Path Query

Yuqing Wu
Indiana University, Bloomington
http://www.cs.indiana.edu/ yuqwu

SYNONYMS

Document Path Query;

DEFINITION

Given a semi-structured data set D, a *path query* identifies nodes of interest by specifying the *path* lead to the nodes and the predicates associated with nodes along the *path*. The *path* is identified by specifying the labels of the nodes to be navigated and structural relationship (parent-child or ancestor-descendant) among the nodes. A predicate can be a path query itself, relative to the node that is associated with.

HISTORICAL BACKGROUND

Using path information in query processing has been studied in the object-oriented database systems, in which most queries require the traversing from one object to another following object identifiers, in the mid 90s. The notion of path query, in which the *path* and predicates along the path are specified as the core of the query, became popular with the growth of the information on the web and the introduction of semi-structured data, especially XML.

Most of the popular query languages for querying XML data, such as XPath and XQuery [4], employ path query as the approach to identify nodes of interest. However, some techniques, such as schema-free XQuery, relax the requirements of specifying the exact path, but rely more heavily on the database management systems to detect the least common ancestors of the keywords specified in the query.

Algebraic research has been fruitful in studying, comparing and characterizing fragments of path queries [2, 9], as well as methods and techniques for optimizing and evaluating path queries.

SCIENTIFIC FUNDAMENTALS

Semi-structured data, for example, XML, consists of data entries and containment relationships among the data entries. The data, usually referred to as *a document*, is frequently represented by an ordered node-labeled tree or graph, depending on whether only the containment relationships are treated as first-class relationship, or the id-reference relationships among the data entries are also treated as first-class relationships. The tree representation is more popular :

A *document* D is a 3-tuple (V, Ed, λ), with V the finite set of nodes, $Ed \subseteq V \times V$ a tree of parent-child edges, and $\lambda\colon V \to \mathcal{L}$ a node-labeling function into a countably infinite set of labels $\mathcal{L}$.

In addition, even though not always substantial, the ordering among siblings is usually an important feature for nodes in a *document*. The pre-order among the data entries are called the *document order*. Among others, the pre-order is a dominant approach used to identify data entries in a document, while the document is stored in a database system, relational or native.

The aim of the path query is to express the precise requirement of retrieving a set of data entries that satisfy certain value and structural requirements.

The algebraic form of a path query consists of the primitives of $\emptyset$, ε, $\widehat{l}\,(l \in \mathcal{L})$ for token matching, $\upharpoonleft$ and $\downharpoonleft$ for upward and downward navigation, and operations $\diamond$ for composition of two algebraic expression $E_1 \diamond E_2$, Π_1 and Π_2 for the first and second projection of an algebraic expression, and set operations $\cap$, $\cup$, and $-$. Given a document $D = (N, Ed, \lambda)$ and a path query E, the path semantics of $E(D)$ is a binary relation:

$$\begin{array}{rcl}
\emptyset(D) & = & \emptyset \\
\varepsilon(D) & = & \{(n,n) \,|\, n \in N\} \\
\widehat{l}(D) & = & \{(n,n) \,|\, n \in N \, and \, \lambda(n) = l\} \\
\downharpoonleft (D) & = & Ed \\
\upharpoonleft (D) & = & Ed^{-1} \\
\Pi_1(E)(D) & = & \pi_1(E(D)) \\
\Pi_2(E)(D) & = & \pi_2(E(D)) \\
E_1 \diamond E_2(D) & = & \pi_{1,4}\, \sigma_{2=3}(E_1(D) \times E_2(D)) \\
E_1 \cap E_2(D) & = & E_1(D) \cap E_2(D) \\
E_1 \cup E_2(D) & = & E_1(D) \cup E_2(D) \\
E_1 - E_2(D) & = & E_1(D) - E_2(D)
\end{array}$$

These primitives and operations identify the path of navigation in a document to reach the resultant data entries. The $\upharpoonleft^*$ and $\downharpoonleft^*$ are frequently used to identify the ancestor-descendant relationship among data entries along a path:

$$\begin{array}{rcl}
\downharpoonleft^* (D) & = & \bigcup_{i=0..height(D)} \underbrace{\downharpoonleft \ldots \downharpoonleft}_{i}(D) \\
\upharpoonleft^* (D) & = & \bigcup_{i=0..height(D)} \underbrace{\upharpoonleft \ldots \upharpoonleft}_{i}(D)
\end{array}$$

The result of a path query against document D under the *path semantics* is a set of node pairs whose neighborhood data entries and structures satisfy the query expression.

