Amoeba Join: Overcoming Structural Fluctuations in XML Data

Taro L. Saito
University of Tokyo
JSPS Research Fellow
leo@cb.k.u-tokyo.ac.jp

Shinichi Morishita
University of Tokyo
moris@cb.k.u-tokyo.ac.jp

ABSTRACT

There are no universal rules for organizing data in XML. Consequently, semantically identical XML documents may have different structures; we call this *structural fluctuation* in XML. Finding all the structural fluctuations in an XML document requires verbose path expression queries. To overcome this problem, we developed a novel query processing primitive, called *amoeba join*. Amoeba join does not require explicit path structures in query statements; tag names or keywords are sufficient to perform searches. This paper introduces several amoeba join processing algorithms and demonstrates their performance.

1. INTRODUCTION

XML is now a global standard for describing structured data. In 2005, many vendors, including the Big Three suppliers of relational databases (IBM, Microsoft, and Oracle), launched new XML database engines. This trend will certainly result in increased XML capability, not only as a text format, but also as data stored in database management systems (DBMSs). The potential handling size and capacity of XML data is huge. Nevertheless, inconveniences have already materialized during the evolution toward this reality. Before the databases are explored using queries, it is difficult to find target elements because such large XML databases have complex and unclear path structures. In addition, it is difficult to write a query without knowledge about path structures.

A summary of path structures such as DataGuides [3] shows all existing paths in an XML database, but this is not sufficient to comprehend the actual structure of data in the target context; a path occurring in one context may not appear in a different context. An XML schema resolves this uncertainty in path occurrence to some extent, but not entirely. Since the XML schema allows the optional appearance of elements, unlike schemata in relational databases, path structures may still vary depending on context.

An XML document without a schema is like a black box for the user, but writing path queries for specific contexts is very difficult. In contrast, relational databases require schemata, making it considerably easier to find tables of the required data. Therefore, considerable effort and intensive research has been put into XML structure indices, allowing the processing of descendant axis queries that require less structural knowledge. However, this trend might not be addressing the real goal. People are enthusiastic about querying *data structures*, when they should be focusing on *structured data*. If we pursue writing paths in order to perform queries, we must first somehow acquire knowledge of path structures. To find the path structure for some specific context, we must issue queries to a 'black box' database. This is a chicken-or-egg situation — which comes first, the path structure or the query?

One way to overcome this problem is by relaxing XPath queries [1]. For example, the XPath query org/manager can be relaxed to org//manager by replacing the parent-child axis with the ancestor-descendant axis. This process reduces the burden of writing exact path query matches. However, the following example illustrates a problem not normally identified in the context of query relaxation:

```
<org department="head office">          <manager person="David">
  <manager>David</manager>          ⇒      <org department="head office">
  <location>Tokyo</location>                 <location>Tokyo</location>
</org>                                     </org>
                                        </manager>
```

Figure 1: An example of structural fluctuation

The two XML fragments shown above (**Figure 1**) represent data with the same meaning, but with different structures: the hierarchical order of org and manager tag is reversed. We call this *structural fluctuation* in XML. It is a structural variation in XML fragments that have the same elements (e.g., org and manager) and different structures.

XPath can track structural fluctuations, using disjunction in path patterns. For example, finding element pairs of org and manager in **Figure 1** requires the concatenation of at least two types of XPath query; /org/manager and /manager/org. In general, however, query statements become more complex because there could be many more elements to query and structural fluctuations in the document. Thus, the number of XPath expressions required to cover all possible path structures can easily balloon. For example, the number of query trees required to cover all structural fluctuations consisting of org, manager, and location elements is $3^{3-1} = 9$ (**Figure 2**) because it is identical to the enumeration of all labeled trees with n nodes, when the differences in axis (// or /) are ignored. Its enumeration size is known to be n^{n-1}. Concatenating all n^{n-1} query trees into a single regular path expression can be a daunting task.

