OPTIMISATION TECHNIQUES FOR STORING AND QUERYING XML DATA IN RELATIONAL DATABASE SYSTEMS

A thesis submitted in fulfilment of the requirements of
the degree of Doctor of Philosophy

Mo'ad Maghaydah

December 2010

Master of Engineering Science (University of New South Wales) 2001

Bachelor of Engineering (Jordan University of Science and Technology) 1995

MACQUARIE UNIVERSITY
SYDNEY ~ AUSTRALIA

Department of Computing

Faculty of Science

Macquarie University, NSW 2109, Australia

Acknowledgements

I am grateful to my supervisor, Professor Mehmet A. Orgun, for his supervision, encouragement, unbounded enthusiasm, and generous support from the preliminary to the concluding level. He led me to the correct direction at every stage of the research. Without his care, supervision and friendship, this thesis would not have been possible.

I am also grateful to my co-supervisor, Dr. Abhaya Nayak, for his suggestions and support.

I am deeply indebted to my wife Hadeel and my two beautiful children Tamir and Talah who brought a great deal of love and inspiration to my life, and have always been by my side to encourage and motivate me.

I would like to express my appreciation to the managers and my colleagues in the IT Department at Star Track Express Pty Limited for their understanding and support during my time there as an IT Systems Specialist (2004–2010), while I was also carrying out this research.

I would like to acknowledge that this thesis was edited by Dr Lisa Lines director and head editor of Elite Editing and Tutoring, the editorial intervention was restricted to Standards D and E of the Australian Standards for Editing Practice.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the thesis.

Declaration

I hereby certify that the work embodied in this thesis is the result of original research. This work has not been submitted for a higher degree to any other university or institution.

Signed: ______________________________

Date : ______________________________

Abstract

Extensible Markup Language (XML) has recently emerged as a standard for electronic information interchange, due to its flexibility and portability. For instance, in service-oriented applications, XML messages are commonly used for inter-company interactions. However, those XML messages may be used for purposes other than transactions and data interchange (for example, purchase orders and invoice statements) and they must often be retained for later use and analysis. This requires scalable technology to effectively store and query XML data.

Due to their widespread availability and robustness, relational database management systems (RDBMS) still offer the most affordable technology to develop XML database systems. However, the XML data model presents new challenges such as maintaining the document order and supporting complex structural-join queries, which require tree-aware processing mechanisms. While the state-of-the-art approaches to support XML data in relational systems require new algorithms and indexing techniques that make them powerful, it has been observed that some of those changes may not be directly applicable to relational database systems and/or they may present a trade-off between performance and storage usage. Further, the modification of the relational system's kernel is hardly an option for many RDBMS vendors. There are still considerable benefits in developing solutions that do not involve changes to the RDBMS's kernel, thereby reducing the cost of re-engineering relational database systems.

In order to improve the process of storing and querying XML data in relational systems, in this thesis, we propose a new compact Dewey-based labelling scheme to support it. The new label structure, composed of two components (parent, child) in the Dewey format, would significantly improve the performance of those XML queries that are based on parent-child and sibling relationships. Moreover, we propose advanced query optimisation techniques based on certain features that exist in Dewey labels and based on a better utilisation of the document schema summary of XML documents. Our techniques are portable and can be applied to any Dewey-based labeling technique proposed for storing and querying XML data. Through extensive experimental studies, we show that these techniques make off-the-shelf relational systems more tree-aware, and significantly improve their capabilities to support XML data.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

Information may be the most valuable commodity in the modern world. It can take many different forms—such as accounting and payroll information, information about customers and orders, scientific and statistical data, graphics, or multimedia. We are virtually swamped with data, and we cannot—or at least we'd like to think about it this way—afford to lose it. These days we simply have too much data to keep storing it in file cabinets or cardboard boxes. The need to store large collections of persistent data safely, 'slice and dice' it efficiently from different angles by multiple users, and update it easily when necessary is critical for every enterprise. That need mandates the existence of databases, which accomplish all the tasks listed, and then some [Kriegel & Trukhnov 2008].

1.1 Introduction

Extensible Markup Language (XML), which appeared in the mid-90s, has emerged as a dominant language for Internet applications and a technology for information representation and exchange over the Internet [Nambiar *et al.*, 2002; Silberschatz *et al.*, 2002].

XML is an open standard, which has been developed and managed by the World Wide Web Consortium (W3C) [Bray *et al.*, 2008]. It provides a unified model for representing data, content, and metadata in a self-describing simple text, which is known as an XML document. An XML document is easily readable by both humans and machines. It is vendor and platform independent, highly flexible and extensible. The XML data model is suited for any combination of structured, unstructured and semi-structured data. These features have led to its adoption by a wide range of industries [Beyer *et al.*, 2006; Lee and Team, 2009]. Beyer *et al*. [2005] state:

Virtually every industry is working to standardize XML representations for their common business objects. As one industry analyst puts it, Hundreds of vertical schemas, in fields as diverse as government, biology, finance, and travel, are publicly available and being actively used. Undoubtedly, there are thousands more in private hands.

Key applications for XML include web services and service-oriented applications; XML messages are commonly used for inter-company interactions. However, these XML messages may be more than transactions and data interchanges (such as purchase orders and invoice statements) and they must be retained for later use and analysis.

The growing trend of using XML in many applications, ranging from a simple format for data exchange to archiving data, requires a scalable technology to effectively store and query XML data. This growth has led to the development of a wide range of XML management systems [Cathey *et al.*, 2007]. However, developing efficient techniques for storing massive XML documents and retrieving information from them is one of the core problems that require investigation in the database and XML area.

1.2 XML Database Systems

An XML database is a data persistence software system that allows data to be stored in XML format at the logical level as a minimum. This data can then be queried, exported and serialised into the desired format. An XML management system should provide natural and efficient ways to:

- Describe the structure of XML documents
- Store XML documents
- Extract information from XML documents

The last point is the most important. However, the way in which XML documents are viewed determines how they can be queried.

The efforts to store and manage XML data have explored most available technologies ranging from simple file systems to advanced Object-oriented database systems [Carey *et al.*, 2000; Klettke and Meyer, 2000; Mertz; Surjanto *et al.*, 2000]. Other systems have been developed from the ground up based on semi-structured and tree-based data models to handle XML data [AG, 2009; Fiebig *et al.*, 2002; Jagadish *et al.*, 2002; McHugh *et al.*, 1997]; these systems are called Native XML Management Systems. However, relational-based XML database systems are still the most available and affordable solution because this approach seeks to utilise the power of the existing, robust and mature relational database technology [Cathey *et al.*, 2007; Härder, 2005; Lewis *et al.*, 2002; Nambiar *et al.*, 2002].

The following subsections briefly highlight the features, motivations, and advantages of the two major XML database approaches: the native approach and relational-based approach.

1.2.1 Relational-based XML Database Systems

Based on a sound theory, the relational database management systems (RDBMS) have evolved over three decades to become arguably the most widely used form of database management systems [Gartner, 2007; Kriegel and Trukhnov, 2008; Melton and Buxton, 2006].

The relational model supports a set of tables in which each table contains a fixed collection of columns (fields). An indefinite number of rows (records) occurs within each table. The order of the rows is not important. The queries issued on relational data need a schema to be known. Each row must have a unique primary key. In addition, tables typically have secondary keys that correspond to primary keys in other tables.

Recently, many studies have been conducted to provide support for the emerging XML data within the mature relational database systems. The most common method is to shred XML documents and store them in relational tables. XML queries are then relationally mapped to structured query language (SQL) to retrieve the desired results. The motivations behind developing relational-based XML database systems can be summarised as follows:

- XML is seen as another data format for relational and object-relational data-processing tools.
- RDB systems are robust and mature, they have been developed over three decades; they support advanced features such as concurrency control, query optimisation, scalability and failure recovery and they should not be discarded [Cathey *et al.*, 2007; Weigel *et al.*, 2005].
- Relational database systems are widespread, their market is still growing and they are reasonably priced [Gartner, 2007; Hoven, 2002].
- RDB systems will survive because they are the best storage solution when the data are well defined and rigid, like in an accounting system. Kriegel and Trukhnov [2008] assert that ‘for better or for worse, relational database systems have come to rule on planet Earth’.
- Most of the data around the globe are stored in (object) relational database systems [Gartner, 2007; Kriegel and Trukhnov, 2008]. This mature platform has been outstandingly supporting different kinds of applications; extending it to support

XML applications will provide one common platform in which different kinds of data models and applications can coexist and communicate.

Starting in 2001, most commercial relational database vendors began adding support for XML data into their products [Beyer *et al.*, 2006; Oracle, 2009; Pal *et al.*, 2005b]. Initially, the focus was on merely storing and retrieving XML documents as a whole, without the ability to perform significant operations on the content of those documents. However, many techniques have been developed over the years; new indexing methods, shredding document's contents techniques, and new XML data types have all been introduced to support efficient storing and querying of XML documents in relational systems. Further, the commercial relational database systems support XML query languages such as XPath and XQuery, as well as the ability to transform ordinary relational data into XML structures of the users' choice.

However, the XML data model (i.e. nested hierarchical ordered data) presents significant challenges to the relational database systems; XML query performance for complex structural-join queries that XQuery language allows is still considered a critical issue. Developing a complete and comprehensive relational-based management system for XML data is still some time off.

1.2.2 Native XML Database Systems

Due to the differences between the relational data model and the XML data model, many researchers and companies started developing dedicated database systems to handle XML data. These systems store and manipulate XML in a more native form. Moreover, they implement the native XML query languages, such as XQuery, and query results are naturally in the XML format [Bourret, 2005b; Fiebig *et al*., 2002; Jagadish *et al.*, 2002].

Unlike relational systems, the native systems lack solid theory; however, there are some valid reasons and motivation behind developing native XML systems:

- The XML model is highly flexible; thus, it may be difficult to decide in advance on a single, correct schema. The structure of data may evolve rapidly, data elements may change types, or data not conforming to the previous structure may be added [McHugh *et al.*, 1997].

- An XML document is ordered, while the order is ignored in the relational model.
- An XML document has a logical structure and hierarchical nested elements. It is very costly to support this structure in flat relations.
- The query result of the relational database system is flat relations, and XML data format needs to be reconstructed again.

Native XML database systems are still relatively new and require a great deal of work to support features that already exist in relational-based systems, such as transactions, scalability and concurrency control. The cost of developing such a system is considered expensive. Further, these systems are dedicated for XML support and do not support any other data format (such as relational data or object data).

With the advances in relational-based systems and their ability to better support XML data in their native format, it is less likely that pure native XML database systems will replace relational database systems any time soon. However, they will present a better option for document-centric XML applications (such as manufacturing parts databases) [Bourret, 2005a; Staken, 2001].

1.3 Motivation

Many recent studies have demonstrated that shredding an XML document, and labelling and indexing its elements to recover the document order is still a competitive approach to develop efficient and affordable XML database systems based on off-the-shelf relational database systems [Florescu and Kossmann, 1999; Grust *et al.*, 2007; Härder, 2005; O'Neil *et al.*, 2004; Yoshikawa *et al.*, 2001]. Moreover, with the advent of the new XML data type in commercial relational databases, the relational database systems have gained significant strength as a base technology to build XML database systems. These new data types allow relational systems to support XML data in their native format, and reduce the cost of reconstructing the original documents.

However, there are significant advantages to improving and developing an approach that is based on shredding and labelling XML document nodes using the existing standard relational technologies:

- Most of the off-the-shelf and widely used open source relational database systems, such as MySQL server [MySQL, 2009], do not yet have a specific advanced XML data type; therefore, using the binary large object (BLOB) data type and shredding XML documents approach are the only available options.
- Some applications are data-centric, and shredding data will significantly improve query performance by avoiding parsing the document during run time. Without doubt, indexing the new XML data types will improve performance for the most frequent queries. However, it is hard to cover a broad range of queries without going back to the document and parsing it during run time.
- Shredding an XML document, or part of it, and building more traditional relations and indexes will better utilise the power of relational databases. Moreover, this approach reduces the cost of data updates in XML applications.
- Labelling techniques, which are heavily used to support the shredding approach, are also used in native XML systems, and in hybrid systems to support the new XML data types. Improving the labelling techniques will also benefit off-the-shelf relational database systems.
- Developing efficient solutions based on this approach will reduce the cost of re-engineering relational systems and enhance application portability by avoiding changes to the database system kernel. Such solutions can be adopted in widely used open source relational database systems at minimal cost.

This research aims to investigate and develop techniques that will improve the capabilities of relational-based XML management systems based on existing technology that is available in off-the-shelf RDBMS without the need to change the system kernel.

1.4 Problem Definitions

When an XML document is shredded and stored in relational tables, the document order and structure are lost. However, the document order and structure are very important features of the XML data model and they are at the centre of most XML queries. Node labelling is a common technique in relational-based systems to capture the document order and support containment relationships (such as ancestor-descendent relationships). Further, node labelling is also used in hybrid XML database systems (i.e. relational systems with a built-in XML data type) to answer document-centric queries with minimal access to the original XML document. However, further work is required to improve the features and structure of the labelling methods to allow more efficient utilisation of the underlying relational technology, such as indexes.

Further, the evolution of XQuery and XPath as query languages for XML data has introduced more challenges to the relational-based systems, since they allow complex queries based on different types of structural-join relationships. The query processors and optimisers in relational systems have been designed and implemented to support flat relational data that is fundamentally different from the XML data model. Thus, the generated query execution plans are far from optimal.

This research investigates and develops techniques to address the following challenges that are related to storing and querying XML documents in off-the-shelf relational database systems:

1. The Dewey-based labelling technique for XML nodes has emerged as the most suitable labelling technique to support dynamic XML documents [Härder *et al.*, 2005; O'Neil *et al.*, 2004; Tatarinov *et al.*, 2002]. It supports variant operations on dynamic XML documents, from inserting large sub-trees, without relabelling the existing nodes, to fine-grained locking thereby avoiding access to external storage as much as possible [Haustein *et al.*, 2005]. In the Dewey-based labelling technique, some special functions may be used to compare node labels to verify parent-child, or ancestor-descendent relationships. However, Dewey labels, which are represented as binary strings, are considered too long for very large and deeply nested XML documents. Further, functions are used to process the labels and validate relationships between nodes, which might not result in efficient use of the relational database indexing mechanisms.
2. Recently, researchers started to focus on the use of the document structure summary to improve the efficiency of XML database systems and relational-based systems in particular [Goldman and Widom, 1997; Moro *et al.*, 2008], because no labelling technique on its own can efficiently support all types of structural-join XML queries. The document structure summary can be helpful to produce optimised queries. Moreover, it can be used to validate some queries, such as validating the existence of certain path expressions, without the need to access the real XML data. More work is required to develop techniques that integrate both the schema

summary and node labels (such as Dewey labels) to improve XML database system capabilities.

3. With the arrival of the XQuery language, XML Query processing in off-the-shelf relational database systems has become more challenging for two major reasons:
    a. Due to the differences between the hierarchical structure of XML data and the flat relational data model, the query optimiser in relational database systems fails, in most cases, to find the right join order for XML queries. Failing to produce an optimal join order has a severe effect on the query execution performance.
    b. XQuery is a powerful query language that allows complex tree-pattern queries. This type of query is very expensive to run in a relational system due to the high number of join operations. While new algorithms and techniques have been proposed to address this type of query, some are not directly applicable to relational database systems and may require changes to the system kernel [Bao *et al.*, 2008; Bruno *et al.*, 2002; Grust *et al.*, 2003; Jagadish *et al.*, 2002; Jiang *et al.*, 2007].

1.5 Research Contributions

This thesis focuses on the development of an XML database system based on the relational technology available in off-the-shelf relational database systems. We summarise the main contributions of this research as follows:

1. A more compact, parameterised and flexible Dewey-based labelling scheme has been developed. This labelling technique can reduce the total label size of an XML document by an average of 20 per cent compared with other recent Dewey-based labelling approaches. Moreover, it can be tuned to provide further label size reduction for particular applications. Reducing the label size can be useful in space-limited environments such as application on hand-held devices. Further, smaller labels means that more labels can be loaded in the main memory, which would reduce the costly I/O disk operations.

    Moreover, we propose an enhanced structure for Dewey-based labels, which would reduce the use of functions and make more efficient use of the current indexing mechanisms that are available in off-the-shelf relational database systems. The enhanced structure provides efficient support for structural-join queries that are based on parent-child and sibling relationships. Chapter 3 provides complete details about our new labelling approach.

2. We have developed alternative techniques, beyond the traditional range methods, to improve the performance of evaluating XPath axis steps in the presence of the Dewey labelling scheme. XPath axis steps are the core of any XML query in the XQuery/XPath query language. Our approach is based on exploiting the document schema summary and features of Dewey labels. Our approach does not rely on the existence of a document schema (such as data type definitions, DTD). We capture the document structure summary during the initial parsing of an XML document. More details are provided in Chapter 4.
3. Chapter 5 discusses new, advanced optimisation techniques to make relational database systems more tree-aware without modifying the system kernel by:
   a. Developing effective rules that can produce an optimal join order for most of the XML queries. The optimal join order produces smaller intermediate results by forcing the use of a structure index (i.e. node labels) at early stages of the query execution plan. Moreover, this technique takes advantage of the document structure summary to make the right selection of the path expressions that should be evaluated first.
   b. Developing techniques to improve the performance of evaluating complex structural-join queries, such as twig queries, by reducing the number of relational join operators that are required to produce correct results. These techniques are based on the features of Dewey-based labels; every Dewey label id contains the label id(s) of its ancestor nodes.

      Combining that with detailed information about each node (path) in the document structure summary table (i.e. node level), ancestor nodes can be evaluated without the need to apply range methods that use an excessive number of join operations. Further, proving that two nodes are descendants of the same ancestor node can be evaluated with a lower number of joins.

1.6 Publications

The findings and major contributions of this thesis have been reported in five international conferences. An extended version of one conference paper has also been invited for post-conference publication in an established journal. Below is a brief summary of each publication starting with the earliest:

1. Maghaydah, M. & Orgun, M. A. (2006). Labelling XML nodes in RDBMS. *The 8th Asia Pacific Web Conference (APWeb) Workshop*. Harbin, China, pp. 122–126.
2. Maghaydah, M. & Orgun, M. A. (2006). XMask: An enabled XML management system. *The 4th International Conference Advances in Information Systems (ADVIS'06)*. Izmir, Turkey, pp. 38–47.

Storing XML documents in relational database systems is still the most affordable and available storage solution. We presented an abstract design for a relational-based XML management system. The paper contains a performance study between some labelling approaches and a superseded labelling scheme that we proposed earlier, and is reported in the first paper with primarily test results. The superseded XMask labelling approach provided a number-based representation for Dewey-like labels. The proposed approach has some limitations, such as supporting deeply nested XML documents. However, that study has led to developing the Prefixing on Demand (PoD) labelling approach. Chapter 3 provides complete details of the PoD labelling approach. Further, the paper highlighted the importance of capturing the document structure summary, which has been used intensively in our subsequent works to develop optimisation techniques for XML queries; the document structure summary is discussed in Chapter 4.

3. Maghaydah, M. & Orgun, M. A. (2007). An adaptive labelling method for dynamic XML documents. *IEEE International Conference on Information Reuse and Integration*. Las Vegas, NV, pp. 618–623.

The Dewey-based labelling technique for XML nodes has emerged as the most suitable labelling approach to support dynamic XML documents. It supports variant operations on dynamic XML documents (such as inserting large sub-trees and deleting nodes) without relabelling the existing nodes. However, Dewey labels, which are represented as binary strings, are considered too long for very large and deeply nested XML documents, or when there is a frequent node insertion. We proposed a new labelling technique based on Dewey Identifiers, called PoD. Our technique reduces the label lengths and the total label size of general XML documents without any prior knowledge about the document structure (i.e. DTD or XSchema). Moreover, PoD, with its parameterised features, can use the XML metadata to further reduce label lengths for specific XML documents in applications in which the label or index size is more important than other considered factors. Further, PoD minimises the processing overhead by not using variable length prefix-free binary strings. Instead, PoD uses predefined fixed-width blocks that can be adjusted throughout a given XML document. The evaluation test showed that PoD reduces the total label sizes by more than 20 per cent compared with other recent Dewey-based labelling approaches. Chapter 3 contains full details on the PoD labelling concept, storage requirements analysis, and extensive evaluation tests using well-known XML benchmarks.

4. Maghaydah, M. & Orgun, M. A. (2010). Efficiently querying XML documents stored in RDBMS in the presence of Dewey-based labelling scheme. *The 2nd Asian Conference on Intelligent Information and Database Systems (ACIIDS)*. Hue City, Vietnam, pp. 43–53.
5. Maghaydah, M. & Orgun, M. A. (2010). Efficiently querying dynamic XML Documents stored in relational database systems. *International Journal of Intelligent Information and Database Systems*. (Accepted).

The reported work proposed techniques for utilising the current indexing mechanisms available in relational database systems. The major contributions of these two papers can be applied to any Dewey-based labelling scheme. First, we proposed a new structure for Dewey labels by splitting any given Dewey label into two components (Parent id, Pid, and Child id, Cid). This new label structure, PoD-Split (PoD-S), would significantly enhance the query performance for XML queries that are based on parent-child and sibling relationships. The second technique provides an efficient mechanism to navigate upwards an XML tree. A new user-defined function has been introduced that uses the label id of the context node to retrieve the label id of any ancestor node by utilising the node’s level value, which is available as part of the document structure summary. The evaluation tests demonstrated significant performance gain. The journal paper discusses the PoD approach in more detail. We also introduced a new approach to evaluate queries involving the XPath ‘ancestor’ axis step. Moreover, we report on extended evaluation tests that cover a wide range of XML benchmarks. The two papers are based on the materials provided in Chapters 3 and 4.

6. Maghaydah, M., Orgun, M. A. & Khazali, I. (2010). Optimizing XML twig queries in relational systems. *The 14th International Database Engineering and Applications Symposium (IDEAS)*. Montreal, Canada, pp. 123-129.

This paper presented advanced optimisation techniques for efficiently evaluating complex structural-join XML queries in off-the-shelf relational database systems. These techniques make a relational database system more tree-aware without changing the kernel of the database system. We presented techniques to produce query execution plans with an optimal join order for XML queries in relational database systems. This has proven to play a significant role in improving the performance of XML queries. The second contribution aimed to reduce the number of joins for XML twig queries by a better utilisation of the XML document structure summary and features of Dewey labels. Finally, the paper discussed an experimental evaluation of our approach and two other established XML database systems using well-known XML benchmarks. The results demonstrated that our approach performed very closely to the Monet system [Boncz *et al.*, 2006], which is a mature and very well-established in-memory XML management system. Moreover, our approach outperformed the native XML database system eXist [Meier, 2006]. The paper demonstrates that off-the-shelf relational database systems still provide a competitive, robust and mature platform for developing XML management systems at an affordable cost. The paper is based on materials presented in Chapter 5.

Chapter 2: XML Overview

XML has emerged as a dominant language for Internet applications. This chapter introduces the XML technology, data model and XML documents. In addition, it highlights the development of XML query languages.

2.1 Introduction

XML is a framework for defining markup languages, which is used for representing structured information in a simple text-based format. Unlike Hyper Text Markup Language (HTML), which is used for web page formatting, there is no fixed collection of markup tags in XML. Instead, XML allows us to define our own specialised tags, tailored for the kind of information that we wish to represent. Each set of XML tags is developed for a particular application domain but they share many features: they all use the same basic markup syntax and they all benefit from a common set of generic tools for processing XML documents [Evjen *et al.*, 2007; Melton and Buxton, 2006; W3C, 2010b].

XML was originally designed to become the successor to HTML, more powerful than HTML yet less complex than the Standard Generalized Markup Language (SGML). SGML parsers require strict adherence to the DTD, making SGML too complex for everyday use, such as web publishing. Conversely, HTML parsers make no such demand, thereby making it difficult to add semantics to HTML data [Nambiar *et al.*, 2002]. Further, HTML tags describe how to display the data and they are primarily used for document formatting [Silberschatz *et al.*, 2002]. XML has been designed with some simple but powerful principles in mind; XML parsers do not require content to adhere to structural rules, yet XML documents must be well formed, a concept that will be explained in the following sections. Moreover, XML tags describe the data itself, so an XML document is in a self-described data format.

The XML and related technologies are still evolving; the W3C [W3C, 2010a] has released several editions of XML recommendations since the first edition in 1998 to refine and enhance the XML framework. The fifth edition of XML recommendations (current) was released in 2008 [Bray *et al.*, 2008].

2.1.1 XML Application Domains

XML was originally designed to meet the challenges of large-scale electronic publishing by isolating content from formatting. Separating the content of a document from how it is to be formatted simplifies development and maintenance. Different people from different fields expertise can work independently on the information captured in a document, on the format, style, and aesthetics [Melton and Buxton, 2006].

XML, with its expressive and extensible key features, has proven useful in data exchange; it can be used to exchange data on the web between applications or between applications and users. Using style sheets, such as eXtensible Stylesheet Language (XSL) or Cascading Style Sheets (CSS), XML documents can be easily transformed into a presentable format such as HTML.

Another key advantage of XML is its ability to integrate data and documents; most languages are designed to be better at expressing the rigid, absolute content and structure of data or the flexible, free-form text of documents, but XML does both equally well. It can capture the structure of scientific or financial data as well as formatting a letter for sending via email, or it can be used to publish a poem. Not only can XML represent data and free-form text, it can also do both in the same document.

XML is designed to communicate content in a flexible and extensible representation. Thus, descriptions of the data can be iteratively refined as the underlying domain changes. This makes XML particularly useful for areas whose knowledge has a complex organisation that undergoes frequent revision. Some of the early adopters of the XML technology include scientific areas such as biology, chemistry and mathematics. Scientific standards based on XML exist in those areas, namely, BioML, CML, and MathML.

There are other advantages for using XML, as provided by the W3C, XML will:

- Enable internationalised media-independent electronic publishing.
- Allow industries to define platform-independent protocols for the exchange of data, especially the data of electronic commerce.

- Deliver information to user agents in a form that allows automatic processing after receipt.
- Make it easier to develop software to handle specialised information distributed over the web.
- Make it easy for people to process data using inexpensive software.
- Allow people to display information in the way they want it under style sheet control.
- Make it easier to provide meta-data (data about information) that will help people find information and help information producers and consumers find each other.

2.2 XML Data Model

XML language and concepts related to its use are created within standards bodies, especially W3C. This is a new phenomenon; traditionally, artefacts in computer science were created in academic and industrial research labs before being adopted by the industry or standards committees. Researchers and academics have been involved in validating and improving XML capabilities, and they have been trying to find a sound theoretical model for XML data.

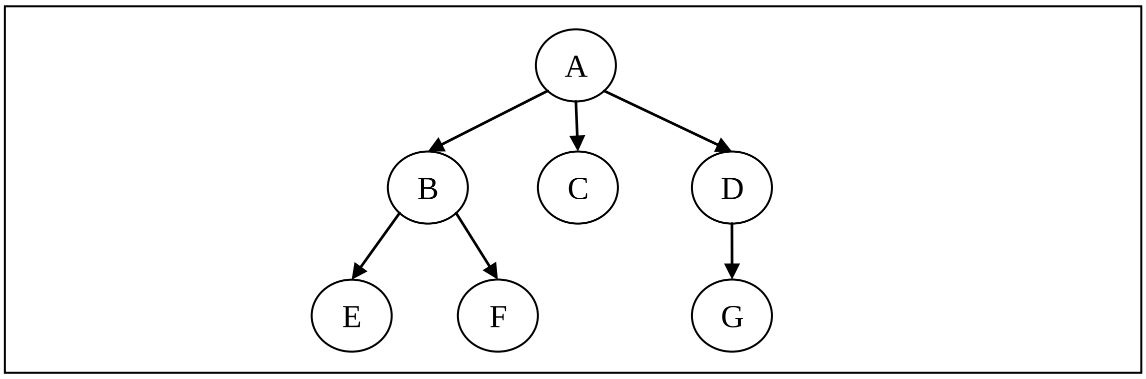


Figure 2-1: Sample XML tree representing the XML data model.

XML is sometimes perceived as a hybrid of other data models: hierarchical, relational and object-oriented. However, all data models proposed for semi-structured data represent that data as some kind of a labelled graph or tree, which also agrees with the XPath data model [Berglund *et al.*, 2007], in which:

- Nodes in the graph correspond to compound objects (elements) or atomic values.

- Each edge indicates an object-subobject or object-value relationship.
- Each node in the tree has only one parent node.
- Leaf nodes, i.e. nodes with no outgoing edges, have a value associated with them
- There is no separate schema and no auxiliary descriptions; the data in the graph is self-describing.

Figure 2-1 shows a sample XML tree; nodes are drawn as circles. The topmost node is called the root node. The edges show the parent-child relationship between the nodes, for example, node B is a child of node A.

2.2.1 Features of the XML Tree Data Model

The nested hierarchy structure of the XML data model is widely represented as tree-structured information. The XML tree is in top-down representation with the root node at the top of the tree. The XML tree is constructed mainly from two node types: first, the intermediate nodes, which are nodes with outgoing edges (i.e. they have child nodes), and they are known as parent nodes. The second type is the leaf nodes, they are nodes without any outgoing edges and they mainly contain textual values (i.e. they contain no child nodes). The content of a parent node is the sequence of its child nodes. In Figure 2-1, the content of node A is the sequence (B, C, D) and subsequently their child nodes (E, F, G).

Document order (i.e. the order of the nodes) is a major feature of the XML data model; the document order of a node represents its location within the document or in the XML tree using top-down and left-to-right tree traversal. For example, node B in Figure 2-1 is before node C in the document order.

The XPath query language for XML documents is based on the XML tree data model. However, XPath extends the simple tree model to define more features and properties; following is a brief description of the major aspects of the XPath data model:

The ancestors of a node: consist of the node’s parent, the parent of the parent, and so forth, all the way back to the root node, which is also included.

The descendents of a node: consist of the node's children, the children of the children, and so forth. However, that does not include attribute nodes.

The siblings of a node: the other children of the same parent node.

Element nodes: an element node defines a logical grouping of the information represented by its descendents.

Attribute nodes: An attribute node is associated with an element node, that is, its parent is always an element. An attribute is a pair of name and value. Every element can have at most one attribute of a given name. However, the attributes of a node are not included with the children group of the node.

Comment nodes: A comment node is a special leaf node labeled with a text string. They are ignored by processing tools.

Processing instruction nodes: A processing instruction node has a target node and a value, and can be used to convey specialised meta-information to various XML processing tools. For example, the target could be an XML-stylesheet, which is recognised by XSLT processors, and the value a URI reference to an XSLT style sheet used by such a processor.

The root node: Every XML tree starts with a single root node, which represents the entire document. The children of the root node consist of any number of comment and processing instruction nodes together with exactly one element node, which is called the root element.

2.2.2 XML Schema

An XML schema is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of the documents of that type, above and beyond the basic syntactical constraints imposed by XML itself. These constraints are generally expressed using some combination of grammatical rules governing the order of elements, Boolean predicates that the content must satisfy data types governing the content of elements and attributes, and more specialised rules such as uniqueness and referential integrity constraints.

There are languages developed specifically to express XML schemas. The Document Type Definition (DTD) language [Bray *et al.*, 2008], which is native to the XML specification, is a schema language that is of relatively limited capability, but it has other uses in XML apart from the expression of schemas. However, the DTD language has a number of limitations [Connolly and Begg, 2010]:

- It is written in a different syntax (non-XML).
- It has no support for namespaces.
- It only offers extremely limited data typing.

The W3C has also developed and approved the XML schema language (which has been renamed recently to XML Schema Definition, XSD) [W3C, 2004] as a successor for DTD; XSD is a more expressive XML schema language and is widely used. The XML schema language specifies how each type of an element in the schema is defined and with which type that element is associated. Moreover, the schema is itself an XML document; it can be processed by the same tools that are used to process XML data. An XML schema can be used to:

- Provide a list of elements and attributes in a vocabulary.
- Associate types, such as integer and string, or more specifically, user-defined types, such as hatsize and sock_colour, with values found in documents.
- Constrain where elements and attributes can appear, and what can appear inside those elements, such as saying that a chapter title occurs inside a chapter, and that a chapter must consist of a chapter title followed by one or more paragraphs of text.
- Provide documentation that is both human-readable and machine-processable.
- Give a formal description of one or more documents.

Figure 2-2 shows a sample XML schema segment. Information in schema documents is often used by XML-aware editing systems so that they can offer users the most likely elements to occur at any given location in a document. Checking a document against a schema is known as validating against that schema; validating against a schema is an important component of quality assurance.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="note">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="to" type="xs:string"/>
   <xs:element name="from" type="xs:string"/>
   <xs:element name="heading" type="xs:string"/>
   <xs:element name="body" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Figure 2-2: Sample XML schema using XSD language.

XML schema can also be used as a base for mapping from XML schema to relational schema, which produces an optimised relational storage for XML data [Deutsch *et al.*, 1999; Lee *et al.*, 2003; Shanmugasundaram *et al.*, 1999]. However, in many XML applications, the data schema might not be available, or due to the nature of some applications in which the document structure keeps changing, it is essential to develop XML management systems that are able to store and query XML data in the absence of XML document schema [Florescu and Kossmann, 1999]. An alternative approach is to capture the document structure summary during parsing of the original XML document; the main use of the structure summary is to develop efficient XML query optimisation techniques [Moro *et al.*, 2008]. Further discussion is provided in Chapters 4 and 5.

2.3 XML Documents

From the data perspective, XML is a standardised approach to storing text-based data in a hierarchical manner and to defining metadata about these data. The data and their representation come together in one unit, which is called an XML document. No matter which conceptual data model is used to represent the XML data model, the actual textual representation of an XML document is the same.

An XML document is written as a Unicode text with markup tags and other meta-information representing the elements, attributes, and other nodes. An XML document consists of three parts:

- XML declaration: starts with <?xml> declaration, which specifies the version and encoding of XML being used, and whether the document is standalone or references other documents.
- XML schema: which can be in DTD or (recently) in XML-schema specifications. It constrains the structure of XML instances and data typing, and corresponds to an extended context-free grammar. XML processors can use a schema to validate an XML document and report errors if the document does not comply with rules and definitions in the schema.
- XML instance: a hierarchy of elements. These elements are nested and start with a root element.

The first two parts are not mandatory.

2.3.1 XML Syntax

The syntax of XML can be summarised in the simple definition of the XML document. An XML document is a hierarchy of nested tagged elements, which may have attributes, starting by a root element. There are no restrictions or predefined set of tags in XML, which gives XML its flexibility and power. Figure 2-3 shows a sample XML document.

Some components of an XML document structure are:

- XML Tags: describe the structure of a document and identify its content. XML tags are opened and closed with angle brackets, ‘<’ and ‘>’, the identification name, or label, is located within these brackets (i.e. <TAGNAME>).
- Elements: provide additional information about the contents of the document. The tags of elements must be properly formatted and paired for the parser to work correctly:
    - Element naming rules:
        1. Names are case specific.
        2. No white space is allowed.
        3. Names cannot start with numbers.

```
<bookstore>
 <book id = ‘B001’>
   <author> Brundage, Michael </author>
   <title> XQuery: The XML Query Language </title>
   <genre> Computer </genre>
   <publish_date> 12-02-2004 </publish_date>
   <description>
    An excellent early look at the emerging XML Query standard
   </description>
 </book>
 :
</bookstore>
```

Figure 2-3: Sample bookstore XML document.

- Attributes: provide metadata about the information contained within an element. Unlike subelements, no order is imposed on attributes. They are defined within the starting tag of an element to which they belong.
    - The use of attributes can provide:
        1. Additional details
        2. A means to control the appearance of an element in addition to style sheets
- Namespaces: give documents the ability to use multiple elements of the same name within the same document. A uniform resource locator (URL) can be attached to an element to specify their correct sources:

  <element name Xmlns: prefix= ‘namespace URL’>.
- XML documents can have a type of referential integrity by using the ID and IDREF attribute types. An element with the attribute type of IDREF can reference another element with the ID attribute type, which happens uniquely within the document.

2.3.2 A Well-formed XML Document

The syntax rules of XML are strict: XML tools will not process files that contain errors, but instead will report error messages so the developer can fix them. This means that almost all XML documents can be processed reliably by computer software. Satisfying these strict rules produces a well-formed XML document. An XML document is well formed if the following conditions are adhered to:

- It has a unique root element that contains all the other elements.
- It contains only properly encoded legal Unicode characters.
- None of the special syntax characters such as ‘<’ and ‘&’ appear except when performing their markup-delineation roles.
- The opening and closing brackets, which delimit the elements, are correctly nested, with none missing or overlapping.
- The element tags are case-sensitive; the beginning and end tags must match exactly.
- An attribute can occur at most once in a given opening tag, its value must be provided, and this value must be quoted [Lewis *et al.*, 2002].
- Empty elements can be closed as normal, ‘<happiness></happiness>’ or a special short-form, ‘<happiness />’ can be used instead.

All XML documents must be well formed.

2.3.3 A Valid XML Document

An XML document is valid if it is well formed and conforms to the rules and definitions in a schema file (such as a DTD or XML schema XSD file). XML data parsers and processors can be configured not to check document validity. However, they must check that the document is well formed before passing the information in the document on to the application.

2.4 Querying XML Data

With the rapid increase in the use of XML documents, there is an increasing need to be able to extract information from these documents. Various XML query languages have been proposed such as Lorel [Abiteboul *et al.*, 1997], XML-QL [W3C, 1998], XQL [W3C, 1999], and Quilt [Chamberlin *et al.*, 2000]. However, most of these have been associated with certain proposed storage systems. Recently, the XQuery/XPath language has emerged as the standard query language for XML data [Fernandez *et al.*, 2007]. XQuery is derived from an XML query language called Quilt [Chamberlin *et al.*, 2000]. XQuery provides the means to extract and manipulate data from XML documents or any data source that can be presented as XML, such as relational databases or office documents.

Sample XPath queries

```
/bookstore/book/title        (returns all the book titles in the entire document).
//book[price >40]/title      (returns all the book titles only with prices above $40).
```

XQuery representation for the second XPath query

```
For $x in doc('books.xml')//book
Where $x/price > 40
Return $x/title
```

Figure 2-4: Sample XQuery/XPath expressions.

SQL/XML is an ANSI and ISO standard that specifies SQL-based extensions for using XML in conjunction with SQL. The XML data type is introduced, as well as several routines, functions and XML-to-SQL data type mappings, to support manipulation and storage of XML in the context of a relational database system that supports SQL. As SQL is the standard language for accessing and managing data stored in relational databases, it is natural that enterprises and users worldwide need the ability to integrate their XML data into their relational data through the use of SQL facilities [SQLX.org, 2004].

XQuery and its subset XPath have been developed and standardised by the W3C. While XPath can only perform selections on an XML document, XQuery supports richer operations (joins, aggregations and element construction). However, XQuery uses the XPath syntax for locating paths within the XML documents. XPath uses expressions that are composed of a sequence of location steps separated by the '/' symbol. Figure 2-4 shows some sample XPath and XQuery expressions using the sample XML document in Figure 2-3.

The major vendors of RDBMS are providing support for SQL/XML specifications to support XML documents that are stored in relational database systems (such as MSSQL server) [Pal *et al.*, 2005a]. SQL language and XQuery language can be used together to access relational data or XML data.

However, many off-the-shelf relational-based systems do not have an XQuery interface and XML data are shredded down and stored into relations (tables). The best approach to support queries on XML data is to translate the query to equivalent SQL statements that are supported in those systems. Further details are provided in Chapters 4 and 5.

2.4.1 XPath Path Expressions Overview

The XPath path expressions can be used to locate nodes within XML documents. They consist of a series of one or more steps separated by "/" or "//", and optionally beginning with "/" or "//" in the following format [Berglund *et al.*, 2007]:

```
PathExpr            ::=   ("/" RelativePathExpr?)
                          |("//" RelativePathExpr)
                          | RelativePathExpr
RelativePathExpr    ::=   StepExpr (("/" | "//") StepExpr)*
```

An initial "/" or "//" is an abbreviation for one or more initial steps that are implicitly added to the beginning of the path expression, a ‘/’ at the beginning of a path expression is an abbreviation for the initial step ‘root’ which is treated as ‘document-node()’. A ‘//’ at the beginning of a path expression is an abbreviation for initial step ‘root’ and its descendent nodes (document-node()/descendent-or-self::node()/’.

StepExpr is a part of the path expression that generates a sequence of items and then filters the sequence by zero or more predicates. The value of the step consists of those items that satisfy the predicates, working from left to right. A step expression may be either an axis step or a filter expression as in the following syntax:

```
StepExpr        ::=   FilterExpr | AxisStep
AxisStep        ::=   (ReverseStep | ForwardStep) PredicateList
ForwardStep     ::=   (ForwardAxis NodeTest) | AbbrevForwardStep
ReverseStep     ::=   (ReverseAxis NodeTest) | AbbrevReverseStep
PredicateList   ::=   Predicate*
```

Chapter 4 provides a detailed discussion on our approach to optimise XPath queries in relational database systems based on optimising the evaluation of XPath axis steps.

2.4.2 XQuery Overview

XQuery is designed for querying XML data that represents the tree data model; each document is regarded as an ordered tree of nodes or items. The items can be simple (i.e. atomic) such as integers, dates and strings. Items can also be complex, which are nodes that can contain other nodes (i.e. elements).

Every input to a query is an instance of the data model as well as every output returned by the query. In other words, the input and output can be an XML document or a fragment of an XML document.

In XQuery, a series of items are called sequences. A sequence may contain nodes, atomic values, or any mixture of nodes and atomic values. However, a sequence cannot contain another sequence. When sequences are combined, the result is always a ‘flattened’ sequence, for example, the sequence (a, (b, c)) is equivalent to the sequence (a, b, c).

One of the main features of XQuery is the FLWOR expression (pronounced as flower), which has a query structure similar to that of SQL and it consists of clauses such as FOR, LET, WHERE, ORDER BY, and RETURN. The FOR and LET clauses use XPath expressions to bind parts of the XML documents to the variables that will be used in the query. The difference between FOR and LET is that the FOR clause iterates over a sequence of items and successively binds a variable $v to be used in the rest of the query. LET clause binds a variable $v to an item or a whole list of items. Other clauses have almost the same meaning and use as in the SQL queries.

The Query example in Figure 2-6 below depicts a typical FLWOR query incorporating XPath expressions evaluated on the sample XML tree representation in Figure 2-5. The shaded nodes in Figure 2-5 represent the tree branch that needs to exist in the document to satisfy the query in Figure 2-6.

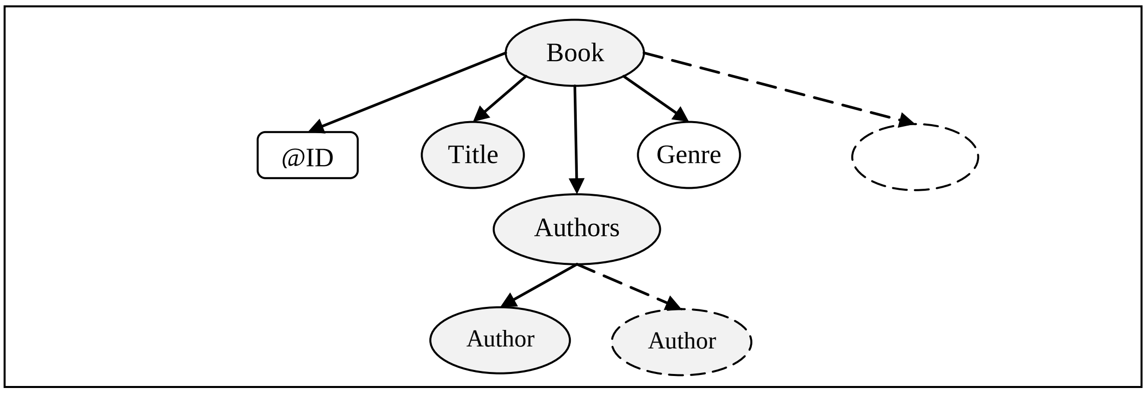


Figure 2-5: Sample XML tree fragment from sample bookstore XML document.

