Pruning XML Trees for XPath Query Optimisation

Colm Noonan

Bachelor of Science in Computer Applications (Information Systems)

A Dissertation submitted in fulfilment of the

requirements for the award of

Master of Science

to the

Dublin City University

Faculty of Engineering and Computing, School of Computing

Supervisor: Dr. Mark Roantree

August, 2007

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters of Science is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed Colm Noonan

Student ID 51712268

Date August, 2007

Acknowledgements

I would like to take this opportunity to thank my supervisor Dr. Mark Roantree, for his inspiring support, encouragement and invaluable comments over the past two years. Working within the Interoperable Systems Group (ISG) has been a highly rewarding experience and the support from all team members has played a huge role in the completion of this thesis. Also, I would like to add a special thanks to Enterprise Ireland [1] and the Faculty of Engineering and Computing for providing me with financial support.

To my family and friends that have supported me during my academic career, I would like to mention just how grateful I am for your inspiration and help. Especially to Mam and Dad who have sacrificed so much to ensure that I achieved my academic goals and to my brother Dr. Liam Noonan for his help and advise. Finally, I would like to thank everybody here at DCU for making the past six years of my life unforgettable.

[1]Funded by Enterprise Ireland Grant PC/2005/0049

Abstract

XML has been widely adopted for interoperable applications. This often requires the construction of large XML repositories resulting in poor query response times. Although the poor performance of XPath queries has attracted a great deal of attention from the research community in the form of specialised indexes, there are still issues concerned with build times for indexes and the lack of support for the full set of XPath axes. This thesis adopts a metadata approach to the problem with two levels of abstraction: schema metadata and index/statistical metadata. The purpose of storing metadata is to optimise the query processing effort at the level of the XPath axes. Each axis has a separate processing logic and they exploit a common set of metadata constructs in different ways. Together with an overall strategy for the management of XPath queries, we demonstrate levels of improvement over the widely used eXist database.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The eXtensible Markup Language (XML) [9] was primarily developed as the document markup language to succeed the less powerful Hyper Text Markup Language (HTML). HTML did not provide adequate support for complex data exchange as it was not sufficiently powerful in structure [27]. XML is rapidly becoming the standard for information exchange across the internet and has resulted in the construction of large XML repositories [4]. To manipulate these large data stores, both academia and industry have developed XML query processing strategies. However, XML query processors cannot perform at a level that allows for general usage and as a result the true power of XML as a data exchange mechanism has not been exploited.

The introductory chapter is structured as follows: §1.1 introduces the XML data model, along with DTDs and XML schema documents; §1.2 presents the different types of XML databases with their advantages and disadvantages highlighted, before introducing the XML query languages in §1.3. Section 1.4 outlines (*i*) the issues outstanding in the provision of an adequate query service for native XML databases, which acts as the motivation for my research and (*ii*) the contributions made in this thesis. In §1.5 the conclusions of this chapter together with the thesis structure are presented.

1.1 XML Data Model

An XML document (see *Example* 1.1) is modeled as a rooted, ordered, labeled tree structure, where each node corresponds to a root, element, attribute, namespace, comment, processing instruction or text node and direct relationships between nodes are modeled as edges [11] (see Figure 1.1). Nodes can be either atomic or composite. Atomic nodes contain raw character data, while composite nodes contain a sequence of nested subelements.

Example 1.1 (Sample DBLP XML document)

```
<dblp>
    <phdthesis key="phd/Smolka89">
        <author>Gert Smolka</author>
        <title>Logic Programming over Polymorphically Order-Sorted Types.</title>
        <year>1989</year>
        <school>Universitt Kaiserslautern</school>
    </phdthesis>
    <mastersthesis key="phd/VanRoy84">
        <author>Peter Van Roy</author>
        <title>A Prolog Compiler for the PLM.</title>
        <year>1984</year>
        <school>University of California at Berkeley</school>
    </mastersthesis>
</dblp>
```

Unlike relational data, XML data is self-describing and irregular as its structure may change rapidly or unpredictably. As a semi-structured data model, XML is a bridge between the structural world of relational systems and the free-form world of text documents. The semi-structured and self-describing nature of the XML data format allows it to model an unlimited number of tree dialects. Irregular XML trees are schema-less, while structured documents can be assigned a Document Type Definition (DTD) or an XML Schema Document (XSD) that define the document structure, similar to table schemas in the relational domain. A DTD (see Figure 1.2(a)) provides a list of legal XML tags for an associated document but it lacks the expressive power to describe highly structured data. Unlike DTDs, XSDs (see Figure 1.2(b)) provide a richer set of constraints for describing the content and structure of a highly structured XML document. However, due to their simplicity DTDs tend to be more popular. Mignet et al [25] conducted a survey of 190,417 XML documents and reported that 48% of their dataset referenced a DTD, while

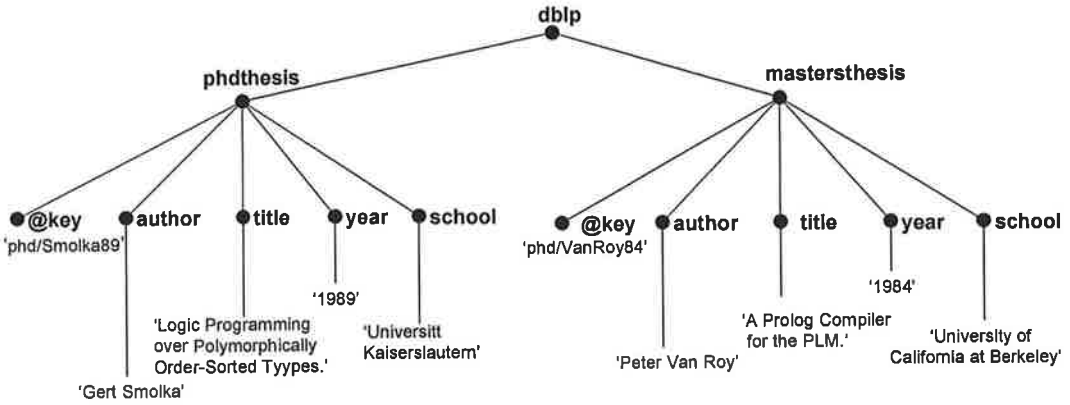

Figure 1.1: Tree representation of a sample DBLP XML document

only 0.09% linked to an XSD.

The popularity of XML is due primarily to its simplistic and interoperable nature. Since XML is extensible, platform independent and separates content from presentation instructions (via XSL stylesheets) it can naturally serve as a format for representing, integrating and exchanging information in both a local and distributed environment. As the volume of XML documents grew in size, a significant number of database products emerged [8]. In the following sections, we will discuss storage for XML documents and the languages used to manipulate XML data.

1.2 XML Databases

In recent years there has been a sharp increase in the availability of XML databases, with a recent report [8] suggesting that there are approximately 60 XML databases currently on the market. XML Databases are classified into two broad groups: XML enabled databases and native XML databases.

1.2.1 XML Enabled Databases

XML enabled databases extend relational technology in order to manipulate XML documents. The main advantage of this approach to XML storage and query processing is that it exploits the maturity, scalability and extensive tuning of relational

```
<!ELEMENT author (#PCDATA)>
<!ELEMENT dblp (phdthesis, mastersthesis)>
<!ELEMENT mastersthesis (author, title, year, school)>
<!ATTLIST mastersthesis key CDATA #REQUIRED>
<!ELEMENT phdthesis (author, title, year, school)>
<!ATTLIST phdthesis key CDATA #REQUIRED>
<!ELEMENT school (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```

(a) DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="dblp">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="phdthesis"/>
                <xs:element ref="mastersthesis"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="mastersthesis">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="author"/>
                <xs:element ref="title"/>
                <xs:element ref="year"/>
                <xs:element ref="school"/>
            </xs:sequence>
            <xs:attribute name="key" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="phdthesis">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="author"/>
                <xs:element ref="title"/>
                <xs:element ref="year"/>
                <xs:element ref="school"/>
            </xs:sequence>
            <xs:attribute name="key" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="author" type="xs:string"/>
    <xs:element name="school" type="xs:string"/>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="year" type="xs:int"/>
</xs:schema>
```

(b) XSD

Figure 1.2: Schema types for sample DBLP XML document

database technology that has been achieved through 20 years of research and development. According to the Gartner Group [38], XML enabled databases can be decomposed into two distinct categories: XML-Wrapped DBMS and XML-Grounded DBMS.

XML-Wrapped DBMS

XML-Wrapped DBMS are relational or object-relational databases that use a mapping technique to store XML data in their own schema specific storage structures. With the mapping technique, the XML dataset is shredded, decomposed and stored in rows and columns within the DBMS. Furthermore, all XML queries must be converted into SQL and in [18, 12] they provide procedures for translating XQuery to SQL. Traditional XML-Wrapped databases are suitable for highly structured documents and often a DTD or XSD is required for data storage e.g. Oracle 10g XML DB. However, less then 50% of XML documents reference a DTD or XSD [25]. Since XML-Wrapped databases basically shred XML documents into relational tables,

information (in document order) in a tree-based index structure. It is the loss of implicit ordering of nodes in conjunction with the loss of structural information that is one of the main reasons why traditional XML-Wrapped DBMS are outperformed by native XML databases [27].

1.3 XML Query Languages

It is essential that XML repositories support the W3C XML query languages such as XPath [11] and XQuery [5] as their semantics are the key enablers in supporting interoperability among XML databases. An XPath expression (see §3.2) corresponds to the tree-centric nature of XML, as it queries an XML document as a tree of nodes and returns the subtrees that match the query structure. In *Example 1.2*, the XPath expression retrieves all subtrees rooted at a `title` element, that are children of `mastersthesis` elements, and have a child node named `author` with the value `'Peter Van Roy'`. XPath queries contain pattern matching constraints that are used to extract nodes that match a specified pattern. These constraints fall into two broad categories [26]:

- *Structural constraints* - impose restrictions on the structure of the retrieved nodes (e.g. `//mastersthesis[author]/title`).
- *Value-based constraints* - select nodes according to their values (e.g. `author = 'Peter Van Roy'`).

Example 1.2 (Return the title of the masters thesis by 'Peter Van Roy')

```
//mastersthesis[author = 'Peter Van Roy']/title
```

As an extension of XPath 2.0, XQuery is a more powerful query language that provides a richer support for manipulating XML data. FLWOR expressions (i.e. `for`, `let`, `where`, `order by` and `return`) are used to build SQL-like XQuery expressions, with an embedded XPath expression used to address specific parts of the target XML dataset. The `for`, `let` and `where` constructs are capable of performing complex querying, while the `return` and `order by` clauses customise the construction of the result set. The `for` and `return` constructs are mandatory in

XQuery FLWOR expressions, unlike the remaining clauses. *Example* 1.3 illustrates an XQuery expression that returns a sequence of `phdthesis` nodes ordered by their respective `year` for the named `school`.

Example 1.3 (Return in chronological order all PhD theses from Kaiserslautern)

```
for $phd in //phdthesis
where $phd/school = 'Universitt Kaiserslautern'
order by $phd/year
return $phd
```

1.4 Motivation and Contribution

1.4.1 Motivation

The work presented in this thesis forms part of the Flexible indexing Algorithm using Semantic Tags (FAST) project [35]. The FAST project sought to create a novel storage manager with a custom query processing framework that optimises the performance of XML queries against the rapidly growing volume of XML data. As discussed in §1.2, native XML databases are superior to XML enabled databases for storing, retrieving and processing persistent XML documents. However, many of these query services fail to provide adequate support for XML query languages [5, 11] or perform badly against a large complex dataset.

Given an XML query, a query processor needs to retrieve the result set that satisfies the content and structural conditions specified by the query. Suitable indexing and query processing strategies can significantly improve the performance of this matching operation. Indexes for native XML databases generally store subtree nodes close to one another and as a result, the children/descendants are stored in a close proximity to their parent/ancestors. Thus, native XML databases perform well for *regular path expressions* i.e. queries that only implement the `child`, `descendant` and `descendant-or-self` axes plus name tests [43, 46]. However, many index-based query processing strategies fail to provide full support for the thirteen XPath axes [3, 7, 43, 46] (see Table 2.1).

Efforts to use an index are hampered by the fact that XML indexes tend to

be very large as each individual node in the XML tree has a series of identifiers (i.e. node name, node type, unique node identifier, etc). Large index structures can be slow to process especially for join and axis (e.g. `isAncestor`) operations that scan the entire index. Several approaches [32] to indexing overcome the index size issue by only indexing nodes up to a certain depth in the XML tree. However, this approach leads to a partial index that cannot process the universal set of XPath queries.

To deal with these shortcomings, we propose an advanced query processing strategy based on a state-of-the-art index repository designed for the FAST project. Our query processor supports the full set of XPath axes and adopts an aggressive axis pruning strategy that limits the search space of the underlying index repository. As a result, large index sizes representing very large number of XML nodes do not degrade query performance.

1.4.2 Contribution

In this thesis, we provide a framework for optimising the performance of XML queries. The XQuery language is very large, as a result our focus is the main building block of XQuery i.e. the XPath query language. We believe that our approach to query processing is superior to the principle industry leader and approaches taken in academia as we provide a greater support for the XPath query language and improved query response times. XPath query optimisation is achieved by a processing strategy that exploits the features of a metamodel for XML databases, which is deployed as the *Extended Schema Repository* (ESR).

The indexing algorithm for the ESR not only indexes the structure of XML documents but also maintains the implicit ordering of nodes using a numbering scheme that allows our work to support traversals along each of the thirteen XPath axes. The ESR indexes can be very large as each node in the target XML tree is indexed, allowing it to support any structural or content constraint and thus, the universal set of XPath queries. Issues regarding large indexes are resolved by the construction of a compact metadata schema that provides structural metadata information regarding our larger indexes. The query processor uses this metadata

information to prune the search space so that only the relevant sections of the index repository are inspected and as a result, large index sizes do not adversely effect query performance.

The crux of our research is the development of a set of index-based algorithms for evaluating XPath expressions along each of the thirteen XPath axes. These algorithms exploit the metadata capabilities of the ESR to swiftly evaluate path expressions by pruning the node indexes of the ESR according to the query structure and axis definition. We demonstrate the success of our query processing framework using a set of detailed experiments. These experiments execute a series of XPath queries against a well known (large and complex) XML dataset in order to compare the performance of our query processing framework against a state-of-the-art native XML database.

1.5 Conclusions

In this Chapter, the XML data format was introduced together with reasons for its popularity and the principle XML query languages were presented. The evolution of both native and enabled XML databases to facilitate the rapid growth of XML was detailed along with the problems still outstanding in the provision of an adequate XML query service were explored. The adoption of a suitable indexing structure for XML databases that incorporates XML metadata was identified as the key enabler of an optimised XML query processing framework. The contributions of our strategy for XML query optimisation were outlined in §1.4.2.

The remainder of the thesis is organised as follows: Chapter 2 reviews the existing state-of-the-art solutions to efficient XML query processing, while Chapter 3 introduces our XML query processing strategy. Chapter 4 begins with an in-depth analysis of our index repository and then proceeds with a detailed discussion of our optimised algorithms for processing location steps along each of the thirteen XPath axes. We then show in Chapter 5 the results of our extensive experimental study of comparing our query processing strategies against what we believe is the most advanced open source native XML database on the market. The conclusions of this

thesis are presented in Chapter 6 together with proposed areas for future research.

Chapter 2

Related Research

In the previous chapter, we discussed the growth of the XML data type and XML databases to motivate the use of advanced XML query processing strategies. To meet this demand, several index-based XML query processing strategies have been developed in recent years [24, 3, 17, 19, 46, 22, 33, 10, 7, 34]. However, most of these techniques provide inadequate support for the XPath and XQuery languages as they concentrate on the evaluation of *regular path expressions* i.e. queries that only implement the `child`, `descendant` and `descendant-or-self` axes plus name tests. In many cases, XML databases tend to be quite large as their index structures contain large quantities of structural and content information, thus many query processors exploit optimising pruning techniques in order to limit the database search space.

In this chapter, several research projects covering existing state-of-the-art indexing and XML query processing techniques are presented. When examining these projects, the emphasis was put on their ability to prune the index search space and support the XPath and XQuery languages. This chapter is structured as follows: in §2.1 to §2.4 four different research projects are evaluated and in §2.5 some conclusions are presented.

2.1 DataGuides

Much of the information stored on the web is of a semi-structured and unstructured nature (e.g. HTML, XML), which is not suitable for relational or object-relational

DBMS and as a result there is a growing demand for semi-structured databases. Traditional DBMS use schema information to boost query performance. But with semi-structured databases there is either no schema or only an inadequate partial schema defined in advance of database population. Early work on generating schema information that boosts query performance for semi-structured sources, introduces the concept of a *DataGuide* [16], which has become an influential index structure for graph-structured data [32, 43, 34, 20, 3, 26].

2.1.1 Overview

A DataGuide is a concise, yet accurate summary of the path structure of the source database that is dynamically generated from the underlying database. DataGuides conform to the data rather then forcing the data to conform to it, as is the case with table schemas in the relational model. It serves a similar role as metadata in relational systems and in the same way that a relational query processor consults metadata, the DataGuide is used as a means of guiding the query processor of semi-structured databases.

A DataGuide consists of two data structures that allow path expressions to be directly evaluated without traversing its data tree. Its principle data structure is an *index tree* that contains a structural summary of the paths found in the source database. Every *label path* - the string formed by the concatenation of the labels of the data graph edges - in the database is recorded once, regardless of the number of times it occurs in the data source. Using the sample database in Figure 2.1(a), which identifies each node by its path label and object identifier (oid), the label path for oid `&11` is `'/dblp/inproceedings'`. Furthermore, it does not contain any atomic values (i.e. integers, reals, strings, etc) as it was designed to only model the structure of the database. Its secondary data structure is a *path table* that matches a given label path to its node instances from the source database e.g. the target set of the label path `'/dblp/book'` in Figure 2.1(b) is {`&3`, `&7`} in Figure 2.1(a).

