World Wide Web (2008) 11:153–168
DOI 10.1007/s11280-007-0029-6

Clustered Chain Path Index for XML Document: Efficiently Processing Branch Queries

Hongqiang Wang · Jianzhong Li · Hongzhi Wang

Received: 20 December 2006 / Revised: 21 December 2006 /
Accepted: 11 April 2007 / Published online: 13 September 2007

Abstract Branch query processing is a core operation of XML query processing. In recent years, a number of stack based twig join algorithms have been proposed to process twig queries based on tag stream index. However, in tag stream index, each element is labeled separately without considering the similarity among elements. Besides, algorithms based on tag stream index perform inefficiently on large document. This paper proposes a novel index, named Clustered Chain Path Index, based on a novel labeling scheme. This index provides efficient support for processing branch queries. It also has the same cardinality as 1-index against tree structured XML document. Based on CCPI, efficient algorithms, KMP-Match-Path and Related-Path-Segment-Join, are proposed to process queries efficiently. Analysis and experimental results show that proposed query processing algorithms based on CCPI outperform other algorithms and have good scalability.

Keywords XML · index · clustered chain path · CCPI · TwigStack · 1-index

1 Introduction

XML data is often modeled as labeled and ordered tree. Queries on XML data are commonly expressed in the form of tree patterns, which represent a very useful subset of XPath [19] and XQuery [20].

This paper is partially supported by Natural Science Foundation of Heilongjiang Province, Grant No. zjg03-05 and National Natural Science Foundation of China, Grant No. 60473075 and Key Program of the National Natural Science Foundation of China, Grant No. 60533110.

H. Wang (✉) · J. Li · H. Wang
School of Computer Science and Technology, Harbin Institute of Technology, Harbin, 150001, China
e-mail: hqwang@hit.edu.cn

J. Li
e-mail: Lijz@mail.banner.com.cn

H. Wang
e-mail: wangzh@hit.edu.cn

Query processing on XML document is usually based on some kind of index. 1-index [13] uses *Bi-Simulation* relationship to classify nodes into many sets. 1-index is brilliant for its very small size against tree structured XML document. It's efficient to process source path (path from root to element containing only Parent–Child relationship) queries based on 1-index. However, 1-index discourages us when evaluating branch queries because (1) the exact PC relationships between elements are lost; (2) the nodes in the same set are indistinguishable.

Bruno et al. [2] proposed the holistic twig matching algorithms TwigStack which is I/O optimal for queries with only ancestor-descendant edge. TwigStack uses tag streams which group all elements with the same tag together and assign each element a region-encoding. The tag streams can be regarded as a trivial index of XML document. Recently, many works [6, 10, 11] are proposed to improve TwigStack by reducing intermediate results. However, Enumerative indexing in tag stream index loses similarity of nodes and brings large cardinality of index. Queries are processed inefficient against large XML document.

Based on the observation of the structure similarity of nodes with same source path, we propose a novel labeling scheme, named Clustered Chain Path labeling scheme (CCP for brief) which can group all elements with the same source path into one labeling in index. With CCP labeling scheme, the cardinality of index is extremely reduced to a very small "skeleton". Actually, CCPI has the same cardinality as 1-index against tree structured XML document. Being different from 1-index, each CCP in CCPI contains all its ancestors. Besides, each element in a CCP distinguishes itself from others. These properties support CCPI for processing branch queries efficiently.

The main contributions of this paper include:

- We propose a novel labeling scheme, CCP labeling which can cluster all elements with the same source path into one CCP.
- We propose a novel index structure CCPI based on CCP labeling by group all CCPs with same leaf tags together. The cardinality of the index equals to the cardinality of 1-index against tree structured document.
- Based on CCPI, we develop efficient algorithms, KMP-Match-Path to process queries without branch and Related-Path-Segment-Join to process queries with branch.
- We perform a comprehensive experiment to demonstrate the benefits of our algorithms over previous approaches.

Organization The rest of this paper proceeds as follows: Firstly we discuss preliminaries in Section 2. Then the CCPI index structure is presented in Section 3. We present query processing algorithms in Section 4. Section 5 is dedicated to our experimental results and we end this paper by related works and a conclusion in Section 6 and Section 7.

2 Preliminaries

In this Section, we first introduce XML data model and pre-ordered path labeling in Section 2.1. Then we define query pattern matching problem based on pre-ordered path labeling in Section 2.2.

2.1 Data model and pre-ordered path

In this paper, we model XML document as a rooted, ordered and labeled tree. We only focus on elements since it is easy to generalize our methods to other types of nodes. A commonly used subset of XPath queries consisting of child axis navigation (/), descendant axis navigation (//), wildcard (*), and branches ([..]) is concerned.

We first give a brief introduction of *pre-ordered element* (Figure 1a) which is an element being labeled with its tag *T* and a number *p* where *p* is the pre-order of the element by left-to-right pre-order traversal of document tree. A *pre-ordered path POP* of a pre-ordered element *E* is denoted as (1) if *E* is root, *POP(E)=null*; (2) if else, let pre-ordered element *P* is the parent of *E*, then *POP(E)=POP(P)/E*. For example, the pre-ordered path of element B [3] (Figure 1a) is "/A[1]/B[3]".

2.2 Query pattern matching

A query *Q* is usually considered as a pattern tree containing branches and related paths. A branch is a root-to-leaf path in the pattern tree. If two or more branches share a crotch, the path from root to the crotch is named related path of these branches. Among all the branches, there is one branch which contains the element to be returned. This branch is called *trunk*.

A match from a pre-ordered path *POP* to a branch *B* is defined as (1) for each element E_i in *B*, there is a pre-ordered element POE_i in *POP* which matches E_i; (2) for any two elements E_i and E_j in *B*, if they satisfy axis μ ε {/, //},the pre-ordered element POE_i and POE_j which match E_i and E_j separately satisfy μ.