XQuery FLWOR example

```
For $x in doc('books.xml')//book
LET $y := $x//author
Where count ($y) > 2
Return $x/title
```

An equivalent query in SQL

```
SELECT title
FROM books
GROUP BY title
HAVING count(author) > 2
```

Figure 2-6: Sample XQuery and equivalent SQL query.

Chapter 5 discuses optimisation techniques for complex structural-join queries and XML twig queries. Moreover, we explain our approach to translate XQuery queries into optimised SQL statements.

2.5 Conclusion

XML has emerged as a standard format for data exchange and representation, in particular, for web applications. Its use has been growing rapidly due to its simple yet powerful data model and structure.

XML schema provides a mechanism to define constraints on the structure and content of the documents of that type. However, the schema definitions may not be available in many applications. In this thesis, we have decided to follow the approach that is based on the absence of XML schema for the following reasons:

- For many applications, the XML schema (i.e. DTD or XSchema) may not be available.
- The query optimisation techniques to be developed for this approach can be portable, scalable and DTD independent.
- It is a more appropriate technique to support dynamic XML documents.

New technologies have also been developed to store and query XML data; the XQuery/XPath language has been approved by the W3C as the standard query language for XML data. In Chapters 4 and 5, we provide a detailed study about optimising XQuery/XPath queries in relational database systems in the presence of a Dewey-based labelling scheme and document structure summary.

Chapter 3: Prefixing on Demand Labelling Approach

Maintaining the document order and structure in a given XML document is the most important feature, yet it is the most challenging task that XML management systems need to efficiently support. Storing an XML document in native XML systems can maintain a document's order and structure, since XML documents will be stored in their native format. However, it will be very expensive to answer queries by parsing and scanning the stored documents during run time, especially for large XML documents. The problem is far more complex in relational systems because XML documents are typically shredded and stored as individual records in database tables. The widely used approach to address this problem is to capture the document order and structure as data and store it beside the actual document data. This process is called 'labelling and indexing' XML document nodes (i.e. elements and attributes).

However, the labelling technique determines the features and the strength of the XML database system that is built on top of it. Label size and structure play a major role in determining the efficiency of the system since they determine storage requirements and the capability of addressing query requirements such as XPath axis steps. This chapter introduces our Dewey-based labelling approach, which provides a more compact and more efficient labelling scheme that better utilises the current relational systems indexing mechanisms.

3.1 Introduction

XML query languages like XQuery/XPath [Fernandez *et al.*, 2007] have been designed to retrieve information from XML documents using two main techniques:

1. The path traversal technique, which uses a sequence of tag names to determine which parts of a given XML document are targeted by any given XML query.
2. The containment join and structural-join techniques, which determine the relationship between a document's nodes as ancestor-descendent, sibling and parent-child relationships.

Several labelling techniques have been proposed to address the document order issue and to support containment-based queries. Some of those techniques have been widely used in both native and relational XML database systems [Beyer *et al.*, 2006; CWI, 2009; exist-db.org, 2009; Jagadish *et al.*, 2002; Oracle, 2009; Pal *et al.*, 2005b; UWMRG, 2002]. The underlying idea in all these techniques is to uniquely identify each node in the document by allocating a unique value or a set of values (numbers or strings) to each node, which also helps to determine the relationship between nodes (i.e. node 'B' is after node 'A' in the document order if label of 'B' > label of 'A'). There are two major labelling techniques: the numeric-based range encoding and the binary string Dewey-based encoding. We cover the features of each technique in the next section. However, Dewey-based labels support dynamic XML documents (i.e. inserting and deleting) more efficiently than other labelling methods.

In this chapter, we revisit our motivation to develop a new labelling technique as a basis for the rest of the work in this thesis. In Section 3, we explain in detail our proposed labelling scheme. Section 4 contains an extensive evaluation study and analysis for our labelling technique, and present comparison studies between our approach and other well-known labelling methods.

3.1.1 Motivation

The Dewey-based labelling technique for XML nodes has emerged as the most suitable labelling approach to support dynamic XML documents [Tatarinov *et al.*, 2002]. It supports variant operations on dynamic XML documents, from inserting large sub-trees without relabelling the existing nodes to fine-grained locking thereby avoiding access to external storage as much as possible [Härder, 2005]. The Dewey label of any given node, known as the context node, is the concatenation of the local order values of all the nodes on the path from the root node to the context node (for example, 1.5.8.1). The work by Tatarinov *et al.*, [2002], which was the first to introduce Dewey labels to XML systems, encoded the Dewey labels in 8-bit unicode transformation format (UTF-8) strings. The major drawback of UTF-8 is its inflexibility since its compression is poor for small ordinals (for example, 1.1.1.1 uses four one-byte components).

Several approaches have been proposed based on the prefix-free algorithm and Dewey identifiers [Böhme and Rahm, 2004; Cohen *et al.*, 2002; Li and Ling, 2005; Lu *et al.*, 2005; O'Neil *et al.*, 2004]. ORDPATH [O'Neil *et al.*, 2004] is a recent Dewey-based approach used in the Microsoft SQL server [Pal *et al.*, 2005b]. ORDPATH eliminates the need for relabelling the existing nodes in a given XML document when new nodes are inserted. However, ORDPATH reserves even numbers for future insertion, which is considered a waste of space because in real applications, we do not insert new nodes between every two existing nodes. Further, it uses the variable length prefix-free technique [Cohen *et al.*, 2002; Kaplan *et al.*, 2002], which is not the ultimate compression technique.

Dewey labels, including ORDPATH, rely on a set of functions to manipulate the labels and evaluate relationships between nodes, in particular, a function to retrieve the parent node's id and a function to find the upper limit of the label value for descendent nodes. Using functions may not make efficient use of the indexing technology in relational systems, especially in the widely used open source systems like MySQL server [MySQL, 2009].

We observe that there is scope for improving the label size and performance of the current Dewey-based approaches, such as ORDPATH [O'Neil *et al.*, 2004], by eliminating the need for complex and excessive variable-length prefix-free strings. Further, current approaches waste space in favour of providing efficient support for frequent node insertion; meanwhile, most of the XML applications are static or require minor update operations. Our approach provides better size compression without losing support for dynamic XML documents. Moreover, we propose an enhanced Dewey-based label structure that reduces the use of functions mainly for the parent-child and sibling relationships to exploit the existing indexing mechanisms in off-the-shelf RDBMS. We also observe a need for capturing and storing the XML document schema and utilise it with the Dewey labelling scheme to further improve the XML query performance in relational-based systems.

3.2 Background

Recently, labelling XML document nodes has become more important than just a technique to capture the document order due to the rapid increase in the use of XML in Internet applications. XML database systems rely on the labelling and indexing techniques to efficiently answer more practical and yet more complex queries that are based on structural-join queries. However, the labelling scheme should have the following characteristics:

- Deterministic: The relationships between two nodes can be determined uniquely and quickly by simply examining their labels.
- Dynamic: Updating XML documents will not require relabelling of the existing nodes in the XML trees.
- Compact: The size of the labels should be minimal in order to fit in the main memory.
- Flexible: The scheme can be used to support all kinds of XQuery/XPath functions.

This section covers in detail the related work in this area. The discussion of related work is divided into two main subsections. The first subsection provides a detailed review of labelling and indexing techniques in XML database systems, some of these techniques apply to both native and relational systems. The second subsection includes schema-mapping approaches for mapping between the XML schema and the relational schema.

3.2.1 Labelling and Indexing Techniques in XML Database Systems

The XML query is initially processed by tree traversal. In the Lore system [McHugh *et al.*, 1997], which was designed specifically for semi-structured data, an XML document is modelled as a labelled directed graph. Elements are represented as objects with a unique object id (oid). Objects can contain other objects; a special indexing mechanism (Link Edge Index) links each element to its ancestor nodes. Lore engine is built around standard operators (such as scan and join). Scan operator returns all oids that are sub-objects of a given object. In Lore, evaluating some queries might be very expensive since scan operator involves top-down traversal, whereas indexes support bottom-up traversal. The DataGuide [Goldman and Widom, 1997] is utilised as a summarisation for the path information in the XML file. Piloted by DataGuide, the query processing system can conduct a vertical tree traversal to determine whether there exists any ancestor-descendant relationship between two nodes. However, such a tree traversal-based technique is dependent on the XML document size and structure, and on the XML query complexity.

The Edge system [Florescu and Kossmann, 1999], which is a relational-based system, uses a simple labelling technique by parsing the document depth-first and assigns a global unique number to every node. Nodes are stored in a single relational table along with their parent id (Pid) and tag name. Query evaluation might be very efficient for simple queries based on parent-child and sibling relationships. However, evaluating ancestor-descendent relationships is very inefficient and might be impossible due to the multiple numbers of self-join operations. Further, path traversal queries in this system are costly because there is no path summary information. However, Edge pioneered the development of XML database systems on top of existing relational technologies.

To overcome the problems of Edge, a number of researchers have proposed new labelling schemes such that it is possible to determine the relationship between any two XML nodes by comparing their label values only. We classify these labelling techniques into two major approaches: the numeric-based range encoding and Dewey-based encoding schemes.

3.2.1.1 Range and Intervals Encoding Techniques

The range encoding, or containment encoding, was proposed to support ancestor-descendant relationships [Yoshikawa *et al.*, 2001]. Nodes are labelled by two values that represent boundaries for other nodes that are contained by the context nodes. These boundary values are commonly called Start (S) and End (E) values. The start and end values can be assigned based on the number of bytes within the document, which represent the location of the start (or opening) and end (or close) tags. However, updating or inserting new nodes is very costly since minor changes, even updating values, can trigger the relabelling of all nodes. More work was performed to address this problem but the node positions were still based on counting the number of bytes in the document [Kha *et al.*, 2001].

As an alternative, the start and end values can be generated by performing a depth-first traversal of the XML tree and sequentially assigning a number when a node is visited for the first time, the Start (S) value, and another number after visiting all the descendant nodes of the context node, the End (E) value [Zhang *et al.*, 2001]. Further, this approach also stores the tree level of each node; the label of each node is represented as a tuple (S, E, L), which can be used to support the parent-child relationship.

Based on the range encoding, a node 'A' is an ancestor of node 'B' if and only if:

$$(A.S < B.S) \text{ And } (A.E > B.E)$$

and B is a child of A if and only if:

$$(A.S < B.S) \text{ And } (A.E > B.E) \text{ And } (A.L + 1 = B.L)$$

The range (or intervals) approach efficiently supports static XML documents. However, the insertion of new nodes can trigger an expensive relabelling process for some of the existing nodes. One of the proposed solutions is to skip a range of numbers and reserve that for inserting new nodes; the end value is replaced by the size of the node and the label becomes the tuple (start, size, level) [Jagadish *et al.*, 2002; Li and Moon, 2001]. The size represents the number of all descendent nodes that are contained by the context node, as in Figure 3-1.

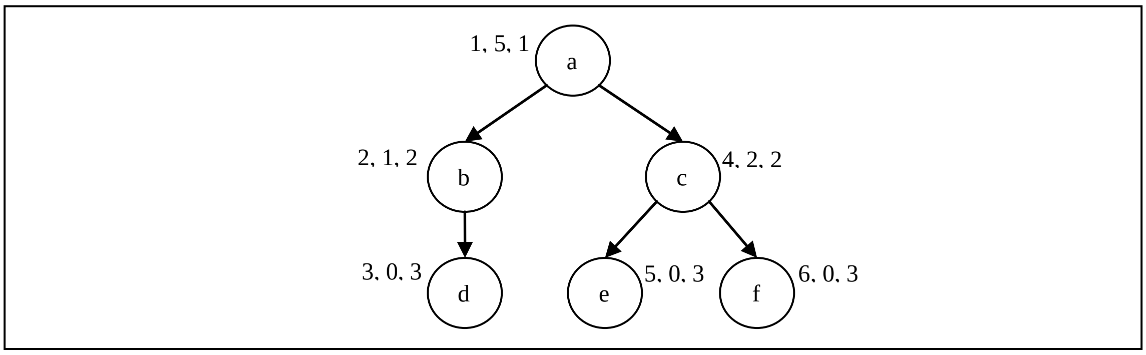


Figure 3-1: Region encoding representation (pre, size, level) for a sample XML tree.

To support the insertion of new nodes, the size value is made larger than the actual number of existing nodes. Relabelling is not required until all spare numbers have been exhausted. The work by Amagasa and Yoshikawa [2003] used floating-point values for numbering the start and end values; the use of floating-point value still does not completely eliminate relabelling since a floating-point value is represented in a computer with a fixed number of bits, which again places a limit on the number of nodes that can be inserted between two adjacent nodes without triggering the relabelling process.

To support certain XPath axis steps such as ‘preceding’ and ‘following’, an enhanced region labelling technique was proposed [Grust *et al.*, 2004]. It uses the same pre-order value as in other region methods; however, the post-order value for the context node represents the number of closing tags before the context node. In other words, it represents the number of nodes before the context node excluding its ancestor nodes.

3.2.1.2 Prefix-free String and Dewey Labels

Due to the limited capabilities of the intervals labelling approach in supporting dynamic XML documents and its inefficient support for some of the XPath axis steps, such as reverse axis steps, a new labelling technique was proposed in which the nodes inherit their parents’ labels as the prefix to their own labels [Abiteboul *et al.*, 2001; Cohen *et al.*, 2002; Kaplan *et al.*, 2002]. In this prefixing scheme, the ancestor-descendant relationship between any two nodes can be evaluated by testing wether one label is a prefix of the other. The labels are bit strings that are built by concatenating ‘0’ and ‘1’ as appropriate to the parent’s label to ensure that:

- Labels are unique.
- Labels are in lexicographic order.

The work by Kaplan *et al.* [2002] studied two cases: fixed-width components and variable-width prefixes. The two techniques provide a trade-off between label size and complexity. Further, the support for insertion of new nodes still has some limitations.

The Dewey labelling concept was first introduced by Tatarinov *et al.* [2002]; a Dewey label is built by concatenating the integer values that represent the local order of each node on the path from the root node to the context node. Figure 3-2 shows a sample XML tree with Dewey labels assigned to the tree nodes.

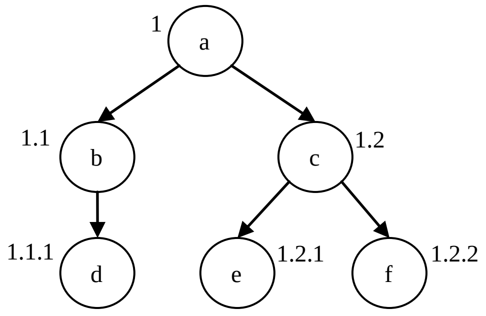


Figure 3-2: Dewey label representation for a sample XML tree.

To avoid using any delimiter (such as the dot '.'), the labels are encoded and stored in UTF-8 format. The major drawback of UTF-8 is its inflexibility since its compression is poor for small ordinals, for example, the label 1.1.1.1 uses four one-byte components.

This approach has reduced the cost of relabelling, since relabelling is still required for all sibling nodes and their descendants after the insertion point. Many later works based on Dewey identifiers focused primarily on reducing the label size and completely eliminating tree relabelling [Böhme and Rahm, 2004; Ha¨rder *et al.*, 2005; Haustein *et al.*, 2005; Li *et al.*, 2006; O'Neil *et al.*, 2004] .ORDPATH is a recent and popular Dewey-based labelling technique that completely eliminates the need for relabelling the existing nodes when new nodes are inserted [Haustein *et al.*, 2005; O'Neil *et al.*, 2004]. It uses the odd numbers for initial labelling and reserves the even numbers as insertion points. Figure 3-3 demonstrates the ORDPATH approach; the insertion of node 'g' does not require tree relabelling.

To reduce the label size, ORDPATH uses the prefix-free algorithm to generate the local value before it concatenates all the local values in a compact bit string label; each local value has the format:

$$O.L$$

Where the *O* represents the prefix-free component, which also indicates the length of the *L* component, and *L* represents the actual local order.

For example, the '*O*' component can be the binary string '001', which is also mapped to a label length '*L*' of 6 bits; the local order of the fourth node is '000100', which leads to the complete label in the form '001000100'.

However, the label size is still considered long and replacing the complex prefix-free technique would save space and reduce overhead processing. Further, the even numbers are reserved for future insertion, which is considered a waste of space, since in real applications we do not insert new nodes between every two existing nodes.

QED [Li and Ling, 2005] is a compact encoding technique, which can be applied to both containment and Dewey labelling techniques. QED provides efficient support for frequent node insertions; however, QED requires XML documents to be parsed twice when they are stored for the first time. Further, QED-Prefix, which uses the special code

‘0’ (2 bits) as a separator between the values that compose the Dewey label, does not provide a significant label size reduction over ORDPATH for XML documents with small fan-outs. Both ORDPATH and QED-Prefix add further processing time overhead since both techniques still require, as in the earlier prefix labelling techniques, parsing labels bit-by-bit to retrieve different components (i.e. the parent label).

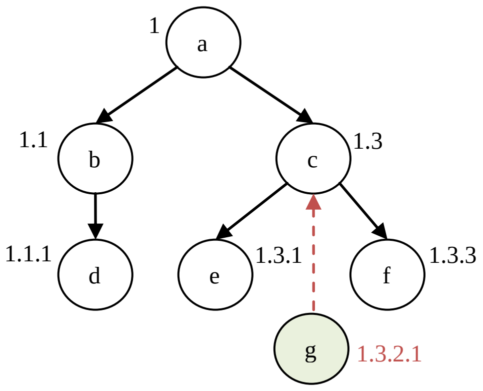


Figure 3-3: ORDPATH labelling scheme, which eliminates relabelling.

3.2.1.3 Other Labelling Approaches

A few labelling approaches have been proposed based on ideas and properties not considered in the major two approaches in the previous two subsections. The reasons behind these approaches are primarily to produce smaller size labels or to support certain XML features efficiently. Prime [Wu *et al.*, 2004] is a number-based labelling scheme that uses the property of prime numbers; the label of each node is the product of its own self-label (a unique prime number) and its parent’s label. Prime mainly supports ancestor-descendent relationships; node A is an ancestor of node B if the label of node B is divisible by the label of node A. When the ancestor-candidate node list has m nodes and the descendant-candidate list has n nodes, m x n scans are necessary to evaluate any structural-join operation. Prime does not provide a significant saving on the label size, especially for deeply nested documents, and the results cannot be sorted by the node’s label to reflect the document order; however, a special mapping algorithm is used to reflect the document order.

EXEL [Min *et al.*, 2007], similar to VLEI code [Kobayashi *et al.*, 2005], is a recent approach that aimed to completely eliminate the relabelling problem in the range encoding technique by replacing the numbers for start and end values with bit strings values. The bit strings are in the lexicographical order, which means the range operators (i.e. ‘<’, ‘>’, and ‘=’) are still applicable. While EXEL and VLEI address the insertion of new nodes, there is no significant saving of label size; especially, there are two values (Start and End) in bit strings format for each node. Moreover, arithmetic operations on numbers are faster than operations on bit strings are.

More sophisticated labelling and indexing techniques have been proposed to make the region encoding (pre and post) more insertion friendly. However, these techniques may add processing overhead as they require special indexing technology [Silberstein *et al.*, 2005], and arithmetic operations (for example, division) to determine relationships between data nodes [Weigel *et al.*, 2005].

3.3 Prefixing on Demand (PoD)

This section discusses in detail our labelling approach ‘Prefixing on Demand’ (PoD). The PoD is a Dewey-based labelling scheme, which can be used in both native and relational XML systems. However, this thesis focuses on developing a relational solution for XML management systems.

3.3.1 Basic Labelling Unit (BLU)

As was mentioned in the previous sections, Dewey-based labels are more suitable to support dynamic XML documents. However, the label length is still relatively long since most of the labelling approaches focus on supporting the insertion of multiple nodes at the same point and between every two existing nodes. In fact, most XML applications are static applications or rarely modifying XML documents.

The Basic Labelling Unit (BLU) approach aims to reduce the label length by replacing the complex prefix-free algorithm with a fixed-width prefix, which reduces overhead processing. Moreover, our approach reduces the label length by using most of the numbers available for labelling within any range, which means it reduces the need for long labels (i.e. for a label of length $\mathcal{L}$ bits: $2^{\mathcal{L}}$ nodes can be labelled). PoD fully maintains the document order and supports dynamic XML documents.

The basic concept of our labelling scheme is to have a BLU with a fixed length of ($\mathcal{L}$ bits). Most of the values in the BLU will be used to label nodes based on their document order. To address a large number of nodes, some of the highest values within the BLU will be preserved for prefixing the extended labels. This technique makes it possible to label more nodes using shorter labels before the need to use extended labels arises.

Proposition 3-1: *A fixed number of bits ($\mathcal{L}$) can be used to label up to $\mathcal{M}$ number of XML elements within the same parent node where the label value (v) does not have a prefixing component and the value (v) is in the range:*

$$0 \leq v < \mathcal{M} \tag{1}$$

$$\mathcal{M} = 2^{\mathcal{L}} - \mathcal{X} \tag{2}$$

Where $\mathcal{X}$ is the number of the highest values within the BLU that are preserved for prefixing the extended labels. We call those values the set of prefix values $\mathcal{P}$, where:

$$\mathcal{M} \leq p < 2^{\mathcal{L}},\ \forall\ p \in \mathcal{P} \tag{3}$$

Based on formulas (**1**), (**2**) and (**3**), we have the following properties:

- We can label up to $\mathcal{M}$ sibling nodes without any need for prefixing; where the label value $v \in \{0,\ldots, \mathcal{M}\text{-}1\}$ and the length (v) = $\mathcal{L}$.
- Each value p in the set $\mathcal{P}$ is used to prefix a certain extended label.
- The number of the extended labels ($\mathcal{X}$), their corresponding lengths ($\mathcal{L}`_1$, $\mathcal{L}`_2,\ldots,$ $\mathcal{L}`_x$) and the distance between them ($\mathcal{L}`_j$ - $\mathcal{L}`_i$) can all be adjusted to achieve an optimal label length based on the document structure and the average fan-out value if prior knowledge about a given XML document is available.
- The extended labels are of the format: $p.v`$, where $p \in \mathcal{P}$ and $v` \in \{0,\ldots, 2^{\mathcal{L}`}\}$.

Example 3-1: For a BLU of length $\mathcal{L}$ = 3 and the number of extended labels $\mathcal{X}$ = 2, based on the proposition (3-1) above, we will have:

- $\mathcal{M} = 2^3 - 2 = 6$, which is the maximum number of child nodes that can be labelled without prefixing or using extended labels {0,…, 5}
- p ∈ P = {6, 7}, prefixing values for the extended labels
- $\mathcal{L}_i` \in \{5, 7\}$; the length of the extended labels (an arbitrary selection for this example)

- $v`$ ∈ {0,..,32} for $\mathcal{L}`_1 = 5$, which will label sequence nodes in the range {6 to 38} since the first range {0 to 5} can be labelled using the BLU itself
- $v`$ ∈ {0,..,128} for $\mathcal{L}`_2 = 7$, which will label sequence nodes in the range {39 to 167}.

The setup in this example can label nodes with fan-out values of up to 167.

Table 3-1 shows some Dewey-based labels and their representations using PoD labelling scheme with the setup in Example 3-1.

Table 3-1: Dewey and PoD representations for some sample labels in Example 3-1.

Dewey Label	PoD Labelling Representation
1.5.2.1	001.101.010.001
1.20	001.$\underline{110}$ 01110
1.103.4	001.$\underline{111}$ 0100011.100

Note: the 3-bit components in underline font represent the prefix part of the label. In addition, the dots between labels components were added for ease of reading.

3.3.2 Supporting the Insertion of New Nodes

The Dewey-based labelling techniques for XML documents support the insertion of new nodes without any need to relabel any of the existing nodes. One approach to achieve this is to use either the even or odd values as insertion points, as in ORDPATH [O'Neil *et al.*, 2004]. For instance, the odd numbers may be used to label the nodes based on their order in an XML document, and the even values will be considered as virtual parents for the new inserted nodes. For example, if a new node were inserted between two adjacent existing nodes with label values 1.1 and 1.3, the new node label value would be 1.2.1.

This approach has a size disadvantage regardless of the compression technique used to build the Dewey labels because the actual number of values that are available for labelling in any range (i.e. 4-bit or 6-bit) would be half of the total number of values in that range. This means that the extended and longer labels would be used more often. For example, for a 4-bit string, the total number of values is $2^4 = 16$, but only half of that number (8) will be available for labelling XML nodes and to label more nodes (>8) within the same parent node (i.e. node's fan-out >8); thus, extended labels need to be used (i.e. 6-bit or 8-bit length labels).

PoD provides an alternative approach that can support the insertion of new nodes without relabelling the existing nodes, whilst simultaneously maintaining shorter label lengths by not using most of the numbers available within any given range. Based on the BLU that is used in PoD, we allocate the maximum (the last) value in any given BLU as an insertion point instead of using it to prefix extended labels.

Proposition 3-2: *Given a BLU with length ($\mathcal{L}$) and two consecutive nodes $\mathcal{N}_1$ and $\mathcal{N}_2$ with labels $\mathcal{V}_1$* and *$\mathcal{V}_2$ respectively, where $\mathcal{V}_2 > \mathcal{V}_1$, if a new node $\mathcal{N}$ is inserted between $\mathcal{N}_1$ and $\mathcal{N}_2$ then the label for node $\mathcal{N}$ is:*

$$\mathcal{V}_1.\,(2^{\mathcal{L}}-1).1 \qquad \textbf{(4)}$$

Example 3-2: For a BLU with $\mathcal{L} = 4$, the values (12, 13, 14) in this BLU are preserved for prefixing the extended labels. Based on Proposition 3-2 above, the last value in BLU (15) is the insertion point and given two adjacent nodes: $\mathcal{N}_1$, with label $\mathcal{V}_1$ = 1.5.1, and $\mathcal{N}_2$, with label $\mathcal{V}_2$ = 1.5.2.

The Dewey and PoD representations for v_1 and v_2 are:

$\mathcal{V}_1$ = 1.5.1 (0001.0101.0001)

$\mathcal{V}_2$ = 1.5.2 (0001.0101.0010)

The label value ($\mathcal{V}$) of the new node $\mathcal{N}$ to be inserted between $\mathcal{N}_1$ and $\mathcal{N}_2$ is:

$$\mathcal{V} = \mathcal{V}_1.\underline{15}.1 = 1.5.1.\underline{15}.1$$

And the PoD representation is:

1.5.1.15-1(0001.0101.0001.<u>1111</u> 0001)

The insertion of the new node still preserves the document order since PoD maintains the labels in lexicographic order, which results in the property:

$$\mathcal{V}_1 < \mathcal{V} < \mathcal{V}_2$$

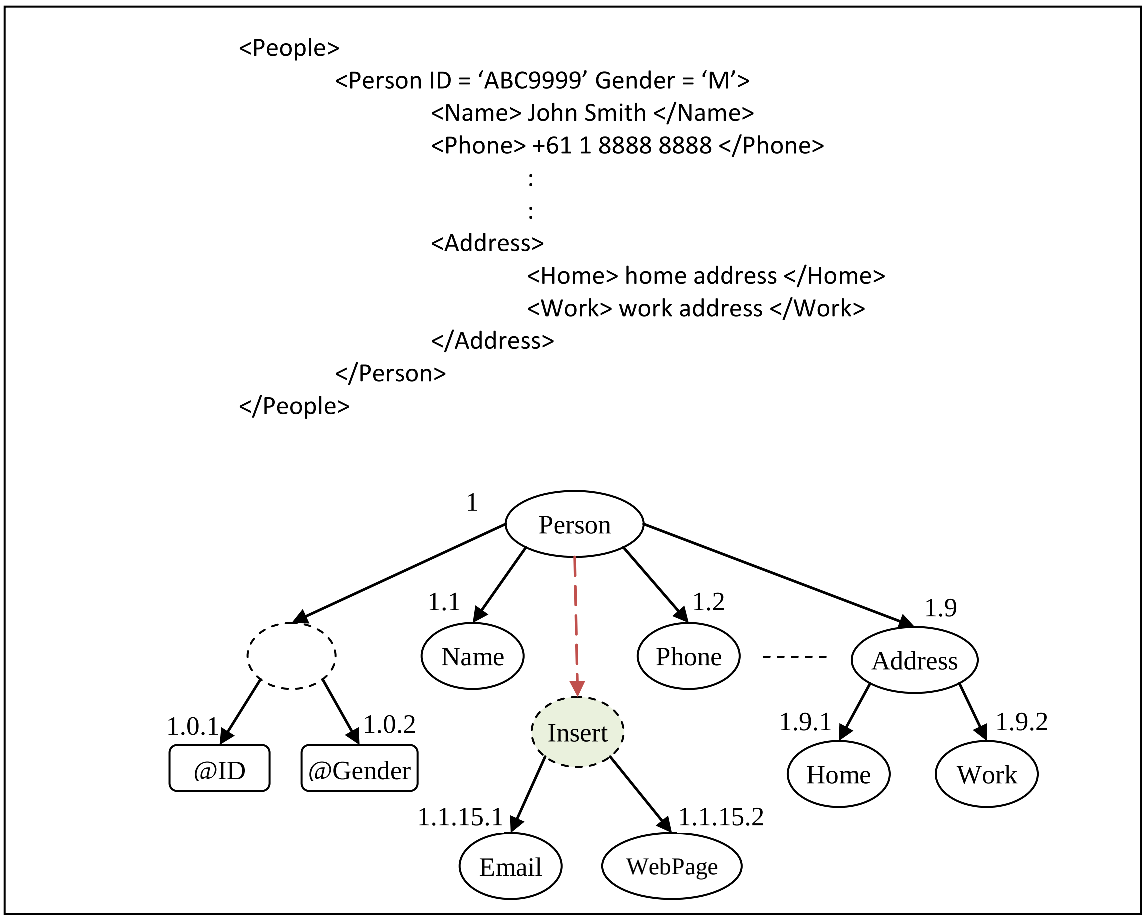
3.3.3 Features of the PoD Labelling Scheme

The PoD labelling scheme provides a size-efficient Dewey-based numbering technique. Further, PoD's compressed labelling technique still supports operations (such as inserting and updating) on dynamic XML documents.

To provide a better structure and more efficient access for attribute nodes in the PoD labelling scheme, the value 0 of any BLU is used to create a virtual parent node for the attributes of any XML node. This is because the attribute nodes are certainly leaf nodes and the label lengths of XML attributes do not have much effect on the total label size in most cases. One advantage of this technique is that it leaves the other values within the BLU to create short labels for element nodes, some of which might have children and grandchildren nodes.

Further, the PoD labelling scheme with its variable parameters ($\mathcal{L}$: length of the BLU, $\mathcal{X}$: the number of prefix values, and $\mathcal{L}_i$`: the length of the extended labels) can be configured to suit various document sizes and structures. This would achieve ultimate label lengths for applications in which data size is of concern, such as those applications that run on resource-limited devices (for example, hand-held devices). However, prior knowledge about the document's size and structure can help to establish a more efficient PoD with variant BLUs within the same document, either by analysing the document schema (i.e. DTD or XSchema) or by conducting double-phase parsing.

Double-phase scanning (parsing) can be used to analyse the document and establish proper BLU values based on the maximum and average fan-out values for each node's type in the document (the unique path, not the unique instance of each path). For example, if the first-phase parsing finds that the path (/Regions) has only six child nodes (the continents) and another path like (/Regions/Europe/Country) has 50 children, then we can assign (BLU=3) to /Regions and assign (BLU=5) to /Regions/Europe/Country, which would give a more efficient label size than using a single BLU for the whole document. The main advantage of this technique is to provide a smaller label size but the label structure would be more complex, and it would naturally take longer to parse for insertion operations. However, the double-parsing technique and multiple BLUs within the same document can be used for applications with static XML documents, which would help in reducing the label size further. Figure 3-4 shows a sample XML document with its XML tree representations in the PoD system.



```
<People>
        <Person ID = 'ABC9999' Gender = 'M'>
                <Name> John Smith </Name>
                <Phone> +61 1 8888 8888 </Phone>
                        :
                        :
                <Address>
                        <Home> home address </Home>
                        <Work> work address </Work>
                </Address>
        </Person>
</People>
```

Figure 3-4: PoD labelling for a sample people database.

The PoD labelling scheme also eliminates the need for parsing the node labels bit-by-bit, which cannot be avoided in other variable length prefix-free models. In PoD, any label-parsing function would identify a fixed-width block of bits equal to the width of the BLU and process it all together, and based on its value, it can easily determine whether this block is a label value or a prefix value and how many bits it should pick up on the next run.

The 4-bit BLU best represents the idea behind PoD since in most of the XML documents the fan-out for most of the nodes is small (less than ten). This means that most of the nodes within an XML document can be labelled without prefixing, which would produce shorter labels than other approaches. Further, with extended labels of lengths (such as 8, 12, 16 and 20), the structure of the resulting label will eliminate the need for parsing the label bit-by-bit to decode it; the processing units will be either a full byte or a half byte. Table 3-2 shows some suggested configurations for the PoD of a 4-bit BLU.

Table 3-2: Some suggested configurations for BLU of 4 bits.

Values for labels	Prefix values / Extended label lengths (bits)	Suitable for
{0-12}	(13/8) (14/12)	Small and medium fan-out
{0-10}	(11/8) (12/12) (13/16) (14/20)	large fan-out (general use)
{0-8}	(9/8) (10/12) (11/16) (12/20) (13/24) (14/32)	Very large fan-out

Note: the value (15), which is the maximum value in 4-bit BLU, is preserved to support new nodes insertion.

We have conducted extensive label size studies using different configurations of BLUs and using different XML benchmarks. The evaluation tests are reported in Section 4.

3.3.4 Two-component Dewey Labels

Dewey-based labels, including those of PoD, are stored as compact binary strings. To evaluate the different relationships that exist between XML nodes, such as parent-child and ancestor-descendent relationships, built-in and user-defined functions are required to process the Dewey-labels (for example, a function to find the parent label from a given context node's label). However, the use of functions may not make efficient use of the indexing techniques in some of the RDBMS systems. Moreover, some of the relational database systems do not support indexes on functions, as in the widely used database engine MySQL.

Splitting the Dewey label into two components (parent label and child label) would significantly improve the performance for queries that have the parent-child relationship or sibling relationship by building a B-tree or B^+-tree index on the two columns of (parent, child). Most XML queries in real applications involve either the parent-child relationship or sibling relationship. Further, evaluating parent-child and sibling types of queries will require an equijoin operation for these two relationships.

The new two-component labels are both in the Dewey format; a special concatenation algorithm is used to retrieve the original PoD label for any given node by stitching the two components together.

Proposition 3-3: *The PoD label for any given node $\mathcal{N}$ can be split into two components as follows:*

$$P_L(\mathcal{N}) \oplus C_L(\mathcal{N}) \tag{5}$$

Where: P_L is the parent's label of node $\mathcal{N}$.
C_L is the self label of node $\mathcal{N}$ (i.e. the local order of the node $\mathcal{N}$ in Dewey format).
$\oplus$ is the XML label concatenation operation.

While the above proposal can be applied to any binary-string Dewey label, the flexible structure of the PoD of 4-bit BLU makes it even easier than any other approach to split the Dewey label into two components: parent label and child label. This research project will focus on using PoD labels of 4-bit BLU; the following rules apply for a 4-bit PoD label:

- If the last byte of the parent label is a full byte, the child label will be the rest of the bytes in the original PoD label.
- If the last byte of the parent label is a half byte, the first byte of the child label will be left padded with $(0000)_b$.
- If the node is an attribute, the first byte of the child (self) component will be zeros $(0000\ 0000)_b$ and then the above rules apply.

Example 3-3: Let Node ($\mathcal{N}_1$) = 1.7, $\mathcal{N}_1$ be the parent node for an attribute node ($\mathcal{N}_2$) = 1.7.0.1, and another element node ($\mathcal{N}_3$) = 1.7.20, which represents the 20th child of ($\mathcal{N}_1$): Table 3-3 shows the PoD and the PoD-S representations for the three labels using the second BLU configuration in Table 3-2.

To retrieve the one-component Dewey label from a split label, we have implemented a user-defined function called XML CONCAT (XCONCAT). The XCONCAT function takes two parameters (parent, child) and returns one label in the Dewey format. Following is the concatenation algorithm, which is used in the XCONCAT function.

Table 3-3: PoD and PoD-S representations for the labels in Example 3-3.

Dewey Label	PoD Label	Pod-S Label	
		Parent Component	Child (Self) Component
1.7	0001.0111	0001	*0000* 0111
1.7.0.1	0001.0111.0000.0001	0001.0111	*0000 0000*.0000 0001
1.7.20	0001.0111.1011 0000 1010	0001.0111	*0000* 1011 0000 1010

Note: the zeros (*0000*) components in italic font are added because the smallest storage unit is a byte and they are not part of the label value.

Algorithm 3-1: Concatenating two component labels.

For any given node $\mathcal{N}$, let P_L and C_L be the parent and child (Self) label components respectively; the following algorithm is used to retrieve the original PoD label without any need to parse the P_L component:

```
Algorithm XML Label CONCAT:
Input ($P_L$PoD Dewey label, $C_L$PoD Dewey label)
Output: (PoD Dewey label)
      if ($C_L$[firstByte] >15)
      {
            Label = CONCAT ($P_L$, $C_L$)
      }
      else
      {
            $P_L$[lastByte] = $P_L$[lastByte] | $C_L$[firstByte]
             Label = CONCAT ($P_L$, $C_L$ [starting form firstByte+1])
      }
```

Where (|) is the bit OR operation, and CONCAT() is the built-in string concatenation function.

Note: the above algorithm is developed for the PoD system. However, the same logic can be applied to any Dewey-based labelling scheme.

3.3.5 New Nodes Insertion Is Still Supported in Pod-S Mode

PoD provides efficient support for operations on dynamic XML documents (such as insertion and deletion) without the need to relabel the existing nodes. In PoD, the maximum (last) value in any given BLU is used as an insertion point, as in Proposition 3-2. However, the insertion of new nodes without relabelling is still supported in Pod-S mode by extending formula (4) in Proposition 3-2 as follows:

Proposition 3-4: *Having a BLU with length ($\mathcal{L}$) and two consecutive nodes $\mathcal{N}_1$ and $\mathcal{N}_2$ with labels $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively, where $\mathcal{V}_2 > \mathcal{V}_1$, if a new node $\mathcal{N}$ is inserted between $\mathcal{N}_1$ and $\mathcal{N}_2$ then the label for node $\mathcal{N}$ is:*

$$\mathcal{V}_1.\,(2^{\mathcal{L}}-1).1 \qquad \textit{(PoD mode)} \qquad \textbf{(4)}$$

From formula (**5**) in Proposition 3-3: $\mathcal{V}_1$ can be represented as $P_L(\mathcal{V}_1) \oplus C_L(\mathcal{V}_1)$; and formula (**4**) can be re-written as:

$$P_L(\mathcal{V}_1) \oplus C_L(\mathcal{V}_1).\,(2^{\mathcal{L}}-1).1 \; \textit{(PoD-S mode)} \qquad \textbf{(6)}$$

Example 3-4: Based on formulas (**4**), (**5**) and (**6**) above, a node $\mathcal{N}_x$ can be inserted between two adjacent nodes $\mathcal{N}_1$ = 1.1 and $\mathcal{N}_2$ = 1.2 as follows:

$$\mathcal{N}_x = \mathcal{N}_1.\underline{15}.1 = 1.1.\underline{15}.1 = 1 \oplus 1.\underline{15}.1$$

Table 3-4: PoD and PoD-Split representation for labels: N_1, Nx, and N_2 in Example 3-4.

Dewey Label	PoD Label	Pod-S Label	
		Parent Component	Child (Self) Component
1.1	0001.0001	0001 *0000*	*0000* 0001
1.1.15.1 (inserted)	0001.0001.<u>1111</u> 0001	0001 *0000*	*0000* 0001.<u>1111</u> 0001
1.2	0001.0010	0001 *0000*	*0000* 0010

Note: the underlined components (<u>1111</u>) represent the insertion point. The italic (*0000*) components are padded since the smallest storage unit is one byte and to maintain the lexicographic order property of Dewey labels.

3.3.6 PoD Space Requirement Analysis

PoD delays the use of prefixes until the node order becomes high. This technique suits Dewey labels more, since the label of any given node (the context node) is constructed by concatenating the local order of all the nodes on the path from the root node to the context node. This means that reducing the label length of the nodes at the top of the tree will significantly reduce the total label size, since the label of the parent nodes at the top of the tree will be part of every node that descends from them.

For example, suppose that we have a BLU of 4 bits, which have the values 12, 13, and 14 reserved as prefixes for extended labels of length 8 bits, 12 bits, and 16 bits respectively. Using this BLU, we can use the values from 0 to 11 to label the first 12 child elements of any node on the XML tree before using the BLU as a prefix for extended labels. Based on the tree-like nature of XML documents and knowing that most XML documents in real applications have most of the nodes with fan-out values less than 10, the use of this labelling approach will substantially reduce the total label size for any given document.

Meanwhile, other techniques use labels composed of a prefix value and a labelling value, which will make the labelling component long, even for low ordinal nodes. For example, ORDPATH (a) requires a labelling component of a length of 5 bits to label the first nodes in the range [0,…,7]. However, ORDPATH reserves the even values for future node insertions, which brings the number of available labels to four. In addition, it has to go to the next level of labelling, 7-bit component, to be able to label another eight nodes. The result will be that PoD can concatenate the local order of two levels in one byte, whereas it requires two bytes to do the same in ORDPATH.

The numeric-based interval approach always produces a fixed length label since it uses three integer value components (star, end, level) and the length of any given label will 12 bytes. The storage requirements of the interval approach will be in most cases equivalent to those of the Dewey labels and it could be larger in many cases. However, the interval approach is a better option for very deeply nested XML documents.

3.3.7 User-defined Functions Used in PoD

As with every other Dewey-based labelling scheme, some functions are required to process the labels such as retrieving the parent node's label from the context node's label [O'Neil *et al.*, 2004; Tatarinov *et al.*, 2002].

However, we have found that implementing new functions can significantly improve the performance for some XML queries. Following are the functions we have implemented in the PoD system followed by a brief explanation of each function, in particular, the newly proposed functions.

Definition 3-1: *XML Parent Function (XP): the XP function takes one Dewey label as input and returns the parent node's label.*

This function removes the last component of the Dewey label and returns the rest of the label in the Dewey format.

Definition 3-2: *XML Child Component Function (XCC): the XCC function takes one Dewey label as input and returns the child (self) component of the label.*

This function returns the last component of the Dewey label by removing all the prefix components (ancestor components).

Definition 3-3: *XML Maximum Child Function (XMC): the XMC function takes one Dewey label as input and returns the upper limit value for all child and descendent nodes.*

In the PoD labelling scheme, the upper limit value is the result of concatenating the label and the value of the insertion point, $\mathcal{V}$.15

Definition 3-4: *XML Grandparent Function (XGP): the XGP function takes one Dewey label and the level of the ancestor node as input and returns the ancestor node's label.*

This function allows navigating upward the XML tree at minimal cost. It takes two parameters: the label of the context node and the level of the ancestor node, whose label we want to retrieve.

Example 3-5: Given a node $\mathcal{N}$ with label value = 1.2.3.4, applying the above functions would return the following results:

Parent of $\mathcal{N}$ = xp($\mathcal{N}$) = 1.2.3

Child (self) component of $\mathcal{N}$ = xcc($\mathcal{N}$) = 4

Maximum child label of $\mathcal{N}$ = xmc($\mathcal{N}$) = 1.2.3.4.15

The ancestor's label of $\mathcal{N}$ at level 2 = xgp($\mathcal{N}$, 2) = 1.2

Chapters 4 and 5 contain details about using these functions to develop more efficient query execution plans in relational database systems.

3.4 Evaluation Experiments

We have conducted extensive space experiments to evaluate PoD's space efficiency. We have also conducted one test to evaluate the effect of the label size on query performance.

3.4.1 Study Overview

The theoretical analysis shows that the PoD labels are expected to be smaller than the labels of other current Dewey-based labelling methods are. However, using benchmark documents to evaluate the space efficiency of any given labelling approach is the common method used in the research community, since benchmark documents are designed to test different aspects including the space efficiency by having different document sizes, different structures, different fan-out values and different document depths. We used XML documents that are commonly used for evaluating the label size. Table 3-5 provides more details about the documents that were used in this test.

