Towards Non-Directional XPath Evaluation in a RDBMS

Sourav S Bhowmick†, Curtis Dyreson‡, Erwin Leonard‡, Zhifeng Ng†

†School of Computer Engineering, Nanyang Technological University, Singapore
‡Department of Computer Science, Utah State University, USA
{assourav | ngzh0006 | lerwin}@ntu.edu.sg, curtis.dyreson@usu.edu

ABSTRACT

XML query languages use directional path expressions to locate data in an XML data collection. They are tightly coupled to the structure of a data collection, and can fail when evaluated on the same data in a different structure. This paper extends path expressions with a new non-directional axis called the rank-distance axis. Given a context node and two positive integers α and β, the rank-distance axis returns those nodes that are ranked between α and β in terms of closeness from the context node in any direction.

This paper shows how to evaluate the rank-distance axis in a tree-unaware XML database. A tree-unaware implementation does not invade the database kernel to support XML queries, instead it uses an existing RDBMS such as Microsoft’s SQL server as a back-end and provides a front-end layer to translate XML queries to SQL. This paper presents an overview of an algorithm that translates queries with a rank-distance axis to SQL.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems – Relational databases.


Keywords: XML, non-directional axis, XPath, rank distance, tree-unaware RDBMS.

1. INTRODUCTION

A wealth of existing literature has extensively studied evaluation of various navigational axes in XPath expressions in a relational environment [5]. These well-studied axes are all directional since they locate nodes in a fixed direction relative to a context node (e.g., the descendend axis corresponds to the “down” direction). Unfortunately, queries that rely on directional axes become dependent on the data being in the specified direction, even though data has no “natural” direction and can be organized in different hierarchies. Users who are unfamiliar with a document structure or are knowledgeable about a structure which subsequently changes will sometimes formulate unsatisfiable queries, which are queries that fail to produce desired results. In contrast to incorrect queries, which result in a compilation error, unsatisfiable queries are difficult to debug since they run to completion and produce a result, though not the intended or desired result.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM’09, November 2–6, 2009, Hong Kong, China.
Copyright 2009 ACM 978-1-60558-512-3/09/11 ...$10.00.

Figure 1: Examples of XML data.

As an example of the directional nature of XPath queries, consider the XML document in Figure 1(a) containing league information organized by teams. Each team consists of a set of players. Suppose that a user, Sally, wishes to find the names of teams in the southwest division founded prior to 1970. Sally can issue the following XPath query to retrieve desired information: Q1: //team/division='southwest' and founded<1970]/name. Suppose now that Sally wishes to also find the names of the players for the teams, she can issue another query Q2: //team/division='southwest' and founded<1970]/player/name, to retrieve this information. Finally, the name of the league the teams play for can be retrieved by issuing the following query Q3: //league/name. Note that these three XPath fragments can be combined into a single XPath query using the union operator (Q1|Q2|Q3), or combined in a single XQuery query.

To properly formulate these queries, Sally has to know something about the hierarchical structure of the XML data. For instance, she must know that the player elements are descendants of a team element and information related to the name of a team is available in some part of the team subtree. Furthermore, the name of a league is available in the league subtree. This subtree also includes information related to teams and players. But if Sally misunderstands the structure or if the structure changes over time then this partial knowledge may not be useful anymore for formulating satisfactory queries as demonstrated below.
Assume that the XML document in Figure 1(a) is now reorganized to the structure depicted in Figure 1(b). Now the league information is organized according to players instead of teams. Both documents contain the same data and same element labels but they have different hierarchical relationships. These documents may reflect the scenario where (a) the structure of a document has evolved into another or (b) two different sources represent similar data in different hierarchies. Due to the lack of non-directional axes in XPath, for some queries different path expressions are needed to query each hierarchy. Consequently, some of the above XPath fragments may become unsatisfiable on the document in Figure 1(b). Sally has to formulate a different set of XPath fragments to retrieve relevant information. For instance, Q2 needs to be replaced now with the following query $Q'_2$://player/team/division='southwest' and team/founded='1970'/name.