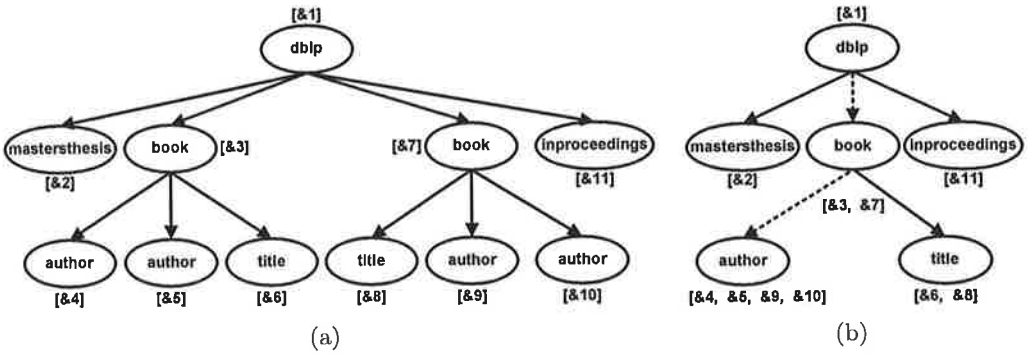

Figure 2.1: Sample DBLP database and DataGuide

2.1.2 Benefits

DataGuides are used by query processing strategies to boost query performance as a lack of schema information can cause a query processor to resort to exhaustive searches of the entire database. Query optimisation is achieved by having fast access to the target node sets during query processing. A DataGuide is generally small compared to the target database as each distinct label path is recorded only once and no atomic values are recorded. This means that DataGuides can improve query processing as they can be queried more efficiently than the original source database.

To highlight the benefits of this technique, consider the regular path expression `'/dblp/book/author'` executed against both the source database and DataGuide in Figure 2.1. For evaluating a path query without a DataGuide, query processing starts at the root node `dblp` (1 instance) and then examines every `book` node (2 instances). Then the `author` (4 instances) node of each `book` is examined and returned. With the DataGuide technique, each label path has direct access to its data source instances, which allows the evaluation of a given path expression to be optimised as only the relevant nodes in the data tree must be traversed. Using the DataGuide approach, each path in the DataGuide is unique, as a result we only need to examine three node instances (see dashed lines) i.e. `dblp`, `book` and `author`. Therefore, the result of evaluating the above path expression with the DataGuide would be all objects in the path table for the label path `'/dblp/book/author'` i.e. {&4, &5, &9, &10}.

Without some concept of the database structure, formulating queries can be extremely difficult especially against large datasets. Since DataGuides provide concise information regarding the database structure, structural summaries can be built into client applications to enable users to comprehend the database structure and formulate meaningful queries against it. Furthermore, the authors [16] recorded low DataGuide creation costs for relatively small datasets, as they require little time for DataGuide creation and yield DataGuides significantly smaller than the source. Additional benefits of DataGuides are:

- The index tree resides in main memory (unlike the path table), which avoids many disk operations during path matching.
- DataGuides are useful for deducing valid path expressions as the index tree can easily determine whether a label or path expression exists in the database.
- Stores a series of useful statistics such as the total number of database objects reachable by that path expression.
- Not only is the DataGuide dynamically generated it is also dynamically maintained as it always represents the current state of the underlying database i.e. supports updates.

2.1.3 Limitations

The DataGuide approach fails to provide a formalised XPath optimisation strategy (as it predates the XPath). While, the authors [16] claim to provide improved query response times, their experimental results failed to demonstrate this as they concentrate on DataGuide construction times. The XPath query language [11] lists a family of thirteen axes, these axes are described in Table 2.1 and are relative to a *context node*, which is the starting point of a traversal along a given axis. The main disadvantage of the DataGuide approach is that, it fails to provide adequate support the XPath query language as it can only support the evaluation of regular path expressions without wildcards i.e. only three (`child`, `descendant` and

Axis Name	Abbreviation	Node Set Result
`child`	/	all element child nodes of c
`descendant-or-self`	//	c and all descendants of c, excluding attribute and namespace nodes
`descendant`		all descendants of c, excluding attribute and namespace nodes
`parent`	..	parent of c
`ancestor`		all ancestors of c
`ancestor-or-self`		c and ancestors of c
`preceding`		all nodes that are before c in document order, excluding ancestors of c, attribute and namespace nodes
`preceding-sibling`		all the preceding siblings of c
`following`		all nodes that are after c in document order, excluding descendants of c, namespace and attribute nodes
`following-sibling`		all the following siblings of c
`self`	.	c
`attribute`	@	all attributes of c
`namespace`		all namespace nodes of c

Table 2.1: Semantics of the 13 XPath axes relative to a context node c

`descendant-or-self`) of the thirteen XPath axes are supported. With this technique, query evaluation always begins at the root node, as a result the ability to start path traversals from any given node in a data tree is not supported. Therefore, DataGuides are unsuitable for XQuery evaluation, as the path expressions embedded in an XQuery query may not commence from the document root node.

2.2 XPath Accelerator

In recent years, a large number of indexing schemes that summarise the database structure were developed [16, 34, 43, 32, 20, 26]. Almost exclusively, these index structure concentrate on providing support for the efficient evaluation of the `child`, `descendant` and `descendant-or-self` axes i.e. regular path expressions. In [17], the author proposes an index structure known as the *XPath accelerator*, which was especially designed to provide full support for the entire set of XPath axes.

2.2.1 Overview

The XPath query language states the following partition property: the `ancestor`, `descendant`, `following`, `preceding` and `self` axes partition an XML tree (ignoring `namespace` and `attribute` nodes). Partitions are disjoint and together form a set that contains all nodes in the XML document. If a given node v resides on the `self` axis, then all the remaining nodes in the XML document fall into one of the four remaining partitions (known as the *major* axes). The XPath accelerator exploits this partition property such that, for any context node, it can efficiently identify the node set in the four document partitions specified by the major axes. The remaining XPath axes specify supersets or subsets of these node sets e.g. for a context node c, axes `ancestor-or-self` and `descendant-or-self` simply add c to the respective ancestor or descendant partitions. Additional information is stored in the index repository to support axis evaluation along with node tests on the element and attribute names.

During XML parsing, each node c is assigned a 5-dimensional descriptor, which is then stored in an indexed table. This node descriptor is based on *preorder* and *postorder* encoding of XML nodes. With preorder traversal, each node in the document tree is visited in document order and assigned a unique preorder rank `pre` *before* its children are recursively traversed from left to right. While a postorder traversal, assigns a distinct postorder rank `post` to each node *after* all its children have been traversed from left to right. The remaining descriptor attributes are as follows:

- `par` records the preorder rank of a node's parent. This attribute is required for the evaluation of the `parent`, `child`, `following-sibling`, `attribute` and `preceding-sibling` axes.
- `att` is a boolean attribute that returns true for nodes found on an `attribute` axis, thus supporting the `attribute` axis.
- `tag` supports the processing of node tests as it stores the node name of c.

The `pre` and `post` values create a 2-dimensional representation of the target

Axis α	pre	post	par	att	tag
child	(pre(c),∞)	[0,post(c))	pre(c)	false	*
descendant	(pre(c),∞)	[0,post(c))	*	false	*
descendant-or-self	[pre(c),∞)	[0,post(c)]	*	false	*
parent	[par(c),par(c)]	(post(c),∞)	*	false	*
ancestor	[0,pre(c))	(post(c),∞)	*	false	*
ancestor-or-self	[0,pre(c)]	[post(c),∞)	*	false	*
following	(pre(c),∞)	(post(c),∞)	*	false	*
preceding	(0,pre(c))	(0,post(c))	*	false	*
following-sibling	(pre(c),∞)	(post(c),∞)	par(c)	false	*
preceding-sibling	(0,pre(c))	(0,post(c))	par(c)	false	*
attribute	(pre(c),∞)	[0,post(c))	pre(c)	true	*

Table 2.2: XPath axes α and their query space *window(α,c)* for context node c

document that can be partitioned into four disjoint regions with direct mappings to the major XPath axes. This means that we can pick any node and use its location in the 2-dimensional `pre/post` space to start an axis traversal as it identifies a rectangular region (known as a *query window*) in the plane. This query window (see Table 2.2) contains the query results for a given context node c along the specified axis type, thus pruning the index search space according to preorder, postorder and axis logic.

Modified XPath Accelerator Encoding

In [18], the authors of the XPath Accelerator modified its index structure. They replaced the postorder rank for a node v with:

1. `size` records the number of nodes in the subtree below node v
2. `level` is the length of the path from the root node to node v in the XML data tree.

The new `pre/size/level` encoding scheme provides the same functionality as the the original encoding, as the postorder rank for a node v can be easily calculated.

- `post(`v`)` = `pre(`v`)` + `size(`v`)` - `level(`v`)`

2.2.2 Benefits

As previously stated, the key contribution of this work is the provision of an index structure that is capable of supporting evaluations along each of the thirteen XPath axes. This allows the XPath accelerator to stand out among related research that concentrates on the efficient processing of regular path expressions [16, 34, 43, 32, 20, 26]. Furthermore, the ability to start axis traversals from any given node in the XML data tree allows the index to support the evaluation of path traversals embedded in XQuery expressions that may not start from the document root node. Thus, the XPath accelerator overcomes two of the main issues with the DataGuide approach to XML query processing.

The modified encoding scheme has numerous advantages over the original encoding scheme. In [42], the experience of building Jungle, a secondary storage manager for the Galax XQuery system is reported. Their indexes were implemented according to the original XPath accelerator specification [17]. However, they found the overhead imposed by a postorder traversal to be significant as it greatly increased document loading costs. The Jungle implementation experience also found the evaluation of the `child` axis to be as expensive as the processing of the `descendant` axis. But the modified XPath accelerator [18] exploits the `level` and `size` attributes to boost the performance of the `child` axis.

To extract the necessary node information (i.e. preorder, level, etc) from an XML document, the XPath accelerator uses the SAX framework [23] to parse the input document set once. Unlike other parsing techniques such as DOM, a SAX parser can efficiently traverse an XML document as the machine memory requirements are not bounded by the data source's size, but by its height. Since this node information is loaded into a relational storage structure, the XPath accelerator exploits the maturity and extensive tuning of relational technology that has been achieved over the past 20 years e.g. bulk loading features and advanced indexing technology. This index structure can be queried using purely relational techniques, unlike other attempts at boosting XML query evaluation that require query processing algorithms that lie outside the relational domain e.g the *Multi-Predicate*

Merge Join (MPMGJN) [45]. Such approaches add software layers in addition to the database host, which can degrade query performance. Furthermore, its performance assessment shows promising results for the XPath accelerator as it comes close to or exceeds measurements published in related research.

2.2.3 Limitations

While the modified encoding scheme boosts the performance of axis evaluations along the `child` axis, it is not optimised as almost all descendants of a given context node need to be processed in order to identify the children of the context node. For a given axis, the XPath accelerator prunes the index search space according to preorder, postorder and axis logic. However, especially for large databases this approach can still yield a large query window with many false hits that must be filtered according to the node tests and predicates found in the query structure. For example, the `following` axis query window (see Table 2.2) ranges from its preorder and postorder values for the context node to infinity i.e. the maximum preorder and postorder values for the database. Some index structures such as the DataGuide approach exploit the metadata features of their index repository to prune the index search space. Therefore, the performance of XPath accelerator is seriously hindered by its index structure. By extending its index structure with compact metadata structures, query processing strategies can be implemented that further reduce the size of its query windows, thus boosting query performance.

2.3 eXist: A Native XML Database

The eXist database [24] is an open source project with XQuery and XPath support that provides schema-less storage for XML documents. The database is platform independent as it was completely written in Java and may be deployed as a standalone server, as an embedded database, or inside a servlet engine like Apache's Tomcat. To boost the performance of XPath and XQuery expressions, eXist's query processor is built on top of an advanced index structure that represents the basis of all query processing techniques.

2.3.1 Overview

During document parsing, eXist's numbering scheme assigns a persistent identifier to each node in the XML tree, with these unique identifiers used in indexes as a reference to the actual node. The numbering scheme employed by eXist is based on *level-order* encoding. With level-order encoding, every node at the current level is visited and assigned a level-order rank, before going to the next level. Its numbering scheme allows the query processor to evaluate the structural relationship between nodes i.e. ancestor-descendant, parent-child, etc. The index repository of eXist consists of four index files that record the structural and content information of the target document set:

- *collections.dbx* - manages the collection hierarchy.
- *dom.dbx* - associates unique identifiers to actual nodes.
- *elements.dbx* - indexes element and attribute nodes.
- *words.dbx* - indexes words and phrases.

Since an XML database can contain numerous collections of XML documents, the *collections.dbx* index is used to manage the collection hierarchy of the database. Each document and collection in the target database is assigned a unique identifier, with this index used to map the collection and document names to their respective identifier. The central component of eXist's native XML database is the *dom.dbx* index file it stores each document node in a tree structure according to the W3C's document object model (DOM). Furthermore, it is used to map each unique node identifier to its physical address in the database. Element and attribute names are mapped to their corresponding node and document identifiers in the *elements.dbx* file. This allows the query processor to efficiently retrieve all instances of a given node name. While the *words.dbx* file is an inverted index that is used to associate a word or phrase with its exact document and node identifiers. This index can only be used by eXist-specific fulltext functions that are extensions to the XPath query language (see §2.3.2).

Figure 2.2: Decomposition of path expression

Using the features of its index structure, eXist's query processor is able to use path join algorithms to evaluate path queries. To evaluate the path expression `'/dblp//mastersthesis[child::author = 'Peter Van Roy']'`, the query parser decomposes the path expression into a series of subexpressions (see Figure 2.3.1). Then the query engine evaluates each subexpression individually. For the first subexpression, eXist retrieves all node instances for the `dblp` (context nodes) and `mastersthesis` elements from elements.dbx. Since these two sets contain potential ancestor and descendant nodes, a path join algorithm must be applied to filter out all `mastersthesis` nodes that are not descendants of `dblp`. The node-set generated by this subexpression becomes the set of context nodes for the next subexpression, which is evaluated in a similar fashion to that of the previous subexpression. Access to the actual nodes, which are stored in the dom.dbx file, is not required for these types of subexpressions i.e. structural constraints. However, for value-based constraints (i.e. the last subexpression), the dom.dbx index must be traversed.

2.3.2 Benefits

Despite it's relatively short project history, eXist has a proven track record of successfully implementing XML based applications in a wide range of scenarios, dealing with different document types and data volumes e.g. FIAT deploys eXist at the core of a multilingual publishing system for technical car manuals. Furthermore, the eXist database can host up to 2^{31} documents and experimental results [24] show linear scalability of eXist's query performance with increasing storage costs. This means that query execution time grows linearly with document and database size. The XPath query language provides limited support for functions that search for a given keyword or phrase in large section of text. This lack of support was deemed unacceptable by its authors, so the eXist query engine was extended with a series of fulltext search functions that exploit the words.dbx index to yield superior results.

Its success has lead some researchers to deem it to be the leading open source native XML database on the market [1, 21] and it recently won the best XML database award at the InfoWorld technology of the year awards [44].

2.3.3 Limitations

The most serious drawback of eXist is its failure to support the entire set of XPath axes as it does not support the `following` and `preceding` axis types. As previously noted, this is a common problem with XML query processing techniques that must be overcome with the next generation of XML database products. Furthermore, eXist's numbering scheme limits the maximum size of a stored document. The limit does not depend on document size alone, but also on the total number of nodes and how well the document tree is nested i.e. it cannot support large and highly unstructured XML datasets like TreeBank [40]. This limitation differs from document to document and is impossible to compute in advance of document loading, which poses a major problem to eXist users.

Although the eXist query engine can process most of the structural relationships between nodes in its data store, its path join algorithm can be highly inefficient. For example, the ancestor-descendant path join algorithm requires every node in the descendant set to be compared with every node in the ancestor set, to compute the query results. Therefore an indexing structure is required that will minimise the number of comparisons required when evaluating structural relationships between nodes.

2.4 Path Summaries in the ToXop Query Optimiser

Although the eXist database is one of the most popular native XML databases on the open source market, its query processor is not optimised. Significant work on developing strategies for XML query optimisation is presented in [3]. Here, the authors exploit the metadata features of an extended ToXin path summary to allow the ToXop query optimiser to dramatically prune the search space of the ToX native XML database [2] and select the most suitable query evaluation technique.

2.4.1 Overview

All path summaries are based on the DataGuide approach [16] as they represent a structural summary of the label paths that occur in the target database. Path summaries can be grouped into two distinct categories: *exact path summaries* [16, 34] that record each distinct label path that occurs in an XML document and *approximate path summaries* [20, 32] that record only paths up to a certain depth in the XML data tree.

ToXin

ToXin [34] is an exact path summary as each distinct label path is represented by one node in its index tree (also known as the *ToXin tree*). Then for each ToXin node there is an *Instance table*, which records the ordered sequence of node identifiers for that label path in the XML data tree, together with parent node identifiers. Most path summaries process path queries in a bottom-up manner, while this can be highly efficient for path queries with selective predicates on its leaf node, it may be slow and cumbersome for twig queries (i.e. queries containing several path expressions). As a result, the ToXin index structure was extended with navigational (*NAV*) tables, in order to support top-down and bottom-up navigation. Furthermore, elements and attributes have their content stored in a *Value table*.