Evaluating the label size normally should focus on evaluating three major values:

- The maximum label size: this value represents the length (in bytes) of the longest generated label in a given document; this value has an effect on the length of the index for individual records, which will affect the total index size.
- The average label size: the length (in bytes) of the average length of all labels in a given document. The average label length reflects the document structure.

- The total label size: the total size (in bytes) of all labels in a given document. This value represents the storage requirement of the all labels. This may be significant for XML database systems that run on limited-resource devices (such as hand-held devices) in which memory-space requirements may affect system stability.

The first test we conducted was to evaluate the parameterised features of PoD, which allow selecting an appropriate size for BLU, the number of extended labels, and their length. In addition, we conducted a space test using double-phase parsing; the double-phase parsing method allows analysing the document and allocating variable BLU sizes, which would achieve the ultimate label size.

Table 3-5: The features of some XML documents from different XML benchmarks.

Doc ID	Benchmark	Size	Depth	Total Number of Nodes	Source of Benchmark
D1	Merchant.xml (Shakes play)	183KB	6	4,152	[Bosak, 1999]
D2	SigmodRecords.xml	482KB	7	19,000	[UWDG, 2002]
D3	Mondial-3.0.xml	1.65MB	6	73,832	[UWDG, 2002]
D4	Movies	5.39MB	9	320,580	[UWMRG, 2002]
D5	Shakespeare plays (37 files)	7.5MB	6	180,253	[Bosak, 1999]
D6	Nasa.xml	24.6MB	8	562,200	[UWDG, 2002]
D7	Treebank.xml	82MB	36	2,437,667	[UWDG, 2002]
D8	dblp.xml	131MB	6	3,816,709	[UWDG, 2002]
D9	SwissProt.xml	109	5	5,166,890	[UWDG, 2002]
D10	XMark benchmark	10MB	12	234,161	[Schmidt *et al.*, 2002]
D11	XMark benchmark	100MB	12	2,328,505	[Schmidt *et al.*, 2002]
D12	XMark benchmark	500MB	12	11,675,206	[Schmidt *et al.*, 2002]
D13	XMark benchmark	1000MB	12	23,346,658	[Schmidt *et al.*, 2002]
D14	Protein Sequence Database	683MB	7	22,435,474	[UWDG, 2002]
D15	Michigan benchmark	50MB	18	603,019	[Runapongsa *et al.* 2003]
D16	Michigan benchmark	500MB	18	6,582,640	[Runapongsa *et al.* 2003]

3.4.2 Evaluating Different Configurations of PoD

3.4.2.1 Test Setup

The test aimed to evaluate the different configurations of PoD against small, medium and large XML documents; the documents all have different structures (i.e. depth and average fan-outs). However, PoD settings were configured without any prior knowledge of the document structures and sizes, as in Table 3-6.

Table 3-6: BLUs of different sizes and configurations, which are used in space evaluation test.

BLU Setup	Values for Labelling	Prefixing Value / Extended Label Length(bit)
BLU_3_S	{0,…,4}	(5/4) (6/7)
BLU_3_M	{0,…,3}	(4/5) (5,8) (6/10)
BLU_3_L	{0,…,2}	(3/6) (4/9) (5/12) (6/16)
BLU_4_VS	{0,…,12}	(13/6) (14/8)
BLU_4_VM	{0,…,11}	(12/6) (13/8) (14/12)
BLU_4_VL	{0,…,9}	(10/6) (11/9) (12/12) (13/16) (14/20)
BLU_4_S	{0,…13}	(14/8)
BLU_4_M	{0,…,12}	(13/8) (14/12)
BLU_4_L	{0,…10}	(11/8) (12/12) (13/16) (14/20)
BLU_5_S	{0,…,28}	(29/7) (30/9)
BLU_5_L	{0,…,24}	(25/7) (26/9) (27/12) (28/16) (29/20) (30/24)

Note: The maximum value of each BLU cannot be used for prefixing since it is reserved for supporting new node insertions.

The more useful length for BLU values are 3 bits, 4 bits, and 5 bits. A smaller BLU length (i.e. < 3) is too small to label documents without using extended labels most of the time. Larger BLU length (i.e. > 5) would waste space since the fan-out value for most of the nodes in most of XML documents is much less than 32. For each BLU, we had three setups: one for small fan-out values (with suffix _S), one for medium fan-out values (with suffix _M), and one for large fan-out values (with suffix _L). It was unnecessary to include a configuration for BLU_5_M because the BLU of 5 bits has enough values to use as prefixes for well-compressed extended labels.

For the 4-bit BLU test, we had two different configurations. The first one denoted as BLU_4V indicates a variable length for extended components (such as 6 bits or 10 bits). The other configuration is denoted with BLU_4, which has extended labels of the length of complete bytes (such as 8, 16 and 24), or half bytes (such as 12 and 20). We evaluated each group of setups (i.e. S, M and L) separately using appropriate XML documents of different sizes and structures.

3.4.2.2 Results and Discussion

BLU Small

The BLU_Small, as expected, successfully labelled three documents with small fan-out sizes, but it could not support the rest of the documents in Table 3-6. Tables 3-7 and 3-8 demonstrate the space test results for BLU_3_S, BLU_4_VS, BLU_4_S, and BLU_5_S.

The BLU of 3 bits achieved better label-size compression than the other two setups (4 bits and 5 bits) due to the very small fan-out values of these documents. However, BLU_3_S performed worse for D1 due to the relatively high average fan-out at two levels within the document. BLU_4_VS and BLU_4_S both performed better than BLU_5_S.

Table 3-7: The maximum label length for small BLU configurations using the evaluation documents from Table 3-5.

Doc ID	Maximum Label Length (Byte) for BLU Small			
	BLU_3_S	BLU_4_VS	BLU_4_S	BLU_5_S
D1	6	5	5	5
D2	6	7	7	7
D3	9	10	10	12

Table 3-8: The total label size for small BLU configurations using the evaluation documents from Table 3-5.

Doc ID	Total Label Size (KB) for BLU Small			
	BLU_3_S	BLU_4_VS	BLU_4_S	BLU_5_S
D1	14.87	14.31	14.33	15.68
D2	76.27	84.35	88.67	94.68
D3	4165	4981	4981	6375

BLU Medium

This test included medium fan-out size documents, which could not be labelled successfully using the small BLU configuration. The maximum label size and total label size results for five XML documents are shown in Tables 3-9 and 3-10 respectively.

Table 3-9: The maximum label length for medium BLU configurations using the evaluation documents from Table 3-5.

Doc ID	Maximum Label Length (Byte) for BLU Medium			
	BLU_3_M	BLU_4_VM	BLU_4_M	BLU_5_M
D4	6	6	7	7
D5	0	8	8	8
D6	7	6	7	7
D7	0	9	10	10
D8	0	8	8	8

Table 3-10: The total label size for medium BLU configurations using the evaluation documents from Table 3-5.

Doc ID	Total Label Size (MB) for BLU Medium			
	BLU_3_M	BLU_4_VM	BLU_4_M	BLU_5_M
D4	0.29	0.3	0.3	0.33
D5	0	1.4	1.39	1.6
D6	0.89	0.77	0.77	0.82
D7	0	1.05	1.05	1.21
D8	0	2.61	2.61	2.94

The 4-bit BLU was the winner and for both configurations (BLU_4_VM and BLU_4_M), the maximum label length was better for BLU_4_VM. However, the total label size results were almost equal for both configurations. The BLU_3_M failed to label three documents that could have been successfully labelled if we had longer extended labels (as the case in BLU_3_L); however, that would not have produced better results than the 4-bit BLU. The 5-bit BLU performed well in terms of the maximum label length; however, the total label sizes were larger than those of the 4-bit BLU for all documents.

BLU Large

This section of the test reported only the BLU_Large results for documents with large average fan-outs. Although the BLU large configurations supported all other smaller documents, we did not report the results for the smaller documents since they were covered by the small and medium BLUs, which had better compression results. The maximum label size and total label size results for six XML documents are shown in Tables 3-11 and 3-12 respectively.

Table 3-11: The maximum label length for large BLU configurations using evaluation documents from Table 3-5.

Doc ID	Maximum Label Length (Byte) for BLU Large			
	BLU_3_L	BLU_4_VL	BLU_4_L	BLU_5_L
D9	25	20	20	25
D10	0	7	7	8
D11	11	10	10	12
D12	7	8	8	8
D13	0	10	10	12
D14	0	7	8	8

Table 3-12: The total label sizes for large BLU configurations using evaluation documents from Table 3-5.

Doc ID	Total Label Size (MB) for BLU Large			
	BLU_3_L	BLU_4_VL	BLU_4_L	BLU_5_L
D9	15.10	14.29	14.29	17.02
D10	0	15.27	15.19	17.5
D11	13.66	11.71	11.64	13.38
D12	23.67	25.79	25.8	26.39
D13	0	61.72	61.23	70.2
D14	0	115.68	115.32	130.72

The 4-bit BLU achieved the best compression values, since the 4-bit BLUs suit the nature of the XML documents better when most of the nodes have small fan-out values and only a few nodes have large fan-out values. The small fan-out values can be labelled using the labelling values within the BLU. The BLU_3_M failed to label three documents, which could successfully have been labelled if we had longer extended labels; however, that would not produce better results than the 4-bit BLU. The 5-bit BLU did not perform well because these documents are deeper than the smaller nodes and the 5-bit components would accumulate into longer labels.

3.4.2.3 Brief Discussion

The PoD labelling technique, which is based on using a fixed-width basic labelling unit (BLU), has the capability to label any XML document of any structure and size. Based on the tests we reported in this section, the parameterised features of PoD labelling scheme can be used to achieve an optimal label length and size. However, the 4-bit BLU is the most appropriate setup for labelling XML documents, especially if the one-phase parsing technique is used and no prior information about the document structure is available. Further, the BLU_4, which uses extended labels of length with multiple bytes or half bytes, has space results that are almost identical to the BLU_4Vx. The BLU_4 has one advantage over the BLU_4Vx in that it makes the label structure simpler and easier to process, which reduces some of the overhead in the processing time.

The 5-bit BLU can achieve an efficient compression rate for shallow documents. However, the deeper the document is, the longer the label becomes and the space requirements become far larger than the space requirements for BLU_4.

3.4.3 Evaluating PoD Space Efficiency Using Double Parsing

Having prior knowledge about the document’s size and structure may help to select a more appropriate BLU setup. However, the double-parsing technique can also provide enough information about the XML document and can be used to select the most appropriate BLU setup. Further, different BLUs can be used within the same document, which can be allocated either by level or by node type.

We conducted the next test to compare the space requirements between the BLU_4 and the optimised BLU setup using the double parsing approach; we denoted this approach as BLU_D for dynamic allocation of BLUs. We only selected the BLU_4 because it is the most practical setup, which represents the idea behind the PoD approach.

3.4.3.1 Test Setup

We evaluated the two approaches using XML documents from two well-known benchmarks: the XMark (XM) and Michigan benchmark (Mich) [Runapongsa *et al.*, 2003; Schmidt *et al.*, 2002]; Table 3-5 contains detailed information about these documents. We used different BLU setups for the BLU_D; the BLUs were based on 3-bit, 4-bit and 5-bit BLUs.

3.4.3.2 Results and Discussion

The dynamic allocation of BLUs (BLU_D) reduced the space requirements for the PoD labelling scheme. Both the maximum label length and total label size dropped by approximately 20 per cent. Figures 3-5 illustrates the maximum label length and total label size for BLU_D and BLU_4.

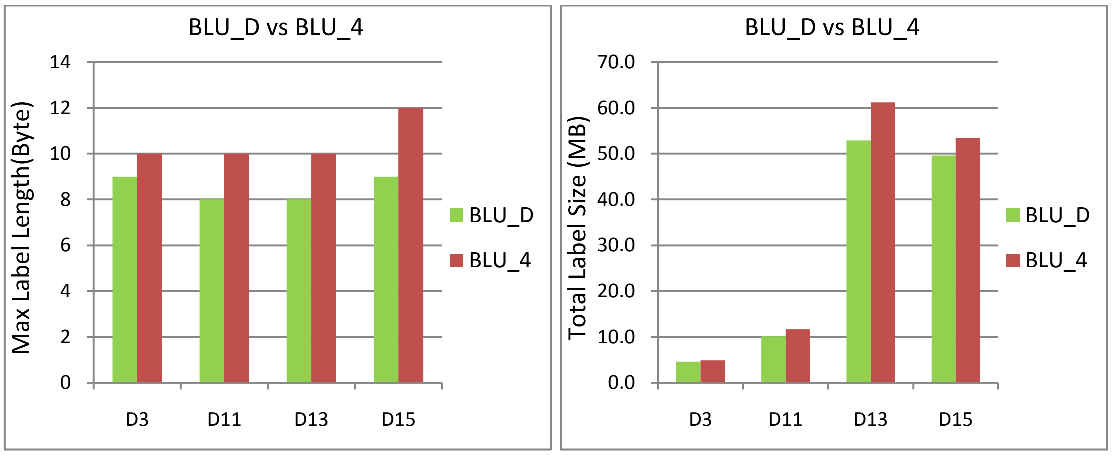


Figure 3-5: Space requirements comparison between BLU_D and BLU_4.

Given that BLU_4 already provides smaller labels compared with other approaches in the literature, the complexity and the cost of the document double-parsing operation makes this option less favourable than the BLU_4. This is because processing labels to evaluate relationships between document nodes would become an expensive operation. However, the BLU_D can still be useful for applications in which space is more important than other factors, in particular, for static XML documents.

3.4.4 Comparing PoD-4 with Other Dewey Labelling Approaches

We conducted a similar test for space requirements to compare the PoD of 4-bit BLU with other recent Dewey-based labelling techniques. We only picked one configuration of PoD based on the BLU of 4-bit length, which was based on the space analysis in Section 3.9 and the test results discussed in the previous subsections.

We compared PoD, which is a Dewey-based labelling scheme, to other Dewey-based labelling schemes. We did not include the numeric-based range encoding because its space requirements are fixed and well known; it requires three integer numbers at least, excluding the document id, to label each node (start, end and level), which means that the total label length is 12 bytes (given that each integer value is a 4-byte length).

3.4.4.1 Test Setup

We conducted this test using most of the documents in Table 3-5 to ensure diversity in document sizes and structures. We mainly focused on medium, large and very large XML documents.

We implemented two well-known Dewey-based labelling schemes: the ORDPATH, with its two configurations (a) and (b) [O'Neil *et al.*, 2004]; and the QED labelling technique [Li and Ling, 2005] using a double-parsing process to produce an optimal label length.

3.4.4.2 Results and Discussion

The results show that PoD-4 outperformed the other two techniques for all documents in this test. PoD-4 reduced the space requirements for Dewey labels despite the document's size and structure, which provides a consistent labelling technique that can guarantee short labels. Figures 3-6 and 3-7 illustrate the maximum label length and total label size respectively for PoD-4, ORDPATH and QED.

Meanwhile, QED performed better than ORDPATH except for XML documents from the Michigan benchmark, which indicates that QED compression techniques are insufficient for deep documents with small fan-out values. QED performed well for documents with a low ratio between the maximum fan-out and the average fan-out.

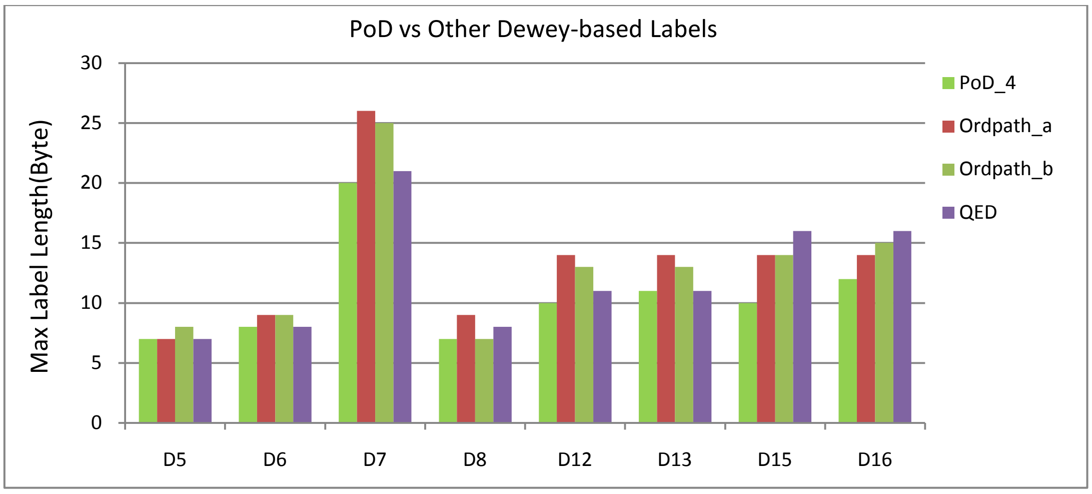


Figure 3-6: Maximum label length comparison between PoD and two other Dewey-based labelling techniques (Ordpath and QED).

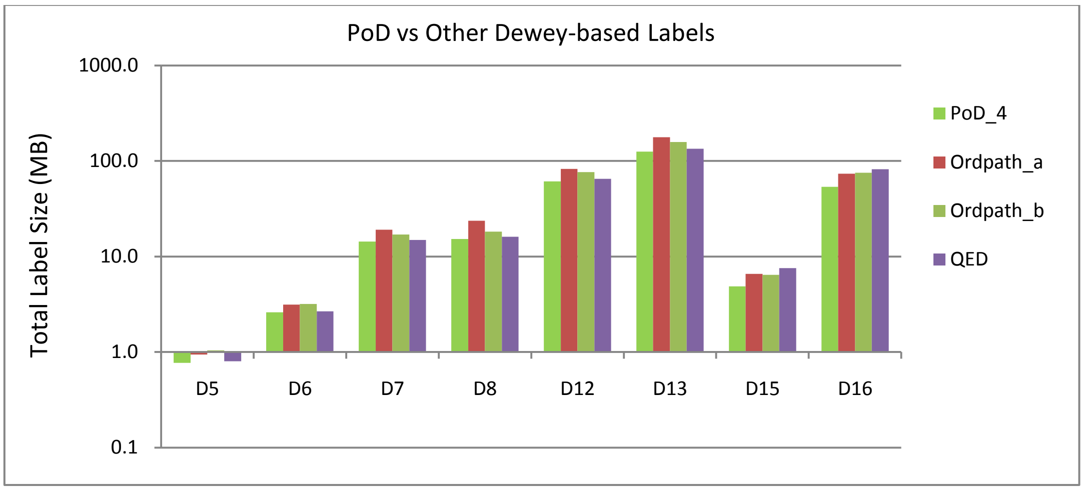


Figure 3-7: Total label size comparison in logarithmic scale between PoD and two other Dewey-based labelling techniques (Ordpath and QED) using documents from Table 3-5.

For documents D5 and D6, PoD and QED had the same maximum label length value; however, the total label size for PoD was slightly smaller for both documents.

The ORDPATH labels required more space due to the use of longer prefix values. Moreover, ORDPATH skips the even values and reserves them to support the insertion of new nodes, which requires using a longer labelling-range most of the time.

3.4.5 Evaluating Space Requirements for PoD-Split

To exploit current indexing technologies in off-the-shelf database systems, we propose to split any given Dewey label into two Dewey-based components: Pid and Cid.

However, this new structure may slightly increase the storage requirements for Dewey identifiers.

This test evaluated the cost of splitting the Dewey label into two components. We conducted the test using our PoD-4 labelling scheme. We did not run the test for other labelling schemes because the same concept applies and it is much easier to implement the split concept using PoD-4.

3.4.5.1 Test Setup

We implemented the PoD of BLU-4 with extended labels of length (for example, 8, 12 and 16). We also implemented a split mode based on the same BLU setup, which we called PoD-S. We conducted the space test using the documents from Table 3-5. In this test, we wanted to measure the total label size increase in split mode.

3.4.5.2 Results and Discussion

The results showed that there is an increase in the space requirements for the PoD-S mode. However, the increase was approximately nine per cent for some (D6, D7, D12, D13 and D15) and up to approximately 20 per cent for others (D5, D8 and D16). This depends on the way in which we store the split components; if the last byte of the parent component is a half-byte, the child component will be right-shifted by a half byte, which may acquire an extra byte. Figure 3-8 shows the increase in total label size using the split mode.

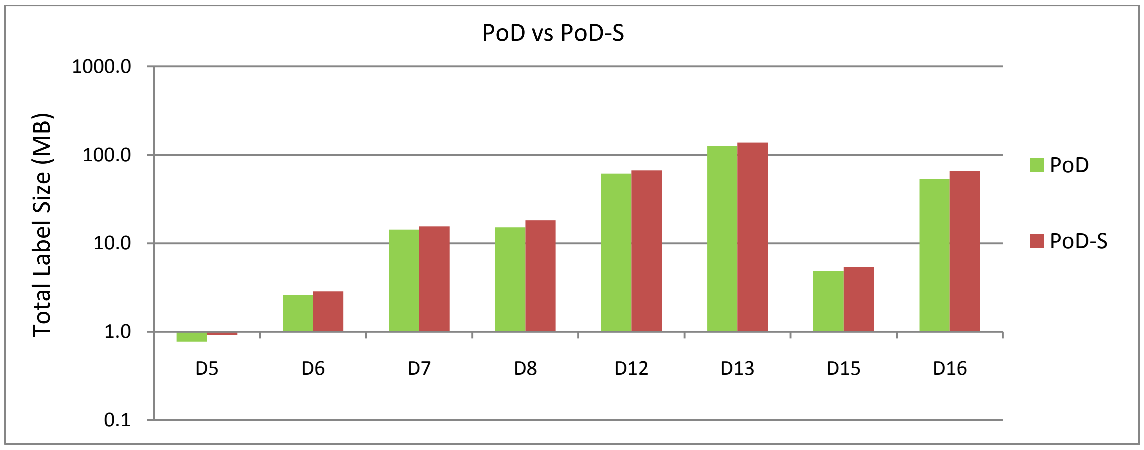


Figure 3-8: Total label size comparison in logarithmic scale between PoD and PoD-S (Split mode) using the documents from Table 3-6.

The advantages of having a split-label, which would significantly improve query performance, might justify this increase in the space requirements, bearing in mind that PoD is a very compact labelling scheme.

3.4.6 Evaluating the Effect of Inserting New Nodes

PoD supports operations on dynamic XML documents, such as inserting new nodes without relabelling any of the existing nodes. In this test, we evaluated PoD and PoD-S capabilities of supporting operations such as insert and delete, and compared the results with those from the ORDPATH technique.

3.4.6.1 Test Setup

The Michigan benchmark provides update queries to evaluate the efficiency of a labelling scheme in handling insert and delete operations. We conducted the test using PoD and ORDPATH_a. We used two different sized documents from the same benchmark. Table 3-13 shows the update queries.

Table 3-13: Details of some update queries from the Michigan benchmark.

Query ID	Query Text	Comments
QU1	Insert a new node below the node with aUnique1 = 10102	Point Insert Operation: will insert a complete eNest node with its attributes
QU2	Delete the node with aUnique1 = 10102 and transfer all its children to its parent	Point Delete Operation: the transferred nodes will be inserted in the new location
QU3	Insert a new node below each node with aSixtyFour = 1. Each new node has attributes identical to its parent, except for aUnique1, which is set to some new large, unique value, not necessarily contiguous with the values already assigned in the database	Bulk Insert Operation.

3.4.6.2 Results and Discussion

Table 3-14 shows the statistics information of the changes that were caused by running each update query. In total, the three update queries inserted 9441 nodes in Doc1

(50MB) and 102330 nodes in Doc2 (500MB). The update queries slightly changed the maximum label size and the total label size for the three labelling schemes PoD, PoD-S and ORDPATH_a. Table 3-15 shows the changes of the maximum label size and the total label size after running the three update queries.

Table 3-14: The number of nodes that were added and deleted after executing each update query in Table 3-13.

Query ID	Inserted Nods		Deleted Nodes		Relabelled because of transfer	
	Doc1(50MB)	Doc2(500MB)	Doc1(50MB)	Doc2(500MB)	Doc1(50MB)	Doc2(500MB)
QU1	9	9	0	0	0	0
QU2	0	0	9	9	54	0*
QU3	9378	102321	0	0	0	0

(*): no nodes were transferred in Doc2 since the deleted node did not have any element child nodes.

Table 3-15: Label size changes after executing update queries in Table 3-13.

Labelling Scheme	Doc1 max label length (byte)			Doc2 max label length (byte)			Doc1 total label size (MB)			(%) change of total label size		
	before	after	%	before	after	%	before	after	%	before	after	%
POD	10	11	10	12	13	7.7	4.86	4.95	1.86	53.47	54.44	1.81
POD-S	11	12	9	14	15	7.2	5.37	5.47	1.86	65.89	67.09	1.82
ORDPATH_a	14	14	0	14	15	7.2	6.55	6.66	1.72	73.26	74.52	1.72

The results show that PoD and PoD-S can support operations on dynamic XML documents without the need to relabel the existing nodes. However, there was a slight increase in both the maximum label length and the total label size. The effect on ORDPATH_a was less than PoD since ORDPATH sacrifices the initial space requirements to support efficient insertion between every two adjacent nodes.

3.4.7 Evaluating the Effect of Label Size on the Query Performance

Reducing the label size might also improve the query performance since it will increase the amount of data that can be cached in the main memory, especially on resource-limited devices. We conducted this test to evaluate the effect of the label size on the query performance by running the same query using different Dewey-based labelling schemes. However, better query optimisation techniques have a major role in improving query performance, which will be discussed in the following chapters.

3.4.7.1 Test Setup

We implemented PoD-4 (PoD), QED, and ORDAPTH (a) and (b). MySQL database engine v5.1 was used as a backend relational storage [MySQL, 2009]. The test was conducted on an Intel Duel Core (1.8GH) machine with 1GB of memory. We used 16 different queries from the Michigan benchmark; we selected queries from each group based on its relevance to the purpose of this test. We used the benchmark tools to generate two XML documents of different sizes: 50MB and 500MB. We conducted the same set of queries 10 times per query against both documents; we ignored the first run time for each query. More details about the Michigan benchmark queries can be found in Appendix A-2.

3.4.7.2 Results and Discussion

The results in Figures 3-9 and 3-10 demonstrate that PoD can be used effectively to address almost all queries against XML documents that are stored in an off-the-shelf relational database system. Further, for almost all queries, PoD ran consistently faster than the ORDPATH and QED labelling scheme by around 10 per cent; this is due to the fact that PoD has shorter label lengths than ORDPATH and QED by around 20 per cent and 35 per cent respectively.

The 10% performance gain is more obvious using the Michigan benchmark’s second document Mich2 (500MB). Since the size of the labels became very large, this also determined the size of the primary index. At this level, space reduction becomes more valuable.

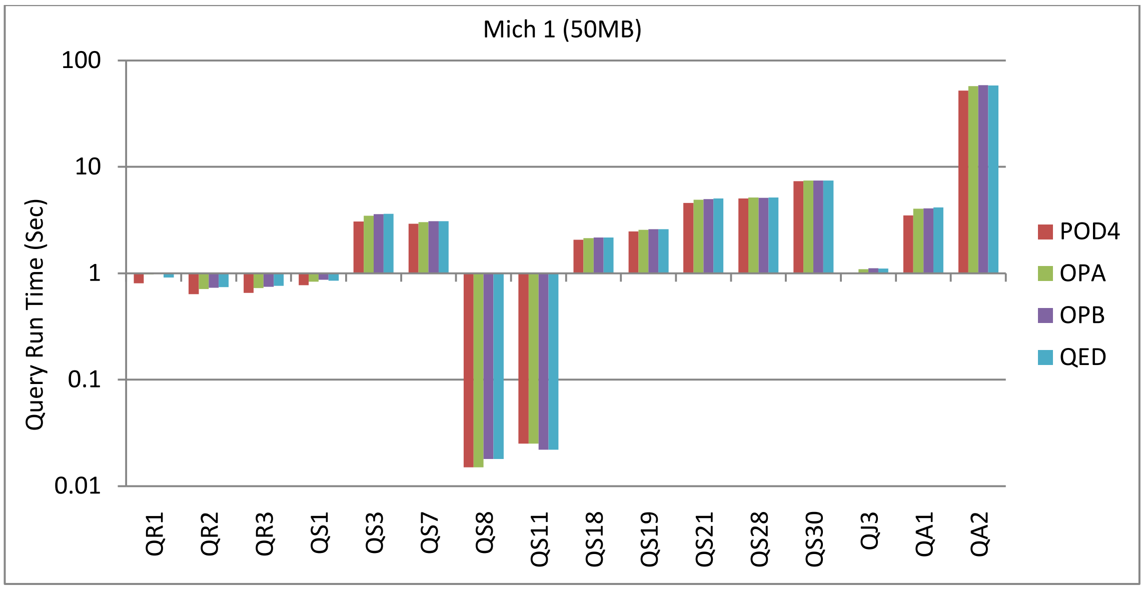


Figure 3-9: Queries' run time in logarithmic scale for 16 queries from the Michigan benchmark evaluated using different Dewey-based labelling schemes.

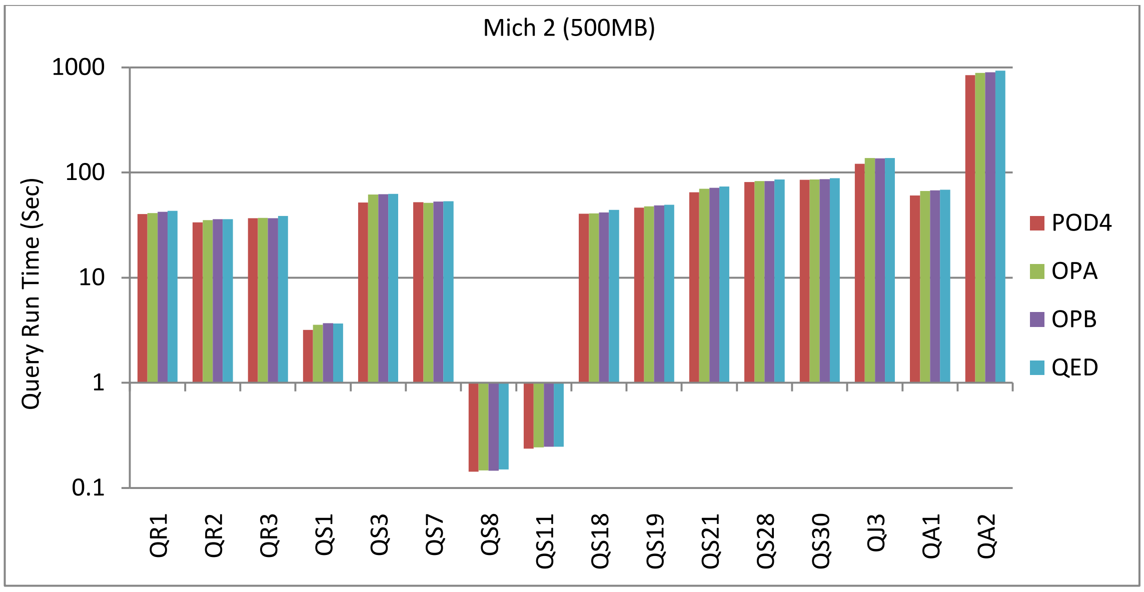


Figure 3-10: Queries' run time in logarithmic scale for 16 queries from the Michigan benchmark evaluated using different Dewey-based labelling schemes.

3.5 Conclusion

The PoD approach provides an efficient, flexible and compact labelling technique. PoD is a Dewey-based labelling technique that supports operations on dynamic XML documents (such as insert and delete) without relabelling the existing nodes. Further, PoD with its parameterised features can be tuned to provide ultimate compression ratio for label lengths by using double-parsing processing and dynamic BLU allocation.

However, using complex techniques to reduce the label size further can add processing overhead. We recommend PoD-Dynamic only for applications in which space is expensive and may affect the system performance, such as hand-held devices.

The space evaluation tests showed that the PoD of 4-bit BLU has outperformed recent Dewey-based labelling approaches by reducing the label size by 20 per cent. Moreover, PoD provides a consistent compression technique for almost all types of XML documents. Meanwhile, other approaches only work well with a certain type and structure of XML documents.

PoD, with its fixed-width basic labelling unit (BLU), simplifies the label structure and removes processing overhead because it eliminates the need to parse any generated label bit-by-bit. Labels can be processed by blocks of bits, especially in the BLU of four.

To take full advantage of the current indexing technologies that are available in off-the-shelf relational database systems, we have introduced a new structure for Dewey-based labels. The new structure is based on splitting any given Dewey id into two components: Pid and Cid. This structure would significantly enhance query performance for queries that are based on parent-child and sibling relationships. The new split mode can be applied to any Dewey-based labelling scheme. However, it is much easier to apply it to PoD-4 due to its simplified structure.

PoD-S mode requires more space than PoD. However, the performance improvement that comes with this approach may justify this slight label size increase (approximately nine per cent on average).

Minimising the Dewey label size and simplifying its structure are major steps towards more efficient storage and processing of XML documents in relational-based systems; the PoD system also integrates the document schema summary, which we capture in the XML_Path table, with XML query processing to provide alternative query optimisation techniques for Dewey-based labelling schemes. The next two chapters will provide detailed discussions about our query optimisation techniques in the PoD system.

Chapter 4: Navigating the XML Tree Using the PoD Approach

XML language has provided a powerful platform for both business-to-business and business-to-user Internet applications. However, XML introduces trade-offs and complications, such as the need to efficiently retrieve information stored in XML documents. The W3C has approved XQuery and its subset XPath as standard query languages for XML data. In this chapter, we introduce the PoD approach to evaluate all XPath axis steps, which are the basic building unit of any XML query, and provide alternative new techniques for efficiently querying XML data. While these techniques are based on PoD, they can be easily applied to other Dewey-based labelling schemes.

4.1 Introduction

Various XML query languages have been proposed, most associated with certain proposed storage systems. Recently XQuery/XPath language has emerged as the standard query language for XML data [Fernandez *et al.*, 2007]. XQuery is a strongly typed functional language that supports both the transformation and querying of XML documents.

XQuery uses XPath syntax for locating paths within the XML documents. The XPath syntax is based on navigating the document using the XML tree model; XPath uses expressions that are composed of a sequence of location steps separated by the '/' symbol. Evaluating these different types of axis steps is a challenging task, especially for XML data stored in relational-based systems. Label ids are used to capture the document order and relationships among the document nodes.

XQuery is relatively new and many issues with respect to implementation and optimisation remain unresolved. In most of the off-the-shelf and widely used open-source relational databases, XQuery and XPath queries are translated into equivalent SQL statements before they are run against shredded XML documents.

Several studies in the literature compare Dewey-based labels to other labelling methods for storage requirements and general performance evaluation. We have not discovered any detailed studies for optimising XPath axis steps based on Dewey labels because most of these studies have used the current techniques that are applied for number-based range encoding.

In the following sections, we discuss XML query-related issues, implementations and optimisation techniques. Further, we introduce alternative query optimisation techniques based on Dewey-labels using off-the-shelf relational databases. We report on extensive experimental evaluation studies using well-known XML benchmarks.

4.2 XML Query Optimisation Objectives

XML and XQuery data models provide great support for semi-structured data, and provide flexibility for data integration and data-exchange applications. However, the new data model and XQuery semantic definitions, as set by the W3C, have introduced complexity and challenges to an efficient implementation of XQuery. For example, the XQuery and XPath semantics require that each intermediate step in a path expression returns its results in the document order, which is not originally part of the relational systems due to the mismatch between the two data models.

Recent works have focused on developing techniques for XML query optimisation and efficient implementation. We can summarise optimisation objectives as follows:

- Eliminate duplicate removal and sorting operations when they are unnecessary; sorting and removing duplicates after each step is an expensive operation and slows down system performance. For example, a singleton result is always sorted and duplicate free, which means there is no need to apply sort and duplicate removal operators [Al-Khalifa *et al.*, 2002; Bao *et al.*, 2008; Fernandez *et al.*, 2005; Grust *et al.*, 2003; Lu *et al.*, 2005].
- Developing XQuery Algebra based on XQuery formal semantic and core mapping as defined by W3C. The presence of XQuery Algebra makes it easier to optimise join operators [Draper *et al.*, 2007; Lee, 2003; Michiels *et al.*, 2007; Pal *et al.*, 2005b; Re *et al.*, 2006].
- Developing appropriate indexing rules and/or new indexing techniques that better suit the XML data model. Some of indexing techniques are necessary to maintain the XML document order and structure (i.e. parent-child and ancestor-descendent relationships) [Florescu and Kossmann, 1999; McHugh *et al.*, 1997; McHugh *et al.*,

1998; O'Neil *et al.*, 2004; Pal *et al.*, 2004; Shui *et al.*, 2005; Tatarinov *et al.*, 2002; Yoshikawa *et al.*, 2001]. Other indexing techniques are proposed to improve queries that use tag names, which are parts of almost every XQuery and XPath query statement [Kaushik *et al.*, 2002; Lu *et al.*, 2005; Pal *et al.*, 2005a].

- Make better use of data schema (when it exists) to further optimise XML query performance. Data schema is helpful in optimising certain types of queries. Further, data schema can be used to answer some queries without going back to the actual data, given that the data schema is much smaller than the data instance [Cooper *et al.*, 2001; Goldman and Widom, 1997; Grinev, 2002; Lu *et al.*, 2005].

4.3 XML Query in Relational Database Systems

Supporting XML data in relational database systems has become more popular due to the reasons mentioned in previous chapters. With the rapid increase in the use of XML documents over the Internet and in many other applications, the challenge of efficiently supporting querying XML data in relational systems has increased. One of the major challenges is retrieving the full XML document or large portions of it. The industry has responded to this challenge by introducing a new data type called XML, which is similar to the character large object (CLOB) data type. The new XML data type can store an XML document in its native format, which eliminates the processing overhead for reassembling the document [Beyer *et al.*, 2006; Liu *et al.*, 2005; Pal *et al.*, 2005b].

The rise of XQuery as a standard query language for XML data has also added new challenges to the relational-based systems. XQuery with its FLWOR statements allows more complex queries against the structure and data within the XML documents. To avoid scanning the stored XML document during run time, relational-based systems make use of the existing relational techniques by capturing the document order and structure in special indexes that are used to answer queries without the need to parse the original document. Moreover, the major vendors of database systems are providing an option to shred the XML document into relations to be used for data-centric queries.

When XML is stored in relational systems (shredded and/or indexed) the labelling technique will play a major role in determining the way the XML query will be implemented or translated into SQL statements. It will also be used in the query performance.

4.3.1 Labelling Techniques Effect

As was mentioned in the previous chapters, there are two main labelling techniques reported in the literature that are used to capture document order and relationships between document nodes (i.e. parent-child and ancestor-descendent relationships). These two techniques are:

- The numeric intervals or ranges encoding.
- The binary string Dewey-based encoding.

The intervals approach has demonstrated competitive performance compared with native approaches due to the fact it uses operations on numbers [Amagasa and Yoshikawa, 2003; Jagadish *et al.*, 2002; Kha *et al.*, 2001; Yoshikawa *et al.*, 2001; Zhang *et al.*, 2001]. Based on the range encoding and the popular (pre, post, level) label, a node 'A' is an ancestor of node 'B' if:

```
(A.pre < B.pre) and (A.post > B.post)
```

And B is a child of A if:

```
(A.pre < B.pre) and (A.post > B.post) and (A.Level +1 = B.Level)
```

However, the theta-join (θ-join) (which is based on '>' and '<' comparisons) is considered slower than the equijoin (i.e. using '=' operator); the interval approach does not provide alternatives to make use of the equijoin, which runs faster with the presence of the BTree indexes. Further, the interval approach cannot efficiently support operations on dynamic XML documents (such as inserting new nodes), which will trigger an expensive node-relabelling operation. Further, some interval approaches require storing additional fields (such as the node's level and parent label) to support additional XPath access steps (like parent-child). This makes the size of the label structure larger than the size of Dewey-based labels.

While the interval approaches have been studied extensively in the literature and optimised to support all XPath path expressions and axes steps, there are no similar detailed studies for Dewey-based labels. In addition, a widely used approach is to use the same range's concept to evaluate the XPath axis steps [Abiteboul *et al.*, 2001; Böhme and Rahm, 2004; O'Neil *et al.*, 2004; Tatarinov *et al.*, 2002].

4.3.2 XML Schema Information Effect

An XML document does not require content to adhere to structural rules. However, to facilitate data processing, some applications might require XML documents to conform to certain structures and data definitions, known as document schema (such as DTD). The existence of XML can be used to perform some optimisations on path expressions by trying to eliminate impossible path expressions (i.e. expressions that are known to be always empty).

Labelling a document's nodes can maintain the XML document's order and some of its structure. However, the node-labelling technique alone still does not provide an efficient and complete answer to all types of XML queries. Recently, the importance of capturing the document schema summary has increased because it can be used in XML query optimisation, path validations and document presentations [Moro *et al.*, 2008]. The most popular usage of schema information is as secondary indexes that can identify XML nodes reachable from specific path patterns, because the schema information is captured in a much smaller representation than the actual data instance [Goldman and Widom, 1997; Pal *et al.*, 2005a].

Recent proposals [Bao *et al.*, 2008; Kwong and Gertz, 2002; Michiels, 2003] have expanded the usage of XML data schema information to include:

- Browsing data structure
- Storing information, such as statistics and sample values
- Enabling XML query optimisation:
    - Removing redundant conditions
    - Simplifying conditions
    - Detecting contradictory conditions and satisfyability

The schema information, such as data types (i.e. integer and string) and node types (i.e. element and attribute) can be used in native XML management systems to avoid scanning the whole tree. In addition, the schema information or summaries can be used to minimise the number of joins in the relational management systems.

In this chapter, we introduce new optimisation and query evaluation techniques for XPath axis steps based on a combination of document schema summary, captured in the XML_Path table, and Dewey-based labels.

4.4 XML Schema Summary in the PoD System

An XML document is in a self-described data format [Nambiar *et al.*, 2002]; XML tags describe the data itself, and their locations in the document declare containment and structural relationships. An XML document does not require content to adhere to structural rules. However, to facilitate data processing, some applications might require XML documents to confirm to certain structures and data definitions, known as document schema (such as DTD and XSD). The existence of document schema can lead to more efficient mapping to relational schema, as mentioned in Section 2.2.2. Further, XML document schema can be used in advanced query optimisation techniques. Some XML queries can be answered or validated by accessing the XML schema without any need to access the actual document itself or its contents.

However, in many other XML-based applications, the XML document schema might be absent or not required, which limits the capabilities of the data management systems (either native or relational) to answer some XML queries in an efficient manner. Further, storing XML documents in relational systems can result in losing the document structure and schema information.

This section will focus on developing techniques to capture XML schema information in the absence of DTD or XSD. Further, the captured schema information can be used to provide alternative execution plans for XML queries in relational database systems with the presence of a Dewey labelling scheme such as the PoD system.

4.4.1 Capturing XML Document Schema Summary

The schema summary can be captured when the document is initially parsed and stored in the relational system; the schema summary can be represented in memory using a hierarchical data structure or it can be stored as a table(s) in the relational system. Since the schema summaries are typically very small, we propose having two versions of the schema summary. First, an in-memory hierarchical structure that can be integrated and used by middleware applications, such as during parsing documents and storing them in RDBMS, or during XML query translation. Second, a table structure that can be used during query executions and in some cases can be used to answer certain groups of XML queries. Figure 4-1 shows the suggested schema summary in the PoD system.

