A FRAMEWORK FOR CAPTURING, QUERYING, AND RESTRUCTURING METADATA IN XML DATA

By

HAO JIN

A dissertation submitted in partial fulfillment of

the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY

School of Electrical Engineering and Computer Science

AUGUST 2005

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of HAO JIN find it satisfactory and recommend that it be accepted.

____

Chair

____

____

ACKNOWLEDGEMENT

First of all, I would like to express my deepest thanks to my advisor Dr. Curtis Dyreson, for his inspiring guidance and uncounted help throughout my Ph.D. program. He has led me into our exciting research field, helped me gaining deeper understanding of the subjects, and given me invaluable comments during the whole process of my research work including the writing of this dissertation. He has set a perfect role model for me not only as a professional researcher but also as a great human being as well.

I would also like to give a special thanks to my graduate committee members, Dr. Dave Bakken and Dr. Murali Medidi too, for their insightful advice and kind assistance.

I am very grateful to all the professors that have taught me and worked with me in Washington State University. What I have learned from them has built a solid foundation for my Ph. D. studies and research, and will definitely contribute greatly to my future career as well.

The four years I spent in Washington State University is a life-changing experience for me. I would like to thank the School of EECS and Washington State Univerisity for giving me this wonderful opportunity.

Last but not least, I want to thank my parents and my wife, Yanru Qu, for their unconditional love and support in all these years.

A FRAMEWORK FOR CAPTURING, QUERYING, AND RESTRUCTURING METADATA IN XML DATA

ABSTRACT

by Hao Jin, Ph.D.
Washington State University
August 2005

Chair: Curtis E. Dyreson

Metadata plays an important role in describing and proscribing data in both traditional and XML applications. This dessertation presents a framework to represent the metadata and maintain its consistency with the data in querying and restructuring XML data. The data model extension is called MetaDOM. MetaDOM separates data and metadata into different scopes and supports multiple levels of metadata (i.e. meta-metadata). The query language extension is called MetaXQuery. MetaXQuery is an extensible framework in which the special semantics of different kinds of metadata are specified as “plug-in” components. Each component is a set of simple, low-level operations. We first present the theoretical foundations of the framework. The framework shows how to utilize each metadata-specific component to retrieve, certify, sanitize, filter, group, and restructure data with metadata. We then show how to convert MetaXQuery expressions into low-level algebraic operators in MetaTAX and how to extend a native XML DBMS, namely eXist, to support the framework. The implementation judiciously reuses eXist’s indexes and query evaluation engine to attain high efficiency. Finally, we setup a benchmark platform as well as a data/metadata generator to test the performance of our system for further optimization and development.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE
INTRODUCTION

The Extensible Markup Language (XML) [79] was developed by the World Wide Web Consortium (W3C) in 1996, and it's quickly becoming the *de facto* standard for formatting and exchanging data on the Web. XML is a subset of the Standard Generalized Markup Language (SGML). It makes use of tags (words bracketed by '<' and '>') and attributes (of the form name="value") to encapsulate data. XML uses the tags to delimit pieces of data, and leaves the interpretation of the data to the application that reads it. In that way, it is able to support the electronic exchange of machine-readable data. An XML document can contain an optional description of its grammar in the form of Document Type Definition (DTD) or XML Schema for use by applications that need to perform structural validation.

XML is actually a family of technologies [80]. XML 1.0 [79] is the specification that defines what "tags" and "attributes" are and how an XML document is used to carry information items. Beyond XML 1.0, "the XML family" is a growing set of modules that offer useful services to accomplish important and frequently demanded tasks. XLink [82] describes a standard way to add hyperlinks to an XML file. XPointer [83] is a syntax for pointing to parts of an XML document. An XPointer is a bit like a URL, but instead of pointing to documents on the web, it points to pieces of data inside an XML file. CSS [75], the style sheet language, can be used to render XML for presentation. XSL [91] is an advanced language for expressing style sheets. It is based on XSLT [92], a transformation language used for rearranging, adding and deleting tags and attributes. The Document Object Model (DOM) [76] is a standard set of function calls for manipulating XML files from a programming language. XML Schema [88] [89] [90] helps developers to precisely define the structures of their own XML-based formats.

The most important and common usage of XML is to format, carry and exchange data between data sources over a network. There are many heterogeneous data sources (databases mostly) on the web. Most of them contain data in incompatible formats. Instead of using a proprietary format for each peer to communicate data, XML can be used as the universal format because it's standard, self-descriptive, text-based, and easy to process.

XML has made it easier to exchange and describe data, but data only makes sense in the context of *metadata*. The following definition of metadata comes from the glossary of Dublin Core Metadata Initiative [15].

> *In general, metadata is the 'data about data' or functionally, 'structured data about data'. Metadata includes data associated with either an information system or an information object for purposes of description, administration, legal requirements, technical functionality, use and usage, and preservation.*

Another definition of metadata comes from the online Hyper Dictionary [99].

> *Data about data. In data processing, metadata is definitional data that provides information about or documentation of other data managed within an application or environment. For example, metadata would document data about data elements or attributes, (name, size, data type, etc) and data about records or data structures (length, fields, columns, etc) and data about data (where it is located, how it is associated, ownership, etc.). Metadata may include descriptive information about the context, quality and condition, or characteristics of the data.*

We will see an example of metadata in the motivating example later.

There are a lot of applications in which metadata plays a critical role. One of the earliest and still one of the most important applications is the *card catalog* in a library. The cards record the essential information about books, journals, and magazines. They are used to classify, store and quickly find the publications. The Dublin Core Metadata Initiative named fifteen standard metadata elements [16] for cross-domain information resource description of document-like networked objects, in a manner similar to the library card catalog. These elements include title, creator, subject and keyword, description, publisher, contributor, date, resource type, format, resource identifier, source, language, relation, coverage, and rights management. Another important application of metadata is *search engines*. A search engine analyzes the data in web pages. The analysis leads to a set of metadata that describes and summarizes each page. The metadata can include the page size, last modification time, number of hyperlinks, frequency of some particular keyword, and content descriptors. The metadata is used primarily to improve the accuracy of search results.

Metadata has always been a vital part of a database management system as well. A database schema has traditionally been considered to be metadata. For example, a relational database contains tables with columns. Metadata names these columns, describes their data types (int, char, etc.) and sizes. Column constraints such as "checks" are also part of the metadata. Without the metadata, a database file would just be a long string of uninterpreted bits. There could also be security metadata that tells the database manager who can access what information. All of the metadata in a relational database is stored in tables, just like the data.

Two of the most extensively studied kinds of metadata in both the database and XML world are *time* and *security*. Temporal metadata is important because many data management applications involve evolution over time. Data changes as it is created, modified, and deleted. For auditing purposes, many systems have to record these changes. The changes involve two temporal dimensions: *valid time* and *transaction time* [34]. The *valid time* of a fact is the time when the fact is true in the modeled reality, and it's usually supplied by the user. The *transaction time*, on the other hand, is when a database fact is current in the database and may be retrieved. Transaction times are consistent with the serialization order of transactions, and they are system generated and supplied. Security metadata is also very important especially on the web. It can impose restrictions on data access and use; only authorized users are allowed to view and edit data.

From the sample metadata discussed above, we can find some general rules to categorize metadata. One of the categorizations is *derivative* vs. *assertive*. Derivative metadata is summarized, calculated, or derived from the data. It is sensitive to the content of the data. For example, search engine metadata is derived from an analysis of a web page, which means it is derivative. If the content of the web page is changed, the metadata will also change and should be derived again. Assertive metadata, on the hand, is assigned to data and is insensitive to the content of the data. It includes things such as author, publisher, rights.

Another way of categorizing metadata is *descriptive*, *interpretive* and *proscriptive*. Descriptive metadata solely describes the property of the data, such as author, last modification time, or size. Interpretive metadata is metadata that is essential to understanding a piece of data. Interpretive metadata is common in scientific data, for instance an integer value that represents a temperature is only meaningful in the context of a particular metric or scale, e.g., Celsius. Proscriptive metadata adds some functional meaning to the data and may change how the data looks to the user. For example, the access rights metadata adds restrictions as to who can manipulate the data. So if there are different access rights to the same piece of data, two different users may have different views of the data. In our framework, these categories are important because they help explain the semantics of the metadata.

One thing that we need to point out is that the distinction between "data" and "metadata" is relative. Data stands in relation to metadata as does metadata to meta-metadata. The distinction between data and metadata is created primarily by a particular use and many times the same resource will be interpreted in both ways simultaneously.

Database management systems have long been an important branch of information technology. A database is a collection of data stored on persistent media (e.g., tapes, disks). The software application to manage, update, and query the database is known as a database management system (DBMS). People have created many types of DBMSs. The most widely used one is the *relational database*, which uses relations as its logical data model and stores data as records. Popular products in this category include the biggest players in the industry such as Microsoft SQL Server, Oracle, IBM DB2, and also open source products like MySQL and Postgres. Another type of commonly used database is the *object-oriented database* (*OODB*), which stores data as objects and can only be interpreted by the methods of its class. The concepts are completely derived from the object-oriented programming languages, and most of the time its usage is also associated with an object-oriented programming environment so that the same model is used across different application levels. Famous products in this category include Jasmine from Computer Associates, Jade, ObjectDB, Objectivity, etc. There are other types of database too, such as *deductive database* which can make deductions based on rules and facts, or *flat-file database* which is just made up of strings in one or more files.

As XML is quickly becoming the *lingua franca* of the Web, there's an increasing need to convert existing data in legacy formats to XML and vice versa. For data already formatted in XML, it is often more efficient and easier to just store the data directly, which has led to the development of a special type of database that can store, query, and manipulate XML data.

Since relational and object-oriented databases have existed for a long time, their technology is well tested and well developed. So the first step is to extend these mature database products to support storing and querying of XML data. These relational or object-oriented database enriched with XML functionalities are called *XML-enabled databases*. Currently all the major database vendors such as Microsoft, Oracle, and IBM have made such moves into the XML market. In an XML-enabled database, the storage model is not XML, but rather relational or object-oriented. So the XML data has to be *shredded*, that is, converted to the underlying storage model. Some products can automatically shred data, but others require the schema to be created first. Queries languages in an XML-enabled databases are often a mix of SQL and an XML query language. Overall, the benefit of an XML-enabled database is that it reuses existing technologies so the data and applications can still be used without any changes. But since the XML document is shredded and then reassembled, some important information may be lost, such as the document order, processing instructions, and comments. This loss is inconsequential for most XML data.

On the other hand, *native XML databases* are designed specifically to store XML documents. The fundamental unit of storage in such a database is an XML document rather than a table as in a relational database. By storing documents, all the information in an XML document can be faithfully preserved. Native XML databases can be queried using an XML query language. Since the XML doucment doesn't have to be shredded into the internal data model, usually the storing and querying process is faster than the XML-enabled database, especially when an entire document is retrieved. In general, native XML database are better for XML documents whose natural format is important, such as documents used for messaging in an e-commerce system.

One of the advantages of native XML database is that they use an XML query language instead of mixing with SQL. There are many query languages for XML, among which XQuery is the most popular and will likely continue to be prominent in the future. XML query languages are evaluated on the logical data model of XML, and so far the most widely accepted data model of XML is a tree (e.g., the Document Object Model). The tree has a single document root and elements as the intermediate nodes, as illustrated in Figure 1(a). The query language of XML usually specifies both the name or value of the nodes and the path to reach them in the expression. The query processor traverses the data model tree to get to the desired nodes. An example of such traversal shown in Figure 1(b). To reach node *Q* in the lower left corner, the query processor has to start from the root node and traverse the entire path as depicted by the dashed line in the figure.

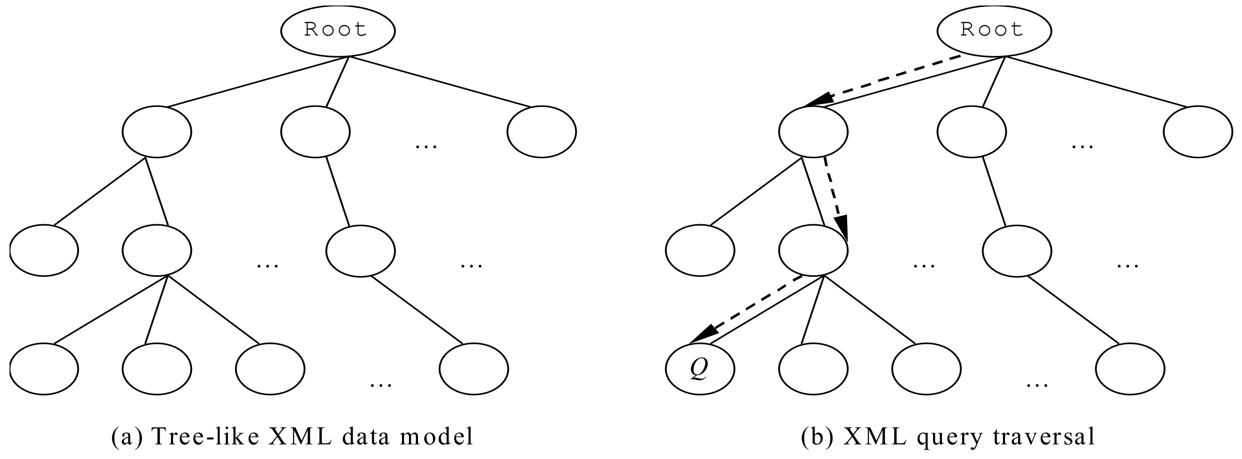


Figure 1 Tree-like XML data model and query traversal

As we have seen in the previous paragraphs, metadata plays a critical role in a lot of applications and has to be carefully maintained in association with the data. But current XML systems and query languages lack the ability to specify and query metadata. We need to add support in an XML application to separate the role of data from metadata and understand the semantics of metadata. Our contribution is to model metadata as another level of data that annotates data. The anntations play an important role in all data model operations, such as querying and restructuring. Figure 2 shows metadata annotating data in an XML data model. The metadata is affixed to individual data nodes.

