


```

<!ELEMENT bib (book*)>
<!ELEMENT book ( author+, title, chapter* )>
<!ELEMENT author (#PCDATA )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, (text lsection ) *)>
<!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 $x_i \bmod n = i$, otherwise, $x_i \bmod n = 0$. 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 $x_i \bmod 3 = 2$.

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, \dots, 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.

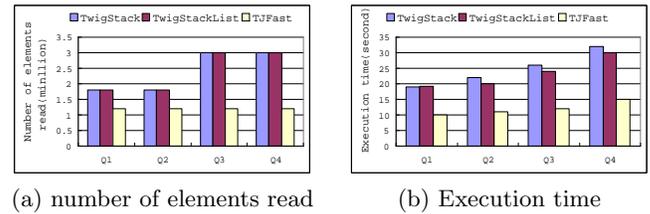


Figure 3: Performance measurements for TJFast, TwigStack and TwigStackList

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.