Mar. 2004, Vol.19, No.2, pp.224–238 J. Comput. Sci. & Technol.

RPE Query Processing and Optimization Techniques for XML Databases

Guo-Ren Wang, Bing Sun, Jian-Hua Lv, and Ge Yu

Department of Computer Science, Northeastern University, Shenyang 110004, P.R. China

E-mail: wanggr@mail.neu.edu.cn

Received July 26, 2002; revised March 26, 2003.

Abstract An *extent join* to compute path expressions containing parent-children and ancestor-descendent operations and two path expression optimization rules, *path-shortening* and *path-complementing*, are presented in this paper. *Path-shortening* reduces the number of joins by shortening the path while *path-complementing* optimizes the path execution by using an equivalent complementary path expression to compute the original one. Experimental results show that the algorithms proposed are more efficient than traditional algorithms.

Keywords XML, regular path expressions, query processing and optimization

1 Introduction

As an emerging standard for data representation and exchange on the Web, XML is adopted by more and more applications for their information description. Even though XML is usually used as an information exchange standard, storing, indexing and querying XML data are still important issues and have become research hotspots both in the academic community and in the industrial community.

So far, there have been three main approaches to manage XML data, the relational, object-oriented and native ways. In the relational way[1,2], XML data are mapped into tables and queries on XML data are translated into SQL statements. STORED[3], Agora[4], Monet[5] and VXMLR[6] are examples following this way. Similar to the relational way, the object-oriented way[7–9] stores XML documents into an OODB with classes and translates XML queries into OQL queries based on XML data schema information. The Lore Project[10] is an example following this way. In the third way, special structures and indexes are designed to store and index XML data and particular query optimization techniques are proposed to query XML data from databases. Rufus[11], Strudel[12], and Tamino[13] are examples following this way.

To improve basic operations on databases such as "/" and "//", indexing is a necessary means and draws more and more attention. *DataGuides*[14] is designed for semi-structured data, and four index structures, *value index*, *label index*, *edge index* and *path index*, are proposed. However, they act as schema more than as indexes. A general index structure for semi-structured data, called *template index* or *T-index*, is proposed in [15] to make evaluating queries involved in several path expressions possible, and the degeneration form of *T-index*, i.e., 1-*index*, is similar to *DataGuides*. These indexing techniques are designed for semistructured data. In [16], three index structures, *element index*, *attribute index* and *structure index*, are proposed to support three essential functionalities respectively. For a given element name, a list of element with the same element name can be found with *element index*. Similarly, for a given attribute name, a list of attributes with the same attribute name can be found with *attribute index*. *Structure index* is used to find the parent element and child elements of a given element. However, there are two main drawbacks. 1) Although the ancestor-descendant relationship between elements and/or attribute in a hierarchy of XML data can be quickly determined based on the proposed numbering scheme, it is difficult to distinguish the parent-child relationship from the general ancestor-descendant relationship and therefore it is difficult to build *structure index*. This is not made clearly in the paper. In this case, it is incorrect to compute a join operation standing for a parent-child relationship using the EE-join algorithm. The ancestor-descendant relationship can be derived from the parent-child relationship, but not vice versa. 2) There is the same problem with

*Regular Paper

This research is partially supported by the National Natural Science Foundation of China (Grant No.60273079) and the Teaching and Research Award Program for Outstanding Young Teachers in High Education Institution of the Ministry of Education, China.

the EA-join algorithm.

To retrieve XML data from databases, many query languages have been proposed so far. Examples are Quilt[17], XQuery[18], XML-QL[19], XQL[20], XPath[21], and Lorel[22]. Because one of common features of these languages is the use of regular path expressions (RPE), query rewriting and optimization for RPE are becoming a research hotspot and some research results have been reported recently. A usual way to optimize the execution of RPE expressions is rewriting an RPE query with simple path expressions (SPE) based on schema information and statistics about XML data, and these SPE queries are then translated into SQL queries, for example, in the relational way. In the Lore system, three basic query processing strategies are proposed for the execution of path expressions, *top-down*, *bottom-up* and *hybrid*. The *top-down* way navigates the document tree from the root to the leaf nodes while the *bottom-up* way does from the leaf nodes to the root. In the *hybrid* way, a longer path is first broken into several sub-paths, each of which is performed with either *top-down* or *bottom-up*. The results of the sub-paths are then joined together. In the VXMLR system[6], regular path expressions containing "//" and/or "*" operators are rewritten with simple path queries based on schema information and statistics. [16] presents an *EE-Join* algorithm to compute "//" operator and a *KC-Join* algorithm to compute "*" operator based on their numbering scheme.

In this paper, we propose an *extent join* algorithm to compute path expressions containing parent-children and ancestor-descendent operations between path steps. To support the *extent join* approach, some indexes preserving parent-children and ancestor-descendent relationships are also proposed. Furthermore, two path expression optimization rules are proposed, *path-shortening* and *path-complementing*. *Path-shortening* reduces the number of joins by shortening the path while *path-complementing* optimizes the execution of a path by using an equivalently complementary path to compute the original path. The performances of the query processing and optimization techniques proposed in this paper are evaluated with two benchmarks, *XMark* and *XMach*, and two real data sets, *Shakes* and *DBLP*.

The remainder of this paper is organized as follows. Section 2 presents some basic concepts for XML query processing, including XML data tree, XML schema graph and path expression. Section 3 describes the *extent join* algorithm along with indexes and rewriting algorithm for "//". Section 4 presents two query optimization rules for regular path expressions. Section 5 gives the experimental results and the performance evaluation. Finally, Section 6 concludes this paper.

2 Basic Concepts

In this section we review some concepts and definitions used throughout the paper, including XML data tree, XML schema graph and path expression.

XML data are represented as an XML data tree $T_d = (V_d, E_d, \delta_d, \Sigma_d, root_d, oid)$, where V_d is the node set including element nodes and attribute nodes; E_d is the set of tree edges denoting parent-children relationships between two elements and element-attribute relationships between elements and attributes; δ_d is the mapping function from nodes to nodes that are actually the relationship constraints. Each node has a unique name that is a string-literal of Σ_d and a unique identifier in set *oid*. Finally, each XML data tree has a root element $root_d$ that is included in V_d.

Fig.1 shows part of an XML document proposed in the XML Benchmark project[23], which is represented as an XML data tree. There are two kinds of nodes, *elements* denoted by *ellipses* and *attributes* by *triangles*. The numeric identifiers following "&" in nodes represent *oids*. The solid edges are tree edges connecting nodes via the δ_d function. In this model, the parents can actually be reached via the δ_d^{-1} function from the children. Node "&1" labelled "*site*" is the $root_d$ of this XML data tree and all other nodes can and only can be reached by $root_d$. Note that in Fig.1, there are two directed dashed lines between some nodes (&23 and &18, &28 and &18), representing the referencing-referenced relationship between elements.

An XML schema graph is defined as a directed, node-labelled graph $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$, where V_t is the node set including element type nodes; E_t is the set of graph edges denoting element-subelement relationships. Attributes are parts of elements; δ_t is the mapping function from nodes to nodes that actually determines which element can contain which sub-elements. Each node has a unique name that is a string-literal of Σ_t and this name is actually element type name. Finally, an XML schema graph has a root element $root_t$ that is included in V_t, which is defined as the node with only outgoing edges and without any incoming edges.