A localized semantics, also called the *node semantics* of a path query Q is to apply the path expression to a document D and a specific node n_0 in the document, such that $Q(D, n_0) = \{n \,|\, (n_0, n) \in Q(D)\}$. Usually, the results are presented in a list that honors the *document order*.

For example, assuming document D is represented as a tree structure as shown in Figure 1, some sample path queries are evaluated as follows:

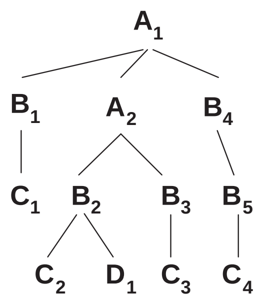

Figure 1: A Sample Semi-structured Document Presented in a Tree Structure (subscripts are used to distinguish data entries with the same label)

Path Query	**Sample Result**
$Q_1 = \downarrow \diamond B$	$Q_1(D) = \{(A_1, B_1), (A_1, B_4), (A_2, B_2), (A_2, B_3), (B_4, B_5)\}$ $Q_1(D, A_1) = \{B_1, B_4\}$
$Q_2 = \Pi_1(\downarrow \diamond \downarrow \diamond C)$	$Q_2(D) = \{(A_1, A_1), (A_2, A_2), (B_4, B_4)\}$ $Q_2(D, A_2) = \{A_2\}$ $Q_2(D, B_4) = \{B_4\}$
$Q_3 = \varepsilon \diamond \Pi_1(\downarrow \diamond \downarrow \diamond C) \diamond \downarrow \diamond B \diamond \downarrow$	$Q_3(D) = \{(A_1, C_1), (A_1, B_5), (A_2, C_2), (A_2, D_1), (A_2, C_3), (B_4, C_4)\}$ $Q_3(D, A_1) = \{C_1, B_5\}$ $Q_3(D, B_4) = \{C_4\}$ $Q_3(D, B_2) = \emptyset$
$Q_4 = \uparrow \diamond \Pi_1(\downarrow^* \diamond D) \diamond \Pi_1(\downarrow \diamond \Pi_1(\downarrow \diamond B))$	$Q_4(D) = \{(B_1, A_1), (A_2, A_1), (B_4, A_1)\}$ $Q_4(D, B_1) = \{A_1\}$
$Q_5 = \varepsilon \diamond \Pi_1((\downarrow \diamond B) \cap (\uparrow \diamond A)) \diamond \downarrow \diamond \downarrow$	$Q_5(D) = \{(A_2, C_2), (A_2, D_1), (A_2, C_3)\}$ $Q_5(D, A_2) = \{C_2, D_1, C_3\}$

Path Query and Pattern Tree Matching
Path Queries are frequently represented as tree structure, called *pattern trees*, in which nodes (optionally labeled) represents the query requirement on the data entries along the path, and edges (labeled with parent-child or ancestor-descendant relationship) represent the structural requirements among the data entries.

The evaluation of a path query is also called the process of *pattern matching*, which is to find the *witness trees* that consist of data entries that satisfy both the node and structural constraint expressed by the pattern tree.

Path Query Language
Path query is the core of various query languages for semi-structured data.

XPath [6] is a W3C standard for addressing part of an XML document and for matching and testing whether a node satisfies a pattern. The primary syntactic construct in XPath is the expression, which is evaluated to yield a node set, a boolean value, a number, or a string. The core of the XPath query language is the path query, which is called *location paths* in XPath. Location path consists of relative location paths and absolute location path. A relative location path consists of a sequence of one or more location steps separated by '/'. A step is composed from left to right and each step selects a set of nodes relative to a context node, which will in turn serve as the context node for the following step. An absolute location path consists of a leading '/', followed optionally by relative location paths.

