Efficient Evaluation of Partial Match Queries for XML Documents Using Information Retrieval Techniques

Young-Ho Park[1], Kyu-Young Whang[1], Byung Suk Lee[2], and Wook-Shin Han[3]

[1] Department of Computer Science and
Advanced Information Technology Research Center (AITrc)**
Korea Advanced Institute of Science and Technology (KAIST), Korea
{yhpark, kywhang}@mozart.kaist.ac.kr
[2] Department of Computer Science
University of Vermont Burlington, VT, USA
bslee@cs.uvm.edu
[3] Department of Computer Engineering
Kyungpook National University, Korea
wshan@knu.ac.kr

Abstract. We propose XIR, a novel method for processing partial match queries on heterogeneous XML documents using information retrieval (IR) techniques. A partial match query is defined as the one having the descendent-or-self axis "//" in its path expression. In its general form, a partial match query has branch predicates forming branching paths. The objective of XIR is to efficiently support this type of queries for large-scale documents of heterogeneous schemas. XIR has its basis on the conventional schema-level methods using relational tables and significantly improves their efficiency using two techniques: an inverted index technique and a novel prefix match join. The former indexes the labels in label paths as keywords in texts, and allows for finding the label paths matching the queries more efficiently than string match used in the conventional methods. The latter supports branching path expressions, and allows for finding the result nodes more efficiently than containment joins used in the conventional methods. We compare the efficiency of XIR with those of XRel and XParent using XML documents crawled from the Internet. The results show that XIR is more efficient than both XRel and XParent by several orders of magnitude for linear path expressions, and by several factors for branching path expressions.

1 Introduction

Recently, there have been significant research on processing queries against XML documents [30]. To our knowledge, however, most of them considered only a limited number of documents with a fixed schema, and thus, are not suitable for large-scale applications dealing with heterogeneous schemas–such as an Internet search engine [20] [29]. A novel method is needed for these applications, and we address it in this paper.

Partial match queries in XPath [7] can be particularly useful for searching XML documents when their schemas are heterogeneous while only partial schema information is known to the user. Here, a partial match query is defined as the one having the

** This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

descendent-or-self axis "//" in its path expression. A full match query [18] can be considered a special case of a partial match query.

Partial match queries can be classified into *linear path expressions (LPEs)* and *branching path expressions (BPEs)*. An LPE is defined as a path expression consisting of a sequence of labels having a parent-child relationship or an ancestor-descendent relationship between labels; a BPE is defined as a path expression having branching conditions for one or more labels in the LPE.

Existing methods for providing partial match queries can be classified into two types: schema-level methods [24] [14] [15] [8] and instance-level methods [17] [26] [4] [6] [16] [5] [9] [10] [12]. The ones of the first type are usable for both partial match queries and BPEs, but they are not designed for use in large-scale documents of heterogeneous schemas [24] [14] [15] or have only limited support for partial match queries and do not explicitly handle BPEs [8]. The ones of the second type can support both, but can not be best used in a large-scale database because of inefficiency. Between these two classes of methods, the schema level methods are much more feasible than the instance level methods for large-scale XML documents because of their abilities to "filter out" document instances at the schema level. We thus adopt the schema-level methods as the basis of our method.

We particularly base our method on the schema-level methods using *relational tables,* such as XRel [24] and XParent [14] [15]. There are two reasons for this. First, those methods can utilize well-established techniques on relational DBMSs instead of a few native XML storages. Second, those methods can also utilize SQLs to query XML documents. For the query processing, they store the schema information and instance information of XML documents in relational tables, and process partial match queries in two phases: first, find the XML documents whose schemas match a query's path expression, and second, among the documents, find those that satisfy selection conditions (if there are any) specified on the path expression.

However, query processing efficiencies of the two existing methods, XRel and XParent, are too limited for large-scale applications, as we will show in our experiments in Section 6. The hurdle in the first phase is the large amount of schema information, and the hurdle in the second phase is the large number of document instances.

The objective of our method (we name it *XIR*) is to improve the efficiencies in both phases. Specifically, for the first phase, we present a method that adopts the *inverted index* [22] technique, used traditionally in the Information retrieval (IR) field, for searching a very large amount of schema information. IR techniques have been successfully used for searching large-scale documents with only a few keywords (constituting partial schema information). If we treat the schema of an XML document as a text document and convert partial match queries to keyword-based text search queries, we can effectively search against heterogeneous XML documents using partial match queries. For the second phase, we present a novel method called, *prefix match join*, for searching a large amount of instance information.

In this paper, we first describe the relational table structures for storing the XML document schema and instance information, and then, describe the structure of the inverted index. We then present the algorithms for processing queries. We also present the prefix match join operator, which plays an essential role in the evaluation of BPEs, and present an algorithm for finding the nodes matching the BPE. Then, we discuss the performance of XIR in comparison with that of XRel and XParent, and verify our comparison through experiments using real XML document sets collected by crawlers from the Internet. The results show that XIR outperforms both XRel and XParent by several orders of magnitude for LPEs and by several factors for BPEs.

This paper makes the following novel contributions toward large-scale query processing on heterogeneous XML documents:

- In XIR, we apply the IR technology to the schema-level information rather than to the instance-level information of the XML documents. In a large-scale heterogeneous environment, schema-level information as well as instance-level information would be extremely large. By applying the IR technique to the schema-level information, we can improve performance significantly by achieving schema-level filtering. i.e., restricting the instances to be searched to those whose schema matches the query's path expression.
- XIR also presents a novel instance-level join called the prefix match join for efficiently processing queries involving BPEs. The prefix match join improves performance significantly by minimizing the number of joins for finding instance nodes satisfying branching predicates.

2 Preliminaries

2.1 XML document model

Our XML document model is based on the one proposed by Bruno et al. [6]. In this model, an XML document is represented as a rooted, ordered, labeled tree. A *node* in the tree represents an element, an attribute, or a value; an *edge* in the tree represents an element-subelement relationship, element-attribute relationship, element-value relationship, or attribute-value relationship. Element and attribute nodes collectively define the document structure, and we assign labels (i.e., names) and unique identifiers to them. Figure 1 shows an example XML tree of a document. In this figure, all leaf nodes except those numbered 15 and 27 (representing the two attribute values "R" and "T") represent values and all non-leaf nodes except those numbered 14 and 26 (representing the attribute `@category`) represent elements. Note that attributes are distinguished from elements using a prefix '@' in the labels.