Fig.2 shows part of the XML schema graph that determines the XML data tree in Fig.1. The nodes

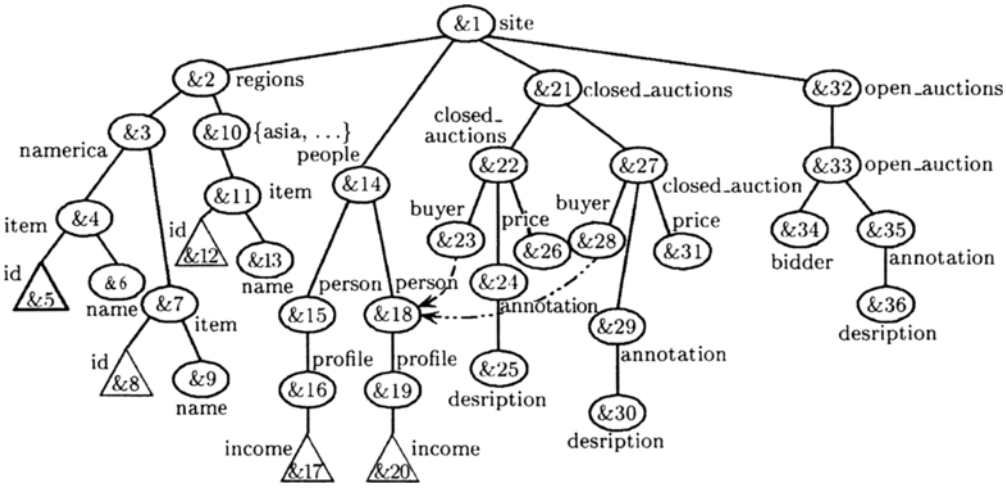

Fig.1. A sample XML data tree.

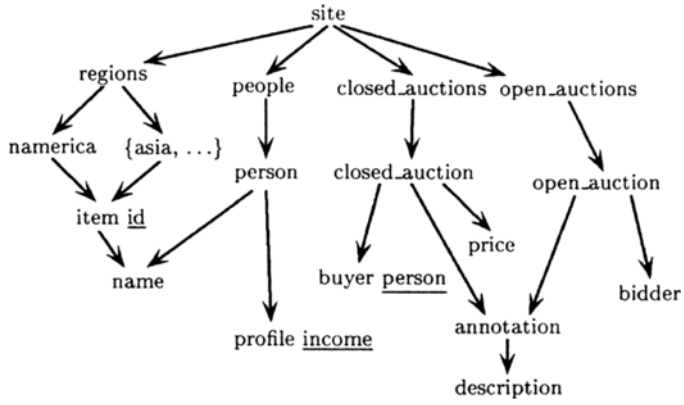

Fig.2. XML schema graph of the sample XML data tree.

are element types and the solid edges are graph edges connecting nodes via the δ_t function. In this model, the parent elements can actually be reached via the δ_t^{-1} function from the children elements and the corresponding reverse edges are omitted in Fig.2. The node labelled "*site*" is the $root_t$ of this XML schema graph. The attributes of element types are listed besides the nodes with underline, for example, *income*.

```
PathExpression ::= CONNECTOR PathSteps
                 | PathSteps CONNECTOR PathSteps
PathSteps ::= Name | Name '|' PathSteps
            | (PathSteps) | '*'
CONNECTOR ::= '/' | '//'
```

Fig.3. BNF syntax of path expression.

Path expressions can be straightforwardly defined as a sequence of element type names ("*site*", "*people*", etc.) connected by some connectors such as "/", "//" and wildcard "*". For example, path expression "*/site//item*" can be used to find all items of the database whose root element $root_d$ is "*site*". The syntax definition of path expression is shown in Fig.3.

A path expression mainly consists of two parts, *path steps* and *connectors*. Each path expression must begin from the root, that is, it begins with a connector "/" or "//". There are two basic kinds of path steps, *Name* and *wildcard* "*". Path step *Name* means that in this step only the element instances with type *Name*(tag name) will be matched and "*" will match all element instances no matter which type they belong to. Between two path steps there must be a connector to specify the relationship between them. Connector "/" appearing in the beginning of a path expression means that the path expression begins from exactly the root and the following path step is the root element type, while connector "//" appearing in the beginning of a path expression means the path expression be-

gins from the root and the following path step is the descendant of the root, that is, "//" covers sub-path-expressions with any length. A connector appearing between two path steps specifies the relationship between them. In this case, connector "/" constrains that between the two path steps there must exist a parent-children relationship while "//" is an ancestor-descendant relationship constraint.

3 Extent Join

In this section, we present the *XML element extent* concept and the *extent join* algorithm. We present some indexes as well in this section to support the concept and the algorithm.

3.1 XML Element Extent

Given an XML data tree $T_d = (V_d, E_d, \delta_d, \Sigma_d, root_d, oid)$ and a corresponding XML schema graph $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$, we have the following definitions.

Definition 1. *pcpair(pid, cid) is a pair of oids, in which pid and cid ∈ oid, and pid is the parent of cid, for example, pcpair(&1, &2).*

Definition 2. *adpair(aid, did) is a pair of oids, in which aid and did ∈ oid, and aid is the ancestor of did, for example, adpair(&1, &3).*

Note that a *pcpair* is a special case of an *adpair*. Additionally, ε is defined to act as any element instance, so $adpair(\varepsilon, \&3)$ can be used to represent $adpair(\&1, \&3)$ or $adpair(\&2, \&3)$. Both *adpair* and *pcpair* can also act as logic operators. For example, if there exists an ancestor-descendent relationship between two element instances e_1 and e_2, then $adpair(e_1, e_2)$ is true. Otherwise, it is false.

Definition 3. *The set of all pcpairs of a given tag name Tag, called parent-child element extent, is represented by Ext(any, Tag) = {pcpair(pid, cid)| cid is an instance of Tag ∧ pcpair(pid, cid) is true}. Similarly, the set of all adpairs of two given tag names an and dn, called ancestor-descendant element extent, is represented by Ext(an,dn) = {adpair(aid, did) | pcpair(ε, aid) ∈ Ext(any, an)∧ pcpair(ε, did) ∈ Ext(any, dn) ∧ adpair(aid, did) is true}.*

For examples, $Ext(any, name) = \{(\&4, \&6), (\&7, \&9), (\&11, \&13)\}$ and $Ext(site, annotation) = \{(\&1, \&24), (\&1, \&29), (\&1, \&35)\}$.

Definition 4. *For two given elements, an and dn, and a given path P, the path constrained element extent is defined as PCExt(an, dn, P) = {adpair(aid, did) | adpair(aid, did) ∈ Ext(an, dn) ∧ did ∈ P(aid)}, where P(aid) is the element instance set that can be reached from aid via path P.*

In the query processing, *PCExt* may be more useful than the basic XML element extents. A *PCExt* is actually an element extent with a path constraint and is a subset of the corresponding extent. For example, in *PCExt(site, annotation, "/site/closed_auctions/closed_auction/annotation")*, the third parameter is the path constraint on *Ext(site, annotation)*. This constraint regulates that in this *PCExt*, the instances of element *annotation* must be the ones that can be reached from the corresponding instances of element *site* via the path expression. As a result, *PCExt(site, annotation, "/site/closed_auctions/closed_auction/annotation")* = {(&1, &24), (&1, &29)}.

3.2 Indexes

Neither the DOM interface nor the XML data tree provides the extent semantic for XML data, so we propose three structural indexes to support it: *ancestor-descendant index* (*ADX*), *parent-children index* (*PCX*) and *path index* (*PX*). We also propose *reference index* (*RX*) to support operations on references.