At first glance, it may seem that the above structural heterogeneity can be addressed by simply appending $Q'_2$ to the XPath query over the document in Figure 1(a) using the union operator. While this approach surely works, it is not a practical solution as it requires a user to be familiar with the structural heterogeneities of different XML documents. This is unrealistic to expect from users as such “structure-awareness” does not scale with increasing structural heterogeneity. Is it possible to retrieve the above information when evaluated on this document? Specifically, in this case the first element is the context node and it will return the names of the players (at distance one). The second element is the test node. By arbitrarily manipulating $\alpha$ and $\beta$ as lower and upper rank, respectively. For example, consider the query $Q_4$://team [founded <'1970']/rank-distance::name[1, 3] on the document in Figure 1(a). It returns the nodes $e_1, e_8, e_{11},$ and $e_{14}$. These nodes contain information related to the name of the league, the names of teams which were founded before 1970, and their players’ names. Similar to traditional XPath axes, the results of rank-distance axis are in document order. Hence, the output of the above example will be [name:NBA, name:Rockets, name:Mutombo, name:Wells, ...]. At first glance it may seem that sorting the results by distance instead of document order is a more appropriate choice. However, introducing a distance-based ordering would impact evaluation of subsequent location steps. Hence we decided to use document order.

Note that $Q_4$ will also return the name element of each of the remaining teams (denoted as $e'$) as its distance from the context node $c_1$ is also three. However, this may not be desirable for certain applications. Fortunately, we can easily filter out $e'$ by post-processing the result set using node type information. Specifically, both $e_1$ and $e'$ have same node type (league/team) but different ranks with respect to $c_1$ (1 and 3, respectively). Hence, for nodes with identical types we can filter out irrelevant nodes by selecting the one with lowest rank ($e_1$) as part of the result set.

Now consider the XML document in Figure 1(b). Although the document structure of Figure 1(b) is different, $Q_4$ returns the above information when evaluated on this document. Specifically, in this case the first element is the context node and it will return the name elements of team, league and players (e.g. Mutombo) as they are ranked 1, 3, and 2, respectively, based on the distance from the context node. Hence, the output in document order will be [name:NBA, name:Mutombo, name:Rockets, ... , name:NBA, name:Wells, name:Rockets,...]. Note that in this case there is no need to post-process the result set based on node types.

Remark. Reconsider the XPath queries in Section 1 over the documents in Figure 1. To retrieve players’ information in Figure 1(a), a query has to navigate down from the team node. On the other hand, in Figure 1(b), the direction of navigation is reversed. Consequently, a key reason for the brittleness of these queries is the directional nature of classical XPath axes. We address this issue in this paper by extending XPath language with a non-directional axis called rank-distance.

Informally, given a context node $c$ and two positive integers $\alpha$ and $\beta$ where $\alpha \leq \beta$, the rank-distance axis returns those nodes that are ranked between $\alpha$ and $\beta$ from the context node based on their “closeness” from $c$. Here “closeness” is measured by the distance from the context node in any direction in the XML tree. Informally, the distance between nodes $u$ and $v$ is the number of edges in the unique, simple undirected path between $u$ and $v$. For example, assume that the team and name nodes in Figure 1(a) are the context and test nodes, respectively. Observe that the name of a team is closest to the team node (at distance one). The second closest node is the name of the league (at distance two). Lastly, the name node(s) that are furthest from the context node are the names of the players (at distance three). Hence, if $\alpha = 1$ and $\beta = 3$ for a rank-distance query $Q$ involving these context and test nodes, then all the above name nodes are part of the answer set. Observe that $Q$ will retrieve the same information when it is evaluated over Figure 1(b) as well. More importantly, a user does not need to be aware of the structural relationship between the context and test nodes. By arbitrarily manipulating $\alpha$ and $\beta$, he/she can retrieve relevant information from a collection of structurally heterogeneous XML documents. The formal definition of the rank-distance axis is given in [7].