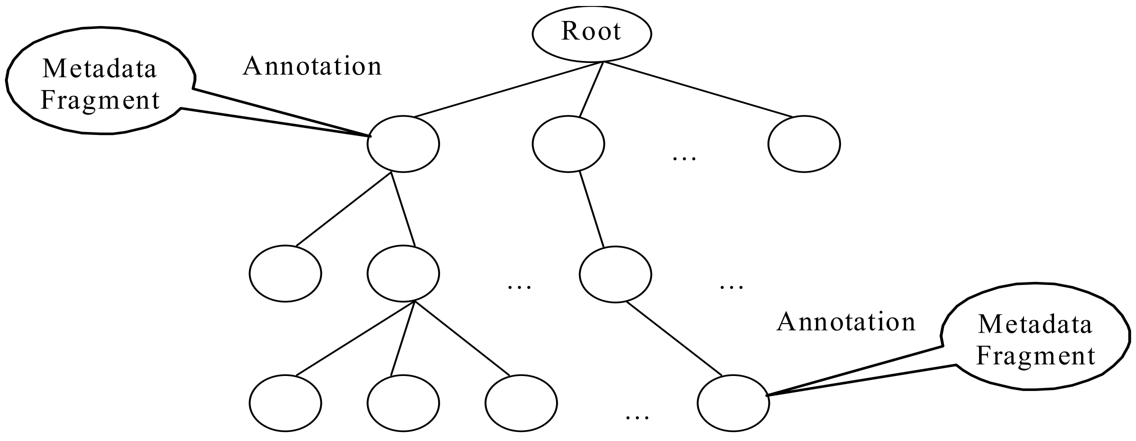


Figure 2 Data with metadata annotation

The DBMS must be extended to add support for metadata. The first change to a DBMS is that query execution has to be aware of the metadata annotations since they are now an integral part of the data model. Metadata adds constraints to the reachability of edges in the data model. A child node should have compatible metadata to its parent so that we can traverse the edge from parent to child, otherwise this edge should be considered invalid and nodes beneath it unreachable. For example, if a node is annotated with security metadata “Joe”, which means only user Joe is allowed to access it, then all of its descendant nodes should be annotated with security metadata “Joe” as well. If a node beneath it is accessible to someone else, he or she can never get to that node in the first place.

Second, in a query result, the data should retain its metadata. In operations that restructure XML data or generate new XML data, their original metadata should be properly combined or calculated and attached to the new data. As the example shown in Figure 3, the two node *A* in the source tree are merged into a new node *A'*. Both nodes have metadata annotation originally, so the new node *A'* should have the combined metadata fragment of these two. The combination rule would depend on the semantics of each particular type of metadata.

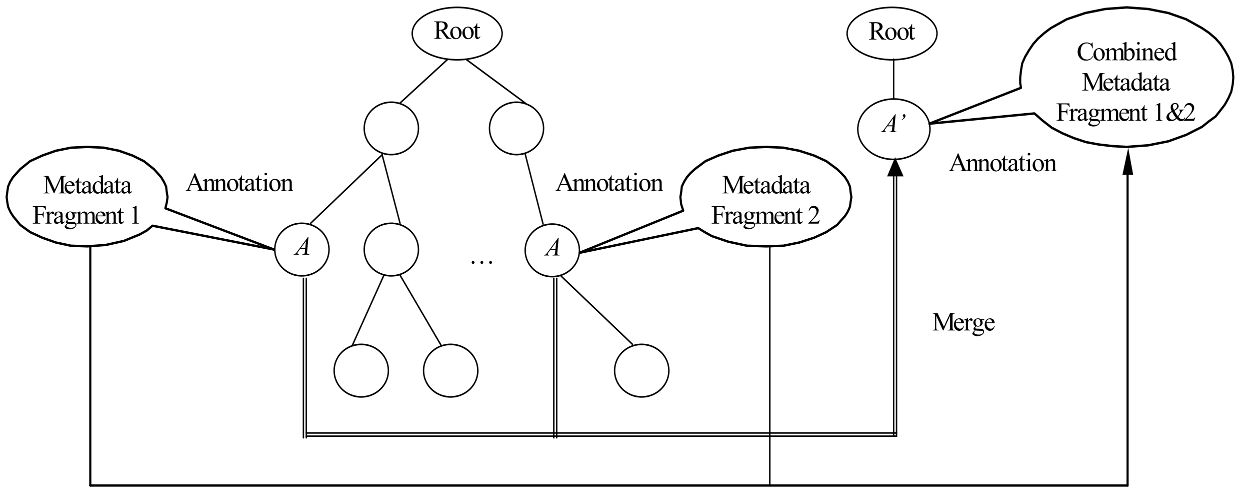


Figure 3 Restructuring XML data with metadata annotations

Finally the query could now have implicit metadata conditions, such as the user that logs on to the system. So instead of evaluating the query on data only, it could match metadata with the implicit conditions, which we call the perspective, as the query is evaluated. As depicted in Figure 4, originally the query processor only traverses the data tree to reach the desired node *Q*. But now, for every node on the path, it needs to match its metadata values with the perspective to see if they match. If they don't, the nodes further along the path are unreachable. This operation is particularly useful for dynamically generating user views, which is a much cheaper operation than materializing views for each condition.

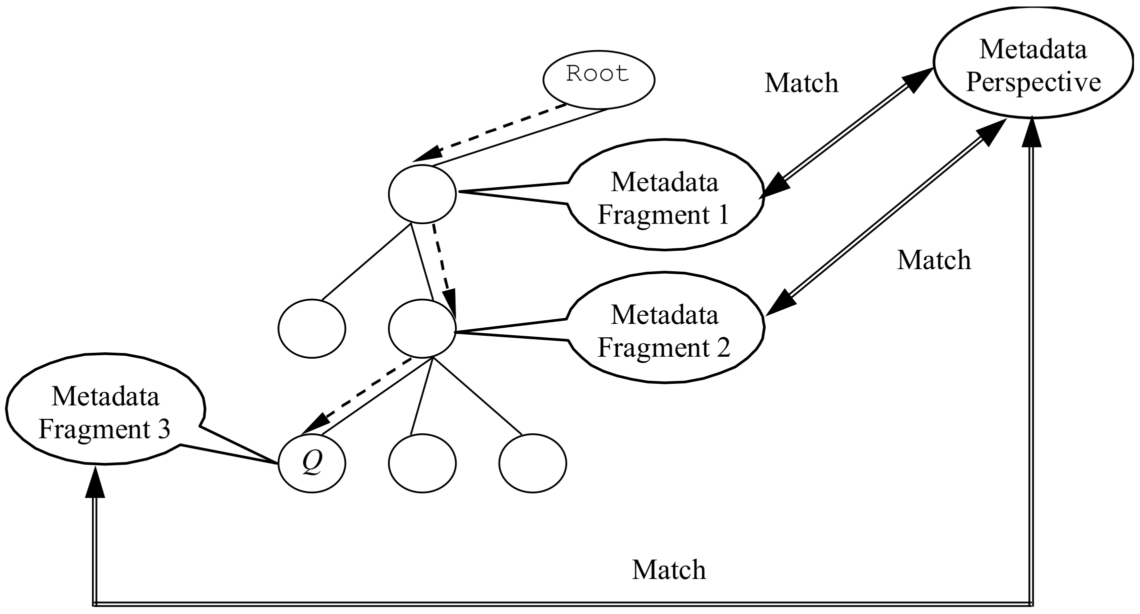


Figure 4 Matching metadata perspective while traversing data

There are other issues related with this framework. First, the XML schema is used to validate the structure of XML data. With metadata annotation, further validation has to be done on the semantics of metadata as well. For example, the snapshot of a bank account should have only one `balance` element as the child of `account`. If we record the changes of the balance over time and annotate it with transaction time metadata, there could be more than one balance element under each account, which violates the schema constraint. We need to extend the schema to be sensitive to embedded metadata. Another issue is the physical representation of metadata. There are several options for storing data together with metadata. The choice is important because it affects the implementation, so careful consideration is needed.

The major work and main contribution of this dissertation is an XML framework that can capture the relationship between data and metadata, as well as preserve their consistency in querying and restructuring. In order to do that, we first create an extended XML data model called MetaDOM based on the Document Object Model (DOM). The data model treats metadata as a different level of data that annotates data. Both the representation of metadata and data model are the same as data, so we reuse the same standards in our framework without introducing anything different or incompatible.

The query language for our framework is also an extension on the existing standard, XQuery, which we call MetaXQuery. We try to make as few changes as possible in MetaXQuery and mostly to encapsulate the tasks into functions so as to make the user's job easier. We also design MetaDOM and MetaXQuery to be upwards compatible to DOM and XQuery so people can still use their original data and application in our framework without noticing any differences.

We choose eXist, which is an open-source native XML database written completely in Java, to build our metadata framework. To implement the metadata functions, we first convert the primitive ones into low-level algebraic operators. W use an existing XML tree algebra, TAX, and expand it into MetaTAX. The algebraic operators can capture the key operations in the framework and also play an important role in optimizing query execution plans. A naïve implementation of MetaXQuery could cause huge overhead, so we also explore optimization algorithms by using indexes. Finally, to evaluate the performance of our framework as well as to compare the nature of different implementation strategies, we build a benchmark platform to test the performance of XML query processing systems. We use this benchmark platform and also another existing XML benchmark to validate the performance and scalability of our framework.

The rest of this dissertation is organized as follows. First we present an example that motivates this research. We then summarize MetaDOM, which is the extension of DOM and present MetaXQuery, the XQuery extensions to incorporate the metadata semantics. To implement these functions, we defined a metadata-aware XML tree algebra to translate MetaXQuery expressions into low-level operators. We implemented our prototype system in both an in-memory model as well as a persistent model. In Chapter 6, we describe the general methodology of benchmarking systems and especially benchmarking XML processing systems, and we also show an XML benchmarking platform we developed. In Chapter 7, we use both our own benchmark and another popular XML query benchmark in the literature to test our prototype system. Finally, we present related work to our research as well as the conclusion and future work.

CHAPTER TWO

MOTIVATING EXAMPLE

Our motivating example is adapted from the W3C's XQuery Use Cases document [87]. Suppose that a publisher makes information about new books available only to online subscribers that pay for the service. The publisher annotates the book data with *security* metadata. The security is intended to limit access to only the users that have paid for their service. The publisher also wants to archive the book information and so decides to record the transaction time of the book data. The transaction time is the system time when the data is available. Suppose that user Joe has only paid for the service from times 3 to 9. The publisher adds meta-metadata to record the lifetime of Joe's changing access rights, which is important to supporting *versioned security*. Versioned security can accurately record and support access rights which change over time. For instance, assume Joe subscribed to the book data from time 3 to time 9, but does not have a current subscription. With versioned security, Joe's previous access rights can be supported (via a rollback of the database to time 5), so that Joe can have access to archived data (whatever Joe could access from time 3 to 9). The data and metadata are parsed to create an instance of a metadata-aware data model, which is shown in Figure 5. Due to space limitations only a small amount of the data and metadata is shown. The metadata and data in the instance are *separated* into different, yet related scopes. The key difference between this instance and an instance of a normal XML data model is the directed edge from the book element to metadata and from the user element in the metadata to meta-metadata. These directed edges represent the relationship between data and its metadata. By following such an edge, a user jumps to a metadata *scope*. The scope limits the search performed by a "wild card". For instance a query that follows the descendant axis from the book node will not descend into the metadata. Existing XML data models do not support separating the scope. A more critical problem with existing data models is that a metadata-aware data model must have some mechanism to enforce the semantics of metadata during path navigation. In current XML data models, metadata can be embedded in the data as a child of each data element. The problem is that query evaluation could then (incorrectly) ignore the existence of metadata, i.e., a wildcard query might traverse edges in a data model instance for which the user is not authorized. Or query evaluation could inadvertently include the metadata when not disired, e.g., an aggregate query counting the number of children of an element node could accidently include the metadata sub-element in its result.

