最优查询改写和对应的查询结果. 如果原始查询不需要改写, 那么 *result* 中就是原始查询的结果.

由于在扫描完关键字索引之前不能确定最优的查询改写, 因此, 在最坏的情况下, 这个算法需要将所有的查询改写候选计算一遍. 因此, 算法复杂度是 $O(d\sum_{i=1}^{m}|S_i|)$, 其中, m 是使用改写规则后涉及的关键字个数, S_i 是直接包含关键字 k_i 的结点集合, d 是 XML 树的深度.

为了减少无用的候选计算, 我们提出了基于划分的优化方法. 该方法能尽早确定最优查询改写, 能避免不必要的 SLCA 计算.

2.2　基于划分的优化算法

根据定义 1, XML 文档根结点可以排除在 SLCA 的计算之外, 除了根结点之外的结点都能看作是有意义的 SLCA, 于是, 基于划分的优化方法不再将文档整体考虑, 而是先对文档进行分割, 在文档片断上推断查询改写, 然后对最优的改写进行计算.

定义 **4.** 文档划分. 将 XML 文档 D 除去根结点 r 得到的子树片断 $D_1, D_2, \ldots, D_m$ 称为 D 的一个划分, 记作 $D(r, D_1, D_2, \ldots, D_m)$, 它具有以下特点:

1) $r \cup D_1 \cup D_2 \cup \ldots \cup D_m = D$;

2) $\forall i, j \in [1, m], i \neq j, D_i \cap D_j = \varnothing$.

根据定义 4, 图 1 中的 XML 文档划分为两部分: 一个是以 author(0.0) 为根结点的子树, 另一个是以 author(0.1) 为根结点的子树.

如果要查询 Q 在文档 D 上有非根结点的 SLCA, 那么 Q 必然在 D 的某个子树片断 D_i 上有解. 根据这个性质和定义 3 可以得出: 对于文档 $D(r, D_1, D_2, \ldots, D_m)$ 和查询 Q, 如果 RQ 是最优查询改写, 那么 $\exists i \in [1, m]$, 对于 $\forall k \in RQ, k \in D_i$.

因此, 寻找最优查询改写可以先将文档进行划分, 然后找出包含最小代价查询改写的子树片断, 并计算改写后的查询在此子树片断上的查询结果即可. 将文档进行划分以后便可轻松获得查询改写候选: 由于同属于一个片断的结点 Dewey 编码前两位相同, 因此, 我们以这两位编码作为一个片断的标识, 记作 pid, 在倒排索引中用 pid 可以找出同属一个片断的结点, 也可以得出这个片断包含的关键字, 即为一个改写候选. 获得所有查询改写候选后, 只要选取代价最小的候选在相应的文档片断上计算 SLCA 即是最优查询改写和结果, 而不用计算其他候选的 SLCA, 这就是基于划分的优化方法. 具体算法如下:

算法 **2.** 基于划分的优化算法.

输入: 查询 $Q = \{k_1, k_2, \ldots, k_n\}$ 和 XML 文档 D;

输出: 结果集 $result$.

① 设置 RQ 为空;

② $RQ.cost = deletion.cost * size(Q)$;

③ $getTerms() = \{k_1, k_2, \ldots, k_m\}$;

④ $getNodeLists = \{S_1, S_2, \ldots, S_m\}$

⑤ while($S_i \neq \varnothing$) do

⑥　$vs = getSmallestNode()$;

⑦　$pis = getDocPartitionID(vs)$;

⑧　$getKLPartition(pid) = \{S'_1, S'_2, \ldots, S'_m\}$;

⑨　for($i = 1$ to m) do

⑩　　if($S_i \neq \varnothing$) then

⑪　　$RQ = K_i$;

⑫　　endif

⑬　endfor

⑭　$RQ.cost = refineCost(RQ, Q)$;

⑮　if($RQ.cost \leqslant result.cost$) then

⑯　　$slcas = computeSLCAs(\{S'_1, S'_2, \ldots, S'_m\}, RQ)$;

⑰　　$result = \{RQ, slcas\}$;

⑱　　$result.cost = RQ.cost$;

⑲　endif

⑳　remove($S'_1, S'_2, \ldots, S'_n$) from $\{S_1, S_2, \ldots, S_m\}$;

㉑　设置 RQ 为空;

㉒ end while

㉓ return $result$.