ADX is used to index *Ext(Pname, Cname)* for given element names *Pname* and *Cname* where *Pname* is the ancestor of *Cname*. Actually, *ADX* indexes the ancestor-descendant relationship between specified elements. For example, *ADX(site, item)* = {(&1, &4), (&1, &7), (&1, &11)}. *PCX* is used to index *PCExt(Pname, Cname, "Pname/Cname")* for given element names *Pname* and *Cname* where *Pname* must be the parent of *Cname*. For example, *PCX(namerica, item)* is {(&3, &4), (&3, &7)}. If the parent element name is not specified, *PCX(any, item)* indexes *Ext(any,item)* = {(&3, &4), (&3, &7) ,(&10, &11)}, written as *PCX(item)*. $PX(E_1/P/E_2)$ is used to index $PCExt(E_1, E_2, "E_1/P/E_2")$. For example, *PX(closed_auctions/closed_auction/buyer)* = {(&21, &23), (&21, &28)}. *RX* is used to support the reference semantics between XML elements. For example, *RX(buyer, person, person)* is {(&23, &18), (&28, &18)}.

About the indexes above, only the principles are introduced. The implementations of them are relatively simple for they have no special demands on the index structures. The traditional index structures, e.g., B+ tree, are suitable for these indexes.

3.3 Extent Join Algorithm

The basic idea of the *extent join* algorithm is replacing the tree traversal procedures with join operations. Before the whole path expression is evaluated, the intermediate result sets to be joined must be first computed. And the ancestor-descendant/parent-children relationship based multi-join operation is then performed to evaluate the whole path expression. The most special characteristic of these indexes is that they maintain the parent-children and ancestor-descendant relationship by the index results.

Consider path expression "*/site//closed_auction/annotation/description*" containing four path steps and three connectors. As shown in Fig.4, each path step corresponds to an intermediate result set, i.e., an element extent; each connector is transformed into a join operation, and the results of joins are the path constrained element extents. For example, the join between *Ext(any, site)* and *Ext(site, closed_auction)* is *PCExt(site, closed_auction, "/site/ closed_auctions/closed_auction")*, and the *PCExt* acts as an intermediate result used to perform another join with *Ext(closed_auction, annotation)* to get another *PCExt*. Path expressions must be transformed into evaluation plans to get evaluated. The art of transformation is focused on the path steps to correspond to extents, and the following shows the full transformation rules.

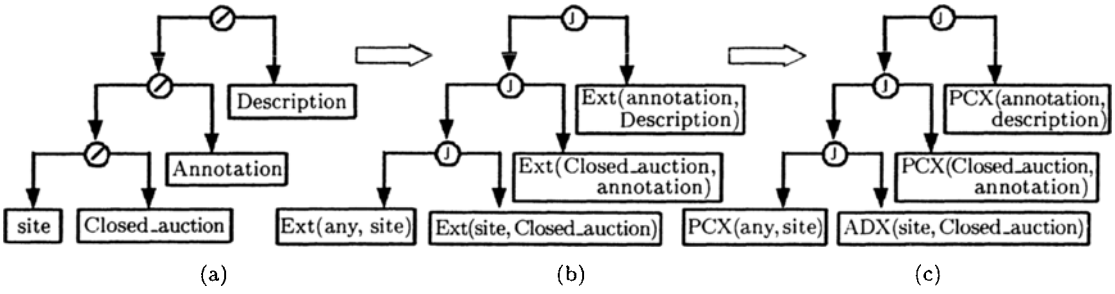

Fig.4. (a) Path expression. (b) Extent join tree. (c) Execution tree.

(1) Connectors ("/" and "//") are transformed into joins between two sets.

(2) Path step "*" is rewritten with element types using the mapping function δ_t. For example, $\delta_t(site)$ = {*regions, people, closed_auctions, open_auctions*} and path expression "*/site/*/person*" is rewritten as "*/site/(regions | people | closed_auctions | open_auctions)/person*".

(3) Path steps following connector "//" are transformed into a corresponding *ADX* operator. For example, path step S_2 in "$S_1//S_2$" is transformed into $ADX(S_1, S_2)$.

(4) Path steps following connector "/" are transformed into a corresponding *PCX* operator. For example, path step S_2 in "S_1/S_2" is transformed into $PCX(S_1, S_2)$.

(5) Path steps containing "|"s are transformed into the unions of corresponding indexes. For example, path step $(S_2|S_3)$ in "$S_1/(S_2|S_3)$" is transformed into $PCX(S_1, S_2) \cup PCX(S_1, S_3)$.

The third transformation rule transforms the "//" connectors into *ADXs*. However, to build this index for every element type pair with ancestor-descendent relationship will spend too much time and space overhead. So in the case of no corresponding *ADX* available, the "//" connectors must be rewritten into path expressions connected only with "/". This procedure should be achieved with the knowledge of the schema information, e.g., DTD of XML documents. For the XML schema graph is a directed graph, the rewriting algorithm is actually to find all possible paths between two nodes in a graph. Before introducing the details of the algorithm for rewriting "//" connector, we first define an important data structure *reverse path tree* (*RPT*) as follows.

Definition 5. *A reverse path tree is defined as a node-labelled tree* $T_r = (V_r, E_r, \Sigma_r, root_r)$, *which organizes several path expressions with a common end path step together, where* V_r *is the node set that is actually the set of corresponding path steps; the edges contained in the edge set* E_r *are connector "/";* Σ_r *is the same as* Σ_t *in* G_t; *and* $root_r$ *is the root of this tree and is just the common end path step. We define* $RPT(E)$ *as a reverse path tree rooted at E which contains all path expressions from* $root_r$ *to E, and define* $RPT(E_1, E_2)$ *as a reverse path tree rooted at* E_2 *which contains all path expressions from* E_1 *to* E_2 *and some path expressions from* $root_r$ *to* E_2.

For example, in Fig.5, (a) shows *RPT* (*description*) and (b) shows *RPT*(*closed_auctions, de-*

scription). We can easily retrieve path expressions with specified starting path step by traversing up through the *RPT* from the tree leaves with the given label. For example, if we only want path expressions beginning with *closed_auctions* from *RPT(closed_auctions, description)*, we can just traverse up from the most left leaf node of Fig.5(b) to the root. So, the connector rewriting algorithm is just the *RPT* constructing algorithm, as shown in Algorithm 1. The 5th step of the algorithm checks the number of occurrences of the same element type on the current path. It ensures that the algorithm can stop normally at last even if there are cyclic ancestor-descendant relationships between elements. Restricting the maximal number of occurrences of an element type on a path to 2 can not only avoid error, but also ensure that *RPT* contains all the ancestor-descendant relationships between elements. With the proposed transformation rules and the algorithm, we have the *extent join* algorithm. The details are shown in Algorithm 2.

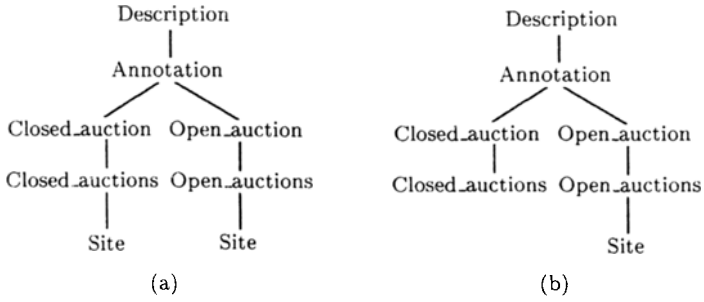

Fig.5. Reverse path trees. (a) RPT (description). (b) RPT (closed_auction, description).