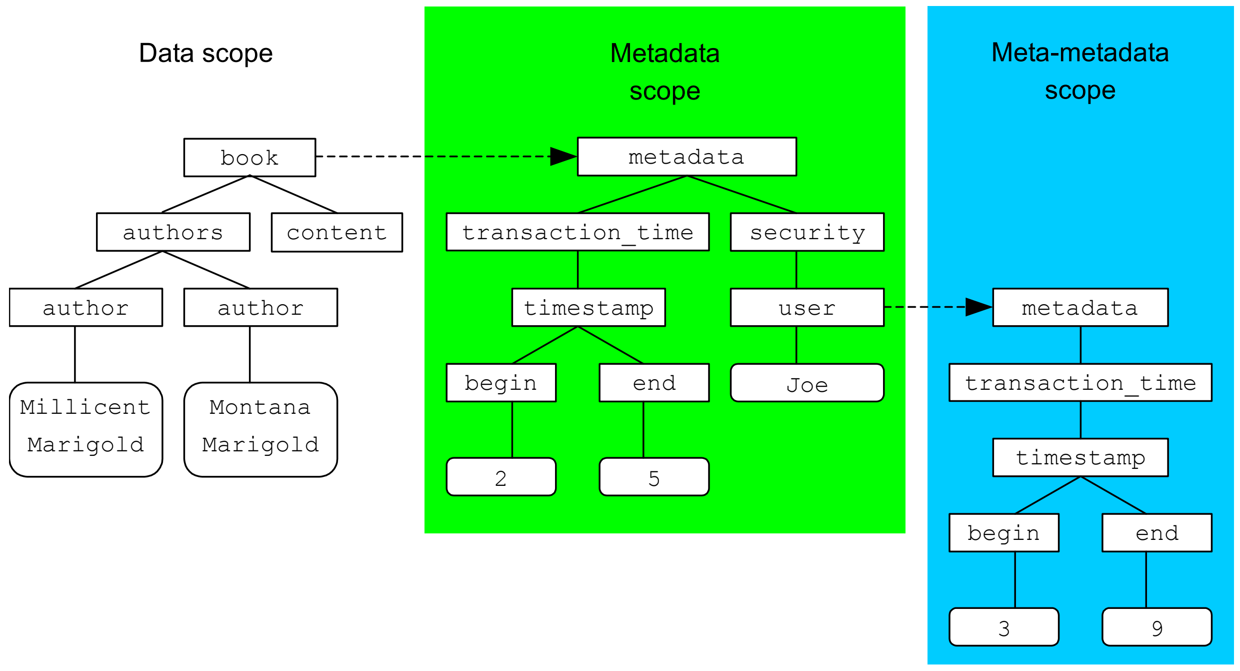


Figure 5 A part of the MetaDOM for the online publisher

Now let's take a look at the challenges that we have with this metadata-aware data model. The first challenge is that the query language has to be aware of the metadata and distinguish data from metadata in query evaluation. We can ask queries about data only, data with metadata conditions, metadata with data conditions, or metadata only. The scopes must be kept separate.

The second challenge is determining the "reachability" of nodes annotated with metadata. In a instance of a data model for a traditional, well-formed XML document every node is reachable from the root. But proscriptive metadata can impose constraints on the data that affect the validity of the path descending from the root to a node. For example, if a parent node exists in the document from times 2 to 5, then its children should only exist within that time period as well since ostensibly, a child cannot exist independent of its parent in any XML data model instance. If one of the children is timestamped with a completely different period, there is something wrong with the metadata in the document. The challenge is to identify problematic metadata and clean it up, if possible.

The third challenge is enforcing the semantics of metadata in path navigation. For instance, if a particular user, say Joe, queries the data model instance, then Joe should only see the part of the document to which he has access. Or if the user wants to query the current version of the data, he should only see the data that has a transaction time including *now*. A metadata-aware system must enforce the semantics of proscriptive metadata, or it will generate incorrect answers. The process of *sanitization* imposes a specific metadata perspective throughout a data model instance. *Eager sanitization* will construct a new instance limited to the selected metadata perspective, which is similar to generating a (materialized) view of the data based on a set of metadata conditions. A user can subsequently query the view without explicitly giving the metadata conditions. In contrast, *lazy sanitization* will sanitize during path expression evaluation.

The fourth challenge is grouping, which is an important operation in restructuring XML documents. There is no grouping construct in XQuery like the GROUP BY clause in SQL. But the distinct-values function can be used to group by determining which nodes have the same "value". The sample data in Figure 5 is organized from the book's perspective, where the authors are listed for each book. So while each book in the document is distinct, an author can be duplicated at several locations if they write more than one book. The data can be restructured to suit an author's perspective by listing the books each author writes. There are several limitations to grouping in this way. First, the distinct-values function of XQuery only returns the first node (in document order) from the set of nodes that have the same value, discarding the others. Sometimes the discarded members of the group are needed, for instance if we want to know the size of each group. The second limitation arises when metadata is considered. The metadata could change the grouping. For instance we may not want to group data that appears in different versions of a document, i.e., that has different transaction time metadata.

We propose several extensions to XQuery to meet these challenges. We call the extensions, collectively, MetaXQuery. Figure 6 lists four sample queries that are not supported in XQuery using the new data model, but are by MetaXQuery. ***Q1*** is a query on metadata with data conditions. ***Q2*** is a query on data with metadata condition. ***Q3*** is a restructuring query that needs to group on the data values. ***Q4*** is a restructuring query that groups metadata by data conditions. We will give the solutions to these queries in the following chapters.

Q1**:** When was book 1 in the document?
Q2**:** Which books are available to the user Joe online?
Q3**:** List authors and the books that they've written, retaining all metadata.
Q4**:** List the books in the document grouped by when they were in the document.

Figure 6 Sample queries not supported by XQuery

CHAPTER THREE
METADOM AND METAXQUERY

This chapter presents the theoretical foundations of our research. We first briefly introduce the Document Object Model (DOM) and XQuery. Next we extend DOM to support metadata, an extension we call MetaDOM. We then describe a series of query language extensions that we call collectively MetaXQuery. The first extension is MetaXPath, which adds a new `meta` axis to XPath. The rest of the extensions are in the form of functions, which include certifying the reachability of data nodes, sanitizing a data model, grouping data with metadata, constructing metadata, and matching an implicit metadata perspective during path evaluation.

3.1 Document Object Model (DOM) and XQuery

Whenever we want to query, process, or manipulate the textual XML documents, we need to build a logical data model of the document first and then operate on that logical model. While XML itself doesn't define any data model, the most popular data model for XML so far is the W3C's Document Object Model (DOM). According to the W3C, DOM is a "platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page" [76]. Usually it is an application programming interface (API) for valid HTML and well-formed XML documents.

Another commonly referred XML data model is the XPath data model [86], which is also defined by the W3C. The XPath data model is based on the XML information set [81]. It is very similar to the DOM data model except that the XPath data model only specifies how to access the information items in an XML document, but not how to update and manipulate it, and the definitions relate to the names and values of nodes are also slightly different in DOM and XPath data model. Most XML processing tools support DOM as their API, so later on we will just use DOM to refer to the data model and programming interface to access and process the contents of an XML document or data collection.

Just like SQL as the query language for database, XQuery [85] is a programming language defined jointly by the W3C XML Query Working Group and the XSL Working Group, to query collections of XML data. It provides a mechanism to extract and manipulate data from XML documents or any data source that can be viewed as XML such as relational databases or office documents. XQuery is designed to be a language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including structured and semi-structured documents, relational databases and object repositories. It is first derived from an XML query language called Quilt [12], which in turn borrowed features from several other languages, including XPath 1.0 [84], XQL [51], XML-QL [19], SQL [74], and OQL [11].

XQuery uses XPath syntax to address specific parts of an XML document. It also has a feature called a FLWOR expression that supports iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The name FLWOR, pronounced "flower", is suggested by the keywords `for`, `let`, `where`, `order by`, and `return`.

The `for` and `let` clauses in a FLWOR expression generate an ordered sequence of tuples of bound variables, called the **tuple stream**. The optional `where` clause serves to filter the tuple stream, retaining some tuples and discarding others. The optional `order by` clause can be used to reorder the tuple stream. The `return` clause constructs the result of the FLWOR expression. The return clause is evaluated once for every tuple in the tuple stream, after filtering by the where clause, using the variable bindings in the respective tuples. The result of the FLWOR expression is an ordered sequence containing the results of these evaluations

The following example of a FLWOR expression includes all of the possible clauses. The `for` clause iterates over all the departments in an input document, binding the variable `$d` to each department number in turn. For each binding of `$d`, the `let` clause binds variable `$e` to all the employees in the given department, selected from another input document. The result of the `for` and `let` clauses is a tuple stream in which each tuple contains a pair of bindings for `$d` and `$e`

($d is bound to a department number and $e is bound to a set of employees in that department). The where clause filters the tuple stream by keeping only those binding-pairs that represent departments having at least ten employees. The order by clause orders the surviving tuples in descending order by the average salary of the employees in the department. The return clause constructs a new big-dept element for each surviving tuple, containing the department number, headcount, and average salary.

```
for $d in fn:doc("depts.xml")/depts/deptno
let $e := fn:doc("emps.xml")/emps/emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
   <big-dept>
      {
      $d,
      <headcount>{fn:count($e)}</headcount>,
      <avgsal>{fn:avg($e/salary)}</avgsal>
      }
   </big-dept>
```

3.2 MetaDOM

Now let's talk about our extensions to the DOM and XQuery standards to support metadata. MetaDOM extends DOM by adding an optional meta property to a node's information set. But in all other respects, the data model is the same as the existing model. The value of the meta property is a reference to the root node of a nested MetaDOM, which contains the metadata for this node. The metadata can be recursively nested, that is, a node in a MetaDOM may itself have a meta property. Each level of nesting adds another level of metadata.

Below is a formal definition of the MetaDOM node constructor. We adopt the notation used by the W3C (e.g., in [86]) to express all XQuery and DOM extensions used in this dissertation.

Definition [node constructor] *Each node in MetaDOM is an extension of a DOM node. It has all of the properties in the node's information set, and adds a meta property. A MetaDOM node is cloned from a DOM node using the following constructor. The parameters are the node name, the sequence of its children, the metadata to add, and the data node to clone.*

```
meta-dm:node($name as xs:string, $children as node()*, $metadata as node(),
             $data as node()) as meta-dm:node()
```

∎

The `meta-dm` namespace is MetaDOM's counterpart of the `dm` namespace used in XQuery's data model [86]. Unqualified parameters can be either in the `dm` or `meta-dm` namespace. We will use the `meta-dm` namespace to encapsulate every definition related to MetaDOM. The other new namespaces that are introduced in this dissertation are: `meta-dt`, which is used for new data types defined for our framework; `meta-fn`, which is used for our metadata-related functions; and `ext-fn`, which is used for our new, data-related functions.

Since the meta property in a MetaDOM node is a reference to a piece of metadata, it is straightforward to share metadata across multiple data nodes. In other words, more than one data node can point to the same metadata. In this way, MetaDOM can achieve good space efficiency by representing shared metadata only once.

To access the metadata, we define an accessor function for the meta property.

Definition [meta accessor] *The meta accessor returns either a meta element node, which is the root of the metadata fragment that's associated with the input node if it exists, or an empty sequence if there's no metadata describing the input node.*

```
meta-dm:meta($node as meta-dm:node()) as meta-dm:node()?
```

∎

Another extension is a special type of node as the result of our merging function, which we will call *MergedNode*. The MergedNode is inherited from the MetaDOM node and adds a list of parents to its information set. The list of parents are the original parents of the nodes that are merged into this one. We will see how it is used later on in the merging function. Right now we will just give its formal definition.

Definition [MergedNode constructor] *The MergedNode is an extension of a MetaDOM node. It has all properties in the node's information set, and adds a parentList property. The type of parentList is a list of nodes. The constructor for a MergedNode takes one more parameter than the MetaDOM node constructor, which is a list of parent nodes.*

```
meta-dm:MegedNode($name as xs:string, $children as node()*, $parentList as
node()*, $metadata as node(), $data as node()) as meta-dm:node()
```

∎

3.3 MetaXQuery

This section presents extensions to XQuery to support querying of data and metadata. The extensions include adding a meta axis in XPath expressions and a set of functions for certifying, sanitizing, filtering, and explicit grouping of both data and metadata. The additional functions can be implemented by extending an XQuery evaluation engine, or in a programming language like Java and loaded into XQuery as external functions. Parts can also be written directly in XQuery as user functions. We will leave it to the implementation to decide how best to achieve these functions in an XQuery implementation. This section also introduces the metadata operation matrix. The matrix records the special semantics of each kind of metadata.

3.3.1 MetaXPath

In order to make the meta accessor available to users of MetaXQuery we adopt the same approach used in MetaXPath [23] and add a new axis: the meta axis. The meta axis follows the meta property of the context node to its metadata. We will use the notation "`meta::`" to denote the meta axis. For example, "`meta::node()`" will locate the metadata's root node from the context node. The meta axis will sometimes be abbreviated as "^". In the data model instance of Figure 5, the MetaXPath expression "`/book^/transaction_time`" will locate the transaction time element(s) in the metadata of the book element.