Our research was motivated by this inconvenient method of path expressions. In this paper, we introduce the notion of an *amoeba*, which represents an equivalent class of

Copyright is held by the author/owner.
Ninth International Workshop on the Web and Databases (WebDB 2006)
June 30, 2006, Chicago, Illinois

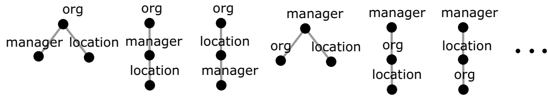


Figure 2: An amoeba (org, manager, location) covers $n^{n-1}(3^2 = 9)$ structural fluctuations.

structural fluctuations. An amoeba (org, manager, location) groups XML fragments that match one of the query trees illustrated in **Figure 2**. Applying this notion of an amoeba, we devised a novel query-processing method, *amoeba join*, which makes it possible to query XML databases without explicitly specifying path structures; tag names (and keywords) are sufficient to perform searches.

Even when using a schema or DataGuides [3], learning the entire XML data path structure is more difficult than creating a list of all types of tag and attribute names. We investigated a benchmark XML document provided by XMark [8] (scalability=1.0, 114 MB). The document contained 83 tags and attribute names and 548 distinct paths. Therefore, database users should have much more information on tags and attributes than path structures, which may differ depending on context. This is why query processing without explicit path structures, which is achieved by amoeba join, is promising.

This paper makes the following contributions:

- It introduces *amoeba join* as a method to capture structural fluctuations in XML data without explicitly specifying them using path queries.
- It presents three essential *amoeba join* processing algorithms and their experimental evaluations.

Semantics of XML Structure

Here, we demonstrate that XML structure provides surprisingly few semantics, clarifying the need to handle structural fluctuations in XML. First, consider the encapsulation of data with a tag. This process is normally used to group data elements or text data. In XML, it inevitably leads to a structural hierarchy among the data elements, which may or may not express high and low ranks. The following XML example (**Figure 3**) represents organization data with both superficial and semantic hierarchy order between the managers David and Michael:

```
<org department="head office">
  <manager> David </manager>
  <org department="R&D">
    <manager> Michael </manager>
  </org>
</org>
```

Figure 3: Nested organization data

It is also possible to reverse the hierarchical order. In the following example, the belongs_to tag is used to switch the hierarchical positions of the managers David and Michael without losing the semantic relationship:

```
<org department="R&D">
   <manager> Michael </manager>
   <belongs_to>
     <org department="head office">
        <manager> David </manager>
     </org>
   </belongs_to>
</org>
```

Furthermore, when a tag is used to group elements, there are generally no semantic ranks among the elements, as the structural change of org and manager in **Figure 1** illustrates. The org element has the manager information, and vice versa.

Therefore, hierarchical order does not directly represent the semantic relationship between data elements; semantic relationships become clear only when they are explicitly given. Consequently, it is natural to assume that XML data with neither explicit semantics nor any schema may contain some structural fluctuation. In our proposed method, we assume that XML databases contain arbitrarily structured information, and the user picks up node tuples matching an amoeba. Then, the retrieved data is transformed into a format designated by the the user.

The rest of this paper is organized as follows: Section 2 discusses the essential differences between the proposed method and other related studies. Section 3 introduces the notion of *amoeba* and *amoeba join*, and Section 4 presents several amoeba join processing algorithms. Section 5 demonstrates the performance of these queries. Finally, Section 6 presents our conclusions and directions for future work.

2. RELATED WORK

Querying an XML database without knowledge of path structure was first addressed by [7], and refined by [11]. Both studies used variations of the least common ancestor method (lca) to find the smallest tree containing all target nodes. Among the lca nodes that connect common node sets (tags or keywords), the one that forms the smallest subtree is defined as the smallest least common ancestor ($slca$) [11]. The precise definition of $slca$ is as follows: given k node sets $D_1, D_2, \ldots, D_k$, for example, D_1 and D_2 are node sets matching XPath //org, //manager, a node v belongs to the $slca$ if $v \in lca(D_1, \ldots, D_k)$ and for all $u \in lca(D_1, \ldots, D_k)$, v is not an ancestor of u. In summary, a subtree rooted from an $slca$ node does not contain other lca nodes.