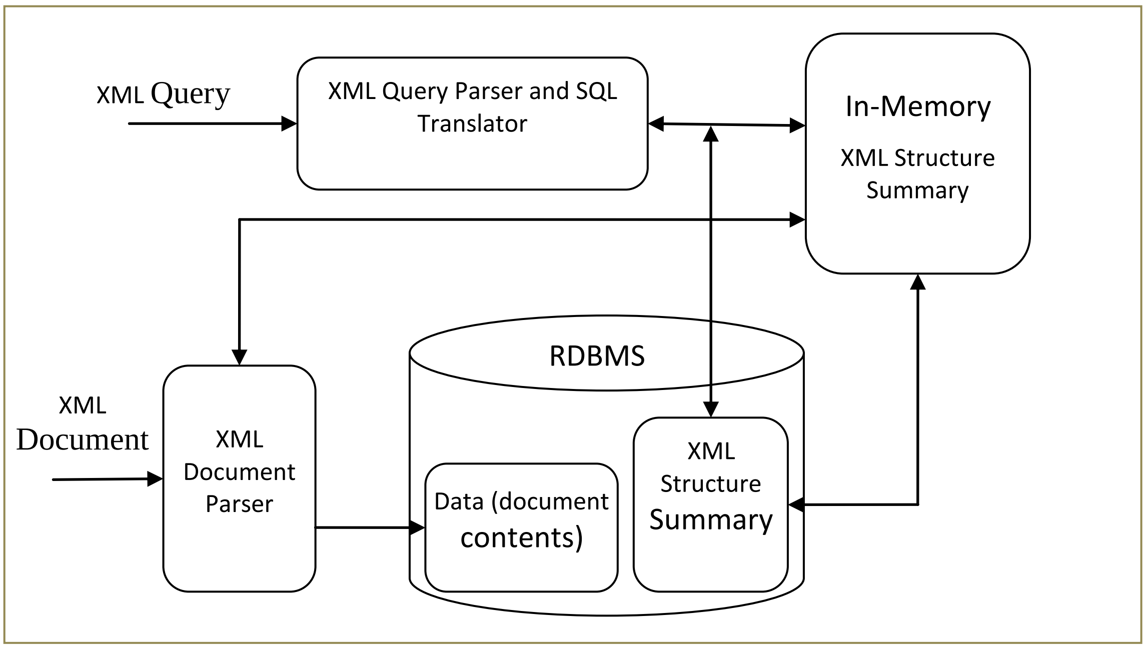


Figure 4-1: Block diagram for XML-schema summary in the PoD System.

Our schema summary (or data guide) captures the document structure by capturing all unique path expressions in the document. Further, we store information about each node name in the document (not node instances), such as node type and the node's level on the XML tree.

The following section defines the items that are used to construct the document structure summary, as in our 'XML_Path' table.

Definition 4-1: *Path Expression: a sequence of concatenated tag names from the root to the context node, separated by '/', which matches the XPath representation.*

For example, the path expression for node 'Name' in Figure 3-4 is:

'/People/Person/Name'

Definition 4-2: *Path id: a unique integer value corresponding to every unique path expression.*

Definition 4-3: *Node Type: the node type, as in the W3C recommendations for the XML Data Model [Bray et al., 2008]; an integer value (such as element or attribute) is used to represent each data type.*

The node type for the node ‘Name’ is an element that is represented by ‘0’.

Recent versions of XML parsers, such as Apache Xerces XML parsers [Apache.org, 2005], comply with W3C recommendations for XML; these parsers can report and handle different types of nodes, such as document, element, processing instruction and text.

Definition 4-4: *Node Level: the node’s level value on the XML data tree with ascending values starting from the root node at level 0.*

Example: the node ‘Name’ in Figure 3-4 is at level 2.

Definition 4-5: *Node Occurrence: statistics information representing the number of instances of one particular node (i.e. path expression) in the actual database.*

This accumulated number can be helpful to generate optimised query execution plans, as will be demonstrated in Chapter 5.

4.4.1.1 In-memory Schema Summary

The in-memory schema summary is a tree structure rooted at a virtual root node. Each XML node is represented with an object, called XNode. The main features of the XNode object are:

- An Object Id: equivalent to the Path id
- Object Name: represents the XML Tag name
- Object Type: represents the node type (such as element and attribute)
- A Pointer to its Parent Object: allows bottom-up traversal
- Occurrence: an integer value representing the actual number of instances of this node in the database
- getPathExp() Function: returns the path expression by traversing up the tree and concatenates the tag name of each object on the path to the root node

The XNode object is used to represent all XML node types except the element node type, which is represented by an extended object from the XNode object.

The element object contains the following extra sub-objects:

- Child Elements: a hash table containing all the node's child elements with 'tag name' as a key to allow searching the collection by a tag name.
- Attributes: a hash table containing all the node's attribute nodes, with 'attribute name' as a key to allow searching the collection by an attribute name.
- Sequence: tracks the number of children that will be used to generate the Dewey labels.

Although the memory size of this approach depends on the number of distinguished paths in the document, the size is still relatively small since the average number of paths in real XML documents is normally a few hundred.

The advantage of this approach over the stack approach is that it eliminates creating and destroying objects during document parsing, which is considered an expensive process. Further, the traversing pointer for this data structure will be moving in the same direction as in the serial SAX XML parser.

4.4.1.2 XML Schema Summary in Table Format

To facilitate using the captured schema summary, we store the captured XML schema summary in the relational system as a table, which we call 'XML_PATH', with the following definitions:

```
XML_PATH = {Path_id: Integer, PathExp: String, Level: Integer,
Node_Type: Integer, Occurrence: Integer}
```

The path_id column is a primary key for this relation and is used as a foreign key in other relations that contain the actual data instances.

In this chapter and Chapter 5, we propose new optimisation techniques for querying XML data in relational systems based on the existence of the XML_Path table and combined with a Dewey-based labelling scheme like PoD and PoD-S.

4.5 PoD Optimisation Approach

In this section, we will exploit the combinations of the Pod-S and the XML_Path table to expand the capabilities of the relational systems to support XML queries efficiently without modifying the relational systems kernel. The PoD optimisation approach is based on shorter labels, split Dewey-labels (i.e. two-component labels), and exploiting the XML schema summary captured in XML_Path table. Figure 4-2 shows a sample XML document, which will be used as an example to demonstrate and explain our techniques.

```
<Booklist>
  <Book>
    <Title> XML: the Complete Reference </Title>
    <Year> 2005 </Year>
    <Authors>
      <Author>
        <Name> Heather Williamson </Name>
        <Address>
          <City> New York </city>
          <zip> 99999 </zip>
          <Contacts>
            <Email>name@net-address</Email>
          </Contacts>
        </Address>
      </Author>
    </Authors>
    <Section>
      <Title> section title </Title>
      <Section>
       :
      </Section>
       :
    </Section>
  </Book>
   :
</Booklist>
```

Figure 4-2: A sample XML document.

Figure 4-3 illustrates the relational schema for the major approaches that will be used in the evaluation tests for easy comparison. We only mentioned the Element_Table because the same techniques can be applied to the Atrribute_Table and any other similar tables containing document nodes.

Element_Table (Dewey and PoD)

Id	path_id	value

Element_Table (PoD-Split)

Pid	Cid	path_id	value

Element_Table (Enhanced Range Encoding)

start	end	level	path_id	value

XML_Path (XRel and others)

path_id	pathExp

XML_Path (PoD)

path_id	pathExp	type	level	occurrence	table_name

Figure 4-3: Relational schema for major relational-based approaches.

4.5.1 Supporting Search on Path Expression

We have noticed that storing path expressions in their original format (i.e. /A/B/C), or as it was proposed in XRel (which does not add the '#' character to the tag name of the last entry of the path expression '/A#/B#/C') [Yoshikawa *et al.*, 2001], does not support XML documents with tag names that might have a few words in common (such as <Book> and <BookList>). To overcome this problem, we add the '#' postfix at the end of each tag name, including attributes and all other node types, to distinguish them.

Proposition 4-1: *For any path expression in the format '/A/B/C' that exists in the XML document d, an equivalent path in the format '#/A#/B#/C#' is created in the XML_Path table.*

The reason for this path representation is to support the SQL wildcard search on the path expressions and tag names in the XML documents (i.e. use ‘%’).

The following example will clarify the effect of that addition. Given a query in XPath language (//Section), this expression should retrieve all the elements in the document with the tag name ‘Section’. Using XRel Path table, which stores path expressions in the format (/A#/B#/C), the (//Section) query can be evaluated in either one of the following ways:

1. `(Where pathExp like ‘%/Section’)`: would select only leaf nodes
2. `(Where pathExp like ‘%/Section#%’)`: would select only parent nodes (nodes in the middle of the tree)
3. `(Where pathExp like ‘%/Section%’)`: would select both types of nodes (i.e. parent and leaf nodes) but might include other irrelevant tag names that start with the word ‘Section’ (for example, <Section-Abstract>)

None of the three options would cover all the correct evaluations of the tag name <Section>.

However, using Proposition 4-1, the XPath query (//Section) can be safely translated to:

`(Where pathExp like ‘#%/Section#%’)`: this will retrieve all the occurrences of <Section> and only <Section> elements regardless of their location in the tree (i.e. leaf or parent nodes).

4.5.2 Optimising Child and Descendants Axis Steps

Many implementations for child or descendant axis steps start by finding the context node and then use the label id (either Dewey or interval encoding) to retrieve all child nodes or descendant nodes as follows:

```
Select t2.results

From element_table t1, element_table 2, PathExp t3

Where t3.path = ‘pathExpression’

      and t3.path_id = t1.path_id

      and t2.id > t1.pre and t2.post < t1.post
```

And using Dewey labels, the last line can be changed to:

```
      and t2.id between the t1.id and xmc(t1.id)
```

The run cost of this plan is the cost of the range join (θ-join) of the label id index on itself, which primarily depends on the number of records in the label id column and their index physical structure. Further, using the Dewey labels can be an advantage because the range evaluation will occur on the same value (‘t2.id’), while the number-based interval approach requires two different values: ‘pre’ and ‘post’.

However, the XML_Path table provides an alternative way to validate such queries by using the SQL wildcard search on the path expression combined with the node type field to filter out the attribute or other types of nodes. Storing the node type in the XML_Path rather than in the data tables along the instance of each path (such as the Edge table or equivalent tables) will save storage space and reduce the search space at early stages during the query run as follows:

```
Select t1.results
From element_table t1, XML_Path t2
Where t2.path like ‘pathExpression#%’
      and t2.nodetype = 0
      and t2.path_id = t1.path_id
```

The run cost of this plan will be the cost of using the foreign key to retrieve the target results, all of which will be part of the result. This technique eliminates any intermediate results, which is one of the most important goals of XML query optimisation.

4.5.3 Evaluating Ancestor at any Level of the Tree

Despite the fact that Dewey labels were originally adapted into XML management systems to support dynamic operations (such as insert, delete and update), Dewey labels also have another powerful feature. The Dewey label of any node contains the label ids of its ancestors, all the way back to the root node, encoded as part of its label. This feature can be used to retrieve any ancestor node’s label id at any level of the tree.

For example, given the Dewey label of a node ‘E’ with path expression like ‘/A/B/C/D/E’, we can retrieve the label of the ancestor node ‘B’ by recursively applying the parent function xp(). Or by applying the XGP function xgp() and passing the level of node B as a second parameter. This technique represents the significance of the document structure information or schema summary (such as level values).

Some of the previous works in the literature suggested that there is no need to store the level value because it can be retrieved or calculated by parsing the label id and counting the number of components in that label [O'Neil *et al.*, 2004; Tatarinov *et al.*, 2002]. Moreover, with absolute path expressions that do not contain the descendent axis '//', it is also easy to find the level of each node by parsing the path expression.

However, it will be a challenging task to find the label of an ancestor node in expressions that contain the descendent axis '//'. For example, if we are given the label id of node E in an expression like '/A//B//D/E' and it is required to find the label id of node B. With such path expression, it is relatively difficult and complex to find the level of the right node B, which is an ancestor of node E by just parsing the path expression. Moreover, this validation or path expression parsing may happen in the middleware layer, which might not take advantage of the capabilities of the relational database systems. The widely used approach in the literature to address that sort of query is to have a self-join statement on the element table or similar tables, and find the label of node B that encloses the label of node E as follows:

```
Select t2.results
From element_table t1, element_table t2, PathExp t3, pathExp t4
Where t3.path like '#/A#%/B#%/D#/E#'
      and t4.path like '#/A#%/B#'
      and t3.path_id = t1.path_id
      and t4.path_id = t2.path_id
      and t2.id < t1.id and xmc(t2.id) > xmc (t1.id)
```

The query plan will not be efficient, especially if the element table contains millions of records.

However, by using the extended information (i.e. the level field) in the XML_Path table, we can optimise the above query to be more cost-effective, which does not need to have expensive self-join on the element table:

```
Select xgp(t1.id, t4.level)
From XML_Path t3, XML_Path t4, element_table t1
Where t3.path like ‘#/A#%/B#%/D#/E#’
      and t4.path like ‘#/A#%/B#’
      and t3.path_id = t1.path_id
```

The same technique can be used to find the labels of all ancestor nodes of a certain node, which we will explore when we discuss optimising the ‘ancestor’ axis step in Section 4.6.9.

4.5.4 Evaluating Document Structure and Statistics Queries

The extended XMP_Path table can also be used to answer some structure queries without the need to run the query against the actual data instance tables. The combinations of the pathExp, level, type and occurrence fields provide enough information to evaluate:

- the existence of a certain path expression;
- the number of certain nodes at a certain level;
- the number of all nodes at a certain level; and
- the existence of attributes, elements, leaf nodes and other type of nodes.

Further, one more field, the ‘table_name’, can be added to the XML_Path to provide information about where to find the instances of each path expression. This can be used with advanced optimisation based on data partitioning. The XML document can be partitioned in more tables than just the traditional ones (such as elements, attribute and text). This can be useful for very large XML documents to group frequently queried nodes in certain tables.

4.6 Evaluating and Optimising the XPath Axis Steps

Some of the range methods to evaluate XPath axis steps were also adapted by the Dewey-based labelling approaches to evaluate ancestor-descendent relationships [Böhme and Rahm, 2004; O'Neil *et al.*, 2004; Tatarinov *et al.*, 2002]. Node B is a descendent of node A if the id (label value) of node B is greater than the id of node A and smaller than the maximum upper limit for all A's descendent nodes. Two main functions are proposed to help in evaluating some axis steps based on the range comparison:

- Parent (id) function: removes the last component of the Dewey label id of the context node to return the Dewey label id of the parent node.
- Maximum Child (id) function: returns the label upper limit for any child node that descends from the context node.

We are building on these findings; however, we will discuss further optimisation techniques for each axis step. The following abbreviations will be used in the next subsections:

- The Dewey label id for context node:
  - One-component label mode: id_{cn} (cn denotes context node)
  - Split-label mode (parent, child): Pid_{cn} and Cid_{cn} respectively
- The Dewey label id for result node (node to find):
  - One component label mode: id.
  - Split-Label mode (parent, child): Pid and Cid respectively.
- Parent (id) function: xp(id), which stands for XML parent id.
- Maximum Child (id) function: xmc(id), which stands for XML Maximum child id.
- Concatenate (parent, child) components to form equivalent Dewey id: xconcat(Pid, Cid), which stands for XML concatenate, and is used in the split-label mode.
- Self (or child) component function: xcc(id), which stands for XML child id, and returns the last component of any Dewey id.

4.6.1 XPath Axis Steps Overview

XPath axis steps are the basic structure unit of every XPath expression; they determine the direction of traversing an XML tree. An axis step returns a sequence of nodes that are reachable from the context node via a specified axis; an axis step may be a forward step or a reverse step that indicates the direction of navigation through the XML tree. Table 4.1 shows the axis steps in XQuery and XPath languages based on the format (contextNode / Step :: node()).

Table 4.1: XPath axis steps.

Step	Description
Forward Axis	
Child	Return the child element nodes of the context node
Attribute	Return the attribute nodes of the context node
Self	Return the context node
Descendent	Return the element nodes descendants of the context node
descendent-or-self	Same as 'descendent' plus the context node
Following	Return the nodes following the context node in document order (excluding the descendent of the context node)
following-sibling	Same as 'following', except return only those node share the same parent with context node
namespace::	Return namespace node of the context node
Reverse Axis	
Parent	Return the parent node of the context node
Ancestor	Return all the ancestor nodes of the context node (i.e. parent, parent of the parent and so forth until the root node)
ancestor-or-self	Same as 'ancestor' plus the context node
Preceding	Return the nodes that are not ancestor of the context node and occur before the context node in document order
preceding-sibling	Same as 'preceding' except return only those nodes share the same parent with context node

4.6.2 The Child Axis Step

There are four methods to find the child nodes of the context node:

1. Return all the nodes that have 'parent id = context node id

$$\mathtt{xp(id)} = \mathtt{id}_{cn}$$

   This will scan the whole label id index and evaluate every label id record using the 'xp' function. This is not the optimum evaluation technique. However, it can be enhanced if we build an index on the function xp(id) in the database systems that support indexes on functions.

2. The most popular alternative option is to use the range technique (RT). This involves finding all the nodes with the label id that is greater than the context node label (the id_{cn}) and less than the upper limit for any child node of the context node:

```
Return all nodes that have
    (id > id_cn and id < xmc(id_cn) and their level = level_cn + 1)
```

   This technique will generate a large fragment of irrelative intermediate results, which can be costly if the context node has a very large number of descendent nodes.

3. Using the schema summary in the XML_Path table, which we have developed as part of the PoD, the wildcard SS on the path field can be used to find possible matches for the context node and its possible child nodes. This can be joined to the element_table using the path_id as a foreign key:

```
Return all nodes in Element_table that have path_id = path_id2 in
XML_Path where:
Path1 = 'context node path'
and path2 like 'context node path/%'
and their level = level_cn + 1
and their node_type = 0
```

   This approach is powerful because the XML_Path table is relatively small. By using the foreign key, this approach can efficiently support very large documents almost in linear time.

4. Using our approach for Split Dewey id (Pid and Cid):

```
Return all nodes that have Pid = xconcat(Pid_cn, Cid_cn)
```

   This technique can be even faster than the third option and it can be applied to any database system that will take advantage of the available indexing techniques. Further, the number of intermediate results will be minimal due to the efficient use of the primary BTree index on (ParentLabel, ChildLabel).

4.6.3 The Descendant Axis Step

The optimised technique is to use the range search, which makes use of the index on the label id, as follows:

```
Return all nodes that have id > id_cn and id < xmc(id_cn)
```

And for split-label:

```
Return all nodes that have Pid between xconcat(Pid_cn, Cid_cn) and xmc(xconcat(Pid_cn, Cid_cn))
```

Another powerful technique is SS using the schema summary in the XML_Path table, which is the same as the technique we proposed for the child axis step:

```
Return all nodes in Element_table that have path_id = path_id1 in XML_Path where:

Path1 = ‘context node path/%’

and their node_type = 0
```

The query has been simplified to a join between the XML_Path table and Element_table based on the foreign key path_id.

Note: special consideration is needed if a wildcard search is used (i.e. //node) mainly in recursive XML schema, as we will cover in detail in Section 4.6.8.

4.6.4 The Descendant-self Step

The descendant-self axis step is similar to the descendant axis, only modifying the range of the search to include the parent node as follows:

```
Return all nodes that have id between id_cn and xmc(id_cn)
```

And for split-label:

```
Return all nodes that have Pid between xconcat(Pid_cn, Cid_cn) and xmc(xconcat(Pid_cn, Cid_cn))

Union

Return all nodes that have Pid = Pid_cn and Cid = Cid_cn
```

The last line is necessary for split-label representation to retrieve the context node itself.

In addition, the SS approach can be used with slight modifications as follows:

```
Return all nodes in Element_table that have path_id = path_id1 in
XML_Path where:
path1 like ‘context node path%’
and their node_type = 0
```

4.6.5 The Attribute Axis Step

This step is similar to the ‘child’ step; however, it should target the right table for the attribute or use the node type to distinguish the attribute nodes as follows:

```
Return all nodes that have id > id_cn and id < xmc(id_cn)
And their node_type = 1
```

In addition, the SS approach can be applied as follows:

```
Return all nodes in Atribute_table that have path_id = path_id1 in
XML_Path where:
Path1 = ‘context node path/@%’
and their node_type = 1
```

4.6.6 The Following Axis Step

The following nodes are all the nodes that come after the context node in a document order, excluding its descendant nodes. This axis step can be evaluated by using the upper limit value for any child node that descends from the context node as follows:

```
Return all nodes that have id > xmc (id_cn)
```

And for split-label

```
Return all nodes that have xconcat(Pid,Cid) > xmc(xconcat(Pid_cn,
Cid_cn))
```

The one-component label has an advantage over the split mode because the split mode has to evaluate each record in the index after applying the xconcat function, which will be slower than the one-component Dewey label.

4.6.7 The Following-sibling Axis Step

We can use the same technique that is used for the ‘following’ axis step; however, one more condition is required to select only sibling nodes:

```
Return all nodes that have id > $id_{cn}$ and xp(id) = xp($id_{cn}$)
```

And by using the optimised RT, it can be rewritten as follows:

```
Return all nodes that have
id between xp($id_{cn}$) and xmc(xp($id_{cn}$))
id > $id_{cn}$
and level = $level_{cn}$
```

However, the spilt-label can take advantage of its structure and evaluate this axis step more efficiently:

```
Return all nodes that have Pid = $Pid_{cn}$ and Cid > $Cid_{cn}$
```

Meanwhile, the ‘following-sibling’ axis step is evaluated very slowly using the number-based range encoding since the parent-child and sibling relationships are the most challenging axis step for this labelling method.

4.6.8 The Parent Axis Step

The ‘parent’ axis step is one of the reverse axis steps that can be evaluated more efficiently in Dewey representation rather than the range labelling method.

```
Return all nodes that have id = xp($id_{cn}$)
```

And for split-label:

```
Return all nodes that have xconcat(Pid, Cid) = $Pid_{cn}$
```

However, the split-label mode would be much slower than the one-component label for this query because it would be scanning the whole label id index and applying the ‘xconcat’ function to each record. To further optimise this query for split-label, we can rewrite it as follows based on the fact the Pid of the context node is the Dewey label of the parent of the context node:

Return all nodes that have Pid = xp(Pid_{cn}) and Cid = xcc(Pid_{cn})

Using the functions xp and xcc allows us to retrieve the split label of the parent node (parent id Pid and child components respectively). Breaking the Pid_{cn} into the two components Pid and Cid significantly improves the efficient use of the label index to search for these values.

Note: it is very important to use the DISTINCT in the select statement because we need to return duplicate-free results and there might be more than one child node that matches the selection conditions, which will result in returning the same Pid more than once.

4.6.9 The Ancestor Axis Step

One option to evaluate this axis step is to apply the ‘xp’ function recursively. However, this might require modification to how relational database systems execute functions. The widely used approach is the range method, which requires no modifications to the database kernel:

Return all nodes that have id < id_{cn} and xmc(id) > xmc(id_{cn})

The run-time highly depends on the global order of the context node within the XML document. The same is true for the split-label mode:

Return all nodes that have Pid < Pid_{cn} and xmc(xconcat(Pid,Cid)) > xmc(xconcat(Pid_{cn},Cid_{cn}))

However, this axis step can be optimised to run faster by utilising the document structure and schema information in the XML_Path table.

A broader search is required on all the valid paths that end with the context node (i.e. '%PathExp') and pass their level values to the xgp() function as a second parameter, as follows:

```
Select xgp(t1.id, t4.level)
From XML_Path t3, XML_Path t4, element_table t1
Where t3.path like '#%/PathExp#'
      and t3.path startsWith(t4.path)
      and t3.path_id = t1.path_id
```

It can be extended at nearly no cost to retrieve more information about the ancestor nodes in case they are required for another evaluation or comparison within the same query:

```
Select t1.id, t1.path_id
From XML_Path t4, XML_Path t3, element_table t2, element_table t1
Where t4.path like '#%/PathExp#'
      and t4.path startsWith(t3.path)
      and t4.path_id = t2.path_id
      and t3.path_id = t1.path_id
      and t3.level < t4.level
      and t1.id = xgp(t2.id,t3.level)
```

The 'startsWith' function, which is used for string comparison, would ensure that the query processor will evaluate correctly only ancestor nodes of the valid context node. The 'startsWith' function is a built-in function that is provided by almost all database vendors and there is no need to invade the relational system to be able to apply the xp() function recursively or run such an evaluation in the middleware layer. For example, the MySQL syntax for the 'startsWith' function looks like:

```
String1 REGEXP concat('^', String2) ;
      which means String1 starts with String2.
```

It is worth mentioning that such a task is challenging if we only use the RTs.

4.6.10 The Ancestor-self Axis Step

The axis step is similar the 'ancestor' axis step and the only difference is replacing the '<' and '>' operators with '<=' and '>=' respectively.

4.6.11 The Preceding Axis Step

This axis step can be evaluated using the same technique that is used in the range labelling method:

```
Return all nodes that have xmc(id) < id_cn
```

This can be optimised by avoiding scanning all nodes in the label id index as follows:

```
Return all nodes that have (id < id_cn and xmc(id) < id_cn)
```

And for split-label:

```
Return all nodes that have
      (Pid <= Pid_cn and xmc(xconcat(Pid,Cid)) < xconcat(Pid_cn,Cid_cn))
```

The spilt-label mode might be slightly slower than one-component label representation in some cases.

4.6.12 The Preceding-sibling Step

This is similar to the 'preceding' axis step, except it returns only sibling nodes that have a document order less than the context node's document order:

```
Return all nodes that have (id < id_cn and xp(id) = xp(id_cn))
```

And for spilt-label:

```
Return all nodes that have (Pid = Pid_cn and Cid < Cid_cn)
```

The split-label will run more efficiently because it better utilises the index on the label id.

4.7 XPath Axis Steps Experimental Evaluation

4.7.1 Study Overview

We have not discovered detailed studies for optimising XPath axis steps based on Dewey labels because most of the studies have made use of the current techniques that are applied for number-based range encoding. This experimental study aims to evaluate the capability and efficiency of Dewey-based labels in addressing queries based on the XPath axis steps and comparing that to the efficiency of the number-based labelling methods.

While the experimental evaluation has been conducted by translating XML queries into SQL statements, we believe some of the proposed techniques can be used or implemented in other XML management systems (such as native and hybrid systems). The reason for using SQL is to evaluate the efficiency of unmodified relational database systems for supporting XML documents.

4.7.2 Experiment Setup

We have implemented PoD and PoD-S as mentioned in the discussion in Chapter 3. In addition, we have implemented all the required functions in section 3.3.7. MySQL server 5.1 was used as a backend relational storage system. We have also implemented numeric-based interval (such as range) encoding as mentioned in the literature [Yoshikawa *et al.*, 2001], using the node identification key (pre, post, level). The three implementations used the similar XML_PATH table, which contains summary information of the document structures and schema.

The sample XML documents were generated using very well-known XML benchmarks: XMark [Schmidt *et al.*, 2002], and the Michigan benchmark [Runapongsa *et al.*, 2006]. We used documents from both benchmarks of different sizes; for XMark, we used documents of two sizes, 10MB and 100MB, and for the Michigan benchmark, we used two documents of size 50MB and 500MB. We had to build some queries to address the axis steps that we were evaluating because the two benchmarks do not have queries that specifically challenge the capability and efficiency to support the XPath axis steps without having other factors included, such as conditions on the data values.

Apache Java parser was used for parsing and loading XML documents. Java was also used for automating and implementing the tests. This test was run on an Intel Core2 CPU (2.8GHz) machine with 4GB of RAM. We ran each query ten times and reported the average run time after excluding the first run.

4.7.3 Child Axis Step

As mentioned in Section 4.6.2, the child axis step can be evaluated in three different ways: Range Technique (RT); using the SS on the path values in the XML_PATH table (SS); and using the parent value in Pod-S (PR). We have evaluated the first technique against PoD, Pod-S and number-based range encoding (Range). However, the second technique can be applied to those systems that have the XML_Path table (such as PoD and Pod-S), and the third approach (PR) can only be applied to Pod-S.

None of the provided queries in either benchmark has simple querying based on the child axis; Q1 of XMark and QR2 of the Michigan benchmark contain child axis evaluation but as a part of simple twig queries or tree-pattern match, Appendix A provides complete list of the queries in each benchmark. However, we created the following queries (see Table 4-2) based on XML documents in each benchmark:

Table 4-2: Queries used to evaluate the child axis step.

Query	Benchmark	Query Statement
Q1	XM	benchmark query: simple twig query, which involves child axis step: *FOR $b IN /site/regions/namerica/item[@id="item20748"]* *RETURN $b/name/text()*
Q2	Using XM	/site/closed_auctions/closed_auction/annotation/description/text/bold/Child::node()
Q3	Using XM	/site/open_auctions/open_auction/Child::node()
Q4	Using Mich	/eNest/eNest/eNest/Child::node()
Q5	Using Mich	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/Child::node()
QR2	Mich	benchmark query: simple twig query which involves Child axis step*:* *//eNest[@aSixtyFour=2]/Child::node()*

Figures 4-3 and 4-4 illustrate the run-time results of the queries in Table 4-2 using XML documents from the XMark and Michigan benchmarks respectively.

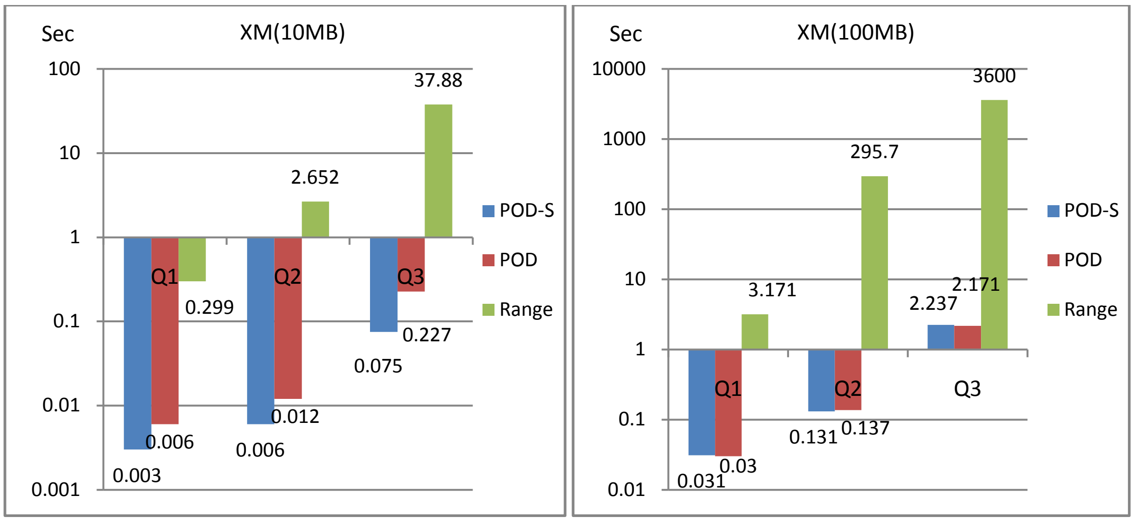


Figure 4-3: Query run-times in logarithmic scale for queries Q1, Q2, and Q3 in Table 4-2.

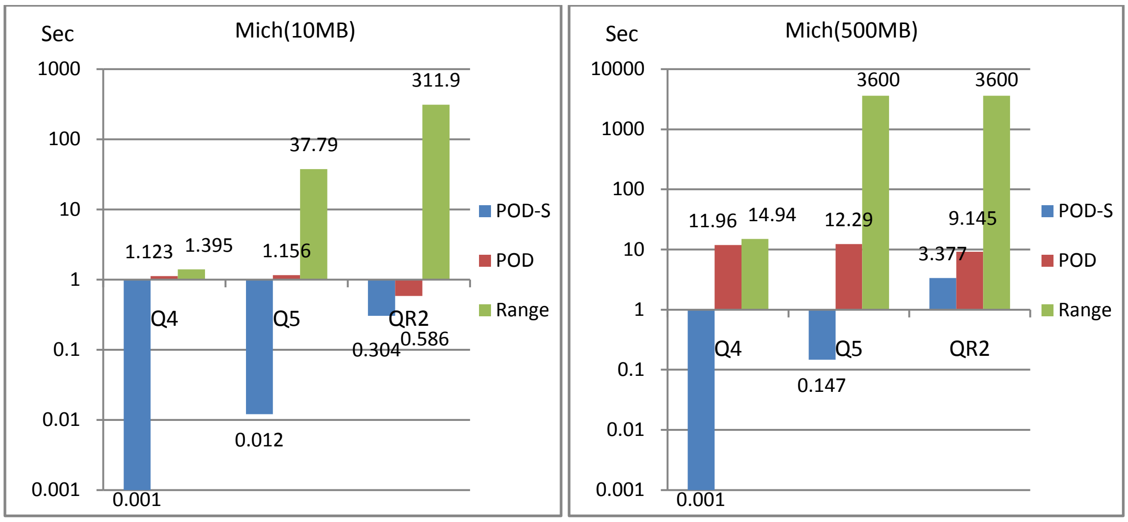


Figure 4-4: Query run-times in logarithmic scale for queries Q4, Q5, and QR2 in Table 4-2.

The results demonstrate that Dewey-based labels outperformed the numeric intervals approach for all queries in Table 4-2 and for all documents from the two benchmarks. The main advantage of the Dewey label is that the evaluation occurs for a single value (i.e. the label id); meanwhile, the numeric intervals approach involves two different values (Pre, and Post), which makes it much slower to scan the index and find the correct results. Further, the numeric intervals approach performance did not scale up well when the number of the targeted nodes increased by increasing the document size or when it went deep down the XML tree as we moved from Q1 to Q2 and from Q4 to Q5. In some cases, the (XRel) approach did not return results within an hour (3600sec) and we had to abort the query run.

As expected and mentioned in the discussion, the Dewey label in split mode (PoD-S) runs much faster than the one-component Dewey label (PoD) due to the more efficient use of the index on the label values; for such axis steps, it used the equijoin on the parent component rather than the range join.

We also evaluated the string search (SS) approach that we proposed in Section 4.5.1. We ran queries Q2, Q3, Q4 and Q5 using translated queries based on the path string match. The results in Figures 4-5 and 4-6 illustrate that the SS approach significantly enhanced the query run time for PoD, especially for the recursive XML documents, such as those in the Michigan benchmark documents. However, the performance gain was less in the case of the PoD-S mode, since the PoD-S approach is efficient for the child axis step queries.

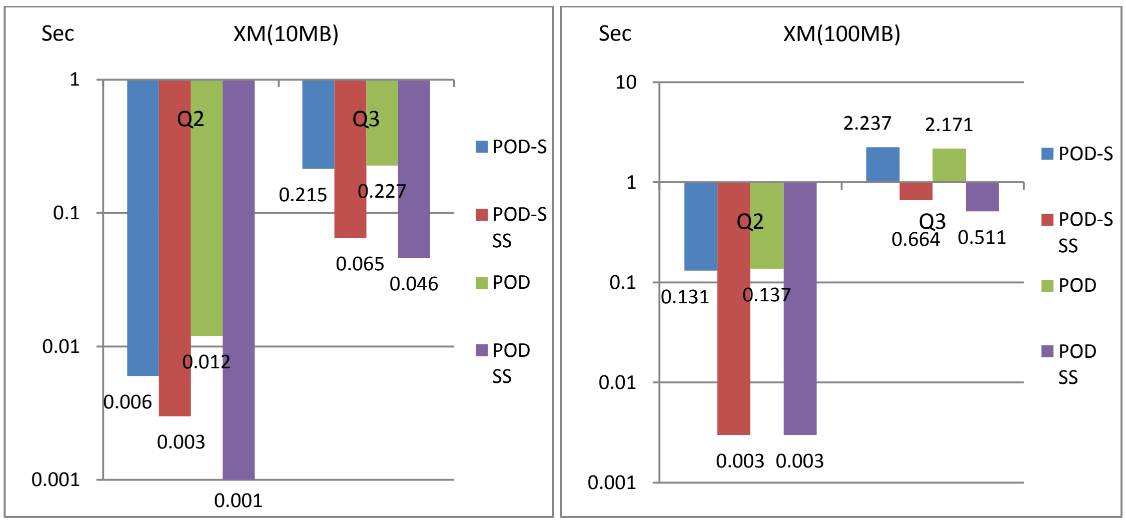


Figure 4-5: Query run-times in logarithmic scale for queries Q2, Q3 in Table 4-2.

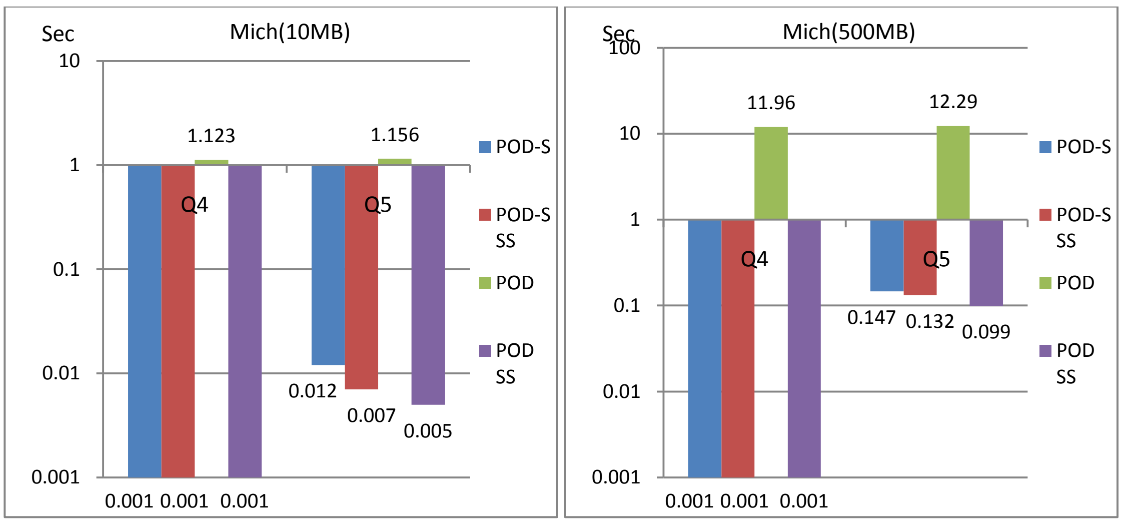


Figure 4-6: Query run-times in logarithmic scale for queries Q4, Q5 in Table 4-2.

4.7.4 Descendant Axis

The descendant axis is similar to the child axis except it returns all the nodes that are enclosed by the context node. In other words, we can apply the same RT in the previous subsection with a slight modification to the level condition; more details are provided in Section 4.6.3. The same technique will be used for the Pod-S mode because the parent approach (PR) cannot be used to evaluate this axis step.

Table 4-3 contains a modified version of the queries in Table 4-2. The queries in table 4-3 were used to evaluate the 'Descendent' axis step, using XML documents from both benchmarks.

Table 4-3: Queries used to evaluate the descendant axis step.

Query	Benchmark	Query Statement
Q1	Using XM	/site/closed_auctions/closed_auction/annotation/description/text/bold'/Descendant::node()
Q2	Using XM	/site/open_auctions/open_auction/Descendant::node()
Q3	Using Mich	/eNest/eNest/eNest/Descendant::node()
Q4	Using Mich	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/Descendant::node()
QR3	Mich	benchmark query: simple twig query which involves Descendant axis step. *//eNest[@aSixtyFour=2]/Descendant::node()*

The results shown in Figure 4-7 for XMark queries and Figure 4-8 for Michigan benchmark queries, demonstrate the superiority of Dewey-based labels over the numeric intervals approach.

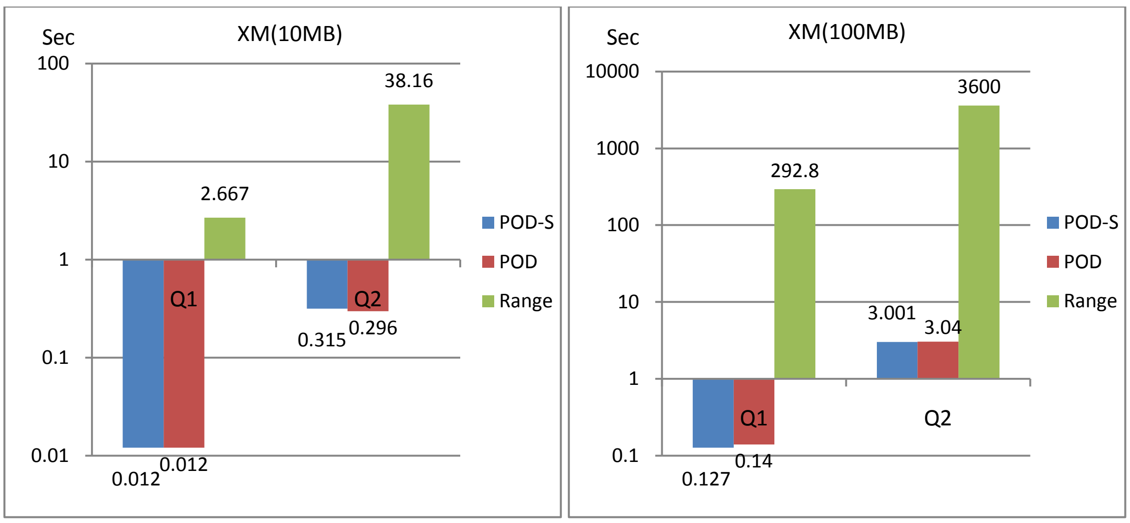


Figure 4-7: Query run times in logarithmic scale for queries Q1 and Q2 in Table 4-3.

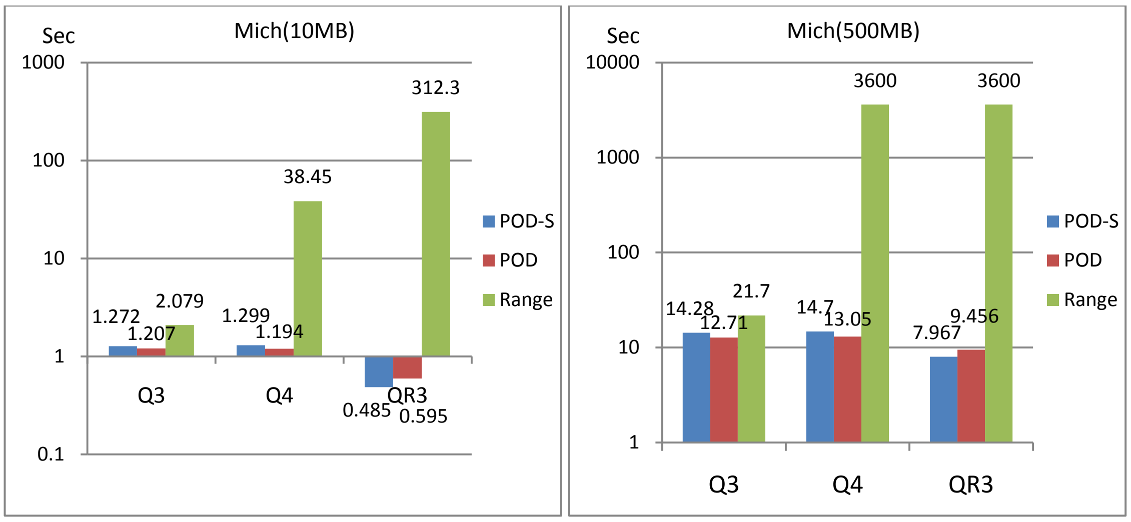


Figure 4-8: Query run times in logarithmic scale for queries Q3, Q4 and QR3 in Table 4-3.

The numeric intervals approach again did not scale up well and three queries did not finish within an hour (3600sec); Q2 against XMark (100MB) and Q3 and Q4 against the Michigan benchmark (500MB). The efficiency of the PoD and PoD-S is due to the size and structure of the labels since the range queries include calculations and comparison on the same value (label id), which is a more expensive operation for intervals because it has to calculate the range condition for two different values (pre and post).

We also ran the queries Q1, Q2, Q3 and Q4 using the SS approach; as expected, there was significant performance enhancement comparing to PoD and PoD-S (RT) techniques. The results are illustrated in Figures 4-9 and 4-10.

