Query Processing and Optimization for Regular Path Expressions

Guoren Wang and Mengchi Liu

School of Computer Science, Carleton University
Ottawa, Ontario, Canada K1S 5B6
{wanggr, mengchi}@scs.carleton.ca

Abstract. Regular path expression is one of the core components of XML query languages, and several approaches to evaluating regular path expressions have been proposed. In this paper, a new path expression evaluation approach, *extent join*, is proposed to compute both parent-children ('/') and ancestor-descendent ('//') connectors between path steps. Furthermore, two path expression optimization rules, *path-shortening* and *path-complementing*, are proposed. The former reduces the number of joins by shortening the path while the latter optimizes the execution of a path by using an equivalent complementary path expression to compute the original path. Experimental results show that the algorithms proposed in this paper are much more efficient than conventional ones.

1 Introduction

With the rapid development of advanced applications on the Web, numerous amount of information becomes available on the Web and almost all the corresponding documents are semi-structured. As the emerging standard for data representation and exchange on the Web, XML has been adopted by more and more applications as their information description mean. Even though XML is mainly used as an information exchange standard, storing, indexing and querying XML data have become research hotspots both in the academic community and in the industrial community.

To retrieve XML data, many query languages have been proposed so far, such as Quilt [3], XQuery [4], XQL [5], XPath [6], and Lorel [7]. Because one of the common features of these languages is the use of regular path expressions (RPE), query rewriting and optimization for RPE is becoming a research hotspot and some research results have been obtained recently. A usual way to optimize the execution of RPE expressions is to first rewrite RPE queries into simple path expressions (SPE) based on schema information and statistics about XML data, and then translate these SPE queries into the language of the database used to store the XML data, for example, into SQL. 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* approach navigates the document tree from the root to the leaf nodes while the *bottom-up* approach does from the

J. Eder and M. Missikoff (Eds.): CAiSE 2003, LNCS 2681, pp. 30–45, 2003.

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 [1], regular path expressions containing '//' and/or '*' operators are rewritten with simple path queries based on schema information and statistics. The paper in [2] 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 a new path expression evaluation approach, called *extent join*, to compute both parent-children ('/') and ancestor-descendent ('//') connectors between path steps. In order to support the *extent join* approach, we introduce indexes preserving parent-children and ancestor-descendent relationships. Furthermore, we propose two path expression optimization rules, *path-shortening* and *path-complementing*. The Path-shortening rule reduces the number of joins by shortening the path while the path-complementing optimizes the execution of a path by using an equivalent complementary path to compute the original path. The performances of the query processing and optimization techniques proposed in this paper are fully evaluated with four benchmarks, *XMark*, *XMach*, *Shakes* and *DBLP*.

The remainder of this paper is organized as follows. Section 2 presents some preliminaries 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 the paper.

2 Preliminaries

In this section we review some concepts and definitions used throughout the paper.

2.1 XML Data Tree

XML is proposed by W3C as a standard for data representation and exchange, in which information is represented by elements that can be nested and attributes that are parts of elements. Document Object Model (DOM) is an application programming interface (API) for XML and HTML documents, which defines the logical structure of documents and the way a document is accessed and manipulated. In DOM, XML data are abstracted into entities, *elements* and *attributes*, and these entities are organized together via parent-children and element-attribute relationships to form a data tree, i.e. DOM tree. In this paper, we model XML data as a node-labelled tree in the following.

Definition 1. Formally, an XML data is 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. Every node has a unique name that is a string-literal of Σ_d and a unique identifier in set *oid*. Finally, every XML data tree has a root element $root_d$ that is included in V_d.

Figure 1 shows part of an XML document proposed in the XML Benchmark project [11], it 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 Figure 1, there are two directed dashed lines between some nodes (*&23* and *&18*, *&28* and *&18*), representing the referencing-referenced relationship between elements.