Algorithm 1. Constructing Reverse Path Tree Algorithm (rewriting "$n_1//n_2$")
Input: XML schema $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$, XML element type n_1, n_2
Output: *RPT rpt*
(01) $rpt.root_r = n_2$;
(02) $currentnode = rpt.root_r$;
(03) $currentnode.children = \delta_t^{-1}(currentnode)$;
(04) **for** each node $cc \in \delta_t^{-1}(currentnode)$ **do**
(05) **if** the *cc* number in the current path does not exceed 1 **then**
(06) **if** $cc \neq n_1$ and $cc \neq root_t$ **then**
(07) $currentnode = cc$;
(08) **goto** (03);
(09) **endif**
(10) **endif**
(11) **endfor**

Algorithm 2. Extent Join Algorithm
Input: Path expression query *P*
Output: Result set *R*
(1) Check the *ADX* and rewrite the no-index-supported "//" connectors using Algorithm 1.
(2) Transform the rewritten path *P* into joins and indexes and organize as a simple query plan.
(3) Execute the query plan including indexes and joins.

4 Optimizing Regular Path Expressions

In Section 3, we have introduced the basic idea of *extent join* that uses joins over sets to evaluate path expression queries. Its performance depends largely on the number of joins and the size of joining sets. In this section, we propose two path expression optimization techniques to reduce the number of joins and the execution cost of path expressions in evaluating a path expression. Meanwhile, the general cost based optimization procedure is also introduced in this section.

4.1 Path Shortening Strategy

Most studies on path expression queries focus on computing a path in different ways, designing special indices to support special queries, or rewriting some specific operators to improve performance. However, some optimizing operations can be done on some path expressions based on XML schema information. Since an XML document is represented as a tree, we have the following theorem.

Theorem 1. *Let G_t be an XML schema and T_d an XML document complying with schema G_t. Then each node except $root_d$ is a descendant of $root_d$, i.e., $(\forall n)(n \in \{V_d - root_t\} \rightarrow n$ is a descendant of $root_d)$.*

Proof. Since T_d is a tree and $root_d$ is the root of the tree, Theorem 1 is evident. □

According to Theorem 1, path operations like $//EType$ and $/root_d//Etype$ can be translated to extent operations, i.e., $EXT(Etype)$, which are used to get a certain type of elements from the database.

Theorem 2. *If an absolute path expression query $Q = \langle G_t, T_d, root_t, PE, RS\rangle$ is a unique path in G_t that can access the final result, then $RS = EXT(End(PE))$, where $End(PE)$ represents the type of the last element of the path.*

Proof. Firstly, $RS \subseteq EXT(End(PE))$ always holds, for each XML instance in RS is of type $End(PE)$. Assume that there is an XML instance $e \in EXT(End(PE)) \wedge e \notin RS$, then there must be an access path from $root_d$ to e according to Theorem 1. It contradicts the fact that Q is a unique path. So $EXT(End(PE)) \subseteq RS$. Thus, $RS = EXT(End(PE))$. □

Obviously, we can optimize a query according to Theorem 2, if the query is the *unique access path* of the type. We can get the definition of *unique access path* according to Theorem 2.

Definition 6. *Let $Q = \langle G_t, T_d, root_t, PE, RS\rangle$ be a path expression query. If $End(root_t + PE) = E_n \wedge (\forall e)(e \in EXT(E_n) \rightarrow e \in RS)$, then PE is the unique access path of element type E_n, in short $UAP(E_n) = PE$.*

Unique access path is defined as that it must start from the root of XML schema graph, because if there are no indices, every XML element can only be accessed from the root of XML instance tree.

We can simplify a long path computation into an XML extent index operation according to Theorem 1 and Theorem 2 to speed up the query. Actually, each element type has *unique access path* for itself. For an element type E, $//E$ is its *unique access path*, i.e., $UAP(E) = //E$. However, it is not common for path expressions to have *unique access path* of its end element. For example, the end element of query */site/closed_auctions/closed_auction/annotation/description* is *description*, which can also be accessed by */site/open_auctions/open_auction/annotation/description*. In this situation, the expression cannot be optimized using Theorem 1 and Theorem 2. However, the front part of the expression */site/closed_auctions/closed_auction* is the *unique access path* of element type *closed_auction*. Using Theorem 2, it can be simplified to $EXT(closed\_auction)$. The rest of the path expression can be evaluated using the *extent join* algorithm. Thus, we can get a corollary of Theorem 2.

Corollary. *Assume that P_1 and P_2 are two path expressions. For two path expression queries $Q_1 = \langle G_t, T_d, root_t, /P_1/E/P_2, RS_1\rangle$ and $Q_2 = \langle G_t, T_d, E, /P_2, RS_2\rangle$, if $UAP(E) = PE + P_1$, then $RS_1 = RS_2$.*

The optimization of absolute path expressions has been discussed. Now we discuss relative path expressions.

Definition 7. *Let G_t be an XML schema and T_d an XML data document complying with schema G_t. If E_1 and $E_2 \in V_t$, and $(\forall e_2 \in V_d)(e_2 \in EXT(E_e) \rightarrow (\exists e_1)(e_1 \in EXT(E_1) \wedge \delta_d^{-1}(e_2) = e_1))$, then E_1 is the unique parent of E_2, in short $UP(E_2) = E_1$.*

Since G_t is a graph structure, an element may have more than one parent. We can define the *unique parent set* of an element similarly. It is easy to determine if an element is the *unique parent* of another element using function δ_t^{-1}. If $\delta_t^{-1}(E_2) = \{E_1\}$, then $UP(E_2) = E_1$, else $UP(E_2) \neq E_1$.

Definition 8. *Let G_t be an XML schema and T_d an XML data document complying with schema G_t. If E_1 and $E_2 \in V_t$, and $(\forall e_2 \in V_d)(e_2 \in EXT(E_e) \rightarrow (\exists e_1)(e_1 \in EXT(E_1) \wedge e_1$ is the ancestor of e_2. then E_1 is the key ancestor of E_2, meaning that accessing an instance of E_2 must be through E_1, in short $KA(E_2) = E_1$.*

Theorem 3. *Let E_i and C_i represent steps and path connectors in absolute path expression $E_1C_1E_2C_2E_3 \ldots C_{n-1}E_n$ for i $(1 \leqslant i \leqslant n)$, respectively. For query $Q_1 = \langle G_t, T_d, E_1, C_1E_2C_2E_3 \ldots C_{n-1}E_n, RS_1\rangle$, if $UAP(E_2) = C_1E_2$, then Q_1 can be equivalently replaced by relative path expression query $Q_2 = \langle G_t, T_d, E_2, C_2E_3 \ldots C_{n-1}E_n, RS_2\rangle$; if E_2 is the unique parent or key ancestor of E_3, then Q_1 can be equivalently replaced by $Q_3 = \langle G_t, T_d, E_3, C_3E_4 \ldots C_{n-1}E_n, RS_3\rangle$, i.e., $UAP(E_3) = C_1E_2C_2E_3$.*

Proof. Assume that C_2 is "/". Consider query $Q_4 = \langle G_t, T_d, E_1, C_1E_2, RS_4\rangle$. Then $RS_4 = EXT(E_2)$, since $UAP(E_2) = E_1C_1E_2$ according to Theorem 2, that is, the instance set containing all the instances from E_1 along path C_1E_2 is $EXT(E_2)$. Because E_2 is the *unique parent* of E_3, each instance in $EXT(E_3)$ is the child of an instance in $EXT(E_2)$. So for query $Q_5 = \langle G_t, T_d, E_1, C_1E_2C_2E_3, RS_5\rangle$, $RS_5 = EXT(E_3)$. According to Definition 6, $UAP(E_3) = C_1E_2C_2E_3$. If C_2 is "//", the proof is similar. □

According to Theorem 3, in certain cases, Q_2 can be optimized to Q_3. Thus, relative path expression queries can also be optimized. The three theorems and the corollary discussed above also apply to other instances. Using them, a path expression query can be shortened according to the information in XML document schema to reduce the cost of the query. These theorems are called *path*

shortening strategy of path expression queries.