首先根据原始查询和改写规则获得相关的关键字, 并根据倒排索引获得包含这些关键字的结点列表(行③~④). 然后从结点列表中找到编码最小的结点, 取这个结点 Dewey 编码的前两位 pid 就可以作为文档片断的标识, 从结点列表中取出编码前两位为 pid 的所有结点子列表, 也即为同一文档片断的结点(行⑥~⑧). 如果关键字对应的子列表不为空, 则说明此片断包含此关键字, 由此可以确定改写候选, 然后通过 1.2 节介绍计算改写代价的方法计算候选改写的代价(行⑨~⑭). 对于比当前代价阈值小的改写候选用已有的方法计算 SLCA, 修改代价阈值(行⑮~⑲). 将结点子列表移除(行⑳). 这个过程一直循环到结点列表中没有任何结点为止, 返回结果.

该优化的算法在推断查询改写候选时方法简单快捷, 只需取结点编码前两位并判断结点集合是否为空即可. 可以跳跃某些结点不进行 SLCA 计算, 以达到优化的目的. 在计算 SLCA 时可以直接运用现有工作中的算法[1,7]. 基于划分的思想具有良好的扩展性, 可以运用到基于其他语义的查询改写算法中.

3　实验与分析

我们通过丰富实验验证了本文查询改写方法的

有效性和算法的高效性.

所有实验都是在处理器为 Intel 双核 2.0 GHz、内存为 1 GB、操作系统为 Windows XP Professional 机器上运行, 实现语言为 Java. 我们在 SIGMOD Record[8] 和 DBLP[9] 两个数据集上进行了实验, 大小分别为 500 KB 和 130 MB.

3.1 改写质量

我们使用一些需要改写的查询来验证本文查询改写方法的有效性, 改写规则是根据本文提出的 4 种改写操作人工生成的. 原始查询没有任何结果返回, 因此本文对比改写后的查询和相应的结果个数来评价改写方法的质量. 表 2 和表 3 列出了一些改写示例. 原始查询没有结果返回或返回整个文档作为结果, 改写的查询结果个数如表 2 和表 3 所示:

Table 2 Sample Query Refinement on SIGMOD Record
表 2 SIGMOD Record 上查询改写示例

id	Original Query	Refined Query	Result Size
Q_1	data, base, codd	database, codd	2
Q_2	data, mining, jia, wei, han	data, mining, han, jiawei	3
Q_3	paper, net, work	article, network	6
Q_4	real, time, systems	real-time, systems	9
Q_5	keyword, search, database, rank	search, database	11
Q_6	ad, hoc, article	article, ad, hoc	1

Table 3 Sample Query Refinement on DBLP
表 3 DBLP 上查询改写示例

id	Original Query	Refined Query	Result Size
Q_7	xml, model, 1995, view	model, 1995, view	3
		xml, model, view	1
Q_8	data, mining, 2006, jia, wei, han	data, mining, han, jiawei	14
Q_9	paper, net, work, 1990	article, network, 1990	59
		inproceedings, network, 1990	88
Q_{10}	fulltext, search	search, full, text	4
Q_{11}	keyword, search, rank, xml	keyword, search, xml	2
Q_{12}	refinement, keyword, search, IR	keyword, search	9
		keyword, refinement	5

从表 2 和表 3 的示例可以看出, 本文的改写方法可以对原始查询进行有效的改写, 改写后的查询在语义上与原始查询具有最大的相似性, 并返回有意义的结果. 本文的算法可以支持 4 种操作的任意组合, 可以根据需要来选择. 有些查询具有多种最优改写, 例如 Q_7, Q_9 和 Q_{12}, 算法将返回所有这些改写和它们对应的结果.

3.2 执行时间

我们通过对查询改写的执行时间来比较本文两种算法的效率. 图 2 和图 3 分别是表 2 和表 3 所示的查询改写的执行时间. 其中, "Stack-Refine" 代表本文基于栈的改写算法, "Partition-Refine" 代表本文基于划分的优化算法.