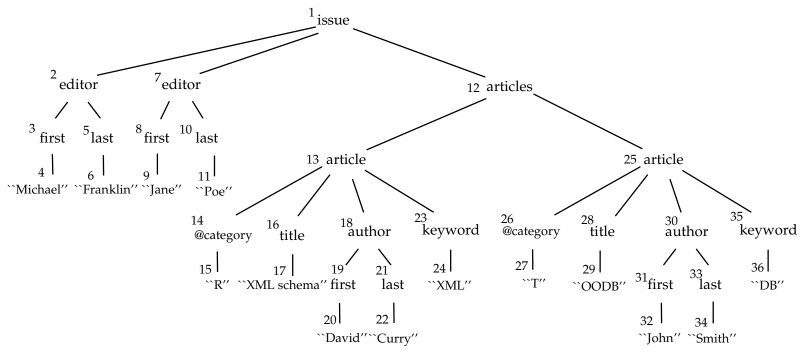

Fig. 1. An example XML tree of a document.

We modify this model so that a node represents either an element or an attribute but not a value. We also extend the model with the notions of label paths and node paths as defined below.

Definition 1. A *label path* in an XML tree is defined as a sequence of node labels $l_1, l_2, ..., l_p$ ($p \geq 1$) from the root to a node p in the tree, and is denoted as $l_1.l_2. \cdots .l_p$. □

Definition 2. A *node path* in an XML tree is defined as a sequence of node identifiers $n_1, n_2, ..., n_p$ ($p \geq 1$) from the root to a node p in the tree, and is denoted as $n_1.n_2. \cdots .n_p$. $\square$

Label paths represent XML document structures and are said to be *schema-level* information. In contrast, node paths represent XML document instances and are said to be *instance-level* information. We say a label path *matches* a path expression, a node path *belongs to* a label path, and a node path *is obtained from* a path expression. For example, in Figure 1, `issue.editor.first` is a label path matching a path expression `//editor//first`, and 1.2.3, 1.7.8 are node paths belonging to the label path. Note that there may be more than one node path belonging to the same label path because there may be more than one instance with the same structure.

2.2 XML query model

Our query language belongs to the tree pattern query (TPQ) class [2]. The query language supports two kinds of path expressions: 1) linear path expressions (LPEs) and 2) branching path expressions (BPEs).

An LPE is expressed as a sequence of labels connected with '/' or '//' as in Definition 3.

Definition 3. A *linear path expression* is defined as $l_0 o_1 l_1 o_2 l_2 \cdots o_n l_n$, where l_i ($i = 0, 1, \cdots, n$) is the i-th label in the path, and o_j ($j = 1, 2, \cdots, n$) is either '/' or '//' which, respectively, denotes a parent-child relationship or an ancestor-descendant relationship between l_{j-1} and l_j. Here, l_0 is the root of the XML tree denoting the set of all XML documents (i.e., `document("*")`) and may be omitted. $\square$

A BPE is expressed as an LPE augmented with 'branch predicate expressions' $[C_i]$ for some labels l_i ($i \in \{1, 2, \cdots, n\}$) [7] as in Definition 5. As in some work in the literature [1] [21], for simplicity, we consider only simple selection predicates[4] for the branch predicate expressions as in Definition 4.

Definition 4. A *branch predicate expression* C_k is defined as an expression L or $L\,\theta\,v$, where L is a linear path subexpression $o_{k1} l_{k1} o_{k2} l_{k2} \cdots o_{kp} l_{kp}$ ($p \geq 1$), v is a constant value, and θ is a comparison operator ($\theta \in \{=, \neq, >, \geq, <, \leq\}$). L specifies the existence of a node path $n_1.n_2. \cdots .n_p$ that belongs to the label path matching the LPE $l_0 o_1 l_1 o_2 \cdots o_k l_k o_{k1} l_{k1} o_{k2} l_{k2} \cdots o_{kp} l_{kp}$, and $L\,\theta\,v$ further specifies the node path to satisfy the selection condition on n_p. $\square$

Definition 5. A *branching path expression* is defined as $l_0 o_1 l_1 [C_1] o_2 l_2 [C_2] \cdots o_n l_n [C_n]$ where (1) $l_0 o_1 l_1 o_2 l_2 \cdots o_n l_n$ is an LPE defined in Definition 3 and (2) C_k ($k = 1, 2, \cdots, n$) is a *branch predicate expression* as defined in Definition 4, where some (not all) of them may be omitted. $\square$

The following query is an example BPE for retrieving the `title` elements that are children of the `article` elements that contain at least one `keyword` element and that are descendants of an `issue` element having a descendant `author` element whose child `last` element has the value `"Curry"`.

Q1:: `/issue[//author/last="Curry"]//article[/keyword]/title`

[4] This can be easily extended to consider compound (e.g., conjunctive) predicates as supported in other work in the literature [4] [6]. However, we omit this issue since it is not the focus of this paper.

Note that this BPE on the element `issue` has a selection condition on the label `last` in the LPE `//issue//author/last` and an existential condition on the label `key-word` in the LPE `//issue//article/keyword`.

2.3 XML query patterns

In this paper, we model a path expression as a *query pattern* defined in Definition 6. We modify the definition of the twig pattern originally used in the Holistic Twig Join [6] to formally represent the notions that we use in this paper. The definition is based on the BPE, as defined in Definition 5.

Definition 6. Given a path expression $o_1 l_1[C_1] o_2 l_2[C_2] \cdots o_n l_n[C_n]$ defined in Definition 5 (with l_0 omitted), we represent it as a *query pattern* that consists of a binary tree and a dangling edge connected to its root and that has the following properties:

- An edge represents o_j ($j \in \{1, 2, \cdots, n\}$) in the path expression. The edge is shown as a single line if o_j is '/' and as a double line if o_j is '//'. The dangling edge represents o_1.
- A node represents a label l_k ($k \in \{1, 2, \cdots, n\}$) in the path expression. The root node represents the label l_1.
- The left child of a node representing l_k ($k \in \{1, 2, \cdots, n-1\}$) represents l_{k+1}.
- The right subtree of a node representing l_k ($k \in \{1, 2, \cdots, n\}$) is the query pattern representing the branching predicate expression $C_k \equiv o_{k1} l_{k1} o_{k2} l_{k2} \cdots o_{kp} l_{kp}$ ($p \geq 1$).
- If the label represented by a node has a selection condition ("$\theta\, v$") on it, then the node is earmarked with "$\theta\, v$". □

The twig pattern [6] does not distinguish between the subtree whose root is also the root of the XML tree and the one whose root is not, if both match the same pattern. In contrast, the query pattern does distinguish between them by showing the dangling edge using a single line in the former case and a double line in the latter case.