The meta axis is orthogonal to other XPath axes. In other words, the metadata can only be located through the meta axis. This ensures the separation of scope between data and metadata. So the first sample query in Figure 6 (***Q1:*** "When was book 1 in the document?") can be constructed in MetaXPath as follows.

Q1: `//book[@number="1"]^/transaction_time`

Note that traditional XPath expressions can still be evaluated in MetaXPath. The query will just ignore the metadata. So it is fully upwards compatible with XPath.

3.3.2 Metadata Operation Matrix

Since different metadata properties can have different semantics and constraints on the data, and introducing new metadata properties can add new semantics to the application, our framework uses an operation matrix that allows a metadata management system to tailor the semantics of each kind of metadata. Below, several common kinds of metadata are listed. Since our framework is extensible, database administrators can define new kinds of metadata as needed.

Time. Temporal metadata involves two time dimensions: *valid time*, which is the real world time of a datum, and *transaction time*, which is the time it is stored in the system. For our purposes, their semantics are similar so we will treat them as a generic time dimension. The time will be a set of intervals (a temporal element [34]).

Security. The security metadata that we will use in this dissertation is a list of the users that are allowed to access the corresponding data and its descendants. Users who are not listed cannot access the data. We would like to emphasize that this is only one possible security model; other kinds of security can be supported as well. There is no limit to the number or kinds of metadata.

Reliability. Reliability is a measurement of the quality of the data. It could be a ranking of its quality by experts or users, or trustworthiness of the information source. We will use it as a numeric scale that ranges from 0 to 1.

Language. Language is the natural language in which the data is written, e.g., English.

Table 1 lists the data type for each kind of metadata. The `meta-fn:meta-value` function retrieves a value of the specified return type for the indicated kind of metadata.

Property	**`meta-fn:meta-value(node()*)` return type**
Time	A list of time intervals, e.g., [0, 6], [2, 4]
Security	A set of strings, each string is a user name, e.g., “Joe”, “Susan”
Reliability	A floating point value between 0 and 1
Language	A string

Table 1 Types of metadata property value

Table 2 is one possible operation matrix for the metadata properties used in this dissertation. The matrix is specific to a metadata application, so other semantics can be defined as needed. New metadata properties can also be added. We will assume that the operations in the matrix are implemented as (external) functions. In MetaXQuery functions the matrix will be represented as a hash table, denoted `meta-dt:MetaFNHash`. The hash table maps the name of the metadata property (e.g., time, security) and the desired operation to the metadata-specific operation. For instance, an application would use temporal-coalescing when coalescing time metadata in the evaluation of a MetaXQuery query. Rather than defining each operation now, we will describe and define each operation when it is used in this dissertation.

Operation	**Time**	**Security**	**Reliability**	**Language**
Certifying	certify-subset	certify-subset	true	true
Sanitizing	sanitize-intersection	sanitize-intersection	no-op	no-op
Meta-Grouping	group-equality	true	true	group-equality
Coalescing	temporal-coalescing	coalesce-union	coalesce-max/min	no-op
Filtering	filter-intersection	filter-intersection	no-op	filter-equality

Table 2 The operation matrix for metadata properties

3.3.3 Certifying Data Reachability

The **certify** function checks the consistency of the metadata between parent and child nodes in a data model, or in other words, whether a child can be reached from a parent. Given a parent node and its metadata, the metadata of each child must be consistent with the metadata of the parent in order for it to be reachable. The consistency check for each kind of metadata is defined in the operation matrix. If the check is satisfied, then the metadata is consistent.

Definition [certify] *The certify function takes a node of a MetaDOM tree and checks if every node under it is reachable. It returns a Boolean value indicating whether the test is successful.*

```
meta-fn:certify($fn as meta-dt:MetaFNHash,
                $node as meta-dm:node()) as xs:Boolean
```

$$= \left\{ \begin{array}{ll} \mathit{True} & \text{If } ((\forall v \in \texttt{\$node/*}, (\mathit{fn{:}exists}(\texttt{\$node\^{}/*} \mid v\texttt{\^{}/*}) \Rightarrow \\ & \quad (\forall m \in \texttt{\$node\^{}/*} \mid v\texttt{\^{}/*}, (\texttt{\$fn} \rightarrow \{m\}(\texttt{\$node\^{}/}m, v\texttt{\^{}/}m)))) \wedge \\ & \quad (\mathit{meta\text{-}fn{:}certify}(\texttt{\$fn}, v))) \wedge (\mathit{fn{:}exists}(\texttt{\$node\^{}}) \Rightarrow \mathit{meta\text{-}fn{:}certify}(\texttt{\$fn},\texttt{\$node\^{}}))) \\ \mathit{False} & \text{Otherwise.} \end{array} \right\}$$

∎

The certifying process starts from the input node and first checks the metadata relationships between this node and its children. After all the children have been checked, it recursively calls itself on each child to check the next level down the tree. If anywhere in the tree, an incompatible metadata value is found, the function will immediately return false, which means that some descendent of the input node is unreachable. The descendent can be found and removed by sanitizing the data model instance as described in the next section. Since there may be several levels of metadata (e.g., meta-metadata), certify also recursively checks each level of metadata for consistency.

Definition [metadata-specific certifying functions] *These functions are used to certify specific kinds of metadata. The certify-subset function makes sure the metadata value of the child is a subset of its parent's.*

```
meta-fn:certify-subset($pNode as meta-dm:node(),
                       $cNode as meta-dm:node()) as xs:Boolean
```

$$= \begin{cases} \textit{True} & \text{If } \textit{meta-fn:meta-value}(\texttt{\$cNode/*}) \subseteq \textit{meta-fn:meta-value}(\texttt{\$pNode/*}) \\ \textit{False} & \text{Otherwise.} \end{cases}$$

∎

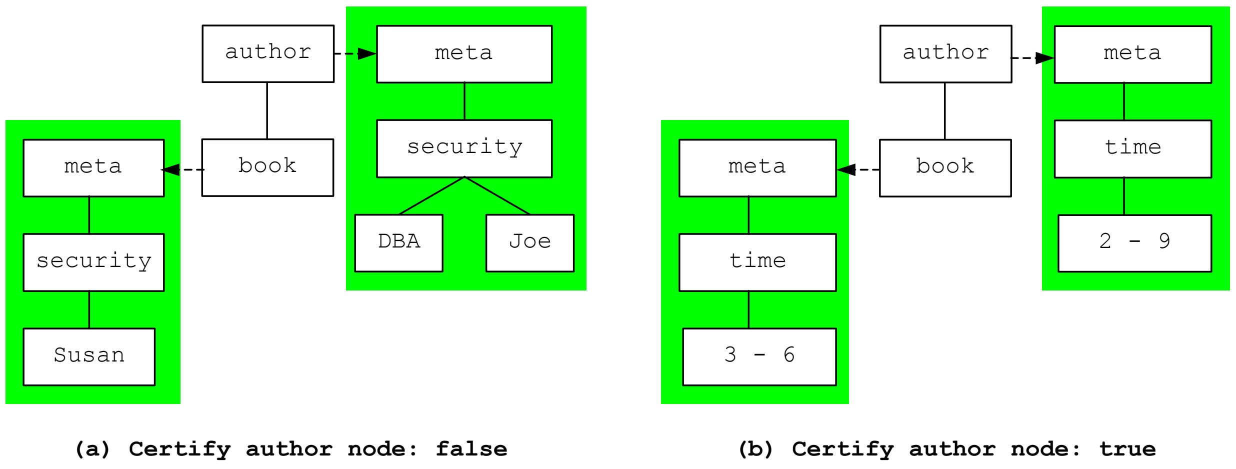


Figure 7 Certifying an author node

An example of the **certify** function is shown in Figure 7. The example shows certifying with security metadata, and separately, with time metadata (typically all kinds of metadata are certified together). In Figure 7(a), the security value of child node book (Susan) is not a subset of the security value of its parent (DBA, Joe), so the certify function will fail and report that the metadata is incorrectly embedded. In Figure 7 (b), the time value of the child node ([3,6]) is a subset of its parent's valid time value ([2,9]), so the certify function will return true in this case.

Though the "subset" operation is used for each of these kinds of metadata, in general, each kind of metadata might have a different certification operation as specified by the operation matrix.

3.3.4 Sanitizing a Data Model Instance

When certification fails due to inconsistent metadata, then the instance of a data model must be pruned to make it consistent. We introduce a **sanitize** function to automatically construct a instance that passes certification. The **sanitize** function could also be used to generate a view of the data based on certain metadata conditions by sanitizing with respect to a selected metadata perspective. So for queries like ***Q2:*** "What books can Joe access?" one possibility is to set the metadata for the root to be user Joe and then sanitize the entire data collection (a better, related method is described in the next section). After doing so a simple XPath query like "`//book`" on the sanitized data model instance will locate books that are available to Joe.

Definition [sanitize] *The sanitize function takes an input node, which is the root of the document/fragment and returns the root of the new valid document/fragment.*

```
meta-fn:sanitize($fn as meta-dt:MetaFNHash,
                 $node as meta-dm:node()) as meta-dm:node()
```

$$
\begin{aligned}
&= \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$node}), (v_{i1}, v_{i2}, \ldots v_{in}), \textit{meta-fn:sanitize}(\texttt{\$fn}, \texttt{\$node\^{}}), \texttt{\$node}) \wedge \\
&\qquad \forall j, 1 \le j \le n, v_{ij} = \textit{meta-fn:sanitize}(\texttt{\$fn}, v_{ij}) \mid i_1 < i_2 < \ldots < i_n \wedge \forall j, 1 \le j \le n, \\
&\quad v_{ij} = \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$node/}*[i_j]), \textit{dm:children}(\texttt{\$node/}*[i_j])), \\
&\qquad \textit{meta-fn:meta-sanitize}(\texttt{\$fn}, \texttt{\$node\^{}}, \texttt{\$node/}*[i_j]\texttt{\^{}}), \texttt{\$node/}*[i_j]) \\
&\quad \wedge \forall m \in v_{ij}\texttt{\^{}/}*, (\textit{meta-fn:meta-value}(v_{ij}\texttt{\^{}/}m) != \text{NULL})
\end{aligned}
$$

∎

Definition [meta-sanitize] *The meta-sanitize function takes an operation matrix and a pair of meta element nodes from a parent and a child node, and creates a new meta element node for the child. The new node represents a metadata fragment in which all its values are compatible with the parent node's metadata.*

```
meta-fn:meta-sanitize($fn as meta-dt:MetaFNHash, $pmNode as meta-dm:node(),
                      $cmNode as meta-dm:node()) as meta-dm:node()
```

$$
\begin{aligned}
&= \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$cmNode}), \\
&\quad (\forall m \in (\texttt{\$pmNode/}* \mid \texttt{\$cmNode/}*), \texttt{\$fn} \rightarrow \{m\}(\texttt{\$pmNode/}m, \texttt{\$cmNode/}m)), \texttt{\$cmNode\^{}}, \texttt{\$cmNode})
\end{aligned}
$$

∎

The **sanitize** function starts by checking the metadata of the input and its children. If the processing of a child's metadata evaluates to NULL, then the child is unreachable from its parent and is removed from the data model instance. Sanitize then recursively calls itself on each remaining child. Internally, the **sanitize** function calls the meta-sanitize function, which modifies the metadata of a child node to be compatible with its parent's.

Definition [metadata-specific sanitize functions] *These functions are used to sanitize specific kinds of metadata. The sanitize-intersection function calculates the intersection of two metadata values of the same type and creates a new fragment representing that value.*

```
meta-fn:sanitize-intersection($pNode as meta-dm:node(),
                               $cNode as meta-dm:node()) as meta-dm:node()
```

$= \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$cNode}), (v_1, v_2, \ldots v_n), \texttt{\$cNode\^{}}, \texttt{\$cNode})$

where $v_i \in \texttt{\$cNode/*} \wedge \textit{meta-fn:meta-value}(v_i) \in \textit{meta-fn:meta-value}(\texttt{\$pNode/*})$

∎

3.3.5 Filtering by Metadata Perspective

Each query is allowed access to only the data that matches the query's metadata perspective. The perspective is often implicit in a user's session, but can also be made explicit for a particular query. The perspective is established as a user acquires access rights, sets the current time, chooses a natural language, or performs some action that sets a query context or session parameter. To restrict data to a particular metadata perspective, the data can be sanitized using that perspective as discussed in the previous section. However, sanitizing is somewhat expensive since it constructs a new data model instance. Typically, an instance should be sanitized only after a metadata update. The **filterByPerspective** function offers a lightweight, inexpensive alternative. The function dynamically matches a perspective to the metadata along a path during query evaluation. It is applied to every path expression in a MetaXQuery program. Though semantically, the **filterByPerspective** is equivalent to sanitizing a subtree of a MetaDOM, it can be implemented more efficiently.

For each of the metadata types in the perspective fragment, the **filterByPerspective** function calls the corresponding metadata-specific filter function to check if its value matches that of the data node's. If it does, the data node will be retained. If not, the data node is thrown away. It also checks recursively if there's meta-metadata.