If a pre-ordered path *POP* matches a branch *B* with related path *RP*, the part in *POP* which matches *RP* is denoted as *related path segment* (RPS for brief) of *POP*. The source path of a RPS is denoted as *SRPS*.

Given an XPath query *Q* and an XML document *D*, a match of *Q* in *D* is identified by a mapping from a pre-ordered paths set of *D* to query *Q*, such that (1) for each root to leaf branch *B* in *Q*, there is a pre-ordered path in the set matches *B*; (2) for each related path *RP* in *Q*, the pre-ordered paths which matched each branch separately shares the related path segment that matches *RP*.

Example 2.1 Query *Q*="//B[./C]/D" has two branches (Figure 2), which are "//B/D" and "//B/C". The two branches have related path "//B". Pre-ordered path set *OPS*={P_1=/A[1] /B[3]/C[3], P_2=/A[1]/B[3]/D[4]} is a match from sample xml to query *Q* because pre-ordered path P_1 and P_2 match branch "//B/C" and "//B/D" separately, besides P_1 and P_2 share the

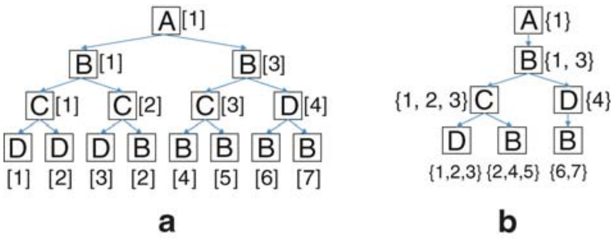

Figure 1 Sample XML tree and 1-index of sample XML. **a** sample XML, **b** 1-index of sample XML.

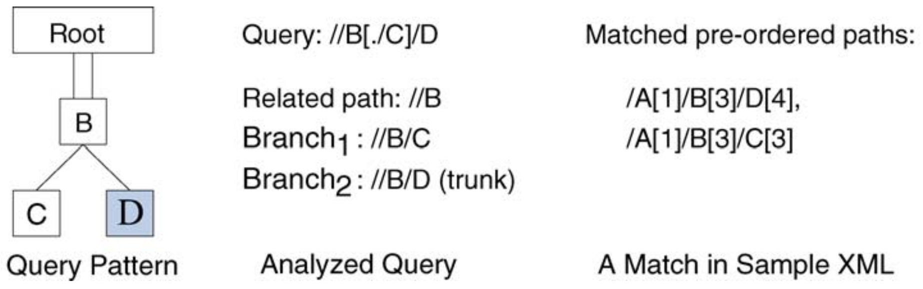

Figure 2 Example of query pattern matching.

same RPS “/A[1]/B[3]”, which matches the related path “//B” in *Q*. Since “//B/D” is trunk, we keep “/A[1]/B[3]/D[4]” only as a match to *Q*.

Finding all matches of a query pattern in an XML document is a core operation in XML query processing. In this paper, we solve the problem in three steps: (1) we cluster pre-ordered paths into CCP sets; (2) for each branch B_i in query pattern *Q*, we find all CCPs which matches B_i; (3) if B_i and B_j have related path *RP*, we join CCP_i and CCP_j matching B_i and B_j separately. Then we return results to *Q*.

3 Clustered chain path index (CCPI)

In this Section, we present the clustered chain path index. CCP labeling scheme and its properties are introduced firstly in Section 3.1, and then the procedure of constructing CCPI and the analyzing of the cardinality of CCPI in Section 3.2. Finally, we introduce the physical storage of CCPI in Section 3.3.

3.1 Clustered chain path labeling scheme

Considering lots of elements sharing a same source path can match/deny a query altogether, we try to cluster them into a clustered pre-ordered path. To avoid losing Parent-Child relationship, we use registry table on each pre-ordered element to record its children’s pre-orders.

Definition 3.1 Clustered Chain Path is an array of Registered Elements. A Registered Element in a CCP is a tuple <*T, O, RT, f*> where

- *T* is the tag of Registered Element.
- is the pre-order set of Registered Element, corresponding to some pre-ordered elements with tag *T.*
- *RT* is the registry tables set of Registered Element. A registry table corresponding to pre-order *o* in *O* contains children’s pre-orders of *T[o].*
- $f{:}O \rightarrow RT$ is a function mapping from pre-order of an element to its registry table.

For example, *C*[1], *C*[2] and *C*[3] share the same source path “/A/B/C”. The CCP of {*C*[1], *C*[2], *C*[3]} is an array *RE* where *RE*[1]=<A, {1}, {(1, 3)}, f_1(1)=(1, 3) > *RE*[2]= <B, {1, 3}, {(1, 2), (3)}, f_2(1)=(1, 2), f_2(3)=(3) > *RE*[3]= <C, {1, 2, 3}, {(ε), (ε), (ε)}, f_3(any)=(ε) > (ε means null). A CCP can be simply expressed as a string “/A[1{1, 3}]/B[1{1, 2}, 3

{3}]/C[1{ε}, 2{ε}, 3{ε}]" which is considered to be the union of pre-ordered paths "/A[1]/B[1]/C[1]", "/A[1]/B[1]/C[2]" and "/A[1]/B[3]/C[3]" .

CCP labeling has special properties (some properties are proven in Section 3.2):

1. For each source path, there is one and only one corresponding CCP in CCPI.
2. Each pre-order set O in registered element is sorted by ascendant order.
3. Each registry table in registered element is sorted by ascendant order.
4. Registry table set RT is sorted by ascendant order. (We say registry table $RT_1 < RT_2$, if for any pre-order p_1 in RT_1 and for any pre-order p_2 in RT_2, we have $p_1 < p_2$)
5. Map function f is indeed a map from index of O to index of RT.

Proof 1. If there are more than one CCP corresponding to a source path, then they can be clustered using algorithm Cluster-Path which is present in Section 3.2.

2. and 3. will be proved in Section 3.2.

