TJFast: Effective Processing of XML Twig Pattern Matching

Jiaheng Lu, Ting Chen and Tok Wang Ling
School of Computing National University of Singapore
3 Science Drive 2, Singapore 117543
{lujiahen,chent,lingtw}@comp.nus.edu.sg

ABSTRACT

Finding all the occurrences of a twig pattern in an XML database is a core operation for efficient evaluation of XML queries. A number of algorithms have been proposed to process a twig query based on region encoding. In this paper, based on a novel labeling scheme: *extended Dewey*, we propose a novel and efficient holistic twig join algorithm, namely TJFast. Compared to previous work, our algorithm only needs to access the labels of *leaf* query nodes. We report our experimental results to show that our algorithms are superior to previous approaches in terms of *the number of elements scanned* and *query performance*.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: [Systems-query processing]

General Terms

Algorithm, Performance

Keywords

labeling scheme, holistic twig join

1. INTRODUCTION

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a *tree-structured* data model. Since the data objects in a variety of languages(e.g. XPath, XQuery) are typically trees, twig (i.e. a small tree) pattern matching is the central issue.

In this paper, motivated by the existing *Dewey ID* [4], we propose a new *powerful* labeling scheme, called *extended Dewey ID* (for short, *extended Dewey*). The unique feature of this scheme is that, from the label of an element alone, we can *derive the names of all elements in the path from the root to this element*. For example, Figure 1 shows an XML document with *extended Dewey* labels. Given the label "1.9.2.2" of element *text* alone, we can derive that the path from the *root* to *text* is "*/bib/book/chapter/section/text*". An immediate benefit of this feature is that, to evaluate a twig pattern, we *only need to access the labels of elements that satisfy the leaf node predicate in the query*. Further, this feature enables us to easily match a simple path pattern by string matching. Take element "1.9.2.2" as an example again. Since we see its path is "*/bib/book/chapter/section/text*", it is quite straightforward to determine whether this path matches a path pattern (e.g. "*//book/chapter*"). As a result, the *extended Dewey* labeling scheme provides us an *extraordinary* chance to develop a new efficient algorithm to match a twig pattern.

WWW 2005, May 10–14, 2005, Chiba, Japan.
ACM 1-59593-051-5/05/0005.

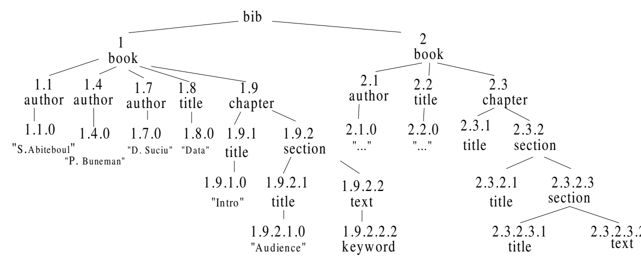


Figure 1: An XML tree with *extended Dewey* labels

Based on the *extended Dewey*, we present a new efficient algorithm, namely TJFast(i.e. a Fast Twig Join algorithm). Unlike previous algorithms TwigStack[1] and TwigStackList[2], in order to answer a twig query. TJFast only access the labels of query leaf nodes. Thus, TJFast significantly reduce I/O cost compared to previous work.

2. EXTENDED DEWEY AND FST

The intuition of *extended Dewey* is to use *module* function to create a mapping from an integer to an element name, such that given a sequence of integers, we can convert it into the sequence of element names. In the *extended Dewey*, we need to know a little additional schema information, which we call a *schema clue*. In particular, given any tag t in a document, the *schema clue* is all possible (distinct) names of children of elements with name t. This clue is easily derived from DTD, XML schema or statistic data on the document. Let us use $CT(t) = \{t_1, t_2, \cdots, t_n\}$ to denote the *schema clue* of tag t. Suppose there is an ordering for tags in $CT(t)$, where the particular ordering is not important. For example, consider the DTD in Figure 2; the tags of all possible children of *book* are *author*, *title* and *chapter*. So $CT(book) = \{author, title, chapter\}$. Using schema clue, we may easily create a mapping from an integer to an element name (i.e. element tag). Suppose $CT(t) = \{t_1, t_2, \cdots, t_n\}$, for any element e_i with name t_i,

```
<!ELEMENT bib  (book*)>
<!ELEMENT book  ( author+, title, chapter* )>
<!ELEMENT author   (#PCDATA )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, (text |section ) *)>
<!ELEMENT text (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph (#PCDATA | bold | keyword | emph)*>
```

Figure 2: DTD for XML document in Fig 1

if $t_i \neq t_n$. we assign an integer x_i to e_i such that $x_i \ mod\ n = i$, otherwise, $x_i \ mod\ n = 0$. Thus, according to the value of x_i, it is easy to derive its element name. For example, $CT(book) = \{author, title, chapter\}$. Suppose e_i is a child element of *book* and $x_i = 8$, then we see that the name of e_i is *title*, because $x_i \ mod\ 3 = 2$.