As previously highlighted in §2.1, schema information is essential for query optimisation as it allows the query processor to comprehend the database structure. Since ToXin is an exact path summary, its index tree is essentially a schema representation of the path structure of the target database. As a result, the ToXin path summary is incorporated into ToXop in order to boost query performance. Furthermore, the ToXin tree is augmented with a number of simple metadata statistics such as the number of instances and the number of distinct values for a given node name. Thus, the role of the ToXop query optimiser is to utilise the features of ToXin to identify the most efficient query processing plan for a given query.

ToXop

In General, path summaries are designed to be used as backends, that is, the XML query is processed by traversing the path summary. In [3], the schema part of the extended ToXin path summary (ToXin tree) is incorporated into the ToXop query optimiser in order to exploit metadata information, while the node instance parts (NAV, Instance and Value tables) serve as the database backend. The authors proposes two state-of-the-art query optimisation strategies that work together to reduce the query plan search space: *holistic path summary pruning* identifies portions of the database containing query results and *access-order selection*, which uses data statistics to compute an efficient query plan.

The difference between holistic and traditional schema pruning is that the traditional technique is tied to a particular path evaluation method, while with the holistic approach any path evaluation technique can be used after database pruning. Consider the path evaluation algorithm TwigStack [10] and the path query from *Example* 2.1 that returns all conference publications (i.e. `inproceedings`) whose children `author` and `year` have the values `'Jim Gray'` and `'1998'`. For each element name that appears in the query structure, TwigStack will load an encoded element stream that contains all elements of the given node name before conducting its respective query processing algorithm. However, these streams will contain many nodes that cannot form part of the query result e.g. the author stream contains all `author` elements in the database even if their parent is not an `inproceedings` element. The advantage of path pruning is that it selects element encoded streams according to the path structure of the *ToXin* tree e.g. its author stream will only contain `author` nodes that are children of a `inproceedings` node. Thus, pruning the index search space and boosting the performance of TwigStack or any other path evaluation algorithm.

Example 2.1 (Return all 1998 conference publication by Gray)

```
//inproceedings[author = 'Jim Gray'][year = '1998']
```

Path queries can be evaluated using either bottom-up or top-down evaluation.

The access-order selection optimisation strategy exploits the statistical and meta-data features of the ToXin tree to determine the most appropriate evaluation technique depending on the query and database structure. Since XML queries can contain multiple path instances, an additional role of this optimisation strategy is to use cost-based heuristics to determine the optimised order of path evaluation.

2.4.2 Benefits

The query optimisation strategies presented in [3] convert ToXop into a state-of-the-art and highly optimised XML query processor. Holistic schema pruning allows ToXop (unlike the eXist query processor) to efficiently locate the portions of the ToX database that contain the query results. Moreover, their strategy for holistic path summary pruning is highly flexible as it can be used before any stack-based path evaluation algorithm e.g. PathStack, TwigStackScan, TwigStack [10]. Similarly, the Instance table of ToXin is not restricted to a certain encoding scheme e.g. preorder, postorder, level-order, etc. It is the flexibility of this approach that allows it to be easily updated with the next generation path evaluation algorithms and to be incorporated into new and existing native XML databases.

The ToXin path summary overcomes some of the problems associated with the DataGuide approach. ToXop provides a greater support for the XPath query language as ToXin's Instance table records the parent information of each node, allowing ToXop to evaluate any parent-child relationship between nodes. Additionally, ToXin can be used for axis navigation from any given node, which allows ToXop to process XQuery expressions. Furthermore, the experimental results presented in [3] show that low cost path summaries (in terms of storage and construction costs) using simple statistics and simple cost-based heuristics achieve speedups in the same order of magnitude as high cost node indexes such as PRIX [33].

2.4.3 Limitations

Despite the advanced nature of this work at optimising XML queries, it fails to support all of the XPath axes. In fact, the level of its axis support lags behind that of the eXist database as it only supports the `descendant`, `descendant-or-self`,

`child`, `attribute` and `self` axes.

2.5 Conclusions

In this chapter, four major research projects [16, 17, 24, 3] that concentrate on the efficient evaluation of XML queries were examined with their benefits and limitations discussed at length. Thus, the aim of this chapter is to identify the key requirements for the next generation of XML query processors. These research projects are ranked in order of importance as follows:

1. **XPath Accelerator**. The most notable aspect of this review is that most XML databases fail to provide adequate support for the set of XPath axes as they tend to concentrate on the evaluation of the `child`, `descendant` and `descendant-or-self` axes i.e. regular path expressions. As a result, the XPath Accelerator [17, 18] stands out as it supports the 13 XPath axes and prunes the index search space using axis, preorder and postorder properties.

2. **Path Summaries in the ToXop**. Employ a pruning strategy that dramatically reduces the search space of the target database. This approach is highly optimised as it exploits the metadata features of its index repository with simple cost-based heuristics to guide the query processor in determining the most efficient query plan.

3. **eXist**. The eXist database is one of the most popular open source native XML databases on the market. While it does not support the entire set of XPath axes, it can process the majority of the XPath query language. However, its query processing strategy is not optimised as it does not incorporate any tree pruning techniques.

4. **DataGuide**. While the concept of pruning semi-structured data trees is a key enabler of XML query optimisation this technique is outdated and thus, unsuitable for optimising XPath query processing.

During our review of the state-of-the-art, we concentrated on techniques that prune the search space of the XML data tree and examined their ability to support

the XML query languages. In conclusion, we propose that the next generation of XML query optimisers provide full support for the set of XPath axes and incorporate a novel pruning strategy that not only uses a structural summary [16, 3], but also prunes according to axis logic [17].

Chapter 3

XPath Processing Strategy

Several of the current approaches to XML query optimisation provide inadequate support for the XPath query language as they fail to support the full set of XPath axes. Furthermore, efforts to use an index to boost query performance are hampered by the fact that XML indexes can be very large as they must contain all the structural and content information for each node in the underlying XML document. In this chapter, we present an index-based strategy for evaluating XPath queries to prune the index search space while supporting the full set of XPath axes.

The chapter is structured as follows: Section 3.1 gives an overview of XML query processing and highlights the major shortfalls of current approaches, while §3.2 provides a detailed analysis of the XPath query language. Then, §3.4 presents a strategy for evaluating location steps along each of the XPath axes. Finally in §3.5, we present the conclusions of this chapter and identify methods of optimising the current processing strategy.

3.1 Overview

Query languages such as XQuery [5] and XPath [11], use path expressions to traverse the hierarchical structure of an XML tree. A path expression detects an ordered set of nodes within an XML document that match its structural and content constraints. However, query evaluation techniques based on tree traversal become very inefficient for large XML repositories. In *Example* 3.1, the XPath expression selects

all `author` elements that are children of `phdthesis` elements that are descendants of the document root node. With conventional top-down tree traversal, the query processor must gain access to each node in the XML tree in order to locate all `phdthesis` elements as there is no way of knowing the possible location of these elements in advance. As a result every node in the XML document must be traversed to test *(i)* if the node is an element and *(ii)* if it has the tag name `phdthesis`. Then each `phdthesis` element must be checked for potential children elements with the tag name `author`.

Example 3.1 (Retrieve all the authors of PhD theses)

```
/descendant::phdthesis/child::author
```

As a result XML databases use indexes to boost query performance. During document loading, tree traversal techniques such as *preorder*, *level-order* and *postorder* encoding [39] are implemented to assign unique identifiers to each individual node in the target tree with these unique identifiers forming the basis of the indexing structure. However, indexes for native XML databases can grow to be very large as each node has a series of additional attributes (i.e. node name, node type, node value, etc). For example, the eXist native XML database [24] has an index approximately two and half times the size of the underlying document. Since large index structures degrade query performance, advanced query processing strategies must adopt an aggressive pruning strategy in order to reduce the potential search space. The XPath query language contains thirteen XPath axes (see Table 2.1). Many approaches to XML query processing fail to provide sufficient support for XPath as they concentrate on the evaluation of regular path expressions (i.e. XPath expressions that only contain `child`, `descendant`, `descendant-or-self` and `self` axes) [31, 46, 17, 43]. Such approaches justify their concentration on evaluation of regular path expressions as their axis types are deemed to be more frequently found in XPath queries than the remaining axes. It is the aim of this research to provide support for the entire set of XPath axes and to exploit axis properties and query structure in order to efficiently prune the potential search space.

Before describing our query processing framework, it is necessary to provide an overview of the XPath query language. This analysis explores the semantics of XPath and highlights issues still outstanding in the provision of an adequate XML query processing strategy. Furthermore, we introduce the concept of XML numbering schemes, which form the cornerstone of all native XML approaches to storage and query processing as they provide fast access to XML instances.

3.2 XPath

XPath [11] is a navigational language that recursively queries an XML document as a tree of nodes and returns a sequence of matching subtrees rooted at the target nodes.

Location Path

The core of the XPath language is the *location path.* A location path is a path expression that specifies a tree traversal operation. The result of evaluating a location path is the node set containing the nodes selected by its tree traversal. This node set can be transformed by the XPath core functions which allow the set of XPath 1.0 expressions [11] to be partitioned into four disjoint categories based on their respective return types:

1. Boolean Queries - Returns false if the node set is empty.

   Example 3.2 (Return the boolean result for the set of masters thesis titles from 1984)

   ```
   fn:boolean(/descendant::mastersthesis[year='1984']/title)
   ```

2. Numeric Queries - Returns the cardinality of node set.

   Example 3.3 (Return the number of master thesis titles from 1984)

   ```
   fn:count(/descendant::mastersthesis[year='1984']/title)
   ```

3. String Queries - Returns the string-value (i.e. a sequence of zero or more characters) for the first node of the node set. If the node set is empty, an empty string is returned.

Step	Axis	NodeTest	Predicates
1	descendant	mastersthesis	child::year = '1984'
2	child	title	

Table 3.1: Location Steps for *Example* 3.5

Example 3.4 (Retrieve the string-value for the first masters thesis title from 1984)

```
fn:string(/descendant::mastersthesis[year='1984']/title)
```

4. Node Set Queries - Returns a subtree for each XML node identified by the location path.

   Example 3.5 (Retrieve the title of each masters thesis from 1984)

   ```
   /descendant::mastersthesis[year='1984']/title
   ```

Location Steps

A location path contains a series of *location steps* syntactically separated by the forward slash symbol (`'/'`), with each location step selecting a node set relative to the context node. These nodes become the context nodes for the next step. The node set returned by a location path is the set of nodes remaining after each step has been processed.

Each location step contains an *axis*, a *nodetest* and zero or more *predicates*. Table 3.1 shows the location steps of the location path described in *Example* 3.5. The *axis* describes the tree relationship between the context node and the nodes selected by the location step i.e. the direction of travel. A *nodetest* specifies the node type and name of the nodes returned by the location step. *Predicates* are enclosed in square brackets and filter the set of nodes returned by the nodetest. For example, the axis type of the first location step in Table 3.1 selects the set of all descendants of the context node. These intermediate results are then filtered by (*i*) its nodetest as it requests all nodes to have the name `mastersthesis` and (*ii*) its predicate that requires all nodes returned by this step to have a child node named `year` with the given value.

3.3 Node Numbering

During document parsing an XML numbering scheme assigns a persistent identifier to each node in the tree in order to record the structure of the underlying document e.g. *preorder* encoding [17, 31, 39]. With preorder traversal, each node v in the document tree is visited in document order and assigned a preorder rank before its children are recursively traversed from left to right. Since preorder traversal provides unique identifiers for XML nodes it can form the basis of an XML index structure to improve access to XML data. Many of these numbering schemes fail to provide sufficient support for the set of XPath axes [17, 31, 43, 46]. Even numbering schemes that support the thirteen XPath axes can suffer from additional drawbacks. For example, the XPath Accelerator [17] has a high construction overhead as highlighted in [42]. The PreLevel index structure [31] overcomes these issues as its numbering scheme provides full support for XML tree traversals along each of the thirteen XPath axes and as we demonstrated in [29], it can be efficiently constructed during a single document pass.

The PreLevel index is based on *preorder traversal* and *level* encoding of XML nodes. Level encoding records the depth of each node in the XML tree. This index exploits the properties of level rank node encoding in order to boost the performance of XML queries along each of the thirteen axes. Figure 3.1 illustrates the preorder and level encoding of the sample XML document provided in *Example* 1.1. The PreLevel index contains a base index and an inverted level index that support the evaluation of level based queries e.g. *level(7)* $= 1$ and *level(12)* $= 2$. However, the query processing techniques provided by the PreLevel index [31] are inefficient as the whole XML data tree must be scanned in order to retrieve the required node set.

3.4 Location Step Evaluation Strategy

This section provides a high-level overview of our strategy for evaluating location steps. This strategy was published in [36] and supports axis traversals along each of the thirteen XPath axes relative to a given set of context nodes. We then describe

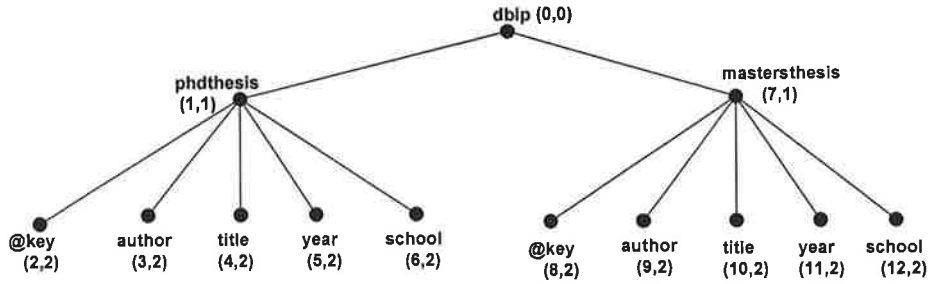

Figure 3.1: PreLevel encoding of sample DBLP XML document

in detail our strategy for processing location step against the *major* XPath axes (i.e. `ancestor`, `descendant`, `following` and `preceding`). For reasons of space, we provide a detailed discussion of our strategy for the remaining axes in [28]. The query processor uses a top-down evaluation strategy to process the set of ordered nodes that support a given path expression. Our query processing framework begins by parsing the XPath expression, which breaks its location path into a series of location steps and decomposes these location steps into their respective axis, nodetest and predicate types (see Figure 3.2). The evaluation of the first location step produces a set of preorder nodes that form the set of context nodes for the next location step. All remaining location steps are evaluated according to their axis type, with the result of the final step returned as the overall result for the location path.

In brief, our query optimisation strategy comprises five processes that prune and filter the search space using query structure, axis properties and preorder encoding logic. Since a location path may contain multiple location steps, processes are repeated for each individual location step with the output of Process 5 (a set of arbitrary nodes) becoming the input for Process 1 (a set of context nodes) in the next iteration.

The first three processes are axis-dependant, while processes 4 and 5 are common to all axes. Processes 1 and 2 are fast as they never traverse the tree but use an index to eliminate preorder values that cannot form part of the result set. Processes three to five require visiting each node in the appropriate set. Thus, they are performed in a single traversal but are separated here for the purpose of discussion.

- Process 1: **Context Prune by Structure**. The role of this process is to

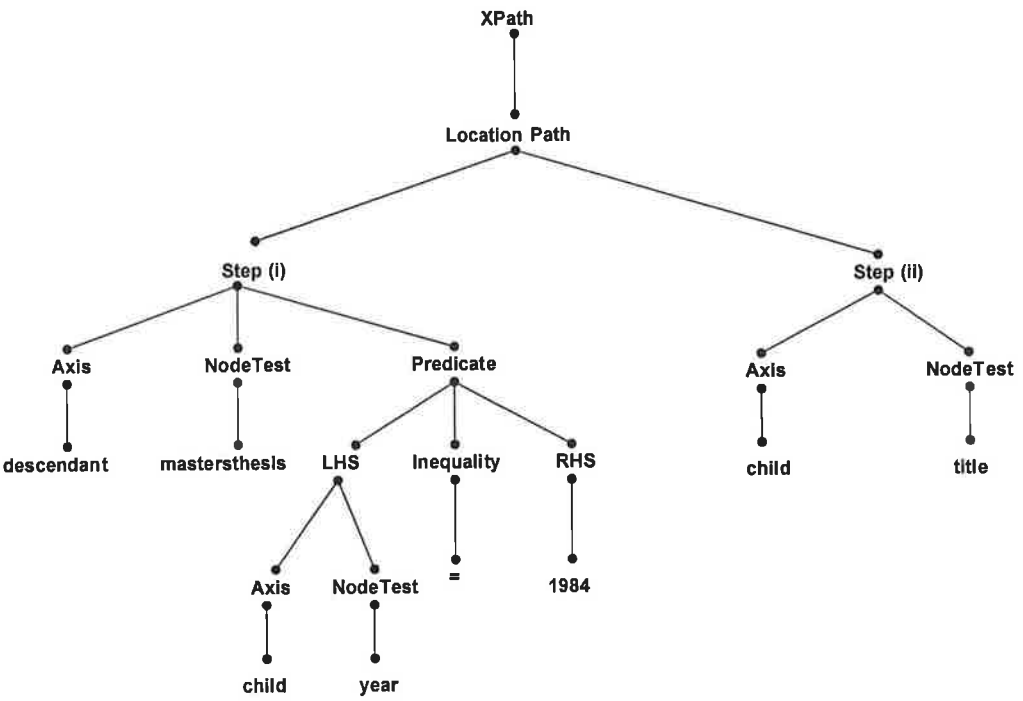

Figure 3.2: Parse Tree for the XPath query in *Example* 3.5

eliminate redundant context nodes. This greatly improves the efficiency of our query processing strategy as processes 2-5 must be executed for each individual context node before the next location step can be processed. Process 1 prunes the current set of context nodes using *(i)* axis logic, *(ii)* the node name from the current location step and *(iii)* preorder logic. For example, if the location step includes the `preceding` axis, then context nodes preceding the last context node are pruned as they occur before it in document order. The initial input (first pass) to this process is the document root node. The input to all remaining passes is the output from Process 5 below. The output from this process is often a far smaller set of context nodes.