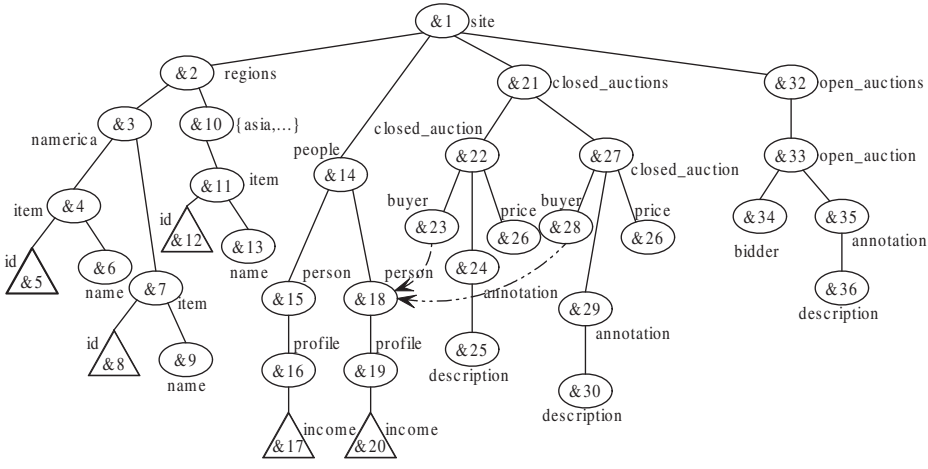

Fig. 1. Sample XML data tree

2.2 XML Schema Graph

Although XML data is self-descriptive, Document Type Definition (DTD) is proposed by W3C to further and explicitly constrains XML data, for example, an element should contain what kind of and/or how many sub-elements. XML mainly defines the parent-children relationship between XML elements and the order between the sub-elements of an element. In this paper, we model XML DTD as a directed, node-labelled graph.

Definition 2. Formally, an XML schema graph is defined as a directed, node-labelled graph $Gt = (V_t, E_t, \delta_t, \Sigma_t, root_t)$, 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 constrains which element can contain which sub-elements. Every node has a unique name that is a string-literal of Σ_t and this name is actually element type name. Finally, every 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.

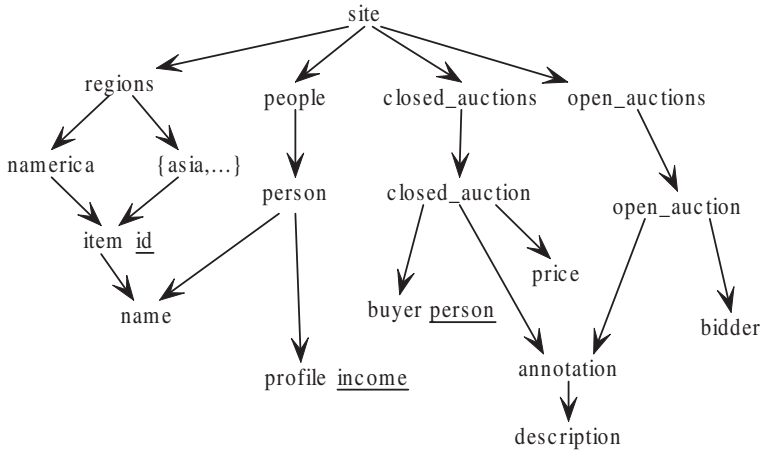

Fig. 2. XML Schema Graph

Figure 2 shows part of the XML schema graph that constrains the XML data tree in Figure 1, it is represented as an XML schema graph. The nodes 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 Figure 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*.

2.3 Path Expression

Path expressions can be straightforwardly defined as a sequence of element type names connected by some connectors such as '/' and '//' 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 Figure 3.

Path expressions mainly consist of two parts, *path steps* and *connectors*. Every path expression must begin from the root, that is, begins with a connector '/' or '//'. There are two basic kinds of path steps, *Name* and *wildcard* '*'. Path

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

Fig. 3. BNF syntax of path expression

step *Name* means that in this step only the element instances with the tag name *Name* will be matched and '*' will match any element instances no matter which type it belongs 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 that the path expression begins from the root and the following path step is the descendant of the root, that is, '//' covers for sub-path-expressions with any length. A connector appearing between two path steps specifies the relationship between them. Connector '/' constrains that between the two path steps there must exist a parent-children relationship and '//' is an ancestor-descendant relationship constraint.

3 Extent Join

In this section, we discuss 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 3. *pcpair(pid, cid)* is a pair of oids, in which *pid*, *cid* $\in$ *oid* and *pid* is the parent of *cid*, for example, *pcpair(&1, &2)*.