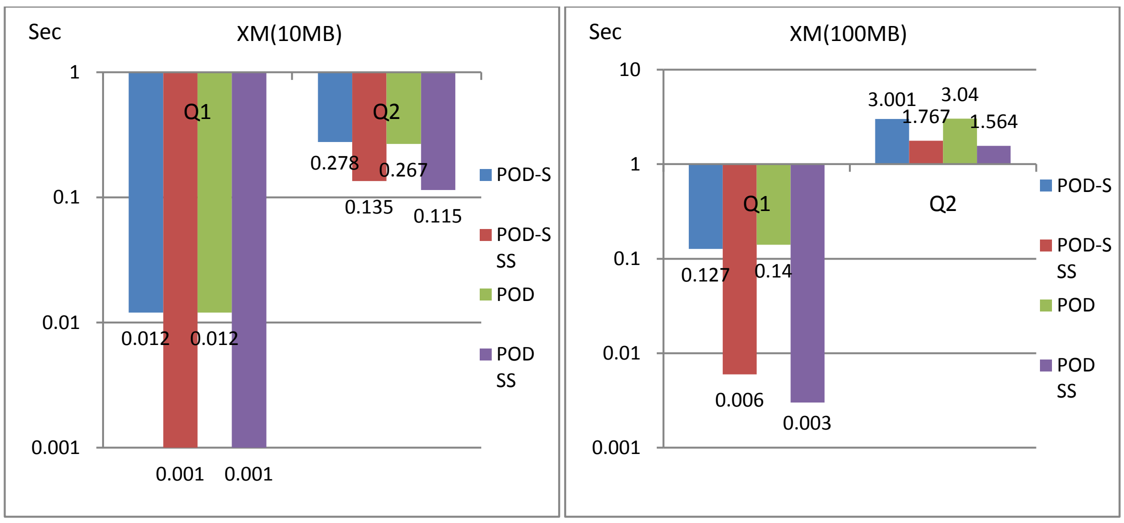


Figure 4-9: Query run times in logarithmic scale for queries Q1 and Q2 in Table 4-3.

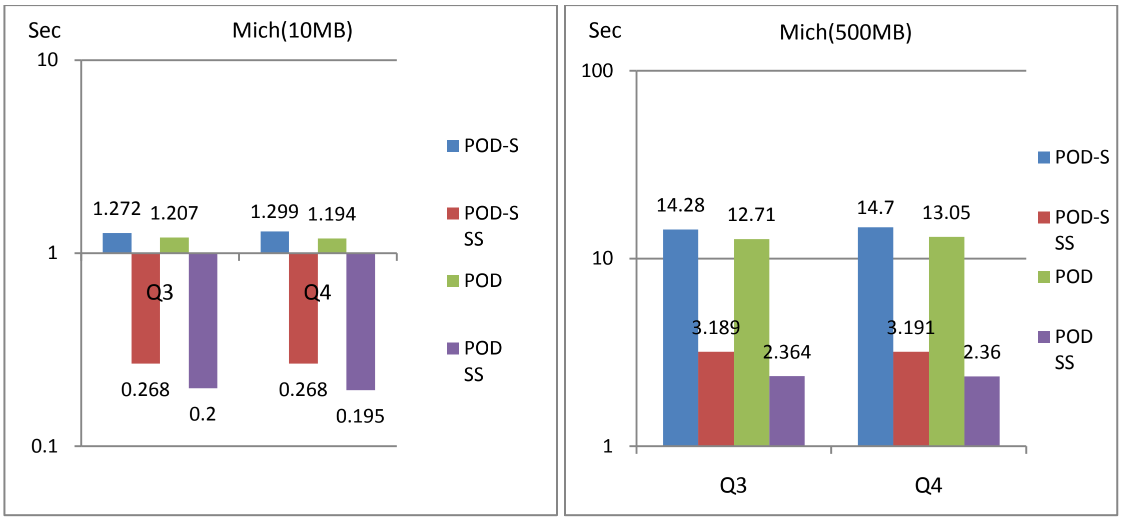


Figure 4-10: Query run times in logarithmic scale for queries Q3 and Q4 in Table 4-3.

4.7.5 Following Axis

The ‘following axis’ step highly depends on the path expression length (i.e. node depth), document global order and document size. To evaluate this axis step for Dewey labels, it is required to return any label id that is greater than the upper limit label‘s value for any descendent of the context node (i.e. id > xmc(id_{cn})). However, this is challenging for the two-component Dewey labels, such as Pod-S, because the split mode has to evaluate each record in the index after applying the xconcat function (see Section 4.6.6).

We conducted this test using the queries in Table 4-4 and by applying the RT, which we believe is the only feasible technique to evaluate this axis step because it is required to find all label ids based on their lexicographic order (i.e. id > xmc(id_{cn})).

Table 4-4: Queries used to evaluate the following axis step.

Query	Benchmark	Query Statement
Q1	Using XM	/site/closed_auctions/closed_auction/annotation/description/text/bold'/Following::node()
Q2	Using XM	/site/open_auctions/open_auction/ Following::node()
Q3	Using Mich	/eNest/eNest/eNest/ Following::node()
Q4	Using Mich	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/ Following::node()

The results in Figures 4-11 and 4-12 demonstrate that this axis step is a challenge for relational systems regardless of the labelling techniques. The numeric-based intervals techniques did better than the Dewey labels due to the structure of the label (pre and post), the query processor will be looking for 'pre' values that are greater than the 'post' value of the context node. As expected, the split mode was the slowest due to the need to apply the xconcat() function to each label (Pid, Cid) record in the label columns.

Further, the run time highly depends on the context node's location in the XML document and whether the query will generate many duplicate nodes. We noticed that this axis is very slow due to the use of the 'distinct' keyword, which is used in SQL statements to remove duplicate records in the final results.

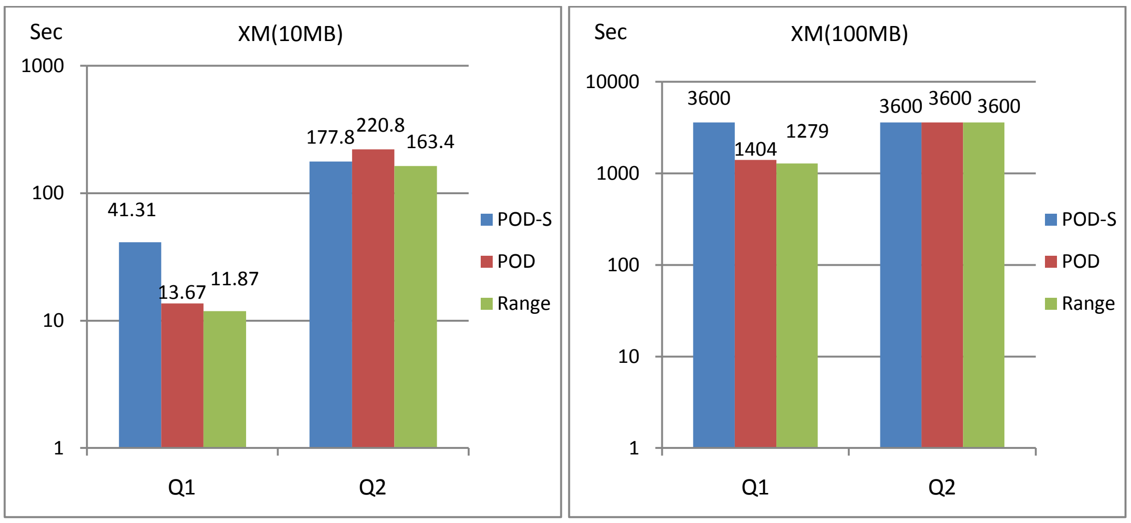


Figure 4-11: Query run times in logarithmic scale for queries Q1 and Q2 in Table 4-4.

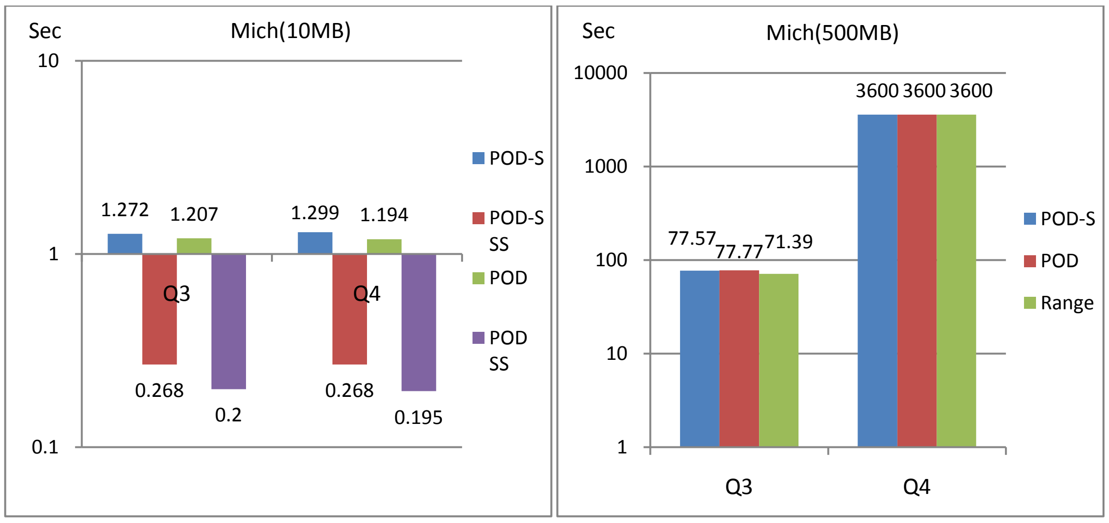


Figure 4-12: Query run times in logarithmic scale for queries Q3 and Q4 in Table 4-4.

4.7.6 Following-sibling Axis

The 'following-sibling axis' can be seen as a mixture of two axis steps: child axis and following axis steps. We compared PoD and intervals approaches using RT. However, the PoD-S was evaluated differently, taking advantage of the Pid value, which has to be the same for sibling nodes.

We used a modified version of the queries in Table 4-4 by replacing Q2 with another more appropriate query to reduce the effect of using the keyword ‘distinct’. Our objective was to focus on evaluating the effect of labelling and optimisation techniques on evaluating this XPath axis step. Further, the new query generates higher load when it is used for evaluating reverse axes as in the following subsections. Table 4-5 shows the queries that were used in this test, we translated the queries based on the discussion in Section 4.6.7.

Table 4-5: Queries used to evaluate the following-sibling axis step.

Query	Benchmark	Query Statement
Q1	Using XM	/site/closed_auctions/closed_auction/annotation/description/text/bold'/Following-sibling::node()
Q2	Using XM	/site/regions/namerica/item/name/ Following-sibling::node()
Q3	Using Mich	/eNest/eNest/eNest/ Following-sibling::node()
Q4	Using Mich	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/ Following-sibling::node()

The PoD-S mode showed outstanding performance in the figures above, even for large XML documents, because of the efficient use of the parent component (Pid) in the split label (Pid, Cid). The PoD label outperformed the numeric-based interval coding despite the fact that both were evaluated using the RT. However, the intervals approach requires three self-joins on the element_table to be able to validate a sibling relationship between any two nodes. Therefore, it must first find the parent node that encloses the two sibling nodes.

Figures 4-13 and 4-14 illustrate the results of the test for the Follwoing-sibling axis step using different document sizes from two benchmarks.

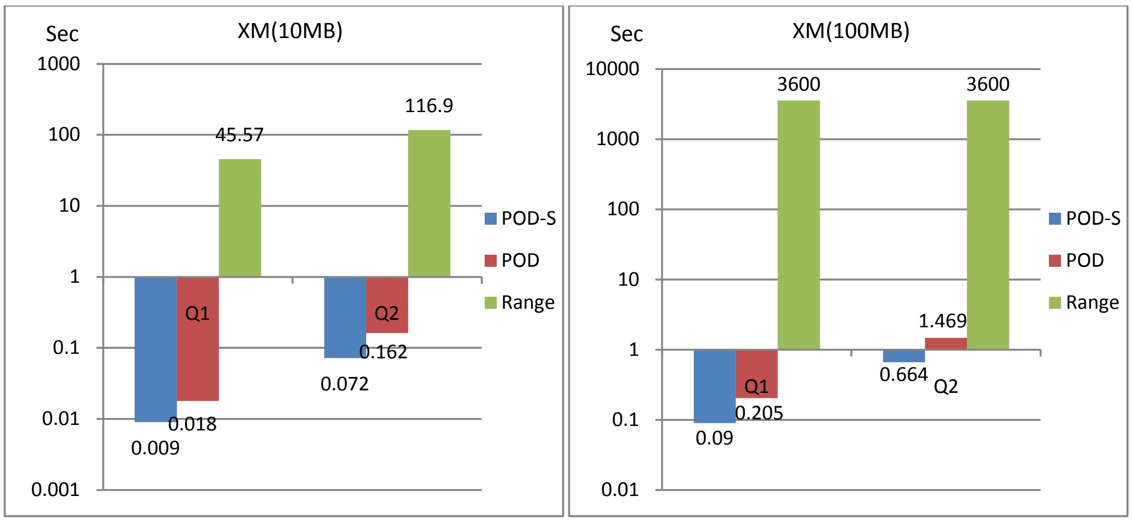


Figure 4-13: Query run times in logarithmic scale for queries Q1 and Q2 in Table 4-5.

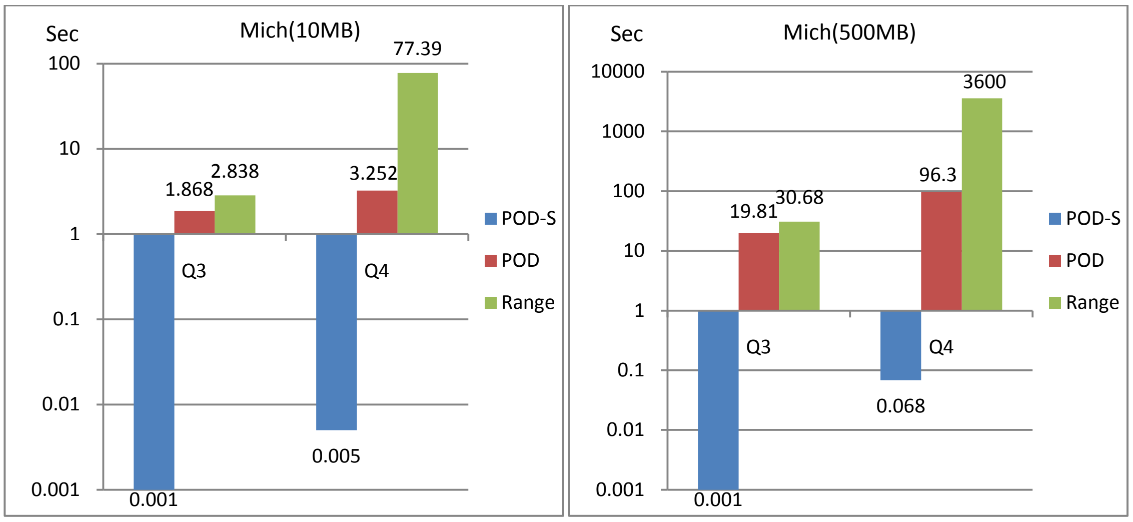


Figure 4-14: Query run times in logarithmic scale for queries Q3 and Q4 in Table 4-5.

4.7.7 Parent Axis

The ‘parent axis’ step is one of the reverse axis steps that navigates up the XML tree. Meanwhile, each Dewey label contains the label values of its ancestors, including the parent node, which makes Dewey labels more suitable and efficient to evaluate reverse axis steps. We evaluated the parent axis for PoD, PoD-S and intervals approaches using the same queries in Table 4-5 with the proper axis step ‘/Parent::node()’. The equivalent SQL queries for PoD and PoD-S were rewritten based on the discussion in Section 4.6.8.

The results illustrated in Figures 4-15 and 4-16 confirm one of the advantages of Dewey labels: the Dewey label of any given node includes the Pid of its parent and ancestor nodes. Applying the xp() function to the label id of the context node can retrieve the id of the parent node, which will result in an equijoin rather than range join on the (pre, post) values. Moreover, the Dewey labels scaled up very well, which was not the case with the numeric-based interval code.

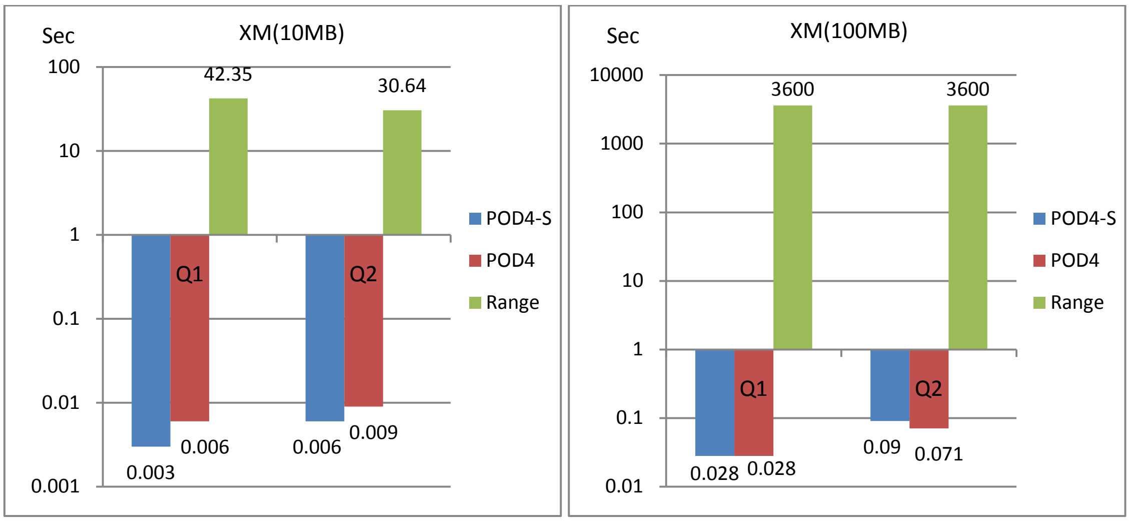


Figure 4-15: Query run-times in logarithmic scale for Q1 and Q2 in Table 4-5.

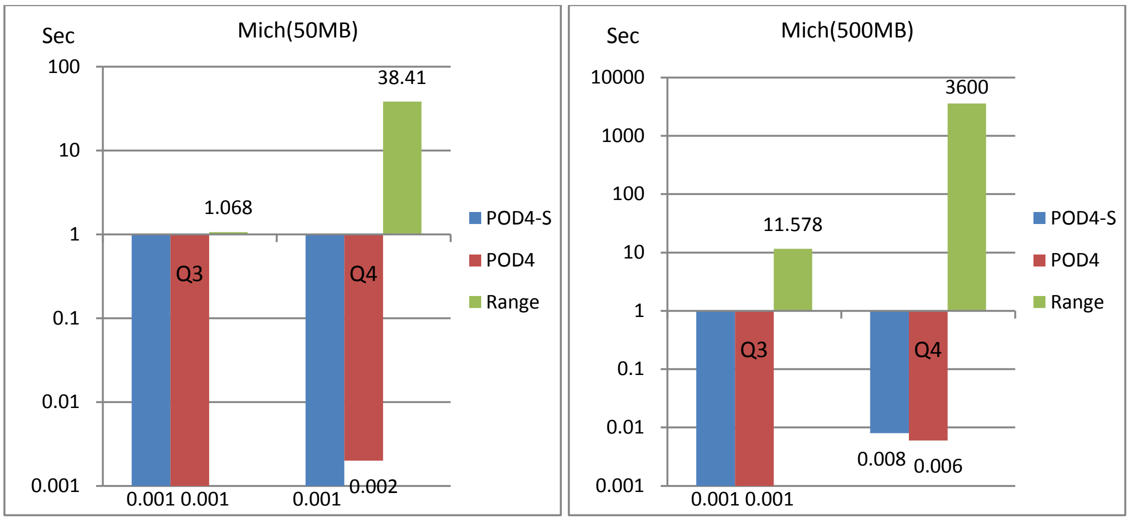


Figure 4-16: Query run times in logarithmic scale for Q3 and Q4 in Table 4-5.

4.7.8 Ancestor Axis

The ‘ancestor axis’ can be represented as a recursive parent axis; however, this would add complexity to the way in which it can be handled in relational systems. We ran the first test to evaluate the efficiency of using the RT to address this axis step.

We used the same query set as those in the previous subsection, only modifying the axis step to ‘/Ancestor::node()’. The results in Figures 4-17 and 4-18 illustrate the advantages of the Dewey labels over the numeric-based intervals encoding, even though we used the same RT for both labelling schemes. The PoD was faster than PoD-S due to the need to apply the xmc() function and the xconcat() function to the two-component labels. Once again, the numeric-based intervals encoding did not scale up well for larger documents and longer path expressions; Q2 and Q4 had to be aborted after one hour of execution.

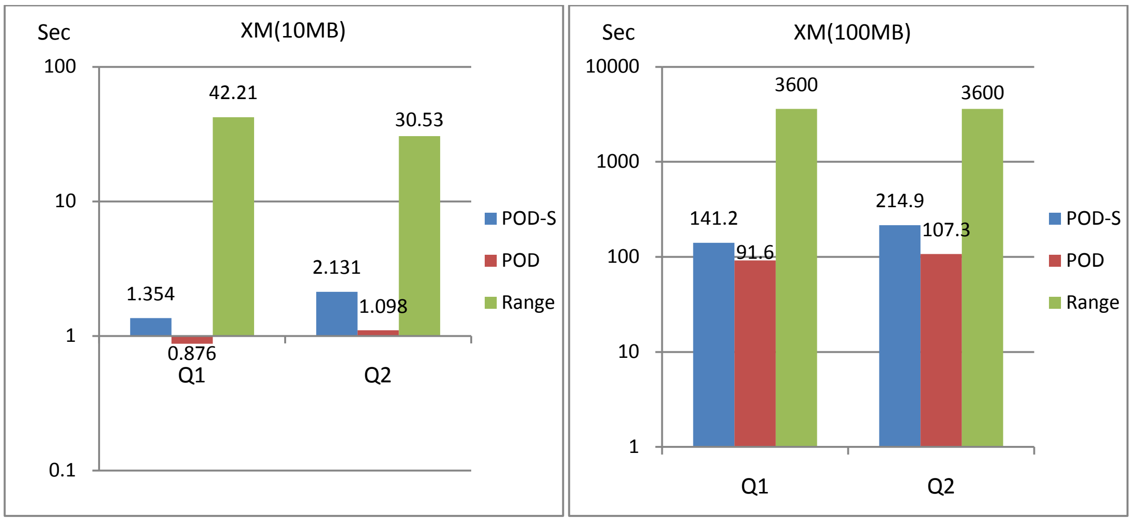


Figure 4-17: Query run times in logarithmic scale for Q1 and Q2 in Table 4-5 using the ancestor axis.

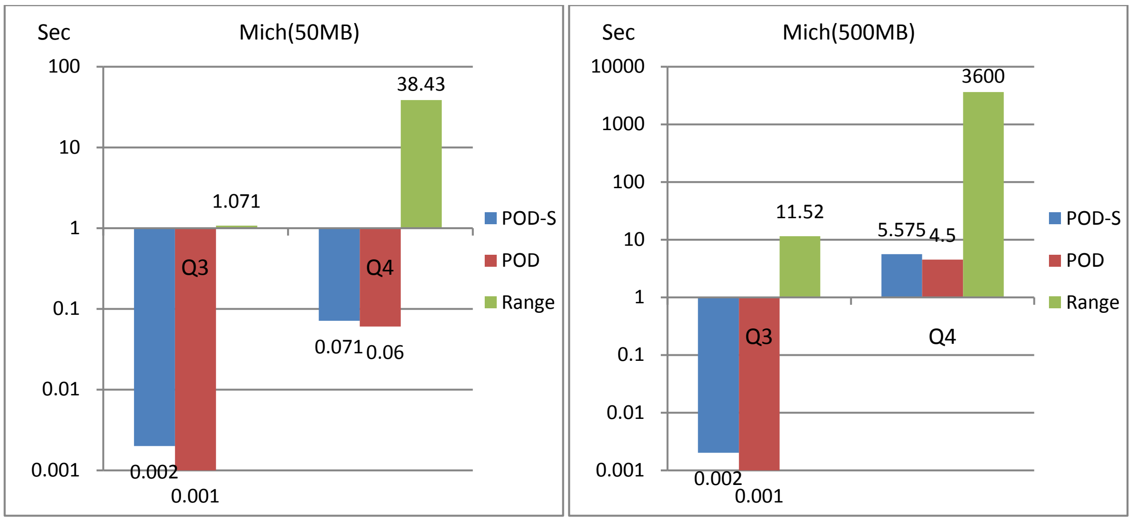


Figure 4-18: Query run times in logarithmic scale for Q3 and Q4 in Table 4-5 using the ancestor axis.

We also evaluated our proposed optimisation technique for this axis step, discussed in Section 4.6.9, which is based on using the schema summary in the XML_Path table together with the XGP function xgp().

As illustrated in Figures 4-19 and 4-20, our new optimised technique outperformed the traditional RT for PoD and PoD-S, especially when the path expression is long or when the location of the context node is deep down the XML tree. The performance gain was significant in large XML documents; it was more than 900 times faster in the XMark 100MB document and more than 100 times faster in the Michigan benchmark 500MB document.

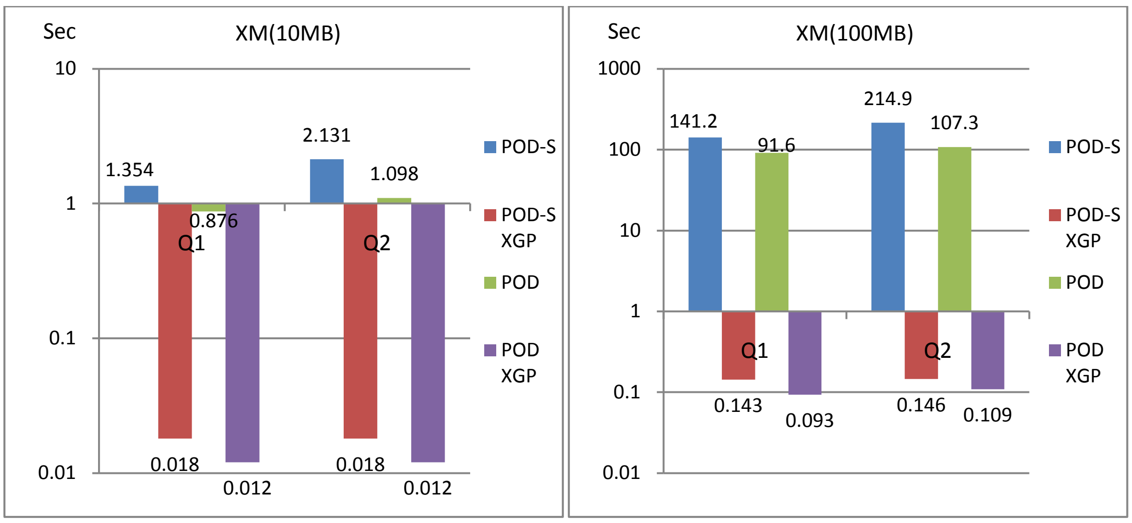


Figure 4-19: Query run times in logarithmic scale for Q1 and Q2 in Table 4-5 for the ancestor axis using XGP function.

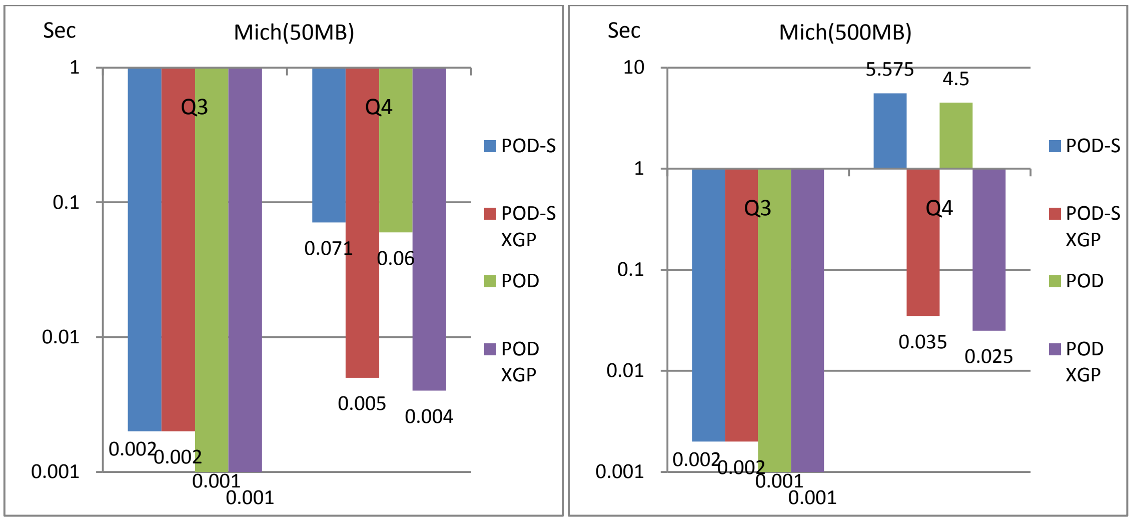


Figure 4-20: Query run times in logarithmic scale for Q3 and Q4 in Table 4-5 for the ancestor axis using XGP function.

4.7.9 Preceding Axis

This axis step can be evaluated using the RT as discussed in Section 4.6.11. This axis step is similar to the 'following' axis step, which is again considered a challenge for Dewey-based labels due to the nature of the Dewey labels and the need to apply functions to each record in the label id column. We tested the RT on PoD, PoD-S and intervals approach using the same queries in Table 4-5 with only modifying the axis step to '/Preceding::node()'. The results in Figures 4-21 and 4-22 demonstrate a slow response time for all labelling schemes with a slight advantage for the numeric-based intervals approach.

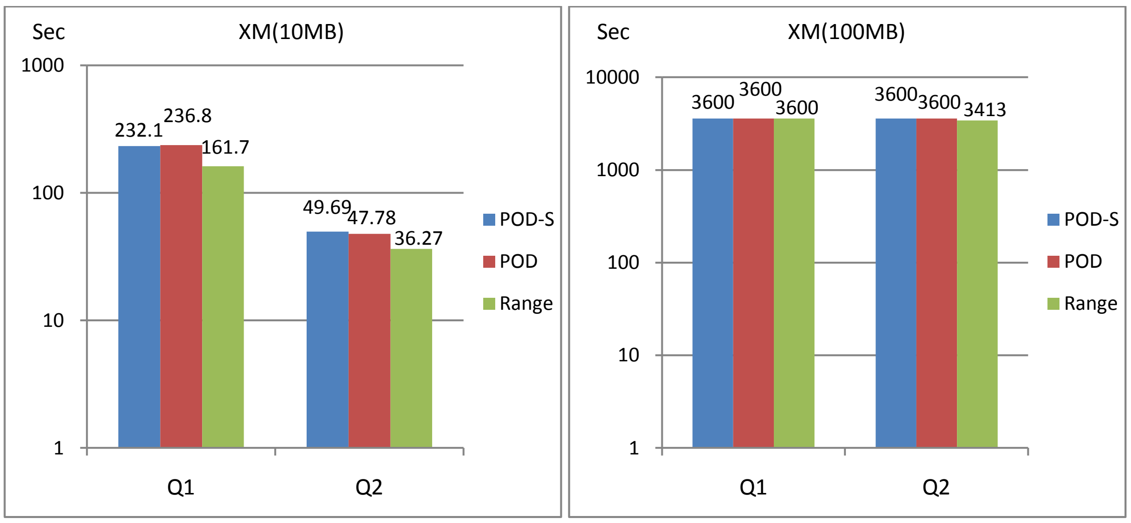


Figure 4-21: Query run times in logarithmic scale for Q1& Q2 in Table 4-5 for preceding axis.

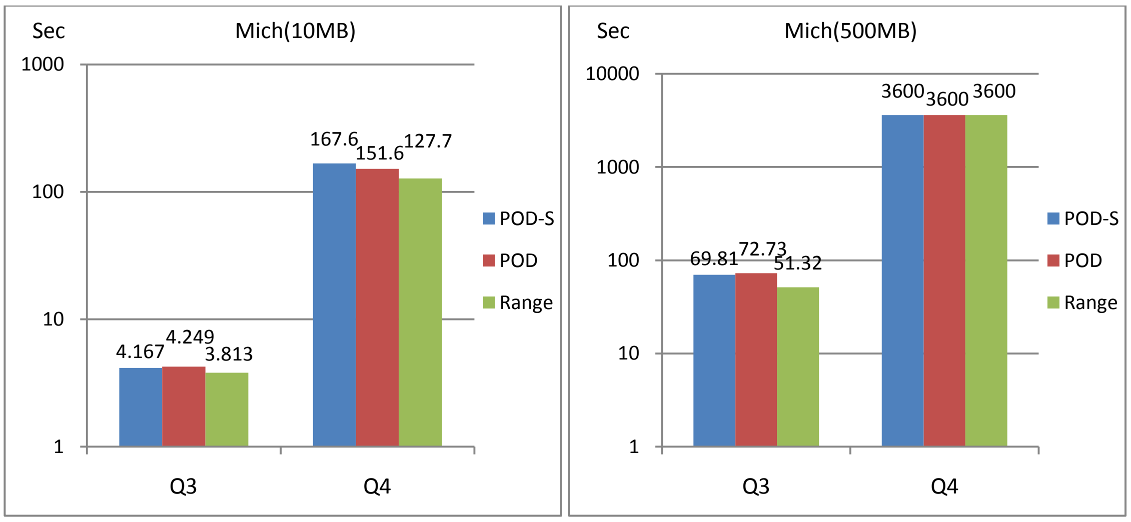


Figure 4-22: Query run times in logarithmic scale for Q3& Q4 in Table 4-5 for preceding axis.

Neither of the three approaches were able to support this axis step against large XML documents. However, the main reason for this problem is using the keyword ‘distinct’, which is used to remove a very large number of duplicate nodes in the final result sets. We cannot see a feasible solution without modifying the database system kernel to be able to handle this axis step as well as the ‘following’ axis step.

4.7.10 Preceding-sibling Axis

This axis is almost identical to the following-sibling axis step with one modification to the condition on the siblings order. We used the techniques discussed in Sections 4.6.11 and 4.6.12 to evaluate PoD, Pod-S and the intervals approaches. The same set of queries were used by only modifying the axis step to ‘/Preceding-Sibling::node()’.

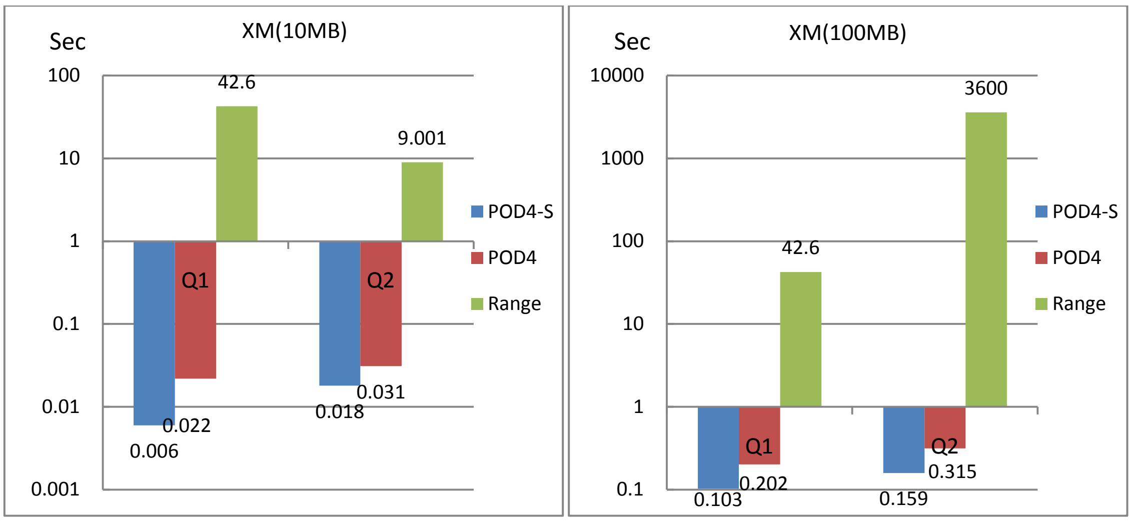


Figure 4-23: Query run times in logarithmic scale for Q1 and Q2 for preceding-sibling axis.

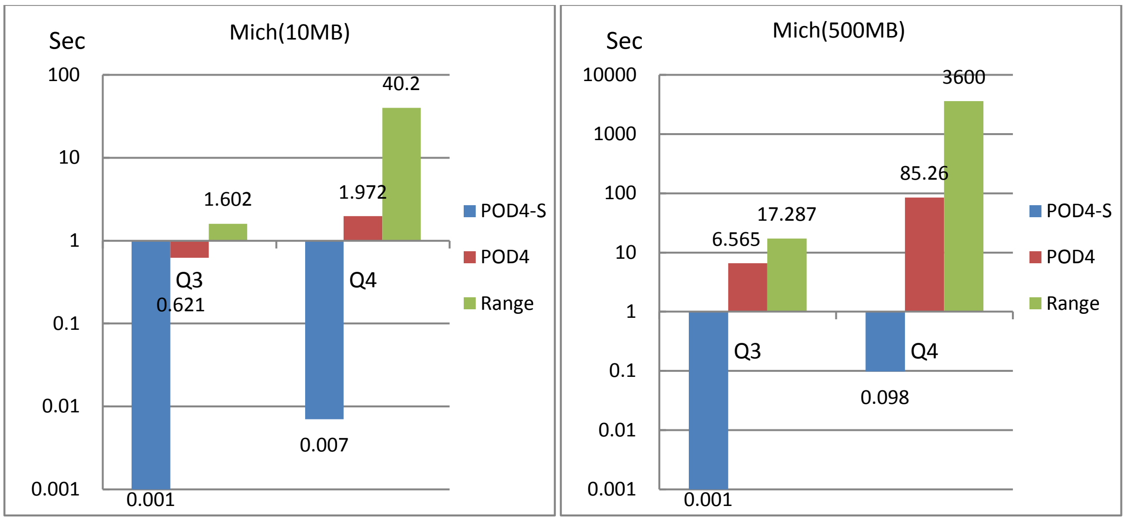


Figure 4-24: Query run times in logarithmic scale for Q3 and Q4 for preceding-sibling axis.

As expected, the PoD-S showed an outstanding performance (see Figures 4-23 and 4-24) due to the efficient use of the parent component Pid which is part of the label id (Pid, Cid); the candidate nodes must have equal Pid values. The numeric-based interval approach suffered a very slow run time due to the fact that it is very expensive to validate sibling relationships among XML nodes using this encoding method.

4.8 Conclusion

Optimising XML query for relational-based systems has been the recent focus of researchers in the industry. As we have already discussed new ideas to improve the performance of XML queries, in this chapter we focused on optimising the XPath axis steps because they are the basic building unit of any XML query. Our techniques are based on Dewey label identifiers, which have not been covered in detail in the literature. Our techniques can be applied to any Dewey-based labelling scheme even though the best results have been achieved using our version of Dewey labels: PoD-S.

The evaluation tests have shown that our Dewey-based PoD-S labelling technique (which is based on splitting the label into two components [parent and child] in Dewey format) has significantly improved the performance of the execution of XML queries. In addition, it outperformed the typical Dewey-based one-component label scheme. Further, PoD-S approach can be applied to other Dewey-based labelling schemes without sacrificing any of the Dewey label features (for example, supporting dynamic XML documents).

The tests also showed that our optimisation techniques for PoD and PoD-S, which exploit the schema summary in the XML_Path table, combined with a set of functions (such as xgp() function), have outperformed the traditional RTs for most of the XPath axis steps. We can conclude that the Dewey-based labels in general, and the PoD-S in particular, are far more efficient than the traditional number-based interval approach.

However, the ‘following’ and the ‘preceding’ axis steps are still considered quite challenging to the relational systems due to a large number of duplicate nodes in the final result for some queries. We believe the run time for these two axis steps can be improved by modifying the relational system algorithms for handling duplicate values.

Chapter 5: Optimising XML Structural-join Queries

XQuery, the W3C's standard query language for XML data, is a powerful declarative query language for XML data. XQuery goes beyond XPath in its ability to build complex queries that can manipulate the XML document structure and content. However, supporting and/or implementing these new features in relational databases require developing new algorithms and optimisation techniques that can efficiently support the tree structural nature of XML data. In this chapter, we investigate developing optimisation rules that make the relational systems more tree-aware without the need to modify the database kernel. Further, we discuss optimisation techniques for the expressive yet expensive structural-join queries using off-the-shelf-relational database. We report on extensive experimental studies including an evaluation study between our approach and established native and modified-relational XML database systems.

5.1 Introduction

The efficient storage and querying of XML data have become an important issue as the XML and related technologies, such as XQuery, have gained popularity in recent years. Effective query optimisation is crucial to obtaining good performance from an XML database. A join is frequently the most expensive physical operation in evaluating a relational query. Thus, selection of join order is a critical component for optimising a relational query. This observation is true for XML query optimisation as well. A join in the relational context is usually a value-based join, which involves two tables and is based on the values of two columns, one in each table. In the XML context, even though there are value-based joins, structural joins occur much more frequently. A structural-join query focuses on the containment relationship (i.e. ancestor-descendant or parent-child) of the XML elements to be joined. The join condition is specified not on the value of the XML elements, but on their relative positions in the XML document.

An XML data model is quite different from relational data model. Queries on XML data have some features that are different from queries on the relational data. Therefore, the query optimisation techniques in the XML context are also different.

While the state-of the-art approaches in both native and relational-based systems involve new join algorithms and indexing techniques that make them powerful and efficient [Al-Khalifa *et al.*, 2002; Boncz *et al.*, 2006; Bruno *et al.*, 2002; May *et al.*, 2004; Wu *et al.*, 2003], some of these approaches are not directly applicable to the relational database systems. Changing the relational system's kernel is hardly an option for many RDBMS vendors.

In this chapter, we discuss the various approaches to optimise XML queries with special focus on relational-based XML databases. We also highlight potential working areas to enhance XML query run time. Further, we present the PoD approach to utilise the current relational storage and indexing technologies to make off-the-shelf relational database more tree-aware by introducing new technique to select the most efficient join order. Moreover, we present PoD approach to enhance the performance of Twig queries by reducing the number of relations (i.e. tables) that are involved in join operations.

We report on query performance tests using XML benchmarks to evaluate the efficiency of our complete approach and compare the results with those of dedicated XML management systems (i.e. native systems and modified relational-based systems).

5.2 XML Query Optimisation Techniques

The efficiency of querying XML documents depends primarily on two factors: first, the way in which the XML documents are stored and indexed. Secondly, efficiency also depends on the query semantics and structure. Query optimisation techniques mainly target these two areas; however, query optimisation can occur at different levels starting from the query itself, and all the way to the physical storage structure of the data.

In general, the query optimisation is carried out based on:

- Query context and information obtained from the query itself;
- Views to which the queries are addressed;
- Integrity constraints;
- Schema of data, when it exists; and
- Data statistics that can be used in cost-based plan optimisation.

However, XML query optimisation techniques can also be platform or storage layout dependent; the two main platforms available for storing XML documents are the native and relational storage engines.

In this section, we discus optimisation directions that apply to both platforms as well as some of the platform-dependent optimisation directions.

5.2.1 Optimisation Based on Query Rewriting

Many user queries, web-based queries in particular, are not in the optimal format for many reasons; one of them can be lacking knowledge of the data schema and structure. Therefore, the need arises for rewriting the user queries into equivalent queries that will produce the same results. However, the equivalent queries will have more efficient execution plans, in most cases, since other optimisation rules can be applied to the equivalent queries. The rewriting process happens before the query being translated into a logical plan.

XQuery rewriting can take place at two different levels: XQuery core level and algebraic level. The XQuery core level makes use of the formal semantic approach [Draper *et al.*, 2007], which translates the query into more functional language representation where existing optimisation techniques for functional languages can be applied.

We can summarise the major areas that are covered by the rewriting process:

- XPath Rewriting: XPath is used in XQuery to identify the XML document nodes that are targeted in the query. Rewriting aims to remove complexity by including more explicit location steps. Or by removing reverse axis (i.e. parent and ancestors) since some systems cannot handle these types of axes (for example, stream data-processing systems) [Olteanu *et al.*, 2002].
- Order and duplicate removing: using formal semantic, XPath is translated to an XQuery core expression such that every step in the XPath is followed by a distinct doc-order operator. This maintains document order and removes duplicate but can be a very expensive operation and slow down system performance [Al-Khalifa *et al., 2002;* Fernandez *et al.*, 2005].

