# Optimising XML-Based Web Information Systems

Colm Noonan and Mark Roantree

Interoperable Systems Group, Dublin City University, Ireland -
{mark,cnoonan}@computing.dcu.ie

**Abstract.** Many Web Information Systems incorporate data and activities from multiple organisations, often at different geographical (and cultural) locations. Many of the solutions proposed for the necessary integration in Web Information Systems involve XML as it provides an interoperable layer between different information systems. The impact of this approach is the construction of large XML stores and often there is a need to query XML databases. The work presented in this paper supports the web computing environment by ensuring that this canonical representation (XML) of data can be efficiently processed regardless of potentially complex structures. Specifically, we provide a metamodel to support query optimisation for Web Systems that employ XPath.

**Key words:** XML Query Optimisation, XPath, Web Information Systems

## 1 Introduction

As today's Web Information Systems (WIS) often incorporate data and processes from multiple organisations, the concept of *data everywhere* is more evident. Furthermore, this distribution and diversity of information will increase as we move towards ambient, pervasive and ubiquitous WIS computing. There has been over 20 years of research into interoperability for information systems and it is likely that many of the solutions proposed for the necessary integration in the WIS environment will involve XML as it provides an interoperable layer between different information systems. The effect of this approach is the construction of large XML stores (referred to as XML databases from now on), together a need to efficiently query these XML databases. In fact, it is often suitable to retain this information in XML format due to its interoperable characteristics or because source organisations have chosen XML as their own storage method. Our motivation is to support the WIS computing environment by ensuring that the canonical representation (XML) of data has the sufficient query performance. Our approach is to provide an XML metamodel to manage the metadata necessary and support query optimisation for WIS applications.

In the WIS environment, the canonical model must be semantically rich enough to represent all forms of data structure and content. However, XML databases perform badly for many complex queries where the database size is large or the structure complex and thus, some form of indexing is required to boost performance. The index-based approach provides an efficient evaluation of XPath queries against target XML data sets. Given an XPath query, a query processor must locate the result set that satisfies the content and structural conditions specified by the query. Suitable indexing structures and query processing strategies can significantly improve the performance of this matching operation. There are numerous index-based query processing strategies for XML databases [2, 3, 9, 15, 16], although many fail to support the thirteen XPath axes as they concentrate on the `child`, `descendant` and `descendant-or-self` axes [15, 16].

### 1.1 Contribution and Structure

In this paper, we present a metamodel for XML databases that models three levels of abstraction: the lowest layer contains the index data; the metamodel layer describes the database (tree) and the meta-metamodel layer describes the schema (tree). However, for this paper we do not describe the third layer. This provides for an advanced schema repository (or system catalog in the relational sense) that is extended to hold index information for query optimisation. We also demonstrate our query processing strategy by discussing methods for evaluation of XPath axes and provide details of experiments to assess the merits of this approach.

The paper is structured as follows: in §2, we provide the concepts and terminology used in our metamodel; in §3, we provide details of the metamodel, and the extended metamodel containing index data; the algorithms that use the metamodel to assist the query optimisation process are described in §4; in §5, we describe the results of our experiments while in §6 we discuss similar approaches; and finally in §7, we offer conclusions.

## 2 XML Tree Fundamentals

For this work, we examined XML trees and schemas and sought to formalise the relationships between them. An attempt at providing some form of structure and statistics for semi-structured sources was proposed in [6] where the authors defined a concept called DataGuides (described later in this paper). We have created a more extensive metadata description of both data and schema trees and differ in the set of allowable paths. In this section, we define our terminology and specify the relationships between metadata and data structures.