Related to the query pattern, we use the following terms in this paper.

- One of the nodes in a query pattern is retrieved as the query result. This node corresponds to the label l_n in the LPE defined in Definition 3 or the BPE defined in Definition 5. We call this node the *result node* and distinguish it from the other nodes by shading it gray.
- Some of the nodes in a query pattern have a right subtree. We call such a node a *branching node*. Any node corresponding to a label l_k followed by $[C_k]$ as in $l_k[C_k]$ shown in Definition 5 is a branching node.

Figure 2 shows the query pattern of the query Q1 in Section 2.2. The node `title` is the result node, and the nodes `issue` and `article` are branching nodes.

As a special case of the query pattern, we define the *linear query pattern* as follows.

Definition 7. The query pattern of a linear path expression is called the *linear query pattern*. Compared with the query pattern defined in Definition 6, a linear query pattern has no branching node. □

In this paper, we use the terms *root label*, *leaf label*, *result label*, and *branching label* in a path expression interchangeably with the *root node*, *leaf node*, *result node*, and *branching node* in a query pattern. For example, in Figure 2, `issue` is the root label of the query Q1; `title`, `keyword`, and `last` are leaf labels; `title` is the result label; and `issue` and `article` are branching labels.

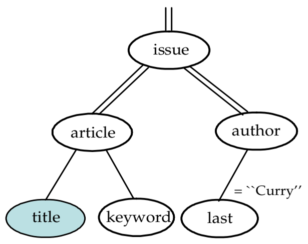

Fig. 2. Query pattern of the query Q1.

3 Related Work

As mentioned in Introduction, there are two kinds of methods for evaluating path expressions: schema-level methods and instance-level methods. A schema-level method uses structural information like the label paths to find nodes matching a path expression [14] [15] [24] [8], whereas an instance-level method uses only node identification information like the start and end positions of a node [4] [6] [16]. In this section, we briefly discuss instance-level methods, and then, focus on schema-level methods.

3.1 Instance-level methods

There have been three different approaches for the instance-level method. The first uses XML tree navigation [3] [13] [19]. It converts a path expression to a "state machine"[5], and then evaluates the path expression by navigating the XML tree guided by the state machine. The second uses node instance information stored for each node in an XML tree [4] [6] [16] [17] [23] [26]. It converts a path expression to a (structural) join query, and then evaluates the join query using the node instance information. The query evaluation in this approach, however, involves comparing the node instance information, and therefore, tends to be more expensive than in the schema-level methods, which can filter out node instances significantly by using the schema information. The third uses information retrieval (IR) technique, particularly an inverted index created on XML documents [5] [9] [10] [12]. Although using inverted indexes, however, they are fundamentally different from XIR, which creates an inverted index on the *label paths*, which are schema-level information.

3.2 Schema-level methods

Schema-level methods are categorized into those using special purpose indexes [8] [11] and those using relational tables [24] [14] [15] depending on where and how label paths are stored. In the former case, label paths are stored dynamically as they are used in the queries. In the latter case, all label paths in the documents are stored in the tables of a relational DBMS a priori.

Index Fabric [8] is considered the representative method in the schema-level methods using special purpose indexes. Index Fabric uses the Patricia trie to index the label paths and values that have occurred in the queries occurring frequently. However, Index Fabric is not meant to support partial match queries. Furthermore, the method is not designed to support BPEs, which are very effective for searching in a heterogeneous environment. These are critical drawbacks that render the method inapplicable in a large-scale, heterogeneous environment. Thus, in this section, we primarily focus on the schema-level methods using relational tables.

[5] A representation of the sequence of labels in the path expression as a sequence of states in finite state automata.

XRel [24] and XParent [14] [15], which are the two representative ones among the schema-level methods using relational tables, provide a basis for our XIR method. We describe each method in this subsection. We use the term *node* interchangeably with *element* or *attribute* as these are represented as nodes in the XML document model and the query pattern.

XRel In XRel, the XML tree structure information is stored in the following four tables [24]:

`Path``(label_path_id, label_path)`
`Element``(document_id, label_path_id, start_position, end_position, sibling_order)`
`Text``(document_id, label_path_id, start_position, end_position, value)`
`Attribute``(document_id, label_path_id, start_position, end_position, value)`

XRel uses two techniques for evaluating LPEs and BPEs: *string match* and *containment join*. The former belongs to the schema-level method and is used to handle LPEs; the latter belongs to the instance-level method and is used to handle BPEs.

In the case of an LPE, XRel first finds the label paths matching the query's path expression from the `Path` table. The matching is done using the SQL string match operator `LIKE`. All label paths in the `Path` table must be scanned in this case because an index like the B+-tree cannot be used to search for a partially matching label path. Then, XRel joins the set of matching label paths with the table `Element` via the column `label_path_id` to obtain the result nodes.

For the case of a BPE, we use the query pattern defined in Section 2.3. XRel first decomposes a BPE into multiple LPEs consisting of one LPE from the root to each branching node and one LPE from the root to each leaf node. For example, a BPE $/l_1[/l_2/l_3 = v_2]/l_4$ is decomposed into three LPEs $/l_1$, $/l_1/l_2/l_3$, and $/l_1/l_4$. Then, for each LPE, XRel finds the set of nodes (we call it a *node set*) obtained from the LPE in the same manner described above and reduces the set to those satisfying a selection condition (e.g., $/l_1/l_2/l_3 = v_2$). Then, it compares the node set obtained from an LPE ending at a branching node (e.g., $/l_1$) with the node set obtained from the LPEs ending at the leaf nodes (e.g., $/l_1/l_2/l_3$, $/l_1/l_4$) and, among the nodes obtained from the latter LPEs, retains only those that are descendants of the nodes in the former node set. This is done using the *containment join* which is implemented as a θ-join comparing the start positions and end positions of nodes.

