Non-Directional XPath Processing in a Tree-Unaware RDBMS

Sourav S Bhowmick[†] **Curtis Dyreson**[§] **Erwin Leonardi**[†] **Zhifeng Ng**[†]

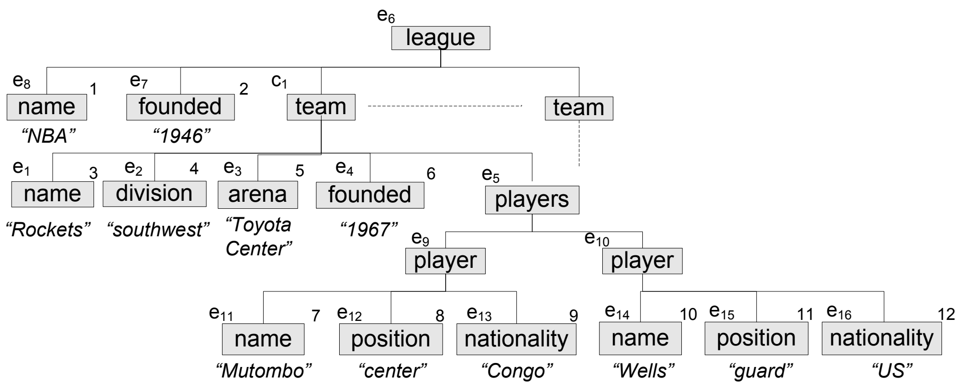
[†]School of Computer Engineering
Nanyang Technological University, Singapore

[§]Department of Computer Science
Utah State University, USA

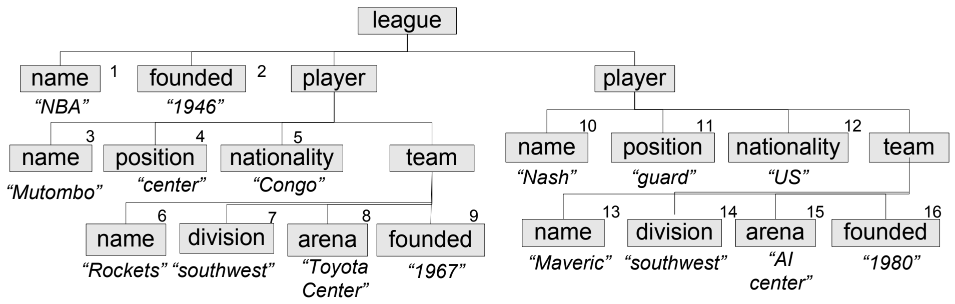
assourav | lerwin | ngzh0006@ntu.edu.sg, curtis.dyreson@usu.edu

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 parameters, 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 algorithm that translates queries with a rank-distance axis to SQL. It also reports on several experiments that demonstrate the efficiency of the query evaluation plan produced by the translation.



(a) XML document



(b) Restructured XML document

Figure 1: Examples of XML data.

1 Introduction

A wealth of existing literature has extensively studied evaluation of various navigational axes in XPath expressions in a relational environment [9]. These well-studied axes are all *directional* since they locate nodes in a fixed direction relative to a context node (*e.g.,* the descendent 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.

1.1 Motivating Example

As an example of the directional nature of XPath queries, consider the XML document in Figure 1(a). It contains 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: Q_1: `//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 Q_2: `//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 Q_3: `/league/name`. Note that these three XPath fragments can be combined into a single XPath query using the `union` operator ($Q_1|Q_2|Q_3$), or combined in a single XQuery.

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 satisfiable 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, Q_2 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 using a single query without being aware of the underlying structural heterogeneities of elements? Ideally, such a query technique should work even if the document structure is reorganized. In order to answer this question affirmatively, in this paper we propose a new non-directional parametric XPath axis called `rank-distance` axis, which enables us to locates elements relative to the context node in *any* direction.

1.2 Current Approaches

The XML community has recognized the difficulty of constructing satisfiable queries and has proposed several solutions. These efforts can be broadly classified into the following three categories.

1. *Inexact query answer:* Techniques have been proposed to find data that inexactly or approximately matches a query [1,2,12,13], relaxing the notion that only strictly satisfying answers be returned by query evaluation [7]. A `rank-distance` axis, like every other axis, is an exact and precise specification, and could be relaxed using the above techniques.
2. *Query correction:* In this approach, similar satisfiable queries are automatically generated when the user query is unsatisfiable [7]. The user can then choose a satisfiable query of interest and receive strictly satisfying results to the query. This method requires the existence of a structural summary (or schema) of the documents in order to generate similar queries. A limitation of this strategy is that the user may have to choose a query from among many potential queries. Requiring further input from a user may be feasible in some interactive sessions, but not for many applications that query a database.
3. *Structure-independent querying:* The above two approaches exploit the underlying document structure to a certain extent. Recently, techniques based on the *Meaningful Lowest Common Ancestor* [14], node interconnection [8], *Smallest Lowest Common Ancestor* (SLCA) [4, 15, 16, 18, 21], and *closest* XPath axis [24] have been proposed to enable formulation and evaluation of queries *independent* of the structure of underlying XML data. This allows a user to query XML data without knowing its exact structure. That is, a user simply needs to know the names of relevant elements and attributes and/or their possible relationships to properly formulate a query.

In this paper, we focus on the last category. Specifically, the XPath language is extended with a *non-directional locator*, called the `rank-distance` axis, to support non-directional exploitation of XML data. The proposed axis allows a user to formulate precise queries knowing only the labels of nodes and unaware of the exact hierarchy.

1.3 Overview

Reconsider the XPath queries in Section 1.1 over the XML 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 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. Here "closeness" is measured by the distance from the context node in *any* direction in the XML tree. 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 most 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 α and β, he/she can retrieve relevant information from a collection of structurally heterogeneous XML documents.

Our proposed algorithm for evaluation of a `rank-distance` axis is built on top of the SUCXENT++ system [5, 17], a *tree-unaware* relational approach designed primarily for query-mostly workloads. Different from other encoding schemes, namely pre-post encoding [10, 23] and Dewey numbering [19], SUCXENT++ uses a novel numbering scheme that only *explicitly* encodes the leaf nodes and the levels of the XML tree. Internal nodes are encoded implicitly. In this paper, we show that this scheme can be effectively used, without any further extension, to evaluate the `rank-distance` axis. Note that this feature is important as queries with non-directional axis should seamlessly blend with conventional XPath processing. Furthermore, our proposed 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*.

In summary, this paper makes three main contributions. Firstly, we extend classical XPath query language with a non-directional `rank-distance` axis in Section 2. Secondly, based on the labeling scheme of SUCXENT++ [17], in Section 3 we present a novel SQL translation algorithm for evaluating queries containing a `rank-distance` axis. Importantly, our proposed algorithm does not require *a priori* computation of distances between all possible pairs of XML nodes and is capable of working with several off-the-shelf RDBMS without any internal modification. *To the best of our knowledge, this is the first concrete implementation of a non-directional XPath axis in a relational environment.* Thirdly, through an extensive experimental study on synthetic and real data sets, in Section 4, we show that our approach can retrieve rank-distance nodes effectively in a tree-unaware RDBMS environment.

2 Rank-Distance Axis

In this section, we first present a data model for XML documents. Then we formally define the `rank-distance` axis.

2.1 XML Data Model

We model XML documents as ordered, labeled trees. We first define such an XML data model.

Definition 1 *A tree is a tuple $(\mathcal{N}, \mathcal{E}, \Sigma, L, F, T, S)$, where*

- *$\mathcal{N}$ is the node set. $r \in \mathcal{N}$ is a special node called the root of the tree,*
- *Let O be the domain of ordinals. Then $\mathcal{E} \subseteq O \times \mathcal{N} \times \mathcal{N}$ is the edge set such that (a) each edge has an ordinal $o_i \in O$ to represent ordering among the children; (b) there is a path between every pair of nodes; (c) there is no cycle among the edges; and (d) every edge has a single incoming edge, except r, which has no incoming edge,*
- *Σ is an alphabet of labels and text values,*
- *$L : \mathcal{N} \rightarrow \Sigma$ is a label function that maps each node to its label,*
- *$F : \mathcal{N} \rightarrow \Sigma \cup \{\varepsilon\}$ is a value function that maps a node to its value, in which $F(n) = \varepsilon$ if node n has an empty value, and*
- *$T : \mathcal{N} \rightarrow S$ is a type function that maps each node to a type, which is a value in the type set S.*

This simple model, which is sufficient for this paper, ignores comments, attributes, processing instructions and namespaces.[1] The model distinguishes between *labels* and *types*. The *label function* maps each node to its label, that is, its element tag. So a <`founded`> node would map to the label `founded`. The *type* function specifies the type of each node, where two nodes with the same label could have different types. The type could be defined in various ways, we assume only that each node has a known type. In schema-validated XML documents, the type is usually specified in the schema. For schema-less documents, the type of a node $n \in \mathcal{N}$ could be defined as the concatenation of the labels on the path from the root to n. For example, suppose that there exist `name` nodes in subtrees rooted at `team` and `player` nodes. Then the path from the root node to a team `name` node and a player `name` node differs; therefore they are of different types.