One problem with this approach is that the $slca$ might be the root node of an XML document. XML is a single rooted tree, so every node set can be connected using the root node. In addition, when the $slca$ approach is applied to the previous example (**Figure 3**) to find pairs of org and manager elements, it misses the pair of org and manager David because these contain the subtree rooted by the $slca$ of org and manager Michael. In general, XML data semantics are too complex to be detected automatically using simple rules. In addition, although the method of [11] is optimized to search for $slca$ nodes, it focuses mainly on keyword versus database queries. It cannot detect element inclusion relationships. For example, it can find the keyword "Michael", but is not capable of assuring that "Michael" is contained within the manager tag.

XRank [5] applies keyword-based search to XML. It locates XML elements that contains all given keywords. Unlike $slca$, XRank is aware of recursion of XML structure. However, it suffers from two drawbacks: (1) it does not distinguish tag name from textual content; (2) it cannot express complex query semantics [7].

Finding an exact match in XPath queries can be difficult, and thus studies have investigated ways to relax the condition of rigorous matching in regular path expressions [1]. The types of relaxation are explained in [1]. These include dropping or weakening predicates or query nodes, and

Figure 4: Amoeba join (org, manager, location = "Tokyo")

adding an explicit disjunction, which is similar to querying all structural fluctuations. The proposed amoeba join method contains the essence of query relaxations, but is novel in that it is also able to handle situations in which the high and low nodes of a query tree are reversed.

DogmatiX [10] attempts to solve structural fluctuations using nearest neighbor heuristics that connect nodes within some metrics. However, the method cannot address all possible n^{n-1} structural fluctuations.

Approximate join [4] locates documents with similar structures and different forms. It is more general than amoeba join because it includes changes in tag names. Although approximate join can accommodate various similarity measures, it is optimized to tree edit distances, which must preserve ancestor node order in a query; that is, unlike amoeba join, it cannot reverse ancestor-descendant relationships.

Static typing of XML [2] is another way to handle structural fluctuation. It detects mismatches between paths described in query statements and schemata. Such discrepancies are reported as compile-time (static) errors of the query. This prevents writing invalid queries that do not match any document path. In other words, it is not necessary to cover all path possibilities because the query compiler presents the available paths. A major drawback of this approach, however, is that it still requires a schema, which is not mandatory in XML.

3. AMOEBA JOIN

The requirement of path structures within query statements is a serious obstacle to using XML DBMSs. *Amoeba join* is a method for overcoming this; problem by allowing structural fluctuations in the underlying XML database, and retrieving data matching the query of interest. This section presents a definition of *amoeba* and our novel query-processing method, *amoeba join*.

DEFINITION 1. [**Amoeba Join**] Let $D_1, \ldots, D_k$ be domains of XML nodes; then each $D_i (1 \leq i \leq k)$ represents a node set matching an XPath expression, an *amoeba join* $AJ(D_1, \ldots, D_k)$ gives a set of node tuples $\{(d_1, \ldots, d_k) | d_1 \in D_1, \ldots, d_k \in D_k$, such that one of $d_1, \ldots, d_k$ is a *common ancestor* of the others$\}$.

We call a node tuple $t = (t_1, \ldots, t_k)$ an amoeba (or an amoeba tuple) if $t \in AJ(D_1, \ldots, D_k)$. Its common ancestor $t_r (1 \leq r \leq k)$ is called the *amoeba root* of t. See **Figure 4** for an example of amoeba join using a tree representation of an XML document. When D_1 represents a node set belonging to a path expression '//org', D_2 = '//manager' and D_3 = '//location/text()="Tokyo"', respectively, an amoeba join $AJ(D_1, D_2, D_3)$ gives a set of node ID tuples $\{(\mathbf{2}, 5, 13), (\mathbf{2}, 8, 13), (21, \mathbf{15}, 17)\}$ (bold numbers represent amoeba roots). This notion of an amoeba can be adapted to various XML structures. It can capture the manager node (8) even if it is under the belongs_to tag (7), and it also tracks location node (17) behind the department tag (16). Furthermore, amoeba join detects hierarchical change of orders between org (21) and manager (15) nodes.