4. For two registry tables RT_1 and RT_2 in a registry table set RT, suppose RT_1 is mapped from pre-order O_1 in O and O_2 for RT_2. Let pre-ordered element $T[O_1]$ is the parent of RT_1 and $T[O_2]$ for RT_2. If $O_1 < O_2$, $T[O_1]$ is traveled before $T[O_2]$, since $T[O_1]$ and $T[O_2]$ are at the same document depth (all clustered pre-ordered paths share the same source path), all children of $T[O_1]$ are traversed before $T[O_2]$. Hence pre-ordered elements in RT_1 are traversed before pre-ordered elements in RT_2.

5. Since O and RT are ordered, if this property is violated, then property 4 is violated. □

3.2 Constructing CCPI

Each element gets its pre-ordered path when it's parsed. Firstly the pre-ordered path is transformed to a CCP. Then if two CCPs share a same source path, they are clustered with algorithm Cluster-Path. For large XML file, we use buffer to hold CCPs in memory and write them to disk when buffer is full. After that, we organize the CCPI to confirm there is exactly one CCP corresponding to a source path. The algorithm Cluster-Path is present in algorithm 1.

```
Algorithm 1 Cluster-Path
      Input: $CCP_1$, $CCP_2$ sharing the same source path
      Output: $CCP = CCP_1 \cup CCP_2$
1     for each RES[i] in CCP
2       RES[i]= Sort-Merge-Union ($RES_1$[i], $RES_2$[i])
3     return CCP
```

In algorithm 1, CCP_1 and CCP_2 are input CCPs which share the same source path. CCP is the output containing all pre-ordered paths in CCP_1 and CCP_2. RES_1, RES_2 and RES are registered elements arrays of CCP_1, CCP_2 and CCP.

When clustering, we Sort-Merge-Union pre-order sets as well as registry tables if they are associated with the same tag in two CCPs (Example in Figure 3b). After clustering, the pre-order sets are sorted as well as registry tables. Then property 2 and 3 of CCP labeling are proved here. These properties contribute to efficiently processing queries with branches.

Theorem 3.1 *The cardinality of CCPI equals to the cardinality of 1-index against tree structured XML document.*

First, we introduce *bi-simulation* defined in 1-index [13]. Let *G* be a data graph in which the symmetric, binary relation ≈, the *bi-simulation*, is defined as: we say that two data nodes *u* and *v* are *bi-similar* ($u \approx v$), if

1. *u* and *v* have the same tag;
2. if u' is a parent of *u*, then there is a parent v' of *v* such that $u' \approx v'$, and vice versa;

Proof In tree structured XML document, if two nodes $u \approx v$, then they have the same tag. Let u' is the parent of *u* and v' is the parent of *v* (In tree structured XML document, a node has unique parent), then $u' \approx v'$ which means u' and v' have the same tag. Hence the parent of u' and the parent of v' have the same tag ..., obviously, *u* and *v* have the same source path, so *u* and *v* can be clustered. □

Analysis of algorithm cluster-path The number of elements clustered in a CCP equals to the size of leaf registered element in the CCP. Let CCP_1 and CCP_2 are input CCPs of algorithm 1. Let the number of pre-ordered paths clustered in CCP_1 is N_1 and N_2 for CCP_2. For each registered element in CCP_1, it contains at most N_1 pre-ordered elements (the worst case), the sum of total pre-orders in its registry table is less than or equal to N_1 (N_1 elements have N_1 parents at most). The time complexity of Sort-Merge-Union of two registered elements is $O(N_1+N_2)$. Let the length of CCP_1 is *L*, The time complexity of Cluster-Path in the worst case is $O(\mathrm{L}\times(\mathrm{N}_1+N_2))$.

3.3 Physical storage of CCPI

We separate CCPI data into two parts: info part and data part. A CCP is represented as sequenced tuples {<Pre-Order, Registry Table>} as data part and <Source Path, Start Pointer, Length> as info part where Start Pointer is the start position of data part, and Length is the bytes of data part.

Definition 3.2 The source path of a CCP is constructed from the tags of Registered Elements of the CCP.

Definition 3.3 The leaf tag of a CCP is the leaf tag of the source path of the CCP. We cluster all CCPs with same leaf tag together and save them in two files, one for data part and one for info part (Figure 4).

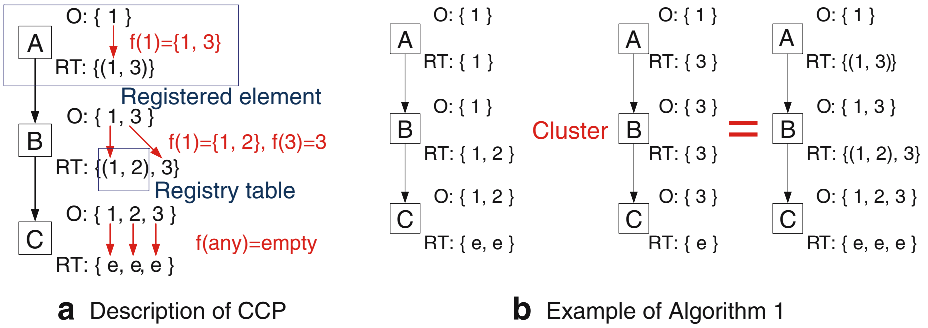

Figure 3 CCP labeling scheme. **a** Description of CCP. **b** Example of algorithm 1.

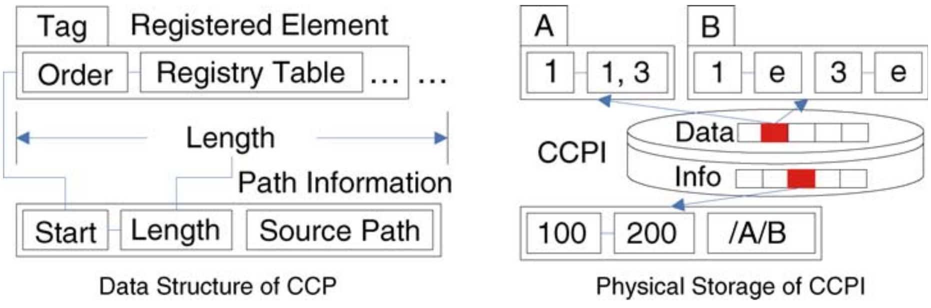

Figure 4 Data structure of CCPI.