- Process 2: **Arbitrary Prune by Structure**. As input, Process 2 receives the set of context nodes generated by Process 1, the axis type and the node name for the location step. Then, for each context node this process uses axis logic to prune the entire XML tree (initial set of arbitrary nodes), to identify a subset of the target tree (pruned set of arbitrary nodes) that contains the

results for the current location step. The results will differ for all axes and some axes (e.g. `following`) can be very large as it may include all nodes in the target document *after* the context node. At this point we have a reduced set of both context and arbitrary nodes for the current location step.

- Process 3: **Arbitrary Axis Test**. The input is the set of arbitrary nodes generated by Process 2 and the axis type for the location step. While the node set contains all the nodes requested by the relevant location step, it may also contain nodes that are not allowed under a given axis definition (due to the fast process used to generate the set). Therefore the *axis* test provides a secondary pruning step.

- Process 4: **Structure Filter**. The input is the set of arbitrary nodes generated by Process 3 and the nodetest for the location step. The role of this process is to filter the arbitrary node set by removing all nodes whose (attribute) name does not match the nodetest (also known as the arbitrary node name). Process 4 is ignored in wildcard cases e.g. return all children of a given node.

- Process 5: **Predicate Filter**. The filter reduces the set of arbitrary nodes produced by Process 4, if the location step has a predicate. Otherwise, this process is ignored.

The following subsections detail our location step evaluation strategy along each of the *major* XPath axes. For each axis, we use sample queries based on a large XML dataset to justify our five step strategy to processing location steps relative to a given set of context nodes in preorder and level encoding. DBLP [40] is a popular computer science bibliography in an XML format, containing a single large document (127 MB) with 3,332,130 elements, 404,276 attributes and 6 levels. Document order is assigned through a PreLevel encoding [31, 29] of the XML data tree, which results in a preorder set (i.e. the initial set of arbitrary nodes) ranging from 0 to 3,736,376.

Step	Context Nodes	Axis	NodeTest	Predicates
1	0 (root)	descendant	phdthesis	

Table 3.2: Location Steps for Descendant Query

3.4.1 Descendant Axis

The `descendant` axis returns all descendants of the context node. A DBLP database query that uses the `descendant` axis is one that retrieves all PhD theses together with associated information (see *Example* 3.6). In XML tree terms, this means "return all `phdthesis` elements that are descendants of the root node". By default, when you retrieve a node (such as `phdthesis`) you will also return the subtree rooted at that node (i.e. its descendants and attributes). In Table 3.2, this query is shown to have a single location step.

Example 3.6 (Retrieve all PhD thesis elements)

```
/descendant::phdthesis
```

- **Process 1: Context Prune by Structure.** For a `descendant` axis, this process reduces the context node set by retrieving the ending preorder value for the node name from the current location step. Then, all context nodes with a value greater than this ending value are deleted as according to preorder and axis logic these nodes cannot support descendants with the required node name. However, this process is redundant for descendant queries when the context node is the *root* node (i.e. *Example* 3.6) as the preorder value of the *root* node (i.e. 0) is always less than or equal to then the maximum preorder value for any node name.

- **Process 2: Arbitrary Prune by Axis.** The first task is to identify the pruned set of arbitrary nodes for the `descendant` axis, which represents a subset of the entire tree that contains the results (`phdthesis` nodes) for the location step. On most occasions the set of arbitrary nodes ranges from the context node plus one, to the sum of the context node plus the size of subtree rooted at the context node. In this particular example however, we use an index lookup to determine the beginning (353) and ending (802) preorders for

Step	Context Nodes	Axis	NodeTest	Predicates
1	0	descendant	author	.='John Sieg Jr.'
2	503, 168528, 866402 ,880490, 1541990	ancestor	phdthesis	

Table 3.3: Location Steps for Ancestor Query

the `phdthesis` node as it identifies a more concise node set. As a result the arbitrary node set ranges from {353,...,802} (451 nodes).

- **Process 3: Arbitrary Axis Test.** This step filters the set of arbitrary nodes by removing all namespace and attribute nodes as they are not allowed for this axis type (see Table 2.1). By removing these nodes the arbitrary node set for *Example* 3.6 is reduced i.e. {352,354,...,801,802} (378 nodes).

- **Process 4: Structure Filter.** Using *Example* 3.6, this process filters the set of arbitrary nodes to remove all nodes whose node name does not match `phdthesis`. The resulting set of arbitrary nodes is considerably smaller as it only contains descendants of the root node that support the given node test i.e. {352, 358, 364,...,796, 802} (72 nodes).

- **Process 5: Predicate Filter.** The *predicate filter* is redundant for *Example* 3.6 as it does not contain any predicates. Thus, for this example the 72 nodes outputted by Process 4 is returned as the final result.

3.4.2 Ancestor Axis

The `ancestor` axis returns all nodes that are ancestors of the current context node. An example of a query that uses the `ancestor` axis is illustrated in *Example* 3.7. In XML tree terms, this sample query selects all `phdthesis` elements that are ancestors of an `author` element with the value `'John Sieg Jr'`.

Example 3.7 (Retrieve the PhD thesis by 'John Sieg Jr.')

```
/descendant::author[.='John Sieg Jr.']/ancestor::phdthesis
```

To explain the processing strategy for the `ancestor` axis we move to the second location step as illustrated in Table 3.3. The input to this step (nodes generated by

the first location step) is the node set {503, 168528, 866402, 880490, 1541990} i.e. all `author` element nodes having the value `'John Sieg Jr.'`.

- **Process 1: Context Prune by Structure**. For the `ancestor` axis, this process uses preorder logic and the given ancestor node name (i.e. `phdthesis`) to reduce the set of context nodes. An index lookup is executed to retrieve the first (352) and last (802) preorder instances of the ancestor node name from the XML database. According to preorder and axis logic the context node must occur after the ancestor node in document order, otherwise the location step returns an empty result set. Therefore, we delete all context nodes that are before the first instance of the given ancestor node name in the XML data tree (in this case no nodes are deleted). Furthermore, all context nodes must occur before the last instance of the given node name plus the size of the subtree rooted at that node (807). In this case, the final four context nodes are deleted as their preorder values are greater then 807. Nodes greater then this value are filtered as they cannot have ancestors with the required node name. As a result the set of context nodes becomes {503}. Note that this process is redundant if the nodetest if a wildcard.

- **Process 2: Arbitrary Prune by Structure**. Prunes the XML tree by identifying a subtree that contains all required arbitrary nodes. This subtree ranges from the minimum preorder value for the given node name to current context node minus one i.e. {352, .., 502} (151 nodes).

- **Process 3: Arbitrary Axis Test**. Filters all arbitrary nodes that are not ancestors of the current context node i.e. {501}. An index structure based on preorder and level encoding of XML trees [31], can determine if an arbitrary node is ancestor of a given context node. However, this process is inefficent as it requires each arbitrary node to be tested against each context node. By extending our index structures with additional parent information this process can be optimised (see Chapter 4).

- **Process 4: Structure Filter**. Once more this process prunes all arbitrary

Step	Context Nodes	Axis	NodeTest	Predicates
1	0 (root)	descendant	article	author='Mark Roantree'
2	3676350, 3678812, 3686988, 3687576	preceding	incollection	author='Amit P. Sheth'

Table 3.4: Location steps for Preceding Query

nodes that do not match the node name from the location step i.e. {501}.

- **Process 5: Predicate Filter.** The *predicate filter* is redundant for *Example* 3.7 as the second location step does not contain any predicates. Therefore the output from Process 4 is returned as the result.

3.4.3 Preceding Axis

The `preceding` axis returns nodes that occur *before* the context node in document order, providing they are *not ancestors* of the context node. Suppose one wanted to view all book chapters (i.e. incollection) by the author 'Amit Sheth' that preceded a specified publication by 'Mark Roantree'. This may happen if one wanted to examine those works by 'Amit Sheth' that may have contributed to a paper published by 'Mark Roantree'. A `preceding` query can be used in this case.

Example 3.8 (Retrieve articles by Sheth before articles by Roantree)

```
/descendant::article[author='Mark Roantree']/preceding::
incollection[author='Amit P. Sheth']
```

To explain the query processing strategy for the `preceding` axis we again move to location step 2. The input to this step (set of nodes generated by the first location step) is the node set {3676350, 3678812, 3686988, 3687576}. Refer to descendant axis processing for this step (see §3.4.1).

- **Process 1: Context Prune by Structure.** For a `preceding axis`, this process reduces the set of context nodes to a single context node (i.e. the node with the largest preorder value). The set of context nodes becomes {3687576}. For reasons of efficiency the pruned context nodes are removed at this early stage as they would only produce a results that are subsets of the result set achieved for the largest context node.

- **Process 2: Arbitrary Prune by Axis.** The role of this process is to generate a pruned set of arbitrary nodes from the target XML tree for the `preceding` axis based on the single context node. This process extracts the maximum and minimum preorder values for the relevant node name i.e. `incollection` from the preorder index. These values provide a beginning and ending preorder values for the set of arbitrary nodes. On most occasions, this process ignores the ending preorder value as it will be greater than the context node. However for *Example* 3.8, we use the ending position supplied by the preorder index and not the context node as the index provides a lesser value. Note that ancestors can still be in the node set at this point. For this example, the set of arbitrary nodes is {2399330, .., 2419224}. At this point the cardinality of this set is 19,895 nodes.

- **Process 3: Arbitrary Axis Test.** Similar to the `descendant` (see §3.4.1) axis, this process removes all namespace and attribute nodes from the set of arbitrary nodes. Additionally, Process 3 is also used to eliminate all nodes that are ancestors of the (single) context node (see §3.4.2). This process results in a more compact arbitrary node set i.e. {2399330, 2399332, .., 2419224} (15,562 nodes).

- **Process 4: Structure Filter.** This process filters the arbitrary node set to remove all nodes whose node name does not match `incollection` and this provides a dramatic reduction to {2399330, 2399339, .., 2419214, 2419224} (1,009 nodes).

- **Process 5: Predicate Filter.** The *predicate filter* is used to select only book chapters for 'Amit Sheth'. Resulting in the arbitrary node set {2406333, 2407129, 2407542, 2413654, 2416555}, which is returned as the query result for *Example* 3.8 as all the location steps have been evaluated.

3.4.4 Following Axis

The `following` axis returns nodes that occur *after* the context node, providing they are *not descendants* of the context node. Suppose one wanted to view all conference

Step	Context Nodes	Axis	NodeTest	Predicates
1	0	descendant	*	author='Amit P. Sheth' year='1990'
2	1482571, 3088244, 3683723	following	inproceedings	author='Mark Roantree'

Table 3.5: Location Steps for Following Query

publications (i.e. inproceedings) by the author `'Mark Roantree'` that may have been based on a 1990 publication (* for all publications) by `'Amit Sheth'`. A `following` query can be used to select all Roantree conferences publications after some specified point in the XML tree. We use *Example* 3.9 to describe our framework for evaluating location steps along the `following` axis.

Example 3.9 (Get conference publications by Roantree after 1990 publications by Sheth)

```
/descendant::*[author='Amit P. Sheth'][year='1990']/following::
inproceedings[author='Mark Roantree']
```

- **Process 1: Context Prune by Structure.** Similar to the `preceding` axis, this process reduces the set of context nodes to a single context node (the first context node i.e. the node with the smallest preorder value). Once again these nodes were pruned as they can only produce result sets that are subsets of the result outputted for the smallest context node. Thus, the set of context nodes becomes {1482571}.

- **Process 2: Arbitrary Prune by Axis.** Index lookups are executed to determine the first and last preorder instances of the given node name in the XML data tree. These values once again provide the beginning (1020) and ending (3736363) preorder values for the set of arbitrary nodes. However, we will ignore the starting preorder value in favour of the context node, if its value is less than the context node. At this point, we also eliminate descendants of the context node. On completion this process produces a set of arbitrary nodes ranging from 1482580 (context node plus its descendants) to 3736363. The cardinality of the set at this point is still very large (2,253,784 nodes).

- **Process 3: Arbitrary Axis Test.** An attribute or namespace node are not

allowed for this axis type, therefore they are filtered and the set of arbitrary nodes becomes {1482580, 1482582...3736363} (2,023,680 nodes).

- **Process 4: Structure Filter.** Removes all nodes whose node name does not match `inproceedings` i.e. {1482580, 1482592...3736322,3736363} (82,262 nodes).
- **Process 5: Predicate Filter.** The *predicate filter* is used to select only publications for Mark Roantree, giving the final arbitrary node set {1511594, 1534805, 1577671, 1577829, 1578036, 1603629, 1923099, 2058649, 2061519} (9 nodes).

As illustrated in the above example, the **Arbitrary Prune by Axis** process identifies a very large set of arbitrary nodes for this axis type. While, this results in slow query performance times (see Chapter 5), our query processing strategy still stands among related research as most approaches to XML query processing including the eXist database [24] do not support traversals along the `following` axis.

3.4.5 Index Requirements for Optimisation

The previous subsections show that our location step evaluation strategy provides full support for the *major* XPath axes. To boost query performance the **Context Prune by Structure** process prunes the set of context nodes, while the **Arbitrary Prune by Structure** process prunes the potential search space. Irrespective of the axis type these processes use the preorder values of the first and last instances of a given node name (that is obtained from the parsed location step) in the XML data tree in order to boost query performance. Extracting the required information from a large data structure can be expensive, therefore an additional metadata index containing the first and last preorder values for each distinct node name is required to further boost performance.

The set of arbitrary nodes produced by Process 2 is refined by processes 3-5 to give the location step results. Processes 3-4 often use node type and name information to achieve their goal, therefore our index structures must be extended to

provide fast access to this information. For the `ancestor` and `preceding` axes our evaluation strategy requires us to retrieve the ancestors of a given context node, which is expensive when the set of arbitrary nodes is large as each node must be tested against the context node (see Process 3 of §3.4.2). However, extending our node index with a direct pointer to its parent node allows this process to be optimised. Furthermore, the `following` and `descendant` axes require the calculation of the size of the subtree rooted a given node, extending our preorder and level index with parent and position (see 4.1.1) information allows this value to be efficiently retrieved.

3.5 Conclusions

In this chapter, we presented a detailed analysis of the XPath query language. We then presented a strategy for evaluating location steps along all the XPath axes (with the four *major* axes described in detail and the full axis set is described in [28]), providing an element of novelty among related research that concentrate on evaluating regular path expressions. We use an index structure based on preorder encoding in order to boost XML query evaluation. However XML indexes tend to be very large, which results in poor query performance. Therefore we adopted a pruning strategy in order to reduce the potential search space.

While our query processing strategy supports the set of XPath axes, it is not optimised. In §3.4.5, we outlined several issues that must be resolved in order to provide an optimised query processing strategy. The most pressing of these issues is a more efficient method of filtering the set of context nodes and pruning the database search space, as calculating the maximum and minimum preorder values for the location step node name can be expensive, especially when the database is large. Our proposed solution is to extend the index repository with concise metadata information for each distinct node name.

The restrictions imposed on our query processing strategy by the structure of a preorder based index require that we evalaute each location step individually. With this strategy, the costs associated with query evaluation are directly proportional

to the number of location steps in the path expression. Therefore the processing of arbitrarily long location paths against a large database is expensive. To optimise query evaluation we require a strategy together with a metadata repository that allows us to evaluate a series of location steps collectively depending on the query structure. By extending our repository with additional structural and metadata information, we can implement an optimised strategy to achieve this goal.

In Chapter 4, we present an extended XML repository that allows our axis-based location step evaluation strategy to be optimised as it provides fast access to XML metadata. Furthermore, its metadata features allow a novel strategy to process a series of location steps simultaneously depending on the query structure. Then in Chapter 5, we demonstrate the effectiveness of our strategy with a series of experiments.

Chapter 4

XPath Query Optimisation

In the previous chapter, we described a high-level processing strategy for individually evaluating location steps along each of the *major* XPath axes. While our framework aggressively prunes the index search space using both query structure and preorder encoding, it is not fully optimised and for certain queries it can be slow and cumbersome against large databases. In this chapter, we provided a series of detailed optimising algorithms that are based on a metamodel for XML databases that was created during the FAST project [35, 30].

Section 4.1 describes this metamodel for XML databases, which is deployed a powerful index structure that forms the basis of query optimisation techniques described here. §4.2 presents our algorithms for *optimising* query evaluation along the *major* XPath axes. Then, §4.3 presents an advanced strategy that gives us the potential to evaluate a series of location steps simultaneously. Finally, in §4.4 we offer our chapter conclusions.

4.1 XML Database Metamodel

XPath query optimisation is achieved by exploiting the features of the FAST metamodel for XML databases. This metamodel has three layers of metadata for XML datasets: (*i*) index metadata to describe the structural and content information of each node in the XML data tree (also known the *database*), (*ii*) metadata to describe the schema of the underlying database and (*iii*) meta-metadata to model schema

properties in general. The absence of an adequate schema for XML documents has lead to the construction of our schema repository. Figure 4.1 illustrates the class diagram of the metamodel for XML databases. I specified and implemented these components needed by my optimiser and presented the results in [30].