Note that amoeba join does not a process of computing the least common ancestors (*lca*) of a given node set. The *lca* nodes of org, manager and location nodes in **Figure 4** include the root node, company (id = 1). Every node in XML can reach the others through the root. For example, node 13 and 15, which are apparently irrelevant, can be connected via the root. Therefore, the lca method is not appropriate for finding relationships between nodes. Amoeba join is similar to the *lca* method in that it finds a common ancestor; however, it limits common ancestor nodes to those belonging to one of the given query domains. By using this rule, the relationships among nodes are bound to a common ancestor, i.e., the amoeba root. The tuple $(11, 8, 13)$ in **Figure 4** is not an amoeba because its nodes are not bound, while the other amoeba tuples are bound by 2 or 15. If there is no such bound, as in the *lca* method, the relationships among the connected nodes are very weak.

When the root node of XML happens to be contained in one of the domains, any node tuple becomes an amoeba. In general, such a query is no use. If the root node is required in order to specify the context of queries, ameba join with context nodes $AJC(context\ nodes, D_1, \ldots, D_k)$ is preferable. It restricts the search region of the query under the specified context nodes.

The result of AJ(org, manager, location = "Tokyo") in **Figure 4** has node overlaps in $(2, 5, 13)$ and $(2, 8, 13)$. They differ in manager nodes 5 and 8. Intuitively, the tuple of $(2, 5, 13)$ is preferable as a query result. However, if the user wants to list all of the managers at the Tokyo location, the pair $(2, 8, 13)$ is also meaningful. Therefore, filtering the results should be left to the user. If the user wants nodes that are close to each other in the tree structure of XML, the nearest-neighbor measure method can be used to filter the results. This filtering encounters problems in detecting the semantics of the XML structure, but this issue should be discussed separately, with the handling of structural fluctuation. Amoeba join is a query-processing technique to be employed before applying such semantic or heuristic filters.

Amoeba Join Syntax

Here, we introduce syntax of amoeba join expressions:

S := "(" E (, E)* ")"
E := F | `$variable` := F | S
F := *XPath-expr* (P `<value>`)? | `<value>`
P := $\Rightarrow$ | $=$ | $\neq$ | $<$ | $>$ | $\leq$ | $\geq$

To make expressive queries for extracting valuable information, we extended amoeba join to incorporate XPath expressions. For example, an amoeba join expression (org, manager) collects all pairs (amoebas) of org and manager elements that have ancestor-descendant relationships. Amoeba join can also be used with explicit path queries. (/org, "David") computes amoebas that have org nodes directly under the root node, and text nodes with a value of David. To express a node x such that a text y occurs in the subtree rooted at x, we provide the statement $x \Rightarrow y$. For example, the expres-

sion manager ⇒ "David" designates the manager tag containing the text node "David", whether it is a child or descendant node of the manager node.

We also offer another operation that allows nodes to be bound to variables. For example, (\$x = org, \$x/manager, location) joins org nodes and its child manager nodes to location nodes.

4. AMOEBA JOIN PROCESSING

Amoeba join processing locates node tuples composing *amoebas*, from given node domains $(D_1, \ldots, D_k)$. Therefore, amoeba join processing is independent of node retrieval from databases. This independence is important because it enables amoeba join to be incorporated into other existing query-processing techniques.

In the algorithm descriptions that follow, we assume that every XML node is labeled with an interval $(start, end)$ [6]. A pair of two arbitrary intervals is disjoint; one subsumes the other as a subrange. By encoding XML tree structure hierarchy in the form of an interval tree, detecting of ancestor-descendant relationships between two nodes becomes a containment test of two intervals, i.e. a node v_i is an ancestor of another node v_j iff $v_{i \cdot start} < v_{j \cdot start} \land v_{j \cdot end} < v_{i \cdot end}$.

First, we describe a process to determine whether a given node tuple is an amoeba. The function isAmoeba(t) receives a node tuple $t = (t_1, \ldots, t_k)$, and returns *true* if it finds a node interval in t with the smallest start value that completely contains the other intervals. Such an interval is the common ancestor of the others; i.e., this node tuple constructs an amoeba.

Brute-force Amoeba Join

With the decision function isAmoeba(t), we can write a simple brute-force amoeba join processing algorithm (**Algorithm 1**). This brute-force version computes all permutations of the input sets, but is apparently inefficient.