### 2.1 Terminology

We refer to the XML data tree as the *database*; the schema tree as the *schema*; and later in the paper, we describe a higher level of abstraction: the *meta-schema*. In figure 1(a), two schemas are illustrated (actual sub-trees from the

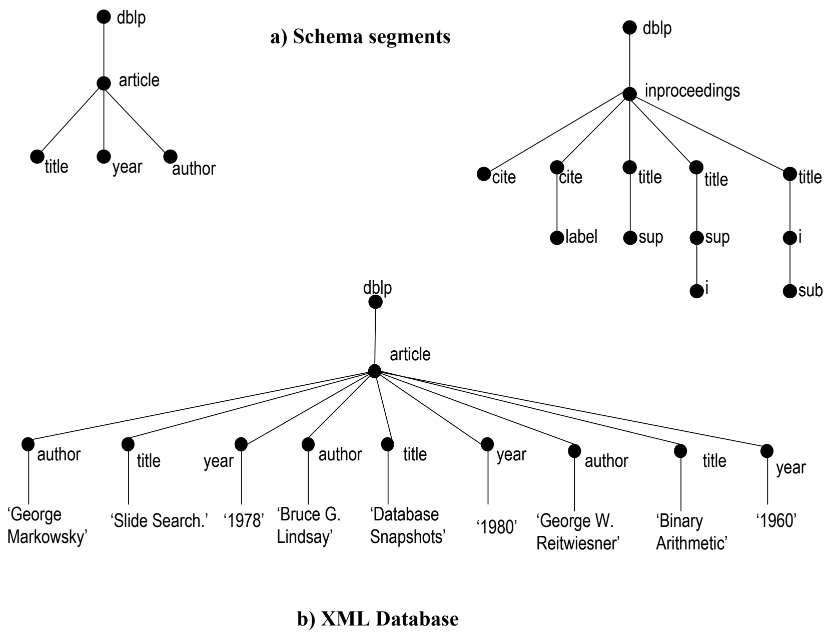

**Fig. 1.** Mapping Across Schema and Data Trees

dblp schema) and in figure 1(b), a database is illustrated for the first of these schemas. We begin with a description of the *schema*.

XML *schemas* are tree structures containing paths and nodes, with the *root node* being the entry point to the tree. A tree-path that begins at the root node, continuing to some *context* node is called a *FullPath*. Tree-paths need not be unique in XML trees[1] but in our metamodel, each FullPath instance is unique. In figure 1(a), the node `title` has the FullPath `//dblp/article/title`. Unlike [1, 6], we do not recognise sub-paths ie. those paths not connected to the document root.

**Property 1** *A Schema S contains a set of FullPaths $\{P\}$, with one or more instances in the database.*

The schema is divided into levels with the root at the topmost level (level 0). Each node has 0 or more child nodes, with child nodes also having 0 or more children (at one level greater than the parent). Rather than use the term subtree, we chose the term *Family* to refer to all child and ancestor nodes for a given context node. Thus, the *Family* for the root is the entire tree, with all other nodes having smaller families.

[1] There are multiple path instances in the dblp schema.

**Property 2** *Each Node N at level(x) is the Head of a Family F of connected nodes at levels (x+1),(x+2),...(x+n).*

In 1(a), the `article` Family is small (3 children) while the `inproceedings` Family is larger with child and descendant nodes.

**Property 3** *A Family F is a set of FullPaths with a common ancestor node.*

The term *Twig* is used to refer to a Family with some members missing (a pruned Family). This is useful where a Family is very large and one wishes to reduce the size of the Family sub-tree (perhaps for querying performance reasons). The term Twig has been used in the past (eg. [4])to refer to data sub-trees returned as a result of queries and our usage of the term is not dissimilar here.

**Property 4** *A Twig T is a subset of some Family F.*

XML databases are instances of XML schemas. Specifically, they have one or more instances of the FullPath construct, and by extension contain Families and Twigs. For example, the **article** Family in figure 1(a) has three instances in figure 1(b).

Similar to many other projects that focus on XPath and XQuery optimisation, we developed an indexing system to both prune the tree space and perform the subsequent query processing. Unlike these approaches, we adopt a more traditional database approach of specifying an Extended Schema Repository (ESR) for the XML database and provide a common interface to the repository. The metadata features of the schema repository (i.e. `FullPath` and `Level` structures) are the principle constructs for query optimisation. These structures describe the database schema, together with some statistics for the data in the tree. In the Extended section of the ESR, we store physical data taken from the XML tree and used to form indexes to boost query performance.