Theorem 3 not only can be used to shorten path expressions, but also gives the principle of executing the path optimization strategy. A path expression query is shortened from its beginning step by step, until it cannot be optimized. It can be simply done when the path connector is "/". It only needs to determine the *unique parent* of an element. It is a little bit complex for the case that the path connector is "//", which needs to determine the *key ancestor* rather than *unique parent*. We can use reverse path tree to determine key ancestor. According to Algorithm 3, the leaf nodes of $RPT(E_1, E_2)$ can only be either E_1 or $root_t$. Furthermore, when E_1 is the *key ancestor* of E_2, the leaf nodes of $RPT(E_1, E_2)$ must be E_1. We can determine if an element is the *key ancestor* of another element based on this property. Thus, the algorithm of the *path shortening* strategy can be described as shown in Algorithm 3.

Algorithm 3. Path Shortening Algorithm

Input: Path expression query $Q = \left(G_t, T_d, root_t, \sum_{i=1}^{n} C_i E_i, RS\right)$, XML schema $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$

Output: Optimized path expression query Q'

```
(01) for i = 1 to n − 1 do
(02)     if (C_i = "/") && (δ_t^{-1}(E_{i+1}) = {E_i}) then
(03)         cs = i + 1;
(04)         continue;
(05)     else break;
(06)     end if
(07)     if C_i ="//" then
(08)         construct RPT(E_i, E_{i+1});
(09)         if all leaves of RPT(E_i, E_{i+1}) are E_i
             then
(10)             cs = i + 1;
(11)             continue;
(12)         else break;
(13)         end if
(14)     end if
(15) end for
(16) Q' = ⟨G_t, T_d, E_cs, Σ_{i=cs+1}^{n} C_i E_i, RS⟩;
```

Finally, whether or not a path expression can be optimized using the *path shortening* strategy depends on the feature of the path expression itself, i.e., whether or not part of the expression is a *unique access path* of certain element. It is a heuristic rule, and it does not need any statistic information of data, since it can reduce join operations while evaluating the query and certainly improve the query performance. Besides the *extent join* algorithm, this optimization strategy can also be used for other path expression computing algorithms with the support of XML extent index. For example, when using the *top-down* strategy, the navigation operations on XML tree can start from instances of certain element extent rather than from the root.

4.2 Path Complementing Strategy

The *path shortening* strategy improves the query performance by optimizing the path expression itself. We now introduce the *path complementing* strategy, which computes complex and higher-cost query expression by simple and lower-cost query expressions. Like the *path shortening* strategy, this strategy also needs information on XML document schema.

Before introducing the *path complementing* strategy, let us look at an example first. Consider an absolute path query $Q_1 = \langle G_t, T_d, root_t, /site/regions/*/item/name, RS_1\rangle$ that retrieves the names of items in all regions based on the example in Fig.1 and schema in Fig.2. Only *item* and *person* may have child element of type *name* among all elements. If an XML instance typed *name* in the document is not the child node of *item*, it must be the child node of *person*; that is, for the instance set of all *name* elements, the instance set of *name* child of *item* and that of *person* are complementary. Since all the *name* children of *person* can be obtained by query $Q_2 = \langle G_t, T_d, root_t, /site/people/person/name, RS_2\rangle$, $EXT(name) = RS_1 \cup RS_2$, i.e., $RS_1 = EXT(name) - RS_2$. Then there are two query plans that can be chosen when executing a path query. One is Q_1 and the other is $EXT(name) - RS_2$. The costs of these two plans may be different from each other, the lower one of which should be chosen to compute the query by the cost evaluator.

Definition 9. *Let $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$ be an XML schema and $T_d = (V_d, E_d, \delta_d, \Sigma_d, root_d, oid, type_d, oid_d)$ an XML data document complying with schema G_t. If $E_2 \in V_t$, $KA(E_2) = E_1$, there is n paths between E_1 and E_2 represented by $P_1, P_2, \ldots, P_n$, and $End(P_i) = E_2$ $(1 \leqslant i \leqslant n)$, then $\bigcup_{j=1}^{i-1} P_j + \bigcup_{j=i+1}^{n} P_j$ $(1 \leqslant i \leqslant n)$ is the complementary paths of P_i between E_1 and E_2, abbreviated as $CP(E_1, E_2, P_i) = \bigcup_{j=1}^{i-1} P_j + \bigcup_{j=i+1}^{n} P_j$.*

Theorem 4. *If $CP(E_1, E_2, P_i) = \bigcup_{j=1}^{i-1} P_j +$*

$\bigcup_{j=i+1}^{n} P_j$, *and the corresponding query of path* P_i *is* $Q_i = \langle G_t, T_d, E_1, P_i, RS_i \rangle$ $(1 \leqslant i \leqslant n)$, *then* $(\forall e_2)(e_2 \in EXT(E_2) \rightarrow e_2 \in \bigcup_{i=1}^{n} RS_i)$, *i.e.,* $EXT(E_2) = \bigcup_{i=1}^{n} RS_i$, *that is* $RS_i = EXT(E_2) - \bigcup_{j=1}^{i-1} P_j - \bigcup_{j=i+1}^{n} P_j$.

Proof. Because $CP(E_1, E_2, P_i) = \bigcup_{j=1}^{i-1} P_j + \bigcup_{j=i+1}^{n} P_j$, $KA(E_2) = E_1$ and $P_1, P_2, \ldots, P_n$ are all the paths between E_1 and E_2. Assume there exists an XML instance $e \in EXT(E_2)$ and $e \notin \bigcup_{i=1}^{n} RS_i$. Since $KA(E_2) = E_1$, there must exist an instance $e' \in EXT(E_1)$ such that e' is an ancestor of e in T_d. Thus, there exists a path $P' \neq P_i$ $(1 \leqslant i \leqslant n)$ and its corresponding query $Q' = \langle G_t, T_d, E_t, P', RS' \rangle$ such that $e' \in RS'$. It is paradoxical. Thus the theory is proved. □

The path expression query obtained after using the *path complementing* strategy can also be ulteriorly optimized using the *path shortening* strategy. The key problem of the *path complementing* strategy is to find the *complementary paths* of a path expression query and choose the query plan according to their costs.

For two element types E_1 and E_2, if $KA(E_2) = E_1$, then the *reverse path tree* $RPT(E_1, E_2)$ can be used to compute all the paths between them. Chase the RPT pointer reversely from its leaf nodes to the root. The connector between nodes is usually "/". When the reverse pointer chasing gets the same element, rewrite the part of path between them to a closure structure or use connector "//" to replace it. For example, the path $/a/b/c/d/b/e$ is rewritten to $/a/(b/c/d)^+/e$. This operation is the reverse work of deducing cyclic paths in the building of RPT. The operation of getting *complementary paths* is fairly straightforward when all paths between two nodes are gotten, and only path matching and set difference operations are needed. Algorithm 4 uses RPT to determine the *key ancestor*. At the same time, the *complementary paths* are obtained and the function $getCP(E_1, E_2, P)$ is used to represent the procedure of getting *complementary paths*. Then, this algorithm performs cost estimating on each query strategy, and executes the query using the least cost strategy. Before the cost estimating, the path expressions should be optimized using the *path shortening* strategy, and the functions $PS(P)$ and $Cost(P)$ are used to represent the *path shortening* procedure and the cost estimating procedure, respectively. Additionally, this algorithm applies the *path complementing* strategy from the end of path, and stops whenever a lower cost strategy than the original one is gotten. In this way, the longer the path is, the greater the performance improvement of *path complementing* is expected.