XParent XParent [14] [15] is similar to XRel, but uses a different table schema so that it can implement the containment join operator using equi-joins instead of θ-joins. The schema is as follows [14].

`LabelPath``(label_path_id, length, label_path)`
`Element``(document_id, label_path_id, node_id, sibling_order)`
`Data``(document_id, label_path_id, node_id, sibling_order, value)`
`Ancestor``(node_id, ancestor_node_id, offset_to_ancestor)`
`DataPath``(parent_node_id, child_node_id)`

In query processing, XParent evaluates an LPE in the same way as XRel. In the case of a BPE, however, XParent generates a smaller number of LPEs than XRel because it generates only those from the root to each leaf node of a query pattern. Then, after retrieving the node set in the same manner as in XRel, the node set obtained from the LPE containing the result node is reduced through joins with those obtained from the other LPEs. Here, the join is performed as an equi-join through the table `Ancestor`, thereby finding the node idenfitier of the common ancestor.

4 XIR Storage Structures

In this section, we present the storage structures used in our XIR method. XIR stores information needed for query processing at two levels – the schema level and the instance level. The schema-level information consists of the label paths occurring in the XML tree and the inverted index on these label paths; the instance-level information consists of all the node paths in the XML tree.

XIR uses two tables and an inverted index to store information about XML document structure:

```
LabelPath(pid, label_path)
NodePath(pid, docid, nodepath, value)
Inverted index on label_path of the table LabelPath.
```

4.1 Schema-level information

The table `LabelPath` represents the *schema-level information* and stores all the distinct label paths occurring in XML documents and their path identifiers (`pid`s). Figure 3 shows the `LabelPath` table and the inverted index for the example XML tree in Figure 1. The labels prefixed with '$' and '&' are added to denote the first label and the last label of each label path. The first label is to match the root label of the document, and the last label is to match the leaf label of a path expression.

pid	labelpath
1	$issue.issue.&issue
2	$issue.issue.editor.&editor
3	$issue.issue.editor.first.&first
4	$issue.issue.editor.last.&last
5	$issue.issue.articles.&articles
6	$issue.issue.articles.article.&article
7	$issue.issue.articles.article.@category.&@category
8	$issue.issue.articles.article.title.&title
9	$issue.issue.articles.article.author.&author
10	$issue.issue.articles.article.author.first.&first
11	$issue.issue.articles.article.author.last.&last
12	$issue.issue.articles.article.keyword.&keyword

(a) LabelPath table.

keyword	posting list
$issue	: <1, 1, {1}, 3> <2, 1, {1}, 4> <3, 1, {1}, 5> ...
issue	: <1, 1, {2}, 3> <2, 1, {2}, 4> <3, 1, {2}, 5> ...
&issue	: <1, 1, {3}, 3>
article	: <6, 1, {4}, 5> <7, 1, {4}, 6> <8, 1, {4}, 6> ...
&article	: <6, 1, {5}, 5>
articles	: <5, 1, {3}, 4> <6, 1, {3}, 5> <7, 1, {3}, 6> ...
&articles	: <5, 1, {4}, 4>
editor	: <2, 1, {3}, 4> <3, 1, {3}, 5> <4, 1, {3}, 5>
&editor	: <2, 1, {4}, 4>
author	: <9, 1, {5}, 6> <10, 1, {5}, 7> <11, 1, {5}, 7>
&author	: <9, 1, {6}, 6>
first	: <3, 1, {4}, 5> <10, 1, {6}, 7>
&first	: <10, 1, {7}, 7>
last	: <4, 1, {4}, 5> <11, 1, {6}, 7>
&last	: <11, 1, {7}, 7>
title	: <8, 1, {5}, 6>
&title	: <8, 1, {6}, 6>
keyword	: <12, 1, {5}, 6>
&keyword	: <12, 1, {6}, 6>
@category	: <7, 1, {5}, 6>
&@category	: <7, 1, {6}, 6>

(b) LabelPath Inverted Index.

Fig. 3. An example LabelPath table and inverted index.

The `LabelPath` inverted index is created on the `labelpath` field in the `LabelPath` table. Here, we consider label paths as text documents and labels in these label paths as keywords. Like the traditional inverted index [22], the `LabelPath` inverted index is made of the pairs of a keyword (i.e., a label) and a posting list. Each posting in a posting list has the following fields: `pid`, `occurrence_count`, `offsets`, `label_path_length`, where `pid` is the identifier of the label path in which the label occurs, `occurrence_count` is the number of occurrences of the label within the label path, `offsets` is the set of the positions of the label from the beginning of the label path, and `label_path_length` is the number of labels in the label path. For instance, in the posting of the label `section` in a label path `$chapter.chapter.sectio-n.section.section.paragraph.&paragraph`, the `occurrence_count` of `section` is 3, the `offsets` of `section` is $\{3, 4, 5\}$, and the `label_path_length` is 7.

4.2 Instance-level information

The table `NodePath` represents the *instance-level information* and stores the node paths to uniquely identify all the nodes in the XML documents. Figure 4 shows an example of the `NodePath` table for the XML tree in Figure 1.

The `NodePath` table stores all the node paths in the column `nodepath`. If the leaf node of a node path has a value, then the value is stored in the column `value`. The column `pid` stores label path identifiers, and is used for join with the `LabelPath` table to find all the node paths belonging to the same label path. The column `docid` stores the XML document identifiers.

pid	docid	nodepath	value
1	1	1	Null
2	1	1.2	Null
3	1	1.2.3	Michael
4	1	1.2.5	Flanklin
2	1	1.7	Null
3	1	1.7.8	Jane
4	1	1.7.10	Poe
5	1	1.12	Null
6	1	1.12.13	Null
7	1	1.12.13.14	R
8	1	1.12.13.16	XML schema
9	1	1.12.13.18	Null
10	1	1.12.13.18.19	David
11	1	1.12.13.18.21	Curry
12.	1.	1.12.13.23	XML
6	1	1.12.25	Null
...	...	...	...
12	1	1.12.25.35	DB

Fig. 4. An example NodePath table.