Definition [filterByPerspective] *The filterByPerspective function takes a sequence of data nodes and the root node of a perspective MetaDOM. It filters the data sequence, keeping only those nodes that match the perspective.*

```
meta-fn:filterByPerspective($fn as meta-dt:MetaFNHash,
                            $seq as meta-dm:node()*,
                            $pNode as meta-dm:node()) as meta-dm:node()*
```

$$= (v_1, v_2, \ldots, v_n) \mid \forall i, v_i \in \texttt{\$seq} \wedge$$
$$(\forall m \in \texttt{\$pNode\^{}/*},\ \texttt{\$fn} \rightarrow \{m\}(\texttt{\$}v_i\texttt{\^{}/}m, \texttt{\$pNode\^{}/}m) \wedge$$
$$(\forall pv \in \texttt{\$pNode\^{}//*},\ (\mathit{fn{:}exist}(pv\texttt{\^{}/*}) \Rightarrow \mathit{meta\text{-}fn{:}filterByPerspective}(v_i\texttt{\^{}//}pv, pv\texttt{\^{}}) \neq \varnothing))$$

∎

Definition [metadata-specific filter functions] *These functions are used to match specific kinds of metadata in the filterByPerspective function. Filter-intersection function tests if the two input metadata values intersect with each other. Filter-equality tests for equality of two metadata values.*

```
meta-fn:filter-intersection($node1 as meta-dm:node(),
                            $node2 as meta-dm:node()) as xs:Boolean
```

$$= \begin{cases} \mathit{True} & \text{If } \mathit{meta\text{-}fn{:}meta\text{-}value}(\texttt{\$node1}) \cap \mathit{meta\text{-}fn{:}meta\text{-}value}(\texttt{\$node2}) \neq \varnothing \\ \mathit{False} & \text{Otherwise.} \end{cases}$$

```
meta-fn:filter-equality($node1 as meta-dm:node(),
                        $node2 as meta-dm:node()) as xs:Boolean
```

$$= \begin{cases} \mathit{True} & \text{If } \mathit{meta\text{-}fn{:}meta\text{-}value}(\texttt{\$node1}) = \mathit{meta\text{-}fn{:}meta\text{-}value}(\texttt{\$node2}) \\ \mathit{False} & \text{Otherwise.} \end{cases}$$

∎

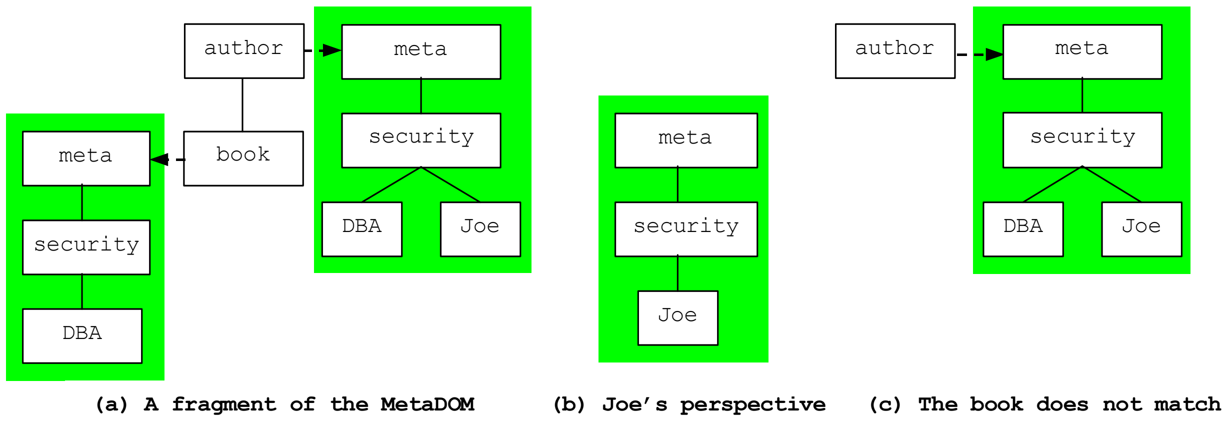


Figure 8 Filtering from Joe's perspective while navigating a path

Figure 8 shows an example of the **filterByPerspective** function. A query, such as '`//book`', will navigate through an author element to reach a book node. The navigation will match the perspective shown in Figure 8 (b), which consists of Joe's security certificate, against the metadata on each node in a path to a book element. The author node will qualify, but the book node will fail the match as illustrated in Figure 8 (c).

3.3.6 Grouping in MetaXQuery

The grouping function is divided into several sub-functions. We make these sub-functions available to users in MetaXQuery in order to provide fine-grained control in grouping. The sub-functions are data-group, meta-group, merge, and coalesce. The data-group function groups nodes based on their data values. The meta-group function groups nodes based on their metadata values. In grouping, the output of data-group is typically input to meta-group. The merge function merges all of the nodes in a group into a single node. It also merges their metadata. The coalesce function then coalesces the merged metadata, removing redundant metadata. Finally, the categorize function effects grouping by performing the sub-functions in sequence.

3.3.6.1 Data Grouping

The **data-group** function groups nodes that have the same value. The value is computed by XQuery's `distinct-values` function, but unlike `distinct-values`, all of the nodes with the same value are retained in a group (in document order).

Definition [data-group] *The data-group function takes a Sequence of nodes and returns a list of groups, where each group is a Sequence.*

```
ext-fn:data-group($seq as meta-dm:node()*) as (meta-dm:node()*)*
```

$$= (s_1, s_2, \ldots, s_n) \mid s_i = (v_{i1}, v_{i2}, \ldots, v_{im}) \text{ where } \textit{fn:distinct-values}(s_i) = \textit{fn:distinct-values}(v_{i1}) \wedge$$
$$\forall j,k, 1 \le j,k \le n, j \ne k, (\textit{fn:distinct-values}(s_j) \ne \textit{fn:distinct-values}(s_k))$$

∎

The return value is an extension of XQuery's Sequence type. In the standard data model a Sequence is an ordered collection of zero or more nodes or atomic values [86]. Extending this to a list of groups or Sequences is straightforward. In this grouping section, we will use `(meta-dm:node()*)*` to denote a list of groups. At a query language level this means that two nested FOR statements will be needed to iterate through the list. Finally, we should note that each group in the list is internally in document order, but the order of the groups is implementation defined.

3.3.6.2 Metadata Grouping

The result of data grouping is passed to the meta-group function for further partitioning based on the metadata. Sometimes the metadata prevents grouping. For example, suppose two temperature values are the same, but one has metadata that shows it was measured in the Celsius scale, while the other is in degrees Fahrenheit. The temperatures should be placed into different groups. The **meta-group** function partitions a group into subgroups based on the metadata. The semantics of meta-grouping is parameterized by the kind of metadata. For example, while nodes with different temperature metadata should be placed in different groups, perhaps nodes with different reliabilities could be placed in the same group.

Definition [meta-group] *The meta-group function takes a sequence of nodes and returns a list of sequences. All nodes in a sequence have the "same" data and metadata.*

```
meta-fn:meta-group($fn as meta-dt:MetaFNHash,
                   $seq as meta-dm:node()*) as (meta-dm:node()*)*
```

$$= (s_1, s_2, \ldots s_n) \mid \forall i,\ (s_j \subseteq \texttt{\$seq} \wedge (\forall v_j, v_k \in s_i, (\textit{meta-fn:meta-match}(\texttt{\$fn}, v_j, v_k))))$$

∎

Internally, **meta-group** calls **meta-match**. The **meta-match** function, which is defined below, compares the metadata to determine if a pair of nodes can be in the same group. The comparison operation is chosen from the operation matrix (the `MetaFNHash`).

Definition [meta-match] *The meta-match function takes an operation matrix and a pair of nodes, and evaluates whether the metadata in the nodes "matches".*

```
meta-fn:meta-match($fn as meta-dt:MetaFNHash, $node1 as meta-dm:node(),
                   $node2 as meta-dm:node()) as xs:Boolean
```

$$= \left\{ \begin{array}{ll} \textit{True} & \text{If } (\textit{fn:exists}(\texttt{\$node1\^{}/*} \mid \texttt{\$node2\^{}/*}) \Rightarrow \\ & \quad (\forall m \in \texttt{\$node1\^{}/*} \mid \texttt{\$node2\^{}/*}, (\texttt{\$fn} \rightarrow \{m\}(\texttt{\$node1\^{}/}m, \texttt{\$node2\^{}/}m)) \wedge \\ & \quad \textit{meta-fn:meta-match}(\texttt{\$fn}, \texttt{\$node1\^{}/}m, \texttt{\$node2\^{}/}m))) \\ \textit{False} & \text{Otherwise.} \end{array} \right\}$$

∎

If there are multiple levels of metadata, **meta-match** recursively explores each level. Every level of metadata should match. The following definitions are for metadata-specific matching functions that appear in the operation matrix of Table 2.

Definition [metadata-specific matching functions] *These functions are used to match specific kinds of metadata. The true function always reports a successful match. Group-equality tests for equality of two metadata values.*

```
meta-fn:true($node1 as meta-dm:node(),
               $node2 as meta-dm:node()) as xs:Boolean
```

= *True*.

```
meta-fn:group-equality($node1 as meta-dm:node(),
                          $node2 as meta-dm:node()) as xs:Boolean
```

$$= \begin{cases} True & \text{If } meta\text{-}fn{:}meta\text{-}value(\$node1) = meta\text{-}fn{:}meta\text{-}value(\$node2) \\ False & \text{Otherwise.} \end{cases}$$

∎

These are only two of the many possible metadata-specific matching functions; which function to use depends on the semantics of the kind of metadata. However, to form groups such functions should be symmetric, transitive, and reflexive.

Let's use time as an example to show how the **meta-group** function works. The semantics of time for the meta-group function in the operation matrix is equality, so nodes in a group after the meta-group function will have the same time. In Figure 9, three book nodes with the same data value (grouped by **data-group**) are subdivided into two groups by **meta-group**.

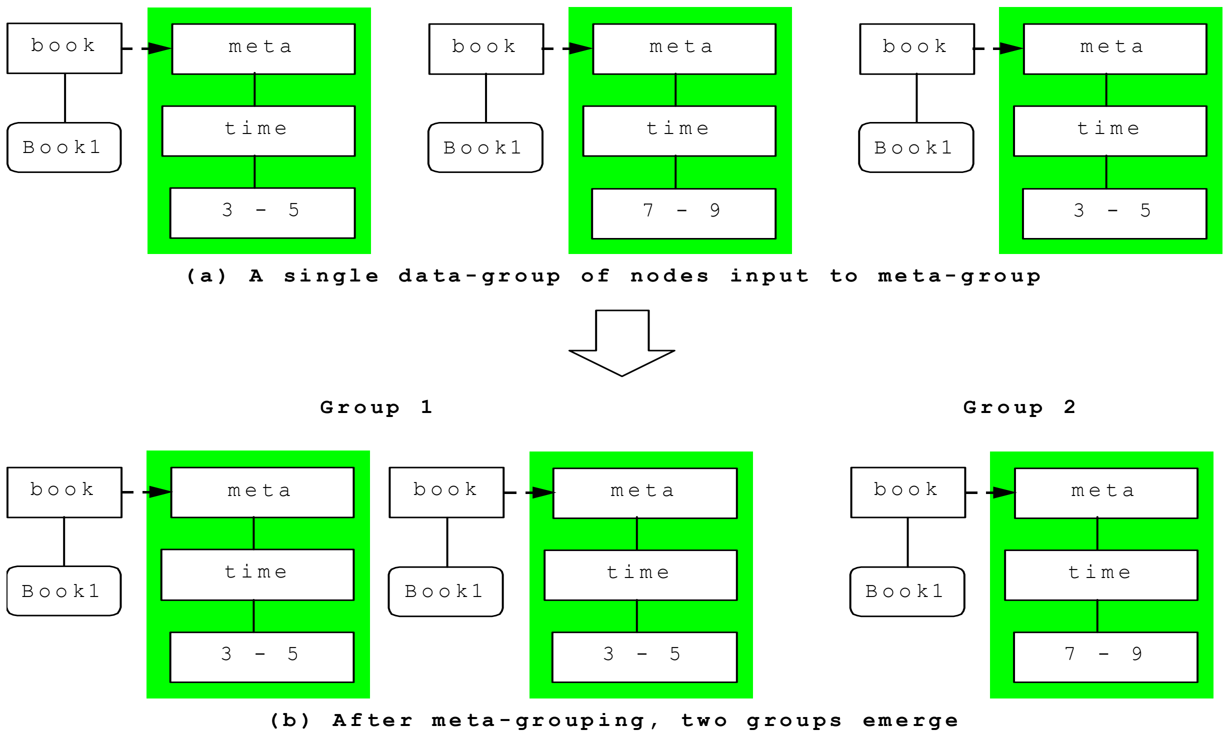


Figure 9 An example of meta-group

3.3.6.3 Merging

Merging is the process of merging a group of nodes into a single, new node. It is invoked after the nodes with the same data value have been divided into groups according to their metadata. It is straightforward to merge the data since every node has exactly the same data. But with metadata, when nodes are merged, their metadata must also be combined to create the metadata for the merged node. Merge also retains the parents of the nodes being merged. The MergedNode class has an extra information item, the parentList, which keeps the list of the parents of the merged nodes. An example is shown in Figure 10. After merging the author Joe, the result looks like Figure 10(b). The new author node is now a MergedNode type. Its parentList points to the two book elements that are the parents of the two original author Joe. Note that parentList pointers are orthogonal to all other node accessors and consist of one-way pointers.