Algorithm 1 Brute-force Amoeba Join Algorithm

```
Input: Node sets D = (D_1, ..., D_k)
Output: A set of amoebas R
1: R ⇐ nil
2: for all node tuple t in the permutation of D do
3:    if isAmoeba(t) then
4:       push t into R
5:    end if
6: end for
7: return R
```

Two more efficient amoeba join algorithms are detailed below. The sweep algorithm improves the brute-force algorithm by sequentially sweeping the input node sets. The quicker algorithm reduces disk I/Os by localizing search regions.

Sweep Algorithm of Amoeba Join

By sorting the input node sets in advance and in the order of their start values, it becomes more efficient to find amoebas because the amoeba root of an amoeba $(t_1, \ldots, t_k)$ always has the smallest start value in $t_1, \ldots, t_k$. The sweep algorithm (**Algorithm 2**) searches amoeba root nodes by sweeping the sorted input node sets.

In Step 7 of **Algorithm 2**, a node t_s with the the smallest value in the input sets is assumed to be an amoeba root. Because no other element in the input sets has a smaller start value than t_s, scanning the range of $(t_s.start, t_s.end)$ in $D_j (1 \leq j \leq k, j \neq s)$ is sufficient to find all descendant nodes of t_s (Step 10). Then using these descendant nodes and t_s, we can enumerate all amoeba tuples rooted by t_s (Step 17). When the algorithm reaches Step 14, it is assured that all amoebas whose root's start value is smaller than or equal to the current amoeba root candidate t_s are found.

Algorithm 2 Sweep Amoeba Join Algorithm

```
Input: Sorted node sets D = (D_1, ..., D_k)
Output: R: a set of amoebas
1: R ⇐ nil.
2: while true do
3:    if some of D_1, ..., D_k is empty then
4:       return R // no more amoeba tuples
5:    end if
6:    create a node tuple t = (t_1, ..., t_k) from
      (D_1.front, ... D_k.front)
7:    Let s be the smallest start node index in t, then t_s is the
      smallest node in D_1, ..., D_k
8:    if isAmoeba(t) then
9:       // s is the amoeba root node index in t
10:      By searching the range of (t_s.start, t_s.end) in each D_j(1 ≤
         j ≤ k, j ≠ s), collect descendant nodes of t_s, then construct
         a set of these nodes A_j.
11:      A_s = {t_s} // contains only the current amoeba root
12:      If every A_j(1 ≤ j ≤ k) is not empty, all permutations of
         (A_1, ..., A_k) construct amoeba tuples, so insert them into
         R.
13:   end if
14:   remove t_s from D_s // all amoebas rooted by t_s are found
15: end while
```

Heuristics for Search Space Reduction

Here, we introduce the *quicker algorithm*, a more elaborate version of amoeba join, which is integrated with index lookups. While the sweep algorithm reads all nodes in the given query domains from the database, the quicker algorithm (**Algorithm 3**) tries to reduce this disk I/Os.

For a node tuple to be an amoeba, each node in the tuple must be a descendant of the amoeba root node. When a node v is considered a part of an amoeba, its amoeba root is either v or one of its ancestor nodes. **Figure 5** illustrates this idea of localizing database scans within the descendant area of an amoeba root node candidate. Given a pivot node, which is considered a component of an amoeba, the quicker algorithm in Step 5 finds its ancestor nodes, i.e., amoeba root candidates, then searches the descendant area for other components of amoeba tuples. The quicker algorithm chooses pivot nodes from the smallest domain, namely D_i, because the smaller the cardinality $|D_i|$, the fewer amoeba root candidates and their descendant nodes (components of amoebas).

For this purpose, we use the frequency count (or its estimation) of nodes belonging to each of the query domains. Given domains of an amoeba join query $(D_1, \ldots, D_k)$, let $E = (e_1, \ldots, e_k)$ be frequency of $D_1, \ldots, D_k$. When the value of $|D_i|$ is available, $e_i = |D_i|$, if not, $e_i = \infty$. A function f sorts e_i so that $e_{f(1)} \leq \cdots \leq e_{f(k)}$. Quicker algorithm chooses pivot nodes from $D_{f(1)}$ (Step 4).