## 3 The XML Metamodel

In this section we provide an overview of the two main layers of the ESR: index data and metadata.

### 3.1 XML Index Data

When the XML database is parsed, all nodes are given {preorder,level} pairs to uniquely identify them. Please refer to our earlier work on indexing XML databases [7] for a detailed discussion on the indexing scheme and to [11] for a description of the object-based interface to index data. This segment of the ESR contains the base and level indexes. To improve efficiently in evaluating XPath location steps along the thirteen XPath axes, a Level index is employed.

- **The BASE Index**
- The `PreOrder` and `Level` values, together with node `parent` are recorded.
- `Type` is used to distinguish between elements and attributes.
- The `name` and `value` of the node.
- `FullPath` is the entire path to the leaf node (direct relationship to FullPath structure.
- The DocID in the event that the database contains multiple documents.
- `Position` This refers to the position of this data item across the *level* and is determined by counting all nodes at this level with smaller preorder value.
- **The LEVEL Index**
- The `Level` value.
- An ordered sequence of `PreOrder` values occurring at a given level in the data tree.

## 3.2 XML Metadata

The purpose of the XML Metadata layer is to describe, through the FullPath and Level meta-objects, the structure and content of the XML database. As this is the main construct for query optimisation, its content was heavily influenced by algorithms for the 13 XPath axes [13]. In addition, every FullPath entry has a relationship with a Level object, providing information about the size of the level in the data tree.

- **The FullPath Structure**
- `FullPath`.
- `NodeCount` is the number of instances of this FullPath.
- `DocID` is necessary where the database contains multiple documents.
- `name` is the name of the leaf node (part of FullPath).
- `Type` attribute is one of Element, Attribute or Root.
- `Level` records the depth of this node.
- `PreorderStart` is the first preorder value for this node.
- `PreorderEnd` is the last preorder value for this node.
- `LeftPos` is smallest PreOrder value at this level for this FullPath (only instances of this FullPath).
- `RightPos` is the biggest PreOrder value at this level (only instances of this FullPath).
- `Children` is the number of child nodes (in the data tree).
- `Descendants` is the number of descendant nodes (in the data tree) excluding child nodes.
- **The Level Structure**
- `Level` is the level number (identical to Level in FullPath).
- `LeftPos` is smallest PreOrder value at this level.
- `RightPos` is the biggest PreOrder value at this level.
- `NodeCount` is the number of nodes at this level (in the data tree).

A further level of meta-metadata is described elsewhere [12] and is used processing view data.

## 4 XPath Query Processing

Algorithms for evaluating location steps against prominent XPath axes are presented here and in [8], we provide algorithms for the entire set of 13 XPath Axes. In brief, our optimisation strategy is to prune the search space using both the query structure and axes properties and then apply content filtering for both content and arbitrary nodes.

– Step 1: **Structure Prune**. A sub-tree is extracted using the content and arbitrary nodes and is specific to the particular axis.
– Step 2: **Context Filter**. Where the context node has a predicate, this is used to filter the remaining sub-tree. For some axes (eg. Following and Preceding) this function involves a single lookup, while other axes (eg. Descendant and Ancestor), it requires a reading of each node in the sub-tree and this step is merged with Step 3 for performance reasons.
– Step 3: **Axis Prune**. Those nodes that do not evaluate to the Axis properties are filtered.
– Step 4: **Arbitrary Filter**. Where the arbitrary node has a predicate, this is used to filter the final result set.