4.1.1 Index Metadata

The basis of all indexes for native XML databases is a tree traversal technique that assigns a unique identifier to each node in the target dataset. During data source parsing all nodes are assigned unique {*preorder*, *level*} pairs. In [31], they show that indexes based on preorder and level encoding of XML trees can support the evaluation of location steps along each of the thirteen axes. Furthermore, this index structure has a minimal construction overhead as demonstrated in [29], unlike indexes using a *postorder* encoding scheme [42].

The `Node` (models XML node information) and `NodeLevel` (models the levels found in XML trees) types in Figure 4.1 are used to create the Base and Level indexes of the Extended Schema Repository, as these indexes contain an entry for each `Node` and `NodeLevel` instance encountered during document parsing. An XML database is modeled as a rooted, ordered, labeled tree structure, with XML nodes found at different levels in the tree hierarchy. Since many nodes may reside at the same level, the `NodeLevel` type has a one-to-many mapping with the `Node` type.

Base Index

This index stores both the structural and content information of each XML node (`Node` type) in the underlying XML data tree. Some XML indexing techniques store structural and content information in separate indexes [3]; however, we store this information in a single index in order to limit the number of expensive joins that are required to return query results. The Base index records the:

- `Preorder` and `Level` values for each node instance. The `Preorder` value uniquely identifies each node in the target database in an ordered fashion,

while the `Level` (direct relationship to the Level index) value is used to optimise algorithms that operate at a single level in the XML tree e.g. a query using the `child` axis.

- `Parent` attribute returns the preorder value of each node's parent, unless the context node is the root node. Since each node contains a pointer to its parent, this attribute can optimise the performance of XML queries along the `parent`, `ancestor` and `ancestor-or-self` axes.

- Tag name of each context node (denoted by `Name`). With the exception of wildcard cases, location steps use a node name to select a set of nodes along a given axis. Thus, this attribute is required for location step evaluation.

- `Type` value is used to distinguish between different node types i.e. elements, attributes, namespaces, root nodes, etc. The axis parameter of a location step selects a set of nodes that belong to a certain node type (e.g. the `namespace` axis selects all namespace nodes of the current context node). Therefore, this attribute is required for the fast pruning of the XML query space. Some approaches [22] store element and attribute nodes in separate indexes; however, this results in a large number of expensive joins during query processing.

- `Value` of each element or attribute node. Indexing the content of each node supports the swift evaluation of value-based constraints (e.g. `author='Mark Roantree'`).

- `Position` attribute, which determines the position of each node (left to right) as it occurs at a given level in the XML data tree. This attribute is used to boost the performance of the Level index and the `getSizeOfSubtree` algorithm [31] that is heavily used in our strategy for axis optimisation.

- `DocID` uniquely identifies each XML document in the database as XML repositories may contain multiple document instances. For example, the Shakespeare XML dataset [6] contains the complete works of William Shakespeare i.e. 37 separate XML documents.

- `Fullpath` records the label path from the root node to the current context node. For example, consider the node in Figure 3.1 with a preorder value of 3, this node instance has a fullpath value of `'/dblp/phdthesis/author'`. This attribute allows a direct relationship between the Base index and the FullPath structure (see §4.1.2).

Level Index

The Level index was created to improve the efficiency of the Base index at evaluating location steps along each of the thirteen XPath axes, as it allows us to boost the performance of the `getSizeOfSubtree(`v`)` function that returns the size of the subtree rooted at a given node v. A detailed explanation of this algorithm can be found in [31]. As outlined in the previous chapter, this function is required for the processing of the `ancestor`, `descendant` and `following` axes with our five step query processing strategy. Calculating the subtree size at a given node is independent of the document size but rather dependent on the number of levels in the target document. An in-depth analysis of the structure of XML documents was undertaken in [25], where they reported that XML documents are relatively shallow as 99% of their query set of over 190,000 documents had less than 8 levels, therefore the `getSizeOfSubtree(`v`)` function is suitable for retrieving the subtree size of any `Node` type in the Base index. This index is essentially an inverted index for every element or root `Node` type (all other node types i.e. attributes, namespaces, etc are ignored as they always have a subtree size of one) in the Base index. For each level encountered during document parsing a `NodeLevel` type records:

- A unique identifier for the target document in the index repository i.e. `DocID`.
- The `Level` value, which specifies a one-to-many relationship between the `NodeLevel` and `Node` types as each level in an XML tree contains one or more XML nodes.
- An ordered sequence of *preorder* values for all the element and root nodes that occur at a given level in a specified XML tree. The `Position` attribute of the

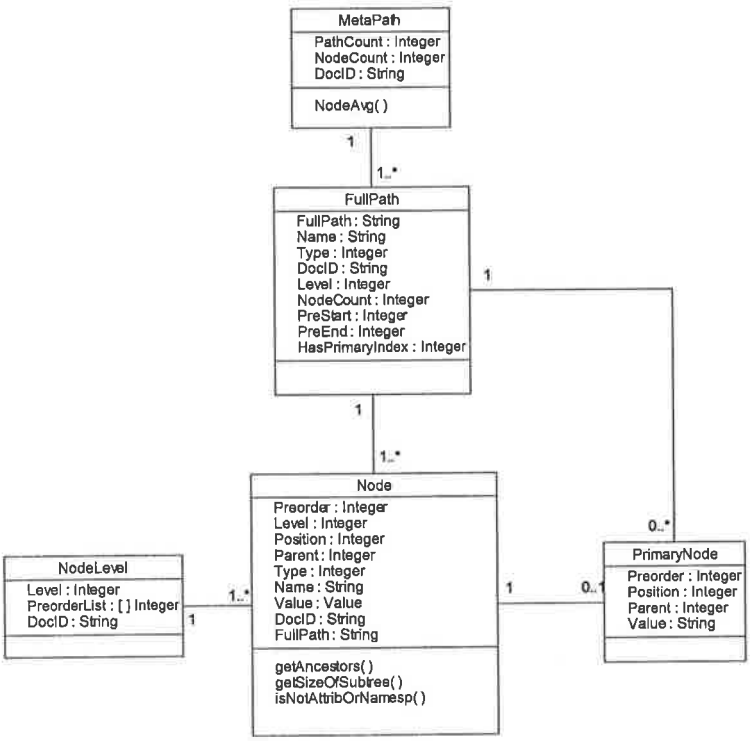

Figure 4.1: FAST XML Metamodel

Base index is the key to exploiting this index, as it facilitates a direct jump to a given preorder rank within the level index in constant time.

Primary Indexes

The `PrimaryNode` type (see Figure 4.1) is used to create the Primary indexes of the Extended Schema Repository (ESR). A `PrimaryNode` is a subset of the `Node` type as it contains the `Preorder`, `Position`, `Parent` and `Value` attributes for a given `FullPath` type. The remaining `Node` type attributes (i.e. `Name`, `Level`, etc) for a known `FullPath` instance can be derived from the FullPath structure (see §4.1.2).

Primary indexes boost query performance (as demonstrated in Chapter 5) as they are significantly smaller then the Base index, thus they can be efficiently queried. Since they only model a small portion of the target XML data tree without a strict ordering of document nodes, they are not suitable for processing individual

location steps along each of the thirteen XPath axes. However, our strategy for evaluating a series of location steps simultaneously exploits this index type (see §4.3). Primary indexes are created on FullPaths with (*i*) the highest selectivity and (*ii*) low selectivity but span large segments of the underlying XML tree. These FullPaths were chosen as commonly occurring FullPaths are highly likely to form some part of a query set, while infrequently occurring FullPaths are less likely to be found in an XPath query set they can be very expensive to process if the `FullPath` spans a large segment of the database.

Primary Index Creation - Part 1. During document parsing several statistics are recorded by the `FullPath` and `MetaPath` types (see Figure 4.1) that identify the FullPaths to be indexed. A Primary index is created on each FullPath that exceeds a threshold T_{ix}. T_{ix} is calculated by multiplying the average number of nodes for a FullPath instance (`MetaPath.NodeAvg` i.e. total number of element and attribute nodes divided by total number of FullPaths) by a variable `IndexFactor` that is currently set to 4, based on a study of XML document content. Thus, a FullPath whose value for `FullPath.NodeCount` $\geq T_{ix}$ (has at least four times the average number of nodes) will have a Primary index.

Primary Index Creation - Part 2. Furthermore, a Primary index is created on each FullPath whose cardinality is less than 1% of the total number of nodes in the XML database (i.e. `FullPath.NodeCount` $\div$ `MetaPath.NodeCount` $<$ 0.01), while also spanning over 95% of the XML data tree i.e. `(FullPath.PreEnd - FullPath.PreStart)` $\div$ `MetaPath.NodeCount` $>$ `0.95`.

Since a Primary index contains all the node instances for a given label path it is similar to a path table with the DataGuide approach [16]. However, for large data sources, the path table of the DataGuide can have very high construction costs in terms of creation time and storage. For example, the DBLP dataset [40] contains 145 distinct label paths that reference a combined total of 3,736,406 nodes. Efficent query processing cannot be conducted on a DataGuide of this magnitude. Hence, our Primary indexes only contain node set information for paths that yield the highest performance gains, thus minimising construction costs and boosting query performance. Following this approach the remaining FullPaths cannot be directly resolved

by a Primary index, instead the required nodes must be retrieved from the Base index. However, the structural properties of the FullPath structure can be exploited to prune the search space of the source database (see §4.3). In §6.2, we identify areas of future research such as the identification of the optimum `IndexFactor` and recording the sensitivity of varying the thresholds of 1% and 95% for parts one and two of our procedure for Primary index creation.

4.1.2 Schema Metadata

The `FullPath` type is a metadata equivalent of the `Node` type. In terms of optimisation, it can be used to quickly prune large segments of the Base index that are not relevant to the processing of the current location step. As this is the main construct for query optimisation, its content was heavily influenced by algorithms for the 13 XPath axes [11]. Furthermore, this index type has a statistical value e.g. number of instances for a given node name, number of node instances at a given level, etc.

FullPath Structure

The FullPath structure contains all `FullPath` instances that occur in the database. That is for each distinct label path in an XML document there is a single entry in the FullPath index. Since `FullPath` nodes are grouped together into one index node they are space efficient but still maintain the structure of the original data. For example, the DBLP database [40] has 3,736,360 element and attribute nodes, but 145 of these nodes have unique FullPath values with the FullPath `'/dblp/inproceedings/author'` having 491,783 matches. The resulting Base index contains 3,736,360 `Node` type entries, with the FullPath structure containing 145 FullPaths and each `FullPath` type has a direct mapping to one or more `Node` types in the Base index (see Figure 4.1).

The number of entries in the FullPath structure is proportional to the depth of the target tree. Since XML trees are generally shallow [25], the FullPath index is sufficiently concise for linear queries. The FullPath structure is modeled as a tree structure containing FullPaths and node names, with the *root node* at the top of the tree. Figure 4.2 illustrates how a FullPath segment has a direct mapping to the

underlying data tree (Base index), to allow us to optimise algorithms against the Base index. Along with each distinct `FullPath` instance this structure records:

- `Name` is the tag name of the leaf node of the FullPath.
- `PreStart` is the preorder value of the first instance of this path in the Base index.
- `PreEnd` is the preorder value of the last instance of this path in the Base index. The `PreStart` and `PreEnd` attributes allow our query processing strategy to prune large segments of the Base index for a given node name (`Name`) or FullPath instance, thus optimising our strategy for location step evaluation (e.g. see Algorithm `DescendantAxis`).
- `Type` is a numeric value that denotes an element, attribute, namespace, root, processing-instruction or comment node.
- `DocID` is necessary as XML datasets can contain multiple documents.
- `Level` records the depth of this path in the FullPath structure.
- `NodeCount` is the number of instances of this FullPath and is used during Primary index creation. Furthermore, in conjunction with other attributes from the FullPath index, this value allows us to retrieve a set of useful statistics such as: total number of attributes, elements and the number of instances of a specific node name in a target document or collection of documents.
- `HasPrimaryIndex` is a boolean value that is set on completion of document parsing, it returns true if there is a Primary index for the current FullPath instance, else it returns false (see Chapter 5). As a result this attribute allows a zero-to-many mapping between the `FullPath` and `PrimaryNode` types.

4.1.3 Schema Meta-metadata

The `MetaPath` type is the metadata equivalent of the `FullPath` type. It models high-level statistical data for the index repository and as a result, the schema is very small.

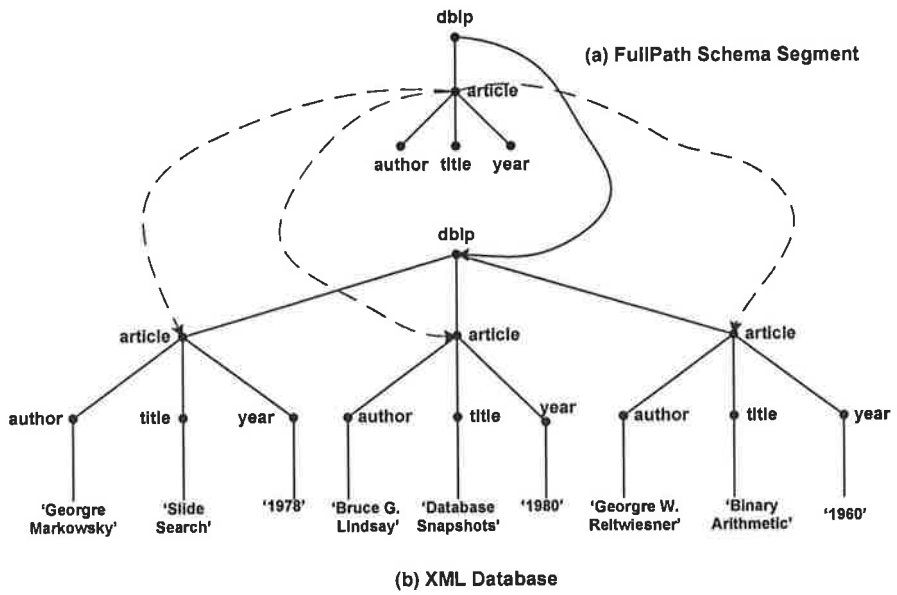

Figure 4.2: FullPath Schema to Database Mappings

MetaPath Structure

- `DocID` identifies the target XML document.
- `PathCount` records the number of FullPath types in a given document in the index repository, as a result there is a one-to-many cardinality between the `MetaPath` and `FullPath` types (see Figure 4.1).
- `NodeCount` records the total number of Node types in the target XML document. This attribute optimises `getSizeOfSubtree` [31] function as it allows fast access to the maximum preorder rank in a document tree.
- `NodeAvg` returns the average number of XML nodes per `FullPath` instance i.e. `NodeCount` divided by `PathCount`. This function is required for the construction of our Primary indexes (see Chapter 5), to boost query performance.

4.2 Axis Optimisation

In §3.4, we presented a strategy for evaluation location steps along each of the *major* XPath axes. This strategy aggressively prunes the search space of the XML tree to

locate the query results. However, the pruning process can be slow in certain cases. The algorithms presented in this section exploit the features of the FAST XML metamodel in order to achieve optimum performance of the location step evaluation strategy as its metadata features allow us the quickly prune and filter the underlying XML tree. Chapter 5 provides a detailed set of experiments that demonstrate the success (i.e. a detailed breakdown of filtering and pruning process times) of executing our optimised algorithms against the Extended Schema Repository.

As input, each axis algorithm receives an ordered set of context nodes (*conNodes*) and a parsed location step (*step*) from which we retrieve the arbitrary node name (*arbNodeName*). Each axis algorithm returns the ordered set of preorder nodes (*resultset*) that supports the current location step under the given axis type.

4.2.1 Descendant Axis

The `descendant` axis contains all nodes that are descendants of the context node (excluding attribute and namespace nodes). In the `DescendantAxis` algorithm, lines 4 and 5 perform the **Context Prune by Structure** by retrieving the maximum preorder value (*maxPre*) for the arbitrary node name and all context nodes greater then this value are deleted. Unlike the Base index, the FullPath index stores metadata and is very compact. Thus, it contains only a single index entry for each unique FullPath in the XML tree and as a result, it can efficiently return the first and last preorders for any given node name. Allowing us to optimise the performance of the first two processes of our location step evaluation strategy (see §3.4).

Processes 2 to 5 are then executed for each context node (lines 6 to 34). Lines 9 to 18 execute the **Arbitrary Prune by Axis** with variables `startSS`, `endSS` and `minPre` containing the preorder values. This process extracts a subset of the entire XML tree (delimited by `startSS` and `endSS`) that contains the query results (line 18). Initially the subtree ranges from the current context node plus 1 to the current context node plus the size of the subtree rooted at the context node (lines 9 and 10). Using our XML database metamodel the `getSizeOfSubtree` function can efficiently calculate the size of a subtree rooted at a given node. However, the initial beginning and ending preorder values of this subtree are updated, if the FullPath

structure can identify a more precise subtree (lines 11 to 17).