Algorithm 4. Path Complementing Algorithm
Input: Path expression query $Q = (G_t, T_d, E_0, P, RS), P = \sum_{i=1}^{n} C_i E_i$, XML schema $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$
Output: Optimized query strategy S

```
(01) if KA(E_n) = E_0 then
(02)     P' = getCP(E_0, E_n, P)
(03)     if Cost(EXT(E_n) − PS(P')) < Cost(PS(P))
         then
(04)         S = EXT(E_n) − PS(P')
(05)         return
(06)     end if
(07) end if
(08) for i = n − 1 to step−1 do
(09)     if KA(E_i) = E_0 then
(10)         tmpp = Σ_{j=1}^{i} C_j E_j
(11)         P' = getCP(E_o, E_i, tmpp)
(12)         if Cost(EXT(E_n) − PS(P')) < Cost(PS
               (tmpp)) then
(13)             S = (EXT(E_n) − PS(P') + Σ_{j=i}^{n} C_j E_j)
(14)             return
(15)         end if
(16)     end if
(17) end for
```

4.3 Querying and Optimizing Path Expressions

In the above subsections, two optimization techniques are proposed for path expression. We have introduced that the *path-shortening* rule is heuristic, while the *path-complementing* technique is not suitable for all cases. Therefore, a cost based query plan selection is used for the path optimization procedure. In this subsection, we show how to use them in path expression query processing procedure. The selection of path expression and cost estimation are not the focuses of this paper, so the details of these issues are omitted. Given a path expression query P and an XML schema graph $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$, the general steps of querying and optimizing path expression queries

are shown as follows.

Step 1. Rewriting of '*'. With the XML schema graph, path steps '*' are rewritten as the unions of all possible sub-paths via function δ_t.

Step 2. Complementary path selection. With the XML schema graph, the complementary paths of user query are found and their costs are estimated. Check if the cost of complementary paths is lower than that of the original path. If it is lower, the complementary approach is chosen. Otherwise, the original path is chosen.

Step 3. *Path shortening.* Using Algorithm 3 to shorten the selected path expressions.

Step 4. Rewriting of connector "//". Check if there exist "//" connectors with no *ADX* support. If so, they are rewritten using Algorithm 1.

Step 5. Index selection and query plan construction. Select correct indexes and transform the path expressions into query plans.

Step 6. Query plan execution. Execute the query plan including indexes and joins.

5 Experiments

In this section, we will discuss the performance evaluation of the *extent join* and the path expression optimization rules proposed in this paper in terms of four benchmarks: *XMark*, *XMach*, *DBLP* and *Shake*.

5.1 Overview

The experiments were conducted on a single 800MHz CPU PC with 184MB main memory. We employed a native XML management system called XBase[24] as the underlying data storage, which stores XML document into an object database with an ODMG-binding DOM interface. The testing programs were coded with MS VC++ 6.0 and ODMG C++OML 2.0[25]. The datasets used are described as follows.

XMark. The first data set is from the XML benchmark project[24]. The scale factor selected is 1.0 and the corresponding XML document size is about 100MB. The hierarchical schema is the same as that in Fig.2. XMark focuses on the core ingredient of XML benchmark including the query processor and its interaction with the database. XMark totally specifies 20 queries that cover a wide range including exact match, ordered access, casting, regular path expressions, chasing references, construction of complex results, join on values, reconstruction, full text, path traversals, missing elements, function application, sorting and aggregation.

XMach. The second data set is a scalable multi-user benchmark to evaluate the performance of XML data management systems proposed by Rahm and Bohme[26]. It is based on a web application and considers different types of XML data, in particular text documents, schema-less data and structured data. The database contains a directory structure and XML documents. It is a multiple DTD and multiple document benchmark that totally consists of 11 queries: 8 retrieval and 3 update queries.

The above two data sets are used to simulate some applications, while the following two data sets are real data.

Shakes. The third data set is the Bosak Shakespeare collection available at `http://metalab.unc.edu/bosak/xml/eg/shakes200.zip`. 8 queries are designed over the *Shakes* data set, as shown in Table 1.

Table 1. Queries on *Shakes*

No.	Path expression queries
Q1	/PLAY/ACT
Q2	/PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR
Q3	//SCENE/TITLE
Q4	//ACT//TITLE
Q5	/PLAY/ACT [2]
Q6	(/PLAY/ACT) [2]/TITLE
Q7	/PLAY/ACT/SCENE/SPEECH[SPEAKER = "CURIO"]
Q8	/PLAY/ACT/SCENE[//SPEAKER = "Steword"]/ TITLE

DBLP. The last data set is from the DBLP bibliography web site, available at `ftp://ftp.informatic.uni-trier.de/pub/users/Ley/bib/records.tar.gz`. 8 queries are defined over the *DBLP* data set, as shown in Table 2.

Table 2. Queries on DBLP

No.	Queries
Q1	Select all conference paper titles published in 2000 on XML
Q2	Select all paper titles written by Michael Stonebraker
Q3	Select all paper titles written by Michael Stonebraker or Jim Gray
Q4	Select all database papers published between 1990 and 1994
Q5	Select all papers that have a citation entry whose label is CARE84
Q6	Select database paper titles with paper length longer than 20 pages
Q7	Select all papers by Michael Stonebraker quoted by papers published in year 1994
Q8	Select all papers by Jim Gray that are quoted by Michael Stonebraker

Table 3. Parameters of Generated Benchmark Databases

Benchmark	Document		Number of		Database Size	Index Size
	Amount	Size	Elements	Attributes		
XMark	1	100M	1,360,720	381,880	319.9M	14.0M
XMach	10,001	71.4M	289,694	129,147	157.8M	5.8M
Shakes	37	7.4M	179,690	0	16.3M	1.1M
DBLP	275,523	108M	2,785,894	350,376	328.7M	19.7M

The parameters of four benchmark databases are shown in Table 3.

In order to fully explore the performance of the *extent join* algorithm and query optimization techniques proposed in this paper, we implemented 3 different query evaluating strategies: *top-dowm*, *extent join*, *optimized*. The *top-down* strategy evaluates path expressions by traversing the XML data tree from top to down with no index support, which is similar to the *top-down* approach described in [10]. The *extent join* approach is supported by indexes including *ADX*, *PCX* and *RX*. The *optimized* way optimizes the *extent join* way by applying query optimization rules. It follows the optimizing steps in Section 4 to select the most optimal query execution plan.

5.2 Extent Join

Fig.6 shows the performance comparison between *top down* and *extent join* in terms of XMark. *Extent join* is much better than *top down* in most cases. The *extent join* is about 2 $\sim$ 20 times, sometimes hundreds of times, faster than *top down*. However, there are some exceptions. 1) For Q2, Q3, Q13 and Q14, the performance of *extent join* is similar to that of *top down*. The reasons are described as follows. a) Q2 and Q3 are ordered accesses to elements. In this case, *extent join* also needs to traverse the XML data trees. b) Q13 is result reconstruction and needs to traverse a relatively big sub-tree to get all results. c) Q14 is a full text query, which also needs to traverse the whole sub-tree to check if elements are right. (2) For Q15 and Q16 containing very long path traversals, *top down* outperforms *extent join* by about 30%. Due to the much smaller selectivity of path expression *top down* does not need to traverse the whole XML data tree, whereas *extent join* must do many join operations (e.g., Q15: 12, Q16: 14). Then we can get a conclusion: *extent join* is better than *top down* in most cases unless it needs to traverse a large XML data tree like *top down* or the path queries are very long such that *extent join* must do too many join operations.