- Join Reordering: widely used techniques in databases that use algebra with joins [May *et al.*, 2004; Wu *et al.*, 2003]. For XML it requires the uses of order-preserving joins. However, these types of joins are associative but not commutative, which reduces the number of alternative plans that can be generated by the optimiser.
- Plan Simplification: use predefined equivalence rules to simplify the plan by pruning, for example some obsolete input columns and/or replace a join with a projection [Grust, 2005].

5.2.2 Implementation of New Join Operators

Due to the differences between the relational data model and the hierarchical XML data model, the use of the traditional relational query techniques will not efficiently address queries against XML data. Modifications, new algorithms, and new query operators have been proposed for both native and relational-based XML management systems.

Researchers in academia and industry have been working on optimising XML queries in native systems by developing new operators and new merge-join algorithms similar to those already exist in the relational technology. However, the new operators and join algorithms are designed from scratch to be more tree-aware. The DDO operator that forces distinct-decorder (document order) has been introduced to comply with XQuery formal semantics [Fernandez *et al.*, 2005]; the new operator ensures that intermediate results are sorted in document order and duplicate free. However, applying the DDO function after each step may be an expensive operation. A special optimisation plan is also used to decide when the DDO operator needs to be applied, which will enhance the query run time and ensure the correctness of the results.

Several new stack-based join algorithms have been proposed to handle structural and twig joins efficiently [Al-Khalifa *et al., 2002;* Bao *et al.*, 2008; Jiang *et al.*, 2007]. The twig-join techniques typically decompose a query into a set of binary patterns or single paths and then search for matches for these individual pattern/paths. Finally, these matches are stitched together to form the answers to the twig query. Most if not all of these approaches use stacks to capture the patterns and they all aim to reduce the size of intermediate results. Another twig-join approach is based on combining a new labelling technique based on Dewey labels and a new twig-join algorithm [Lu *et al.*, 2005]. The Dewey labels are extended to include information about the tag name and element order. It is a tree-aware labelling scheme that is used in a new join algorithm TJ Fast to reduce the cost of the twig-join queries. The TJ Fast can skip a significant number of nodes making use of the extended Dewey labels. However, this technique requires extra storage space (30 per cent increase). Further, it depends on the presence of a document schema.

New operators and join algorithms have also been proposed for relational-based XML storage systems to enhance XML document support within existing RDBMS [Boncz *et al.*, 2006; Dehaan *et al.*, 2003; Grust *et al.*, 2004; Li and Moon, 2001]. The Staircase join algorithm is a tree-aware join approach which requires modification to the RDBMS kernel to be aware of the tree nature of the XQuery [Grust *et al.*, 2003]. The Staircase join is based on the numeric interval labelling technique that partitions the document around any given context node into four areas: preceding, descendent, ancestor, and following. The pre/post-plane can be used to optimise XPath queries:

- Using pruning: remove redundant nodes when evaluate each step.
  - Example: a node v2 is a potential descendant of node v1 if the following holds:

    ```
    v2.pre <= v1.post+h and v2.post>= v1.pre+h
    ```

    Where h: is the tree height.
    Pre: is the node’s order when visited for the first time.
    Post: is the node’s order after visiting all its descendant nodes.
- Using staircase join rather than normal join; only join potential nodes based on the valid partitions for that query and can be enhanced by skipping empty regions.

Further, the major relational database vendors have introduced new XML data type, functions, and operators to support XML documents [Beyer *et al.*, 2006; Liu *et al.*, 2005; Pal *et al.*, 2005b]. The following sections contain more detailed discussion about the XML query support and optimisation in relational database systems.

5.2.3 New Labelling and Indexing Techniques

Labelling XML document nodes were initially proposed to capture XML document order. However, recent labelling techniques such as containment labelling and Dewey-based labelling schemes can also provide efficient support for structural queries.

However, these labelling techniques have limitations because they cannot efficiently support all types of structural queries. The containment labels can support ancestor-descendent relationship but not sibling or parent-child relationships [Yoshikawa *et al.*, 2001]. Enhancements were proposed such as storing the level information, which would provide support for more XPath axis steps such as child axis step [Li and Moon, 2001; Zhang *et al.*, 2001], and ‘Preceding’ and ‘Following’ axis steps [Grust *et al.*, 2004]. However, that will slightly increase the storage requirements and still cannot address the requirements of dynamic XML documents.

Conversely, the Dewey labels can efficiently support the parent-child relationship [O'Neil *et al.*, 2004; Tatarinov *et al.*, 2002]; however, using functions to process labels may cause the inefficient use of current relational indexing technology that is available in off-the-shelf relational database systems.

Neither technique on its own can provide efficient support for the complex structural queries, such as twig queries and chain structural queries. Other labelling and indexing approaches were proposed to encode more data in the label id itself, either tag names or data values, to provide more information about the document structure and data values [Chen *et al.*, 2007; Silberstein *et al.*, 2005; Weigel *et al.*, 2005; Xu *et al.*, 2009]. However, these techniques increased the level of complexity and reduced efficient support for dynamic XML documents. Further, they required more storage space than traditional labelling schemes and it might be difficult task to implement them in relational database systems.

Recently, researchers became more aware of importance of other document structural information, which is called document schema summary [Moro *et al.*, 2008]. This information needs to be integrated with the labelling approach to develop more efficient techniques to support twig queries without sacrificing the space, which is part of our contribution in this chapter.

5.2.4 Effect of Indexing Data Values

Many XML queries include predicates on values, which represent a challenge for XML data management systems. However, indexing the data values is a widely used approach [Pal *et al.*, 2004]. In some cases, the index size might become a concern that affects performance. We believe this part is highly application dependent and it would be more effective if the most frequently queried element values are indexed. Further, more than one index might be also needed depending on the values that are covered by that index.

Partitioning the data value table based on the tag name (horizontal partition) can also help improving query performance, and that has been proved in some studies [Florescu and Kossmann, 1999].

Data value indexing is beyond the scope of this thesis.

5.2.5 XML Schema to Relational Schema Mapping Techniques

Motivated by the reliability and capabilities of the relational systems that have evolved over three decades, many research efforts have focused on storing XML data in RDBMS. Recently, the major RDBMS vendors have introduced the new 'XML' data type, which is similar to the CLOB and BLOB data types [Beyer *et al.*, 2006; Lee, 2003; Pal *et al.*, 2005b]. The new data type allows storing an XML document as one record in a traditional relational field, which is considered as the most effective method to retrieve the whole original document. However, to effectively answer queries against data within the document, the process of shredding the XML document and labelling its individual nodes still cannot be avoided.

The approaches based on shredding XML documents can be classified into two major categories:

- Model-mapping approach: database schemas represent constructs of the XML document model. In this approach, a fixed database schema is used to store the structure of all XML documents.
- Structure-mapping approach: database schemas represent the logical structure (DTDs [Bray *et al.*, 2008] or XSchema [W3C, 2004] if they are available) of target XML documents. In a basic design method, a relation is created for each element type in XML documents. In the structure-mapping approach, a database schema is defined for each XML document.

5.2.5.1 Model-Mapping (Fixed Relational Schema)

The basic Edge approach [Florescu and Kossmann, 1999] is the most common approach; it is a simple approach that places all the edges in an edge-labelled XML data tree into a single relational table. The schema of the basic Edge approach would have the following definition:

```
Edge = (Label: String, Source: Integer, Target: Integer, Flag: String,
        Value: String)
```

The pair (S*ource*, *Target*) represents the two end points of an edge. *Label* represents the tag name, whereas *Flag* and *Value* give the type and value, respectively, of the target node of an edge. The Edge labelling technique has limited capabilities and can efficiently support only parent-child queries '/' and sibling queries.

```
      Node1.Target = Node2.Source   (Parent-Child Relationship)
      Node1.Source = Node2.Source   (Sibling Relationship)
```

Despite that fact that other labelling techniques can significantly enhance the Edge approach's capabilities to answer ancestor-descendent ('//') and twig queries; its major drawback is that all nodes are stored in one table, which means that the evaluation of XML queries would involve accessing a large amount of unrelated data. Further, it requires a high number of join operations to process some queries; for example, the number of required join operations is the number of nodes (edges) in a twig query minus 1.

The Binary approach was also evaluated in Edge study; in the binary approach, every edge name ('Label') is stored in its own table. This means only related data would be included during query evaluation. The evaluation tests for the basic Edge approache showed that binary approach outperformed the one-table Edge approach [Florescu and Kossmann, 1999]. However, reconstructing the original XML document or major parts of it would be very expensive process.

To reduce the number of join operations, a path-based approach was proposed in [Yoshikawa *et al.*, 2001]. It stores all the possible path expressions in a given XML document in a separate relational table. The path expression for any node is the concatenated labels, tag names, from the root node to the context node separated by '/'. Each path expression is allocated a unique path id that will be used as a foreign key in the Edge table to replace the 'Label' field. Further, it stores the actual data instances of elements, attributes, and text nodes in three separate tables. Various combinations of the

Path table and other data tables have been proposed [Jiang *et al.*, 2000; Schmidt *et al.*, 2000]. This approach can significantly reduce the number of required joins for twig queries and queries that involve '//' axis. However, it may not return correct results for recursive XML schema.

An enhanced Path approach was proposed in Pal *et al.* [2004], which is based on using reversed path expressions. The reversed path approach can support recursive XML schema.

5.2.5.2 Structure-Mapping (DTD Dependents)

In many practical applications, XML data also conforms to an XML 'schema' (such as DTD or XSD). The existence of such schema information provides an opportunity for more compact storage and efficient querying of XML data [Tian *et al.,* 2002]. Unlike the schemaless approaches (such as Edge and Path), which generate the same relational schema for all types of XML data regardless of their structure, the Structure-Mapping approach generates a different relational schema for a different XML document schema (i.e. different DTDs) [Shanmugasundaram *et al.*, 1999].

This technique is the most efficient technique to support data-centric queries since it will require the least number of I/O operations among all other approaches as well as least storage size. This approach relies on the existence of the XML schema (DTD or XSchema); the XML schema is analysed and the required and optional items are identified. The idea is to inline as much as possible child nodes with their parent nodes as one tuple in the same table. After retrieving the parent node, retrieving its attribute nodes and child leaf nodes can happen at minimal cost since they will be stored on the same physical page. The Other child element nodes will overflow into other tables with references to the parent nodes [Deutsch *et al.,* 1999; Lee *et al.,* 2003; Shanmugasundaram *et al.*, 1999].

This technique is more suitable for XML documents that were originated from relational data, or for applications with more structured data rather than semi-structured data with the existence of the XML schema. Further, this technique is highly dependent on the XML schema and any simple change to the XML schema may trigger significant changes to the relational schema that might affect the optimisation methods.

The cost of reconstructing the original XML document from the shredded tables will be much higher than the first approach; however, this approach is the best approach to take full advantage of the features that exist in the powerful relational systems

5.2.5.3 XML Data Type

XML data type, similar to CLOB data type, is a very recent data type that allows storing the whole XML document in its native format as one record in relational database systems. The main advantage of this technique is the efficient retrieval of the original XML document, since no recomposing or rebuilding is required.

Special indexes can be built on the new data type, which support document-centric queries and some data-centric quires without the need to parse the original XML document during run time. However, it is hard to cover a broad range of queries without going back to the document and parse it during run time.

The main concern with this technique is the storage requirements; the storage size would be twice if not more than the size of the original XML documents. Further, updating operations can be very expensive, since any slight modification to the original XML document requires the whole XML document to be restored in the database.

This technique is more suitable for small-size XML documents in applications where the whole document or large portions of it are more likely to be retrieved, and where the full XML document can be loaded in memory at minimal cost.

Recently, major database vendors are providing more than one technique, including the shredding option, to allow users chose the most appropriate storage option for their applications [Oracle, 2009].

5.3 PoD Relational Schema

We have discussed in the background section the different approaches to build a relational schema for an XML document; the fixed-map approach has some advantages over the structural-mapping approach. The fixed-map approach reduces the cost of supporting dynamic XML documents and applications where the XML schema is absent or not required.

The focus of this research project is to enhance the efficiency of relational-based XML database systems by introducing more efficient labelling techniques and by introducing more efficient query optimisation techniques, which can be portable as well as integrated with the existing relational technologies.

We have followed the fixed-map approach, which allows us to evaluate the effect of our proposed techniques directly and isolate the effect of other factors like having the ultimate relational schema design.

We used our XML_Path table, discussed in Chapter 4, combined with two mapping approaches for the various tests in this thesis. First, we used the Edge technique where all nodes are stored in one table. Second, we created a table for each node type (for example, Elements, Attributes and Text).

5.3.1 The Edge Approach

The Edge approach is a simple approach that has been widely used by researchers, as we discussed earlier in Section 5.2.5.1. Following is the schema definition for this approach:

```
Edge for PoD = {id: binary string, path_id: integer, value: string}
```

Where: the 'id' is the Dewey label id in PoD format.

Path_id: is an integer value to distinguish the path expression of each node; path _id is foreign key from the XML_Path relation. A B-tree index is built on the (id) column as a primary index.

And for the Split-Mode

```
Edge for PoD-S = {Pid: binary string, Cid: binary string, path_id:
                    integer, value: string}
```

Where: 'Pid' and 'Cid' are the parent and child (self) components of the Dewey label id. A B-tree index is built on the (Pid, Cid) columns as the primary index.

Figure 5-1 shows the complete enhanced relational schema for this mapping approach.

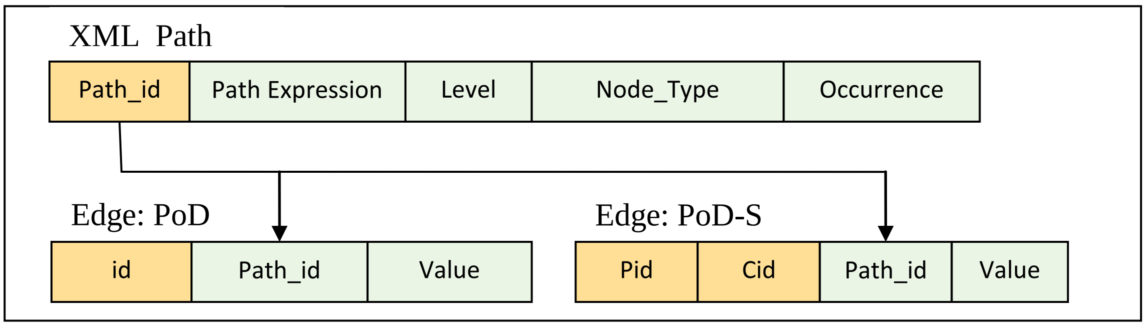


Figure 5-1: Edge relational schema in PoD and PoD-S.

5.3.2 The Node Type Approach

Having all nodes in one table may have some disadvantages for the query execution and mainly for value-join queries since a large amount of irrelevant data might be involved. Storing each node type in a separate table would help reduce the effect of that problem [Yoshikawa *et al.*, 2001]. However, the ultimate solution for this issue would be storing each node name (tag) or each path_id in a separate relation. In this approach, we used our XML_Path table with the following relations:

```
Element = {id: binary string, path_id: integer}
```

```
Text = {id: binary string, path_id: integer, value: string}
```

The element table contains only parent nodes, and the Text table contains text nodes, leaf nodes and their values.

```
Attribute = {id: binary string, path_id: integer, value: string}
```

The primary index is based on the id column, and the path_id is a foreign key in the three tables. Figure 5-2 shows the complete relational schema for this approach.

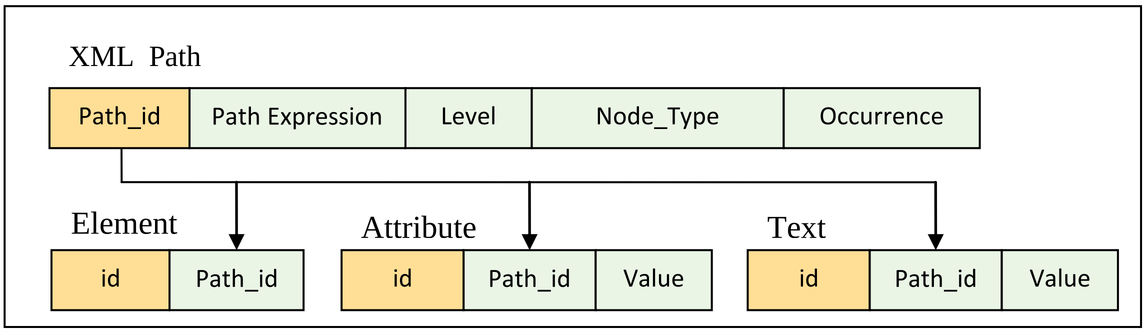


Figure 5-2: Node type relational schema in PoD.

The same schema is applied to PoD-S with a slight modification: the id field is replaced by two columns: ‘Pid’ and ‘Cid’.

5.4 Finding Optimal Join Order in the PoD System

Several studies have reported that some XML queries run very slowly in relational-based management system even though the query was expected to run much faster than that. Query analyses have shown that in some cases the query optimiser and processor in the relational database system generates an execution plan with a certain join order that would generate a large amount of intermediate results for some queries, which means more I/O operations and slower query run. The query processor’s behaviour can be explained within the scope of the fundamental differences between the two data models. Figure 5-3 shows how RDBMS may generate a join order that might not be the optimal join order and can result in a slower execution plan.

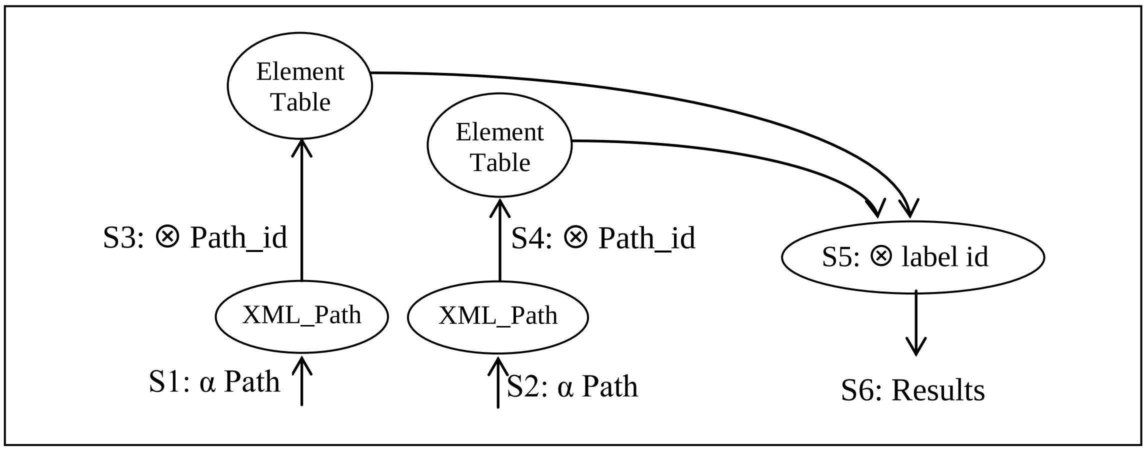


Figure 5-3: Join order that might result in a slower execution plan.

It has become very well known that the relational systems need to be more aware of the tree data model of XML data. One of the proposed solutions is to modify the relational systems kernel to be more tree-aware by introducing new operators and modifying the join algorithms [Grust *et al.*, 2003]. The other approach, which is the motivation for this thesis, is developing techniques that would bridge the gap between the two data models without the need to change the relational systems kernel.

We have observed that forcing a certain join order can achieve consistent high performance query runs without the need to change the database system kernel. Forcing certain join orders can be considered to make the relational system a more tree-aware system. In this section, we discuss rules that can be applied to determine the join order that will produce the best query performance in most cases, if not in all.

The basic idea here is to reduce the number of I/O operations and ensure efficient use of the indexes. We need to make efficient use of the label id index in particular, which also represents the document structure and inter-nodes relationships. This is quite important in the case of twig quires and complex queries that contain predicates.

The flowchart in Figure 5-4 shows a more efficient join order for a typical twig query that has a faster execution plan in most cases. Using this join order, the relational systems become more tree-aware without the need to change the systems kernel.

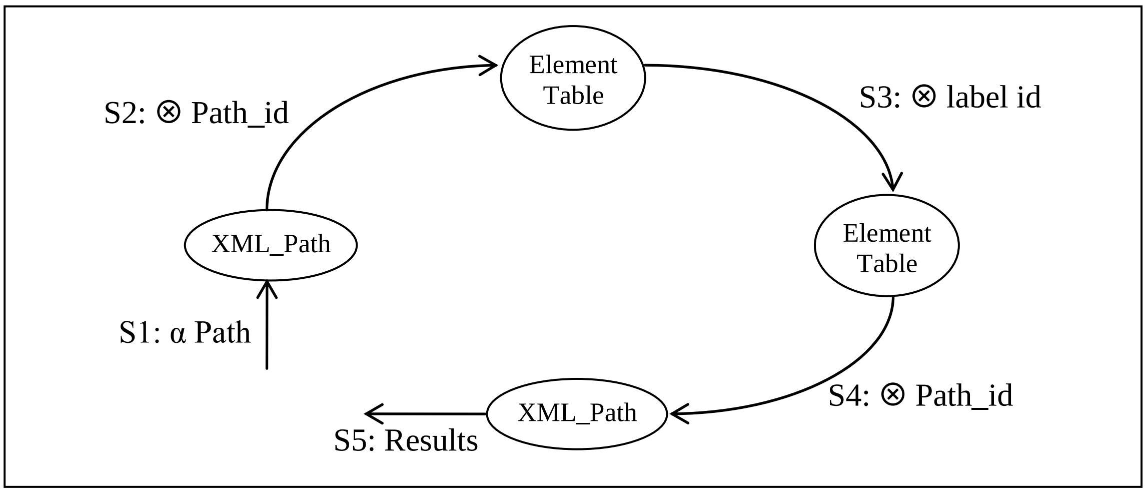


Figure 5-4: A more efficient join order for XML queries.

Figure 5-4 shows the join flow for a simple twig-join, which is represented in stages from S1 to S5 for clarification, as follows:

S1: one path expression is evaluated from the XML_Path table.

S2: the results are joined with the first instance of element_table using the foreign key path_id.

S3: the first instance of element_table is joined with the second instance of element_table (self-join) using the label id to evaluate the values for the second path expression (for example, find all siblings and find parent).

S4: the second instance of element_table is joined with the second instance of XML_Path table using the foreign key path_id to include only the nodes that match the second path expression.

However, some other factors should be taken into account when that join order is decided:

Which path expression should we start with?

What will happen if there is a value predicate?

What order should we have if we intend to use xgp function?

The XML_Path table, which serves as a schema summary or data guide, provides extended information about each node like level and occurrence, which can be used to make more appropriate decisions in terms of join order. We suggest that the XML query should be broken down to its basic path expressions and consult the XML_Path table for each path expression to gather information like existence, level, number of occurrence and relationship with other path expressions in the query. Based on this information, we can drop redundant path expressions, and simplify the query by pushing operators and functions down or up the tree. This phase should precede the query translation. Following are rules that can be used to decide the join order:

Proposition 5-1: *For a twig query of two path expressions with no value predicates, we start with the path expression of less occurrence value.*

Proposition 5-2: For *a twig query of two path expressions with a value predicate and recursive schema, we start with evaluating the data value if there is a data value index and then join back on the XML_Path table before continuing with the rest of the tables. In the absence of data value index, proposition 5-1 applies.*

Proposition 5-3: *The path expression of the common ancestor node that provides the level information for xgp function needs to be evaluated before the element table on which it will be applied.*

Example 5-1: return the book titles for all books that were published in 2005:

/BookList/Book[Year = 2005]/Title

We can identify three path expressions as follows:

P1 = /BookList/Book

P2 = /BookList/Book/Year

P3 = /BookList/Book/Title

From the query semantics, we can discard P1 because we can ensure that P2 and P3 are sibling nodes by evaluating their label id values. Moreover, based on Propsition 5-2, we start with P2. The full translation and join order is as follows:

```
Select Straight_Join t3.results
From XML_Path t2, element_table t1, element_table t3, XML_Path t4
Where t2.path like ‘/BookList/Book/Year’
 and t4.path like ‘/BookList/Book/Title’
 and t2.path_id = t1.path_id
 and t4.path_id = t3.path_id
 and t1.value = ‘2005’
 and t1.Pid = t3.Pid
```

Some special considerations need to be taken into account if the XML document is recursive, or if it contains similar tag names at different levels and locations. This implies a slight change to the join order to start with the value index rather than the path_id index as follows:

```
From element_table t1, XML_Path t2, element_table t3, XML_Path t4
```

5.5 PoD Approach for Optimising Twig Queries

The XQuery language is as powerful as the SQL query language that allows complex queries beyond the basic XPath axis steps queries. One of the major types of XML queries is the twig-join queries, which search for the occurrence of certain tree patterns rather than individual nodes. The twig-join queries are very expensive to evaluate since they normally produce large intermediate results.

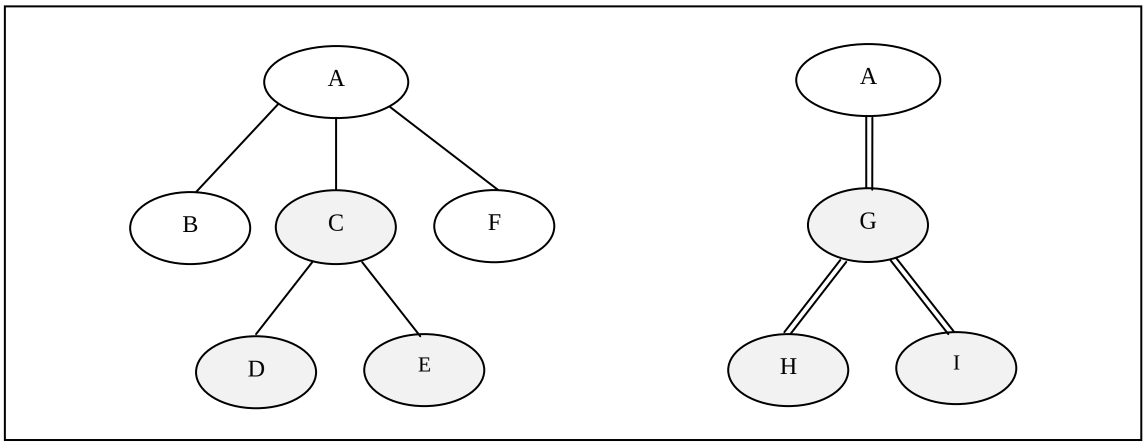


Figure 5-5: Sample XML twig queries that search for the occurrence of the shaded node patterns.

The shaded node patterns in Figure 5-5 represent sample twig-join queries; the pattern (C, D, E) in the tree on the left side represents a simple but common twig-join query that contains parent-child and sibling relationships, in which the pattern (G, H, I) in the tree on the right side represents the twig-join query with ancestor-descendent relationships. Moreover, the twig-join queries can be a mixture of both cases and they may include more branches in some complex queries.

As mentioned earlier in this chapter, new join algorithms have been proposed to handle and improve the performance of the Twig-join queries in native XML management systems and relational database systems. However, these techniques require changes to the relational systems that might not be an easy option for many database vendors and it might affect portability.

In this section, we present a technique that will substantially improve the execution of twig queries in off-the-shelf relational systems and without the need to modify the database system kernel.

5.5.1 Sibling-based Twig Queries

In many real applications, it might be a common query to find a certain XML document element that has another sibling element with certain conditions. In other words, for node E to be included in the result set, another node D must exist in which the parent's id of E equals the parent's id of D.

Applying the range encoding technique for Dewey labels will result in three costly joins on the element_table, as follows:

```
Select t5.results
From element_table t1, xml_path t2, element_table t3, xml_path t4,
      element_table t5, xml_path t6
Where t2.path like '//C'
 and t4.path like '//C/D'
 and t6.path like '//C/E'
 and t2.path_id = t1.path_id
 and t4.path_id = t3.path_id
 and t6.path_id = t5.path_id
 and t3.id between t1.id and xmc(t1.id)
 and t5.id between t1.id and xmc(t1.id)
```

However, using the PoD-S technique with the extended XML_Path table, the sibling-based queries can be reduced to two self-joins on the element_table rather than three self-joins with a more efficient use of the label id index as follows:

```
Select t3.results
From element_table t1, xml_path t2, element_table t3, xml_path t4
Where t2.path like '//C/D'
 and t4.path like '//C/E'
 and t2.path_id = t1.path_id
 and t4.path_id = t3.path_id
 and t3.Pid = t1.Pid
```

5.5.2 Ancestor-descendent Twig Queries

The ancestor-descendent twig-join queries are among the most expensive query structures, which normally generate large intermediate results. The main objective of all optimisation techniques and the new proposed join algorithms in the literature is to reduce the size of the intermediate results. This means reducing the number of joins in a relational database system and pushing some of the projection operators down the query execution plan [Seah *et al.*, 2007].

Proposition 5-4: *for any two nodes N2 and N3 at any two equal or different levels, both nodes are descended from the node N1 if:*

$$(N_1 < N_2 < xmc(N_1)) \text{ and } (N_1 < N_3 < xmc(N_1))$$

Where xmc() function returns the maximum label value for any child that descends from node N_1.

This translation is still faster than the numeric-based intervals because the later requires evaluation on a larger number of different values of the 'pre' and 'post' values of each node involved in that pattern.

Using the range technique (RT) above to evaluate the tree pattern in Figure 5-3(b) will produce the following SQL query:

```
Select t5.results
From element_table t1, xml_path t2, element_table t3, xml_path t4,
      element_table t5, xml_path t6
Where t2.path like '//G'
 and t4.path like '//G//H'
 and t6.path like '//G//I'
 and t2.path_id = t1.path_id
 and t4.path_id = t3.path_id
 and t6.path_id = t5.path_id
 and t3.id between t1.id and xmc(t1.id)
 and t5.id between t1.id and xmc(t1.id)
```

However, this approach might be very expensive since it involves three joins (or three self-joins) on a very large table(s). Based on the fact that the Dewey-based label for any given node does contain the labels of its parent and ancestor nodes, the xgp() function can be used to produce an optimised alternative translation that eliminates one expensive join to evaluate the id(s) of N_1 as follows:

Proposition 5-5: *for any two nodes N2, and N3 at any equal or different level, both nodes are descended from the node N1 if:*

$$xgp(N_2, level(N_1)) = xgp(N_3, level(N_1))$$

The above rule can be further optimised for efficient use of the label index as follows:

N_2 BETWEEN xgp(N_3, level(N_1)) AND xmc(xgp(N_3, level(N_1)))

The sibling-based twig query is seen as a subset or a special case of the ancestor-descendant scenario in which both nodes are at the same level and their nearest enclosing node is their parent node; the above formula can be reduced to:

N_2 BETWEEN xp(N_3) AND xmc(xp(N_3))

This can be evaluated more efficiently in PoD-S mode as follows:

$$Pid_{N2} = Pid_{N3}$$

We can now rewrite the SQL query for the tree pattern in Figure 5-3-b with a lower number of joins as follows:

```
Select t5.results
From xml_path t2, xml_path t4, element_table t3, element_table t5,
      xml_path t6
Where t2.path like ‘//G’
 and t4.path like ‘//G//H’
 and t6.path like ‘//G//I’
 and t4.path_id = t3.path_id
 and t6.path_id = t5.path_id
 and t5.id between xgp(t3.id,t2.level) and xmc(xgp(t3.id, t2.level))
```

We can also add the string match statement to reduce the size of the intermediate results as follows:

```
 and t4.path startsWith(t2.path)
```

This might be helpful in recursive XML schemas or when there are similar tag names at different levels of the XML tree.

5.6 Minimising XML Queries

XQuery, like SQL, is a powerful structural query language; however, many XML queries on the Internet may not be in optimal form due to different reasons, such as lacking knowledge or missing the document schema. Rewriting queries in a more efficient form, or at least removing redundant path expressions, will improve the performance of the query run by reducing the number of join operations.

In this section, we propose a technique called XPath Matrix, which would eliminate redundant paths in the query and provide alternative execution plans at a high level. We believe this technique can be used for either relational-based or native-based XML management systems. However, we are focusing on relational-based systems.

5.6.1 Building the XPath Matrix

The basic idea is to parse the XML twig query (either XPath or XQuery) and build a matrix that contains all the paths in the query and information from the XML_Path table about each path. The matrix can be used to determine relationships between paths and to identify which paths can be discarded without affecting the query results.

	P1	P2	…	Pn	Conditions	Result	Occurrence
P1	-						
P2		-					
…							
Pn				-			

Figure 5-6: Initial XQuery and XPath query path matrix.

Figure 5-6 shows the suggested matrix in which P1, P2 and Pn are the distinctive paths in the query. The matrix cells will have different key information about each path as follows:

- Condition column: this indicates the type of condition on the corresponding path:
    - 0: no conditions.
    - 1: must have a condition (like value predicates) or it is part of the results.
    - 2: condition that may be pushed to another node in the matrix and discard the original path.
    - 3: this path might be required to provide level information for xgp() function.
- Result (Yes/No): indicates whether this path is part of the final result set.
- Occurrence (Integer): indicates how many instances of this path are in the element table.

- The intersection cells between paths hold information about the relationship between each pair of paths as follows:

  P = P1 is Parent of P2.
  C = P1 is Child of P2.
  S = P1 is Sibling of P2.
  D = P1 is Descendant of P2.
  A = P1 is Ancestor of P2.
  GC = P1 and P2 are descendents of the same ancestor node.
  VE = P1 has value equivalency condition with P2.

Example 5-2: The query (/A[B = ‘value’]/C) is a typical twig query. We can extract three paths from this query:

P1 = /A
P2 = /A/B and B = ‘value’
P3 = /A/C (result)

The initial XPath matrix for this query is shown in Figure 5-7:

	P1	P2	P3	Conditions	Result	Occurrence
P1	-	P	P	3	-	Number
P2	C	-	S	1	-	Number
P3	C	S	-	1	Yes	Number

Figure 5-7: Initial XPath matrix for query in Example 5-2.

5.6.2 Minimising the XPath Matrix

The first phase of our technique is to scan the XPath matrix, remove redundant paths or paths that are not required (mainly parent and ancestor paths) to produce correct results, and identify the paths to be joined together by creating a Join_List.

We would like to introduce some XML query features that can be used in the query minimisation and translation algorithms to reduce the number of joins between tables and to select the most efficient join options.

Cyclic Structural Join: several complex twig queries may contain redundant join entries, referred to as cyclic structural-join entries; Figure 5-8 shows an example of a cyclic structural join.

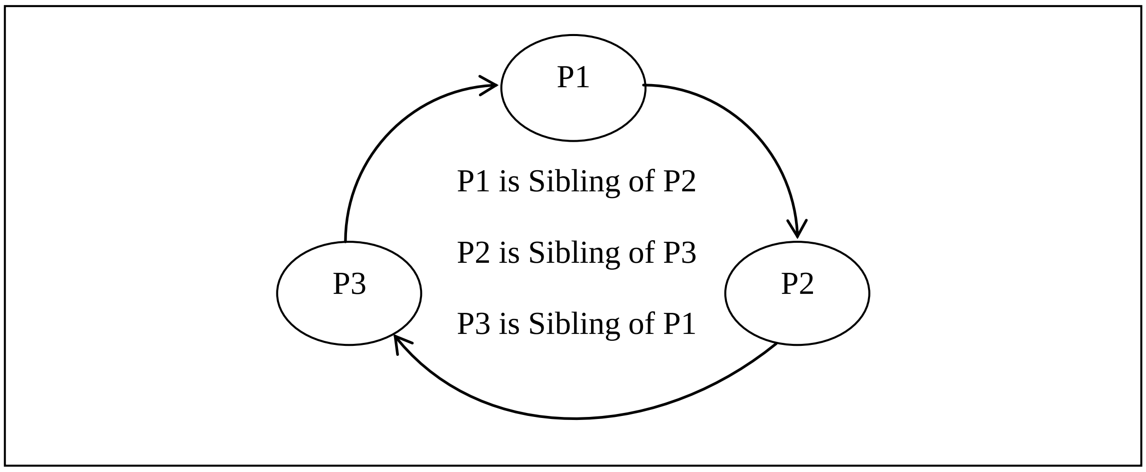


Figure 5-8: An example of cyclic structural relationship joins.

One of these join-entries in Figure 5-8 can be removed without affecting the correctness of the result. This kind of optimisation can be applied to join operations that are based on the S and GC relationships.

Priority of Join Operations: When it is decided to remove one or more join operations, it is important to know that some join-operations between path expressions are either essential or can be evaluated faster than other relationships in the PoD (for example, a sibling relationship is faster to evaluate than any other relationships). Table 5-1 summarises the priority of the XML join operations; the entries in the table are in descending order based on the priority level.

Table 5-1: Join relationship priority.

Join Relationship
VE
S
GC
Lower Occurrence

Note: the last row represents the number of occurrences for a certain path, which can be used when the join relationships are at the same level, which can also be used to determine the join order

XML Functions and Conditions: some of the XML conditions in the query can be pushed either down or up the query tree, which leaves the original path expression redundant and able to be removed from the join_list. Following are primarily rules that can be used to move conditions to other paths that already exist in the query and reduce the number of required path expressions:

- The order of node N_{11}, which is a child of node N_1, is *BEFORE* the order of N_{21}, which is a child of node N_2, if N_1 is before N_2. The same is true for the operator *AFTER*.
- For a given query, the Count of Node N_1 is equal to the count of a child Node N_c that meets a certain condition that is used to filter the result of Node N_1.
- To retain a self node and its children: one path string can be used to search the path table and filter the results by level and node type.
- To retain a self node and its children: one path string can be used to search the path table and filter the results by level only to avoid recursion.
- The absolute order of node N can be evaluated in a subquery.

The Join_List would look like a table with four columns, as in Figure 5-9; the fourth column is only required with the join relationship of type GC (both P1 and P2 are descendants of P3). This is required to provide the level information for the xgp function.

Path	Path	Join Relationship	Level Path for GC
P1	P2	S	-
P1	P2	GC	P3

Figure 5-9: Sample Join_list based on the query in Example 5-2.

5.7 Putting It All Together

In this section, we introduce algorithm outlines that can be a base for more advanced algorithms to minimise and translate XML query into an optimised SQL query. The algorithm outlines include the minimisation and optimisation techniques in this chapter and Chapter 4. We separate these techniques into three algorithms: minimising XML query; generating join list; and translating to SQL query in sequence, in which the output of each is the input of the following one. Moreover, the first two algorithm outlines can also be used in native XML database systems.

Algorithm 5-1: Minimizing XPath Matrix

Input: Initial XPath Matrix

Output: Minimised XPath Matrix

```
01: For each path in XPath Matrix
02:  identify relationship with all other paths:
   // (e.g. if P1:P:P2, P1:P:P3, then P2:S:P3)
03: End loop
04: For each condition in Condition column
05:  if (condition == 2) And (condition can be moved)
06:   move condition to another node in the query
07:   condition = 0
08:  End if
09: End loop
10: For each path in XPath Matrix
11:  if (condition == 0)
12:   Remove path;
13: End loop
14: End
```

Figure 5-10: Algorithm outline to minimise XPath matrix.

The next algorithm has the minimised XPath matrix as input; it adds join entries to the Join_List for related paths in the matrix. Finally, the Join_List will be scanned and redundant join entries will be removed.

Algorithm 5-2: Generating Join_List

Input: Minimised XPath Matrix
Output: Join List Table

```
01: For each path $P_i$ in XPath Matrix
02:  For each path $P_j$ in XPath Matrix
03:   If (($P_i$ relationWith $P_j$) == S, VE, or GC)
04:    add to Join_List ($P_i$ join $P_j$)
05:   End if
06:  End loop
07: End loop
08: For each entry in Join_List
09:  Search for cyclic join
10:  If (cyclic join exists)
11:    remove the most expensive join
12:    update the join_list accordingly
13:  End if
14: End loop
15: Sort the Join_List
// using Table 5-1
16: Reorder the elements of each join entry
// using rules in Section 5.4
17: End
```

Figure 5-11: Algorithm outline to generate optimised join list.

The last part has the ordered join_list as input and it generates the appropriate SQL statements. As a result of applying the three algorithms (minimising, optimising, and translation), the equivalent SQL statements are expected to be optimal.

Algorithm 5-3: Generating SQL Query

Input: Join_List

Output: SQL Query

```
01: For each entry in Join_List
    //Generate SQL equivalent statements
02:  Switch (join relationship)
03:   Case S:
04:    Add_to_FROM (‘XML_Path t_i, Elements t_j,
            Elements t_k, XML_Path t_n’)
05:    Add_to_WHERE (‘t_i.path like ‘P1’
           and t_n.path like ‘P2’
           and t_j.Pid = t_k.Pid’)
06:   Case GC:
07:    Add_to_FROM (‘XML_Path t_a, XML_Path t_i,
            Elements t_j, Elements t_k,
            XML_Path t_n’)
08:    Add_to_WHERE (‘t_i.path like ‘P1’
           and t_n.path like ‘P2’
           and t_a.path like ‘P3’
           and(t_k.Pid between xgp(t_j.Pid,t_a.level)
             and xmc(xgp(t_j.Pid,t_a.level)))’)
09:   Case VE:
10:    If (paths do not exist in the From clause)
11:     Add_to_FROM (the required XML_Path and Element
             tables)
12:     Add_to_WHERE (the required path
             string-match statements)
13:      End if
14:      Add_to_Where (‘and tx.value VE ty.value’)
              //VE: can be <, >,=,<>…etc
15:  End Switch
16: End Loop
17: End
```

Figure 5-12: Algorithm outline to generate optimised SQL query.

The algorithm above focuses on building the optimised structural-join part of any given XML query; there are other parts to be added to the translated query; however, they are straightforward and have been covered in the literature.

Example 5-3: We demonstrate our optimisation and translation technique using query Q4 from the XMark benchmark.

Q4 (XMark benchmark): List the reserves of those open auctions in which a certain person issued a bid before another person.

```
FOR  $b IN document("auction.xml")/site/open_auctions/open_auction
WHERE $b/bidder/personref[id="person18829"] BEFORE
    $b/bidder/personref[id="person10487"]
RETURN <history> $b/initial/text() </history>
```

We can break down this query into different paths as follows:

P1 = /site/open_auctions/open_auction

P2 = /site/open_auctions/open_auction/bidder/personref

P3 = /site/open_auctions/open_auction/bidder/personref/id

P4 = /site/open_auctions/open_auction/bidder/personref

P5 = /site/open_auctions/open_auction/bidder/personref/id

P6 = /site/open_auctions/open_auction/initial

Condition1 = P3 value = ‘person18829’

Condition2 = P5 value = ‘person10487’

Condition3 = P2 Doc order < P4, which means Dewey label id of P2 < P4 label id

Figure 5-13 shows the initial XPath Matrix for this query example (we have ignored the occurrence field because it can be used when statistics about real data are available).

After applying the first phase of the algorithm, we identify the relationships between other paths and check the conditions of P2 and P4, and whether it can be pushed to a query leaf node. The document order constraint on P2 and P4 can be safely pushed to P3 and P5 since:

```
P3 BEFORE P5 if and only if P2 BEFORE P4
```

	P1	P2	P3	P4	P5	P6	Conditions	Result
P1	-	A	A	A	A	P	3	-
P2	D	-	P	-	-	-	2	-
P3	D	C	-	-	-	-	1	-
P4	D	-	-	-	p	-	2	-
P5	D	-	-	C	-	-	1	-
P6	C	-	-	-	-	-	1	Yes

Figure 5-13: Initial XPath Matrix for XML query in Example 5-3.

In addition, the occurrence of P3 implies the occurrence of P2 and the occurrence of P5 implies the occurrence of P4. Therefore, P2 and P4 can also be removed from the XPath matrix since there are no other conditions on them. Figure 5-14 demonstrates the minimised XPath Matrix.

	P1	P3	P5	P6	Conditions	Result
P1	-	A	A	P	3	-
P3	D	-	GC	GC	1	-
P5	D	GC	-	GC	1	-
P6	C	GC	GC	-	1	Yes

Figure 5-14: Minimised XPath Matrix for XML query in Example 5-3.