The syntax for expressing rank-distance nodes is of the form rank-distance::NodeTest($\alpha$ to $\beta$). We refer to $\alpha$ and $\beta$ as lower and upper rank, respectively. For example, consider the query $Q_2$://team [founded <'1970']/rank-distance::name[1, 3] on the document in Figure 1(a). It returns the nodes $e_1, e_8, e_{11},$ and $e_{14}$. These nodes contain information related to the name of the league, the names of teams which were founded before 1970, and their players’ names. Similar to traditional XPath axes, the results of rank-distance axis are in document order. Hence, the output of the above example will be [name:NBA, name:Rockets, name:Mutombo, name:Wells, ...]. At first glance it may seem that sorting the results by distance instead of document order is a more appropriate choice. However, introducing a distance-based ordering would impact evaluation of subsequent location steps. Hence we decided to use document order.

Note that $Q_4$ will also return the name element of each of the remaining teams (denoted as $e'$) as its distance from the context node $c_1$ is also three. However, this may not be desirable for certain applications. Fortunately, we can easily filter out $e'$ by post-processing the result set using node type information. Specifically, both $e_1$ and $e'$ have same node type (league/team) but different ranks with respect to $c_1$ (1 and 3, respectively). Hence, for nodes with identical types we can filter out irrelevant nodes by selecting the one with lowest rank ($e_1$) as part of the result set.

Now consider the XML document in Figure 1(b). Although the document structure of Figure 1(b) is different, $Q_4$ returns the above information when evaluated on this document. Specifically, in this case the first element is the context node and it will return the name elements of team, league and players (e.g. Mutombo) as they are ranked 1, 3, and 2, respectively, based on the distance from the context node. Hence, the output in document order will be [name:NBA, name:Mutombo, name:Rockets, ... , name:NBA, name:Wells, name:Rockets,...]. Note that in this case there is no need to post-process the result set based on node types.

Remark. Reconsider the XPath queries in Section 1 over the documents in Figure 1. In order to ensure $Q_2$ is satisfiable on the document in Figure 1(b), Sally needs to modify the axis of one or more steps in $Q_2$ or rearrange the labels to satisfy document hierarchy. As mentioned earlier, this requires partial knowledge of the underlying document(s). In contrast, in a rank-distance query a user does not need to undertake such modifications. He/she can explore different results of the query by setting different values for $\alpha$ and $\beta$. Intuitively, this has lesser cognitive overhead as a user does not need to have knowledge of the underlying document structure. In the next section, we shall see that our proposed evaluation strategy supports such exploratory querying by exploiting the previously computed answer set whenever a user modifies $\alpha$ or $\beta$.
an attribute called RValue. Each leaf node \( n \) is associated with four attributes, namely LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum. The LeafOrder captures the document order of a leaf node. The BranchOrder of \( n \) is the level of the nearest common ancestor (NCA) of this node and the preceding leaf node. The DeweyOrderSum is used to encode a node’s order information together with its ancestors’ order information using a single value. Each non-leaf node \( n' \) is implicitly assigned the DeweyOrderSum of the first descendant leaf node. The SiblingSum attribute is introduced to evaluate position predicates with name test. The reader may refer to [3, 11] for detailed description of these attributes and the role they play in XPath evaluation. This encoding information is captured by the schema of SUCXENT++ as shown below. Figure 2 depicts an example of storage of XML representation of league data (Figure 1(a)) in SUCXENT++.