**Algorithm 1** PrecedingAxis (Query Q)

```
Require: Parsed Q provides context node con and arbitrary node arb
 1: Vector resultset = null
 2: int startSS = Min(FullPath.getPreStart(Q,arb))
 3: int endSS = Max(FullPath.getPreEnd(Q,con))
 4: if hasPredicate(con) then
 5:    endSS = Max(BaseIndex.getPre(Q,con))
 6: end if
 7: arbPreOrd = startSS // and now we need the corresponding preorder for con
 8: conPreOrd = Min(FullPath.getPreStart(Q,con))
 9: while arbPreOrd < endSS do
10:    if IsAncestor(arbPreOrd, conPreOrd) != TRUE then
11:       if supportsPredicate(Q, arbPreOrd) then
12:          resultset.add(arbPreOrd) // if no predicate, always add to resultset
13:       end if
14:    end if
15:    arbPreOrd = BaseIndex.getnextPre(arbPreOrd)
16:    conPreOrd = BaseIndex.getnextPre(conPreOrd)
17: end while
18: return resultset // a set of preorder values
```

### 4.1 Preceding Axis

The `preceding` axis contains all nodes in the same document as the context node that are before the context node in document order, excluding `ancestor` nodes of the context node. Consider the query in *example* 1 to retrieve all **title** elements preceding the last **mastersthesis** (context node) element for the named **author** (arbitrary node).

*Example 1.* //mastersthesis[child::author = 'Peter Van Roy']/preceding::title

In the `PrecedingAxis` Algorithm, Q is an object of type Query, variables `con` and `arb` are strings while `StartSS`, `endSS`, `arbPreOrd` and `conPreOrd` are preorder values. Lines 2 and 3 perform the **Structure Prune** step using two single lookup functions. For Preceding queries, **Context Filter** is separated from **Axis Prune** as Context Filter requires a single lookup function and reduces the search space further (in this case) by moving the end point backwards. In lines 7 and 8, we obtain the initial {context,arbitrary} preorder pair. Lines 9 to 17 read through all nodes in the search space, testing for `Preceding` and *predicate* evaluations each time. In prior work [9], we demonstrated that functions `IsPreceding`, `IsFollowing`, `IsDescendant` and `IsAncestor` each execute with optimal efficiency given our index structure (nodes as {preorder,level} pairings). `BaseIndex.getnextPre` is passed a preorder value and returns the next value by checking its type against the FullPath index.

**Algorithm 2** FollowingAxis (Query Q)

```
Require: Parsed Q provides context node con and arbitrary node arb
 1: Vector resultset = null
 2: int startSS = Min(FullPath.getPreStart(Q,con))
 3: int endSS = Max(FullPath.getPreEnd(Q,arb))
 4: if hasPredicate(con) then
 5:     startSS = Min(BaseIndex.getPre(Q,con))
 6: end if
 7: arbPreOrd = startSS // no need for context preorder value
 8: while arbPreOrd < endSS do
 9:     if supportsPredicate(arbPreOrd) then
10:         resultset.add(arbPreOrd) // if no predicate, always add to resultset
11:     end if
12:     arbPreOrd = arbPreOrd + BaseIndex.getSizeOfSubTree(startSS)
13: end while
14: return resultset
```

## 4.2 Following Axis

The `following` axis contains all nodes in the same document as the context node that are after the context node in document order, excluding `descendant` nodes of the context node. An XPath expression to retrieve all `volume` (arbitrary) elements with a value of 2 after the first `incollection` (context) element in the DBLP dataset is displayed in example 2.

*Example 2.* //incollection/following::volume[self::volume = '2']

The `Following` axis is similar to that of `Preceding` in that the **Structure Prune** and **Context Filter** steps are alike. However, the **Axis Prune** step uses the `getSizeOfSubTree` function (demonstrated in [9] to execute efficiently) is used to perform a jump within the search space. This jump ignores all descendants of the arbitrary node.

### 4.3 Ancestor Axis

The `ancestor` axis contains all ancestors of the context node. Consider the query in example 3 to retrieve all **book** elements (arbitrary node) that are ancestors of the **year** (context node) element with the given value.

Note that the algorithm creates a more refined search space (actually multiple sub-trees) than for the `Preceding` axis and thus, the **Prune Structure** step is inside a `for-next` loop. The algorithm begins by creating a set of FullPaths that meet the structure specified in Q, and then for each FullPath the algorithm will generate a result. Lines 3 and 4 preform the **Structure Prune** step using two lookup functions. Steps 2-4 are merged as all require a traversal of the sub-tree(s).