Algorithm 1 Returns all nodes that support a location step with a `descendant` axis

Name: DescendantAxis
Given: A sorted set of context nodes (*conNodes*),
a location step (*step*)

```
 1: SortedSet resultset = null
 2: String arbNodeName = getNodeName(step)
 3: // Context Prune by Structure
 4: int maxPre = Max(FullPath.getPreEnd(arbNodeName))
 5: conNodes.removeGreaterThen(maxPre)
 6: while conNodes != null do
 7:    int conPre = conNodes.getFirst() // get next context node
 8:    // Arbitrary Prune by Axis
 9:    int startSS = conPre + 1
10:    int endSS = conPre + Base.getSizeOfSubTree(conPre)
11:    int minPre = Min(FullPath.getPreStart(arbNodeName))
12:    if startSS < minPre then
13:       startSS = minPre
14:    end if
15:    if endSS > maxPre then
16:       endSS = maxPre
17:    end if
18:    SortedSet arbNodes = all nodes in interval [startSS, endSS]
19:    while arbNodes != null do
20:       int arbPre = arbNodes.getFirst() // get next arbitrary node
21:       // Arbitrary Axis Test
22:       if Base.isNotAttribOrNamesp(arbPre) then
23:          // Structure Filter
24:          if Base.getName(arbPre).equals(arbNodeName) then
25:             // Predicate Filter
26:             if supportsPredicate(step, arbPre) then
27:                resultset.add(arbPre) // if no predicate, always add to resultset
28:             end if
29:          end if
30:       end if
31:       arbNodes.remove(arbPre)
32:    end while
33:    conNodes.remove(conPre)
34: end while
35: return resultset
```

The node set identified by process 2 is refined by the next three processes to provide the results for a location step along the `descendant` axis (lines 19 to 32). The **Arbitrary Axis Test** (line 22) uses the `isNotAttribOrNamesp` (see Algorithm 2), which uses the `type` attribute of the Base index to efficiently filter all attribute and namespace nodes from the node set produced by process 2, as these nodes are not allowed under the axis definition. Then, the **Structure Filter** (line 24) uses

the `Name` attribute from the Base index to eliminate all preorder nodes whose node name does not match the query structure. Finally, the **Predicate Filter** (line 26) removes all nodes that do not support a given predicate (note: processes 4 and 5 are identical for all axis types).

Algorithm 2 Returns false if the node is an attribute or namespace, else returns true

Name: isNotAttributeOrNamesp
Given: A preorder node *pre*
Called By: `DescendantAxis`, `PrecedingAxis` and `FollowingAxis`

```
1: Boolean result = true
2: if Base.getType(pre) == ATTRIBUTE then
3:     result = false
4: else if Base.getType(pre) == NAMESPACE then
5:     result = false
6: end if
7: return result
```

4.2.2 Ancestor Axis

The `ancestor` axis contains all ancestors of the context node. Key constructs of the ESR allow the `AncestorAxis` algorithm to enhance our strategy for evaluating ancestor queries. Firstly, the FullPath structure facilitates the optimisation of the **Context Prune by Structure** process (lines 4 to 8) as it supplies fast access to node metadata that allows us to prune the set of context nodes. These preorder nodes are eliminated as they cannot have ancestors with the arbitrary node name supplied by the given location step (see §3.4.2). The remaining processes are then executed for each context node (lines 9-25).

Secondly, the `Parent` attribute of the Base index allows us to boost the performance of the **Arbitrary Prune by Axis** process. As outlined in §3.4.2, this process can produce a large node set depending on the query and database structure, with many of these nodes not supporting the `ancestor` axis e.g. attribute nodes. As a result, an **Arbitrary Axis Test** is required to remove these nodes. However, as demonstrated in Chapter 5, the ESR allows this process to be optimised (line 12) as each Node type has a pointer to its parent node, allowing the `getAncestors` function (see Algorithm 6) to quickly traverse through the XML data tree and return the ancestors of a given node. Since XML documents tend to be shallow, the

node set produced by this process is small. The **Arbitrary Axis Test** is now redundant as all the nodes identified by process 2 are in fact ancestors of the current context node. Then for each node returned by process 2 the **Structure Filter** and **Predicate Filter**, are performed in a single traversal (lines 13-18).

Algorithm 3 Returns all nodes that support a location step with a `ancestor` axis

Name: AncestorAxis
Given: A sorted set of context nodes (*conNodes*),
a location step (*step*)

```
 1: SortedSet resultset = null
 2: String arbNodeName = getNodeName(step)
 3: // Context Prune by Structure
 4: int minPre = Min(FullPath.getPreStart(arbNodeName))
 5: int maxPre = Max(FullPath.getPreEnd(arbNodeName))
 6: maxPre = maxPre + Base.getSizeOfSubTree(maxPre)
 7: conNodes.removeLessThen(minPre)
 8: conNodes.removeGreaterThen(maxPre)
 9: while conNodes != null do
10:    int conPre = conNodes.getFirst()
11:    // Arbitrary Prune by Axis
12:    SortedSet arbNodes = Base.getAncestors(conPre)
13:    while arbNodes != null do
14:       int arbPre = arbNodes.getFirst()
15:       // Structure Filter
16:       if Base.getName(arbPre).equals(arbNodeName) then
17:          // Predicate Filter
18:          if supportsPredicate(step, arbPre) then
19:             resultset.add(arbPre) // if no predicate, always add to resultset
20:          end if
21:       end if
22:       arbNodes.remove(arbPre)
23:    end while
24:    conNodes.remove(conPre)
25: end while
26: return resultset
```

4.2.3 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 attribute, namespace and ancestor nodes of the context node. For the `PrecedingAxis` algorithm, the **Context Prune by Structure** is executed at line 4 (see Algorithm 4) where this process reduces the set of context nodes to a single node i.e. the context node with the largest preorder value. Lines 6 to 11 perform the **Arbitrary Prune by**

Axis (see §3.4.3) and select a subset of the entire XML tree. Once more, we utilise the FullPath structure to optimise the performance of this pruning process, as it can quickly return the maximum and minimum preorders for the node name from the location step. The **Arbitrary Axis Test**, **Structure Filter** and **Predicate Filter** processes filter the arbitrary node set returned by process 2 to produce the final result set (lines 16 to 21). With the **Arbitrary Axis Test** we eliminate all attributes, namespaces and ancestors (retrieved by `getAncestors` function) of the context node from the node set produced by process 2. As will be shown in our experimental results (Chapter 5), the optimising features of this algorithm allow for rapid processing along the `preceding` axis, while many popular approaches to XML query processing (e.g. eXist [24]) fail to support this axis type.

Algorithm 4 Returns all nodes that support a location step along the `preceding` axis

Name: PrecedingAxis
Given: A sorted set of context nodes (*conNodes*),
a location step (*step*)

```
 1: SortedSet resultset = null
 2: String arbNodeName = getNodeName(step)
 3: // Context Prune by Structure
 4: int conPre = conNodes.getLast()
 5: // Arbitrary Prune by Axis
 6: int startSS = Min(FullPath.getPreStart(arbNodeName))
 7: int endSS = Max(FullPath.getPreEnd(arbNodeName))
 8: if endSS > conPre then
 9:    endSS = conPre
10: end if
11: SortedSet arbNodes = all nodes in interval [startSS, endSS]
12: SortedSet ancestors = Base.getAncestors(conPre)
13: while arbNodes != null do
14:    int arbPre = arbNodes.getFirst();
15:    // Arbitrary Axis Test
16:    if Base.isNotAttribOrNamesp(arbPre) & !ancestors.contains(arbPre) then
17:       // Structure Filter
18:       if Base.getName(arbPre).equals(arbNodeName) then
19:          // Predicate Filter
20:          if supportsPredicate(step, arbPre) then
21:             resultset.add(arbPre) // if no predicate, always add to resultset
22:          end if
23:       end if
24:    end if
25:    arbNodes.remove(arbPre)
26: end while
27: return resultset
```

4.2.4 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 descendants of the context node. In the `FollowingAxis` algorithm, we highlight each of the five processes (see Algorithm 5). Once again, the key enablers of axis optimisation are the FullPath structure and the `getSizeOfSubtree` function of the Base index, which support rapid pruning of the XML tree. Furthermore, the `isNotAttribOrNamesp` and `getName` functions allow the set of arbitrary nodes to be efficiently filtered in order to provide the location step results. Using a wide range of experiments, Chapter 5 demonstrates the success of the `FollowingAxis` algorithm with some of the query times broken down into their individual process times (e.g. time for **Context Prune**) in order to support further evaluation of our axis algorithms.

Algorithm 5 Returns all nodes that support a location step along the `following` axis

```
Name: FollowingAxis
Given: A sorted set of context nodes (conNodes),
       a location step (step)
 1: SortedSet resultset = null
 2: String arbNodeName = getNodeName(step)
 3: // Context Prune by Structure
 4: int conPre = conNodes.getFirst()
 5: // Arbitrary Prune by Axis - lines 6 to 11
 6: int startSS = Min(FullPath.getPreStart(arbNodeName))
 7: int endSS = Max(FullPath.getPreEnd(arbNodeName))
 8: if startSS < conPre then
 9:    startSS = conPre + Base.getSizeOfSubtree(conPre)
10: end if
11: SortedSet arbNodes = all nodes in interval [startSS, endSS]
12: while arbNodes != null do
13:    int arbPre = arbNodes.getFirst();
14:    // Arbitrary Axis Test
15:    if Base.isNotAttribOrNamesp(arbPre) then
16:       // Structure Filter
17:       if Base.getName(arbPre).equals(arbNodeName) then
18:          // Predicate Filter
19:          if supportsPredicate(step, arbPre) then
20:             resultset.add(arbPre) // if no predicate, always add to resultset
21:          end if
22:       end if
23:    end if
24:    arbNodes.remove(arbPre)
25: end while
26: return resultset
```

Algorithm 6 Returns all ancestors of a given node

Name: AncestorAxis
Given: A preorder node *pre*
Called By: `AncestorAxis` and `PrecedingAxis`

```
1: SortedSet resultset = null
2: if pre == 0 then
3:    return resultset
4: else
5:    int parent = Base.getParent(pre)
6:    resultset.add(parent)
7:    while parent != 0 do
8:       parent = Base.getParent(parent)
9:       resultset.add(parent)
10:   end while
11:   return resultset
12: end if
```

4.3 Simultaneous Processing of Location Steps

As detailed in chapter 3, the core component of the XPath language is the *location path*, which is broken down into a series of *location steps*. Our query optimiser uses a top-down evaluation strategy to determine the node sequence that supports a given location path. Conventional top-down evaluation strategies [24, 17] for XML query processing select all the root nodes of stored documents and use these values as the context nodes for the first location step. Their respective location step evaluation technique is then executed for each location step in left to right fashion, with the nodes produced by the final location step constituting the query result. With this strategy, the costs associated with query evaluation are directly proportional to the number of location steps i.e. each location step must be evaluated individually. Therefore the processing of arbitrarily long location paths against a large database is expensive.

Example 4.1 (Return all mastersthesis authors)

```
/descendant::mastersthesis/child::author
```

The Extended Schema Repository and in particular the FullPath structure contains sufficient metadata constructs to allow us to bundle a series of location steps together during query evaluation, thus reducing the number of location steps that

must be individually processed. The remaining location steps are then processed according to the strategy outlined in §3.4. Example 4.1 is taken from our experiment set in Table 5.3 and queries the DBLP dataset [40] to select all `author` nodes that are children of `mastersthesis` nodes, which are descendants of the root node. This query is used to motivate our strategy for simultaneous processing of location steps. In brief, our framework for the simultaneous evaluation of location steps comprises of five processes (that have no correlation to our strategy for the fast evaluation of individual location steps). Processes 1 to 3 are very fast as they never traverse the data tree (unlike Processes four and five) but instead exploit the semantics of the query structure and schema tree to identify index segments that contains the results for a series of location steps. Processes 4 and 5 filter these segments to return the query results.

- Process 1: **Identify Context Step**. The role of this process is to identify two or more simple location steps (i.e. a simple subexpression) that can be simultaneously evaluated, with all remaining location steps processed according to the baseline strategy described in Sections 3.4 and 4.2. These location steps range from the the first location step to the *context step*. The context step is identified by traversing each location step in a given query parse tree from left to right and testing if it supports property 4.1. Using *Example* 4.1, the second location step (i.e. `child::author`) is identified as the context step. Processes 2 to 5 will allow us to evaluate the first two location steps simultaneously.

  Property 4.1 (Identify Context Step - Part I)
  We define the context step as the right most step in the parse tree with an axis of type `namespace`, `attribute`, `child`, `descendant`, `descendant-or-self` *or* self, *while supporting property 4.2.*

  Property 4.2 (Identify Context Step - Part II)
  The location steps preceding the context step cannot contain predicates and the context step cannot be the first location step.

- Process 2: **Identify FullPath**. This process takes the context step and its

preceding location steps as input. The schema tree returns the FullPath instance(s) that correspond to these location steps. Since this process may return a series of FullPath instances, processes 3 to 5 are repeated for each FullPath instance. With *Example* 4.1, this process identifies the FullPath instance `'/dblp/mastersthesis/author'`.

- Process 3: **Index Identification**. Then, a suitable index is identified. If the FullPath instance does not map to a Primary index we use the Base index delimited by the beginning (`PreStart`) and ending (`PreEnd`) preorder values for the FullPath instance. The FullPath `'/dblp/mastersthesis/author'` does not map to a Primary index, therefore the Base index is selected within the range 3 to 810 (808 nodes).

- Process 4: **FullPath Filter**. This process merely retrieves all preorder instances that match the FullPath instance from the index identified by Process 3. For *Example* 4.1, this process selects all nodes from the Base index within the search range identified by Process 3 that have a FullPath value of `'/dblp/mastersthesis/author'` i.e. {3, 9, 485, 632, 810} (5 nodes).

- Process 5: **Predicate Filter**. If the context step has a predicate, this filter reduces the set of preorder nodes produced by Process 4, otherwise this process is ignored. Thus, for this example the 5 nodes outputted by Process 4 are returned as the result.

4.4 Conclusions

In this chapter, we presented an index structure that is incorporated inside the FAST schema repository. This metamodel for XML databases, incorporates three layers of abstraction for XML trees i.e. index metadata, schema metadata and schema meta-metadata. Key constructs of our index system allow us derive algorithms that optimise the performance of location steps along each of the *major* XPath axes, with algorithms for the entire set of XPath axes described in [28]. Since the "non *major*" axes are subsets or supersets of the *major* axes there was no outstanding issues to be

found in terms of optimisation for the remaining axes. XML schema metadata is the key to query optimisation as it allows for an efficient means of pruning the underlying XML data tree in order to reduce the potential search space, thus boosting query response times. While these algorithms are highly optimised, the evaluation of an arbitrarily location path can be expensive especially against a large database, as each location step must be individually evaluated. However in §4.3, we derived a procedure that facilitates a series of location steps to be processed simultaneously.

The key features presented in this chapter (i.e. index structure, optimising algorithms and the simultaneous processing of location steps) allow for the optimised performance of XPath queries. These features provide an element of novelty among related research as many XML databases fail to provide adequate support for the entire set of XPath axes and location steps are generally processed individually. In the next chapter, we provide a detailed set of experiments that demonstrate the performance of our query processing framework against a state-of-the-art native XML database.

Chapter 5

Experiment Results

In previous chapters, we described an optimised processing strategy for the XPath query language that supports the thirteen XPath axes. Optimising algorithms for the major XPath axes were presented and our underlying index structure was discussed in detail. In this chapter, we implement our index repository and axis algorithms in order to demonstrate the usefulness of our optimised XPath query processing strategy. Using a set of detailed experiments, we analyse our optimised processing framework against that of a number of state-of-the-art native XML databases.

The chapter is structured as follows: 5.1 presents a processing framework for building the ESR with a set of ESR construction times for a series of popular XML datasets. Then in Sections 5.2 to 5.4, using a large XML query set, we present our results for our optimised XPath evaluation strategy. Finally, in §5.5 we offer our some conclusions regarding these experimental results.

5.1 Extended Schema Repository Construction

While our work focuses on providing fast support for the evaluation of XPath queries along the XPath axes, further innovation stems from the development of a technique to provide a fast method for creating the ESR. In this context, we demonstrated a fast method of creating the Base index for an XML dataset [29]. We now describe the processing framework that allows for the fast creation of the entire ESR. The architecture of the FAST XML document processor is illustrated in Figure 5.1 and

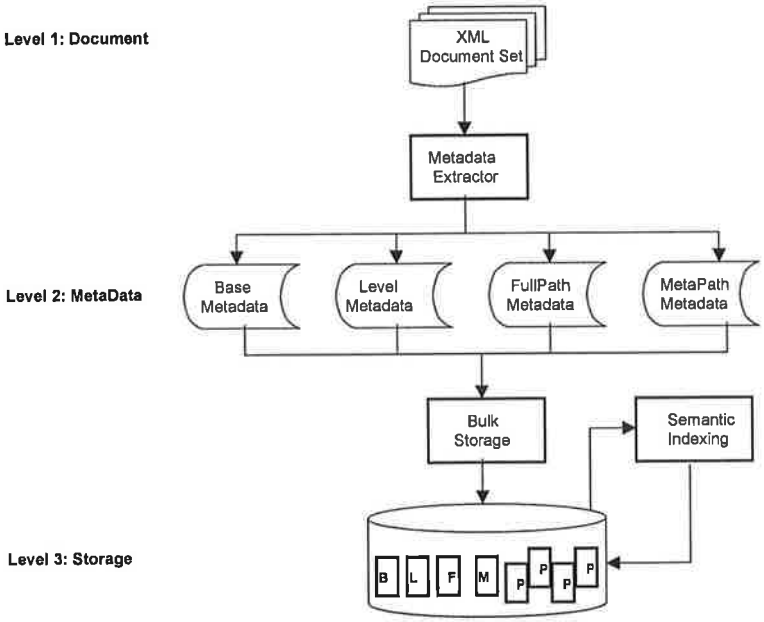

Figure 5.1: FAST XML Document Processing Architecture

has three levels: *document* level, *metadata* level and *storage* level. The remainder of this section describes the role of the processors connecting the levels and at the end of this section, we present the construction times for a range of XML datasets.

- **Metadata Extraction**. The *Metadata Extractor* processes the XML dataset from level 1 to generate a set of metadata documents at level 2. These documents contain all of the metadata information (see §4.1) required for the population of the Base, Level, FullPath and MetaPath indexes. This metadata is extracted from the target document set using an enhanced SAX parser [23]. Since XML documents are generally shallow [25], SAX is ideal for parsing them as its temporary memory requirements are bounded by a documents height, unlike DOM parsers, which are bounded by the document size.