Definition 4. *adpair(aid, did)* is a pair of oids, in which *aid*, *did* $\in$ *oid* and *aid* is the ancestor of *did*, for example, *adpair(&1, &3)*.

A *pcpair* is a special case of an *adpair*. Additionally, ε is defined to act as any element instance, so *adpair(ε,&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)* returns true. Otherwise, it returns false.

Definition 5. 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* $\wedge$ *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)* $\in$ *Ext(any, an)* $\wedge$ *pcpair(ε, did)* $\in$ *Ext(any, dn)* $\wedge$ *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 6. 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 introduce 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)* 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")* 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)*}.

For the indexes above, only the principles are introduced. Their implementations are relatively simple as 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 to replace 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. Then 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 the path expression *"/site//closed_auction/annotation/description"* that contains four path steps and three connectors. As shown in Figure 4, each path step corresponds to an intermediate results 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_auc-tions/closedc_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 corresponding extents, and the following shows the full transformation rules.

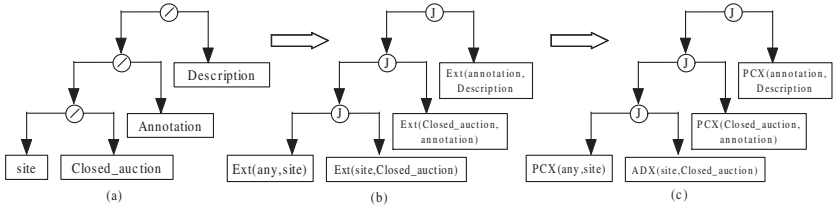

Fig. 4. (a) path expression, (b) extent join tree, and (c) execution tree

(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, δ_t*(site)* = {*regions, people, closed_auctions, ope_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 take 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 rewritting '//' connector, we first define an important data structure *reverse path tree (RPT)* as follows.

Definition 7. 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 same end path step together, where V_r is the node set that are 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 E which contains all path expressions from $root_r$ to E, and define $RPT(E_1, E_2)$ as a reverse path tree rooted E_2 which contains all path expressions from E_1 to E_2 and some path expressions from $root_r$ to E_2.

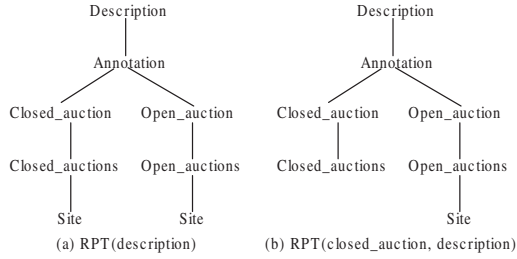

Fig. 5. Reverse path trees

For example, Figure 5 (a) shows *RPT(description)* and (b) shows *RPT(closed_acutions, description)*. 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_acutions* from *RPT(closed_acutions, description)*, we can just traverse up from the most left leaf node of Figure 5(b) to the root. So, the connectors rewriting algorithm is just the *RPT* constructing algorithm, shown as in Figure 6. With the proposed transformation rules and the algorithm, we have the *extent join* algorithm, details are shown in Figure 7.

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 present two path expression optimization techniques to reduce the number of joins and the execution cost of path expressions when evaluating a path expression. Meanwhile, the general cost based optimization procedure is also introduced in this section.

4.1 Path-Shortening

Most of the existing algorithms proposed for path expressions are based on the entire paths. Actually, a path expression can sometimes be computed with relative path rather than absolute path, depending on DTD or schema information. A simple example is path *"/site/closed_auctions/closed_auction/price"*. From the XML schema graph in Figure 2, we know that only through this path we can find price elements, so the result of this path is exactly all elements tagged *price*, i.e. *Ext(any, price)*. This principle can be summarized as follows.

Problem: construct a RPT (rewrite "$n_1//n_2$")
Input: XML Schema Graph $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$,
two graph nodes n_1, n_2
Output: result RPT rpt.
Algorithm body:
(1) $rpt.root_r = n_2$; // set the root of rpt
(2) $currentnode = rpt.root_r$; // set the current node
(3) $currentnode.children = \delta_t^{-1}(currentnode)$; // set children of current node.
(4) For every node $cc in \delta_t^{-1}(currentnode)$ do // recursively construct the RPT
(5) if $cc! = n_1$ and $cc! = root_t$
(6) $currentnode = cc$
(7) goto (3)
(8) endfor

Fig. 6. Constructing RPT Algorithm

InputPath Expression Query P
Output: Result Set R
Algorithm body
(1) Checking the ADX and rewrite the no-index-supported '//' connectors using the algorithm in Figure 6.
(2) Transforming the rewritten path P into joins and indexes and organized as a simple query plan.
(3) Executing the query plan including indexes and joins.