*Example 3.* //year[self::node() = '1998']/ancestor::book

**Algorithm 3** Ancestor(Query Q)

```
Require: Parsed Q provides context node con and arbitrary node arb
 1: Vector resultset = null
 2: for each Fullpath do
 3:    int startSS = FullPath.getPreStart(Q,con)
 4:    int endSS = FullPath.getPreEnd(Q,con)
 5:    conPreOrd = startSS
 6:    arbPreOrd = Min(FullPath.getPreStart(Q,arb))
 7:    while conPreOrd < endSS do
 8:       if IsAncestor(conPreOrd,arbPreOrd) == TRUE then
 9:          if supportsPredicate(Q, conPreOrd) then
10:             if supportsPredicate(Q, arbPreOrd) then
11:                resultset.add(arbPreOrd)
12:             end if
13:          end if
14:       end if
15:       arbPreOrd = BaseIndex.getnextPre(arbPreOrd)
16:       conPreOrd = BaseIndex.getnextPre(conPreOrd)
17:    end while
18: end for
19: return resultset
```

## 5 Details of Experiments

Experiments were run using a 3.2GHz Pentium IV machine with 1GB memory on a Windows XP platform. Algorithms were implemented using Java virtual machine (JVM) version 1.5. The Extended Schema Repository was deployed using an Oracle 10g database (running the Windows XP Professional operating system, with a 3GHz Pentium IV processor and 1GB of RAM). The eXist database (version 1.0b2-build-1107) operated on a machine with an identical specification to that of the Oracle server to ensure an equal set of experiments. The default JVM settings of eXist were increased from -Xmx128000k to -Xmx768000k to maximise efficiency. All experiments used the DBLP dataset [14], containing 3.5 million elements, 400k attributes with 6 levels and a size of 127mb.

**Table 1.** DBLP Queries

| # | **XPath** | **Matches** |
|---|---|---:|
| Q1 | //title[. = ‘A Skeleton Library.’]/ancestor::inproceedings | 1 |
| Q2 | //year[self::node() = ‘1998’]/ancestor-or-self::book | 32 |
| Q3 | //mastersthesis[child::author = ‘Peter Van Roy’]/preceding::title | 77 |
| Q4 | //incollection/following::volume[self::volume = ‘2’] | 3,123 |
| Q5 | /dblp/descendant::phdthesis | 72 |
| Q6 | /descendant-or-self::article[* = ‘Adnan Darwiche’] | 6 |
| Q7 | //book/parent::title | 0 |
| Q8 | /dblp/phdthesis[* = ‘1996’] | 4 |
| Q9 | /dblp/descendant::book[child:author = ‘Bertrand Meyer’] | 13 |
| Q10 | //article/@rating | 61 |

Table 1 presents our XPath query set and for each query, we provide the number of results returned by the DBLP dataset. Each query is executed eleven times with execution times recorded in milliseconds (ms), together with the number of matches. The times were averaged with the first run eliminated to ensure that results are warm cache numbers.

Table 2 displays the execution times for eXist, the Extended Schema Repository (ESR), and in the final column the factor at which the ESR out-performs eXist. A value of 1 indicates that both are equal and anything less than 1 indicates that the ESR is slower than eXist. The range from 0.84 to 69.0 indicates that that at best, we were 69 times faster than eXist.