2.2 Node Distance

Informally, the *distance* between nodes u and v is the number of edges in the unique, simple undirected path between u and v. Formally, the *distance* is defined as follows.

[1]Though the rank-distance axis will locate only elements in this simple model used for expository purposes, in a completely defined data model all kinds of nodes could be in the axis.

Definition 2 *Suppose $(\mathcal{N}, \mathcal{E}, \Sigma, L, F, T, S)$ is a tree and $c \in \mathcal{N}$ is the context node. Let $level(v)$ denotes the level of a node v and $\textsc{nca}(u, v)$ be the nearest common ancestor (*NCA*) of nodes u and v where $u, v \in \mathcal{N}$. Then, the* ***distance*** *between nodes c and v, denoted as $dist(c, v)$, is defined as follows: $dist(c, v) =$*

$$|level(c) - level(\textsc{nca}(c, v))| + |level(v) - level(\textsc{nca}(c, v))|$$

For example, consider the leftmost `team` (c) and `founded` (v) nodes in Figure 1(a). Here $level(c) = level(v) = 2$, $level(\textsc{nca}(c, v)) = 1$. Therefore, $dist(c, v) = |2 - 1| + |2 - 1| = 2$.

2.3 Defining the Axis

Informally, given a context node c and two parameters, α and β where $\alpha \leq \beta$, the `rank-distance` axis returns those nodes that are ranked between α and β from the context node based on their distances from c. Note that the `rank-distance` axis can be thought of as a family of axes in which a specific axis is specified by the parameters α and β. We now formally define the notion of `rank-distance` axis. We first introduce some terminology to facilitate the exposition. Given a context node c, let $P^k_\ell(c)$ be a set of nodes such that $\forall\ n_i \in P^k_\ell(c)$, $L(n_i) = \ell$ and $dist(c, n_i) = k$. A *k-distance node set*, denoted as Ω^k_c, is a set of all nodes having different labels that are k distance away from the c. That is, $\Omega^k_c = P^k_{\ell_1}(c) \cup P^k_{\ell_2}(c) \cup \ldots, \cup P^k_{\ell_m}(c)$. Then, a *k-distance nodeset list*, denoted as $\mathfrak{L}_c$, is a list of all k-distance node sets ordered by their k values in ascending order. That is, $\mathfrak{L}_c = [\Omega^{k_1}_{c,1}, \Omega^{k_2}_{c,2}, \Omega^{k_3}_{c,3}, \ldots, \Omega^{k_{max}}_{c,\rho}]$ where $k_1 < k_2 < \ldots < k_{max}$. The subscript $0 < i \leq \rho$ is called the *rank* of a k-distance node set in $\mathfrak{L}_c$. When the context is obvious, we simply denote them as P^k_ℓ, Ω^k, and $\mathfrak{L}$, respectively. For example, consider the context node c_1 in Figure 1(a). Then, $P^1_{name} = \{e_1\}$ as the label of e_1 is `name` and $dist(c_1, e_1) = 1$. Similarly, $P^1_{founded} = \{e_4\}$ and $P^2_{name} = \{e_2\}$. Then, $\Omega^1 = P^1_{name} \cup P^1_{division} \cup P^1_{arena} \cup P^1_{founded} \cup P^1_{players} \cup P^1_{league} = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, $\Omega^2 = P^2_{name} \cup P^2_{founded} \cup P^2_{player} = \{e_7, e_8, e_9, e_{10}\}$, and $\Omega^3 = P^3_{name} \cup P^3_{position} \cup P^3_{nationality} = \{e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}\}$. Then, $\mathfrak{L}_{c_1} = [\Omega^1_{c_1,1}, \Omega^2_{c_1,2}, \Omega^3_{c_1,3}]$.

Next, we introduce the notion of *selection* on k-distance nodeset list. Given a context node c, a *selection* on $\mathfrak{L}_c$ with respect to label ℓ, denoted as $\sigma(\mathfrak{L}_c, \ell)$, generates a list $\mathfrak{L}_c(\ell) = [\Omega^{k_1}_{c,1}(\ell), \Omega^{k_2}_{c,2}(\ell), \Omega^{k_3}_{c,3}(\ell), \ldots, \Omega^{k_m}_{c,\rho}(\ell)]$ where $\Omega^{k_j}_{c,i}(\ell) \subseteq \Omega^{k_j}_{c,s}$, $\Omega^{k_j}_{c,s} \in \mathfrak{L}_c$, $\forall d \in \Omega^{k_j}_{c,i}(\ell)$, $L(d) = \ell$, and $\forall i\ |\Omega^{k_j}_{c,i}(\ell)| > 0$. That is, the selection returns a set of Ω^k in ranked order where each element contains one or more nodes with label ℓ. For instance, in the above examples $\sigma(\mathfrak{L}_{c_1}, name) =$ where $\Omega^1_{c_1,1}(name) = \{e_1\}$, $\Omega^2_{c_1,2}(name) = \{e_8\}$, $\Omega^3_{c_1,3}(name) = \{e_{11}, e_{14}\}$.

Definition 3 *Suppose $(\mathcal{N}, \mathcal{E}, \Sigma, L, F, T, S)$ is a tree and $c \in \mathcal{N}$ is the context node. Let α and β be two integers such that $\alpha, \beta > 0$, $\alpha \leq \beta$ and β is less than or equal to the maximum distance from a node $n \in \mathcal{N}$ to c. Let $\mathfrak{L}_c = [\Omega^{k_1}_{c,1}, \Omega^{k_2}_{c,2}, \Omega^{k_3}_{c,3}, \ldots, \Omega^{k_v}_{c,\rho}]$*

be the k-distance nodeset list for c. Then, ***rank-distance*** *nodes of c are:* $rank - distance(c, \alpha, \beta) = [n_1, n_2, \ldots, n_j]$ *where*

- $n_1, n_2, \ldots, n_j \in \mathcal{N}$ *and* $j \geq 1$,
- $\forall\ n_i \in rank - distance(c, \alpha, \beta)$, $n_i \in \Omega^{k_a}_{c,\rho_r}$, $\Omega^{k_a}_{c,\rho_r} \in \mathfrak{L}_c$ *where* $0 < a \leq v$, $\rho_r \geq \alpha$ *and* $\rho_r \leq \beta$,
- $\forall\ n \in \mathcal{N}$, $n \notin rank - distance(c, \alpha, \beta)$ *iff* $n \in \Omega^{k_b}_{c,\rho'}$, $\Omega^{k_b}_{c,\rho'} \in \mathfrak{L}_c$ *where* $0 < b \leq v$ *and* $\rho' < \alpha$ *or* $\rho' > \beta$,
- $\forall p, q$, $1 \leq p < q \leq j$, n_p *precedes* n_q *in document order.*

Observe that the `rank-distance` axis is defined as a function that takes a context node and returns a node sequence. The function has two key conditions. First, the *node selection condition* constrains the nodes that appear in the result. It stipulates that the `rank-distance` axis retrieves for each label in the tree the node that is between a pair of specified ranks based on its distance to the context node. Second, the *node ordering condition* states that the result preserves document order. 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.

The syntax of for expressing rank-distance nodes is of the form `rank-distance(`α `to` β`)::NodeTest`. We refer to α and β as *lower* and *upper* rank, respectively. For example, consider the query Q_4: `//team [founded <'1970']/rank-distance(1 to 3)::name` 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. Note that these nodes are in $\Omega^1_1(name)$, $\Omega^2_2(name)$ and $\Omega^3_3(name)$ of $\mathfrak{L}_{c_1}(name)$. That is, $\sigma(\mathfrak{L}_{team}, name) = \{e_1, e_8, e_{11}, e_{14}\}$. Since the results are in document order, the output will be [`name`:*NBA*, `name`:*Rockets*, `name`:*Mutombo*, `name`:*Wells*, ...].

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.name`) 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. Note that we do not incorporate this filtering into the axis by default as its usefulness depends on specific applications as some users may wish to view *all* nodes instead.

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 `team` 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 1.1 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 α and β. 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 α or β.

3 Evaluation of Rank-Distance Axis

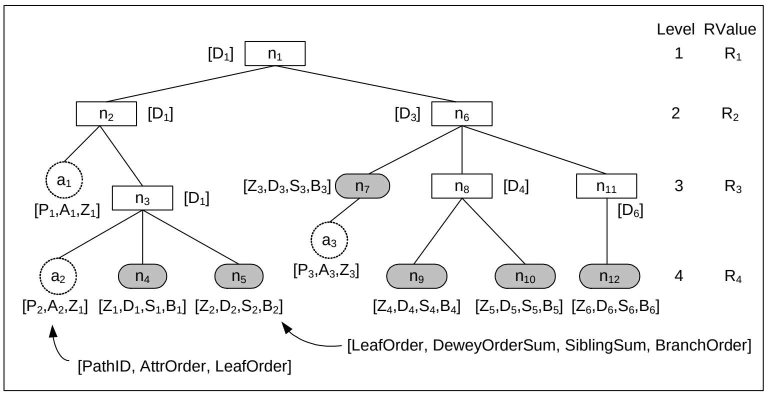
We now formally present the algorithm for translating an XPath expression with a `rank-distance` axis into an SQL query over SUCXENT++ [5, 17]. We begin by first justifying our choice of using SUCXENT++ as the underlying framework and then briefly review its storage scheme.