The quicker algorithm (**Algorithm 3**) utilizes three types of index scan; for retrieving nodes in $D_{f(1)}$, which is the smallest domain (Step 2); for retrieving ancestor nodes of a pivot node (Step 5); and for scanning descendant nodes of an amoeba root candidate (Step 11). A database index that supports these three types of index scans is required to perform the quicker algorithm.

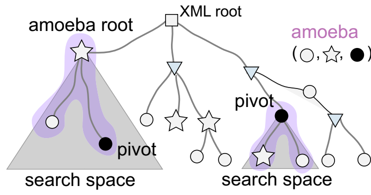


Figure 5: A small number of pivot nodes helps to reduce index scan ranges.

This type of search space reduction (**Figure 5**) is not available in the *lca* method, because a *lca* node tends to be the root of XML; it does not reduce the search space at all. Another reason that makes this optimization possible is the design concept of amoeba join, which tries to find common ancestor nodes from *specific domains*, while the approach of the *lca* or *slca* [11, 7] is to find common ancestors from the *entire nodes* in an XML document.

Disk I/O Performance

When the height of an XML tree is h, the number of nodes retrieved by one ancestor query is at most h. The quicker algorithm retrieves ancestor nodes for each node in $|D_{f(1)}|$, and thus it requires $h|D_{f(1)}|$ node retrievals from the database. Another factor that defines the disk I/O performance of the quicker algorithm is how many node retrievals the heuristic of **Figure 5** saves. Let D'_i be a subdomain of D_i, which is retrieved by the quicker algorithm; then, the number of nodes scanned in the quicker algorithm is $h|D_{f(1)}| + |D'_1| + \cdots + |D'_k|$. However, the sweep algorithm consumes all nodes in the query domains; i.e., it searches $|D_1| + \cdots + |D_k|$ nodes. When $|D_{f(1)}|$ is sufficiently small, as in the example shown in **Figure 5**, $|D'_i|$ is typically considerably smaller than $|D_i|$. In addition, the height of the XML, h, is generally limited; only rarely is h larger than 100. Consequently, the quicker algorithm is often less costly in terms of disk I/O than the sweep algorithm.

This search space reduction is similar to *pushing selection*, a query optimization technique for relational databases. XML typically contains many repeat paths, and therefore, reducing the size of query domains by attaching conditions, such as predicates on text values, to the path expression queries is a common method. Hence, the quicker algorithm utilizes a simple optimization to reduce disk I/Os.

5. EXPERIMENTAL RESULTS

We measured the performance of three amoeba join algorithms, brute-force (BF), sweep(SW), and quicker algorithm(QK). The first two algorithms can incorporate various indexing techniques, so we compared them using sequential scans (S) of XML nodes, and more efficient index-based scans (I). This let to five types of amoeba join algorithms: BF/S (brute-force with sequential scan), BF/I (brute-force with index scan), SW/S (sweep join processing with sequential scan), SW/I (sweep join processing with index scan), and QK (quicker algorithm), which is a mixture of index scanning and join processing.

Implementation

We implemented our amoeba join algorithms in C++ using B+-trees provided by the BerkeleyDB library [9]. We labeled each XML node with (start, end, level, path ID, parent ID, text). The pair (start, end) is an interval representation of XML nodes [6]. The start value can be used as a unique node ID, so parent ID is the start value of a parent node. The level is the depth of a node in the XML tree. The path ID represents an ID assigned to each independent path. The text is a text content encapsulated by tags or attributes.

XML nodes are stored in a B+-tree in ascending order of their start values. The sequential scan method (S) reads the stored nodes in this order. The parent node retrieval in the quicker algorithm (QK) also utilizes this B+-tree index. As for the index-base scan methods (SW/I and BF/I) and the quicker algorithm (QK), to make node retrieval faster, we generated a secondary B+-tree index using a compound key (path ID, start), which aligns XML nodes first in the order of path IDs, then that of start values. This secondary index is useful for finding descendant nodes that belong to specific paths. In addition, we constructed an inverted index for text values (text ⇒ start) that looks up the start value (ID) of a node from its text value.

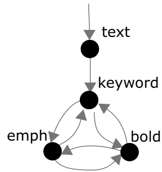
Because the sequential scan method reads the entire list of nodes to perform a query, it is somewhat analogous to node stream processing, such as in handling SAX events. Another reason to compare the index-based scan methods to the sequential scan methods is required to assure that the former, using secondary indexes, is not too complex to invoke a lot of random disk access. Too much random access may make query-processing algorithms slower than a sequential scan of all records.