Fig. 7. Extent join Algorithm

Definition 8. Suppose that $/P/E$ be a path starting from the root element. If $(\forall e)(pcpair(p, e) \in Ext(any, E) \rightarrow adpair(\varepsilon, p) \in R(P))$, then P is a *unique path* from the root to E, written as $UP(E_n) = P$, where P is a path expression and *R(P)* means the result set of path *P*.

Rule 1. *Path-shortening*: if $UP(E) = P$, then $R(/P/E) = Ext(E)$.

However, the situation that the end node is in the unique path does not happen very often, for example, the end node in the path

"/site/closed_auctions/ closed_auction/annotation/description" can also be reached by path *"/site/openauc-tions/open_auction/annotation/description"*. The characteristic of this path expression is that its head segment is unique, for example, *"/site/closed_auctions/ closed_auction"* is a unique path. In this case, we can shorten the long path into a relative shorter one *"closed_auction/annotation/description"*. Then we can get the general *path-shortening* rule.

Rule 2. *General Path-shortening*: if $P_1/E/P_2$ is a path expression starting from the root and $UP(E) = P_1$, then $R(P_1/E/P_2) = R(E/P_2)$.

This optimization technique is heuristic one because it reduces the sets to be joined by shortening the path expressions, and its key problem is how to determine if a path expression is the unique path of a given element type. Actually we can shorten a path expression step by step till the most optimal case. The algorithm in Figure 8 shows how to shorten a path expression. For the simplicity of algorithm description, we assume that the path steps of the path expression to be shortened do not include '*' for they can easily be rewritten to paths not containing '*' using function δ_t. In this algorithm, the path expression is shortened from the head to the tail, and if one path step could not be shortened, the rests then do not need to be checked any more. The reverse path tree is used to help to shorten path steps with connector '//'.

Input: Path expression P = $\Sigma_{i=1}^{n} S_i C_i$, XML schema graph $G_t = (V_t, E_t, \delta_t, \Sigma_t, root_t)$
Output: Shortened path expression P'
Algorithm body:
(1) For i = 1 to n-1 do
(2) If S_i == '/' && $\delta_t^{-1}(S_{i+1}) = \{S_i\}$ then
(3) cs = i+1;
(4) continue;
(5) else break;
(6) end if
(7) If S_i == '//' then
(8) construct $RPT(S_i, S_{i+1})$
(9) if all leaves of $RPT(S_i, S_{i+1})$ are S_i then
(10) cs = i+1;
(11) continue;
(12) else break;
(13) end if
(14) end if
(15) End for
(16) P' = $\Sigma_{i=cs}^{n} S_i C_i$

Fig. 8. Path Shortening algorithm

4.2 Path-Complementing

Path-shortening reduces the cost of evaluating a path expression by optimizing the path itself. For example, consider the query "find all the names of items of all regions" that can be expressed as "/site/regions/*/item/name". From the XML schema graph in Figure 2 we know only elements *item* and *person* have element *name*, so these *name* element instances in database are either item names or person names. Actually all item names can be reached by path *"/site/regions/*/item/name"*, and all person names can be reached by path *"/site/people/person/name"*. So for element *name*, these two path expressions are complementary paths; that is, the former path can be evaluated by subtracting the results of the latter from the element extent *Ext(any, name)*. This gives us an alternative way to compute path expressions and we can choose the better one to get better performance.

Definition 9. Let E_1 and E_2 be element names, if $(\forall e_2)(pcpair(\varepsilon, e_2) \in Ext(any, E_2) \rightarrow (\exists e_1)(pcpair(\varepsilon, e_1) \in Ext(any, E_1) \wedge adpair(e_1, e_2)))$, then E_1 is a *key ancestor* of E_2.