3.1 Choice of Underlying Framework

Querying XML data over relational framework has gained popularity due to its stability, efficiency, expressiveness, and its wide spread usage in the commercial world. On the one hand, there has been a host of work, *c.f.*, [6], on enabling relational databases to be *tree-aware* by invading the database kernel to support XML. On the other hand, some completely jettison the invasive approach and resort to a *tree-unaware* approach, *c.f.*, [5, 10, 17, 19, 23], where the database kernel is not modified to support XML queries.

Generally, the tree-unaware approach reuses existing code, has a lower cost of implementation, and is more portable since it can be implemented on top of off-the-shelf RDBMSs. This has triggered recent efforts to explore how far we can push the idea of using mature, tree-unaware RDBMS technology to design and build a relational XQuery processor [10, 11]. Furthermore, several commercial RDBMS provide efficient support for ranking (*e.g.,* `DENSE_RANK` in MS SQL server) which we can exploit to *rank* nodes based on their distances. Consequently, we build a `rank-distance` axis evaluation strategy on a tree-unaware relational framework.

Since there are several tree-unaware schemes proposed by the community [5, 9, 10, 17, 19, 23], our selection choice was primarily influenced by the following



Level RValue
1 R1
2 R2
3 R3
4 R4
[D1] n1
n2 [D1]
[D3] n6
a1 [P1,A1,Z1]
n3 [D1]
[Z3,D3,S3,B3] n7
n8 [D4]
n11 [D6]
a3 [P3,A3,Z3]
a2 n4 n5 n9 n10 n12
[P2,A2,Z1] [Z1,D1,S1,B1] [Z2,D2,S2,B2]
[Z4,D4,S4,B4] [Z5,D5,S5,B5] [Z6,D6,S6,B6]
[LeafOrder, DeweyOrderSum, SiblingSum, BranchOrder]
[PathID, AttrOrder, LeafOrder]

Figure 2: Encoding scheme of SUCXENT++.

two criteria. First, the representative storage scheme should not be dependent on the availability of DTD/XML schema. Second, the selected approach must have good query performance for a variety of conventional XPath axes. Since superior query performance of SUCXENT++ system compared to several state-of-the-art tree-unaware schemes have been demonstrated in [5,17], we chose it for our framework.

3.2 SUCXENT++ Schema

We begin by providing an overview of the encoding scheme of SUCXENT++ for XML trees. This scheme does not require a relational back-end to support SQL/XML standard or XML data type. Consider Figure 2. Each level ℓ of an XML tree is associated with an attribute called RValue (denoted as R_ℓ). Each leaf node n is associated with four attributes, namely LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum. Each non-leaf node n' is *implicitly* assigned the DeweyOrderSum of the first descendant leaf node for reasons discussed later. Each attribute node (denoted as a_i) is associated with AttrOrder, LeafOrder of its parent node, and its PathId.

The schema of SUCXENT++ [5, 17] is as follows.

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

Document stores the document identifier DocID and the name Name of a given input XML document D. Each distinct root-to-leaf path appearing in D, namely PathExp, is associated with an identifier PathId and stored in Path table. Essentially each path is a concatenation of the labels of the nodes in the path from the root to the leaf. An example of the Path table containing the root-to-leaf paths of Figure 1

Path

Path ID	PathExp
1	.league#.name#
2	.league#.founded#
3	.league#.team#.name#
4	.league#.team#.division#
5	.league#.team#.arena#
6	.league#.team#.founded#
7	.league#.team#.players#.player#.name#
8	.league#.team#.players#.player#.position#
9	.league#.team#.players#.player#.nationality#

PathValue

DocID	Leaf Order	Branch Order	Path ID	Dewey Order Sum	Sibling Sum	Leaf Value
1	1	0	1	0	0	NBA
1	2	1	2	919	0	1946
1	3	1	3	1838	0	Rocket
1	4	2	4	1839	0	southwest
1	5	2	5	1940	0	Toyota Center
1	6	2	6	1991	0	1967
1	7	2	7	2042	0	Mutombo
1	8	4	8	2043	0	center
1	9	4	9	2044	0	Congo
1	10	3	7	2047	5	Wells
1	11	4	8	2048	5	guard
1	12	4	9	2049	5	US

DocID	Level	RValue
1	1	460
1	2	26
1	3	3
1	4	1

DocumentRValue

DocID	Name
1	NBA.xml

Document

Figure 3: Relations in SUCXENT++.

is shown in Figure 3. Note that '#' is used as a delimiter of steps in the paths instead of '/' for reasons described in [23]. The Attribute table stores the attribute nodes. We do not elaborate on this table as it is beyond the scope of the paper.

For each leaf node n in D, SUCXENT++ creates a tuple in the PathValue table which stores the LeafOrder, BranchOrder, DeweyOrderSum, and SiblingSum values of n. The data value of n is stored in LeafValue. We now elaborate on these attributes. Given two leaf nodes n_1 and n_2, n_1.LeafOrder $<$ n_2.LeafOrder *iff* n_1 precedes n_2 in document order. LeafOrder of the first leaf node in D is 1 and n_2.LeafOrder = n_1.LeafOrder+1 *iff* n_1 is a leaf node immediately preceding n_2. For example, the integer superscript of each leaf node in Figure 1 denotes its LeafOrder value. Given two leaf nodes n_1 and n_2 where n_1.LeafOrder+1 = n_2.LeafOrder, n_2.BranchOrder is the level of the nearest common ancestor (NCA) of n_1 and n_2. For example, the BranchOrder of the `division` leaf node with LeafOrder value 4 in Figure 1(a) is 2 as the NCA of this node and the preceding `name` node is at the second level. Note that the BranchOrder of the first leaf node is 0.

The BranchOrder has an interesting property. Let s be a non-leaf node at level ℓ. Let n_1 be the first descendant leaf node of s. Then, except for n_1, BranchOrder values of all the descendant leaf nodes of s are at least ℓ. The BranchOrder of n_1 is less than ℓ. Observe that the NCA of n_1 and its immediately preceding leaf node is not a descendant of s. For example, consider the non-leaf node n_6 at level 2 in Figure 2. The descendants n_9, n_{10}, and n_{12} all have BranchOrder greater than or equal to 2. However, the BranchOrder of the first descendant n_7 is 1. This property will be exploited later to implicitly encode the non-leaf nodes.

We now introduce the notion of *maximal k-consecutive leaf-node list* which is used to define RValue. Consider a list of consecutive leaf node $\mathcal{S}$: $[n_1, n_2, n_3, \ldots, n_r]$ in D. Let $k \in [1, L_{max}]$ where L_{max} is the largest level of D. Then, $\mathcal{S}$ is called a *k-consecutive leaf-node list* of D *iff* $\forall 0 < i \leq r$ n_i.BranchOrder $\geq k$. $\mathcal{S}$ is called a *maximal k*-consecutive leaf-node list, denoted as M_k, if there does not exist a k-consecutive leaf-node list $\mathcal{S}'$ such that $|\mathcal{S}|<|\mathcal{S}'|$. For example, M_2 in Figure 1(a)

Algorithm 1: The ***RankDistance*** algorithm.

```
Input: XPath $P$
Output: Translated SQL $S_{rd}$
1 $(E_1, E_2) \leftarrow$ decomposeXPath($P$) ;
2 $S_1 \leftarrow$ translate($E_1$) ;
3 $S_2 \leftarrow$ SQLToFindNCA($S_1, E_1, E_2$) ;
4 $S_3 \leftarrow$ SQLToComputeDistance($E_1, E_2$) ;
5 $S_4 \leftarrow$ SQLForRankDistance() ;
6 $S_{rd} \leftarrow$ finalTranslatedSQL($S_2, S_3, S_4$) ;
7 return $S_{rd}$
```

contains nine leaf nodes as $|S| = 9$ for M_2.

The RValue of level ℓ, denoted as R_ℓ, is defined as follows: (i) If $\ell = L_{max} - 1$ then $R_\ell = 1$; (ii) If $0 < \ell < L_{max} - 1$ then $R_\ell = 2R_{\ell+1} \times |M_{\ell+1}| + 1$. For example, consider Figure 1(a). Here $L_{max} = 5$. The values of $|M_2|$, $|M_3|$, $|M_4|$, and $|M_5|$ are 9, 5, 2, and 1, respectively. Then, $R_4 = 1$, $R_3 = 2 \times 1 \times |M_4| + 1 = 5$, $R_2 = 2 \times 5 \times |M_3| + 1 = 51$, and $R_1 = 2 \times 51 \times |M_2| + 1 = 919$. In order to facilitate evaluation of XPath queries, the RValue attribute in DocumentRValue stores $\frac{R_\ell - 1}{2} + 1$ instead of R_ℓ (denoted as R'_ℓ). For instance, in Figure 3 the R_1 is stored as 460 instead of 919.