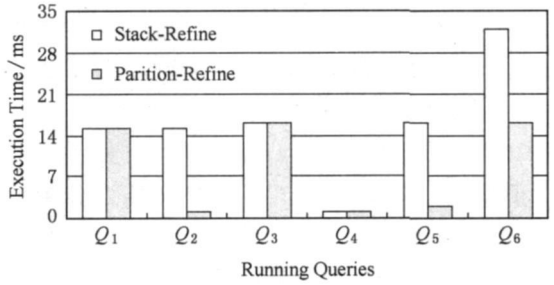

Fig. 2 Processing time on SIGMOD record.
图 2 SIGMOD Record 上查询改写示例执行时间

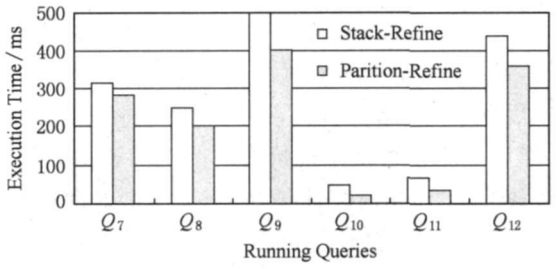

Fig. 3 Processing time on DBLP.
图 3 DBLP 上查询改写示例执行时间

从图 2 和图 3 可以看出, 基于划分的优化算法比基于栈的算法效率更高, 因为它可以避免不必要的 SLCA 计算.

为了验证本文方法处理正常查询的效率, 我们选择了一些不需要改写的查询与文献[1] 提出的基于栈的计算 SLCA 的算法进行了效率对比. 图 4 是

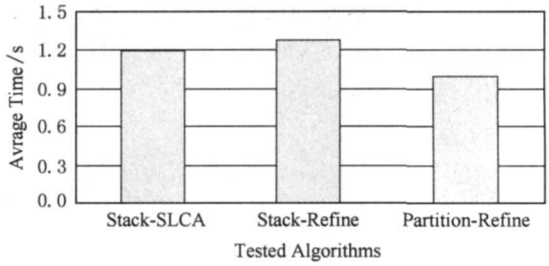

Fig. 4 Average processing time on DBLP.
图 4 DBLP 上平均执行时间

在 DBLP 上的平均执行时间对比. 其中"Stack-SLCA"代表文献[1]中基于栈的算法.

从图 4 可以看出, 本文的两种算法都能处理不需要改写的查询. 基于栈的查询改写算法要比基于栈的计算 SLCA 算法耗费更多一些时间, 因为它考虑了可能出现需要改写的情况, 会生成改写候选, 并计算改写代价. 从整体来看, 改写算法提高了查询的质量, 平均多耗费 10% 的时间在可以接受的范围内. 而基于划分的算法则比其他两种算法效率都高, 因为它可以避免在某些无解的划分片断上的计算.

3.3　可扩展性

我们分别在 10 MB, 20 MB, 70 MB 和 130 MB 的 DBLP 文档上执行一些查询, 并计算平均执行时间来验证本文两种算法的可扩展性, 结果如图 5 所示. 可以看出, 随着文档大小的递增, 平均执行时间基本呈线性增长, 因此我们的算法具有良好的可扩展性.

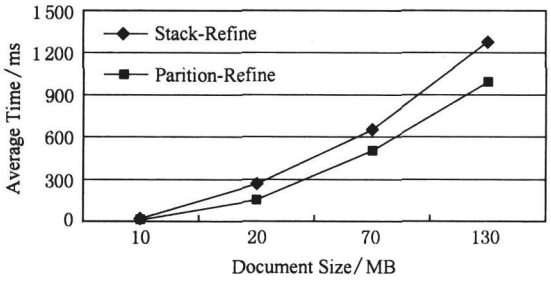

Fig. 5　Processing time with increasing document size.

图 5　平均执行时间随数据集大小变化的关系

4　结　　论

本文研究了 XML 关键字查询改写问题, 查询改写不仅能够提高查询质量, 并且能够帮助用户更快地确定查询目标. 本文针对用户输入的没有查询结果的查询, 通过词删除、替换、拆分和合并 4 种操作进行改写. 对于多个查询改写通过改写代价来衡量它们的优劣, 并设计了动态规划方法计算改写代价. 为了有效计算最优查询改写并生成相应的结果, 我们提出了基于栈的算法和基于划分的优化算法. 这两种算法都只需扫描结点列表一次, 并能处理不需要改写的情况. 特别是基于划分的思想能应用于其他查询语义, 具有很好的扩展性. 实验结果进一步验证了本文方法的有效性. 未来工作可以考虑在原始查询增加关键字的操作来改写查询.