Applying the first phase of the algorithm will also produce a join_list as in Figure 5-15.

Path	Path	Join Relationship	Level Path for GC
P3	P5	GC	P1
P3	P6	GC	P1

Figure 5-15: The join_list for XML query in Example 5-3.

Applying the second phase of the algorithm will translate the XML query into an optimised SQL statement as follows:

```
SELECT   straight_join t61.value
FROM     xml_path  t1,  xml_path  t3,  elements  t31,  elements  t51,
         xml_path t5, elements t61, xml_path t6
WHERE t1.path = '#/site#/open_auctions#/open_auction#'
  AND t3.path =
    '#/site#/open_auctions#/open_auction#/bidder#/personref#/@person#'
  AND t5.path =
    '#/site#/open_auctions#/open_auction#/bidder#/personref#/@person#'
  AND t6.path = '#/site#/open_auctions#/open_auction#/initial#'
  AND t3.path_id = t31.path_id
  AND t5.path_id = t51.path_id
  AND t6.path_id = t61.path_id
  AND t31.value = 'person18829'
  AND t51.value = 'person10487'
  AND t31.id < t51.id
  AND t51.id BETWEEN xgp(t31.id,t1.level) AND
         xmc(xgp(t31.id,t1.level))
  AND t61.id BETWEEN xgp(t31.id,t1.level) AND
         xmc(xgp(t31.id,t1.level))
```

5.8 Experimental Evaluations

The experimental studies in this chapter have two major directions:

- Evaluating the optimisation techniques proposed in this chapter with a special focus on XML Twig queries and with a special focus on the join order effect using off-the-shelf relational database systems.
- Evaluating the efficiency of storing XML documents in off-the-shelf relational database systems using our proposed techniques on top of the schema mapping and labelling techniques that have been developed over nearly a decade, and comparing that to the efficiency of dedicated XML management systems.

While the experimental evaluation has been conducted by translating XML queries into SQL statements, we believe some of the proposed techniques can be used or implemented in other XML management systems (such as native and hybrid systems).

The reason for using SQL is to evaluate the efficiency of unmodified relational database systems for supporting XML documents.

5.8.1 Experiment Setup

We have implemented PoD and PoD-S, as mentioned in Chapter 3. MySQL server 5.1 was used as a backend relational storage system. We have also implemented range encoding as mentioned in the literature [Yoshikawa *et al.*, 2001], using a node identification key (pre, post, level). Apache Java parser was used for parsing and loading XML documents. Java was also used for automating the tests.

The sample XML documents were generated using very well-known XML benchmarks: XMark and Michigan. Documents of different sizes were used in the tests. To standardise the test and make it repeatable, we used the same query sets provided by each benchmark to evaluate different aspects of our solution, Appendix A contains the complete list of queries in each benchmark. Further details about the setup of each test will be provided in the proper subsection of each test.

5.8.2 Join Order Effect

The join order problem is quite common when XML data are stored in relational systems; one of the main reasons for this problem is the differences between the hierarchal XML data model and the flat relational model. As we discussed in Section 5.4, we worked on developing an approach to pick up the right join order for XML queries. Our approach does not require any modification to the relational systems kernel; however, our approach relies on having XML schema summary available, or to be built during document parsing and loading into the database system. The schema summary contains information about each node and path expression in the XML document.

The join order is either managed by the query processor or by forcing a certain join order using the keyword ‘STRAIGHT_JOIN’. The objective of this test is to evaluate the effect of having the right or the optimal join order for any query execution plan. We compared the query run time results between the forced order and those of the same set of queries but without forcing any join order (the database system produces the join order).

5.8.2.1 Experiment Setup

We used XMark benchmark's documents and queries for this test; we ran the test against a medium document size of 100MB and once more against a larger document of 500MB. We conducted the test on an Intel Core2 CPU (2.8GHz) machine with 4GB of RAM. The queries were translated identically for both approaches; the only difference was the join order, we used the SQL keyword 'STRAIGHT_JOIN' to force the join order. In the following example, we demonstrate the difference between the two approaches:

Q19 (XMark benchmark): Give an alphabetically ordered list of all items along with their location.

```
FOR $b IN document("auction.xml")/site/regions//item
LET $k := $b/name/text()
RETURN <item name=$k> $b/location/text() </item>
SORTBY (.)
```

We translated this query into SQL statements based on our approach and as follows:

```
SELECT STRAIGHT_JOIN t1.value,t3.value
FROM XML_Path t4, Elements t3, Elements t1, XML_Path t2
WHERE t4.path like '#/site#/regions#%/item#/location#'
  AND t2.path like '#/site#/regions#%/item#/name#'
  AND t4.path_id = t3.path_id
  AND t2.path_id = t1.path_id
  AND t1.id between xp(t3.id) and xmc(xp(t3.id))
ORDER BY t1.value
```

We call this group SJ for the STRAIGHT_JOIN keyword. However, the second group was translated in exactly the same manner but without the keyword 'STRAIGHT_JOIN'.

We focused on Dewey-based labels, so we used our labelling schemes PoD, PoD-S, along with ORDPATH (a), which is another well-known Dewey-based labelling scheme. We evaluated this test using these three labelling schemes to ensure the consistency of the results.

5.8.2.2 Results and Analyses

The results demonstrated that the typical relational system query processor failed to find the optimal join order for most of the translated XML queries. Failing to find the optimal join order has a serious effect on the query run time. The system-controlled queries showed no consistency in terms of join order for the three labelling methods and for the different document sizes; for the majority of the queries, the query processor failed to find the optimal join order across the three labelling methods. Further, we had to abort some of the system-controlled queries (non-SJ) after running for more than half an hour (1800s) without returning results.

However, the results in Figures 5-16 and 5-17 illustrate that our join order approach has achieved the best query run time for all queries and across the different labelling schemes. The performance gain was significant and it was approximately 400 times faster for queries like Q8 and Q9. The results for the 'STRAIGHT_JOIN' (SJ) approach show a steady outcome for both small and large XML documents. Our approach is consistent and is based on the rules in Section 5.4.

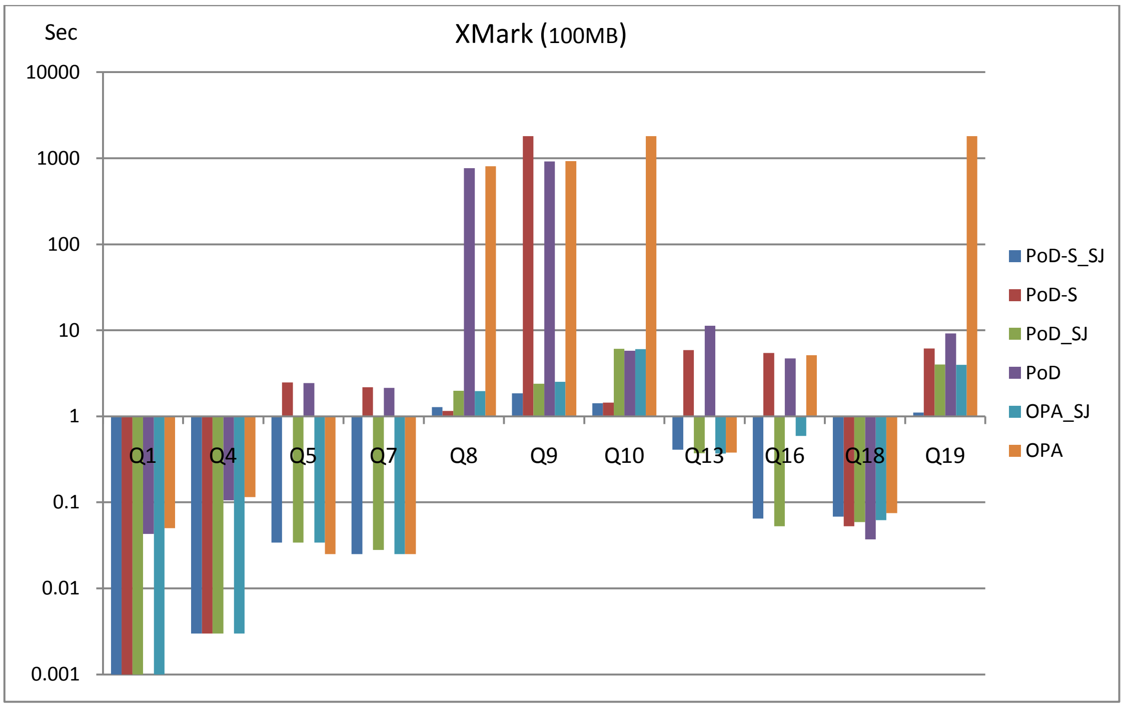


Figure 5-16: Query run times in logarithmic scale for 11 queries from XMark benchmark for three Dewey-based labelling schemes showing the difference between forced join order (SJ) and the join order generated by the system built-in optimiser.

The results in Figure 5-16 demonstrate that the database system optimiser produced optimal join order for Q1 for PoD-S but not for the other two labelling schemes, the same result was observed for Q4, Q10 and Q18. The optimiser produced optimal join order for ORDPATH (a) OPA for Q5, Q7 and Q13. For PoD labelling scheme, the optimiser produced optimal join order only for Q10. It is difficult to explain the behaviour of the system query processor; however, the labels' structures, sizes and their physical layout may play a role in this inconsistent behaviour of the query optimiser.

Interestingly, the system optimiser produced only one execution plan with a more efficient join order than our approach, which was for Q8 and only for the PoD-S labelling scheme.

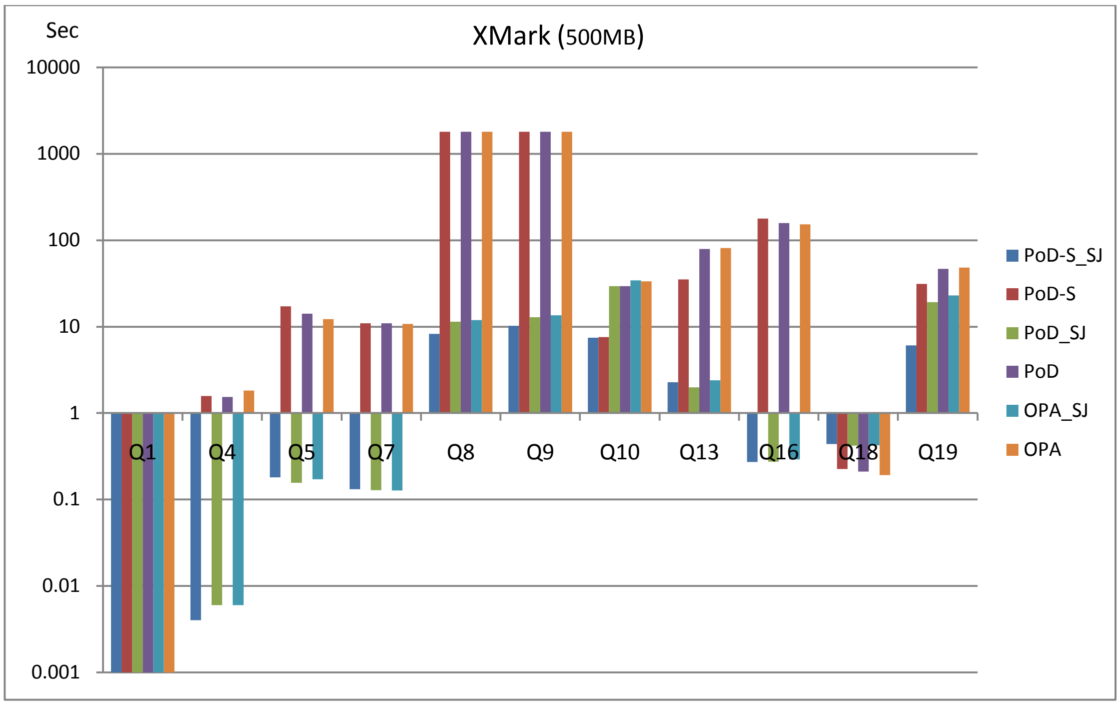


Figure 5-17: Query run times in logarithmic scale for 11 queries from XMark benchmark for three Dewey-based labelling schemes showing the difference between forced join order (SJ) and the join order generated by the system built-in optimiser.

The same scenario occurred for the larger XML document (500MB), illustrated in Figure 5-17. The optimiser produced the optimal execution plan only for Q1 and Q18. The Q18 run time for the non-SJ approach (system optimiser elected the join order) looked faster than our approach; however, analysing the query execution plan showed that the optimiser produced the same join order as our approach but it happened to be faster by a few milliseconds.

We analysed each query execution plan using the provided tools from MySQL. We found that the system optimiser produced a join order similar to the one in Figure 5-4 for most queries.

5.8.3 Twig Query Optimisation

The XQuery language allows tree-pattern match queries as well as queries for individual nodes. These tree pattern queries are called Twig-join queries. Twig-join queries are considered expensive queries because they normally produce large intermediate results.

Most of the Twig-join queries are based on either parent and sibling relationships or ancestor-descendant relationships. Dewey labels in general and PoD-S in particular can efficiently support the first type of Twig queries. However, we proposed a new technique in Section 5.5 to support the second type of twig queries based on a combination of schema summary and xgp() function. This technique considers the sibling-based twig queries as a special case of the ancestor-descendant twig queries. Our technique can be used to optimise both types of Twig queries or any combination of them. Further, it can be applied to any Dewey-based labelling scheme.

We evaluated our approach against the typical RT as discussed in Section 5.5.

5.8.3.1 Test Setup

We evaluated the PoD of 4 bits and ORDPPATH (a) as two Dewey-based labelling techniques. We did not include PoD-S because the split mode might have taken advantage of its split structure and we only want to evaluate the effect of the technique we proposed in Section 5.5. We included ORDPATH, a very well-known labelling scheme, as another Dewey labelling scheme to evaluate the possibility and effect of applying our optimisation techniques on another Dewey-based labelling method.

We used XMark benchmark to generate two documents of different sizes (100MB and 500MB). We also picked up five queries with obvious twig-join case from the XMark benchmark; we evaluated Q1, Q4, Q9, Q16, and Q19. Appendix A-1 contains details about these queries. We ran the same set of queries using XML documents of 100MB and 500MB. The queries were translated into SQL statements based on the techniques in Sections 5.4 and 5.5.

The test was conducted on an Intel Duel Core (1.8GHz) machine with 1GB of memory. As with all other tests, we used MySQL server 5.1 as the backend relational storage. The documents were stored in one main table (Element_table) similar to the Edge table. However, the XML_Path table was used for both labelling methods and the proper implementation of the xgp() function for both labelling schemes.

5.8.3.2 Result and Analysis

The results show that our optimisation techniques provided a significant performance gain over the default technique for both PoD and ORDPATH. Further, the performance gain was achieved for the large XML document (500MB) and in some cases, the performance gain was higher than that of the smaller document, as in Q4. Table 5-2 shows the run time results of this test.

Table 5-2: Run time results for selected twig queries from XMark benchmark.

XMark Query	XM (100MB)/Time (S)				XM (500MB)/Time (S)			
	PoD		ORDPATH (a)		PoD		ORDPATH (a)	
	DEF	XGP	DEF	XGP	DEF	XGP	DEF	XGP
Q1	0.125	0.0001	0.122	0.0001	0.58	0.001	0.59	0.001
Q4	0.39	0.001	.406	0.001	1.99	0.001	2.09	0.001
Q9	5.19	3.65	5.33	3.85	277.2	270.7	295.5	289.4
Q16	1.343	0.084	1.4	0.087	6.859	0.412	6.953	0.437
Q19	10.22	5.684	10.39	6.10	58.02	34.98	65.52	45.97

The performance of the default technique, which is based on the RT, implies the numeric-based intervals approach would not achieve better results, as we found in the evaluation tests in the previous chapter.

It is worth mentioning that the performance enhancement for Q9 is less than other queries. This is due to other factors in the query; this query is complex and it involves data value comparisons (join on value), which took a major part of the query run time.

Note: Queries Q19 and Q1 are sibling-based queries (i.e. For Each item Return '//Item/Name and //Item/Location'). However, we translated it as if it were an ancestor-descendant relationship ('//Item//Name' and '//Item//Location'), which returned the same results. We did this to evaluate our approach against more benchmark queries.

5.8.4 Efficiency of Off-the-shelf Relational Systems

There are acknowledged advantages of using the mature relational database systems to develop and build storage systems that can efficiently support the XML data model and XML documents. However, there is another approach based on developing new data management systems from scratch to support the new XML data model due to the existing limitation in the relational systems. We conducted a test to evaluate the efficiency of using off-the-shelf relational database systems with labelling and optimisation techniques that are used in PoD-S, and compared the results to those of other well-known XML management systems. We have chosen one native XML system (eXist) and another relational-based system (MonetDB/XQuery), which has been modified to support features in XML models by introducing new structural-join algorithms.

The test also aimed to identify the strength and weakness of each approach.

5.8.4.1 Test Setup

We have used our PoD of a 4-bit BLU system in split mode (PoD-S); the PoD-S has proven to be a very efficient configuration and has outperformed other Dewey-based labels by order of magnitudes for most of the evaluation tests that we have conducted thus far. The document nodes (i.e. elements, attributes and leaf nodes) have been stored in three separate tables to reduce the effect of involving irrelevant data in some query evaluations. We translated each query into an equivalent SQL statement using the optimisation techniques and algorithms in this chapter and Chapter 4.

For other XML management systems, we have downloaded the (eXist) management system [exist-db.org, 2009; Meier, 2006]. eXist is a native system that stores XML documents in their original format. We configured eXist to use most of the available memory on the system by increasing the memory space for the Java virtual machine (VM) to 1GB.

We also download the latest version of the MonetDB system for XQuery [Boncz *et al.*, 2006; CWI, 2009]. The core of this system is an in-memory relational-base system that also shreds XML documents and stores them in binary relations. Both systems have XQuery interfaces that allowed us to run the benchmark queries without any need to translate them. However, MonetDB translates XQuery queries internally into SQL-equivalent statements.

Both systems report the query translation time and query execution time; we ignored the query translation times and only reported the query execution time as it was reported by both systems.

We used the Michigan and XMark benchmarks. Every query ran six times; we ignored the first run time and reported the average run time for each query. We aborted any query that did not return results within half an hour (1800sec). We ran almost all queries in each benchmark except for the queries that are not applicable (as QS13, QS14 and QA3 in the Michigan benchmark), or the queries that contain function definitions like QA2 and QA6 in the Michigan benchmark. In addition, we did not run the queries that included 'Full Text' SS (such as QS11 and QS12 in the Michigan benchmark and Q14 in the XMark benchmark) because 'Full Text' indexing is not supported by MySQL InnoDB engine while the other two systems support this feature.

This test ran on an Intel Core2 CPU (2.8GHz) machine with 4GB of RAM using the latest version of MySQL server (5.1) and the latest version of Java VM (1.6.0_18).

5.8.4.2 Results and Discussion

The results of both benchmark queries showed a very promising outcome for our approach, which is based on using a new Dewey-based label structure and advanced optimisation techniques without any change to the relational system kernel; our approach outperformed the other two XML management systems for a certain group of queries. Further, our approach outperformed the native eXist system by returning results for more queries in the test. This demonstrates the reliability and scalability of the mature relational database systems and the advantages of developing solutions on top of them.

We are not going to compare the absolute run time of each query on each system due to the differences between these systems and also to isolate the effect of other factors like configuration and setup. However, we will be focusing on the capabilities and behaviour of each system.

XMark Benchmark Results: Figure 5-18 shows the run time of the XMark queries for XML document of 100MB; the three systems did well for most of the queries in this part of the test. As expected, MonetDB showed strong performance and stability for many reasons: first, MonetDB is a relational-based system; second, MonetDB is an in-memory system; third, MonetDB uses modified structural-join algorithms, which combined with other features in mature relational systems, produced a steady performance. The native system eXist demonstrated a strong performance for ordered access queries (Q2 and Q3), which is an advantage of native systems. However, eXist ran slowly for Q4, even though Q4 is a mix of value's exact match (such as Q1) and nodes global ordering. eXist also did not finish within the 1800 second mark for 'chasing references' queries (Q8 and Q9) since they involved structural joins and value joins (attribute 'id' referencing other nodes). The same outcome occurred for joins on values (Q11 and Q12). eXist did very well for regular path expression queries (Q6 and Q7), path traversal queries, missing element queries (Q15, Q16 and Q17), and sorting and aggregation queries (Q19 and Q20).

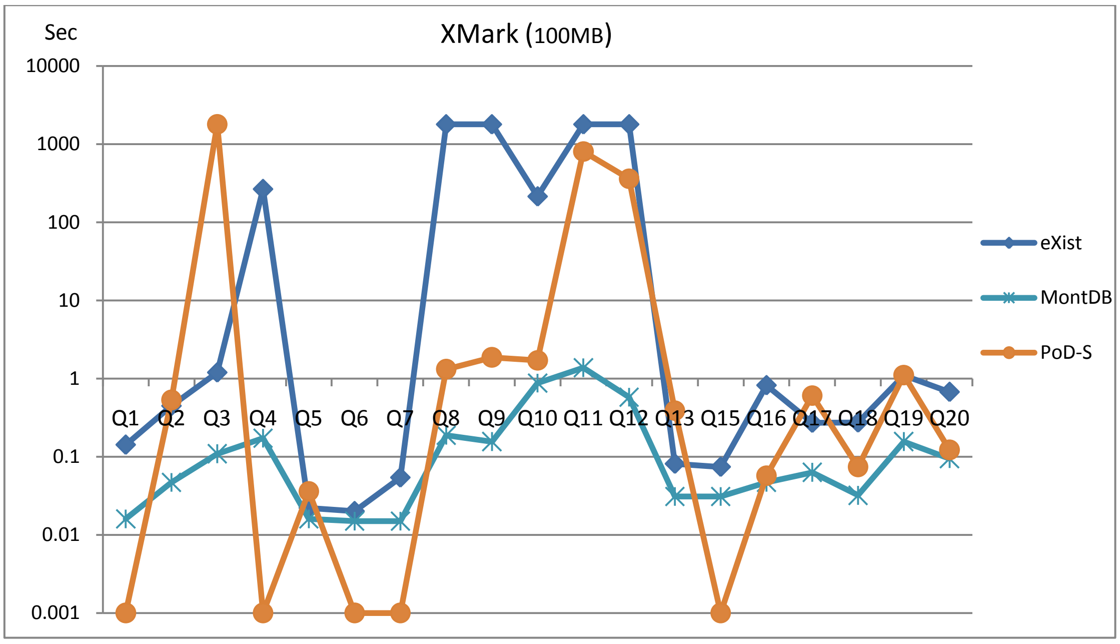


Figure 5-18: Query run times in logarithmic scale for XMark benchmark queries against a medium XML document for three different approaches in XML management systems.

However, PoD-S was not far behind MonetDB; it was more reliable and efficient than the native system eXist. PoD-S showed strong performance for regular path expression queries and path traversal queries due to the use of the XML_Path table, which contains information about the nodes and paths in the documents, and due to the mature indexing mechanism, such as the 'foreign key' feature. We did not focus PoD-S on supporting all ordered access queries like Q2, Q3 and Q4; however, some queries, especially those about global document order (like node 'A' before/after node 'B') as in Q4, can be addressed very efficiently in PoD-S. Q11 and Q12 were slightly challenging for PoD-S because they are complex structural-join queries that also include joins on value. PoD-S did very well for the rest of the queries due to the advanced optimisation techniques that we developed around PoD-S and due to the mature relational systems query-processing engine.

The same trends almost occurred for the larger XML document except MonetDB could not return results for Q11 and the server crashed after exhausting all free memory available to the system (2.6GB). Further, MonetDB ran relatively slowly for Q12, which may raise a question about this system's scalability for this type of query. PoD-S showed excellent scalability and capability to support even larger documents. Figure 5-19 illustrates the results of XMark benchmark queries running against an XML document of 500MB.

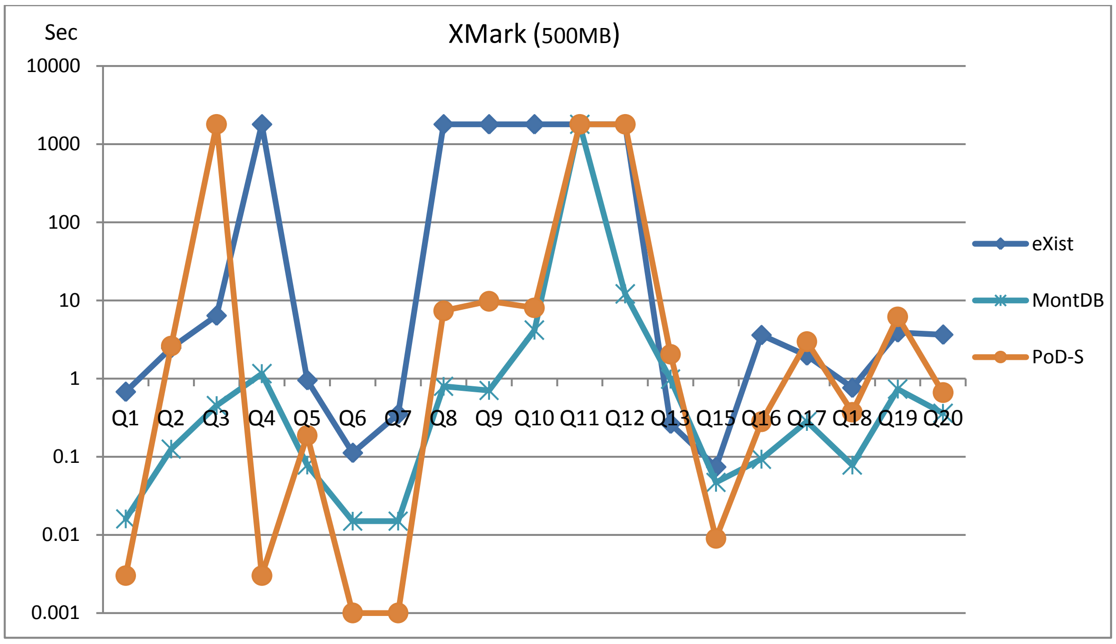


Figure 5-19: Query run times in logarithmic scale for XMark benchmark queries against a large XML document for three different approaches in XML management systems.

The Michigan Benchmark Results: The Michigan benchmark is a micro-benchmark that covers different aspects of the XML data model, in particular, twig queries. We evaluated two documents of sizes 50MB and 500MB using more than 40 queries from this benchmark. Figure 5-20 shows the results for the 50MB document.

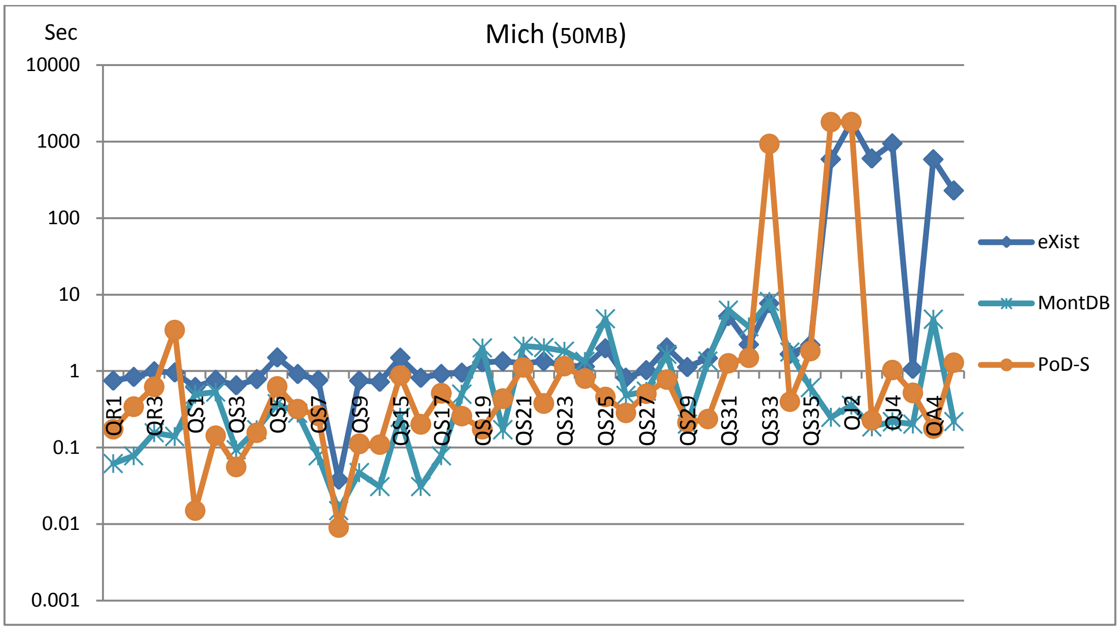


Figure 5-20: Query run times in logarithmic scale for three different XML management systems using the Michigan benchmark with a 50MB document.

The three systems demonstrated excellent performance for most of the queries. However, MonetDB was steadier and it outperformed the eXist system for the value-join and reference-join queries (QJ1, QJ2, QJ3 and QJ4). The native system eXist did not perform well for some of the structural aggregation queries like QA4 and QA5. However, both systems did well for structural join, chain and containment, and twig-join queries (Q20–Q34) since both systems use special structural-join algorithms for tree data.

PoD-S performance was outstanding for this benchmark and surprisingly, it did very well for queries Q20–Q34 even when we are using an unmodified off-the-shelf relational system. This outstanding performance of PoD-S is due to the contribution of the small Dewey-label PoD with a split structure (parent, child), and the right join order rules and algorithms. Further, we found that PoD-S followed the same performance pattern as MonetDB. However, PoD-S could not answer one of the very complex twig queries (Q33) due to the massive size of intermediate results. In addition, PoD-s did not return results for the two value-join queries (QJ1 and QJ2).

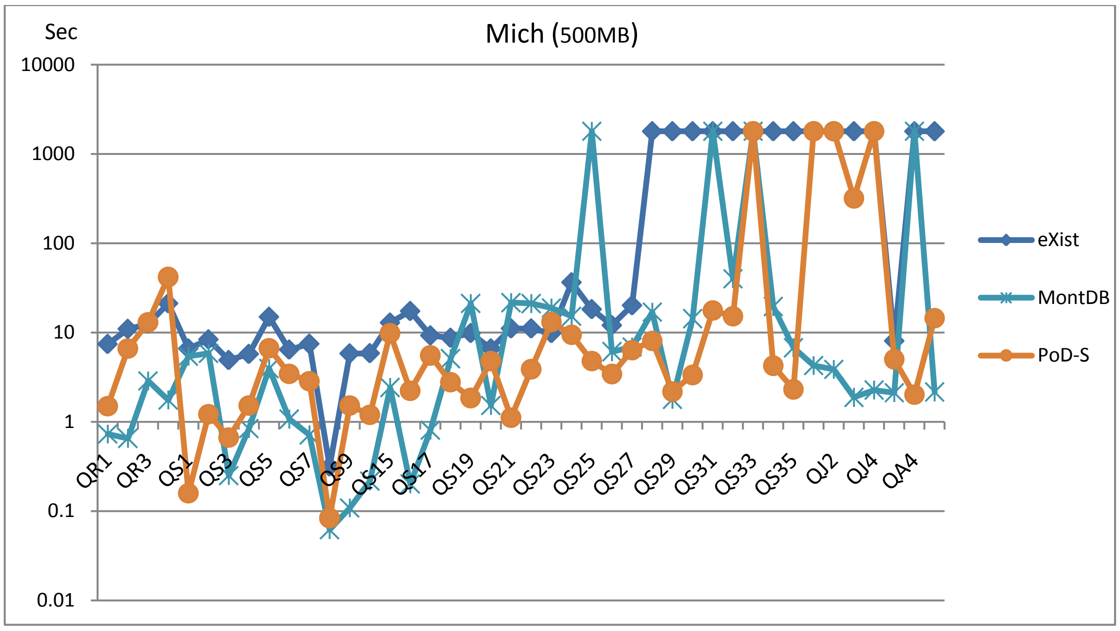


Figure 5-21: Query run times in logarithmic scale for three different XML management systems using the Michigan benchmark with 500MB document.

Figure 5-21 shows the run time of the same set of queries from the Michigan benchmark against the 500MB XML document. Surprisingly, the native system eXist did not scale up for the complex chain and twig queries (Q28–Q34). In addition, it did not return results for Q35 and as expected, it did not answer value-join queries QJ1–QJ4. Further, it did not return results for structural aggregation queries QA4 and QA5.

MonetDB showed a strong performance and scaled up very well for most of the queries and it performed very well for value-join queries (QJ1–QJ4). However, MonetDB crashed on Q25, Q31, and Q33 after exhausting all the available free memory of the system (2.6GB), and unexpectedly, it crashed on QA4.

PoD-S performance was again outstanding and it was very close to the MonetDB performance and capabilities. Further, it outperformed MonetDB for Q25, Q31 and QA4 since it returned results within a relatively short time (less than five seconds) and it did not crash. We can conclude that PoD-S scaled up better than the other two systems for this benchmark. Again, the value-join queries were a challenge for PoD-S because it was the same case for the XMark benchmark.

5.9 Conclusion

We have discussed new ideas to improve the performance of XML queries without the need to change the relational system kernel. Our approach addresses three major issues for optimising XML queries in relational database systems:

- Finding the optimal join order for XML queries by developing rules that can generate the right join order and evaluation sequence for XPath path expressions in any given query.
- Improving the performance of structural-join queries by reducing the number of joins based on using XML document schema summary and an enhanced Dewey-based label structure such as Pod-S.
- Minimising XML queries by developing algorithms outline to minimise any given XML query to the only required XPath paths and elements.

The evaluation tests have shown that our Dewey-based PoD-S labelling technique has significantly improved the performance of the execution of XML queries, and the sibling-based twig queries in particular. Further, using the XGP function has significantly improved the performance of ancestor-descendent-based structural-join queries.

The join order tests demonstrated some of the difficulties that face relational system query optimisers due to the hierarchical nature of the XML data model. However, using the schema summary 'XML_Path' table along with the rules we have discussed in Section 5.4, we managed to produce an optimal join order for more than 90 per cent of the queries.

Finally, by comparing our complete approach to other mature XML management systems, we showed that the relational systems can still be used effectively to store and manage XML documents. Further, as the labelling and optimisation techniques have significantly improved (as in PoD-S), the relational-based systems can easily outperform native XML management systems at an affordable cost.

Chapter 6: Conclusions and Future Work

6.1 Conclusions

Recent studies have shown that relational systems are still competitive as a base technology to develop and build affordable and reliable XML database systems that support all type of XML applications. However, there are major differences between the XML hierarchical data model and the relational data model. These differences, such as document order and containment relationships, present a challenge to the current relational technology available in off-the-shelf relational database systems. Several approaches have been proposed to address these challenges in both native and relational-based systems. However, some of those approaches are not directly applicable to the relational database systems and they require changes to the system's kernel, which is hardly an option for many RDBMS vendors, especially the widely used open source relational database systems.

We have presented the PoD system as an XML database system using an off-the-shelf relational database system. The PoD approach provides new techniques to minimise the storage requirements of XML documents, efficient navigation of XML trees, and advanced optimisation techniques for complex XML structural-join queries without the need to modify the relational database kernel. The PoD approach turns traditional relational systems into affordable, reliable and portable systems that can seriously compete with the expensive, immature and feature-limited native XML database systems.

We set out by introducing the importance of XML technology, in particular for Internet-based applications. We also explained the need to have XML databases to support the rapidly increasing XML documents.

In Chapter 2, we provided an overview of XML, which covers the foundations of XML technology, the XML data model, and XML document structure and syntax. We briefly highlighted related technologies such as DTD and XML schema, which can be used to add domain constrains and to enforce the required document structure. The W3C has recently released XQuery 1.0 as a standard query language for XML data. XQuery is a powerful query language; it is equivalent to the SQL query language. Yet, full support and optimised implementation of XQuery is still an open research area in both native and relational-based systems.

In Chapter 3, we presented the PoD system. PoD is a Dewey-based labelling technique that supports operations on dynamic XML documents (such as insert and delete) without relabelling the existing nodes. Node labelling is a widely used technique in both relational-based and native XML systems to capture the document order and support structural relationships such as parent-child and ancestor-descendant relationships. The PoD approach provides an efficient, flexible and compact labelling technique. Smaller label sizes mean that more labels can be loaded in the memory, which reduces the costly I/O disk operations.

The space evaluation tests showed that the PoD of a 4-bit BLU outperformed recent Dewey-based labelling approaches by reducing the label size by a magnitude of 20 per cent. Further, PoD provides a consistent compression technique for most types of XML documents. Meanwhile, other approaches only work well with a certain type and structure of XML documents.

We have also introduced a new structure for Dewey-based labels. The new structure is based on splitting any given Dewey id into two components: Pid and Cid. The new structure would significantly enhance query performance for queries that are based on parent-child and sibling relationships. The PoD in split mode (PoD-S) makes more efficient use of the current indexing technologies that are available in off-the-shelf relational databases. Further, the PoD-S approach can be applied to other Dewey-based labelling schemes without sacrificing any of the Dewey label features (for example, supporting dynamic XML documents).

Minimising the Dewey label size and simplifying its structure is a major step towards efficiently storing and processing XML documents. However, the labelling technique on its own cannot provide an efficient support for advanced XML queries such as complex structural-join queries. In Chapter 4, we introduced new and efficient techniques to navigate the XML tree and evaluate XPath axis steps by utilising the document structure information. PoD captures the document structure summary (i.e. schema summary) during initial document parsing and stores that in two formats: the in-memory data structure, which can be used during query translation and minimisation; and the schema summary in the relational format in the XML_Path table. The XML_Path table is used to provide alternative query optimisation techniques for Dewey-based labelling schemes. To the best of our knowledge, this is the most detailed study for optimising XPath axis steps in the presence of Dewey labels and schema summary.

The evaluation tests in Chapter 4 demonstrated that our Dewey-based PoD-S labelling technique has significantly improved the performance of the evaluation of XPath axis steps. In addition, it outperformed the typical Dewey-based one-component label scheme. The tests also showed that our optimisation techniques for PoD and PoD-S, which exploit the schema summary in the XML_Path table, have outperformed the traditional RTs for most of the XPath axis steps.

In Chapter 5, we introduced advanced XML query optimisation techniques. While some of these techniques can be applied to native XML database systems, the focus was to address XML query challenges in relational database systems. In particular, we developed techniques to find the optimal join order for query path expressions, improve the performance of structural-join queries by reducing the number of joins, and developing algorithms to minimise XML queries and translate them into equivalent optimised SQL queries. The evaluation tests have shown that our approach has significantly improved the performance of the execution of XML queries in general, and the sibling-based twig queries in particular. The join order tests demonstrated some of the difficulties that face relational system query optimisers due to the nested hierarchical nature of the XML data model. However, by using the PoD system techniques that were discussed in Section 5.4, we managed to produce an optimal join order for more than 90 per cent of the queries.

Finally, by comparing our complete approach to other mature XML management systems (both native and modified relational systems), we showed that the unmodified relational systems can still be used effectively to store and manage XML documents. Further, as the labelling and optimisation techniques have significantly improved (as in PoD-S), the relational-based systems can easily outperform native XML management systems at a very affordable cost.

6.2 Future Work

While we consider the PoD system a successful project, there are still plenty of research opportunities since XML database management systems are relatively new. Recently, the major vendors of relational database systems have started to provide integrated support for XML data in their commercial products. These new hybrid systems support XML data in its native format and integrate that with the existing mature relational technology. However, node-labelling still plays a major role in these systems. For future work, we would like to investigate the opportunities of using the compact labelling scheme and optimisation techniques of PoD with these hybrid systems.

XQuery, which has emerged as the standard query language for XML data, is quite a new technology and there is no complete implementation and support for this language yet in relational systems. Moreover, some of the XQuery language features are still a challenge for relational systems, such as absolute and relative order-based queries (for example, ‘preceding’ and ‘following’ axis steps). These types of queries generate a large number of duplicate values. Further, absolute order (i.e. position) queries may require complex subquery translations. A promising direction for future work is to investigate the possibility of introducing new operators or new algorithms that handle duplicate values at earlier stages of the query execution plan in a way that reduces the cost of reengineering the relational database systems.

Finally, there is no doubt that document-shredding techniques improve XML query performance for data-centric queries. However, based on our own research findings, supporting value-join queries for large XML documents with recursive features and the absence of an XML schema is still relatively slow. We observed the same outcome in other native and modified relational-based XML database systems. More work is still required to address the challenge of value-join queries in relational systems in the absence of XML document schema.

Appendix A: XML Benchmarks

A-1. XMark Benchmark

The complete list of XMark benchmark Queries in XQuery language

-- Q1: Return the name of the person with ID `person0'.

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"] return
$b/name/text()
```

-- Q2: Return the initial increases of all open auctions.

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
return <increase>{$b/bidder[1]/increase/text()}</increase>
```

-- Q3: Return the IDs of all open auctions whose current increase is at least
-- twice as high as the initial increase.

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2
<= $b/bidder[last()]/increase/text()
return
 <increase
 first="{$b/bidder[1]/increase/text()}"
 last="{$b/bidder[last()]/increase/text()}"/>
```

-- Q4: List the reserves of those open auctions where a certain person issued a
-- bid before another person.

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where
 some $pr1 in $b/bidder/personref[@person = "person20"],
   $pr2 in $b/bidder/personref[@person = "person51"]
 satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>
```

-- Q5. How many sold items cost more than 40?

```
let $auction := doc("auction.xml") return
count(
 for $i in $auction/site/closed_auctions/closed_auction
 where $i/price/text() >= 40
 return $i/price
)
```

-- Q6: How many items are listed on all continents?

```
let $auction := doc("auction.xml") return
for $b in $auction//site/regions return count($b//item)
```

-- Q7: How many pieces of prose are in our database?

```
let $auction := doc("auction.xml") return
for $p in $auction/site
return
 count($p//description) + count($p//annotation) + count($p//emailaddress)
```

-- Q8: List the names of persons and the number of items they bought.
-- (joins person, closed_auction)

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
 for $t in $auction/site/closed_auctions/closed_auction
 where $t/buyer/@person = $p/@id
 return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

-- Q9: List the names of persons and the names of the items they bought in
-- Europe. (joins person, closed_auction, item)

```
let $auction := doc("auction.xml") return
let $ca := $auction/site/closed_auctions/closed_auction return
let
  $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a :=
 for $t in $ca
 where $p/@id = $t/buyer/@person
 return
  let $n := for $t2 in $ei where $t/itemref/@item = $t2/@id return $t2
  return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>
```

-- Q10: List all persons according to their interest; use French markup in the
-- result.

```
let $auction := doc("auction.xml") return
for $i in
 distinct-values($auction/site/people/person/profile/interest/@category)
let $p :=
 for $t in $auction/site/people/person
 where $t/profile/interest/@category = $i
 return
  <personne>
   <statistiques>
    <sexe>{$t/profile/gender/text()}</sexe>
    <age>{$t/profile/age/text()}</age>
    <education>{$t/profile/education/text()}</education>
    <revenu>{fn:data($t/profile/@income)}</revenu>
   </statistiques>
   <coordonnees>
    <nom>{$t/name/text()}</nom>
    <rue>{$t/address/street/text()}</rue>
    <ville>{$t/address/city/text()}</ville>
    <pays>{$t/address/country/text()}</pays>
    <reseau>
     <courrier>{$t/emailaddress/text()}</courrier>
     <pagePerso>{$t/homepage/text()}</pagePerso>
    </reseau>
   </coordonnees>
   <cartePaiement>{$t/creditcard/text()}</cartePaiement>
  </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>
```

-- Q11: For each person, list the number of items currently on sale whose price
-- does not exceed 0.02% of the person's income.

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
 for $i in $auction/site/open_auctions/open_auction/initial
 where $p/profile/@income > 5000 * exactly-one($i/text())
 return $i
return <items name="{$p/name/text()}">{count($l)}</items>
```

-- Q12: For each richer-than-average person, list the number of items currently
-- on sale whose price does not exceed 0.02% of the person's income.

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
 for $i in $auction/site/open_auctions/open_auction/initial
 where $p/profile/@income > 5000 * exactly-one($i/text())
 return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)}</items>
```

-- Q13. List the names of items registered in Australia along with their
-- descriptions.

```
let $auction := doc("auction.xml") return
for $i in $auction/site/regions/australia/item
```

```
return <item name="{$i/name/text()}">{$i/description}</item>
```

-- Q14: Return the names of all items whose description contains the word
-- 'gold'.

```
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```

-- Q15: Print the keywords in emphasis in annotations of closed auctions.