DeweyOrderSum is used to encode a node's order information together with its ancestors' order information using a single value. Let *parent*(w) denote the parent of an node w. Consider a leaf node n at level ℓ in D. Then, for $1 < k \leq \ell$, Ord(n, k) = i *iff* (i) there exists an node a at level k which is either an ancestor of n or n itself; and (ii) a is the i-th child of *parent*(a). For example, consider the rightmost leaf node in Figure 1(a) (denoted as d). Ord(d, 2) = 3 as the `team` node in the second level is an ancestor of d as well as the third child of the root. Similarly, Ord(d, 3) = 5.

Then DeweyOrderSum of n, n.DeweyOrderSum, is defined as $\sum_{j=2}^{\ell} \Phi(j)$ where $\Phi(j)$=[Ord(n, j)-1]$\times R_{j-1}$. The DeweyOrderSum of the first leaf node is 0. Reconsider the rightmost leaf node again. It has a Dewey path "1.3.5.2.3". DeweyOrderSum of this node is: n.DeweyOrderSum = $(Ord(n,2) - 1) \times R_1 + (Ord(n,3) - 1) \times R_2 + (Ord(n,4) - 1) \times R_3 + (Ord(n,5) - 1) \times R_4 = 2\times919 + 4\times51 + 1\times5 + 2\times1 = 2049$. The DeweyOrderSum of remaining nodes are shown in the DeweyOrderSum attribute of the PathValue table in Figure 3. Note that the DeweyOrderSum is not sufficient to compute position-based predicates with name tests, *e.g.*, `team`[2]. Hence, the SiblingSum attribute is introduced. We do not elaborate on this as it is beyond the scope of the paper.

Comparison of ordering of non-leaf nodes: SUCXENT++'s strategy for comparing the order of non-leaf nodes is based on the following observation. If node n_0 precedes (resp. follows) another node n_1, then descendants of n_0 must also precede (resp. follow) the descendants of n_1. Therefore, instead of comparing the order be-

```
Input: SQL Query S1, XPath E1, XPath E2
Output SQL Query S

C = getMaxPV(S1)
Level = findMaxLevel(E1)
Name = getLabel(E2)
PathIDList = getPathIDs(E2)
R1 = getRValue(1)
S1.addItemIntoFromClause("DOCUMENTRVALUE RX")
S1.addItemIntoFromClause("PATHVALUE V"+(C+1))
S1.addAndCondition("V"+C+".BranchOrder < "+ Level)
S1.addAndCondition("V"+(C+1)+".PathID in ("+ PathIDList +")")
QA = S1
QA.addAndCondition("V"+(C+1)+".DeweyOrderSum BETWEEN
 V"+C+".DeweyOrderSum - CAST(RX.RValue AS BIGINT) + 1 AND "+
 "V"+C+".DeweyOrderSum + CAST(RX.RValue AS BIGINT) - 1")
QA.addAndCondition("V"+(C+1)+".DeweyOrderSum <>
                    V"+C+".DeweyOrderSum")
QA.replaceSelectClause("SELECT V"+C+".DeweyOrderSum , V"+C+".PathID,
V"+C+".BranchOrder, V"+(C+1)+".DeweyOrderSum, "+ "V"+(C+1)+".PathID,
MAX(RX.Level+1) AS NCA_LEVEL")
QA.addGroupBy("GROUP BY V"+C+".DeweyOrderSum , V"+C+".PathID,
V"+C+".BranchOrder, V"+(C+1)+".DeweyOrderSum, V"+(C+1)+".PathID")
QB = S1
QB.addAndCondition("ABS(V"+(C+1)+".DeweyOrderSum -
                    V"+C+".DeweyOrderSum) >= "+ R1)
QB.replaceSelectClause("SELECT V"+C+".DeweyOrderSum , V"+C+".PathID,
V"+C+".BranchOrder, V"+(C+1)+".DeweyOrderSum, "+ "V"+(C+1)+".PathID,
1 AS NCA_LEVEL")
QC = S1
QC.addItemIntoFromClause("PATH P")
QC.addAndCondition("V"+(C+1)+".DeweyOrderSum = V"+C+".DeweyOrderSum)
QC.addAndCondition("V"+(C+1)+".PATHID = P.PATHID)
QC.replaceSelectClause("SELECT V"+C+".DeweyOrderSum , V"+C+".PathID,
V"+C+".BranchOrder, V"+(C+1)+".DeweyOrderSum, "+ "V"+(C+1)+".PathID,
MINVAL("+ Level +", computeLevel('"+ Name +"', P.PATHEXP)) AS NCA_LEVEL")
return QA + " UNION "+ QB + " UNION " + QC
```

Figure 4: The ***SQLToFindNCA*** algorithm (Phase 3).

tween non-leaf nodes, the order between *their descendant leaf nodes* is compared. For this reason, the first descendant leaf node of a non-leaf node n is defined as the *representative leaf node* of n. Here the property of BranchOrder as discussed earlier is exploited to identify the first descendant leaf node. The DeweyOrderSum of the representative leaf node is *conceptually* propagated to its ancestor non-leaf nodes. Figure 2 illustrates this propagation by depicting the DeweyOrderSum values of the non-leaf nodes (shown inside square brackets). Note that these values are not stored explicitly in SUCXENT++ as they can be retrieved from the DeweyOrderSum of representative leaf nodes.

Figure 3 depicts an example of storage of XML representation of league data (Figure 1(a)) in SUCXENT++. Note that large text content (*e.g.,* protein sequence) is stored in a separate relation called TextContent that has same schema as PathValue.

Computation of NCA. We now briefly discuss how to compute the NCA of two nodes efficiently in SUCXENT++ using the following theorem.

Theorem 1 *Let n_1 and n_2 be two distinct leaf nodes in an* XML *tree and $\ell > 0$. If* $\frac{R_{\ell+1}-1}{2} + 1 \leq |n_1.\text{DeweyOrderSum} - n_2.\text{DeweyOrderSum}| < \frac{R_{\ell}-1}{2} + 1$ *then the level of the* NCA *of n_1 and n_2 is $\ell + 1$.* □

The reader may refer to [5] for the proof. Consider the last leaf node in Figure 1(a). The DeweyOrderSum of this node is 2049. Let X be the DeweyOrderSum of leaf nodes that have NCA at level 3. Using the above theorem, X falls within the following range: $(R_3-1)/2+1 \leq |X-2049| < (R_2-1)/2+1 \Rightarrow 3 \leq |X-2049| < 26$ which returns the 7th, 8th, and 9th leaf nodes (DeweyOrderSums are 2042, 2043, and 2044, respectively). Note that Theorem 1 can also be used for internal nodes as SUCXENT++ represents each internal node with its first descendant leaf node.

Corollary 1 *Let n_1 and n_2 be two distinct leaf nodes in an* XML *tree. If* $|n_1.\text{DeweyOrderSum} - n_2.\text{DeweyOrderSum}| \geq \frac{R_1-1}{2} + 1$ *then the* NCA *of n_1 and n_2 is the root node.* □

For example, consider the leaf nodes with LeafOrder values 2 and 4 in Figure 1(a). The DeweyOrderSums of these nodes are 919 and 1839, respectively. Since 1839 – 919 > 460, the NCA is the root node (`league`).

Note that the above theorem and corollary involve non-identical nodes. When the pair of nodes are identical, then the NCA is computed as follows. (a) If n_1 and n_2 are non-leaf nodes and their representative leaf nodes are identical, then the level of the NCA of n_1 and n_2 is $MIN(level(n_1), level(n_2))$. (b) Suppose that n_1 is a non-leaf node and n_2 is a leaf node. If the representative leaf node of n_1 is identical to n_2, then the level of the NCA of these nodes is the level of n_1. (c) If n_1 and n_2 are identical leaf nodes then the NCA level is the level of n_1 or n_2.

3.3 SQL Translation Algorithm

Algorithm 1 depicts the algorithm for SQL query translation. For simplicity, 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.

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(1 to 3)::name` 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. As we shall see later, the DeweyOrderSum and RValue attributes enable us to efficiently prune `NodeTest` nodes that are not "close" to the context node.

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 [5, 17], we do not elaborate on this further. Here we focus our attention on the translation of the `rank-distance` axis component. Figure 6(a) depicts the translated SQL query of E_1 (denoted as S_1).