- **Document**: DocID, Name
- **Path**: PathId, PathExp
- **PathValue**: DocID, DeweyOrderSum, PathId, BranchOrder, LeafOrder, SiblingSum, LeafValue
- **Attribute**: DocID, LeafOrder, PathId, LeafValue, AttrOrder
- **DocumentRValue**: (DocID, Level, RValue)

Algorithm 1 depicts the algorithm for SQL query translation. For simplicity, in the sequel we assume that an XPath expression has a single rank-distance axis and parent-child directional axis. Note that our strategy can be extended to expressions containing multiple rank-distance axes and we plan to explore this exhaustively in the future. The algorithm consists of the following phases. We briefly describe them in turn. The reader may refer to [7] for detailed description of the algorithm.

**Phase 1: XPath Decomposition.** In this phase, the algorithm splits the XPath expression \( P \) into two types of XPath components (Algorithm 1, Line 01). One of them represents the XPath fragments that do not contain a rank-distance axis and the other represents the rank-distance axis expressions. For example, consider the expression \( P = /league/team/rank-distance:name[1 to 3] \) over the XML document in Figure 1(a). In this phase, \( P \) is split into the fragments \( E_1 \) and \( E_2 \) representing /league/team and //name, respectively. Note that we transform rank-distance::NodeTest into NodeTest as the rank-distance axis is non-directional and the path of NodeTest cannot yet be determined.

**Phase 2: Directional XPath to SQL Translation.** Next, the algorithm invokes the SQL translation algorithm for the XPath expression without a rank-distance axis in Line 02 in Algorithm 1. Since this algorithm has already been described in [3, 11], we do not elaborate on this further. Here we focus our attention on the translation of the rank-distance axis component.

**Phase 3: NCA Computation.** In Algorithm 1, Line 03 is used to generate an SQL query to determine the level of the NCA using \( S_1, E_1 \), and \( E_2 \) as input. The idea is to find the level of the NCA of the context node (in our example, team node) and the test node (name node). This is achieved by exploiting the DeweyOrderSum, BranchOrder, and RValue attributes of SUCXENT++.

**Phase 4: Ranked Distance Computation.** The next step is to compute the ranked distances between the context and test nodes (Algorithm 1, Line 04). Intuitively, we can compute the distances of only those nodes that have rank at most \( \beta \) based on their distances from the context node. The advantage of this approach is that it is not necessary to compute and rank distances of all test nodes. However, if a user wishes to explore more results by modifying values of \( \alpha \) or \( \beta \) then the algorithm may have to either compute new results incrementally or from scratch. Hence, we first compute distances of all pairs of context and test nodes for the query and rank them. Then, nodes satisfying user-specified \( \alpha \) and \( \beta \) values can be efficiently retrieved using a simple SELECT query (achieved in Phase 5). Note that this approach supports efficient exploratory query evaluation as the ranks of all relevant nodes have been already computed. Also, observe that the algorithm computes distance information of relevant nodes “on-the-fly” for generating the answer set. Consequently, there is no overhead of computing, maintaining, and storing distance information \( O(n^2) \) space complexity a priori.

Next, the algorithm ranks the test nodes based on their distances from the context node (Algorithm 1, Line 05) by invoking the SQL-ForRankDistance function. Here we exploit the ranking function "DENSE_RANK"¹ of an industrial-strength RDBMS.

**Phase 5: SQL Merge.** At this point of time, we have three SQL queries, namely, \( S_2 \), \( S_3 \), and \( S_4 \). The finalTranslatedSQL function combines these SQL queries to generate the final translated query of \( P \) (Algorithm 1, Line 06). We illustrate this procedure using the running example discussed in Phase 1. Figure 3 depicts the final translated SQL query. Lines 01–19 are used to determine the level of the NCA. In particular, Lines 06 and 13 exploit the DeweyOrderSum and RValue values to determine the level of NCA of the context and rank-distance nodes. The MINVAL function (Line 15) is used to compute the minimum value between the level of the context node, and the level of the test nodes, computed by invoking the user-defined SQL function computeLevel. The distances between context nodes and test nodes are computed by Lines 20–24. It first determines the node labels of the context and test nodes and then compute the distance using the user-defined SQL function computeDistance. Lines 25–28 rank the test nodes based on their distance (Phase 4). Lines 29–39 return tuples containing pairs of context nodes and the test nodes satisfying the lower and upper ranks.