4 Query processing based on CCPI

In this Section, we present query processing algorithms based on CCPI. We introduce algorithm KMP-Match-Path to process queries without branch in Section 4.1. Algorithm Related-Path-Segment-Join to process queries with branches is discussed in Section 4.2. About how to process query with nested branches or multi branches is given in Section 4.2.3.

4.1 Processing query without branches

The basic idea of processing query *Q* without branches is to match source paths of CCPs in CCPI to *Q*. The info part of CCPI is preloaded into memory as a hash table from tags to an array of path info objects before queries arrived. The reasons of preloading info part are that (1) path matching algorithm uses source path instead of CCP and (2) info part is usually quite small although the document may be large.

When query *Q* arrived, we use string match algorithm to match source path to *Q*. Although it's a co-NP problem to judge the containment of two XPath expressions [12], the problem in our paper is to judge the containment of a source path and a XPath expression without branches.

To make the problem simple, we need to know the concept of broken XPath expression sequence first. A *broken XPath expression sequence* is a sequence of sub strings decomposed from query XPath expression using delimiter "//". For example, XPath expression "//A//B" is decomposed into (Root, A, B), while "/A//B/C" is decomposed into (Root/A, B/C). After decomposing, there is no A-D axis "//" in broken XPath expressions.

```
Algorithm 2 KMP-Match-Path
      Input: Source Path SP, XPath
      Output: true if SP matches XPath
1     brokenXPaths= sub strings of XPath using delimiter
          "//"
2     for each brokenXPath in brokenXPaths {
3        SP=KMP-Match-Broken(SP, brokenXPath)
4        if (SP==null)  return false
5        else if (all brokenXPaths are matched & SP=="")
6        return true }

      Function KMP-Match-Broken(SP, brokenXPath)
    //see Knuth-Morris-Pratt algorithm for detail
```

We use Knuth–Morris–Pratt string matching algorithm [4] to perform matching from a source path to query XPath expression (Algorithm 2). Each time when function KMP-Match-Broken is called, we check if *brokenXPath* is matched by *SP*. If matched, the sub string from beginning to the matched point of *SP* is abandoned, the rest part of *SP* is returned. If not matched, function KMP-Match-Broken returns null which means matching failed.

Example 4.1 Suppose we match "//B//C/D" with source path "/A/B/B/C/C/D". The broken XPath sequence is {Root, B, C/D}. We add "Root" element to source path and perform matching. The first time function KMP-Match-Broken is called, since root matches root, it returns "/A/B/B/C/C/D". The second time when it is called, since "/A/B" matches "B", it returns "/B/C/C/D" and the last time it returns "" while the broken XPath sequence are all matched, and KMP-Match-Path returns true.

Since the algorithm is based on string matching, it's easy to match query expression with wildcard by add the rule that any element can match wildcard element "*" in query.

Notice that if there is no match in path info part for a query *Q*, then we can allege no solutions for *Q* without any disk IO.

Analysis of KMP-Match-Path Time complexity of KMP-Match-Broken is $O(L+QL_i)$ where *L* is length of source path *SP* and QL_i is the *ith* length of broken XPath expression, let *QL* is length of query XPath, then $QL=\Sigma_i QL_i$. The time complexity of KMP-Match-Path is $O(L+QL)$.

4.2 Processing query with branches

The basic idea of processing query with branches is to split the query into several root-to-leaf branches and corresponding related paths. Then we evaluate each branch as a query without branch, finally we join the intermediate results to final results by their related paths.

4.2.1 Algorithm related-path-segment-join

Algorithm Related-Path-Segment-Join joins two input CCPs with same SRPSs is present in algorithm 3. In algorithm 3, RES_1, RES_2 and *RES* are arrays of registered elements in CCP_1's *RPS*, CCP_2's *RPS* and output *CCP*'s *RPS*.

Algorithm Related-Path-Segment-Join operates in two phases. In the first phase (lines 1–2), related path segment in both CCPs are joined. Since CCP_1 is trunk, registered elements not contained in RPS of CCP_1 should also be returned to keep a complete *CCP*. In the second phase (lines 3–5), each registered element is filtered referring to its parent registered element.

Example 4.2 Consider query "//B[.//C]/D" on sample xml. Trunk is "//B/D" and branch is "//B//C" with related path is "//B". First, we evaluate "//B/D" using KMP-Match-Path, we get {CCP_1=/A[1{3}]/B[3{4}]/D[4{ε}]}. Then we evaluate "//B//C", we get {CCP_2=/A[1 {1, 3}]/B[1{1, 2}, 3{3}]/C[1{ε}, 2(ε), 3{ε}]}. Next we join CCP_1 and CCP_2 on their RPS, we get joined RPS "/A[1{3}]/B[3]". Since CCP_1 is trunk, D[4] does exist in B[3]'s registry table {4}, CCP_1 is solution.

Compared with TwigStack algorithm, algorithm 3 based on CCPI select "exact" D[4] which matches branch "//B/D", while D[1], D[2] and D[3] will not be considered at all. But in TwigStack algorithm, all D nodes will participate algorithm which may bring more disk IO and more comparisons.