Phase 3: NCA Computation. Recall that the distance computation between nodes u and v requires the level of NCA(u,v). In Algorithm 1, Line 03 is used to generate an SQL query to determine the level of 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). Figure 4 depicts the *SQLToFindNCA* algorithm in detail. First, it computes the number of instances of the PathValue table in the query S_1 (denoted as C). In our example, $C = 1$. In the next step, it finds the level of the context node by analyzing E_1. In our example, the context node is `team` whose level is 2. Next, the algorithm fetches the names and ids of paths that satisfy E_2 (Lines 03-04 in Figure 4). The RValue of level 1 is fetched in Line 05. Lines 06–07 in Figure 4 are used to add two additional instances of

```
Input: XPath P1, XPath P2
Output SQL Query S

Name1 = getLabel(P1)
Name2 = getLabel(P2)
S.set("SELECT B.V1DeweyOrderSum, B.V1PathID,
    B.V1BranchOrder, B.V2DeweyOrderSum, computeLevel('" + Name2
    + "', P2.PATHEXP), "+ "   computeDistance(B.NCALEVEL, '" +
    Name2 + "', P2.PATHEXP, '" + Name1 + "', P1.PATHEXP) AS
    DISTANCE "+ "FROM [S2] B, PATH P2 , PATH P1 "+ "WHERE
    B.V2PATHID = P2.PATHID AND B.V1PATHID = P1.PATHID");
return S
```

(a) The *SQLToComputeDistance* Algorithm

```
Input: NCA Level NLevel
       Closest node test name Name1
       Context node name Name2
       Path Expression of candidate closest node P1
       Path Expression of context node P2

Output: Distance between context and candidate  closest nodes D

P1Level = computeLevel(Name1, P1)
P2Level = computeLevel(Name2, P2)
D = ABS(P1Level - NLevel) + ABS(P2Level - NLevel)
return D
```

(b) The compute*Distance* Algorithm

```
Input: -
Output: S4

S4="SELECT C.V1DeweyOrderSum, C.V2DeweyOrderSum, C.DISTANCE, "+
      "  DENSE_RANK() OVER(PARTITION BY  C.V1DeweyOrderSum "+
      "                    ORDER BY C.DISTANCE) "+
      "FROM [S3] C ";
return S4
```

(c) The *SQLForRankDistance* Algorithm

Figure 5: Algorithms for Phase 4.

```
SELECT V1.PATHID, V1.DEWEYORDERSUM,
       V1.LEAFVALUE
FROM PATHVALUE V1
WHERE V1.PATHID IN (5,4,6,3,7,9,8)
```

(a) SQL query for Phase 2

```
SELECT B.V1DEWEYORDERSUM, B.V1PATHID, B.V1BRANCHORDER,
   B.V2DEWEYORDERSUM, computeLevel('.name#', P1.PATHEXP),
   computeDistance(B.NCALEVEL, '.name#', P1.PATHEXP, '.team#',
                   P2.PATHEXP) AS DISTANCE
FROM [S2] B, PATH P2 , PATH P1
WHERE B.V2PATHID = P2.PATHID AND B.V1PATHID = P1.PATHID
```

(c) SQL query for distance computation (Phase 4)

```
SELECT V1.DEWEYORDERSUM, V1.PATHID, V1.BRANCHORDER,
V2.DEWEYORDERSUM, V2.PATHID, MAX(RX.LEVEL+1) AS NCA_LEVEL
FROM DOCUMENTRVALUE RX, PATHVALUE V1, PATHVALUE V2
WHERE V1.PATHID IN (5,4,6,3,7,9,8)
 AND V1.BRANCHORDER < 2
  AND V2.PATHID IN (1,3,7)
  AND V2.DEWEYORDERSUM BETWEEN V1.DEWEYORDERSUM -
      CAST(RX.RVALUE AS BIGINT) + 1
 AND V1.DEWEYORDERSUM + CAST(RX.RVALUE AS BIGINT) - 1
  AND V1.DEWEYORDERSUM <> V2.DEWEYORDERSUM
GROUP BY V1.DEWEYORDERSUM, V1.PATHID, V1.BRANCHORDER,
      V2.DEWEYORDERSUM, V2.PATHID
UNION
SELECT DISTINCT V1.DEWEYORDERSUM, V1.PATHID,
V1.BRANCHORDER, V2.DEWEYORDERSUM, V2.PATHID, 1 AS NCA_LEVEL
FROM PATHVALUE V1, PATHVALUE V2
WHERE V1.PATHID IN (5,4,6,3,7,9,8) AND V1.BRANCHORDER < 2
  AND V2.PATHID IN (1,3,7)
  AND ABS(V1.DEWEYORDERSUM - V2.DEWEYORDERSUM) >= 460
UNION
SELECT DISTINCT V1.DEWEYORDERSUM, V1.PATHID,
V1.BRANCHORDER, V2.DEWEYORDERSUM, V2.PATHID, 2 AS NCA_LEVEL
FROM PATHVALUE V1, PATHVALUE V2
WHERE V1.PATHID IN (5,4,6,3,7,9,8) AND V1.BRANCHORDER < 2
  AND V2.PATHID IN (1,3,7)
  AND V1.DEWEYORDERSUM = V2.DEWEYORDERSUM
```

(b) SQL query for NCA computation (Phase 3)

Figure 6: SQL queries generated by Phases 1 - 3 for the running example.

the DocumentRValue (denoted as R_X) and PathValue (denoted as V_{C+1}) tables into the `FROM` clause of S_1. Line 08 ensures that only representative leaf nodes of the context nodes are used in the comparison. Line 09 adds the path ids that satisfy E_2 (the node test of `rank-distance` axis).

Next, the algorithm generates three different SQL queries based on S_1, namely Q_A, Q_B, and Q_C, to handle different scenarios of NCA computation. Q_A computes the level of NCA(u,v) if nodes u and v are distinct. Q_B is used when NCA(u,v) is the root node. If nodes u and v are identical, then Q_C is used. Lines 11-12 are used to add two additional conditions to Q_A. Line 11 is used to determine the level of NCA of the context and rank-distance nodes based on Theorem 1. Line 12 ensures that the NCA between two non-distinct leaf nodes (the context nodes is identical to the rank-distance node) is not computed. Then, the algorithm replaces the `SELECT` clause and adds the `GROUP BY` function (Lines 13 and 14, respectively). For Q_B, the algorithm adds an additional condition based on Corollary 1 to ensure that the context and rank-distance nodes that have NCA at level 1 are only used in the comparison here (Line 16). Lines 19-22 are used to complete the construction of Q_C.The algorithm adds two conditions in the `WHERE` clause (Lines 20-21) and replaces the `SELECT` clause (Line 22). In particular, the `MINVAL` function is used to computer the minimum value between the level of the context node, and the level of the test nodes, computed by invoking the *computeLevel* function. We shall elaborate on the *computeLevel* function in the next phase. Finally, Line 23 combines Q_A, Q_B, and Q_C using the `UNION` operator and returns the generated query. In our example, the invocation of the *generateSQLToFindNCA* algorithm will generate the SQL query shown in Figure 6(b).

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 β 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 α or β 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 α and β 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.

The *SQLToComputeDistance* algorithm depicted in Figure 5(a) generates an SQL query S_3 for computing the distance between the context nodes and the test nodes. First, the algorithm determines the node labels of the context and test nodes (Lines 01–02 in Figure 5(a)). Then, these labels are used in a query template as shown in Line 03. Finally, the algorithm returns the generated SQL query. In our example, Figure 6(c) depicts the SQL query generated due to invocation of the *SQLToComputeDistance* algorithm. Note that `[S2]` in Line 02 of Figure 6(c) refers to the translated SQL query returned by Phase 3 (Algorithm 1, Line 03).

Note that the user-defined SQL function *computeDistance* (Figure 6(c), Line 01) is used to compute the distance. Figure 5(b) shows the details of this function. Let us elaborate on it with an example. Suppose that the `team` and right-most `name` nodes in Figure 1(a) are candidate context and test nodes, respectively. The inputs to the *computeDistance* function are NCA levels of these nodes (denoted as *NLevel*), labels of the nodes denoted as *Name*1 and *Name*2, and path expressions of the test and context nodes denoted as $P1$ and $P2$, respectively. Here *Name*1 = `".name#"`, *Name*2 = `".team#"`, $P1$ = `".league#.team#.players#.player#.name#"`, and $P2$ =`".league#.team#.name#"`. Observe that as `team` is a non-leaf node, it is represented by its left-most descendant leaf node (`name` node). The first step in the *computeDistance* function (Lines 01-02, Figure 5(b)) is to find the levels of `name` and `team` nodes in $P1$ (denoted as $P1Level$) and $P2$ (denoted as $P2Level$), respectively. Here $P1Level = 5$ and $P2Level = 2$. Next, the function computes the distance $D = |5 - 2| + |2 - 2| = 3$ (Line 03, Figure 5(b)). In addition, we have to compute the level of rank-distance node test by invoking a user-defined SQL function *computeLevel* (Figure 6(c), Line 1). The level of rank-distance node test is used for retrieving the final result of the translated SQL query (Figure 6(d), Line 24). The intuition behind this function is straight-forward. It computes the level of node n in the given path expression P. In our example, as n = `".name#"` and P = `".league#.team#.players#.player#.name#"`, the function returns 5.