XPath, in turn, is the core of other XML query languages and transformation languages, such as XSLT [5] and XQuery [4].

Path Query Evaluation
Even though the way in which the document is stored has great impact on how a path query can be evaluated, some common challenges exist in the evaluation of path queries, comparing to their relational peers, and numerous new operators, optimization techniques and index structures have been proposed to facilitate efficient path query evaluation.

One major character of path query is that the query requirements are expressed not only on data entries, such as the tag (label), attributes, and text values of the entries, but also on the structural relationship among the data entries, which are highlighted by the primitives $\uparrow$ (parent axis), $\downarrow$ (child axis), $\uparrow^*$ (ancestor axis), $\downarrow^*$ (descendant axis).

Navigation is a natural approach for evaluating a path query. the nagivational approach scans the whole document to verify the query requirement on data entries and the structural relationship among data entries. This approach works more naturally while the document is stored as file, in object-oriented database, or in a native data store. In addition, the nagivational approach is potentially very expensive in cost, especially when the ancestor/descendent axis is involved.

The invention of structural join operator [1, 13] and multiple algorithms for efficient computing of structural join advanced the evaluation technique of path queries dramatically. A structural join operation takes two sets of data entries as input and returns pairs of data entries that satisfy the desired structural relationship. The basis of the structural join operation is the pre-order and post-order numbering encoding of data entries in the document. The parent-child relationship is a special case of the ancestor-descendant relationship, in which the nodes that satisfies the desired structural relationship have to be exactly one level apart from each other.

$$\begin{aligned}(a,b) \in \downarrow^* (D) &\Leftrightarrow a_{pre} < b_{pre} \wedge a_{post} > b_{post} \\ (a,b) \in \downarrow (D) &\Leftrightarrow a_{pre} < b_{pre} \wedge a_{post} > b_{post} \wedge a_{level} + 1 = b_{level}\end{aligned}$$

The structure join operation and various algorithms that support efficient evaluation of the operation enables the evaluation of path queries by decomposition. This approach partition a path query into twigs that consists of either a node or a pair of nodes with a desired parent-child or ancestor descendant relationship. The matching of the nodes can be easily evaluated via indices that are similar to value indices in RDB. The structural relationships between node pairs are evaluated by the structural join. Holistic approach has been proposed and widely adopted in answering path queries. This approach uses a chain of linked stacks to compute and represent intermediate results of a path in a compactly fashion.

Path Query Optimization
As any database management system, optimization is a critical step in evaluating path queries.

Syntax based optimization rewrites a path query into a path query in normal format, in pursue of minimum expression length, minimum number of predicates, and no redundant value and structural requirements. The rewrite may also aim at decomposing a path query into sub-queries that belong to some fragments of path queries that are simpler to answer.

The cost-based optimization relies on the data statistics of the document to be queried, and the cost-model of the physical operators to be employed, to enumerate various physical evaluation plans and choose one with minimum or acceptable estimate performance. This process involves the study of access method selection, query decomposition, cost estimation, etc.

Indices for Path Query Evaluation
Indexing, being one of the most important ingredients in efficient query evaluation, has seen its importance in the context of XML. Over twenty different types of indices have been proposed and have led to significant improvements in the performance of XML query evaluation.

Indices similar to the ones used in RDBs, namely value indices on element tags, attribute names and text values, are first used, together with the structural join algorithms [3, 10, 1, 13], in XML query evaluation. This approach turns out to be simple and efficient, but is not capable of capturing the structural containment relationships native to the XML data.