Definition 10. Let E_1 is a key ancestor of E_2 and there exist path expressions P_1, P_2, $\cdots$, P_n from E_1 to E_2, if $(\forall e_2)(pcpair(\varepsilon, e_2) \in Ext(any, E_2) \rightarrow adpair(\varepsilon, e_2) \in \cup_{i=1}^{n} P(p_i))$, then $\cup_{j=1}^{i-1} p_j + \cup_{j=i+1}^{n} p_j$ is the complementary paths of P_i with respect to E_1 and E_2 .

Rule 3. *Path complementing*: if $\cup_{j=1}^{i-1} p_j + \cup_{j=i+1}^{n} p_j$ is the complementary paths of $P_i (1 \leq i \leq n)$ with respect to E_1 and E_2, then $R(P_i) = Ext(E_2)$ - $(\cup_{j=1}^{i-1} p_j + \cup_{j=i+1}^{n} p_j)$.

Obviously, reverse path tree is very helpful to determine the key ancestors of a given element type by checking the names of leaves and finding the complementary paths. Actually with reverse path tree the procedure is quite simple, so for saving the space, the details of this algorithm are omitted in this paper.

4.3 Querying and Optimizing Path Expressions

In the above subsections, two optimization techniques are proposed for path expression. We have mentioned 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 selectivity of path expression and cost estimation are not the focuses of this paper, so the details of these issues are ignored.

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 to 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 does, the complementary approach is chosen. Otherwise, the original path is chosen.

Step 3. *Path shortening.* Using the algorithm in Figure 8 to shorten the selected path expressions.

Step 4. Rewriting of connector '//'. Checking if there exists '//' connectors with no *ADX* support. If does, they are rewritten using the algorithm in Figure 6.

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

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

5 Experiments

5.1 Overview

In this section, we discuss the performance evaluation of the extent join path expression evaluation strategy and the path-shortening and path-complimenting path expression optimization rules in terms of four benchmarks. The experiments were made on a single 800MHz CPU PC with 184MB main memory. We employed a native XML management system called XBase [8] as the underlying data storage, which stores XML documents 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 [9]. The datasets used are described as follows.

XMark is from the XML benchmark project [11]. The scale factor selected is 1.0, and the corresponding XML document size is about 100MB. The structure of the document is modelled for a database as deployed by an Internet auction site. The hierarchical schema is the same as in Figure 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* is a scalable multi-user benchmark to evaluate the performance of XML data management systems proposed by Rahm and Bohme [10]. 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. *Shakes* is the Bosak Shakespeare collection available at `http://metalab.unc.edu/bosak/xml/eg/shakes200.zip`. 8 queries are designed over *Shakes* data set [12]. *DBLP* is from the DBLP bibliography web site, available at

`ftp://ftp.informatic.unitrier.de/pub/users/Ley/bib/records.tar.gz`. 8 queries are defined over the *DBLP* data set [12].

In order to full explore the performance of the *extent join* algorithm and query optimization techniques proposed in this paper, we implemented 4 different query evaluating strategies: *DOMTR*, *EJX*, *EJPX* and *EJOPX*. *DOMTR* evaluates path expressions by traversing the XML date tree from top to down with no index support, which is similar to the *top-down* approach. *EJX* is implemented as an *extent join* approach with all indexes except path indexes, including *ADX*, *PCX* and *RX*. Path indexes are optional for they must be specified explicitly by users, while other indexes are indispensable to *EJX*. *EJPX* is a full *extent join* algorithm with all indexes used to explore the performance of path indexes. In the extreme cases where all indexes are available, EJX and EJPX do not need to access the XML data trees. *EJOPX* is an all query optimization rule applied *extent join* algorithm. It follows the optimizing steps in Section 4 to select the most optimal query execution plan.

5.2 Extent Join

Figure 9 shows the performance comparison of *DOMTR*, *EJX* and *EJPX* in terms of XMark. *Extent join* algorithm is much better than DOMTR in most cases. The *extent join* is about $2 \sim 20$ times, sometimes hundreds of times faster than *DOMTR*. However, there are some exceptions. (1) For Q2, Q3, Q13 and Q14, the performance of *extent join* is similar to *DOMTR*. The reasons are different: a) Q2 and Q3 are order accesses to elements. In this case, even *extent join* needs to traverse the XML data trees; b) Q13 is result reconstruction and needs to traverse a relative 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, *DOMTR* outperformed *extent join* by about 30%. Due to the much smaller selectivity of path expression *DOMTR* 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 *DOMTR* in most cases except it needs to traverse a large XML data tree like *DOMTR* or the path queries are very long which *extent join* must do too many join operations.