Finally, the algorithm ranks the test nodes based on their distances from the context node (Algorithm 1, Line 05) by invoking the *SQLForRankDistance* function. It returns the SQL query S_4 as depicted in Line 01 (Figure 5(c)). Here we exploit the ranking function `"DENSE_RANK"`[2] of an industrial-strength RDBMS. Note that it ranks the result set, without any gaps in the ranking.

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 (Figure 7) combines these SQL queries to generate the final translated query of P (Algorithm 1, Line 06). We illustrate this procedure using the running example. Figure 8 depicts the final translated SQL query. Lines 01–19 in Figure 8 are used to determine the level of the NCA. The distances between context nodes and test nodes are computed by Lines 20–24. Lines 25–28 rank the test nodes based on their distance (Phase 4). Lines 29-39 re-

[2]The `"DENSE_RANK"` is part of the SQL:1999 OLAP amendment.

```
Input: SQL Queries S2, S3, S4
Output: SQL Query S

01  S = "WITH S2 (V1DeweyOrderSum, V1PathID, V1.BranchOrder, "+
        "     V2DeweyOrderSum, V2PathID, NCALEVEL) AS "+
        " ( " + S2 +" ), "+
        "     S3 (V1DeweyOrderSum, V1PathID, V1BranchOrder, "+
        "         V2DeweyOrderSum, V2Level, Distance) AS "+
        " ( " + S3 +" ), "+
        "     S4 (V1DeweyOrderSum, V2DeweyOrderSum Distance, Rank) AS "+
        " ( " + S4 +" ) "+
        "SELECT DISTINCT C.V1DeweyOrderSum, U.LeafValue, V.* "+
        "FROM [S4] X, [S3] C, PathValue U, DocumentRValue R, "+
        "     PathValue V "+
        "WHERE C.V1DeweyOrderSum = X.V1DeweyOrderSum "+
        "  AND C.V2DeweyOrderSum = X.V2DeweyOrderSum "+
        "  AND C.Distance = X.Distance "+
        "  AND X.RANK BETWEEN α AND β "+
        "  AND U.DeweyOrderSum = C.V1DeweyOrderSum "+
        "  AND U.PathID = C.V1PathID "+
        "  AND U.BranchOrder = C.V1BranchOrder "+
        "  AND R.Level = (C.V2Level - 1) "+
        "  AND V.DeweyOrderSum BETWEEN "+
        "         C.V2DeweyOrderSum – CAST(R.RVALUE AS BIGINT) + 1 "+
        "     AND C.V2DeweyOrderSum + CAST(R.RVALUE AS BIGINT) - 1 "+
        "ORDER BY C.V1DeweyOrderSum, V.DeweyOrderSum "+
        "OPTION (FORCE ORDER) ";
02  return S
```

Figure 7: The *finalTranslatedSQL* algorithm (Phase 5).

turn tuples containing pairs of context nodes and the test nodes satisfying the lower and upper distance ranks. The algorithm in Figure 7 includes two optional steps which are common to producing results in real-world queries. First, it extends 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 [10, 17].

3.4 Temporary Tables-Based SQL Translation

Although the above algorithm can identify rank-distance nodes accurately, we observe that the subqueries of the translated SQL query in Figure 8 are executed more than once. For instance, S_2 is executed two times. It is referred by S_3 (Line 22) and S_3 itself is referred two times in Lines 27 and 30. In fact, as we shall see later, S_2 is often the most expensive subquery. Therefore, multiple execution of S_2 can adversely affect the performance of the `rank-distance` axis evaluation.

```
01 WITH S2 (V1DEWEYORDERSUM, V1PATHID, V1.BRANCHORDER, V2DEWEYORDERSUM, V2PATHID, NCALEVEL) AS (
02    SELECT V1.DEWEYORDERSUM, V1.PATHID, V1.BRANCHORDER, V2.DEWEYORDERSUM, V2.PATHID,
             MAX(RX.LEVEL+1) AS NCA_LEVEL
03    FROM DOCUMENTRVALUE RX, PATHVALUE V1, PATHVALUE V2
04    WHERE V1.PATHID IN (5,4,6,3,7,9,8) AND V1.BRANCHORDER < 2
05      AND V2.PATHID IN (1,3,7)
06      AND V2.DEWEYORDERSUM BETWEEN V1.DEWEYORDERSUM - CAST(RX.RVALUE AS BIGINT) + 1
                                 AND V1.DEWEYORDERSUM + CAST(RX.RVALUE AS BIGINT) - 1
07      AND V1.DEWEYORDERSUM <> V2.DEWEYORDERSUM
08    GROUP BY V1.DEWEYORDERSUM, V1.PATHID, V1.BRANCHORDER, V2.DEWEYORDERSUM, V2.PATHID
09    UNION
10    SELECT DISTINCT V1.DEWEYORDERSUM, V1.PATHID, V1.BRANCHORDER, V2.DEWEYORDERSUM, V2.PATHID, 1
          AS NCA_LEVEL
11    FROM PATHVALUE V1, PATHVALUE V2
12    WHERE V1.PATHID IN (5,4,6,3,7,9,8) AND V1.BRANCHORDER < 2
13      AND V2.PATHID IN (1,3,7) AND ABS(V1.DEWEYORDERSUM - V2.DEWEYORDERSUM) >= 460
14    UNION
15    SELECT DISTINCT V1.DEWEYORDERSUM, V1.PATHID, V1.BRANCHORDER, V2.DEWEYORDERSUM,
                      V2.PATHID, MINVAL(2, computeLevel('.name#', P.PATHEXP))  AS NCA_LEVEL
16    FROM PATHVALUE V1, PATHVALUE V2, PATH P
17    WHERE V1.PATHID IN (5,4,6,3,7,9,8) AND V1.BRANCHORDER < 2 AND V2.PATHID IN (1,3,7)
18      AND V2.PATHID = P.PATHID AND V1.DEWEYORDERSUM = V2.DEWEYORDERSUM
19 ),
```

```
20 S3 (V1DEWEYORDERSUM, V1PATHID, V1BRANCHORDER, V2DEWEYORDERSUM, V2LEVEL, DISTANCE) AS (
21    SELECT B.V1DEWEYORDERSUM, B.V1PATHID, B.V1BRANCHORDER, B.V2DEWEYORDERSUM,
         computeLevel('.name#', P1.PATHEXP), computeDistance(B.NCALEVEL,
         '.name#', P1.PATHEXP, '.team#', P2.PATHEXP) AS DISTANCE
22    FROM [S2] B, PATH P2 , PATH P1
23    WHERE B.V2PATHID = P2.PATHID AND B.V1PATHID = P1.PATHID
24 ),
25 S4 (V1DeweyOrderSum, V2DeweyOrderSum, DISTANCE, RANK) AS (
26   SELECT C.V1DeweyOrderSum, C.V2DeweyOrderSum, C.DISTANCE,
            DENSE_RANK() OVER(PARTITION BY C.V1DeweyOrderSum ORDER BY C.DISTANCE)
27   FROM [S3] C
28 )
29 SELECT DISTINCT C.V1DeweyOrderSum, U.LeafVALUE, V.*
30 FROM [S4] X, [S3] C,  PATHVALUE U, DOCUMENTRVALUE R, PATHVALUE V
31 WHERE C.V1DeweyOrderSum = X.V1DeweyOrderSum AND C.DISTANCE = X.DISTANCE
32   AND C.V2DeweyOrderSum = X.V2DeweyOrderSum
33   AND U.DEWEYORDERSUM = C.V1DeweyOrderSum AND U.BRANCHORDER = C.V1BRANCHORDER
34   AND U.PATHID = C.V1PATHID AND V.PATHID = C.V2PATHID
35   AND R.LEVEL = (C.V2LEVEL - 1)
36   AND X.RANK BETWEEN 1 AND 3
37   AND V.DeweyOrderSum BETWEEN C.V2DeweyOrderSum - CAST(R.RVALUE AS BIGINT) + 1
                             AND C.V2DeweyOrderSum + CAST(R.RVALUE AS BIGINT) - 1
38 ORDER BY C.V1DeweyOrderSum, V.DEWEYORDERSUM
39 OPTION (FORCE ORDER)
```

Figure 8: Final SQL query generated by Algorithm 1.

```
S2 (V1DeweyOrderSum, V2DeweyOrderSum,
V1PathID, V1.BranchOrder, V2PathID,
NCALevel)
S3 (V1DeweyOrderSum, V2DeweyOrderSum,
V1PathID, BranchOrder, V2Level,
Distance)
S4 (V1DeweyOrderSum,V2DeweyOrderSum,
Distance, Rank)
```

(a) Schema of temporary tables

```
01  INSERT INTO S2
02    … … … (Lines 02-18, Figure 8) … …
03  OPTION (FORCE ORDER);
04  INSERT INTO S3
05    … … … (Lines 21-23, Figure 8) …
06  OPTION (FORCE ORDER) ;
07  INSERT INTO S4
08    … … … (Lines 26-27, Figure 8) … …
09  ;
10    … … … (Lines 29-39, Figure 8) … …
```

(b) Temporary table-based SQL query

Figure 9: Schema of temporary tables and SQL statement.