参　考　文　献

[1] Xu Y, Papakonstantinou Y. Efficient keyword search for smallest LCAs in XML database [C] //Proc of ACM SIGMOD 2005. New York: ACM, 2005: 527-538

[2] Cohen S, Mamou J, Kanza Y, et al. XSEarch: A semantic search engine for XML [C] //Proc of VLDB 2003. San Francisco: Morgan Kaufmann, 2003: 45-56

[3] Li G, Feng J. Wang J, et al. Effective keyword search for valuable LCAs over XML documents [C] //Proc of ACM CIKM 2007. New York: ACM, 2007: 31-40

[4] Cohen S, Kanza Y, Kimelfeld B, et al. Interconnection semantics for keyword search in XML [C] //Proc of ACM CIKM 2005. New York: ACM, 2005: 389-396

[5] Huang Jing, Xu Junjin, Zhou Junfeng, et al. MLCEA: An entity based semantics for XML keyword search [J]. Journal of Computer Research and Development, 2008, 45 (Supplement): 372-377 (in Chinese)
(黄静, 徐俊劲, 周军锋, 等. MLCEA: 一种基于实体的 XML 关键字查询语义[J]. 计算机研究与发展, 2008, 45(增刊): 372-377)

[6] Spink A, Jansen B J, Wolfram D, et al. From E-sex to E-commerce: Web search changes [J]. IEEE Computer, 2002, 35(3): 107-109

[7] Sun Chong, Chan Chee-Yong, Goenka Amit K. Multiway SLCA-based keyword search in XML data [C] //Proc of WWW 2007. New York: ACM, 2007: 1043-1052

[8] UW XMLRepository [OL]. [2009-04-20]. http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[9] DBLP XML Repository [OL]. [2009-04-20]. http://dblp.uni-trier.de/xml/

Huang Jing, born in 1984. Master. Her main research interests include XML database and XML keyword search.
黄　静, 1984 年生, 硕士, 主要研究方向为 XML 数据库、XML 关键字查询.

Lu Jiaheng, born in 1975. Associate professor. His main research interests include XML data management, approximate string matching, and cloud computing technology.
陆嘉恒, 1975 年生, 副教授, 主要研究方向为 XML 数据管理、字符串模糊匹配和云计算技术等.

Meng Xiaofeng, born in 1964. Professor and PhD supervisor. He is the secretary general of the Database Society of China Computer Federation (CCF DBS). His main research interests include Web data integration, XML database, mobile data management, and flash database.

孟小峰, 1964 年生, 教授, 博士生导师, 中国计算机学会理事, 中国计算机学会数据库专委会秘书长, 主要研究方向为 Web 数据集成、XML 数据库、移动数据管理、闪存数据库等.

Research Background

This research was partially supported by the grants from National Science Foundation China (NSFC) under the number 60903056, 60573091; National 863 High-Tech Research and Development Plan of China (No: 2009AA01Z133, 2009AA011904), SRFDP Fund for the Doctoral Program (No: 20090004120002) and Key Project in Ministry of Education (No: 109004).

As XML is gradually becoming the standard in exchanging and representing data, effective and efficient methods to query XML data has become an increasingly important problem. XQuery can convey complex semantics meanings and therefore retrieve precisely the desired results. However, in order to write the right query, the user should know the underlying data schema and master the complex query syntax. Keyword search enables users to easily access XML data without the need to learn a structured query language and to study complex data schemas. So keyword search over XML data has attracted a lot of research efforts. However, no existing work has touched the field of automatic query refinement for XML keyword search yet. In this paper, we first define the problem of keyword query refinement in XML keyword search, in which four refinement operations are defined, namely term deletion, merging, split and substitution. In order to achieve the goal of finding the best refined queries and generating their associated results within a one-time node lists scan, we propose a stack-based algorithm, followed by a partition-based optimization.