The quicker algorithm (QK), used rough estimates of node frequencies; if D_i, a domain of an amoeba join query has a text predicate, we assume $|D_i| = 1$ or otherwise $|D_i| = \infty$, because the response size of a keyword search is usually less than that of a path query. Although a more accurate es-

Algorithm 3 Quicker Amoeba Join Algorithm

Input: Query domains $D_1, \ldots, D_k$ and sorting function f
Output: A set of amoebas, R

1: Initialize priority queues (sorted by start order) $Q_i \Leftarrow empty$ ($i = 1, \ldots, k$)
2: fill the $Q_{f(1)}$ with nodes in $D_{f(1)}$ by fetching from the database (index scan)
3: **for** $i = 1 \ldots |D_{f(1)}|$ **do**
4:   $pivot = Q_{f(1)}$.top
5:   query pivot's ancestor nodes (index scan), then push them into corresponding $Q_p (p \neq f(1))$.
6:   **repeat**
7:     $s =$ the smallest start node index in Q_1.top, ..., Q_k.top.
8:     $t_s = Q_s$.top // an amoeba root candidate
9:     pop all entries q ahead of the t_s, i.e. $\forall q \in Q_i, q.start < t_s.start$
10:     **for** $j = f(1) \ldots f(k)$ **do**
11:       push unread descendant nodes of t_s in D_j into Q_j. (index scan)
12:       **goto** Step 18 **if** Q_j is empty (t_s cannot be an amoeba root)
13:     **end for**
14:     // all of the $Q_p (p \neq f(1))$ is not empty
15:     By searching the range of $(t_s.start, t_s.end)$ in each $Q_j (1 \leq j \leq k, j \neq s)$, collect descendant nodes of t_s, then construct a set of these nodes A_j.
16:     $A_s = \{t_s\}$ // contains only the current amoeba root candidate
17:     If every $A_j (1 \leq j \leq k)$ is not empty, all permutations of $(A_1, \ldots, A_k)$ construct amoeba tuples, so insert them into R.
18:     pop Q_s // all amoebas rooted by t_s is computed
19:   **until** $s == f(1)$ // exit when the pivot node is popped
20: **end for**
21: **return** R



	XMark (factor = 0.1, 12M)					XMark (factor = 0.5, 57M)					XMark (factor = 1.0, 114M)				
	QK	**SW/I**	**SW/S**	**BF/I**	**BF/S**	**QK**	**SW/I**	**SW/S**	**BF/I**	**BF/S**	**QK**	**SW/I**	**SW/S**	**BF/I**	**BF/S**
Q1	2.71	**0.39**	5.47	*> 8d*	*> 8d*	22.91	**1.97**	30.81	*> 3y*	*> 3y*	62.20	**4.17**	69.09	*> 24y*	*> 24 y*
Q2	**0.06**	0.32	5.57	106.75	115.94	**0.05**	1.20	29.34	*> 0.5h*	*> 0.5h*	**0.06**	2.67	67.12	*> 11h*	*> 11h*
Q3	**0.05**	0.11	5.43	20.02	26.42	**0.07**	3.97	29.41	*> 0.1h*	*> 0.1h*	**0.06**	8.95	66.02	*> 0.5h*	*> 0.5h*
Q4	**0.06**	0.41	7.98	*> 30y*	*> 30y*	**0.05**	10.96	43.41	*> 162c*	*> 162c*	**0.07**	22.12	90.95	*> 2631c*	*> 2631c*

Q1 : (emph, bold, keyword) **Q2** : (emph, bold, keyword=>"aboard notes")
Q3 : (item, @id="item100", description) **Q4** : (item, @id="item100", description, location, text)
h : hours (= 3600 sec), d : days (= 24h), y: years (= 365d), c: centuries (= 100y)

Figure 6: Structural fluctuation in XMark (left). Amoeba Join Performance (sec.) (right).

timation strategy could be accommodated, this is sufficient for locating one of the small domains.