The algorithm for Phase 5 includes two optional steps which are common to producing results in real-world queries. First, it extends

---

¹The "DENSE_RANK" is part of the SQL-1999 OLAP amendment.
the query result to include all of the nodes in the subtree rooted at the test nodes by performing an additional join with the PathValue table (denoted as $V$). Second, the context node values are added to the result through another join with the PathValue table (denoted as $U$). Line 38 sorts the result according to the document order. Line 39 enforces the join order option due to performance benefits as highlighted in [6, 11].

4. RELATED WORK

Our objective to flexibly issue XML queries independent of the structure is shared by several recent papers [1, 2, 4, 8]. [4] presents a semantic search engine for XML. The search relies on an inter-connection relationship to decide whether nodes are semantically related. Two nodes are interconnected if and only if the path between them contains no other node that has the same label as the two nodes. [8] proposes a schema-free XQuery, facilitated by a Meaningful Lowest Common Ancestor Structure (MLCAS) operation. Both these techniques are similar to the "closest" relationship between nodes. Unlike rank-distance axis, these approaches do not retrieve nodes based on distances from the context node. Furthermore, these approaches do not leverage on relational technology for structure-independent query evaluation.

Recently, several XML keyword search techniques [9, 10, 12] have been proposed to offer more user-friendly solution for retrieving relevant results. Essentially, these approaches return variants of the subtree rooted at the lowest common ancestor (e.g., VLCA, SLCA) of all the keywords. Due to the lack of expressivity and inherent ambiguity of keyword search, several techniques have been also been developed to infer and retrieve relevant results for a search query [9, 10]. Our work differs from the keyword search paradigm in the following ways. Firstly, we retrieve nodes based on distances from the context node and not the entire LCA-variant of all the keywords. Note that existing keyword search strategies do not exploit node distances for retrieving results. Secondly, as a rank-distance query is an extension of conventional XPath query, it can impose more complex predicates compared to keyword search queries. Furthermore, it does not suffer from expressivity and ambiguity issues similar to keyword search.

More germane to this work is the effort by Zhang and Dyreson [13]. They extended the XPath language with a symmetric locator, called the closest axis, which locates nodes that are closest to a context node. The authors focused on the syntax and semantics of closest axis and showed how the closest axis can be implemented using main-memory and a native XML DBMS. It was shown that this axis can replace many directional steps in path expressions in XML queries. Our work differs from this effort in the following ways. First, rank-distance is a more generic non-directional axis compared to the closest axis. Not only it can find closest node(s) (by setting $\alpha$ and $\beta$ to one) but also nodes that are further away from the context node. Second, in [13] the closest node types are computed prior to query execution and stored in a special index to facilitate closest axis evaluation. In contrast, in our proposed approach the node distances are computed on-the-fly during query execution. Third, closest axis is built on top of a native XML database whereas we show how an industrial-strength RDBMS can be exploited effectively to support a non-directional XPath axis.

5. CONCLUSIONS

In this paper, we present a relational-based strategy to evaluate a non-directional XPath axis, called the rank-distance axis, to locate nodes that are within the specified distance range with respect to a context node. The rank-distance axis is useful for formulating XML queries in an environment where there is insufficient familiarity with an underlying XML document’s structure or changes to the structure. Our scheme is built on top of the SUXENT++ system. It exploits the encoding scheme of SUXENT++ and ranking facility of an off-the-shelf RDBMS to effectively compute the distances between pairs of nodes and rank them in order to compute rank-distance nodes. In this context, we presented an overview of an XPath-to-SQL translation algorithm to translate a rank-distance axis query to its equivalent SQL form.

6. REFERENCES