Algorithm 3 Related-Path-Segment-Join

```
     Input: CCP$_1$ (trunk), CCP$_2$ with same SRPSs
     Output: CCP satisfy both query branches
1    for each RES[i] in CCP's RPS
2      RES[i]= Sort-Merge-Join (RES$_1$[i], RES$_2$[i])
3    for each RES[i] not in CCP's RPS
4      if (RES[i-1]==null)   return null
5        RES[i]=Fileter(RES$_1$[i], RES[i-1])
6    return CCP

     Function Filter(RES[i], RES[i-1])
       //RES[i-1] must has participated Sort-Merge-
       //Join or has been filtered
1    for each RT in RES[i-1]
2      Referring=Referring unite(RT)
3    Return Sort-Merge-Join(RES[i], referring)
```

Analysis of related-path-segment-join Because pre-orders in pre-order set O are different from each other, so does pre-orders in registry set RT, time complexities of Sort-Merge-Join on these sets are linear to the size of input. Suppose there are N_1 pre-ordered paths in CCP_1 and N_2 for CCP_2, there are at most N=min(N_1, N_2) elements in each registered elements in joined CCP. Each time when function filter is called, there are at most 2N pre-orders as the input of Sort-Merge-Join. Let L is the length of trunk or the length of SRPS if both CCP_1 and CCP_2 are not trunk, the time complexity of algorithm 3 in the worst case is $O(L\times N)$.

Theorem 4.1 *Algorithm 3 generates all correct join results.*

Proof Suppose a pre-ordered path $P_1=RPS+P_3$ is a solution to Q. Then there must exist a pre-ordered path $P_2=RPS+P_4$ which matches another branch in Q. P_1 is clustered in CCP_1 and P_2 is clustered in CCP_2 in algorithm 3. When we perform join on the RPS, the RPS part in P_1 is kept because P_2 in CCP_2 has the same RPS with P_1. Next, by filtering one by one, P_1 is outputted. □

In some situations, the RPS may be not unique. For example, when the query is "//B[.//C]//D", one source path to trunk "//B//D" is "/A/B/B/D", and the other to branch "//B//C" is "/A/B/B/C". In this situation, we can choose "/A/B" or "/A/B/B" as RPS. Actually, we always choose the shortest RPS to perform Related-Path-Segment-Join if there are many choices. The reason is that if we join with longer RPS, the results produced are sub set of result produced by choosing shorter RPS.

4.2.2 Unite RPSs before join

When several CCPs match a branch with same SRPS, we extract the whole RPS part W from them by uniting their RPSs, then we perform join on W only. Let's first see an example.

Example 4.3 Consider query Q "//B[.//C]//D", and the results to branch "//B//C" are {P_1=/A[1{1}]/B[1{1}]/C[1{ε}], P_2=/A[1{1}]/B[1{1}]/E[1{2}]/C[2{ε}], P_3=/A[1{3}]/B[3 {2}]/F[2{3}]/C[3{ε}]}, and the results to trunk "//B//D" are {T_1=/A[1{1}]/B[1{1}]/D[1{ε}], T_2=/A[1{1}]/B[1{2}]/G[2{2}]/D[2{ε}]}. Since P_1, P_2, P_3 share the same SRPS "/A/B" which matches related path "//B", we unite RPSs of P_1, P_2 and P_3 by algorithm Cluster-Path, and we get RPS_1="/A[1{1, 3}]/B[1{1}, 3{2}]". Similarly we can get RPS_2="/A[1 {1}]/B[1{1, 2}]" for T_1 and T_2. Then we join RPS_1 and RPS_2 by algorithm Related-Path-Segment-Join and get RPS_3="/A[1{1}]/B[1{1, 2}]" (notice that registry tables of B does not join). Next, we perform filtering with RPS_3 and T_1 and T_2, and output results (Figure 5).

If there are N_1 CCPs with same SRPS matching trunk and N_2 CCPs with same SRPS matching branch in algorithm 3. Let the length of RPS is L, the *ith* ($0<i<N_1+1$) RPS of CCP matches trunk has n_i pre-orders, the *jth* ($0<j<N_2+1$) RPS of CCP matches branch has m_j pre-orders. Then cost of uniting N_1 CCPs matching trunk is at most $L(\Sigma_i n_i)$, the cost of uniting N_2 CCPs matching branch is at most $L(\Sigma_j m_j)$. The cost of join operation on united RPS is at most $L(\Sigma_i n_i+\Sigma_j m_j)$. The cost of filtering is $L(\Sigma_i(n_i+\Sigma_i n_i)=2L(\Sigma_i n_i)$. The total cost is $L(3\Sigma_i n_i+2\Sigma_j m_j)$.

If we don't unite the RPSs first and perform Related-Path-Segment-Join on CCPs directly, the cost is $L(\Sigma_i\Sigma_j(n_i+m_j))=L(N_2\Sigma_i n_i+N_1\Sigma_j m_j)$. We can see that uniting RPSs first is faster when $N_2>3$ and $N_1>2$.

Theorem 4.2 *Algorithm 3 generates no repetitive results if we unite RPSs first.*

Proof If we unite RPS first, a CCP which matches trunk is decided as a solution or not to query only when it's filtered by joined RPS just once. It's impossible for it to output twice.

However, if we perform Related-Path-Segment-Join directly on CCPs, a CCP may be outputted more than once like T_1 in Example 4.3. □

4.2.3 Processing query with nested branches or multiple branches

If query Q has nested branches or multiple branches, we first split Q into multiple branches and their related paths. Q is divided as several sub queries; each sub query contains one related path and corresponding branches. A related path can only appear in one sub query