If there is more than one level of metadata, the **merge** function will recursively merge the metadata at each level. At each level it invokes the meta-merge function. The **meta-merge** function calls the merge function on each group of nodes representing one type of metadata.

Definition [merge] *The merge function takes a list of groups and returns a MergedNode sequence, which is a list of constructed, merged nodes.*

meta-fn`:merge($seq as` *`(meta-dm:node()*)*`*`) as` *`meta-dm:MergedNode()*`*

$$= (x_1, x_2, \ldots x_n) \mid (\forall i, 1 \le i \le n, (x_i = \textit{meta-dm:MergedNode}(\textit{dm:node-name}(\texttt{\$seq}[i][1]), (\textit{dm:children}(\texttt{\$seq}[i][1]), \textit{dm:children}(\texttt{\$seq}[i][2]), \ldots), \texttt{\$seq/..}, \textit{meta-fn:meta-merge}(\texttt{\$seq}[i]^\wedge), \texttt{\$seq}[i][1]))$$

∎

Definition [meta-merge] *The meta-merge function takes a sequence of meta element nodes, which are roots of the metadata fragments, and concatenates the values of each metadata type.*

meta-fn`:meta-merge($seq as` *`meta-dm:node()*`*`) as` *`meta-dm:node()`*

$$= x \mid \textit{fn:exists}(\texttt{\$seq}) \Rightarrow (x = \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$seq}[1]), (\forall m \in \bigcup \texttt{\$seq/*}, \textit{meta-fn:merge}(\texttt{\$seq/}m)), \textit{meta-fn:meta-merge}(\texttt{\$seq}^\wedge), \texttt{\$seq}[1])$$

∎

An example of merging is depicted in Figure 11. Initially, two author nodes end up in the same group Figure 11(a). Both authors have transaction time metadata. When the authors are merged, a new node is constructed as shown in Figure 11(b). The new node has all the children of the original author nodes (in document order), and its metadata is computed by **meta-merge**. **Meta-merge** merges two transaction times into a single transaction time. The two transaction times overlap, so some redundant metadata is present, but can be removed through coalescing, as described next.

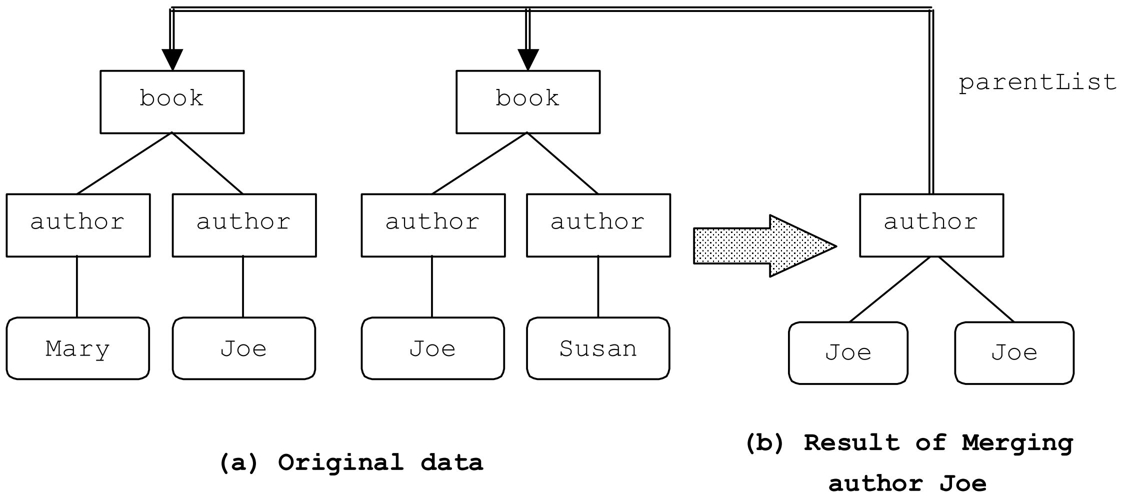


Figure 10 Example of MergedNode's parent list

3.3.6.4 Coalescing

The **merge** function simply concatenates metadata values, regardless of the different semantics for kinds of metadata. Coalescing refines the metadata, invoking metadata-specific semantics.

Definition [coalesce] *The coalesce function takes the operation matrix and a sequence of MetaDOM nodes and coalesces their metadata.*

```
meta-fn:coalesce($fn as meta-dt:MetaFNHash,
                 $seq as meta-dm:node()*) as meta-dm:node()*
```

$$= (v_1, v_2, \ldots, v_n) \mid v_i = \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$seq}[i]), \textit{dm:children}(\texttt{\$seq}[i]), \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$seq}[i]\texttt{\^{}}), (\forall m \in \texttt{\$seq}[i]\texttt{\^{}/*}, \texttt{\$fn} \rightarrow \{m\}(\texttt{\$seq}[i]\texttt{\^{}/}m), \textit{meta-fn:coalesce}(\texttt{\$fn}, \texttt{\$seq}[i]\texttt{\^{}})), \texttt{\$seq}[i])$$

∎

The nested levels of metadata are recursively coalesced by the **coalesce** function. But how to coalesce a particular kind of metadata depends on the semantics of that kind of metadata.

Internally, the coalesce function calls the metadata-specific coalescing operation as defined by the operation matrix. Two example functions are defined below.

Definition [Metadata-specific coalesce functions] *These functions are used to coalesce specific kinds of metadata. The coalesce-union function computes the union of the metadata values and the coalesce-max function computes the maximum of the metadata values.*

`meta-fn:coalesce-union($node as meta-dm:node()) as meta-dm:node()`

$= \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$node}), (v_1, v_2, \ldots, v_n), \texttt{\$node\^{}}, \texttt{\$node})$

where $\textit{meta-fn:meta-value}(v_1, v_2, \ldots, v_n) = \bigcup \textit{meta-fn:meta-value}(\texttt{\$node/*})$

`meta-fn:coalesce-max($node as meta-dm:node()) as meta-dm:node()`

$= \textit{meta-dm:node}(\textit{dm:node-name}(\texttt{\$node}), v, \texttt{\$node\^{}}, \texttt{\$node})$

where $\textit{meta-fn:meta-value}(v) = \max(\textit{meta-fn:meta-value}(\texttt{\$node/*}))$

∎

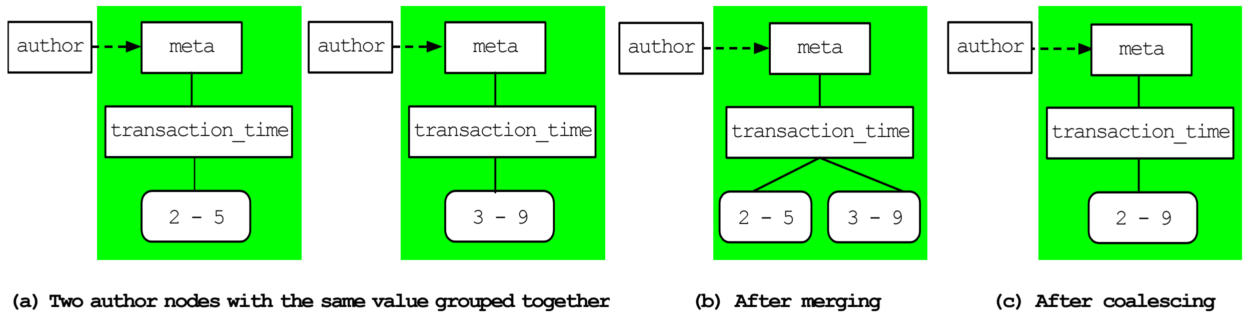


Figure 11 Example of grouping, merging, and coalescing

Figure 11(c) shows an example of coalescing. After merging the transaction time, coalesce is called. For a time dimension, coalescing is done by temporal-coalescing which computes disjoint, maximal intervals [21].

3.3.6.5 Categorizing

With the **data-group**, **meta-group**, **merge**, and **coalesce** functions defined, we are now able to define a **categorize** function that does all these four operations in sequence. The **categorize** function is useful because it gives users a simple, query-level grouping of both data and metadata. The first step of our categorize function is the data grouping function. The data groups are then input to meta-group which further divides the groups based on metadata conditions. Each of these groups is then merged into a single node and, finally, their metadata is coalesced.

Definition [categorize] *The categorize function takes an operation matrix and a sequence of MetaDOM nodes, and returns a list of singletons, each of which contains a merged, coalesced node.*

```
meta-fn:categorize($fn as meta-dt:MetaFNHash,
                   $seq as meta-dm:node()*) as meta-dm:node()*
```

$$= (s_1, s_2, \ldots, s_n) \mid s_i = \textit{meta-fn:coalesce}(\texttt{\$fn}, \textit{meta-fn:merge}(\texttt{\$fn}, \textit{meta-fn:meta-grouping}(\texttt{\$fn}, c_i)))$$

$$\text{where } (c_1, c_2, \ldots, c_n) = \textit{ext-fn:data-group}(\texttt{\$seq})$$

∎

Categorize is useful in answering sample query ***Q3*** in Figure 6, "List authors and the books that they've written retaining all metadata". The MetaXQuery is given in Figure 12. The query first categorizes every author. It then goes back to all the book elements that are ancestors of the author nodes with the same value. Here we see the benefit of keeping the parent list. The query will produce a list of all the books written by an author.

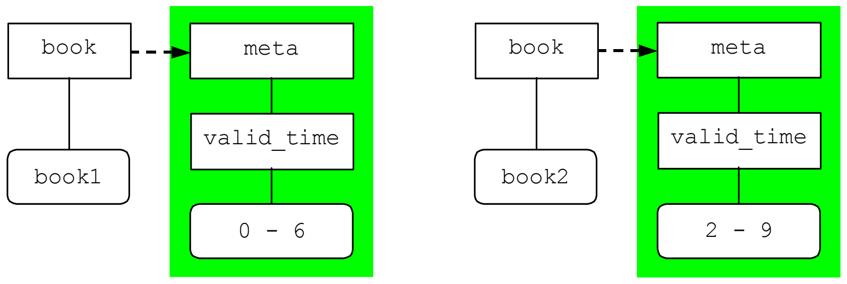
```
FOR $a IN meta-fn:categorize(document("books.xml")//author)
  RETURN
  <author>{$a/text()}
    <books>
      { RETURN meta-fn:getParentList($a)/../book/title }
    </books>
  </author>
```

Figure 12 MetaXQuery solution to Q3

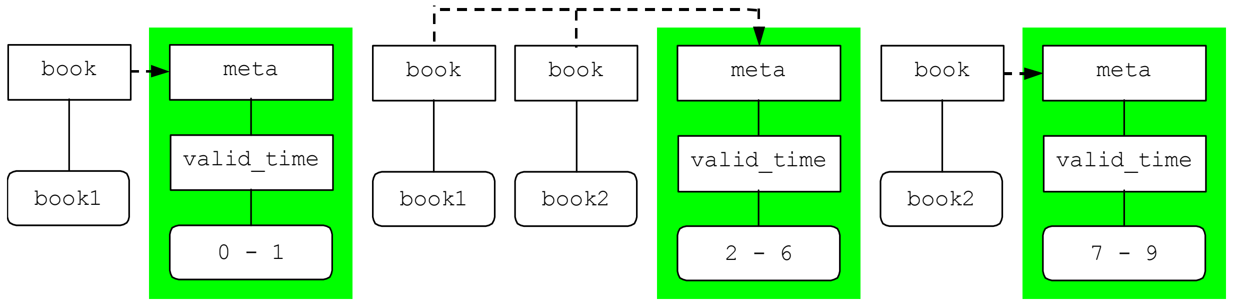
3.3.6.6 Grouping metadata, coalescing data

Categorize partitions the data into groups with distinct data values, and then processes the metadata of each group to reflect the reorganization of the data. The key benefit of grouping is that it provides a view from the perspective of the grouped data. Consider the MetaXQuery solution to ***Q3***. After the authors are categorized, there's only one group for the author "Millicent Marigold".

Users might also want to group from other perspectives such as grouping the metadata while coalescing the data. Consider the data and metadata fragments in Figure 13(a). The figure shows two groups with distinct data values, i.e., the result of normally grouped books. Consider a query to compute which books coexist in each version the document. The data is such a way that a concise query can be used to determine the coexistence. Figure 13(b) shows a better organization, which is the result of grouping using the metadata and then coalescing the data.



(a) Two book nodes before grouping from metadata's perspective



(b) After grouping from metadata's perspective

Figure 13 Example of grouping from the metadata's perspective

This section describes the **categorize-meta** function that is similar to the categorize function, but groups using the metadata.