In order to address this issue, we present a translation strategy that exploits materialized indexed temporary tables (instead of using `WITH` clause) for storing the results of the subqueries and reusing them whenever necessary. This is achieved by replacing the `WITH` statement used in the query by an `INSERT` statement for each subquery. Due to space constraints, we do not present the formal algorithm here. Rather, we illustrate the translated SQL using an example based on the SQL query depicted in Figure 8. Observe that this query has three subqueries: S_2, S_3, and S_4. Consequently, the algorithm creates temporary tables for storing the results of S_2, S_3, and S_4. The schemas of these temporary tables are shown in Figure 9(a). Figure 9(b) depicts the modified SQL query based on temporary tables. Observe that the `WITH` clause in Figure 8 defines three *common table expressions* (CTE)[3]. These CTEs are replaced by a set of `INSERT` statements (Lines 01, 04, and 07). Also, S_2 is only referenced once. Note that these SQL subqueries must be executed sequentially.

4 Performance Evaluation

We have implemented the `rank-distance` axis support on SUCXENT++ in Java JDK1.6. The experiments were conducted on an Intel Core2 Duo E6550 2.33GHz

[3]A common table expression (CTE) can be thought of as a temporary result set that is defined within the execution scope of a single `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `CREATE VIEW` statement.

ID	Dataset	File Size (MB)	Number of Nodes	Depth
DC10	XBench	10	225,234	8
DC100	XBench	100	2,242,200	8
DC1000	XBench	1,000	22,442,612	8

ID	Dataset	File Size (MB)	Number of Nodes	Depth
U28	Uniprot	28	574,738	5
U284	Uniprot	284	5,733,977	5
U2843	Uniprot	2,843	57,368,191	5

(a) Data Sets

ID	Dataset	Query
XQ1	XBench	/catalog/item[position()=1 to 100]/rank-distance(1 to 1)::name_of_city
XQ2		/catalog/item[position()=1 to 100]/rank-distance(2 to 2)::name_of_city
XQ3		/catalog/item[position()=1 to 100]/rank-distance(1 to 2)::name_of_city
XQ4		/catalog/item/subject[text()='SUBJECT_2']/rank-distance(1 to 1)::phone_number
XQ5		/catalog/item/subject[text()='SUBJECT_2']/rank-distance(2 to 2)::phone_number
XQ6		/catalog/item/subject[text()='SUBJECT_2']/rank-distance(1 to 2)::phone_number
UQ1	Uniprot/KB	/uniprot/entry[sequence='MAAA%']/rank-distance(1 to 1)::name
UQ2		/uniprot/entry[sequence='MAAA%']/rank-distance(2 to 2)::name
UQ3		/uniprot/entry[sequence='MAAA%']/rank-distance(3 to 3)::name
UQ4		/uniprot/entry[sequence='MAAA%']/rank-distance(1 to 2)::name
UQ5		/uniprot/entry[sequence='MAAA%']/rank-distance(1 to 4)::name

(b) Query Set

Non-Unique Clustered Index: (PathID, BrancOrder)

Unique Non-Clustered Index: (DocID, PathID, DeweyOrderSum)

Non-Unique Non-Clustered Indexes
(PathID, SiblingSum)
(PathID, DeweyOrderSum, BranchOrder, SiblingSum) include (LeafValue)
(PathID, DeweyOrderSum)

(c) Indexes on The PathValue Table

Figure 10: Data and query sets.

processor machine running on Windows XP SP3 with 3.25GB of RAM. The RDBMS used was Microsoft SQL Server 2005 Developer Edition. We denote the SQL translation techniques described in Sections 3.3 and 3.4 as SX-WITH and SX-TEMP, respectively.

Data and Query Sets. We use XBench DCSD [22] as a synthetic data set and Uniprot/KB XML[4] as a real-world data set. We vary the size of XML documents from 10MB to 1GB for XBench DCSD data set and from 28MB to 2.77GB for the Uniprot/KB data set. Figure 10(a) depicts the characteristics of the data sets. Figure 10(b) depicts the query sets for XBench DCSD (XQ1–XQ6) and Uniprot/KB data sets (UQ1–UQ5).

Test Methodology. Appropriate indexes were constructed for SUCXENT++ (Figure 10(c)). Furthermore, since our data set consists of a single XML document, we removed the DocID column from the tables in SUCXENT++. For all queries, we return entire subtrees of the matched test nodes (optional) as it gives us the upper bound of query performance. Prior to our experiments, we ensured that statistics had been collected. The bufferpool of the RDBMS was cleared before each run. Each query was executed six times and the results from the first run were always discarded.

4.1 Query Evaluation Times

Figure 11 depicts the query evaluation times of our benchmark query set. We can make the following observations. First, query evaluation cost increases with the number of nodes for all queries. Second, temporary tables-based strategy improves the query performance in all cases. In particular, the performance gain increases with the size of the data set (highest observed factor being 3.8). Next, we shall justify why SX-TEMP performs consistently better than SX-WITH by analyzing the performance of subqueries.

Evaluation times of sub queries. In this set of experiments, we shall show the evaluation time of each subquery executed in our approach. Recall that SX-TEMP is proposed because in SX-WITH S_2 is executed multiple times. We fix the lower and upper ranks to 1 ($\alpha = \beta = 1$) and use the queries in Figure 12(a) to study the evaluation times of the subqueries. Recall that although we set α and β to

[4]Downloaded from ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/ knowledgebase/complete/uniprot_sprot.xml.gz.

Query	DC10			DC100			DC1000		
	# of Rows	SX-WITH	SX-TEMP	# of Rows	SX-WITH	SX-TEMP	# of Rows	SX-WITH	SX-TEMP
XQ1	100	613.60	463.88	100	810.00	724.42	100	13,698.70	3,613.22
XQ2	258	723.42	499.96	249	858.90	767.46	266	13,804.30	3,557.16
XQ3	358	921.62	513.30	349	1,442.52	759.84	366	13,935.64	3,630.86
XQ4	50	294.68	229.52	50	1,240.64	685.36	50	13,969.66	3,853.34
XQ5	131	364.56	259.14	131	1,481.82	704.44	146	14,136.02	3,840.22
XQ6	181	450.18	270.00	181	1,788.24	699.80	196	14,126.70	3,890.38

(a) Query Evaluation Time: XBench

Query	U28			U284			U2843		
	# of Rows	SX-WITH	SX-TEMP	# of Rows	SX-WITH	SX-TEMP	# of Rows	SX-WITH	SX-TEMP
UQ1	11	951.54	707.06	93	5,426.92	3,512.84	984	49,190.16	30,468.44
UQ2	69	974.56	690.90	581	5,262.44	3,287.62	6309	52,807.78	29,754.80
UQ3	8	883.16	663.20	81	4,517.94	2,677.98	596	44,367.16	22,275.94
UQ4	80	1,048.36	716.48	674	5,451.60	3,318.00	7293	55,295.72	29,918.46
UQ5	88	1,071.40	762.82	755	5,842.64	3,338.54	7889	59,575.76	29,693.50

(b) Query Evaluation Time: Uniprot/KB

Figure 11: Query evaluation times (in msec.).

1, our approach already computes the distance between *all* context and test nodes (Phase 4). Figures 12(b)-(c) report the evaluation times of the subqueries on the largest benchmark data sets. Note that S_5 denotes the subquery used to filter nodes based on user-specified lower and upper ranks and retrieve the final results (*e.g.,* Lines 29–39 in Figure 8). The highest observed percentage of the contribution of S_2 to the overall query evaluation time is 97.9% (query XQ10). In almost all queries, the evaluation time of S_2 is the largest. The second most expensive query is S_5 that retrieves the final results. S_4, used to compute node distances and rank them, has the least contribution to the overall performance. The above results explain why SX-WITH is slower than SX-TEMP consistently. In SX-WITH, S_2 is executed multiple times. On the other hand, SX-TEMP executes S_2 only once.

Observe that although S_5 is second most expensive component, in most cases it takes less than a second to execute. This is beneficial for exploratory querying as any modification to α or β by the user results in re-evaluation of S_5. Note that we do not need to re-evaluate the remaining subqueries as the ranks of all matched test nodes have already been computed by S_4.

4.2 Effect of Number of Context Nodes

We now study the effect of number of context nodes that satisfies a `rank-distance` axis query. Note that the number of distance computations increases with the number of context nodes that match a query. We denote the number of context nodes as Z and vary it from 0 to 500. We use the queries XQ7, XQ8, and XQ9 in Figure 12(a) as representative queries and fix $\alpha = \beta = 1$. Figure 13 reports the query evaluation times. We observe that the query evaluation times increases sub-linearly with Z.