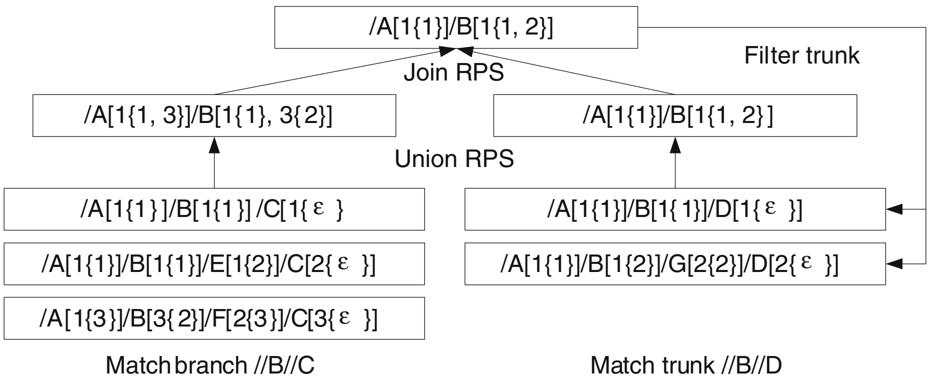
/A[1{1}]/B[1{1, 2}]
Join RPS
Filter trunk
/A[1{1, 3}]/B[1{1}, 3{2}]
/A[1{1}]/B[1{1, 2}]
Union RPS
/A[1{1}]/B[1{1}] /C[1{ ε }
/A[1{1}]/B[1{1}]/E[1{2}]/C[2{ ε }]
/A[1{3}]/B[3{2}]/F[2{3}]/C[3{ ε }]
/A[1{1}]/B[1{1}]/D[1{ ε }]
/A[1{1}]/B[1{2}]/G[2{2}]/D[2{ ε }]
Match branch //B//C
Match trunk //B//D

Figure 5 Example of uniting RPSs.

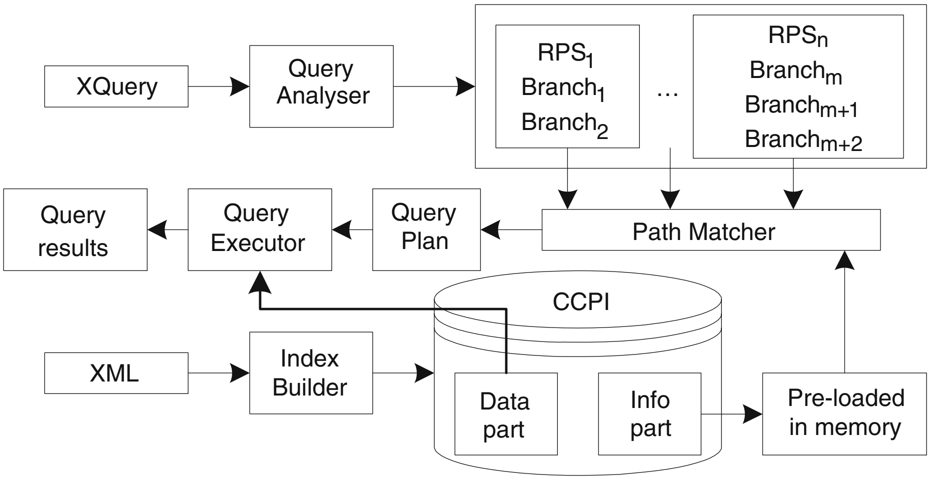

Figure 6 The architecture of query processing based on CCPI.

of *Q*, but a branch can appear in many sub queries of *Q*. Each branch is alleged solvable using algorithm KMP-Match-path. After that, a query plan is created, and then we perform Related-Path-Segment-Join for each sub query (Figure 6).

The problem of join order selection in query plan is an important issue, and we left it as our future work. In this paper, we select the sub query with the longest related path inside as the processing sub query.

Example 4.4 Consider query “//B[./C[./D]/B]/D” to sample xml (Figure 1). There are two sub queries with three branches and two related paths. One sub query is subQ$_1$ whose branches are “//B/C/D” and “//B/C/B” with related path “//B/C” the other is subQ$_2$ whose branches are “//B/C/D”, “//B/C/B” and “//B/D” with related path “//B”. Since “//B/C” is the longest related path, we process subQ$_1$ first. Because the branches in subQ$_1$ are not trunk, we return only joined RPS of the branches since only RPS is useful to join with other sub queries. The result RPS is directly the input data part of “//B/C/D” and “//B/C/B”, we just need to join it with data part of “//B/D” using related path “//B” and return query results (Figure 7).

If the sub query with longest related path is processed first, the fewest join operations can be assured. Since the costs of join operations may not be optimal, we left it as our future work.

Figure 7 Processing query with nested branches.

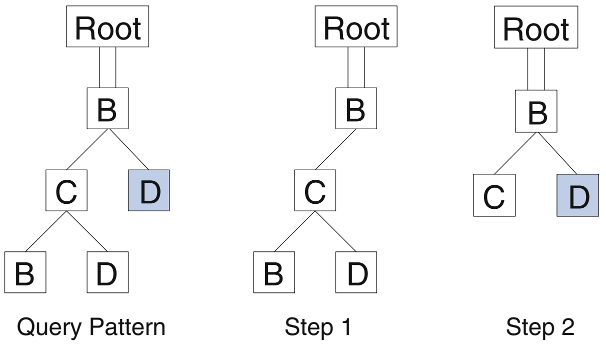

Table 1 Datasets of experiments.

Dataset	Elements	File size	Max depth	Tags	CCPs
XMark	1666315	116M	12	74	514
DBLP	3795138	156M	6	35	127
TreeBank	2437666	84M	36	250	338,748

Likewise, query with multiple branches such as "//B[./D]/C[./B]/D" consist of two sub queries with three branches and two related paths which are "//B/C/D" and "//B/C/B" with related path "//B/C" "//B/C/D", "//B/C/B" and "//B/D" with related path "//B". The processing procedure is just like Example 4.4.

Since query with branches is broken into many sub queries without branches and all branches will be checked if it's solvable before access data part in CCPI, we can allege that a query has no solutions if any branch is not matched in info part of CCPI.

4.2.4 A fast method when trunk is related path

For query *Q*="//B[./C]", and a matched *CCP* is "/A[1{1, 3}]/B[1{1, 2}, 3{3}]/C[1{ε}, 2(ε), 3{ε}]". The RPS of matched *CCP* which is "/A[1{1, 3}]/B[1{1, 2}, 3{3}]" is a solution to *Q*. Because the RPS in *CCP* does mean that "all possible B nodes which have child C and follow the source path /A/B", we just output the RPS of matched *CCP.*

If the RPS of *CCP* does not unique, each RPS is a separate solution. For example if a matched *CCP* is "/A[1{1}]/B[1{2}]/ B[2{1, 2}]/C[1{ε}, 2(ε)]", then both "/A[1{1}]/B[1 {1}]" and "/A[1{1}]/B[1{2}]/ B[2{1, 2}]" are solutions to *Q*.