Fig.7 shows the performance comparison between *top down* and *extent join* in terms of *XMach*. From the figure, we can see that the performance of *extent join* for $Q1$, $Q2$, $Q6$, $Q7$ and $Q8$ outperforms that for $Q3$, $Q4$ and $Q5$. In the cases of Q_1, Q_2, Q_6, Q_7 and Q_8, the *top down* approach has to navigate a large portion of XML trees while the *extent join* approach can save I/O overhead with the help of indexes. $Q3$ is a recursive query. The *top down* approach computes the recursive operation by navigating a small portion of XML trees while the *extent join* approach conducts this recursion by a recursive join operation in all XML trees. Therefore, the I/O cost of *top down* is lower than that of *extent join*. $Q4$ needs to reconstruct the query result according to the document order. *top down* outperforms *extent join* because navigating XML

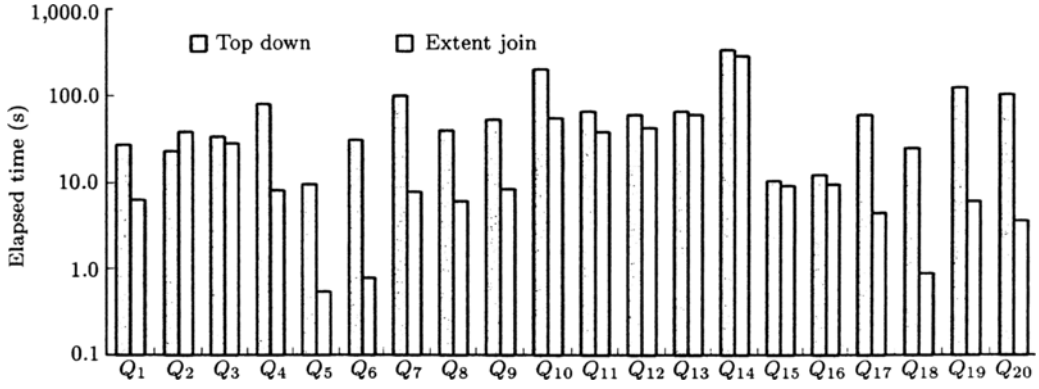

Fig.6. Extent join (XMark).

trees is a natural result-reconstructing procedure. As for $Q5$, the reason why *top down* outperforms *extent join* is similar to $Q3$.

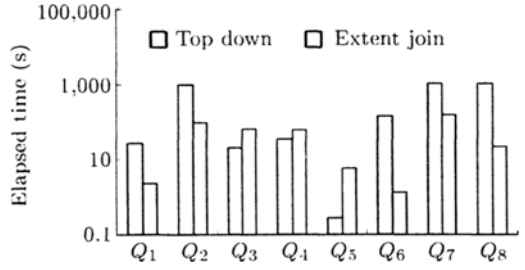

Fig.7. Extent join (XMach).

Figs.8 and 9 are the performance comparison between *top down* and *extent join* for the two real data sets, respectively. First, consider DBLP where most of queries are very long and have predicates at the end. *Extent join* is much better than *top down* (Q2, Q3, Q4, Q5 and Q6). There exists a containing operator in Q1 and the path expressions in it are relatively short. All these factors cause *top down* to be better than *extent join* for this query. The performance of *extent join* on Q7 and Q8 is very bad and we cannot get performance results. The reason may be that they all contain several (4 or 5) long path expressions with more than 10 steps.

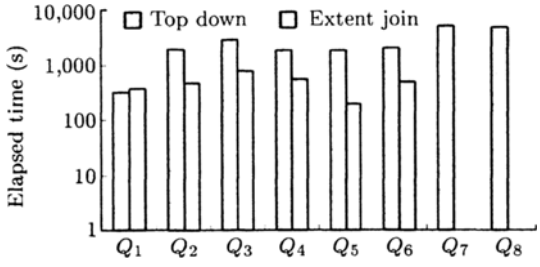

Fig.8. Extent join (DBLP).

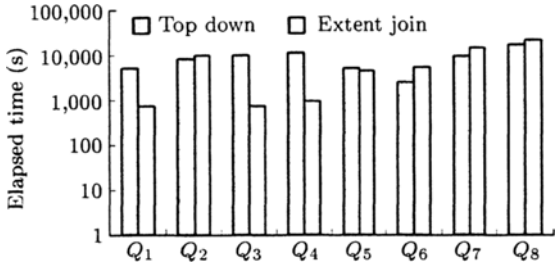

Fig.9. Extent join (Shakes).

From Fig.9, we can also see that the *extent join* performance for some queries ($Q1$, $Q3$, $Q4$ and $Q5$) outperforms the *top down* performance. From systematic analysis, we can see that the performance of *top down* heavily depends on the portion of XML trees navigated while the performance of *extent join* mainly depends on the number of joins. So we have proposed some techniques to improve the performance of the *extent join* approach and the performance evaluation is given in the next section.

5.3 Query Optimization

Figs.10 and 11 show the performance comparison between *extent join* and *optimized* for *XMark* and *XMach*, respectively. The *optimized* approach is the winner in all query results, and queries are divided into several categories.

(1) Query performance is improved greatly. Examples are Q5, Q6, Q7, Q18 and Q20 of XMark, whose path expressions are shortened greatly, and these queries have no predicates or the predicates are at the last step of the paths. In these cases, the *optimized* approach can be 10 $\sim$ 200 times faster than *extent join*.

(2) Query performance is improved moderately. Q1, Q2, Q3, Q4, Q8, Q9, Q10, Q11, Q12 and Q17 belong to this category. They are either queries that can only be shortened a little by *path-shortening* rule and the saved *extent join* operations take relatively small costs (Q1), or queries that have some other high-cost operations, for example, join on values, ordered access and reference chasing. In this category the benefits of query optimization rules cannot be seen clearly (Q2, Q3, Q4, Q8, Q9, Q10, Q11, Q12). They also may be queries whose complementary paths are still very complex (Q17). For queries of this category, the *optimized* approach can save the evaluating time by 10% $\sim$ 400%. Most queries fall in this category.

(3) Query performance is improved slightly. Q13, Q14, Q15 and Q16 of XMark fall in this category and the benefit of the *optimized* approach for them is only 0.3% $\sim$ 8%. The reasons are that these queries have operations of very high cost (Q13: complex result reconstruction, Q14: full text scanning) or they are expressions of very long path and can only be shortened little (Q15: 2 out of 12, Q16: 2 out of 14). The XMach results in Fig.11 also indicate the similar result (Q2, Q3, Q4, Q5, Q6 and Q7 belong to category 1, Q8 belongs to category 2 and Q1 belongs to category 3).

Figs.12 and 13 are the performance comparison between *extent join* and *optimized* for the two real data sets, DBLP (Fig.12) and Shakes (Fig.13).

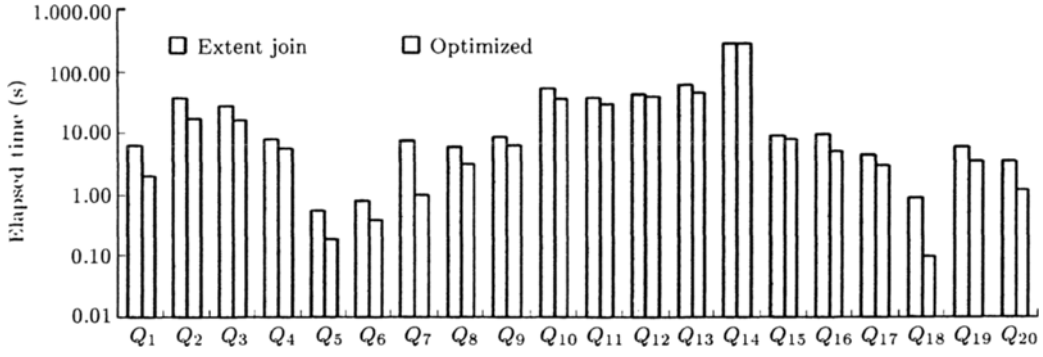

Fig.10. Query optimization (XMark).