For queries Q3 and Q4, the eXist query processor failed to return any results as it does not support the `following` and `preceding` axes. The features of our metamodel allow us to quickly identify queries with paths not supported in the target database (see query Q7). For query Q6, we achieved a speed up of 69, as the eXist query processor is inefficient at processing predicates containing wildcards on frequently occurring nodes scattered throughout the database (i.e. the `article` node occurs 111,609 times throughout DBLP). In contrast, our processing strategy allows us to quickly filter for any predicate. Query Q8 contains

**Table 2.** Query Results

| # | eXist (ms) | ESR (ms) | Factor |
|---|---|---|---|
| Q1 | 989.2 | 86.0 | 11.5 |
| Q2 | 6,419.8 | 535.7 | 12.0 |
| Q3 | | 143.9 | |
| Q4 | | 1,265.5 | |
| Q5 | 140.7 | 167.2 | 0.84 |
| Q6 | 28,533.2 | 413.4 | 69.0 |
| Q7 | 149.6 | 57.4 | 2.6 |
| Q8 | 123.9 | 119.3 | 1.04 |
| Q9 | 414.4 | 147.3 | 2.8 |
| Q10 | 233.1 | 38.9 | 6.1 |

a wildcard and while the ESR was slighter faster for this query, we encountered instances where eXist out-performs the ESR as it can efficiently process predicates with wildcards on nodes that are *clustered into a very small segment* of the XML database.

For queries Q1 and Q2, we obtain improvements ranging from 11.5 to 12. The eXist database cannot process predicates on these queries efficiently as the context nodes have a very high frequency scattered throughout the database (i.e. for Q2 the `year` node occurs 328,831 times in DBLP). Although `year` nodes are scattered throughout the database, the `getNextPre` function provides us with direct access to them. Furthermore, our pruning and filtering techniques use metadata information within the FullPath index to quickly identify the relevant search space along the `ancestor` axis, or over any wide search space.

Queries Q5, Q9 and Q10 do not contain any predicates and the improvements range from 0.84 to 6.1. These are less impressive as queries without predicates bypass our optimisation steps 2 and 4. The ESR performs best against queries with selective predicates as they allow us to quickly prune and filter the database using all four optimisation steps. Query Q5 performs marginally better with eXist as the few `phdthesis` nodes in the database are clustered closely together.

## 6 Similar Approaches

In this work, optimisation takes place at the axes level, an approach also taken in [5] whereby recording minimal schema information (preorder, postorder and level values), they provide a significant optimisation for XPath axes. However, they do not employ the different levels of abstraction presented here and thus, operations such as `GetPreStart` and `GetPreEnd` for the FullPath object (to limit the search space) are not easily computable.

One of the earliest effort at generating schema information for semi-structured sources was in [6] where the authors introduced the concept of a DataGuide. They are used by query processing strategies to limit or prune the search space of the target database. However, DataGuides do not provide any information about the parent-child relationship between database nodes and unlike our approach, they cannot be used for axis navigation along an arbitrary node. In [10], they overcame this problem by augmenting their DataGuide with a set of instance functions that keep track of parent-child relationships between nodes although [10] was an early indexing schema that did not cover all categories of XML queries.

The FIX index [16] is based on spectral graph theory. During the initial parsing of the XML dataset, FIX calculates a vector of *features* based on the structure of each twig pattern found in the dataset. These features are used as keys for the twig patterns and stored in the index. During query processing, the FIX processor converts a twig query into a feature vector and subsequently searches the index for the required feature vector.

Experiments illustrate that FIX has strong path pruning capabilities for high selectivity queries (i.e. queries that query a small portion of an XML document), especially on highly unstructured XML documents (e.g. TreeBank). However, results also indicate that FIX is poor at processing low selectivity queries, especially when the target database is highly structured. Furthermore, FIX supports only the `child`, `descendant`, `descendant-or-self` and `self` axes and does not support the evaluation of queries containing the `text`, `node`, `comment` and `processing-instruction` functions. In contract, we provide algorithms for the full set of XPath axes [8].

Perhaps the most significant work in this area can be found in [2] where they compile a significant amount of metadata to support the optimisation process but also provide a full query processor that optimises at the level of *query* rather than the level of *location path* as we do. This facilitates an access-order selection process where more than one location path can be processed simultaneously. Furthermore, they can process queries in both top-down and bottom-up directions, thus, providing a further level of optimisation based on the query type and database statistics. However, we have discovered improvements can be achieved by fine-tuning the axes algorithms. In §4, we illustrated how optimisation steps differ for each axis and in some cases, it was necessary to merge these steps.