5 XIR Query Processing Algorithms

In this section we present the algorithms for evaluating LPEs and BPEs based on the XIR storage structures described in the previous section, and analytically compare XRel, XParent, and XIR with a focus on their performance-related features.

5.1 LPE evaluation algorithm

Figure 5 shows the algorithm for evaluating an LPE. In this algorithm, XIR first finds matching label paths in the `LabelPath` table using the `LabelPath` inverted index, and then, performs an equi-join between the set of the label paths found and the `Node-Path` table via the column `pid`. It then returns the matching node paths as the query result.

Formally, an LPE is evaluated as

$$\Pi_{nodepath}(\sigma_{MATCH(labelpath,LPE)} \; LabelPath \bowtie_{pid=pid} NodePath) \tag{1}$$

```
Algorithm XIR_LPE_evaluation
Input: LPE $P$, LabelPath inverted index, NodePath table
Output: set of node paths $NPset$ matching the LPE
begin
1. Convert the input LPE $P$ to an IR expression $E$ using the syntactic
   mapping rule LPE-to-IRExp (Rule 1).
2. Find the set of pids ($pidSet$) using the LabelPath inverted index
   given the IR expression $E$.
3. Find the set of node paths ($NPset$) through an equi-join between
   $pidSet$ and the NodePath table.
4. Return $NPset$.
end
```

Fig. 5. XIR LPE evaluation algorithm.

Since the selection $\sigma_{MATCH(labelpath,LPE)}LabelPath$ is implemented as a text search on the `labelpath` column, XIR should first convert an LPE to a keyword-based text search condition (we call it *information retrieval expression(IRExp)).* The following rule specifies how the conversion is done.

Rule 1 [*LPE-to-IRExp*] An LPE $o_1 l_1 o_2 l_2 \cdots o_p l_p$, where $o_i \in \{\text{'/'}, \text{'//'}\}$ for $i = 1, 2, \cdots,$-p, is mapped to an IRExp using the following rule:

$$o_1 l_1 \Rightarrow \begin{cases} l_1 & \text{if } o_1 = \text{'//'} \\ \$l_1 \text{ near(1) } l_1 & \text{if } o_1 = \text{'/'} \end{cases}$$

$$l_i o_{i+1} l_{i+1} \Rightarrow \begin{cases} l_i \text{ near}(\infty)\ l_{i+1} & \text{if } o_{i+1} = \text{'//'} \\ l_i \text{ near(1) } l_{i+1} & \text{if } o_{i+1} = \text{'/'} \end{cases}$$
$$\text{for } i = 1, 2, \cdots, p-1$$

$$l_p \Rightarrow l_p \text{ near(1) } \&l_p$$

where *near(w)* is the proximity operator, which retrieves the documents in which the two operand keywords appear within w words apart. □

Note that l_1 and l_p are respectively the root (i.e., first) node and the leaf (i.e., last) node of the linear query pattern representing the LPE. For example, an LPE `//article//author/last` is converted to an IRExp `article near(`∞`) author near(1) last near(1) &last`; an LPE `/issue/articles//author` is converted to an IRExp `$issue near(1) issue near(1) articles near(`∞`) author near(1) &author`. Note `$issue` indicates that `issue` is the root of the document.

5.2 BPE evaluation algorithm

Figure 6 shows the algorithm for evaluating a BPE. In this algorithm, XIR first decomposes a BPE into LPEs in the same way as XParent does, that is, one LPE from the root to each leaf node. It then evaluates each LPE to obtain a set of node path sets(*NPsets*).

This evaluation is done in the same manner as in Equation 1, with a slight modification to handle a branch predicate expression as

$$\begin{aligned} &\Pi_{nodepath}(\sigma_{MATCH(labelpath,LPE)} \\ &\quad LabelPath \bowtie_{pid=pid} \sigma_{value\,\theta\,v} NodePath) \end{aligned} \tag{2}$$

where "*value* θ v" is a selection condition on the leaf label of the branch predicate expression (see Definition 4) included in the LPE being evaluated.

Only one of the LPEs includes the result node (defined in Section 2.3). Let us call such an LPE the *result LPE* (P_0 in Figure 6). Then, XIR reduces the *result_NPset*

```
Algorithm XIR_BPE_evaluation
Input: BPE P, LabelPath inverted index, NodePath table
Output: set of node paths result_NPset matching the BPE
begin
   { Assume the BPE P has r leaf labels $l_0$, $l_1$, $l_2$, ..., $l_{r-1}$ and that $l_0$ is
     the result label. }
1.    Set $P_0$ as the LPE from the root to $l_0$.
2.    Obtain result_NPset by evaluating the LPE $P_0$.
3.    for each of the remaining leaf nodes $l_i$ (i=1,2, ..., r-1)
4.    begin
5.        Set $P_i$ as the LPE from the root to $l_i$.
6.        Obtain $NPset_i$ by evaluating the LPE $P_i$.
7.        Reduce result_NPset through a prefix match join
          with $NPset_i$.
8.    end.
9     return result_NPset.
end.
```

Fig. 6. XIR BPE evaluation algorithm.

obtained from the result LPE through *prefix match join* with each of the other LPEs $(P_1, P_2, \cdots, P_{r-1}$ in Figure 6). Definition 8 shows a formal definition of the prefix match join.

Definition 8. Given two relations having the schemas $R(A_1A_2 \cdots A_cB_1 \cdots B_m)$ and $S(A_1A_2 \cdots A_cC_1 \cdots C_k)$ that share the attributes $A_1A_2 \cdots A_c$, the *prefix match join* between R and S, denoted by $R \ \triangleright \ S$, is defined as

$$R \triangleright S = \sigma_{R.A_1=S.A_1 \ and \ \cdots \ and \ R.A_c=S.A_c}(R \times S) \qquad \square$$

In Definition 8, the relational schema refers to a label path and the relation instance refers to a node path set. According to this definition, given two LPEs, the prefix match join between the two NPsets obtained from them is performed as follows: (1) find the longest common prefix label subpath $l_1l_2 \cdots l_c (\equiv A_1A_2 \cdots A_c$ in Definition 8) matching both LPEs; (2) find the set of common prefix node subpaths, $\{n_1n_2 \cdots n_c\}$, belonging to the label subpath $l_1l_2 \cdots l_c$; and (3) for each node subpath $n_1n_2 \cdots n_c$ in the set, select all node paths that have the subpath in common.