Data Sets

It is difficult to manipulate XML documents with structural fluctuations using current XML technology. As a result, XML document structure is currently rather simple and monotonous in order to facilitate processing with SAX, DOM or other APIs. Therefore, we could not present a real world example of fully fluctuated XML data. Such an example will be possible when XML databases are widespread. Instead, we used a section of XMark benchmark [8], which contains a lot of structural fluctuations under its text tags. **Figure 6** shows a part of its DataGuide [3], a summary of path structure. The cycles in the DataGuide show that three tags keyword, emph, and bold occur in arbitrary order within the document.

We prepared three types of XMark document, varying the scaling factors (f = 0.1, 0.5 and 1.0). Their structures were too enormous and too complex to determine the path structures for a specific context, showing that amoeba join is also useful for querying such complicated XML data.

Amoeba Join Performance

Figure 6 shows the performance of the amoeba join queries (**Q1** to **Q4**). In the brute-force algorithms BF/I and BF/S, some the computational complexity was too huge to compute the result; thus, we show their estimation time, which was calculated using the permutation size of a query and the elapsed time for processing its first 500,000 nodes.

In **Q1**, the quicker algorithm was slower than the sweep algorithm (SW/I) because the sizes of emph, bold, and keyword were fairly large. As a consequence, excessive ancestor node retrievals in the quicker algorithm deteriorated its performance. When a query contains predicates (**Q2**, **Q3**, and **Q4**), the quicker algorithm performs an order of magnitude faster than the others because the size of the domain constrained by a constant gets smaller. Therefore, a combination of QK and SW/I algorithms provides the fastest performance; when there is no low-frequency domain in a query, it uses the SW/I, and otherwise it uses the QK.

The performance of SW/S scaled according to the database size. Although the time requried to scan the entire database was the same from **Q1** to **Q4**, the processing of **Q4** in SW/S was the slowest; because the tuple size k of a query affects the join performance. The same is true for SW/I. However, the performance of the quicker algorithm was stable regardless of tuple size.

6. CONCLUSION & FUTURE WORK

Managing structural fluctuations in XML is a challenge because the hierarchy of XML does not always have a significant meaning. Amoeba join is a method for querying XML data with various structures without using explicit path expressions. Among the presented amoeba join algorithms, the quicker algorithm performed well, and it is scalable to the size of an XML document.

There are several interesting problems that we did not address in this paper. One of them is optional or multiple appearances of nodes within an amoeba. Nodes not included in an amoeba require another amoeba join algorithm. Eliminating duplicate node appearances in an amoeba join result is also an interesting problem to be addressed in the future. This issue is somewhat similar to the operation of the 'distinct' keyword in XQuery and SQL, although the semantics of XML structure might be required to reflect the intention of the user on in query results.

In addition, nested amoeba join should be supported. For example,(manager, (org, department $\Rightarrow$ "R&D")) first computes an amoeba set AJ(org, department $\Rightarrow$ "R&D"), then for each amoeba $(v_i, v_j) \in AJ$ repeats a process of the amoeba join (org, v_i, v_j). Due to limited space, we cannot mention in this manuscript, its further details will be reported elsewhere.

7. REFERENCES

[1] S. Amer-Yahia, L. V. Lakshmanan, and S. Pandit. FleXPath: Flexible structure and full-text querying for XML. In *proc. of SIGMOD*, 2004.

[2] D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler. *XQuery from the Experts.* Addison Wesley, 2004.

[3] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *proc. of VLDB*, 1997.

[4] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *proc. of SIGMOD*, 2002.

[5] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *proc. of SIGMOD*, 2003.

[6] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *proc. of VLDB*, 2001.

[7] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *proc. of VLDB*, 2004.

[8] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. manolesch, and R. Busse. XMark: A benchmark for XML data management. In *proc. of VLDB*, 2002.

[9] Sleepycat Software. BerkeleyDB. available at http://www.sleepycat.com/.

[10] M. Weis and F. Naumann. DogmatiX tracks down duplicates in XML. In *proc. of SIGMOD*, 2005.

[11] Y. Xu and Y. Papaconstantinou. Efficient keyword search for smallest LCAs in XML databases. In *proc. of SIGMOD*, 2005.