Definition [categorize-meta] *The categorize-meta function takes a sequence of MetaDOM nodes and the name of a metadata property. It reorganizes the nodes from the perspective of the named metadata property, returning a list of groups. Each node in the group has the same metadata.*

```
meta-fn:categorize-meta($seq as meta-dm:node()*,
                        $prop as xs:string) as (meta-dm:node()*)*
```

$$= (s_1, s_2, \ldots, s_n) \mid \forall i, ((s_i \subseteq \texttt{\$seq}) \wedge \forall v_j, v_k \in s_i, (v_j\hat{} = v_k\hat{})) \wedge \textit{meta-fn:meta-value}(s_i[1]) = x_i$$

$$\text{where } (x_1, x_2, \ldots, x_n) = \textit{meta-fn:meta-distinct-values}(\texttt{\$seq\^{}/prop})$$

∎

Categorize-meta preserves the original document order of nodes within each group, but the document order among groups is not important because the same data node can be shuffled into several groups. Categorize-meta is also a projection of the original document on a single metadata property because the other kinds of metadata will not appear in the result.

The `meta-fn:meta-distinct-values` function takes a sequence of metadata values (possibly duplicated or overlapping) and returns a set of distinct values. The implementation depends on whether the value of metadata is an interval type (such as time) or a point type (such as security, language, etc.). For point types, it is a union function that returns all the unique values from the list. For range types, it is more complicated. Basically the function has to determine the intersections of all of the intervals. For example, if the intervals are [0, 6], [2, 4], [1, 8], then the result will be [0, 1], [2, 4], [5, 6], [7, 8].

We are now able to provide the last sample query, Q4, in Figure 6: "List the books grouped by when they were in the document".

Q4: `meta-fn:categorize-meta(`

```
                document("books.xml")//book, "transaction_time")
```

The query reorganizes the books into a list of groups of book elements, each of which has a distinct transaction-time period.

3.3.6.7 Constructing Metadata

An important part of XQuery is the RETURN clause, which is used to format query results, usually as XML. Similarly MetaXQuery also has to format results, but formatting results is more complicated because data and metadata are in different scopes. Though metadata can be formatted as data without extending the RETURN clause, to maintain the scopes in a query result we add a new function, associateMetadata, to annotate data with formatted metadata.

Definition [associateMetadata] *The associateMetadata function takes a sequence of data nodes and the root node of a metadata tree, and assigns the metadata to each data node in the sequence. If the data node has metadata, associateMetadata will overwrite the original metadata.*

```
meta-fn:associateMetadata($dNodes as meta-dm:node()*,
                          $mNode as meta-dm:node())
                          as meta-dm:node()*
```

$$= (v_1, v_2, \ldots, v_n) \mid v_i = \mathit{meta\text{-}dm{:}node}(\mathit{dm{:}node\text{-}name}(\$dNodes[i]), \mathit{dm{:}children}(\$dNodes[i]), \$mNode, \$dNodes[i])$$

∎

Once the **associateMetadata** function has established a relationship in the result MetaDOM, a serializer can transform the data model instance into an appropriate XML representation. An example of metadata construction is given in Figure 14. The query presumes that the metadata is initially contained within a `<metadata>` element in each `<book>` (i.e., the data and metadata are in the same scope; the example comes from the first use case in the XQuery and XPath Full-text Use Cases document). The query locates book contents. It also locates book metadata. The return clause then builds a MetaDOM by moving the book metadata into the metadata scope and associating it to the each book content node (the book content stays in the data scope).

```
 <books>
   FOR $book IN fn:distinct-value(document("books.xml")//book)
   LET $content := {<book>{$book/content}</book>},
       $metadata := {<meta>$book/metadata/*</meta>}
   RETURN meta-fn:associateMetadata($content, $metadata)
</books>
```

Figure 14 Example of constructing metadata

3.3.7 Directly Implementing MetaXQuery in XQuery

MetaXQuery adds constructs and functions to XQuery to help programmers query metadata together with data, but MetaXQuery does not fundamentally enhance the expressiveness of XQuery (XQuery is Turing-complete [43]). This section shows how to reuse XQuery to support separating the metadata scope from the data scope and the *filterByPerspective* function.

A separate metadata scope can be implemented by the judicious use of namespaces. The idea is to put the metadata into a "`meta`" namespace, meta-metadata into a "`meta-meta`" namespace, and so on (this strategy assumes that the metadata does not already have a namespace). The scope is enforced in a query by rewriting every path expression in the query to circumscribe its scope. Consider a simple query to locate book data, e.g., "`FOR $b IN //book`". The expression must be rewritten to the following to filter out the metadata (and meta-metadata).

```
FOR $b IN //book
WHERE $b.namespace != 'meta' AND $b.namespace != 'meta-meta' AND …
```

Since users do not know *a priori* how many levels of metadata are present such a strategy is only possible to implement with a query pre-processor that transforms every path expression. In MetaXQuery we accomplished the same end by changing the DOM. The namespace strategy outlined above is less efficient at query time than supporting a meta axis in the data model, but has the advantage of not requiring any data model changes (however modest).

The *filterByPerspective* function can also be implemented directly in XQuery. Let's assume that there is only one kind of metadata. Consider once again a simple query to locate book data. When the query is evaluated each kind of metadata must be checked to ensure that the perspective matches each ancestor in the path to a node. Let's assume that there is only one kind of metadata, time, and that there is no meta-metadata. Further assume that the perspective is bound to the variable "`$p`", and that the "`contains`" function checks temporal containment.

```
FOR $b IN //book
LET $a := $b/ancestor::*
LET $t := $a//time
WHERE $b.namespace != 'meta' AND $t.namespace == 'meta'
      AND contains($t, $p//time) …
```

Every path expression in a query must be treated similarly. When other kinds of metadata are present the template outlined above must be expanded to consider each kind. Further, meta-metadata must also be checked. While this would be beyond the programming capabilities of most users, it could be implemented in a query pre-processor. Such a query pre-processor would complicate a query tremendously by adding many LET clauses and padding the WHERE clause with lots of conjuncts possibly impacting query optimization. Furthermore, the use of the ancestor axis would adversely impact query evaluation efficiency. We believe that it is better to package the functionality into *sanitize* and *filterByPerspective* function calls so these operations can be efficiently implemented in the back-end in a query evaluation engine. Efficient implementation is critical since every path expression must be filtered.

It is possible to model the other parts of MetaXQuery directly in XQuery (using a pre-processor, and a post-processor for constructing metadata), but for brevity the details are omitted.

3.3.8 MetaXQuery Completeness

We made the above extensions to XQuery and call them collectively MetaXQuery. The extensions are captured by the sample queries in Figure 6 that are not well supported by XQuery if using our new data model. The criteria of choosing these queries is by no means to be comprehensive. For example, we can query both data and metadata in current MetaXQuery, but we can't order data with metadata conditions, or vice versa. It is both hard and not our primary goal to come up with a complete set of extensions that can capture every possible change that needs to be addressed when we add metadata support to the system. Our purpose is rather to show that if an extension were needed, what the best way was as well as how to do that. We chose a subset of the problems, which we thought are the most obvious, representative or meaningful and showed how to extend the existing standards to solve these problems. Readers can follow exactly the same approach to tackle other related problems as well. We will leave it to the future work to find out and prove what a complete set of extensions is for MetaXQuery to capture every possible scenario.

CHAPTER FOUR
METADATA TREE ALGEBRA

In relational databases, the relational algebra is a set of low-level operations that manipulate the relations, such as selection and projection. High-level queries (i.e., SQL) are usually converted into a series of such operations by a query compiler for evaluation by the DBMS. The algebra is also heavily used in the query optimization phase of the compiler to obtain a more efficient version of the query by reorganizing these operations.

In the XML world, an algebra is also essential for applying database-style optimizations to XML queries. There have been several XML algebras proposed in the literature, but none has yet become a standard. We chose one of them, TAX (Tree Algebra for XML) [32], as the cornerstone of our work to compile MetaXQuery expressions into a low-level algebra, and to optimize them.

In this Chapter we first give a brief review of TAX and then describe how to extend it into MetaTAX. We also show how to use MetaTAX to optimize MetaXQuery.

4.1 TAX

TAX provides low-level operations for the evaluation of XQuery queries. XQuery queries can be translated to TAX expressions for fast evaluation. Typical operators in TAX (selection, projection, etc.) take a collection of data trees as input, a *pattern tree* and an *adornment* as parameters, and produce a collection of data trees as output. A pattern tree is a simple, intuitive specification of how to locate nodes of interest. Each node in a pattern tree represents a variable that is bound to some nodes in the data model (e.g., a DOM node). Each edge represents a relationship between a pair of bound variables. A TAX pattern tree has two types of edges, parent-child (**pc**) and ancestor-descendant (**ad**). A **pc** edge is used for a parent or child axis in a path expression while an **ad** edge represents an ancestor or descendent axis. Additionally, a pattern tree has an adornment which is a Boolean formula of predicates. Figure 15 shows a simple pattern tree for the path expression "`/books/book`". Variables \$1 and \$2 are related by a parent-child edge meaning that \$1 must be a parent of \$2 in the data model. When both variables are bound to a node, the associated adornment can be evaluated. The adornment tests to ensure that the name attribute of \$1 is "`books`" and the name attribute of \$2 is "`book`".

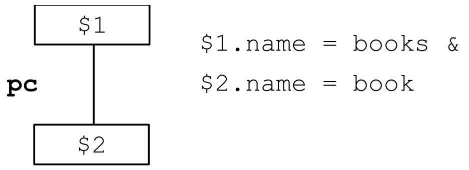


Figure 15 A pattern tree for /books/book

4.2 METATAX

MetaTAX is an extension of TAX that support metadata. MetaTAX introduces a **meta** edge to the pattern tree. The **meta** edge is inserted into the pattern tree whenever a meta axis is used in a MetaXQuery path expression. Figure 16 gives an example of a pattern tree for the path expression "`/books/book/meta::security`".

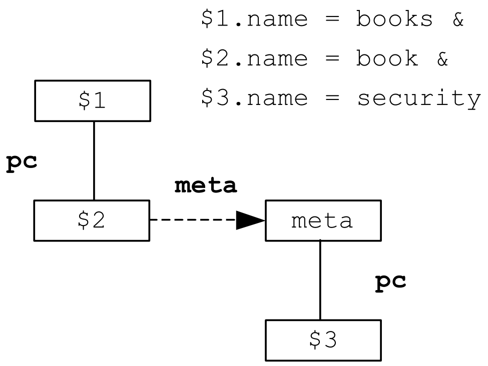


Figure 16 A MetaTAX pattern tree that explores the meta axis

In addition to the meta axis MetaXQuery has a **filterByPerspective** function that is invoked in most queries. The **filterByPerspective** function is a combination of the **getMetadataValues** and **filterByMetadataValues** function. In the rest of this section we show how to translate each of these functions into MetaTAX operators or plans.

The **getMetadataValues** function retrieves a specified type of metadata value for the input data nodes. In MetaTAX the function call translates to a simple pattern tree with a **meta** edge and a selection list (SL) on the metadata type node. For example, the function `getMetadataValues("security", /books/book)` would translate into a pattern tree shown in Figure 17. The pattern tree specifies which nodes are of interest in this query (`books`, `book`, `meta`, and `security`), and the Selection List (SL) acts as the adornment parameter. It lists the nodes (and their descendants) that are output from the evaluation of the pattern tree.

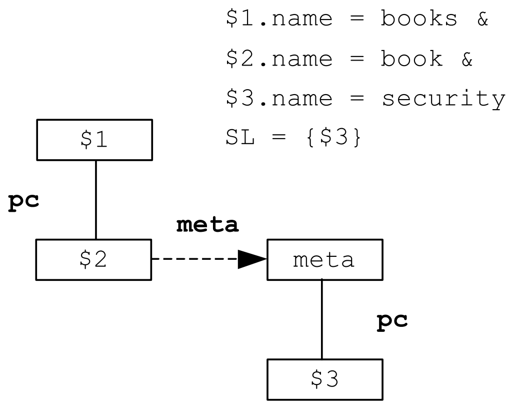


Figure 17 Sample pattern tree for the getMetadataValues function

The **filterByMetadataValues** operation selects data nodes that satisfy a given metadata condition. In MetaTAX, the operation is modeled as a *projection* of the data nodes with certain metadata conditions. Projection in TAX is different from projection in the relational algebra. In relational algebra, selection and projection are orthogonal operations: selection chooses rows and projection chooses columns. But in a tree data model, there are no such obvious orthogonal dimensions like rows and columns, so the role of projection is quite similar to selection. Projection selects only certain nodes, eliminating others. A sample pattern tree for the function `filterByMetadataValues("security", /books/book, "Joe")` is shown in Figure 18. The pattern tree specifies which part of the input data is of interest. The projection list (PL) tells the query processor which nodes to preserve in the output. In projection, no matter whether the nodes are in the projection list or not, all of their contents are preserved from the input. That's obviously contrary to the selection operator where only these in the selection list (SL) are preserved.