There are only some queries in XMark can be evaluated using *EJPX* (Q8∼Q11, Q15∼Q17 and Q19) due to their characteristics. From Figure 9, we can see *EJPX* can improve the query performance by about 10%∼30 times over *EJX*. In the extreme case, such as Q15, *EJPX* can be thousands times faster than *EJX*. However, for some queries like Q11, Q12, Q14, and Q18, which either have join on values or are full text queries, the benefit of path indexes are drowned.

Figure 10 shows the update performance with and without indexes in XMach. Since only XMach has specified update queries, the test has not been done on other benchmarks. The total size of indexes is about 0.1 to 0.2 times of the size of the original XML document, and the response times of the update operations

are decreased by only about 10% to 20%. Thus, these indexes are much efficient and effective both in space utilization and supports for queries.

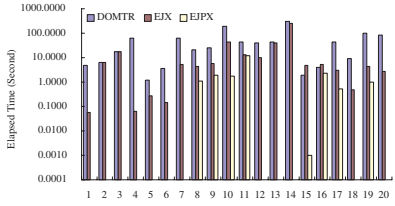

Fig. 9. Extent join (XMark)

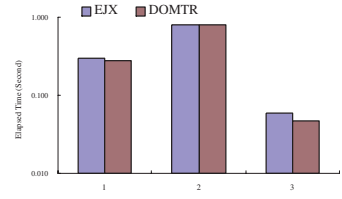

Fig. 10. Update queries (XMach)

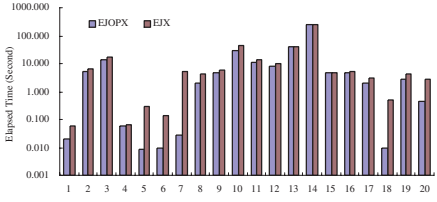

Fig. 11. Query optimization(XMarh)

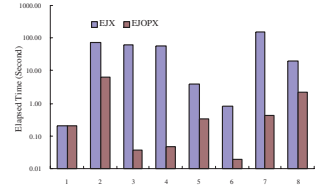

Fig. 12. Query optimization(XMach)

5.3 Query Optimization

Figures 11 and 12 show the performance comparison of *EJX* and *EJOPX* on XMark and XMach respectively. *EJOPX* is the winner in all query results, and queries are divided into several categories. (1) Query performance is improved greatly. Examples are Q5∼Q7, Q18 and Q20 of XMark, whose path expressions are shortened to a very short one, and these queries have no predicates or the predicates are at the last step of the paths. In these cases, *EJOPX* can be 10∼200 times faster than *EJX*. (2) Query performance is improved moderately. Q1∼Q4, Q8∼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 relative small costs (Q1), or queries have some other high cost operations, such as join on values, ordered access and references chasing, in which the benefits of query optimization rules cannot be seen clearly (Q2∼Q4, Q8∼Q12), or queries whose complementary paths are still very complex (Q17). For queries of this category, *EJOPX* can save the evaluating time by about 10%∼400%. Most queries fall in this category. (3) Query performance is improved slightly. Q13∼Q15 and Q16 of XMark fall in this category and the benefit of

EJOPX on them is only 0.3%∼8%. The reasons are that these queries have very high cost operations (Q13: complex result reconstruction, Q14: full text scanning) or they are very long path expressions and can only be shortened very little (Q15: 2 out of 12, Q16: 2 out of 14). The XMach results in Figure 12 also indicates the similar result (Q2∼Q7 belong to category 1, Q8 belongs to category 2 and Q1 belongs to category 3).

Figures 13 and 14 are the performance comparison of *DOMTR*, *EJX* and *EJOPX* over the two real data sets, DBLP in Figure 13 and Shakes in Figure 14. First, consider DBLP where most of queries are very long and have predicates at the end. *EJX* is much better than *DOMTR* (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 *DOMTR* is better than *EJX* in this query. The performance of *EJX* on Q7 and Q8 is very bad and we cannot get their performance results, the reason should be they all contain several (4 or 5) long path expressions with more than 10 steps. Nevertheless, *EJOPX* performs very well on all queries of DBLP. For some queries of Shakes (Q2, Q3, Q5, Q6, Q7 and Q8), the performance of *EJX* is not very good since they either have long and complex path expressions (Q2, Q3, Q7 and Q8) or contain order based operators (Q5 and Q6). However, *EJOPX* performs very well over the queries where *EJX* performs very poor (Q2, Q3, Q4, Q7 and Q8) since these long and complex path expression are optimized largely.