Example 1. Consider the BPE `//article[/keyword="XML"]//author[/last="Curry"]/first`. The following three LPEs are generated: (1) `//article/keyword`, (2) `//article//author/last`, and (3) `//article//author/first`. Among these, LPE 3 is the result LPE. XIR retrieves the following three node path sets from these LPEs and the selection conditions on the leaf labels of LPE 1 and LPE 3: NPset$_1$ $\{$1.12.13.23$\}$ from LPE 1 and `keyword = "XML"`, NPset$_2$ $\{$1.12.13.18.21$\}$ from LPE 2 and `last="Curry"`, and NPset$_3$ $\{$1.12.13.18.19, 1.12.25.30.31$\}$ from LPE 3. Then, the prefix match join with NPset$_2$ reduces NPset$_3$ to $\{$1.12.13.18.19$\}$ based on the common prefix label subpath `/issue/articles/article/author`, and a further join with NPset$_1$ keeps NPset$_3$ to be $\{$1.12.13.18.19$\}$ based on the common prefix label subpath `/issue/articles/article`.

Figure 7 shows the SQL statement generated for this BPE. It implements the prefix match join of node sets shown in Algorithm XIR_BPE_evaluation by performing tuple-by-tuple prefix matches. The function `Prefix_matching` performs a prefix match between two node paths provided as the first two input arguments (e.g., `n1.nodepath` and `n2.nodepath`) by comparing only the prefix characters whose length is returned from the function `getCommonPrefixLength`. This function takes as inputs two LPEs and two label paths matching them and calculates the prefix length in the following steps: (1) identify the longest common prefix subexpression of the two input LPEs (e.g., `//article` common to `//article/keyword` and `//article/author-/last` in Figure 3), (2) for each of the two input label paths (e.g., `p1.labelpath` and `p2.labelpath`), count the number of prefix labels matching the common prefix subexpression (e.g., count 3 for a label path `$issue.`*`issue.articles.article-`*`.keyword.&keyword` in Figure 3, excluding `$issue`, which is an extra addition to the label path), and (3) if the two counts are equal then return `the count` and otherwise return -1. □

```
SELECT   DISTINCT n3.docid, n3.nodepath
FROM     LabelPath p1, LabelPath p2, LabelPath p3,
         NodePath n1, NodePath n2, NodePath n3
WHERE    p1.pid = n1.pid
AND      p2.pid = n2.pid
AND      p3.pid = n3.pid
AND      n1.value = ``XML''
AND      n2.value = ``Curry''
AND      MATCH(p1.labelpath, 'article' NEAR(1) 'keyword')
AND      MATCH(p2.labelpath, 'article' NEAR(MAXINT) 'author'
                              NEAR(1) 'last')
AND      MATCH(p3.labelpath, 'article' NEAR(MAXINT) 'author'
                              NEAR(1) 'first' NEAR(1) '&first')
AND      Prefix_matching (n1.nodepath, n2.nodepath,
         getCommonPrefixLength(p1.labelpath, p2.labelpath,
             ``//article/keyword'', ``//article//author/last''))
AND      Prefix_matching (n2.nodepath, n3.nodepath,
         getCommonPrefixLength(p2.labelpath, p3.labelpath,
             ``//article//author/last'', ``//article//author/first''));
```

Fig. 7. XIR SQL statement for the BPE in Example 1.

The query processing algorithms of XRel, XParent, and XIR share the same outline, but have some different implementations leading to their performance differences. In the case of LPEs, XIR's performance advantage over both XRel and XParent comes from using inverted index search instead of string match for finding label paths matching the LPE. In the case of BPEs, XIR has the performance advantage over XRel in that it generates a smaller number of LPEs for the same BPE. Another major performance advantage of XIR for BPEs over both XRel and XParent comes from the number of joins performed and the cardinalities of node sets joined to determine the node (or node path) set returned as the query result. XIR requires a far less number of joins compared with XRel and XParent. Besides, the cardinalities of node sets joined in XIR or XRel are smaller than those in XParent. Details of the analysis can be found in the reference [25].

6 Performance Evaluation

We compare the query processing performance of XIR with those of XRel and XParent. The results show that XIR is far more efficient than both XRel and XParent.

6.1 Experimental setup

Databases We have collected 10008 real-world XML documents from the Internet using two web crawlers: Teleport Pro Version 1.29.1959 [28] and ReGet Deluxe 3.3 Beta (build 173) [27]. For crawling XML documents, we first start with base URLs, and then, crawl all XML documents reachable from the base URLs. The base URLs include web sites of major universities, companies, and publishers in several countries. Note that about 91% of the XML documents are 4 Kbytes or less.

Using the collected XML documents, we have constructed five sets of data files of different sizes. Each set contains approximately 5000, 10000, 20000, 40000, and 80000 distinct label paths. The last set has 1460000 node paths. A larger set contains all label paths in a smaller set, i.e., is a superset of smaller sets. Documents in each set are then parsed, and the parsed results are loaded into three databases, each containing tables used by XRel, XParent, and XIR methods. The total number of databases thus generated is fifteen.

For XRel and XParent, we have used the database schema and indexes as they were used in the original designs [14, 24]. For XIR, we have loaded the data files into the `LabelPath` and `NodePath` tables, created B+-tree indexes on the columns `pid`, `docid` of each table as in XRel or XParent, and created an inverted index on the `labelpath` column of the `Labelpath` table.

The resulting database size for XIR is 454 Mbytes for 79943 distinct label paths and is 10% - 29% smaller than those of XRel or XParent. Details of the analysis can be found in the reference [25].

Queries Table 1 shows three groups of tree pattern queries: a group of LPEs, a group of BPEs whose branch predicate expressions do not contain selection conditions, and a group of BPEs whose branch predicate expressions do contain selection conditions. Each group has two sets of queries: one is on `issue` documents; the other on `movie` documents. The former has far more document instances than the latter.

Table 1. Queries.