5 Experiments

In this Section, we give our experimental setup in Section 5.1. Then we study the constructing time of CCPI in Section 5.2 and performance of algorithms based on CCPI in Section 5.3. Finally we discuss the scalability of our algorithms in Section 5.4.

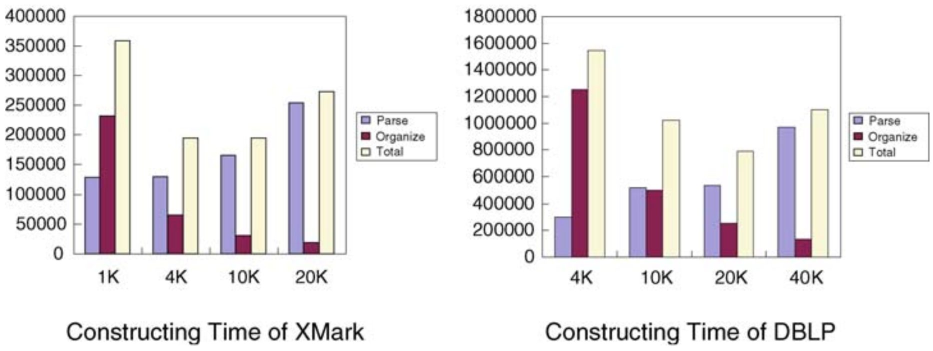

Figure 8 Constructing time of CCPI with different size of buffer.

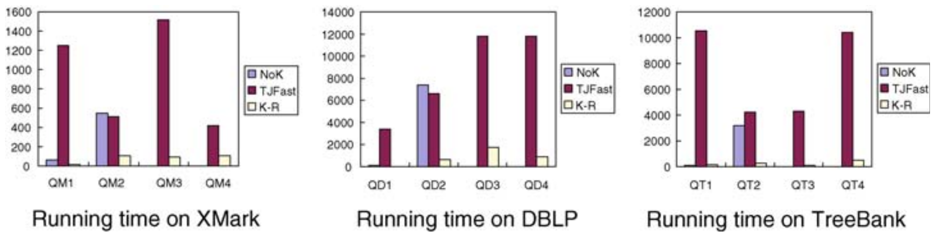

Figure 9 Query performance, K-R Is KMP-Match-Path or Related-Path-Segment-Join.

5.1 Experimental setup

All experiments are run on a PC with Pentium IV 3.0 G processor, 2 G DDR400 memory and 20 G SCSI hard disk. The OS is Windows 2000 Server. We implement our system using JDK1.5. We obtained the source code of Nok [21] and TJFast [10] from the original authors.

We used three different datasets, including one synthetic and two real datasets. The first synthetic dataset is the well-known XMark [18] benchmark data (with factor 1). The two real datasets are DBLP and TreeBank [15] (Table 1).

Table 1 shows that CCPI is so highly clustered that only a small amount of CCP against large amount of elements. Since the cardinality of CCPI is quite small, we can pre-load the info part into memory and build a hash table mapping from tag to corresponding path info set ended with that tag. When accessing data part of CCPI, we create a buffer whose size equals to the length of CCP data, thus we can load a CCP with one disk IO.

5.2 Constructing time of CCPI

First, we study the constructing time of CCPI with different size of buffer. If we use bigger buffer when constructing CCPI, the parsing time will be longer because more Cluster-Path is called and the organize time will be shorter. We test different size of buffer on XMark and DBLP.

It can be seen that it's not true that constructing CCPI with bigger buffer is faster than that with smaller buffer (Figure 8). Also, it's hard to know the exact buffer size with which the constructing time is the shortest.

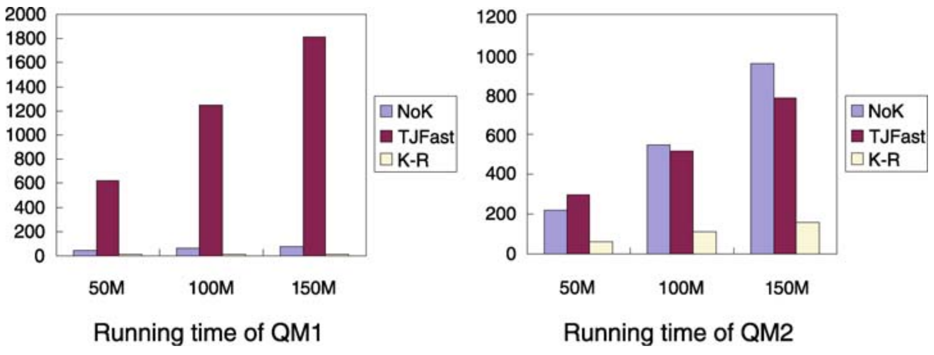

Figure 10 Scalability of algorithms.

Table 2 Query expressions.

Query	Query expressions
QM1	/site/regions/africa/item/description/parlist/listitem/text/keyword
QM2	/site/people[./person/profile[./education]/age]/person/phone
QM3	/site/closed_auctions//emph
QM4	/site/people/person[.//age]//education
QD1	/dblp/mastersthesis/year
QD2	/dblp/article[./ee]/year
QD3	/dblp//author
QD4	/dblp/article[.//author]//year
QT1	/FILE/_QUOTES_/S/VP
QT2	/FILE/EMPTY/S[./VP]/SBAR
QT3	/FILE/EMPTY/S/NP//SBAR
QT4	/FILE/_QUOTES_/S[.//NP]//VP

5.3 Queries and performance

We choose 4 queries on each dataset including: (1) a source path query; (2) a branch query without "//" (3) a path query with "/" and "//" but no branch; (4) a query with "/","//" and branches. The running times of the queries are shown in Figure 9.