- **Bulk Storage Processor**. XML indexes tend to be very large, as each XML node contains one or more entries in the underlying index structures. Thus, building XML indexes can be time-consuming. Using the bulk loading features of an Oracle 10g database, it is possible to minimise the time required for large amounts of XML metadata to be inserted and indexed. For example, the Base

index for the DBLP dataset [40] can be populated with over 3,700,000 node entries in only 198 seconds (see §5.1.1). The *Bulk Storage* processor (Oracle SQL*Loader) provides a means of bypassing time-consuming SQL insert commands to bulk load the metadata documents from level 2 (see Figure 5.1) into the ESR.

- **Semantic Indexing Processor**. This processor is used to create a set of Primary indexes that boost the performance of our XPath query processing strategy (see Sections 5.2 to 5.4). As described in §4.1.1, primary indexes are created on FullPaths with (*i*) the highest selectivity and (*ii*) low selectivity but span large segments of the underlying XML tree. These FullPaths were chosen as commonly occurring FullPaths are more likely to form some part of a query set, while infrequently occurring FullPaths are less likely to found in an XPath query set they can be very expensive to process if the FullPath spans a large segment of the XML data tree.

5.1.1 ESR Deployment

In this section, we present the ESR construction times for a series of XML datasets that are used again in Sections 5.2 to 5.4, where we perform a comparative analysis of our query processing strategy against that of a number of XML databases.

Experimental Setup

Experiments to create and query (Sections 5.2 to 5.4) the Extended Schema Repository ran on a 3.2GHz Pentium IV machine with 1GB of RAM using a Windows XP professional operating system. These procedures were implemented in Java using Sun's jre1.5.0_06 and an Xerces SAX parser [14] was employed by the metadata extractor. The Extended Schema Repository was deployed using an Oracle 10g database running on a Windows XP professional platform with 3.0GHz Pentium IV processor and 1GB of RAM. In [19], their experimental results return warm cache numbers and we follow this approach by executing each ESR experiment (including those in Sections 5.2 to 5.4) eleven times and discarding the first result from the

Name	Size	Elements	Attributes	Levels	FullPaths	Tags
DBLP	127.0MB	3,332,130	404,276	6	145	40
Shakespeare	7.6MB	179,689	0	6	57	21
XMARK	1.1MB	17,132	3,919	12	454	77

Table 5.1: XML dataset characteristics

average result time.

XML Datasets

The datasets used in our experiments are presented in Table 5.1. The DBLP, Shakespeare and XMARK datasets were chosen as they are of varying sizes and tree structures. Furthermore, they are among the most popular XML data sources used by related research [24, 17, 3, 7, 10, 26]. As previously described in Chapter 3, DBLP [40] is a large (127 MB), yet shallow (6 levels) XML document containing millions of element and attribute nodes (3,736,406) that describe a computer science bibliography. DBLP has a regular structure, as it only contains 40 distinct element and attribute tag names and only 145 distinct FullPaths are to be found in its data tree. This means that different types of DBLP FullPaths repeat themselves many times e.g. FullPath `'/dblp/article/title'` occurs 111,609 times.

Shakespeare [6] is another shallow XML dataset of a regular structure, containing 37 XML documents with each document describing a Shakespearean play. While, the first two datasets model data from the real world, XMARK [37] is a synthetic dataset that models data from an imaginary Internet auction. We used the *xmlgen* tool [37] with an input factor of 0.01 to automatically generate a small XMARK document of 1.1 MB. Unlike the the first two datasets, XMARK has a deep data tree (12 levels) and a more irregular tree structure, as each FullPath has relatively few instances in its XML data tree, when compared to DBLP and Shakespeare.

ESR Construction Times

Table 5.2 displays our reasonably fast ESR construction times for the DBLP, Shakespeare and XMARK datasets. Furthermore, we fragment the total creation cost for a given dataset into the time required for document parsing and creation of each

Name	Parse	Base	Level	Primary	FullPath	MetaPath	Total
DBLP	107.7	198.0	184.0	375.6	0.5	0.1	865.9
Shakespeare	7.9	84.5	10.0	2.8	0.5	0.1	105.1
XMARK	1.2	1.1	1.4	1.0	0.3	0.1	5.1

Table 5.2: Construction costs (seconds) of the Extended Schema Repository

#	XPath	Dataset	Matches
Q1	/descendant::phdthesis	DBLP	72
Q2	/descendant::author[.='John Sieg Jr.'] /ancestor::phdthesis	DBLP	1
Q3	/descendant::article[author='Mark Roantree'] /preceding::incollection[author='Amit P. Sheth']	DBLP	5
Q4	/descendant::*[author='Amit P. Sheth'][year = '1990'] /following::inproceedings[author='Mark Roantree']	DBLP	9
Q5	/descendant::mastersthesis/child::author	DBLP	5
Q6	//inproceedings[author = 'Jim Gray'][year = '1980'] /@key	DBLP	6
Q7	//www[editor]/url	DBLP	5
Q8	//book/author[text() = 'C.J. Date']	DBLP	13
Q9	//inproceedings[title/text() = 'Semantic Analysis Patterns']/author	DBLP	2
Q10	//ACT//SPEECH	Shakes.	30,951
Q11	//open_auction//description	XMARK	120
Q12	//open_auction//description//listitem	XMARK	126
Q13	//open_auction//description//listitem//keyword	XMARK	62
Q14	//author[. = 'Peter Van Roy']/parent::mastersthesis	DBLP	1
Q15	/dblp/article/title	DBLP	111,609
Q16	//mastersthesis/www	DBLP	0

Table 5.3: XPath Queries

individual index type. The DBLP database contains a total of 29 Primary indexes (containing 3,468,300 nodes) as a result it has a much higher Primary index construction cost then that of the XMARK (18 Primary indexes with 8,208 node entries) and Shakespeare (3 Primary indexes containing 168,801 element entries) databases. Even though, the combined size of the Primary indexes for DBLP is less than its Base index its construction time is greater as the semantic indexing processor does not utilise bulk loading when creating its 29 indexes, instead it extracts the required nodes from the Base index.

In the following sections, we will provide detailed results arising from our query processing experiments. Table 5.3 describes our XPath query set. Queries Q1 to Q5 are taken from Chapters 3 and 4 of this thesis, while queries Q6 to Q9 are taken

from [3]. Queries Q10 to Q13 come from the XPath Accelerator query set [17] with the remaining queries chosen to demonstrate the fast features of our optimised query processing strategy. The following sections, conduct an experimental evaluation of our optimised query processor against (*i*) our basic query processing strategy introduced in Chapter 3, (*ii*) the eXist database [24] and (*iii*) the ToXop query optimiser using path summaries [3]. While, our response times for queries Q10 to Q13 outperform published times for the XPath Accelerator [17], we were unable to conduct a valid comparative study of these results as the ESR runs on superior hardware.

5.2 Basic Optimisation vs Full Optimisation

In Chapter 3, we introduced our processing strategy containing five processes and optimised it in Chapter 4 by exploiting the metadata features of the ESR. Table 5.4 presents the average execution times (warm cache) for queries Q1 to Q5 from the query set for our query processing strategies from Chapters 3 (`Chpt3`) and 4 (`Chpt4`). While our optimised query engine (`Chpt4`) utilises features from the entire ESR, the basic query engine (`Chpt3`) can only query the Base (excluding its `parent` and `FullPath` attributes) and Level indexes. The purpose of this comparison is to demonstrate how additional metadata constructs and primary indexes allow us to achieve high performance gains over the baseline strategy. The `Comparison` divides the times for the basic query engine by those outputted from our optimised query engine, with a value greater than 1 indicating the level of improvement achieved by the optimised approach. The optimised query processor achieved improvements over its predecessor ranging from 135.18 times faster to 983.19 times faster. As highlighted in the previous chapter, the optimised approach has access to XML metadata that supports the efficient evaluation of XPath queries.

Tables 5.5 and 5.6 partition our query engine times for queries Q1 to Q4 into their respective Process types, with the exception of Processes 3 to 5 as these are performed in a single traversal (see §3.4). The main construct of our processing strategies is the **Arbitrary Prune by Structure** (Process 2) as it identifies one

#	Chpt3	Chpt4	Comparison
Q1	9,429	31	304.16
Q2	27,662	63	439.08
Q3	15,731	16	983.19
Q4	8,381	62	135.18
Q5	16,841	31	543.26

Table 5.4: Query times (ms) for processing strategies from Chapters 3 and 4

#	P. 1	P. 2	P. 3-5
Q1	0	9,414	15
Q2	7,912	16,951	2,799
Q3	0	15,653	78
Q4	0	8,350	31

Table 5.5: Chpt3 Process (P.) times

#	P. 1	P. 2	P. 3-5
Q1	0	16	15
Q2	15	31	17
Q3	0	14	2
Q4	0	31	31

Table 5.6: Chpt4 Process (P.) times

or more subtrees from the XML data tree that contain the location step results, which are then filtered by the remaining processes. However, using the basic query engine, this process is very expensive (e.g. 9,414 ms for Q1) as it searches much of the Base index (see §4.1.1) to identify the start and end points of its respective subtrees. However, the optimised query processor exploits metadata features of the FullPath index (see §4.1.2) to efficiently execute this process (e.g. 16 ms for Q1). This experiment provides a clear indication of where potential improvements lie.

Process 1 is redundant for location steps with a `descendant` axis when the context node is the document root node (see §3.4.1). Additionally, for the `following` and `preceding` axes the **Context Prune by Structure** do not involve any index lookups. Hence, this process is executed in 0 milliseconds for queries Q1, Q3 and Q4. However, for an `ancestor` axis (Q2), Process 1 requires each query engine to execute index lookups against its respective database. Similar to Process 2, the concise metadata features of the ESR allow this process to be efficiently executed by the ESR query engine (15 ms), unlike the basic strategy from Chapter 3 (8,147 ms).

For the `descendant` and `following` axes, Processes 3-5 of the ESR query processor do not contain any additional optimising features e.g. both query engines return identical results for Processes 3-5 for queries Q1 and Q4. However, for `ancestor`

and `preceding` axes (Q2 and Q3), the optimised query engine uses the `parent` attribute of the Base index to make much of the **Arbitrary AxisTest** (Process 3) redundant, thus boosting query performance.

Our basic query processing strategy *must* evaluate each location step separately. However, the ESR query engine *can* evaluate two or more location steps simultaneously depending on the query structure (see §4.3). Furthermore, this processing technique can utilise a suitable Primary index to further boost query performance. During query parsing, the optimised query engine ensures the first two location steps of Q5 are processed simultaneously. As a result, we can efficiently process this query in 31ms. However, in this case the query results were retrieved from the Base index as no suitable Primary index existed in the ESR.

By reducing the `IndexFactor` parameter for Primary index creation from 4 to 0.0002 a Primary index is created on the FullPath `'/dblp/mastersthesis/author'` as its cardinality is equal to the new threshold value (i.e. 5). Using this Primary index, query Q5 is processed in just 27 milliseconds i.e. a performance improvement of 623.74 over the basic query engine is achieved. As highlighted by this experiment, we have not determined the optimun `IndexFactor` value for Primary index creation. This issue is discussed in more detail in §6.2 as an area of future research.

5.3 ESR vs eXist Database

As highlighted in Chapter 2, the eXist database [24] is one of the leading native XML databases on the open source market. In this section, we use it as a benchmark to evaluate the performance of our optimised query processor (ESR). The eXist database was deployed on the same machine as the Oracle 10g server (see §5.1.1) and populated with the DBLP, Shakespeare and XMARK datasets. To further ensure compatible of our experiments, all queries to the eXist database were executed on an identical machine to that of our optimised query engine (see §5.2). However, in order to maximise eXist's performance, we altered its default JVM settings from -Xmx128MB to -Xmx768MB.

In Table 5.7 we present the average execution times (warm cache numbers) for

#	Main Axis Types	eXist	ESR	Comp.
Q9	descendant-or-self, child	16,008	58	276.00
Q2	descendant, ancestor	1,555	63	24.68
Q14	descendant-or-self, parent	1,503	85	17.68
Q8	descendant-or-self, child	401	31	12.94
Q16	descendant-or-self, child	109	10	10.90
Q5	descendant, child	304	31	9.81
Q6	descendant-or-self, attribute	1,612	204	7.90
Q1	descendant	131	31	4.23
Q11	descendant-or-self	147	50	2.94
Q12	descendant-or-self	148	60	2.47
Q15	child	6,403	3,331	1.92
Q10	descendant-or-self	1,824	1,060	1.72
Q13	descendant-or-self	151	100	1.51
Q7	descendant-or-self, child	204	140	1.46
Q3	descendant, preceding	n/a	16	n/a
Q4	descendant, following	n/a	62	n/a

Table 5.7: Query times (ms) for eXist and our optimised query engine (ESR)

each of the queries from Table 5.3 for the eXist processor and our own optimised query engine. These results are displayed in decreasing order of improvement along with the main axis types found in the query structure (ignoring predicate axes). The ESR approach achieved improvements over eXist ranging from 1.46 times faster to 276 times faster (ignoring queries Q3 and Q4 as eXist cannot process the `preceding` and `following` axes). Positive query results primarily result from the schema repository structure and query processing techniques that efficiently prune the index search space. After query parsing the FullPath index can (1) quickly determine the segments of the Base index where the query results are located or (2) identify a suitable Primary index structure that boosts query performance.

- Query processing begins for a specific location step by limiting the search space of the target database while the eXist database fails to limit the search space in this way. For example, query Q9 selects all `author` elements that are children of `inproceedings` elements that have a child named `title` with the given value. During query evaluation the eXist query processor selects all `inproceedings`, `title` and `author` nodes from its indexes and then joins these lists using their path join algorithm. The FullPath index allows us to only select `title` and `author` nodes that are children of a `inproceedings`

node, thus dramatically reducing the search space.

- The FullPath structure allows us to quickly identify paths that are not present in the target database (see query Q16).

- Each entry in the Base index contains a pointer to its parent node allowing a procedure to select the parent (Q14) or ancestors of a given node (Q2). This can be efficiently evaluated as XML documents tend to have a low number of levels [25]. Furthermore, the encoding scheme employed by the Base index allows us to quickly identify the attributes for a given node (Q6).

- Our query parser determines that the location steps of queries Q5, Q8, Q10, Q11, Q12, Q13 and Q15 can be processed collectively using our optimised query engine (see §4.3). Furthermore, queries Q10 and Q15 make use of Primary indexes to further boost performance.

- For the remaining queries (Q1, Q7 and Q9), queries Q7 and Q9 are clustered together as they have a similar query structure i.e. they both have two location steps with identical axis types and a predicate is found on their first location step. For the first location step of queries Q1, Q7 and Q9, the schema repository allows us to quickly identify all descendants of a given context node using the FullPath structure and the `getSizeOfSubtree` procedure. For the second location step of queries Q7 and Q9, we filter the subtree produced by the **Arbitrary Prune by Axis** to remove all descendants that are not children of the context node, allowing us to achieve performance gains from 1.46 to 276. Of these queries, Q9 scores best due to our aggressive pruning strategy, which proves efficient for location steps when the potential search space is high (i.e. the `inproceedings` and `author` nodes in the DBLP dataset have a total cardinality of 928,761) or for highly selective predicates.

- Since the Base and Primary index structures contain the value of each node in the XML data tree, the **Predicate Filter** can efficiently evaluate any value-based predicates (see queries Q9, Q2, Q14, Q8, Q6, Q3 and Q4).

5.4 ESR v's ToXop with Path Summaries

In [3], a highly optimised query processing strategy for the ToX native XML database was presented. They used *holistic path summary pruning* and *access-order selection* techniques to boost the performance of the ToXop query processor (see §2.4). Access-order selection exploits XML metadata to select the most appropriate query plan i.e. top-down or bottom-up evaluation. While holistic path summary pruning, prunes the index search space it is not tied to any particular path evaluation technique (e.g. TwigStackScan, ToXinScan), unlike traditional approaches to path summary pruning.

Table 5.8 presents the execution times of the ESR and ToXop query processors using the ToXinScan path evaluation technique for queries Q6 to Q9. The ToXinScan results were taken directly from [3]. Furthermore, their experimental setup is similar to that of the ESR except for their clock speed, as all their experiments were ran on a Windows XP professional machine with a 1.6 GHz Pentium M processor and 1GB of RAM. As demonstrated in Table 5.8, the highly optimised query processing strategy of the ToXop query processor outperforms the ESR query engine as it supports multiple query evaluation techniques. With the exception of query Q8, as its query structure allows us to process both its location steps collectively, unlike the remaining queries from this table. However, unlike the ESR query engine, the underlying index structure of ToXop (i.e. ToXin [34]) cannot support query evaluation along the thirteen XPath axes as its primary purpose is to optimise the performance of XML twig queries. In §6.2, we identify areas of future research such as multiple query evaluation techniques and the use of stack-based algorithms to limit the number of intermediate results that do not contribute to the final result set. By exploring these areas in more detail we hope to match or even outperform the ToXop project.

#	ToXinScan (1.6GHz)	ESR (3.0GHz)	Comp.
Q6	130	204	0.64
Q7	39	140	0.28
Q8	90	31	2.90
Q9	46	58	0.79

Table 5.8: ESR vs ToXinScan

5.5 Conclusions

While, Chapters 1 and 2 discuss some of the major issues still outstanding in the provision of an adequate XML query service, Chapters 3 and 4 provide a theoretical approach to optimising the performance of XPath queries along the thirteen XPath axes. In this Chapter, we present experimental results that demonstrate the performance gains of our optimised query processing strategy.