```
let $auction := doc("auction.xml") return
for $a in
 $auction/site/closed_auctions/closed_auction/annotation/description/parlist/
 listitem/
 parlist/
 listitem/
 text/
 emph/
 keyword/
 text()
return <text>{$a}</text>
```

-- Q16: Return the IDs of those auctions that have one or more keywords in
-- emphasis. (cf. Q15)

```
let $auction := doc("auction.xml") return
for $a in $auction/site/closed_auctions/closed_auction
where
 not(
  empty(
   $a/annotation/description/parlist/listitem/parlist/listitem/text/emph/
   keyword/
   text()
  )
 )
```

```
return <person id="{$a/seller/@person}"/>
```

-- Q17: Which persons do not have a homepage?

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
where empty($p/homepage/text())
return <person name="{$p/name/text()}"/>
```

-- Q18: Convert the currency of the reserve of all open auctions to another
-- currency.

```
declare namespace local = "http://www.foobar.org";
declare function local:convert($v as xs:decimal?) as xs:decimal?
{
 2.20371 * $v (: convert Dfl to Euro :)
};
let $auction := doc("auction.xml") return
for $i in $auction/site/open_auctions/open_auction
return local:convert(zero-or-one($i/reserve))
```

-- Q19: Give an alphabetically ordered list of all items along with their location.

```
let $auction := doc("auction.xml") return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending empty greatest
return <item name="{$k}">{$b/location/text()}</item>
```

-- Q20: Group customers by their income and output the cardinality of each
-- group.

```
let $auction := doc("auction.xml") return
<result>
 <preferred>
  {count($auction/site/people/person/profile[@income >= 100000])}
 </preferred>
 <standard>
  {
   count(
    $auction/site/people/person/
    profile[@income < 100000 and @income >= 30000]
   )
  }
 </standard>
 <challenge>
  {count($auction/site/people/person/profile[@income < 30000])}
 </challenge>
 <na>
  {
   count(
    for $p in $auction/site/people/person
    where empty($p/profile/@income)
    return $p
   )
  }
 </na>
</result>
```

A-2. The Michigan Benchmark

The complete list of the Michigan benchmark Queries in XQuery language:

-- QR1: Select all elements with aSixtyFour = 2
-- (Return only the element in question)

```
for $e in //eNest[@aSixtyFour=2] return
 <eNest aUnique1="{$e/@aUnique1}">
 </eNest>
```

-- QR2: Select all elements with aSixtyFour = 2
-- (Return the element and all its immediate children)

```
for $e in //eNest[@aSixtyFour=2] return
 <eNest aUnique1="{$e/@aUnique1}">
  {
   for $c in $e/eNest return
   <child aUnique1="{$c/@aUnique1}">
   </child>
  }
 </eNest>
```

-- QR3: Select all elements with aSixtyFour = 2 (Return the entire subtree)

```
for $e in //eNest[@aSixtyFour=2] return
 <eNest aUnique1="{$e/@aUnique1}">
  <descedants>
  {
   for $c in $e//eNest return
   <descedant aUnique1="{$c/@aUnique1}" aFour="{$c/@aFour}">
   </descedant>
  }
  </descedants>
 </eNest>
```

-- QR4: Select all elements with aSixtyFour = 2 and selected descendants

-- with aFour = 1

```
for $e in //eNest[@aSixtyFour=2] return
 <eNest aUnique1="{$e/@aUnique1}">
  <descedants>
  {
   for $c in $e//eNest[@aFour=1] return
   <descedant aUnique1="{$c/@aUnique1}" aFour="{$c/@aFour}">
   </descedant>
  }
  </descedants>
 </eNest>
```

-- QS1: Select elements with aString = 'Sing a song of oneB4'

```
for $e in //eNest[@aString = 'Sing a song of oneB4'] return
 <eNest aUnique1="{$e/@aUnique1}" aString="{$e/@aString}">
 </eNest>
```

-- QS2: Select elements with aString = 'Sing a song of oneB1'

```
for $e in //eNest[@aString = 'Sing a song of oneB1'] return
 <eNest aUnique1="{$e/@aUnique1}" aString="{$e/@aString}">
 </eNest>
```

-- QS3: Select elements with aLevel = 10

```
for $e in //eNest[@aLevel=10] return
 <eNest aUnique1="{$e/@aUnique1}" aString="{$e/@aLevel}">
 </eNest>
```

-- QS4: Select elements with aLevel = 13

```
for $e in //eNest[@aLevel=13] return
 <eNest aUnique1="{$e/@aUnique1}" aString="{$e/@aLevel}">
 </eNest>
```

-- QS5: Select nodes that have aSixtyFour between 5 and 8.

```
for $e in //eNest[@aSixtyFour>=5 and @aSixtyFour<=8] return
 <eNest aUnique1="{$e/@aUnique1}" aSixtyFour="{$e/@aSixtyFour}">
 </eNest>
```

-- QS6: Select nodes with aLevel = a13 and have the returned nodes
-- sorted by aSixtyFour attribute.

```
for $e in //eNest[@aLevel=13]
 order by (/@aSixtyFour)
 return
 <eNest aUnique1= "{$e/@aUnique1}" aString="{$e/@aLevel}"
  aSixtyFour="{$e/@aSixtyFour}">
 </eNest>
```

-- QS7: Select nodes with aSixteen = 1 and aFour = 1.

```
for $e in //eNest[@aSixteen=1 and @aFour=1] return
 <eNest aUnique1="{$e/@aUnique1}" aSixteen="{$e/@aSixtyFour}"
  aFour="{$e/@aFour}">
 </eNest>
```

-- QS8: Selection based on the element name, eOccasional

```
for $e in //eOccasional return
 <eOccasional aRef="{$e/@aRef}">
 </eOccasional>
```

-- QS9: Select the second child of every node with aLevel = 7

```
for $e in //eNest[@aLevel=7] return
 <parent aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
 {
  for $c in $e/eNest[position()=2] return
  <secondChild aUnique1="{$c/@aUnique1}" aLevel="{$c/@aLevel}">
  </secondChild>
 }
 </parent>
```

-- **QS10:** Select the second child of every node with aLevel = 9

```
for $e in //eNest[@aLevel=9] return
 <parent aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
 {
  for $c in $e/eNest[position()=2] return
  <secondChild aUnique1="{$c/@aUnique1}" aLevel="{$c/@aLevel}">
  </secondChild>
 }
 </parent>
```

-- **QS11:** Get 'eOccasional' nodes that have element content
-- contains "oneB4"

```
for $e in //eOccasional
 where contains($e/text(), "oneB4") return
 <eOccasional aRef="{$e/@aRef}">
 </eOccasional>
```

-- **QS12:** Get nodes that have element content contains "oneB4"

```
for $e in //eNest
 where contains($e/text(), "oneB4") return
 <eNest aUnique1="{$e/@aUnique1}">
 </eNest>
```

-- **QS13:** select all nodes with element content that the distance
-- between keyword "oneB5" and the keyword "twenty" is not more than four

```
N/A
```

-- **QS14:** select all nodes with element content that the distance between
-- keyword "oneB2" and the keyword "twenty" is not more than four

```
N/A
```

-- **QS15:** Local ordering. Select the second element with aFour = 1
-- below each element with aFour = 1 if that second element also has aFour = 1

```
for $e in //eNest[@aFour=1] return
 for $c in $e/eNest[position()=2 and @aFour=1] return
 <parent aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}"
  aFour="{$e/@aFour}">
  <secondChild aUnique1="{$c/@aUnique1}" aLevel="{$c/@aLevel}"
   aFour="{$c/@aFour}">
  </secondChild>
 </parent>
```

-- **QS16:** Global ordering. Select the second element with aFour = 1
-- below any element with aSixtyFour = 1

```
for $e in //eNest[@aSixtyFour=1] return
 for $c in $e/eNest[position()=2 and @aFour=1] return
 <parent aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}"
  aSixtyFour="{$e/@aFour}">
  <secondChild aUnique1="{$c/@aUnique1}" aLevel="{$c/@aLevel}"
   aFour="{$c/@aFour}">
  </secondChild>
 </parent>
```

-- **QS17:** Reverse ordering. Among the children with aSixteen = 1 of the parent
-- element with aLevel = 13, select the last child

```
for $e in //eNest[@aLevel=13] return
 for $c in $e/eNest[@aSixteen=1 and position()=last()-1] return
 <parent aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
  <child aUnique1="{$c/@aUnique1}" aLevel="{$c/@aLevel}"
   aSixteen="{$c/@aSixteen}">
  </child>
 </parent>
```

-- **QS18:** Moderate selectivity of both parent and child.
-- Select nodes with aLevel = 13 that have a child with attribute aSixteen = 3

```
for $e in //eNest[@aLevel=13] return
 if (exists($e/eNest[@aSixteen=3])) then
 <eNest aUnique1="{$e/@aUnique1}">
 </eNest>
 else()
```

-- QS19: High selectivity of parent and low selectivity of child.
-- Select nodes with aLevel = 15 that have a child with attribute aSixtyFour = 3

```
for $e in //eNest[@aLevel=15] return
 if (exists($e/eNest[@aSixtyFour=3])) then
 <eNest aUnique1="{$e/@aUnique1}">
 </eNest>
 else()
```

-- QS20: Low selectivity of parent and high selectivity of child.
-- Select nodes with aLevel = 11 that have a child with attribute aFour = 3

```
for $e in //eNest[@aLevel=11] return
 if (exists($e/eNest[@aFour=3])) then
 <eNest aUnique1="{$e/@aUnique1}">
 </eNest>
 else()
```

-- QS21: Moderate selectivity of both ancestor and descendant.
-- Select nodes with aLevel = 13 that have a descendant with aSixteen = 3

```
for $e in //eNest[@aLevel=13] return
 if (exists($e//eNest[@aSixteen=3])) then
 <eNest aUnique1="{$e/@aUnique1}">
 </eNest>
 else()
```

-- QS22: High selectivity of ancestor and low selectivity of descendant
-- Select nodes with aLevel = 15 that have a descendant with aSixtyFour = 3

```
for $e in //eNest[@aLevel=15] return
 if (exists($e//eNest[@aSixtyFour=3])) then
 <eNest aUnique1="{$e/@aUnique1}">
 </eNest>
```

```
    else()
```

-- QS23: Low selectivity of ancestor and high selectivity of descendant
-- Select nodes with aLevel = 11 that have a descendant with aFour = 3

```
   for $e in //eNest[@aLevel=11] return
    if (exists($e//eNest[@aFour=3])) then
    <eNest aUnique1="{$e/@aUnique1}">
    </eNest>
    else()
```

-- QS24: Moderate selectivity of both ancestor and descendant.
-- Select nodes with aSixteen = 3 that have a descendant with aSixteen = 5

```
   for $e in //eNest[@aSixteen=3] return
    if (exists($e//eNest[@aSixteen=5])) then
    <eNest aUnique1="{$e/@aUnique1}">
    </eNest>
    else()
```

-- QS25: High selectivity of ancestor and low selectivity of descendant
-- Select nodes with aFour = 3 that have a descendant with aSixtyFour= 3

```
   for $e in //eNest[@aFour=3] return
    if (exists($e//eNest[@aSixtyFour=3])) then
    <eNest aUnique1="{$e/@aUnique1}">
    </eNest>
    else()
```

-- QS26: Low selectivity of ancestor and high selectivity of descendant
-- Select nodes with aSixtyFour = 9 that have a descendant with aFour = 3

```
   for $e in //eNest[@aSixtyFour=9] return
    if (exists($e//eNest[@aFour=3])) then
    <eNest aUnique1="{$e/@aUnique1}">
    </eNest>
    else()
```

-- QS27: Low selectivity of ancestor and high selectivity of descendant

-- Select nodes with aSixtyFour = 9 that have a descendant with
-- aFour = 3. Return a pair of ancestor and descendant nodes.

```
for $e in //eNest[@aSixtyFour=9] return
 if (exists($e//eNest[@aFour=3])) then
 <eNest aUnique1="{$e/@aUnique1}">
 {
  for $d in $e//eNest[@aFour=3] return
  <descendant aUnique1="{$d/@aUnique1}">
  </descendant>
 }
 </eNest>
 else()
```

-- QS28: One chain query with three parent-child joins with the selectivity
-- pattern: high-low-low-high, to test the choice of join order in evaluating
-- a complex query. To achieve the desired selectivities, we use the following
-- predicates: aFour = 3, aSixteen = 3, aSixteen = 5, and aLevel = 16

```
for $e in //eNest[@aFour=3] return
 if (exists($e/eNest[@aSixteen=3]/eNest[@aSixteen=5]/eNest[@aLevel=16]))
 then
 <firstLevel aUnique1="{$e/@aUnique1}">
 {
  for $c1 in $e/eNest[@aSixteen=3] return
  <secondLevel aUnique1="{$c1/@aUnique1}">
  {
   for $c2 in $c1/eNest[@aSixteen=5] return
   <thirdLevel aUnique1="{$c2/@aUnique1}">
   {
    for $c3 in $c2/eNest[@aLevel=16] return
    <fourthLevel aUnique1="{$c2/@aUnique1}">
    </fourthLevel>
   }
   </thirdLevel>
  }
 </secondLevel>
}
</firstLevel>
else()
```

-- QS29: One twig query with two parent child selection, low selectivity of
-- parent aLevel = 11, high selectivity of left child aFour = 3,
-- and low selectivity of right child aSixtyFour = 3

```
for $e in //eNest[@aLevel=11] return
 if (exists($e/eNest[@aFour=3]) and exists($e/eNest[@aSixtyFour=3])) then
 <parent aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
{ for $c1 in $e/eNest[@aFour=3] where $c1/position()=1 return
  <child1 aUnique1="{$c1/@aUnique1}" aFour="{$c1/@aFour}">
  </child1>
}
{   for $c2 in $e/eNest[@aSixtyFour=3] where $c2/position()=1 return
  <child2 aUnique1="{$c2/@aUnique1}" aSixtyFour="{$c2/@aSixtyFour}">
  </child2>
}
 </parent>
else()
```

-- QS30: One twig query with two parent child selection, low selectivity of parent
-- aLevel = 11, high selectivity of left child aFour = 3, and low selectivity of right
-- child aSixtyFour = 3

```
for $e in //eNest[@aFour=1] return
 if (exists($e/eNest[@aLevel=11]) and exists($e/eNest[@aSixtyFour=3])) then
 <parent aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
{ for $c1 in $e/eNest[@aLevel=11 and position()=1] return
  <child1 aUnique1="{$c1/@aUnique1}" aLevel="{$c1/@aLevel}">
  </child1>
}
{   for $c2 in $e/eNest[@aSixtyFour=3 and position()=1] return
  <child2 aUnique1="{$c2/@aUnique1}" aSixtyFour="{$c2/@aSixtyFour}">
  </child2>
}
 </parent>
else()
```

-- QS31: One chain query with three ancestor-descendant joins with the
-- selectivity pattern: high-low-low-high, to test the choice of join order in
-- evaluating a complex query. To achieve the desired selectivities, we use the
-- following predicates: aFour = 3, aSixteen = 3, aSixteen = 5, and aLevel = 16

```
for $e in //eNest[@aFour=3] return
 if (exists($e//eNest[@aSixteen=3]//eNest[@aSixteen=5]//eNest[@aLevel=16]))
 then
 <firstLevel aUnique1="{$e/@aUnique1}">
 {
  for $c1 in $e//eNest[@aSixteen=3] where
  exists($c1//eNest[@aSixteen=5]//eNest[@aLevel=16]) return
  <secondLevel aUnique1="{$c1/@aUnique1}">
  {
   for $c2 in $c1//eNest[@aSixteen=5] where exists($c2//eNest[@aLevel=16])
   return
   <thirdLevel aUnique1="{$c2/@aUnique1}">
   {
    for $c3 in $c2//eNest[@aLevel=16] return
    <fourthLevel aUnique1="{$c3/@aUnique1}">
    </fourthLevel>
   }
   </thirdLevel>
  }
  </secondLevel>
 }
 </firstLevel>
else()
```

-- QS32: One twig query with two ancestor descendant selection, low selectivity
-- of ancestor aLevel = 11, high selectivity of one descendant aFour = 3,
-- and low selectivity of another descendant aSixtyFour = 3

```
for $e in //eNest[@aLevel=11] return
 if (exists($e//eNest[@aFour=3]) and exists($e//eNest[@aSixtyFour=3])) then
 <ancester aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
{ for $c1 in $e//eNest[@aFour=3 and position()=1] return
  <descendant1 aUnique1="{$c1/@aUnique1}" aFour="{$c1/@aFour}">
  </descendant1>
}
{   for $c2 in $e//eNest[@aSixtyFour=3 and position()=1] return
  <descendant2 aUnique1="{$c2/@aUnique1}" aSixtyFour="{$c2/@aSixtyFour}">
  </descendant2>
}
 </ancester>
else()
```

-- QS33: One twig query with two ancestor descendant selection, low selectivity
-- of ancestor aFour = 1, low selectivity of one descendant aLevel = 11,
-- and low selectivity of another descendant aSixtyFour = 3

```
for $e in //eNest[@aFour=1] return
 if (exists($e//eNest[@aLevel=11]) and exists($e//eNest[@aSixtyFour=3])) then
 <ancester aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
{ for $c1 in $e//eNest[@aLevel=11 and position()=1] return
  <descendant1 aUnique1="{$c1/@aUnique1}" aLevel="{$c1/@aLevel}">
  </descendant1>
}
{   for $c2 in $e//eNest[@aSixtyFour=3 and position()=1] return
  <descendant2 aUnique1="{$c2/@aUnique1}" aSixtyFour="{$c2/@aSixtyFour}">
  </descendant2>
}
 </ancester>
else()
```

-- QS34: One twig query with two ancestor descendant selection, high
-- selectivity of ancestor aFour = 1, low selectivity of a child with aLevel = 11,
-- and low selectivity of another descendant aSixtyFour = 3

```
for $e in //eNest[@aFour=1] return
```

```
 if (exists($e/eNest[@aLevel=11]) and exists($e//eNest[@aSixtyFour=3])) then
 <ancester aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
{ for $c1 in $e/eNest[@aLevel=11 and position()=1] return


  <child aUnique1="{$c1/@aUnique1}" aLevel="{$c1/@aLevel}">
  </child>
}
{   for $c2 in $e//eNest[@aSixtyFour=3 and position()=1] return
  <descendant aUnique1="{$c2/@aUnique1}" aSixtyFour="{$c2/@aSixtyFour}">
  </descendant>
}
 </ancester>
else()
```

-- QS35: Missing Elements. Find all BaseType elements that there is no
-- OccasionalType elements below them. 1) Find all BaseType elements that
-- there is some OccasionalType elements below them. 2) Return elements that
-- are not in 1)

```
for $e in //eNest return
 if (not(exists($e/eOccasional))) then
 <eNest aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}">
 </eNest>
 else()
```

-- QJ1: Low selectivity join: select nodes based on aSixtyFour = 2 and join
-- with nodes with aSixtyFour = 3

```
for $e1 in //eNest[@aSixtyFour=2] return
for  $e2  in  //eNest[@aSixtyFour=2]  where  $e2/@aUnique1=$e1/@aUnique1
return
 <eNest1 aUnique1="{$e1/@aUnique1}" aSixtyFour="{$e1/@aSixtyFour}"
  aLevel="{$e1/@aLevel}">
 <eNest2 aUnique1="{$e2/@aUnique1}" aSixtyFour="{$e2/@aSixtyFour}"
  aLevel="{$e2/@aLevel}">
 </eNest2>
 </eNest1>
```

-- QJ2: High selectivity join: select nodes based on aSixteen = 2 and join with
-- nodes with aSixteen = 3

```
for $e1 in //eNest[@aSixteen=2] return
for $e2 in //eNest[@aSixteen=2] where $e2/@aUnique1=$e1/@aUnique1 return
 <eNest1 aUnique1="{$e1/@aUnique1}" aSixteen="{$e1/@aSixteen}"
  aLevel="{$e1/@aLevel}">
 <eNest2 aUnique1="{$e2/@aUnique1}" aSixteen="{$e2/@aSixteen}"
  aLevel="{$e2/@aLevel}">
 </eNest2>
 </eNest1>
```

-- QJ3: Low selectivity join: select all OccasionalType nodes that point to a
-- node with aSixtyFour = 3

```
for $e1 in //eOccasional return
for $e2 in //eNest[@aSixtyFour=3] return
 if ($e2/@aUnique1=$e1/@aRef) then
 <eOccasional aRef="{$e1/@aRef}">
 <eNest aUnique1="{$e2/@aUnique1}" aSixtyFour="{$e2/@aSixtyFour}">
 </eNest>
 </eOccasional>
 else()
```

-- QJ4: High selectivity join: select all OccasionalType nodes that point to a
-- node with aFour = 3

```
for $e1 in //eOccasional return
for $e2 in //eNest[@aFour=3] return
 if ($e2/@aUnique1=$e1/@aRef) then
 <eOccasional aRef="{$e1/@aRef}">
 <eNest aUnique1="{$e2/@aUnique1}" aSixtyFour="{$e2/@aSixtyFour}">
 </eNest>
 </eOccasional>
 else()
```

-- QA1: Over all nodes at level 15, compute the average value for the

-- aSixtyFour attribute

```
<avgaSixtyFour average="{avg(for $e1 in //eNest[@aLevel=15] return
$e1/@aSixtyFour)}">
 </avgaSixtyFour>
```

-- QA2: Over all nodes at all levels, compute the average value for the
-- aSixtyFour attribute

```
define function one_level(eNest $e)
{
 <average avgaSixtyFour="{avg(for $a in $e/eNest return $a/@aSixtyFour)}"
  aLevel="{$e/@aLevel+1}">
  {one_level($e/eNest)}
 </average>
}
one_level( /eNest)
```

-- QA3: Select elements that have at least two occurrences of keyword "oneB1"
-- in their content

N/A

-- QA4: Amongst the nodes at level 11, find the node(s) with the largest fan-out.

```
<maxFanout aUnique1="{
 for $e in //eNest[@aLevel=11] return
  for $p in //eNest[@aLevel=11] where count($e/eNest)=max(count($p/eNest))
return
  $e/@aUnique1
 }">
</maxFanout>
```

-- QA5: select elements that have at least two children that satisfy aFour = 1

```
for $e in //eNest where count($e/eNest[@aFour=1])>=2 return
<eNest aUnique1="{$e/@aUnique1}">
</eNest>
```

-- QA6: For each node at level 7 (7,3), determine the height of the sub-tree
-- rooted at this node

```
DEFINE FUNCTION depth($e) RETURNS xsd:integer
{ IF (empty($e/eNest)) THEN 1 ELSE max(depth($e/eNest)) + 1 }
for $e in //eNest[@aLevel=7] return
<eNest aUnique1="{$e/@aUnique1}" aLevel="{$e/@aLevel}"
depth="{depth($e)}">
</eNest>
```

QU1-QU6

N/A

Bibliography

Abiteboul, S., Kaplan, H. & Milo, T. (2001). Compact labelling schemes for ancestor queries. *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms.* Washington, DC: Society for Industrial and Applied Mathematics.

Abiteboul, S., Quass, D., Mchugh, J., Widom, J. & Wiener, J. L. (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1, 68–88.

AG, S. (2009). *Tamino: XML managment system* [Online]. Available: http://www.softwareag.com/au/products/wm/tamino/default.asp [Accessed 2010].

Al-Khalifa, S., Jagadish, H. V., Koudas, N., Patel, J. M., Srivastava, D. & Wu, Y. (2002). Structural joins: A primitive for efficient XML Query pattern matching. *The 18th International Conference on Data Engineering (ICDE)*, San Jose, CA, IEEE Computer Society, 141–152.

Amagasa, T & Yoshikawa, M (2003). QRS: A Robust numbering scheme for XML documents. *The 19th ICDE Conference.* Bangalore, India, 705-707.

Apache.org. (2005). *Apache XML Projects* [Online]. Apache Software Foundation. Available: http://projects.apache.org/indexes/category.html#xml [Accessed 2006].

Bao, Z., Ling, T. W., Lu, J. & Chen, B. (2008). Semantic twig: A semantic approach to optimize XML query processing. *The 13th DASFAA Conference*. New Delhi, India: Springer, 282-298.

Berglund, A., Boag, S., Chamberlin, D., Fernández, M., Kay, M., Robie, J. & Siméon, J. (2007). *XML path language (XPath) 2.0* [Online]. Available: http://www.w3.org/TR/xpath20/ [Accessed 2008].

Beyer, K., Cochrane, R., Hvizdos, M., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G., Lyle, R., Nicola, M., Zcan, F. O., Pirahesh, H., Seemann, N., Singh, A., Truong, T., Linden, R. C. V. D., Vickery, B., Zhang, C. & Zhang, G. (2006). DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal,* 45**,** 271–298.

Beyer, K., Cochrane, R. J., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G., Lyle, B., Özcan, F., Pirahesh, H., Seemann, N., Truong, T., Linden, B. V. D., Vickery, B. & Zhang, C. (2005). System RX: One part relational, one part XML. *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data.* Baltimore, Maryland: ACM, 347-358.

Böhme, T. & Rahm, E. (2004). Supporting efficient streaming and insertion of XML data in RDBMS. *Third International Workshop on Data Integration over the Web*. Riga, Latvia, 70–81.

Boncz, P., Grust, T., Keulen, M. V., Manegold, S., Rittinger, J. & Teubner, J. (2006). MonetDB/XQuery: A fast XQuery processor powered by a relational engine. *ACM SIGMOD Conference on Management of Data.* Chicago, IL, 479-490.

Bosak, J. (1999). *Shakespeare 2.00* [Online]. Available: http://www.cs.wisc.edu/niagara/data/shakes/shaksper.htm [Accessed 2005].

Bourret, R. (2005a). *Going Native: Making the case for XML Databases* [Online]. Available: http://www.xml.com/pub/a/2005/03/30/native.html [Accessed 2008].

Bourret, R. (2005b). *XML Database Products* [Online]. Available: http://www.rpbourret.com/xml/XMLAndDatabases.htm [Accessed 2005].

Bray, T., Paoli, J., Sperberg-Mcqueen, C. M., Maler, E. & Yergeau, F. (2008). *Extensible markup language (XML) 1.0 (Fifth Edition). W3C Recommendation* [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/ [Accessed 2009].

Bruno, N., Koudas, N. & Srivastava, D. (2002). Holistic twig joins: Optimal XML pattern matching. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data.* Madison, Wisconsin: ACM, 310-321.

Carey, M., Kiernan, J., Shanmugasundaram, J., Shekita, E. & Subramanian, S. (2000). XPERANTO: A middleware for publishing object-relational data as XML documents. *The 26th of VLDB conference*, Cairo, Egypt, 646-648.

Cathey, R. J., Beitzel, S. M., Jensen, E. C., Grossman, D. & Frieder, O. (2007). Using a relational database for scalable XML search. *Journal of Supercomputing*, 4, 146–178.

Chamberlin, D., Robie, J. & Florescu, D. (2000). Quilt: An XML query language for heterogeneous data sources. In *WebDB.* Dallas, Texas: Springer-Verlag, 1-25.

Chen, Z., Gehrke, J., Korn, F., Koudas, N., Shanmugasundaram, J. & Srivastava, D. (2007). Index structures for matching XML twigs using relational query processors. *Data & Knowledge Engineering,* 60, 283–302.

Cohen, E., Kaplan, H. & Milo, T. (2002). Labelling dynamic XML trees. *Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Madison, Wisconsin: ACM, 271–281.

Connolly, T. & Begg, C. (2010). *Database Systems: A practical approach to design, implementation, and management*, Pearson Education International.

Cooper, B. F., Sample, N., Franklin, M. J., Hjaltason, G. & Shadmon, M. (2001). A fast index for semistructured data. *The 27th VLDB Conference*, Roma, Italy, 341-350.

CWI, D. G. (2009). *MonetDB/XQuery* [Online]. Centrum Wiskunde & Informatica (CWI). Available: http://monetdb.cwi.nl/ [Accessed 2009].

Dehaan, D., Toman, D., Consens, M. P. & Ozsu, M. T. (2003). A Comprehensive XQuery to SQL translation using dynamic interval encoding. *ACM SIGMOD Conference on Management of Data*, San Diego, CA: ACM, 623-634.

Deutsch, A., Fernandez, M. & Suciu, D. (1999). Storing semistructured data with STORED. *ACM SIGMOD*, Philadelphia, PN: ACM, 431–442.

Draper, D., Fankhauser, P., Fernandez, M., Malhotra, A., Rose, K., Rys, M., Simeon, J. & Wadler, P. (2007). *XQuery 1.0 and XPath 2.0 Formal Semantics* [Online]. World Wide Web Consortium. Available: http://www.w3.org/TR/xquery-semantics/ [Accessed 2007].

Evjen, B., Sharkey, K., Thangarathinam, T., Key, M., Vernet, A. & Ferguson, S. (2007). *Professional XML,* Wiley Publishing.

exist-db.org. (2009). *eXist-db Open Source Native XML Database* [Online]. sourceforge.net. Available: http://exist.sourceforge.net/ [Accessed 2009].

Fernandez, M., Hidders, J., Michiels, P., Simeon, J. & Vercammen, R. (2005). Optimizing sorting and duplicate elimination in XQuery path expressions. *16th DEXA Conference.* Copenhagen, Denmark: Springer, 554-563.

Fernandez, M., Malhotra, A., Marsh, J. & Nagy, M. (2007). *XQuery 1.0 and XPath 2.0 Data Model* [Online]. World Wide Web Consortium. Available: http://www.w3.org/TR/xpath-datamodel/ [Accessed 2007].

Fiebig, T., Helmer, S., Kanne, C.-C., Mildenberger, J., Moerkotte, G., Schiele, R. & Westmann, T. (2002). Anatomy of a native XML base management system. *The VLDB Journal*, 11**,** 292-314.

Florescu, D. & Kossmann, D. (1999). Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22**,** 27–34.

Gartner. (2007). *Market share: Relational database management systems by operating system, worldwide* [Online]. Available: http://www.gartner.com/it/page.jsp?id=507466.

Goldman, R. & Widom, J. (1997). DataGuides: Enabling query formulation and optimization in semistructured databases. *The 23rd VLDB Conference*, Athens, Greece, 436–445.

Grinev, M. (2002). XQuery optimization based on rewriting. *6th East European Conference on Advances in Databases and Information Systems.* Bratislava, Slovakia.

Grust, T. (2005). Purely relational FLWORs. *XIME-P*. Paris, France.

Grust, T., Keulen, M. V. & Teubner, J. (2003). Staircase join: Teach a relational DBMS to watch its (axis) steps. *The 29th VLDB Conference*. Berlin, Germany, 524-535.

Grust, T., Keulen, M. V. & Teubner, J. (2004). Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29**,** 91-131.

Grust, T., Rittinger, J. & Teubner, J. (2007). Why off-the-shelf RDBMSs are better at XPath than you might expect. *The ACM SIGMOD International Conference on Management of Data*, Beijing, China: ACM, 949–958.

Härder, T., Haustein, M., Mathis, C. & Wagner, M. (2005). Node labelling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering*, 60, 126–149.

Härder, T. (2005). XML databases and beyond—Plenty of architectural challenges ahead. *Advances in Databases and Information Systems, 9th East European Conference.* Tallinn, Estonia: Springer, 1-16.

Haustein, M. P., Härder, T., Mathis, C. & Wagner, M. (2005). DeweyIDs—The key to fine-grained management of XML documents. *20th Brazilian Symposium on Databases*, Uberlandia, Brazil: UFU, 85–99.

Hoven, J. V. D. (2002). And then there were three. *Information Systems Management,* 19**,** 88–90.

Jagadish, H., Al-Khalifa, S., Chapman, A., Lakshmanan, L., Nierman, A., Paparizos, S., Patel, J., Srivastava, D., Wiwatwattana, N., Wu, Y. & Yu, C. (2002). TIMBER: A native XML database. . *The VLDB Journal*, 11**,** 274-291.

Jiang, H., Lu, H., Wang, W. & Yu, J. X. (2002). Path materialization revisited: An efficient storage model for XML data. *The 13th Australasian Database Conference*, Melbourne, Australia: Australian Computer Society, 85-94.

Jiang, Z., Luo, C., Hou, W.-C., Zhu, Q. & Che, D. (2007). Efficient processing of XML twig pattern: A novel one-phase holistic solution. *18th DEXA.* Regensburg, Germany.

Kaplan, H., Milo, T. & Shabo, R. (2002). A comparison of labelling schemes for ancestor queries. *The Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* San Francisco, CA, 954-963.

Kaushik, R., Bohannon, P., Naughton, J. & Korth, H. (2002). Covering indexes for branching path queries. *ACM SIGMOD,* Madison, WI: ACM, 133-144.

Kha, D. D., Yoshikawa, M. & Umeura, S. (2001). An XML indexing structure with relative region coordinate. *The 17th International Conference on Data Engineering*, Heidelberg, Germany: IEEE Computer Society, 313–320.

Klettke, M. & Meyer, H. (2000). XML and object-relational database systems—Enhancing structural mappings based on statistics. *The 3rd International Workshop on the Web and Databases (WebDB).* Dallas, TX, 63-68.

Kobayashi, K., Liang, W., Kobayashi, D., Watanabe, A. & Yokota, H. (2005). VLEI Code: An efficient labelling method for handling XML documents in an RDB. *The 21st ICDE Conference.* Tokyo, Japan.

Kriegel, A. & Trukhnov, B. M. (2008). *SQL bible*, Wiley Inc.

Kwong, A. & Gertz, M. (2002). *Schema-based optimization of XPath expressions. Techincal Report.* Davis: University of California.

Lee, D., Mani, M. & Chu, W. (2003). Schema conversions methods between XML and relational models. *Knowledge Transformation for the Semantic Web, IOS Press,* 95**,** 1–17.

Lee, G. (2003). *SQL 2003 standard support in Oracle Database 10g* [Online]. Available: http://www.oracle.com/technology/products/database/application_development/pdf/SQL_2003_TWP.pdf [Accessed 2008].

Lee, G. & Team, O. X. D. (2009). Oracle database 11g release 2 XML DB new features. . Available: http://www.oracle.com/technetwork/database/features/xmldb/oracle-19.pdf.

Lewis, P. M., Bernstein, A. & Kifer, M. (2002). *Databases and transaction processing, an application-oriented approach,* Addison-Wesley.

Li, C. & Ling, T. W. (2005). QED: A novel quaternary encoding to completely avoid relabelling in XML updates. *Conference on Information and Knowledge Management (CIKM).* Bremen, Germany : ACM, 501-508.

Li, C., Ling, T. W. & Hu, M. (2006). Efficient processing of updates in dynamic XML data. *The 22nd ICDE Conference.* Atlanta, GA, 13-23.

Li, Q. & Moon, B. (2001). Indexing and querying XML data for regular path expressions. *The 27th VLDB Conference*, Roma, Italy: ACM, 361-370.

Liu, Z. H., Krishnaprasad, M. & Arora, V. (2005). Native XQuery processing in oracle XMLDB. *ACM SIGMOD.* Baltimore, MD: ACM, 828-833.

Lu, J., Ling, T. W., Yong, C. & Chen, T. (2005). From region encoding to extended dewey: On efficient processing of XML twig pattern matching. *The 31st VLDB Conference.* Trondheim, Norway, 193-204.

May, N., Helmer, S., Kanne, C.-C. & Moerkotte, G. (2004). XQuery processing in Natix with an emphasis on join ordering. *XIME-P.* Paris, France, 49-54..

McHugh, J., Abiteboul, S., Goldman, R., Quass, D. & Widom, J. (1997). Lore: A database management system for semistructured data. *SIGMOD*, 26, 54–66.

McHugh, J., Widom, J., Abiteboul, S., Luo, Q. & Rajaraman, A. (1998). Indexing semistructured data. Stanford, CA: Stanford University.

Meier, W. (2006). *Index-driven XQuery processing in the eXist XML database* [Online]. Available: http://exist-db.org/xmlprague06.html.

Melton, J. & Buxton, S. (2006). *Querying XML: XQuery, XPath and SQL/XML in Context*, Elsevier.

Mertz, D. (2001). *Putting XML in context with hierarchal, relational, and object-oriented models* [Online]. IBM developerWorks. Available: http://www-106.ibm.com/developerworks/library/x-matters8/index.html [Accessed 2008].

Michiels, P. (2003). XQuery optimization. *VLDB PhD Workshop.* Berlin, Germany.

Michiels, P., Mihaila, G. & Simeon, J. (2007). Put a tree pattern in your algebra. *ICDE.* Istanbul, Turkey, 246-255.

Min, J.-K., Lee, J. & Chung, C.-W. (2007). An efficient encoding and labelling for dynamic XML data. *The 12th DASFAA, Bangkok, Thailand: Springer*, 715-726.

Moro, M., Vagena, Z. & Tsotras, V. (2008). XML structural summaries. *Proceedings of the VLDB Endowment*, 1, 1524-1525.

MYSQL. (2009). *MySQL database server* [Online]. Available: http://www.mysql.com [Accessed 2009].

Nambiar, U., Lacroix, Z., Bressan, S. & Lee, M. L. (2002). Current approaches to XML management. *Internet Computing, IEEE,* 6, 43–51.

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G. & Westbury, N. (2004). ORDPATHs: Insert-friendly XML node labels. *The ACM SIGMOD International Conference on Management of Data*. Paris, France: ACM, pp. 903–908.

Olteanu, D., Meuss, H., Furche, T. & Bry, F. (2002). XPath: Looking forward. *The 8th Extending Database Technology—EDBT Workshops*. Prague, Czech Republic.

Oracle. (2009). *Oracle XML DB* [Online]. Oracle Technology Network. Available: http://www.oracle.com/technology/tech/xml/xmldb/index.html [Accessed 2009].

Pal, S., Cseri, I., Seeliger, O., Rys, M., Schaller, G., Yu, W., Tomic, D., Baras, A., Berg, B., Churin, D. & Kogan, E. (2005a). XQuery implementaion in a relational database system. *The 31st VLDB Conference*, Trondheim, Norway: VLDB Endowment, 1175-1186.

Pal, S., Cseri, I., Seeliger, O., Schaller, G., Giakoumakis, L. & Zolotov, V. (2004). Indexing XML data stored in a relational database. *The 30th VLDB Conference*. Toronto, Canada: ACM, 1146-1157.

Pal, S., Fussell, M. & Dolobowsky, I. (2005b). *XML support in Microsodt SQL server 2005* [Online]. Available: http://msdn.microsoft.com/en-au/library/ms345117(SQL.90).aspx [Accessed 2008].

Re, C., Simeon, J. & Fernandez, M. (2006). A complete and efficient algebraic compiler for XQuery. *The 22nd ICDE Conference*, Atlanta, GA, 14-26.

Runapongsa, K., Patel, J. M., Jagadish, H. V., Chen, Y. & Al-Khalifa, S. (2003). *The Michigan benchmark* [Online]. EECS The University of Michigan. Available: http://www.eecs.umich.edu/db/mbench/ [Accessed 2007].

Runapongsa, K., Patel, J. M., Jagadish, H. V., Chen, Y. & Al-Khalifa, S. (2006). The Michigan benchmark: Towards XML query performance diagnostics. *Information Systems,* 31, 73–97.

Schmidt, A., Kersten, M., Windhouwer, M. & Waas, F. (2000). Efficient relational storage and retrieval of XML documents. *The World Wide Web and Databases, Third International Workshop WebDB*. Dallas, TX: Springer, 47–52.

Schmidt, A., Waas, F., Kerten, M., Carey, M. J., Manolescu, I. & Busse, R. (2002). XMARK: A benchmark for XML data management. *Proceedings of the 28th VLDB Conference*. Hong Kong, China: VLDB Endowment, 974–985.

Seah, B.-S., Widjanarko, K., Bhowmick, S., Choi, B. & Leonardi, E. (2007). Efficient support for ordered XPath processing in tree-unaware commercial relational databases. *DASFAA.* Bangkok, Thailand, 793-806..

Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., Dewitt, D. & Naughton, J. (1999). Relational databases for querying XML documents: Limitations and opportunities. *The 25th VLDB Conference*. Edinburgh, Scotland. Morgan Kaufmann, 302–314.

Shui, W. M., Lam, F., Fisher, D. K. & Wong, R. K. (2005). Querying and maintaining ordered XML data using relational databases. *The 16th Australasian Database Conference*. Newcastle, NSW: Australian Computer Society, 85-94.

Silberschatz, A., Korth, H. & Sudarshan, S. (2002). *Database system concepts,* McGraw Hill.

Silberstein, A., He, H., Yi, K. & Yang, J. (2005). BOXes: Efficient maintenance of order-based labelling for dynamic XML data. *The 21st International Conference on Data Engineering, ICDE*. Tokyo, Japan: IEEE Computer Society, 285-296.

SQLX.ORG. (2004). *SQL & XML Working Together* [Online]. Available: http://sqlx.org/ [Accessed 2008].

Staken, K. (2001). *Introduction to native XML databases* [Online]. Available: http://www.xml.com/pub/a/2001/10/31/nativexmldb.html.

Surjanto, B., Ritter, N. & Loeser, H. (2000). XML content management based on object-relational database technology. *The 1st WISE*. Hong Kong, China, 70-79.

Tatarinov, I., Beyer, K., Shanmugasundaram, J., Viglas, S. D., Shekita, E. & Zhang, C. (2002). Storing and querying ordered XML using a relational database system. *ACM SIGMOD*. Madison, Wisconsin: ACM, 204–215.

Tian, F., Dewitt, D. J., Chen, J. & Zhang, C. (2002). The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record*, 31, 5–10.

UWDG. (2002). *University of Washington XML repository* [Online]. University of Washington database group Available: http://www.cs.washington.edu/research/xmldatasets/ [Accessed 2008].

UWMRG. (2002). *NIAGARA Experimental Data* [Online]. University of Wisconsin-Madison database research group. Available: http://www.cs.wisc.edu/niagara/data.html [Accessed 2009].

W3C. (1998). *XML-QL: A query language for XML* [Online]. Available: http://www.w3.org/TR/NOTE-xml-ql/ [Accessed 2008].

W3C. (1999). XML query language (XQL). Available: http://www.w3.org/TandS/QL/QL98/pp/xql.html.

W3C. (2004). *XML Schema* [Online]. Available: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/ [Accessed 2007].

W3C. (2010a). *World Wide Web consortium* [Online]. Available: http://www.w3c.org.

W3C. (2010b). *XML essentials* [Online]. Available: http://www.w3.org/standards/xml/core.

Weigel, F., Schulz, K. U. & Meuss, H. (2005). The BIRD numbering scheme for XML and tree databases—Deciding and reconstructing tree relations using efficient arithmetic operations. *The 3rd XML Database Symposium, XSym2005*. Trondheim, Norway, 49-67.

Wu, X., Lee, M. L. & Hsu, W. (2004). A prime number labelling scheme for dynamic ordered XML trees. *The 20th ICDE Conference*. Boston, MA: IEEE Computer Society, 66–78.

Wu, Y., Patel, J. & Jagadish, H. (2003). Structural join order selection for XML query optimization. *The 19th ICDE Conference*. Bangalore, India, 443-454.

Xu, L., Ling, T. W., Wu, H. & Bao, Z. (2009). DDE: From dewey to a fully dynamic XML labelling scheme. *SIGMOD*. Providence, Rhode Island: ACM, 719-730.

Yoshikawa, M., Amagasa, T., Shimura, T. & Uemura, S. (2001). XRel: A path-based approach to storage and retrieval of XML documents using relational database. *ACM Transactions on Internet Technology,* 1, 110–141.

Zhang, C., Naughton, J., Dewitt, D., Luo, Q. & Lohman, G. (2001). On supporting containment queries in relational database management systems. *SIGMOD Rec.,* 30, 425–436.