Group	Label	Tree Pattern Query
Group 1 (LPEs)	LPE1	//issue//author/first
	LPE2	//issue//article//author/first
	LPE3	//movie//actor//first
	LPE4	//movie/cast//actor//first
Group 2 (BPEs without selection conditions)	BPE1	//issue[//keyword]//author[/last]/first
	BPE2	//issue[//keyword]//article[/summary]//author[/last]/first
	BPE3	//movie[/director]//actor[/award]//first
	BPE4	//movie[/director]/cast[/actress]//actor[/award]//first
Group 3 (BPEs with selection conditions)	BPS1	//issue[//keyword=``XML'']//article[/summary]//author[/last]/first
	BPS2	//issue[//keyword=``XML'']//article[/summary]//author[/last=``Smith'']/first
	BPS3	//movie[/director/last=``Mendes'']//actor[/award]//first
	BPS4	//movie[/director/last=``Mendes'']//actor[/award=``Oscar'']//first

Computing environment We have conducted the experiments using the Odysseus object-relational database management system[6] on SUN Ultra 60 workstation with 512 Mbyte RAM. In order to eliminate the unpredictable buffering effect in the operating system, we have used a raw disk device to bypass the OS buffer. We have also flushed the DBMS buffer after each query execution so that the execution does not affect later ones. The cost metrics used are the elapsed time and the number of disk I/O's.

6.2 Experimental results

Since the crawlers collect *arbitrary* documents from the Internet, new label paths are added as new documents are added by crawling. We have extracted the number of distinct label paths from the XML documents collected. We crawled a total of 10009 XML documents extracting 79943 distinct label paths.

Figure 8 shows the costs of the query LPE4 in Table 1 for the three methods as the number of distinct label paths increases. Figures 9 and 10 show those for BPE2 and BPS1 in Table 1. The buffer size has been set to 200 4Kbyte-pages to eliminate extra disk I/O's caused by an insufficient buffer size. Due to space limit, we omit the figures of the other queries; their costs show similar trends with typical curves.

[6] Odysseus has been developed at the KAIST Advanced Information Technology Research Center, and provides the key operations needed by a text search engine.

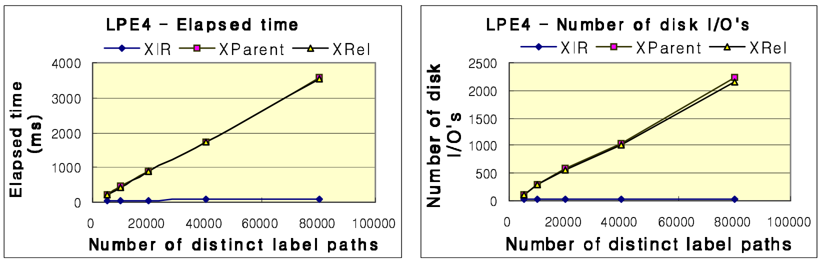

Fig. 8. Query costs of XRel, XParent, XIR for LPE4 (buffer size = 200 pages).

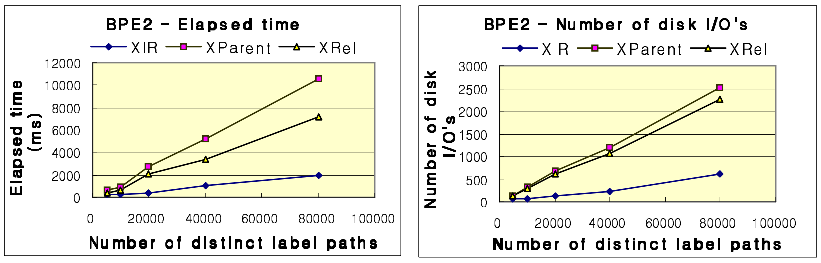

Fig. 9. Query costs of XRel, XParent, XIR for BPE2 (buffer size = 200 pages).

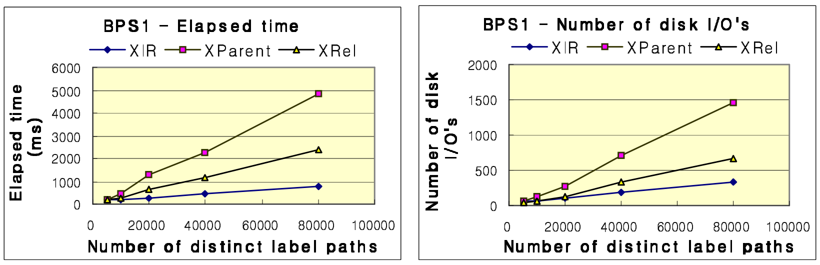

Fig. 10. Query costs of XRel, XParent, XIR for BPS1 (buffer size = 200 pages).

In Figures 8 through 10, we see that XIR is more efficient than both XRel and Xparent[7]. The performance gap varies from several orders of magnitude in the case of LPEs to several factors in the case of BPEs. In particular, the costs of LPEs increase nearly linearly for XRel and XParent while sublinearly – nearly constant – for XIR. This amounts to the difference between the string match and inverted index search for finding matching label paths. In the case of BPEs, the costs increase linearly for all three methods, but the slope is the smallest for XIR. This comes from XIR's join performance advantage [25]. When comparing the BPEs without selection conditions (Figure 9) and those with selection conditions (Figure 10), we see that the gaps among the costs of the three methods are smaller for BPEs *with* selection conditions. The reason for this is that XRel or XParent can take advantage of the B+-tree index created on the `'value'` column of the table `Text` or `Data`.

[7] As mentioned in Section 3.2, we use the table `Ancestor` to be able to support partial match queries in XParent [14], [15]. This causes XParent to show poorer performance than XRel due to the cardinality of the `Ancestor` table that is heavily involved in joins. This is in contrast with the results shown in the XParent papers [14], [15], where the performances of only full match queries using the DataPath table were presented.

7 Conclusions

We have proposed a novel approach called XIR to processing partial match queries on a large number of heterogeneous XML documents typical in the Internet environment. For this purpose, we have presented two key techniques. In the first technique, we treat the label paths occurring in XML documents as texts and create an inverted index on them. This inverted index supports much faster partial match than XRel's or XParent's string match when evaluating a linear path expression. In the second technique, we use prefix match joins to evaluate a branching path expression. A branching path expression is decomposed into linear path expressions, and the results of evaluating each linear path expression are combined using the prefix join. Using the prefix join significantly reduces the number of joins compared with the containment join used in XRel or XParent.