Section 5.1 provides a processing framework for the construction of our ESR and demonstrates its fast construction features for a series of popular XML datasets. Even a large schema repository such as the DBLP database (3,736,407 element and attribute nodes) containing many index types can be constructed in less then 15 minutes. Thus, allowing us to quickly rebuild the ESR if updates are made to the target XML dataset.

Then in Sections 5.2 to 5.4, we execute our XML query set against the FAST schema repository using our optimised ESR query engine. With these results demonstrating fast support for axis evaluations, as our query processor adopts an aggressive pruning strategy that uses XML metadata to prune the index search space according to preorder and axis logic. Our experimental results are then analysed against a number of competing concepts that show how we can outperform our basic query engine from Chapter 3 and the eXist database. While, the enhanced ToXop query processor [3] exhibits superior query response times, it does not achieve the same level of axis support that is presented in this thesis. We found our ability to process two or more location steps collectively depending on the query structure to be quite motivating, as it allowed us to outperform an enhanced ToXop processor for a single query instance. Furthermore, we justified the construction overhead of our Primary indexes, as they facilitated even greater performance improvements over eXist. In

the next chapter, a summary of the work presented in this thesis is provided and some areas of future research are outlined.

Chapter 6

Conclusions

The aim of this research was to build a framework for XPath query evaluation that facilitates query opimisation of XPath evaluation by aggressively pruning the search space of the target database. Unlike other research projects that concentrated on the evaluation of regular XPath expressions, this work focused on providing a query processing strategy that supports the evaluation of location steps along the thirteen XPath axes. In this chapter a review of the research is presented in §6.1 before areas for future research are offered in §6.2.

6.1 Thesis Summary

In Chapter 1, an introduction to the XML data model was presented with the structure of XML documents described. In recent years, there has been a rapid growth in the XML data format as its semi-structured and self-describing nature allow it to model an unlimited number of tree dialects. Two forms of XML databases exist: XML enabled databases and native XML databases. Native XML databases are more suited for XML query processing, as XML enabled approaches provide poor support for the structural constraints found in XPath and XQuery expressions as structural information regarding the XML data tree is lost during document loading. However, many native XML databases fail to provide full support for the XPath and XQuery languages as their query engines fail to implement the thirteen XPath axes. Furthermore, native XML databases perform badly for queries where the database

size is large or the structure complex. Thus, the motivation of this thesis arose from the absence of a viable query service as even the industry leader eXist still has problems with performance for certain axis types. Consequently, this research focused on methods for pruning the XML data tree according to axis properties in order to limit the potential search space.

Chapter 2 investigated several research projects covering existing *state-of-the-art* index-based query processing strategies for XML databases. When examining these projects, the emphasis put on their ability to support the full set of XPath axes and to efficiently prune the underlying XML database. The DataGuides project focused on exploiting schema tree information in order to limit the potential search space. However, this approach is only suitable for the evaluation of regular path expressions and it predates XPath. The eXist database incorporates a large index repository that allowed its query processing engine to evaluate queries against large collections of XML documents. Furthermore, it provides support for eleven of the thirteen XPath axes. However, its query processing strategy is not optimised as it does not incorporate any tree pruning techniques as a result large irrelevant sections of the XML database are processed during XML query evaluation. Extending the ToXop query processor with path summaries has resulted in an advanced XML query engine that allows for the rapid evaluation of XML twig queries as it exploits schema tree properties to prune the search space and determine the optimum query plan. However, similar to other approaches its index structure does not allow for the evaluation of the full set of XPath axes. The XPath Accelerator stands out in our review of related research as it provides full support for the XPath axes. Furthermore, it prunes the target database according to preorder, postorder and axis properties. Nevertheless, the XPath Accelerator fails to incorporate XML metadata constructs into its index repository and as result its query processor is not truly optimised as it cannot use schema tree properties to further prune the database. Consequently, it was decided that an approach that uses XML metadata, axis and preorder encoding properties to prune the index search, yet supporting the thirteen XPath axes was required to realise our goal of optimising XPath query evaluation.

In Chapter 3, we devised a strategy [36] for evaluating location steps along all

of the XPath axes (with the four *major* axes described in detail and the full set described in [28]), providing an element of novelty among related research that concentrates on the evaluation of regular path expressions. Our query processing strategy contains five processes that prune and filter the search space using query structure, axis properties and preorder encoding logic. Since a location path may contain multiple location steps, processes are repeated for each individual location step with the output of Process 5 (set of arbitrary nodes) becoming the input to Process 1 (set of context nodes) in the next iteration.

- Process 1 eliminates redundant context nodes, which improves the efficiency of our query processor as Processes 2-5 must be executed for each context node before the next location step can be processed.

- For each context node from Process 1, Process 2 uses axis logic and a series of index-lookups to identify a subtree (set of arbitrary nodes) from the XML data tree that contains the query results.

- Processes 3-5 are performed in a single traversal and filter the subtree outputted by Process 2 to produce the location step results. These processes filter all arbitrary nodes that are not allowed by the axis definition together with any nodes that do not support the predicates and nodetest of the current location step.

While our processing framework from Chapter 3 aggressively prunes the index search space and supports axis traversals along all the XPath axes, it is not optimised as many of its processes require expensive index lookups. Chapter 4 presented an advanced index structure for XML databases known as the Extended Schema Repository (ESR) [30], containing rich metadata constructs that allowed us to fine tune our location step evaluation strategy from Chapter 3 as detailed in our optimised algorithms for query evaluation along the *major* XPath axes. While, these algorithms are highly optimised, the evaluation of an arbitrarily long location path can be expensive especially against a large database as each location step must be individually evaluated. However, in Chapter 4, we derived a procedure that allows

two or more location steps to be collectively processed, thus further boosting query performance.

The deployment and justification of our hypothesis for optimised XPath query evaluation was presented in Chapter 5. Firstly, we demonstrated fast build times for the ESR using a series of XML datasets [29]. Secondly, we presented a performance comparison of our ESR query engine against (*i*) our basic query processing strategy from Chapter 3, (*ii*) the eXist database and (*iii*) the ToXop query processor using path summaries. The experiments demonstrated that:

- The metadata features of the ESR allowed us to boost the performance of our processing strategy from Chapter 3. XML schema metadata is the key to query optimisation as it allowed for an efficient means of pruning the underlying XML data tree in order to reduce the potential search space, thus boosting query optimisation.

- For a large query set, the ESR query engine outperforms the eXist database and allowed for a greater level of axis support, even though eXist is generally regarded as one of the industry leaders in terms of speed. Furthermore, these experiments justified the construction of our Primary indexes as demonstrated with improved query response times.

- Extending ToXop with XML metadata resulted in a highly optimised query engine that generally outperforms the ESR query processor. However, our ability to evaluate a series of location steps simultaneously allowed us to outperform ToXop for a subset of queries. Furthermore, unlike our approach they concentrate on the swift evaluation of twig queries and not the thirteen XPath axes.

6.2 Future Research

The bulk loading features of the ESR facilitate a very fast population of our index. However, the storage overhead is still quite high, as each XML instance in the target document has one or more entries in its index repository. Furthermore, much

of the character data (i.e. `FullPath`, `Name`, `DocID` and `Value`) recorded during data source parsing is duplicated throughout the entire Base index e.g. FullPath `'/dblp/article/title'` has 111,609 instances in the Base index. By extending the ESR with additional index structures that record a unique identifier for each distinct string value in the target document the total storage costs of the ESR will be greatly reduced. However, this proposal needs to be explored in more detail as it will increase the number of joins between indexes during query evaluation, which will adversely affect the response times.

As shown in the previous chapter, Primary indexes boost query performance for certain query types as they perform quicker than the Base index. However, we choose not to create Primary indexes on all FullPath instances in order to reduce the construction and storage overheads. The majority of Primary indexes are created on FullPath instances that exceed a threshold value in the Base index. This threshold value is calculated by retrieving the average number of preorder nodes per FullPath and multiplying it by a variable `IndexFactor`. In [20], they took a similar approach by only creating indexes on paths with a depth less than or equal to a parameter `k`, as they observed that regularly occurring paths in a semi-structured database are found near they root of tree. However, our approach is more advanced as we exploit statistical metadata to identify commonly occurring paths.

The parameter `IndexFactor`, varies the level of detail of the Primary indexes. For a small `IndexFactor` value the number of the Primary indexes is substantially reduced. However, this increases our query processing costs as the Primary indexes can only support a reduced subset of XPath queries. Furthermore, the optimum `IndexFactor` value varies from document to document depending on its structure and the query requirements of the end user. Calculating the optimum `IndexFactor` for an XML dataset using cost-based metrics and structural XML metadata offer interesting challenges and avenues of research to further boost our ESR query engine.

Our query processing framework uses a top-down evaluation strategy, which offers a fast means of resolving queries with selective predicates near the root of the

query. However, a bottom-up strategy is superior at processing queries with selective predicates on the query leaf node. Therefore, future versions of the ESR must support top-down and bottom-up evaluation strategies. The challenge to the query processor is to quickly determine the most efficient query plan. In [3], the authors use simple heuristics based on XML metadata to determine the most appropriate evaluation to employ. These heuristics may be investigated in more detail to determine their suitability for FAST project.

The focus of our research has been a read-only index to support the evaluation of XPath queries against XML data trees. As a result the work presented in this thesis does not support updates to the underlying index structures, which would have a serious impact on our numbering scheme for the Base and Level indexes. The challenge of determining cost-effective strategies for updating the index structures of the ESR would be an interesting problem to address with additional research.

Since the numbering scheme [31] of our Base and Level indexes allows us to start path traversals from arbitrary nodes in a document tree, it enables our query processing strategy to be extended to support path traversals embedded in XQuery expressions. In [18], the authors describe a compiler that translates XQuery expressions into a relational algebra which may be efficiently implemented on top of any RDBMS. However, the approach of Grust et al. [18] did not exploit XML metadata, which formed the key enabler of our query processing strategy. Extending their XQuery compiler [18] with XML metadata constructs forms an interesting avenue of future research.

The intermediate result sets produced by our query processing strategy often contain results that do not contribute to the final result set. Access methods such as TwigStack [10], TwigStackScan and ToXinScan [3] use a chain of linked stacks to compactly represent partial results for query paths. For example, TwigStack guarantees that for evaluating queries with only ancestor-descendant edges each intermediate path solution contributes to the final answer. Therefore, future research could specify a suitable stack-based access method that is not just I/O and CPU optimal for ancestor-descendant queries (TwigStack), but supports optimal evaluations for the thirteen XPath axes.

Bibliography

[1] Patrizia Asirelli, Massimo Martinelli, and Ovidio Salvetti. An Infrastructure for MultiMedia Metadata Management. In *Proceedings of the 1st International Workshop on Semantic Web Annotations for Multimedia (SWAMM) in cooperation with the 15th World Wide Web (WWW) Conference*. CEUR, 2006.

[2] Denilson Barbosa, Attila Barta, Alberto O. Mendelzon, George A. Mihaila, Flavio Rizzolo, and Patricia Rodríguez-Gianolli. ToX - The Toronto XML Engine. In *Proceedings of the International Workshop on Information Integration on the Web (WIIW)*, pages 66–73, 2001.

[3] Attila Barta, Mariano P. Consens, and Alberto O. Mendelzon. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB)*, pages 133–144. ACM, 2005.

[4] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Ozcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One Part Relational, One Part XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 347–358. ACM Press, 2005.

[5] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, 2006.
`http://www.w3.org/TR/2006/PR-xquery-20061121/`.

[6] Jon Bosak. The Plays of Shakespeare in XML. Online Resource, 1999. `http://www.oasis-open.org/cover/bosakShakespeare200.html`.

[7] Jihad Boulos and Shant Karakashian. A New Design for a Native XML Storage and Indexing Manager. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)*, volume 3896 of *Lecture Notes in Computer Science*, pages 755–772. Springer, 2006.

[8] Ronald Bourret. XML Database Products. Online Resource, 2006. `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`.

[9] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). World Wide Web Consortium, 2006. `http://www.w3.org/TR/2006/REC-xml-20060816`.

[10] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321. ACM Press, 2002.

[11] James Clark and Steven J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[12] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive xquery to sql translation using dynamic interval encoding. In *SIGMOD Conference*, pages 623–634. ACM, 2003.

[13] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[14] Apache Software Foundation. Xerces2 Java Parser. Online Resource, 2007. `http://xerces.apache.org/xerces2-j/`.

[15] SMB GmbH, dbXML Group, and OpenHealth Care Group. XML:DB Initiative for XML Databases. Online Resource, 2006. `http://xmldb-org.sourceforge.net`.

[16] Roy Goldman and Jennifer Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, pages 436–445. Morgan Kaufmann, 1997.

[17] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 109–120. ACM Press, 2002.

[18] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, pages 252–263. Morgan Kaufmann, 2004.

[19] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML Streams. *The VLDB Journal*, 14(2):197–210, 2005.

[20] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 129–140. IEEE Computer Society, 2002.

[21] Petr Kolar and Pavel Loupal. Comparison of Native XML Databases and Experimenting with INEX. In *Proceedings of the Dateso 2004 Annual International Workshop on DAtabases, TExts, Specifications and Objects*, pages 116–119. CEUR, 2006.

[22] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 361–370. Morgan Kaufmann, 2001.

[23] David Megginson. SAX - Simple API for XML. Online Resource, 2007. `http://www.megginson.com/SAX/index.html`.

[24] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Proceedings of Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops*, volume 2593 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2002.

[25] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML Web: a First Study. In *Proceedings of the 12th international conference on World Wide Web (WWW)*, pages 500–510. ACM Press, 2003.

[26] Mirella Moura Moro, Zografoula Vagena, and Vassilis J. Tsotras. Tree-Pattern Queries on a Lightweight XML Processor. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 205–216. ACM, 2005.

[27] Ullas Nambiar, Zoé Lacroix, Stéphane Bressan, Mong-Li Lee, and Ying Guang Li. Efficient XML Data Management: An Analysis. In *Proceedings of the 3rd International Conference on E-Commerce and Web Technologies (EC-Web)*, volume 2455 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 2002.

[28] Colm Noonan. Optimising the Evaluation of the Thirteen XPath Axes. Technical Report ISG-07-03, Dublin City University, 2007. `http://www.computing.dcu.ie/~isg/technicalReport.html`.

[29] Colm Noonan, Cian Durrigan, and Mark Roantree. Using an Oracle Repository to Accelerate XPath Queries. In *Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA)*, volume 4080 of *Lecture Notes in Computer Science*, pages 73–82. Springer, 2006.

[30] Colm Noonan and Mark Roantree. Optimising XML-based Web Information Systems. In *Proceedings of the International Workshop on Web Information Systems Modeling (WISM) in cooperation with the 19th International Conference on Advanced Information Systems Engineering (CAISE)*, volume 2, pages 803–814. Tapir Academic Press, 2007.

[31] Martin F. O'Connor, Zohra Bellahsene, and Mark Roantree. An Extended Preorder Index for Optimising XPath Expressions. In *Proceedings of the 3rd International XML Database Symposium (XSym)*, volume 3671 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2005.

[32] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 134–144. ACM, 2003.

[33] Praveen Rao and Bongki Moon. PRIX: Indexing And Querying XML Using Prüfer Sequences. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 288–300. IEEE Computer Society, 2004.

[34] Flavio Rizzolo and Alberto O. Mendelzon. Indexing XML Data with ToXin. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB)*, pages 49–54, 2001.

[35] Mark Roantree. The FAST Prototype: a Flexible indexing Algorithm using Semantic Tags. Technical Report ISG-06-02, Dublin City University, 2006. `http://www.computing.dcu.ie/~isg/technicalReport.html`.

[36] Mark Roantree, Colm Noonan, and John Murphy. Specifying and Optimising XML Views. In *British National Conference on Databases (BNCOD)*, Lecture Notes in Computer Science. Springer, 2007.

[37] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael Carey, Ioana Manolescu, and Ralph Busse. XMARK: A Benchmark for XML Data Management. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, page 974985. Morgan Kaufmann, 2002.

[38] Susana Schwartz. XML's Reality Check: Database Management. Online Resource, 2002.
`http://www.taborcommunications.com/dsstar/02/0212/103910.html`.

[39] Robert Sedgewick. *Algorithms in Java, Parts 1-4 (Fundamental Algorithms, Data Structures, Sorting, Searching)*. Addison-Wesley, third edition, 2002.

[40] Dan Suciu and Gerome Miklau. University of Washington's XML Repository. Online Resource, 2002.
`http://www.cs.washington.edu/research/xmldatasets/`.

[41] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, pages 204–215. ACM, 2002.

[42] Avinash Vyas, Mary F. Fernández, and Jérôme Siméon. The Simplest XML Storage Manager Ever. In *Proceedings of the 1st International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P) in cooperation with ACM SIGMOD*, pages 37–42, 2004.

[43] Felix Weigel, Holger Meuss, Klaus U. Schulz, and François Bry. Content and Structure in Indexing and Ranking XML. In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB)*, pages 67–72, 2004.

[44] Tom Yager, PJ. Connolly, Mario Apicella, Andrew Binstock, Sean McCown, Mike Heck, and Neil McAllister. InfoWorld 2006 Technology of the Year Awards. Online Resource, 2006.
`http://akamai.infoworld.com/pdf/special_report/2006/01SRtoy.pdf`.

[45] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.

[46] Ning Zhang, M. Tamer Özsu, Ihab F. Ilyas, and Ashraf Aboulnaga. FIX: Feature-based Indexing Technique for XML Documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 259–270. ACM, 2006.