To directly capture the structural information of XML data, a family of structural indices has been introduced. DataGuide [7] was the first to be proposed, followed by the 1-index [11], which is based on the notion of bi-simulation among nodes in an XML document. These indices can be used to evaluate some path expressions accurately without accessing the original data graph. Milo and Suciu [11] also introduced the 2-index and T-index, based on similarity of pairs (vectors) of nodes. Unfortunately, these and other early structural indices tend to be too large for practical use because they typically maintain too fine-grained structural information about the document. To remedy this, Kaushik et al. introduced the $A(k)$-index which uses a notion of bi-similarity on nodes relativized to paths of length k [8]. This captures localized structural information of a document, and can support path expressions of length up to k. Focusing just on local similarity, the $A(k)$-index can be substantially smaller than the 1-index and others. Several works have investigated maintenance and tuning of the $A(k)$ indices. The $D(k)$-index and $M(k)$-index extend the $A(k)$-index to adapt to query workload. Yi et al. developed update techniques for the $A(k)$-index and 1-index. Finally, the integrated use of structural and value indices has been explored , and there have also been investigations on covering indices and index selection.

Other directions of XML indexing techniques proposed by researchers include indexing frequent sub-patterns, indexing XML tree and queries as sequences, forward and backward index, HOPI index, XR-tree, and encoding-based indices.

KEY APPLICATIONS

Path query is the core concept behind the query languages for semi-structured data. It is also the foundation of path algebra that are used to represent and reason about queries expressed against semi-structured data, especially those focusing on retrieving fragments of the document that satisfy certain value and structural constraints.

FUTURE DIRECTIONS

Even though the basic idea of the path query has been around for decades, its popularity exploded since XML prevails.

Path query has been adopted as the core of expressing queries again semi-structural data, especially XML. Even though XPath query language has been very stable, new language and language features keep emerging. As to the path query itself, there are on-going study of the sub-language and the characteristics of such language, their relationship to each other, the decidability of the queries in these languages, and the complexity of answer the queries.

On the practical side, systems have been developed to answer path queries. Various query evaluation, query optimization and indexing techniques have been proposed to facilitate efficient evaluation of path queries. However, the level of maturity of these techniques are not at the same level as those of relational queries.

In the relational world, the use cases are well understood and various benchmark have been developed to measure the performance of relational DBMSs. Those of the queries on semi-structured documents are less understood, despite the existence of a few XML benchmarks, such as XMark [12]. In depth research on the usage of path queries and the design and development of benchmarks is yet another promising direction.

DATA SETS

Example semi-structured data and queries can be found in benchmarks such as XMark (http://www.xml-benchmark.org), XMach (http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html), and MBench (http://www.eecs.umich.edu/db/mbench).

CROSS REFERENCES

Semi-structured data models and query languages
XML data management
bi-similarity
XSLT
XQuery

RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.

[1] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Nick Koudas, Divesh Srivastava and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] Michael Benedikt, Wenfei Fan and Gabriel M.Kuper. Structural properties of XPath fragments. In *Theoretical Computer Science*, pp 3-31, v 226, no. 1, 2005.

[3] Nicolas. Bruno, Nick. Koudas and Divesh. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, pages 310–321, 2002.

[4] Don Chamberlin , James Clark , Daniela Florescu, Jonathan Robie, Jerome Simeon and Mugur Stefanescu. XQuery 1.0: An XML query language, May 2003. `http://www.w3.org/TR/xquery`.

[5] James Clark. XSL Transformations (XSLT) version 1.0. `http://www.w3.org/TR/XSLT`.

[6] James Clark and DSteve DeRose. XML path language (XPath) version 1.0. `http://www.w3.org/TR/XPATH`.

[7] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[8] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon and Ehud Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, 2002.

[9] Christoph Koch. Processing queries on tree-structured data efficiently. In *PODS*, page 213-224, 2006.

[10] Jason McHugh and Jennifer Widom. Query optimization for XML. In *VLDB*, pages 315–326, 1999.

[11] Tova Milo and Dan Suciu. Index structures for path expressions. *Lecture Notes in Computer Science*, 1540:277–295, 1999.

[12] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB* 2002. pp 974-985.

[13] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.