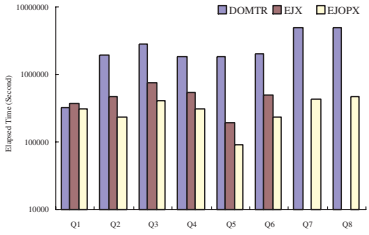

Fig. 13. Performance comparison(DBLP)

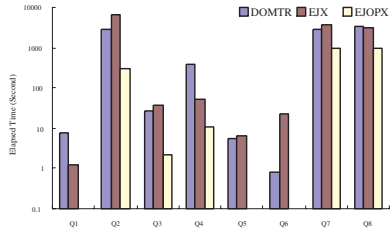

Fig. 14. Performance comparison(Shake)

6 Conclusions

In this paper, we have proposed the *extent join* approach to evaluating regular path expressions. In order to further improve the query performance, we also proposed two novel query optimization techniques, *path-shortening* and *path-complementing*. The former reduces the number of joins by shortening the path while the latter is a technique that uses 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% on average.

Acknowledgement. Guoren Wang's research is partially supported by the Teaching and Research Award Programme for Outstanding Young Teachers in Post-Secondary Institutions by the Ministry of Education, China (TRAPOYT) and National Natural Science Foundation of China under grant No. 60273079. Mengchi Liu's research is partially supported by National Science and Engineering Research Council of Canada.

References

1. Zhou, A., Lu, H., Zheng, S., Liang, Y., Zhang, L., Ji, W., Tian, Z.: VXMLR: A Visual XML-Relational Database System. In: Proceedings of the 27th VLDB Conference. Morgan Kaufmann. Roma, Italy (2001) 719–720
2. Li, Q., Moon, B.: Indexing and querying XML Data for regular path expressions. In: Proceedings of the 27th VLDB Conference. Morgan Kaufmann. Roma, Italy (2001) 361–370
3. Chamberlin, D., Robie, J., Florescu, D.: Quilt: An XML Query Language for Heterogeneous Data Sources. In: Proceedings of 3rd International Workshop WebDB. Lecture Notes in Computer Science Vol 1997. Dallas (2000) 1–25
4. Fankhauser, P.: XQuery Formal Semantics: State and Challenges. SIGMOD Record. **3** (2001) 14–19
5. Robie, J., Lapp, J., Schach, D.: XML Query Language (XQL). `http://www.w3.org/TandS/QL/QL98/cfp` (1998)
6. Cark, J., DeRose, S.: XMP Path Language (XPath). Technical Report REC-xpath-19991116, W3C (1999)
7. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.: The Lorel Query Language for Semistructured Data. Int'l J. Digital Libraries. **1** (1997) 68–88
8. Lu, H., Wang, G., Yu, G., Bao, Y., Lv, J., Yu, Y.: Xbase: Making your gigabyte disk queriable. In Proceedings of the 2002 ACM SIGMOD Conference. ACM Press. Madison, Wisconsin (2002) 630–630
9. Cattel, R.G.G, Barry, D., Berler, M., et al.: The object data standard: ODMG 3.0. Morgan Kaufmann (2000)
10. Rahm, E., Bohme, T.: XMach-1: A Multi-User Benchmark for XML Data Management, In: Proceedings of 1st VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques. Lecture Notes in Computer Science Vol 2590. Springer-Verlag, Berlin Heidelberg. Hong Kong, China (2002) 35–46
11. Schmidt, A., Waas, M., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: Proceedings of 28th VLDB Conference. Morgan Kaufmann. Hong Kong, China (2002) 974–985
12. Jiang, H., Lu, H., Wang, W., Yu, J.X.: Path Materialization Revisited: An Efficient Storage Model for XML Data. In: Proceedings of Thirteenth Australasian Database Conference. Australian Computer Society Inc. Melbourne, Victoria (2002)