Given the *extended Dewey* label of any element, we may use a *finite state transducer* (FST) to convert this label into the sequence of element names which reveals the *whole path from the root to this element.*

Definition 1. (**Finite State Transducer**) Given *schema clues* and an *extended Dewey* label, we can use a *finite state transducer* (FST) to translate the label into a sequence of element names. FST is a 5-tuple (I, S, i, δ, o), where (i) the input set $I = N \cup \{0\}$; (ii) the set of states $S = \Sigma \cup \{PCDATA\}$, where $PCDATA$ is a state to denote text value of an element; (iii) the initial state i is the tag of the *root* in the document; (iv) the state transition function δ is defined as follows. For $\forall t \in \Sigma$, if $x = 0$, $\delta(t, x) = PCDATA$, otherwise $\delta(t, x) = F(t, x)$. No other transition is accepted. (v) the output value o is the current state name.

3. TWIG PATTERN MATCHING

It is straightforward to evaluate a query path pattern in our approach. We *only need to scan the elements whose tags appear in leaf nodes of query.* For each visited element, we first use FST to convert its label into element names along the path from the root to it, and then perform string-matching against it. If the path from the root to this element matches the desired path pattern, then we directly output the matching answers. As a result, we evaluate the path pattern efficiently by scanning the input list once and ensure that each output solution is our desired final answer.

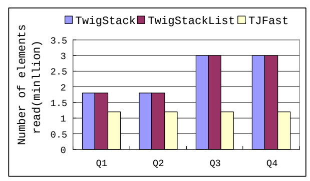
To answer a twig pattern,we propose a holistic twig join algorithm, called TJFast. The main idea of TJFast is to first output some solutions to individual root-leaf path patterns and then merge them to compute the answers to the whole query pattern. We call TJFast as a *holistic* approach. This is because when we output solutions for one root-leaf path in the first phase, the nodes in other paths are also taken into account. Holistic twig join algorithms can effectively control the size of intermediate results. The detail of the TJFast algorithm has to be omitted here due to space limitation but can be found in [3].

THEOREM 3.1. *Consider an XML database D and a twig query q with only ancestor-descendant relationships in branching edges. The worst case I/O complexity of TJFast is linear to the sum of the sizes of input and output lists.*

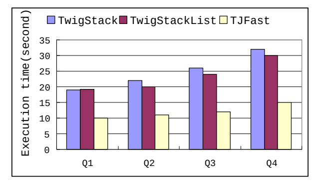
4. EXPERIMENTAL EVALUATION

We implemented three XML twig join algorithms:TJFast, TwigStack[1], TwigStackList[2] in JDK 1.4 using the file system as a simple storage engine. All experiments were run on a 1.7G Pentium IV processor with 768MB of main memory and 2GB quota of disk space, running windows XP system. We use the random data sets (with 3 millions nodes) consisting of five labels, namely a,b,...,e. The node labels in the data were uniformly distributed. We issue four twig queries: $a[.//b]//c$, $a[./b]/c$, $a[./b/c]/d/e$, $a[.//b/c]//d/e$, which have different structures and the combinations of *parent-child* and *ancestor-descendant* relationships.

Figure 3(a) shows the number of elements scanned by three algorithms and Figure 3(b) shows the execution time. Our first conclusion is that TJFast scan much less elements than TwigStack and TwigStackList. For example, in query Q3,Q4, TwigStack/TwigStackList read 3 millions elements, but TJFast/TwigStackList only read 1.2 millions elements. Our second conclusion is that TJFast outperforms TwigStack and TwigStackList for all ten queries. TwigStack/TwigStackList is comparable to TJFast only when the number of elements for *internal* nodes is very small.



(a) number of elements read



(b) Execution time

Figure 3: Performance measurements for TJFast,TwigStack and TwigStackList

5. CONCLUSION AND FUTURE WORK

XML twig pattern matching is a key issue for XML query processing. In this paper, we have proposed TJFast as an efficient algorithm to address this problem based on a novel labeling scheme: *extended Dewey*. Through this, not only do we reduce the disk access by only reading the labels of *leaf* nodes in query pattern, but we also improve the performance of twig pattern matching. We are currently researching how to use B trees, along with TJFast, to achieve sub-linear performance when the selective of query is high.

6. REFERENCES

[1] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[2] J. Lu, T. Chen, and T. W. Ling. Efficient processing of xml twig patterns with parent child edges: a look-ahead approach. In *CIKM*, pages 533–542, 2004.

[3] J. Lu, T. Chen, and T. W. Ling. TJFast: Efficient processing of XML twig pattern matching. Technical report, National university of Singapore, 2004.

[4] I. Tatarinov et al. Storing and querying ordered XML using a relational database system. In *SIGMOD*, pages 204–215, 2002.