We conclude that the algorithms based on CCPI outperform Nok and TJFast on all datasets from the experimental results shown in Figure 9. Since CCPI has small cardinality compared with other index, algorithms based on CCPI have less disk accesses than Nok and TJFast.

5.4 Scalability

To compare our algorithms with algorithms Nok and TJFast in team of scalability, all the algorithms were run on the datasets of sizes 50 M, 100 M and 150 M in XMark. The queries that were used in the experiments are QM1 and QM2 in Table 2. The results are shown in Figure 10.

From Table 3, it can be seen that the cardinality of CCPI against different dataset does not changes. Query processing based on CCPI against different dataset has the same IO counts and a few more IO bytes, therefore the scalability of our algorithms based on CCPI is much better than that of Nok and TJFast.

6 Related works

Bruno et al. [2] proposed a holistic twig join algorithm based on region encoding, namely TwigStack, TwigStack is I/O optimal for queries with only A-D edge but it is sub optimal for queries with P-C edge. Lu et al. [11] proposed a look-ahead method to reduce the

Table 3 Dataset for scalability.

Dataset	Elements	File size	Max depth	Tags	CCPs
XMark50	832911	57M	12	74	514
XMark100	1666315	115M	12	74	514
XMark150	2502484	174M	12	74	514

number of redundant intermediate paths. Jiang et al. [6] used an algorithm based on indexes built on containment labels. The method can "jump" elements and achieve sub-linear performance for twig pattern queries. Lu et al. [10] proposed TJFast based on a new labeling scheme called extended Dewey. TJFast only needs to access labels of leaf nodes to answer queries and significantly reduce I/O cost. BLAS by Chen et al. [3] proposed a *bi-labeling* scheme: *D-Label* and *P-Label* for accelerating P-C edge processing. Their method decomposes a twig pattern into several P-C path queries and then merges the results. Zhang et al. [21] proposed Nok pattern tree and algorithm for efficiently evaluating path expressions by NoK pattern matching.

1-index [13] is based on the backward bi-simulation relationship. It can easily answer all simple path queries. F&B Index [8] uses both backward and forward *bi-simulation* and has been proved as the minimum index that supports all branching queries. These "exact" indexes usually have large amount of nodes and hence size, therefore, a number of work has been devoted to find their approximate but smaller counterparts. *A(k)*-index [7] is an approximation of 1-index by using only *k-bi-simulation* instead of *bi-simulation*. *D(k)*-index [14] generalizes *A(k)*-index by using different *k* according to the workload. *M(k)*-index and *M*(k)*-index [5] further optimize the *D(k)*-index by taking care not to over-refining index nodes under the given workload. Wong et al. [17] give a survey on many index techniques for XML document. XQBE [1] provides a graphical environment to query XML. A short version of CCPI is proposed in [16].

7 Conclusions and future work

In this paper, we have proposed a novel index structure CCPI. It can extremely reduce the cardinality of index and support branch queries processing efficiently. Based on CCPI, we design efficient algorithms to process queries with or without branches. We implemented the algorithms and compared the performance with other algorithms'. We conclude that our algorithms performed better than others by experimental results. The advantage comes from less disk access due to high clustering of CCPI. Besides, our algorithms based on CCPI have good scalability.

Query optimization is important when there are many branches and related paths in query. We need to build a cost model based on CCPI to create query plan. It's one of our future works.

Acknowledgement Special thanks to Ning Zhang of University of Waterloo, and Jiaheng Lu of National University of Singapore for providing support for our experiments.

References

1. Braga, D., Campi, A.: XQBE: a graphical environment to query XML data. J. World Wide Web 8(3) 287–316
2. Bruno, N., Srivastava, D., Koudas, N.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD Conference, 310–321 (2002)
3. Chen, Y., Davidson, S.B., Zheng, Y.: BLAS: An efficient XPath processing system. In: Proc. of SIGMOD, 47–58 (2004)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms, 2nd edn., MIT(2001)
5. He, H., Yang, J.: Multi resolution indexing of XML for frequent queries. In: ICDE (2004)
6. Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic twig joins on indexed XML documents. In: Proceeding of VLDB 2003, 273–284 (2003)

7. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for efficient indexing of paths in graph structured data. In: ICDE (2002)
8. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: SIGMOD (2002)
9. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proc. of VLDB, 361–370 (2001)
10. Lu, J., Ling, T.W., Chan, C.Y., Chen, T.: From region encoding to extended dewey: on efficient processing of XML twig pattern matching. 193–204. In: Proc. of VLDB (2003)
11. Lu, JH., Chen, T., Ling, TW.: Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In Proceedings of CIKM Conference 2004, 533–542, (2004)
12. Miklau, G., Suciu, D.: Containment and equivalence for an XPath fragment. In: PODS, 65–76, (2002)
13. Milo, T., Dan Suciu, D.: Index structures for path expressions. In: ICDT, 277–295, Jerusalem, Israel (1999)
14. Qun, C., Lim, A., Ong, K.W.: D(k)-index: an adaptive structural summary for graph-structured data. In: ACM SIGMOD, 134–144 (2003)
15. U. of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/
16. Wang, H., Li, J., Wang, H.: Clustered chain path index for XML document: efficiently processing branch queries. In: Proc. of WISE, 474–486 (2006)
17. Wong, K.-F., Yu, J.X., Tang, N.: Answering XML queries using path-based indexes: a survey. J. World Wide Web 9(3):277–299
18. XMark: The XML-benchmark project. http://monetdb.cwi.nl/xml
19. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/
20. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/
21. Zhang, N., Kacholia, V., Özsu, M.T.: A succinct physical storage scheme for efficient evaluation of path queries in XML. In: ICDE 2004, 54–65 (2004)