ID	Query	Max# of Rows
XQ7	/catalog/item/authors/author/contact_information/ phone_number[text()='PHONE_1']/rank-distance(1 to 1)::email_address	10
XQ8	/catalog/item[title='TITLE_1']/rank-distance(1 to 1)::quantity_in_stock	10
XQ9	/catalog/item[description='DESCRIPTION_1']/rank-distance(1 to 1)::pricing	40
XQ10	/catalog/item/subject[text()='SUBJECT_1']/rank-distance(1 to 1)::author	395
UQ6	/uniprot/entry[position()=1 to 100]/name/rank-distance(1 to 1)::sequence	100
UQ7	/uniprot/entry[sequence='MAAA%']/rank-distance(1 to 1)::accession	1,632
UQ8	/uniprot/entry/reference/scope[text()='PROTEIN SEQUENCE']/rank-distance(1 to 1)::source	5,819

(a) Query set

Query	S2	S3	S4	S5
XQ7	2,681.94	86.10	74.40	181.60
XQ8	2,689.48	109.90	83.24	178.80
XQ9	4,920.22	96.50	80.38	190.60
XQ10	30,708.72	119.50	74.20	403.76

(b) XBench DC1000

Query	S2	S3	S4	S5
UQ6	365.04	102.00	76.28	279.80
UQ7	6,503.06	240.78	118.22	987.92
UQ8	3,049.10	612.80	189.14	4,577.16

(c) Uniprot/KB U2843

Figure 12: Evaluation times of subqueries (in msec.).

Also, SX-TEMP is much more scalable than SX-WITH and outperforms it for majority of the queries (highest observed factor being 61.6). This further confirms that the strategy for using the temporary tables improves the query performance consistently as smaller queries are less likely to stress the query optimizer. Although tree-unaware schemes are not designed to compute node distances in an XML tree, the above results demonstrate that by exploiting the encoding scheme of SUCXENT++ and ranking support of underlying RDBMS, as well as materializing relevant intermediate results in temporary tables we can effectively support non-directional axis evaluation.

5 Related Work

There is a wealth of work on evaluating XPath expressions in a tree-unaware RDBMS [9]. Our work differs from these efforts in the following ways. Firstly, unlike a conventional XPath axis, the `rank-distance` axis is non-directional. Secondly, evaluation of the `rank-distance` axis requires the computation of distances between pairs of nodes in an XML tree. None of the traditional XPath evaluation mechanisms require such distance computation.

Our objective to flexibly issue XML queries independent of the structure is shared by several recent papers. [8] presents a semantic search engine for XML. The search relies on an interconnection 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. [14] 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 [15,16,18,21] 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

Query	SX-WITH				SX-TEMP			
	Z=0	Z=50	Z=100	Z=500	Z=0	Z=50	Z=100	Z=500
XQ7	142.54	226.96	277.32	710.12	120.72	234.58	277.70	482.00
XQ8	191.28	598.54	805.32	2,296.22	102.30	331.02	402.00	519.74
XQ9	201.18	666.84	907.62	2,866.06	109.96	601.54	492.00	762.88

(a) DC10

Query	SX-WITH				SX-TEMP			
	Z=0	Z=50	Z=100	Z=500	Z=0	Z=50	Z=100	Z=500
XQ7	232.66	364.12	407.26	873.92	149.82	344.72	393.94	627.26
XQ8	278.78	3,053.52	5,078.46	20,761.16	181.30	797.58	802.92	927.28
XQ9	434.78	3,627.10	5,669.52	21,911.00	316.82	1,293.40	1,384.54	1,708.24

(b) DC100

Query	SX-WITH				SX-TEMP			
	Z=0	Z=50	Z=100	Z=500	Z=0	Z=50	Z=100	Z=500
XQ7	1,324.34	1,577.12	1,570.28	2,078.70	637.54	2,855.62	2,901.82	3,137.08
XQ8	970.90	19,272.82	37,038.58	179,619.18	582.52	2,748.60	2,756.94	2,915.48
XQ9	1,857.30	34,010.96	51,864.14	192,811.92	1,416.00	5,108.50	5,164.16	5,443.74

(c) DC1000

Figure 13: Effect of number of context nodes (in msec.).

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 [4, 15, 16]. 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.

In [3], Balmin et al. propose a system, called SEDA, that enables users to start with simple keyword-style querying, and interactively refine the query based on result summaries, and obtain result *data cubes*. In contrast, our proposed approach does not require iterative keyword-style querying. In [20], the data cube is computed based on matching tree patterns with structural relaxations. Both these methods still require the formulation of directional XML queries to perform OLAP computation. Furthermore, none of these techniques exploit node distances similar to rank-distance queries.

More germane to this work is the effort by Zhang and Dyreson [24]. 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 α and β to one) but also nodes that are further away from the context node. Second, in [24] 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.

6 Conclusions and Future Work

In this paper, we present an efficient relational-based strategy to evaluate a non-directional parametric 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 SUCXENT++ system [17]. We showed that by exploiting the encoding scheme of SUCXENT++ and ranking facility of an off-the-shelf RDBMS, we can effectively compute the distances between pairs of nodes and rank them in order to compute rank-distance nodes. In this context, we proposed two variants of an XPath-to-SQL translation algorithm (denoted as SX-WITH and SX-TEMP in the paper) to translate a `rank-distance` axis query to its equivalent SQL form. Our empirical study showed that SX-TEMP is superior to SX-WITH as smaller queries in the former are less likely to stress the relational optimizer. More importantly, we demonstrated that a tree-unaware relational framework can be effectively used to support a non-directional XPath axis.

In future, we plan to extend our approach to yet other non-directional axes, which we believe can be supported using the techniques presented in this paper. For instance, we are currently investigating the `neighborhood` axis that can find *common neighbors* between a pair of nodes in any direction.

References

[1] S. AMER-YAHIA, S. CHO, ET AL. Tree Pattern Relaxation. *In EDBT*, 2002.

[2] S. AMER-YAHIA, L. LAKSHMANAN, S. PANDIT. Flexpath: Flexible Structure and Full-Text Querying for XML. *In SIGMOD*, 2004.

[3] A. BALMIN, L. COLBY, E. CURTMOLA ET AL. Search Driven Analysis of Heterogeneous XML Data. *In CIDR*, 2009.

[4] Z. BAO, T. W. LING, B. CHEN, J. LU. Effective XML Keyword Search with Relevance Oriented Ranking. *In ICDE*, 2009.

[5] S. S. BHOWMICK, E. LEONARDI, H. SUN. Efficient Evaluation of High-Selective XML Twig Patterns with Parent Child Edges in Tree-Unaware RDBMS. *In CIKM*, 2007.

[6] P. Boncz, T. Grust, et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. *In SIGMOD* , 2006.

[7] T. Brodianskiy, S. Cohen. Self-Correcting Queries in XML. *In CIKM*, 2007.

[8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. *In VLDB*, 2003.

[9] G. Gou, R. Chirkova. Efficiently Querying Large xml Data Repositories: A Survey. *In IEEE TKDE*, 19(10), 2007.

[10] T. Grust, J. Rittinger, J. Teubner. Why Off-the-Shelf RDBMSs are Better at XPath Than You Might Expect. *In SIGMOD* , 2007.

[11] T. Grust, S. Sakr, J. Teubner. XQuery on SQL Hosts. *In VLDB*, 2004.

[12] Y. Kanza, W. Nutt, Y. Sagiv. Queries with Incomplete Answers over Semistructured Data. *In PODS*, 2001.

[13] Y. Kanza, Y. Sagiv. Flexible Queries over Semistructured Data. *In PODS*, 2001.

[14] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. *In VLDB*, 2004.

[15] Z. Liu, Y. Chen. Identifying Meaningful Return Information for XML Keyword Search. *In SIGMOD*, 2007.

[16] Z. Liu, Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. *In VLDB*, 2007.

[17] B.-S Seah, K. G. Widjanarko, S. S. Bhowmick, et al. Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases. *In DASFAA*, 2007.

[18] C. Sun, C.-Y. Chan, A. Goenka. Multiway SLCA-based Keyword Search in XML Data. *In WWW*, 2007.

[19] I. Tatarinov, S. Viglas, K. Beyer, et al. Storing and Querying Ordered XML Using a Relational Database System. *In SIGMOD*, 2002.

[20] N. Wiwatwattana, H. Jagadish, L. Lakshmanan, D. Srivastava. X^3: A Cube Operator for XML OLAP. *In ICDE*, 2007.

[21] Y. Xu, Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. *In SIGMOD*, 2005.

[22] B. Yao, M. Tamer Özsu, N. Khandelwal. XBench: Benchmark and Performance Testing of XML DBMSs. *In ICDE*, 2004.

[23] M. Yoshikawa, T. Amagasa, et al. XRel: A Path-based Approach to Storage and Retrieval of XML documents Using Relational Databases. *In ACM TOIT*, 1(1):110-141, 2001.

[24] S. Zhang and C. Dyreson. Symmetrically Exploiting XML. *In WWW*, 2006.