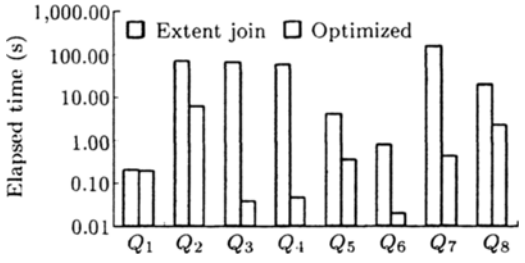

Fig.11. Query optimization (XMach).

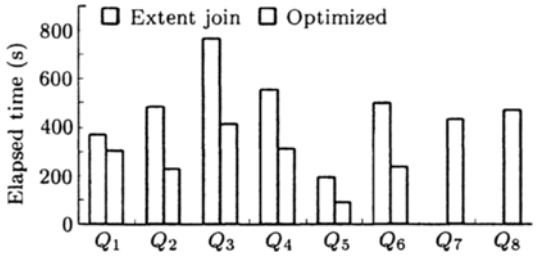

Fig.12. Query optimization (DBLP).

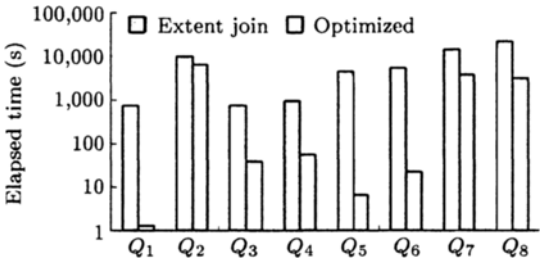

Fig.13. Query optimization (Shakes).

From these two figures we can see that *optimized* outperforms *extent join* overall queries in both DBLP and Shakes. The performance analysis is similar to benckmarks *XMark* and *XMach*.

6 Conclusions

In this paper, we proposed the *extent join* approach to evaluate regular path expressions. In order to further improve the query performance, we also proposed two novel query optimization techniques: *path-shortening* and *path-complementing*. *Path-shortening* reduces the number of joins by shortening the path and *path-complementing* is a technique to use an equivalent complementary path expression to compute the original path specified in a user query. They can reduce the path computing cost by decreasing the length of paths and using equivalent complementary expressions to optimize long and complex paths. From our experimental results, 80% of the queries can benefit from these optimization rules, and path expression evaluating performance can be improved by 20% $\sim$ 400%.

References

[1] Florescu D, Kossmann D. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. INRIA Tech. Report, INRIA , No.3680, 1999.

[2] Florescu D, Kossmann D. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 1999, 3: 27–34.

[3] Deutsch A, Fernandez M, Suciu D. Storing semistructured data with STORED. In *Proc. the 1999 SIGMOD Conference*, Philadelphia, USA, 1999, pp.431–442.

[4] Manolescu I, Florescu D, Kossmann D *et al.* Agora: Living with XML and relational. In *Proc. the 26th VLDB Conference*, Cairo, Egypt, 2000, pp.623–626.

[5] Schmidt A, Kersten M, Windhouwer M *et al.* Efficient relational storage and retrieval of XML documents. In *Proc. the 3rd International Workshop WebDB*, Dallas, USA, 2000, pp.137–150.

[6] Zhou A, Lu H, Zheng S *et al.* VXMLR: A visual XML-relational database system. In *Proc. the 27th VLDB Conference*, Roma, Italy, 2001, pp.719–720.

[7] Fegaras L, Elmasri R. Query engines for Web-accessible XML data. In *Proc. the 27th VLDB Conference,* Roma, Italy, 2001, pp.251–260.

[8] Hou J, Zhang Y, Kambayashi Y. Object-oriented representation for XML data. In *Proc. the 3rd CODAS Conference,* Beijing, China, 2001, pp.43–52.

[9] Renner A. XML data and object databases: A perfect couple? In *Proc. the 17th ICDE Conference,* Heidelberg, 2001, pp.143–148.

[10] McHugh J, Abiteboul S, Goldman R *et al.* Lore: A database management system for semistructured data. *SIGMOD Record,* 1997, 3: 54–66.

[11] Shoens K, Luniewski A, Schwarz P *et al.* The Rufus system: Information organization for semi-structured data. In *Proc. the 19th VLDB Conference,* Dublin, 1993, pp.97–107.

[12] Fernandez M, Florescu D, Kang J *et al.* Catching the boat with Strudel: Experiences with a Web-site management system. In *Proc. the 1998 SIGMOD Conference,* Seattle, USA, 1998, pp.414–425.

[13] Schoning H. Tamino — A DBMS designed for XML. In *Proc. the 17th ICDE Conference,* Heidelberg, Germany, 2001, pp.149–154.

[14] Goldman R, Widom J. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. the 23rd VLDB conference,* Athens, Greece, 1997, pp.436–445.

[15] Milo T, Suciu D. Index structures for path expressions. In *Proc. the International Conference on Database Theory,* Jerusalem, Israel, 1999, pp.277–295.

[16] Li Q, Moon B. Indexing and querying XML data for regular path expressions. In *Proc. the 27th VLDB Conference,* Roma, Italy, 2001, pp.361–370.

[17] Chamberlin D, Robie J, Florescu D.:Quilt: An XML query language for heterogeneous data sources. In *Proc. 3rd International Workshop WebDB,* Dallas, 2000, pp.1–25.

[18] Fankhauser P. XQuery formal semantics: State and challenges. *SIGMOD Record,* 2001, 3: 14–19.

[19] Deutsch A, Fernandez M, Florescu D *et al.* Xml-ql: A query language for XML. 1999, http://www.w3.org/TR/NOTE-xml-ql/.

[20] Robie J, Lapp J, Schach D. XML query language (XQL). 1998, http://www.w3.org/TandS/QL/QL98/cfp.

[21] Cark J, DeRose S. XMP path language (XPath). Technical Report REC-xpath-19991116, W3C, 1999.

[22] Abiteboul S, Quass D, McHugh J *et al.* The Lorel query language for semistructured data. *International Journal on Digital Libraries,* 1997, 1: 68–88.

[23] Schmidt A, Waas M, Kersten M L *et al.* XMark: A benchmark for XML data management. In *Proc. 28th VLDB Conference,* Hong Kong, China, 2002, pp.974–985.

[24] Wang G, Lu H, Yu G, Bao Y. Managing very large document collections using semantics. *Journal of Computer Science and Technology,* 2003, 18(3): 403–406.

[25] Cattel R G G, Barry D, Berler M *et al.* The Object Data Standard: ODMG 3.0. Morgan Kaufmann, 2000.

[26] Rahm E, Bohme T. XMach-1: Multi-user evaluation of XML data management systems with XMach-1. In *Proc. 1st VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT2002),* Hong Kong, China, 2002, pp.148–158.

Guo-Ren Wang is a professor at Northeastern University, China. He received his B.E. degree, M.E. degree and Ph.D. degree from Northeasern University in 1988, 1991 and 1996, respectively. His research interests include XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, and parallel database systems.

Bing Sun is a Ph.D. candidate at Northeastern University, China. His research interests include XML data management, query processing and optimization.

Jian-Hua Lv is a Ph.D. candidate at Northeastern University, China. His research interests include XML data management, query processing and optimization.

Ge Yu is a professor at Northeastern University, China, and a supervisor of Ph.D. students. He received his B.E. degree and M.E. degree from Northeasern University in 1982 and 1986, respectively, and his Ph.D. degree from Kyushu University, Japan in 1996. He is a member of IPSJ, ACM, and ACM SIGMOD. His research interests include distributed and parallel databases system, objected-oriented database system, multi-database and information integration, data warehousing and data mining, transactional workflow management, and Web-service.