## 7 Conclusions

In this paper, we introduced our metamodel for XML databases and demonstrated how it could be used to optimise XPath queries by rewriting the axes algorithms. Our experiments were conducted against the eXist database as many web sources indicated that eXist outperformed its competitors. Two important characteristics in the engineering of WIS applications is that they are interoperable and can be viewed at different levels of abstraction. XML an provide the platform for interoperability as it acts as a canonical model for heterogeneous systems and by using a metamodel approach, it has been shown to assist the integration process. We discovered that using a metamodel with different functioning layers facilitated well-engineered algorithms but also greater performances due to the type of information stored at each level. While similar efforts in this area have demonstrated long build times for their indexes, we reported relatively small build times in our earlier work on constructing XML indexes [7]. This is extremely useful in our current area of research: managing updates for XML databases.

## References

1. Aboulnaga A., Alameldeen A. and Naughton J. Estimating the Selectivity of XML Path Expressions for Interney Scale Applications. *Proceedings of the 27th VLDB Conference*, Morgan Kaufmann, pp. 591-600, 2001.
2. Barta A., Consens M. and Mendelzon A. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. *Proceedings of the 31st VLDB Conference*, Morgan Kaufmann, pp 133-144, 2005.
3. Boulos J. and Karakashian S. A New Design for a Native XML Storage and Indexing Manager. *Proceedings of EDBT 2006*, LNCS vol. 3896, pp. 755-772, 2006.
4. Bruno N., Srivastava D., and Koudas N. Holistic Twig Joins: Optimal XML Pattern Matching. *Proceedings of SIGMOD 2002*, ACM Press, 2002.
5. Grust T. Accelerating XPath Location Steps. *Proceedings of ACM SIGMOD Conference*, pp.109-120, ACM Press, 2002.
6. Goldman R. and Widom J. DataGuides: Enabling Query Formulation and Optimisation in Semisztructured Databases. *Proceedings of the 23rd VLDB Conference*, Morgan Kaufmann, pp 436-445, 1997.
7. Noonan C., Durrigan C. and Roantree M. Using an Oracle Repository to Accelerate XPath Queries. *Proceedings of the 17th DEXA conference*, LNCS vol. 4080, pp. 73-82, Springer, 2006.
8. Noonan C. The Algorithm Set for XPath Axes. *Technical Report ISG-06-03*, Dublin City University, November 2006.
   at: URL http://www.computing.dcu.ie/∼isg/technicalReport.html.
9. O'Connor M., Bellahsene Z. and Roantree M. An Extended Preorder Index for Optimising XPath Expressions. *Proceedings of 3rd XSym*, LNCS Vol. 3671, Springer, pp 114-128, 2005.
10. Rizzolo F. and Mendelzon A. Indexing XML Data with ToXin. *Proceedings of the 4th WebDB Workshop*, pp. 49-54, 2001.
11. Roantree M. The FAST Prototype: a Flexible indexing Algorithm using Semantic Tags. *Technical Report ISG-06-02*, Dublin City University, January 2006.
    at: URL http://www.computing.dcu.ie/∼isg/technicalReport.html.
12. Roantree M. and Noonan C. A Metamodel Approach to XML Query Optimisation (submitted for publication). *Proceedings of the 11th ADBIS Conference*, 2007.
13. The XPath Language,
    at: URL http://www.w3.org/TR/xpath, 2006.
14. Suciu D. and Miklau G. University of Washingtons XML Repository.
    at: URL http://www.cs.washington.edu/research/xmldatasets/, 2002.
15. Weigel F. et al. Content and Structure in Indexing and Ranking XML. *Proceedings of the 7th WebDB Workshop*, pp. 67-72, 2004.
16. Zhang N. et al. FIX: Feature-based Indexing Technique for XML Documents. *Proceedings of the 32nd VLDB Conference*, pp.359-370, 2006.