Through extensive experiments, we have compared the performance of XIR with those of XRel and XParent using real XML documents crawled from the Internet. The results show that XIR is significantly more efficient than XRel or XParent.

References

1. A. Aboulnaga, A. R. Alameldeen, and J. Naughton, "Estimating the Selectivity of XML Path Expressions for Internet Scale Applications," In *Proc. the 27th Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 591-600, Rome, Italy, Sept. 11-14, 2001.
2. S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, D. Srivastava, "Minimization of Tree Pattern Queries," In *Proc. 2001 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 497-508, Santa Barbara, California, May 21-24, 2001.
3. M. Altinel, M. J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," In *Proc. the 26th Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 53-64, Cairo, Egypt, Sept. 10-14, 2000.
4. S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," In *Proc. the 18th Int'l Conf. on Data Engineering (ICDE)*, pp. 141-152, San Jose, California, Feb. 26 - Mar. 1, 2002.
5. Jan-Marco Bremer and Michael Gertz, "XQuery/IR: Integrating XML Document and Data Retrieval," In *Proc. the Fifth Int'l Workshop on the Web and Databases (WebDB 2002)*, pp. 1-6, Madison, Wisconsin, 2002.
6. N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," In *Proc. 2002 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 310-321, Madison, Wisconsin, June 3-6, 2002.
7. J. Clark and S. DeRose, XML Path Language (XPath), W3C Recommendation, *http://www.w3.org/TR/xpath*, Nov. 1999.
8. B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," In *Proc. the 27th Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 341-350, Rome, Italy, Sept. 11-14, 2001.
9. Daniela Florescu, Donald Kossmann, and Ioana Manolescu,"Integrating Keyword Search into XML Query Processing ," In *Proc. the 9th WWW Conference/Computer Networks*, pp. 119-135, Amsterdam, NL, May 2000.
10. Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents," In *Proc. 2003 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 16-27, San Diego, California, June 9-12, 2003.
11. R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," In *Proc. the 23th Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 436-445, Athens, Greece, Aug. 26-29, 1997.

12. A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt, “Mixed Mode XML Query Processing,” In *Proc. the 29th Int’l Conf. on Very Large Data Bases (VLDB)*, pp. 225-236, Berlin, Germany, Sept. 9-12, 2003.
13. Z. Ives, A. Levy, and D. Weld, Efficient Evaluation of Regular Path Expressions on Streaming XML Data, Technical Report UW-CSE-2000-05-02, University of Washington, 2000.
14. H. Jiang, H. Lu, W. Wang, and J. Xu Yu, “Path Materialization Revisited: An Efficient Storage Model for XML Data,” In *Proc. the 13th Australasian Database Conference (ADC)*, pp. 85-94, Melbourne, Australia, Jan. 28 - Feb. 1, 2002.
15. H. Jiang, H. Lu, W. Wang, and J. Xu Yu, “XParent: An Efficient RDBMS-Based XML Database System,” In *Proc. the 18th Int’l Conf. on Data Engineering (ICDE)*, pp. 335-336, San Jose, California, Feb. 26 - Mar. 1, 2002.
16. H. Jiang, W. Wang, H. Lu, and J. X. Yu, “Holistic Twig Joins on Indexed XML Documents,” In *Proc. the 29th Int’l Conf. on Very Large Data Bases (VLDB)*, pp. 273-284, Berlin, Germany, Sept. 9-12, 2003.
17. Q. Li and B. Moon, “Indexing and Querying XML Data for Regular Path Expressions,” In *Proc. the 27th Int’l Conf. on Very Large Data Bases (VLDB)*, pp. 361-370, Rome, Italy, Sept. 11-14, 2001.
18. F. Mandreoli, R. Martoglia, P. Tiberio, “Searching Similar (Sub)Sentences for Example-Based Machine Translation,” In *Proc. 2002 Italian Symposium on Sistemi Evoluti per Basi di Dati (SEBD’02)*, Isola d’Elba, Italy, June 2002.
19. J. McHugh, J. Widom, “Query Optimization for XML,” In *Proc. the 25th Int’l Conf. on Very Large Data Bases (VLDB)*, pp. 315-326, Edinburgh, Scotland, UK, Sept. 7-10, 1999.
20. J. Naughton et al., “The Niagara Internet Query System,” *IEEE Data Engineering Bulletin*, pp. 27-33, Vol. 24, No. 2, June, 2001.
21. N. Polyzotis and M. Garofalakis, “Statistical Synopses for Graph-structured XML Databases,” In *Proc. 2002 ACM SIGMOD Int’l Conf. on Management of Data*, pp. 358-369, Madison, Wisconsin, June 3-6, 2002.
22. G. Salton and M. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York 1983.
23. I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang, “Storing and Querying Ordered XML Using a Relational Database System,” In *Proc. 2002 ACM SIGMOD Int’l Conf. on Management of Data*, pp. 204-215, Madison, Wisconsin, June 3-6, 2002.
24. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, “XRel: A Path-based Approach to Storage and Retrieval of XML Documents using Relational Databases,” *ACM Trans. on Internet Technology(TOIT)*, pp. 110-141, Vol. 1, No. 1, 2001.
25. Y. Park, K. -Y. Whang, B. Lee, W. Han, “Efficient Evaluation of Partial Match Queries for XML Documents Using Information Retrieval Techniques,” Technical Report CS-TR-2004-212, Department of Computer Science, KAIST, Dec., 2004. Also, available on AITrc Technical Report No. 04-11-048, *http://aitrc.kaist.ac.kr/util_tr.htm*, Dec. 28, 2004.
26. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohmann, “On Supporting Containment Queries in Relational Database Management Systems,” In *Proc. 2001 ACM SIGMOD Int’l Conf. on Management of Data*, pp. 425-436, Santa Barbara, California, May 21-24, 2001.
27. ReGet Deluxe 3.3 Beta (build 173), *http://deluxe.reget.com/en/*.
28. Teleport Pro Version 1.29, *http://www.tenmax.com/teleport/pro/home.htm*.
29. Xyleme, *http://www.xyleme.com*.
30. eXtensible Markup Language(XML), *http://www.w3.org/XML/*.