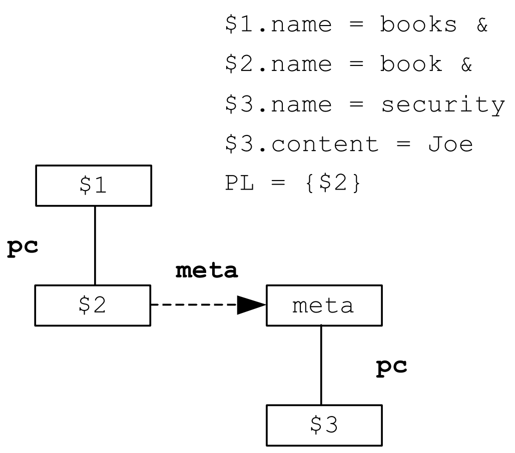


Figure 18 Sample pattern tree for the filterByPerspective function

The **filterByPerspective** function is a combination of the **getMetadataValues** and **filterByMetadataValues** functions. Theoretically it iterates through each metadata type element in the perspective fragment, getting the value of that metadata type. It then calls the **filterByMetadataValues** function on each data node to determine whether the data node matches the perspective. Only those that match the perspective are kept in the node list.

Such a naïve implementation using MetaXQuery is shown in Figure 19. The method first iterates through each data node (line 2). For each of them, it goes through every metadata type element in the perspective fragment (line 3) and gets its value using the **getMetadataValues** function (line 4). Then it calls the **filterByMetadataValues** function to check if that data node satisfies the perspective value. If not (the **filterByMetadataValues** function returns an empty sequence), that data node is discarded from the result (line 5 and 6).

```
1 LET $books := /books/book
2 FOR $book IN $books
3   FOR $type IN doc("perspective.xml")/perspective/*
4   LET $value := meta-fn:getMetadataValues(name($type), $book)
5   IF (empty(meta-fn:filterByMetadataValues(name($type), $book, $value))) THEN
6       $books := $books except $book
7 RETURN $books
```

Figure 19 Implementation of filterByPerspective using getMetadataValues and filterByMetadataValues

```
Algorithm MetadataAssociationJoin(DataList, MetaList[ ])
/* DataList is the list of data nodes that satisfy the query condition */
/* Each item in the MetaList[ ] array is a list of metadata fragments that
satisfy one metadata perspective condition in the query context. */

shortest = find the index of the shortest list in the MetaList[ ] array
for (m = MetaList[shortest].firstNode; m!=NULL; m = m.nextNode)
      Btree.add(m);

/* Join metadata fragments first */
for ( i = 0; i < length(MetaList); i++ ) {
      if ( i == shortest ) continue;
      for (m = MetaList[i].firstNode; m!=NULL; m = m.nextNode) {
            if ( Btree.find(m) != NULL )
                  Btree1.add(m);
      }
      Btree = Btree1;
}

/* After joining the metadata, now join the result metadata fragments with
the data */
result = new sequence;
for (d = DataList.firstNode; d!=NULL; d = d.nextNode()) {
      if ( Btree.find(d.getMetadata) != NULL )
            result.add(d);
}
return result;
```

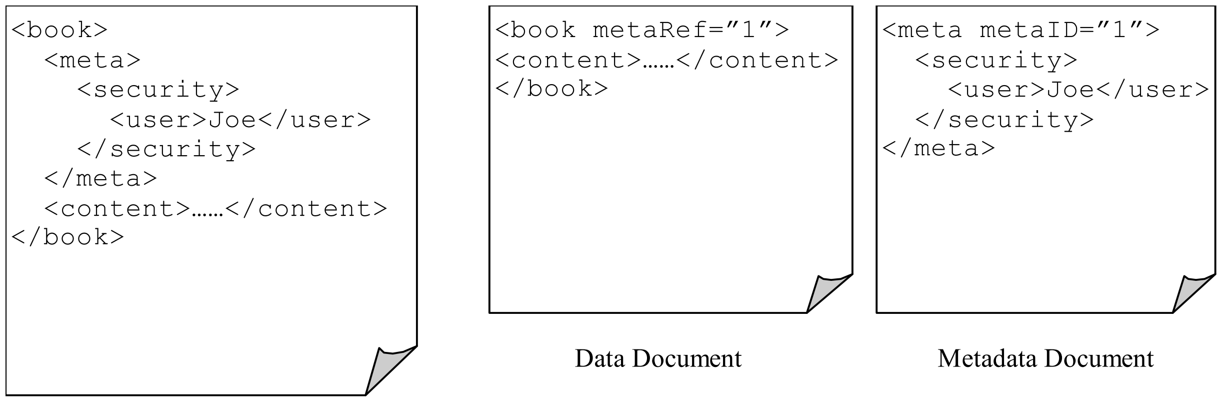
Figure 20 Metadata association join algorithm

The implementation in Figure 19 is obviously inefficient because for each data node it has to grab the perspective fragment and compare every metadata type to see if they match. For persistent data collections a better strategy is to use indexes on the metadata to quickly find which metadata matches a given perspective. For each kind of metadata, the index lookup will return a list of metadata trees that match the perspective. The lists for each kind of metadata are then joined together to produce a final list of metadata that matches the perspective, and that list is in turn joined with the data to produce a result. Assuming there are *N* kinds of metadata, this strategy will require *N* index lookups and *N* joins (*N*-1 joins of the different kinds of metadata and one join with the data). The join order can be rearranged to improve efficiency. Most of the time there will be far more data than metadata, so the join with the data should be delayed as long as possible. The metadata candidate lists should be joined first to find out which combinations of metadata match the perspective. The resulting combinations are then joined with the data. We call this join algorithm a *Metadata Association Join*. Pseudo code for this algorithm is shown in Figure 20. The input of the function is the list of data nodes and the array of metadata fragments. The output of the function is the remaining data nodes that match the perspective.

The essential extension we made to TAX is the **meta** edge in the pattern tree. We showed how to use it to capture the key functions we need in our framework (get the metadata values, and filter the data with metadata conditions). This may not be a complete set of functions that we need, so MetaTAX also may not be a complete extension either. As mentioned in Section 3.3.8, we still don't have a comprehensive set of MetaXQuery extensions yet and it's also not our purpose to do so. So it would be premature to claim that MetaTAX is a comprehensive extension of TAX. We have to have the complete set of MetaXQuery functions in order to give a complete set of MetaTAX extensions. But we have shown here the approach to convert these key functions into MetaTAX operators, and readers should be able to use the same approach for other possible functions as well.

4.3 METATAX IN PHYSICAL LEVEL

The **meta** edge in MetaTAX works at the logical data model level and needs to be converted to operators at the physical level to run on the database. There could be several different conversion strategies depending on the physical representation of data and metadata. One strategy is to store metadata by adding a special metadata child to each data element. The metadata child is the root of the metadata tree for that element. Implementing the meta axis is then straightforward: the **meta** edge would be converted to a **pc** edge in the pattern tree along with a constraint to ensure that the metadata child is chosen. The special metadata elements must be specially distinguished to avoid conflicts with pre-existing data elements, e.g., as `<&meta>` elements ("`&meta`" is not a legal element name in XML). While the strategy of using special children is straightforward and easy to understand it precludes the same piece of metadata from being shared by two or more data nodes.



```
<book>
  <meta>
    <security>
      <user>Joe</user>
    </security>
  </meta>
  <content>……</content>
</book>
```

```
<book metaRef="1">
<content>……</content>
</book>
```

```
<meta metaID="1">
  <security>
    <user>Joe</user>
  </security>
</meta>
```

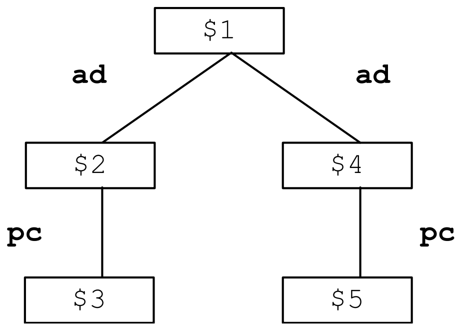
Data Document
Metadata Document

(a) Metadata embedded as the child of the element

(b) Metadata in a separate document and match by MetaID/MetaRef

Figure 21 Different strategies of physical representation

An alternative strategy is to store the metadata in a separate document and use ID and IDREF attributes to join data to metadata. The advantage of this strategy is that the same piece of metadata can be shared by multiple data elements. Another benefit is that data and metadata are naturally separated into different scopes. The disadvantage is that the join process could potentially be very expensive. If this strategy is used, it requires a very different implementation from the first one. An example is depicted in Figure 22. To evaluate the **filterByMetadataValues** function using this strategy, we first take the *product* of the book nodes and the metadata fragments (suppose the book nodes on the left) and then apply a left semi-join using the pattern tree above. The result is the list of the book nodes that satisfies the metadata condition specified in the pattern tree.



```
$2.name = books &
$3.name = book &
$4.name = meta &
$3.metaRef = $4.metaID
$5.name = security &
$5.content = Joe
```

Figure 22 Sample pattern tree for the execution of meta axis

CHAPTER FIVE
IN-MEMORY AND PERSISTENT IMPLEMENTATIONS

This section describes our prototype implementation of MetaDOM and MetaXQuery. We first implemented an in-memory version by extending the Apache Xerces2 Java Parser platform [60]. Since Xerces is only an XML parser without an XQuery processor, we applied the certify, sanitize, and group functions on the in-memory DOM model directly. We then shifted into a full-fledged, persistent XML data store with XQuery support, namely eXist. Since it's both a persistent native XML database and has a built-in XQuery processor, it's much better to build our system on top of eXist. So in this chapter, we first briefly describe our in-memory implementation using Xerces and then concentrate on the details of the persistent version.

5.1 IN-MEMORY MODEL SYSTEM ARCHITECTURE

The architecture of the in-memory prototype is shown in Figure 23. Creating a MetaDOM begins with parsing the data and metadata document. The metadata is assumed to be formatted in XML and associated with a data element by adding a "`metaRef`" attribute in the data element that points to the metadata with the specified "`metaId`". The data and metadata documents are separately parsed with a traditional DOM parser. Afterwards, the two DOMs are combined to create a MetaDOM. MetaDOM is implemented by extending DOM's Node class with a meta property and meta accessor methods. When the MetaDOM has been set up, calling the meta accessor of a node would directly return its metadata fragment. If there is no metadata, this extension is completely transparent, so this architecture is upwards-compatible with DOM.

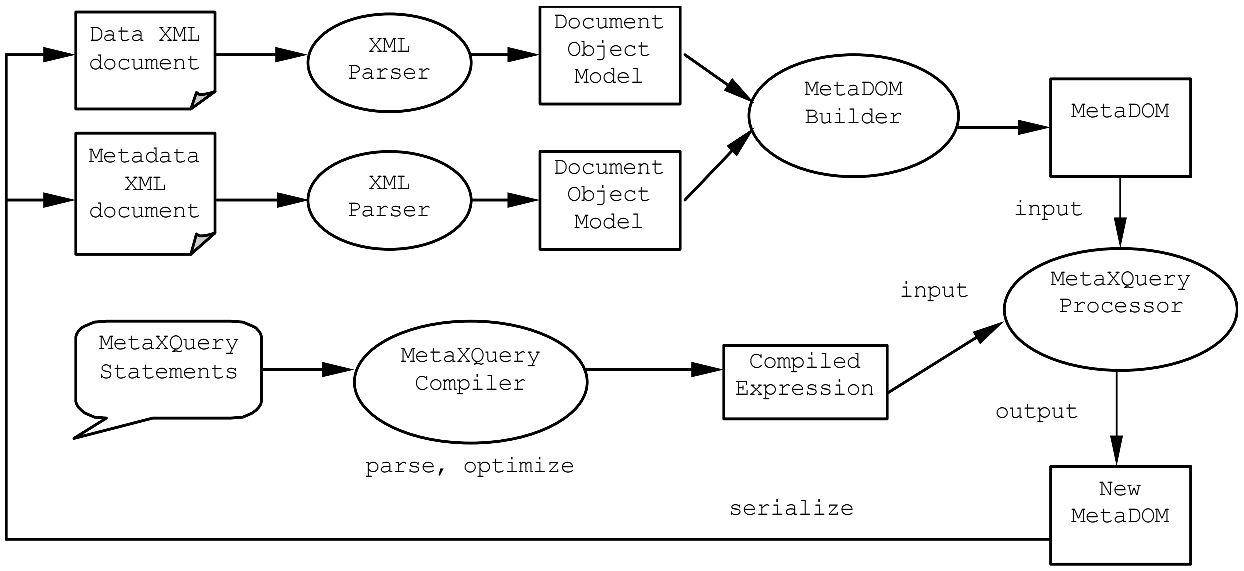


Figure 23 MetaDOM and MetaXQuery in-memory implementation architecture

After the MetaDOM instance is created, MetaXQuery queries or other operations can be evaluated on the instance. The result of a query is a new MetaDOM, which can be serialized into XML documents if desired.

5.2 PERSISTENT MODEL SYSTEM ARCHITECTURE

The architecture of the persistent prototype is shown in Figure 24.

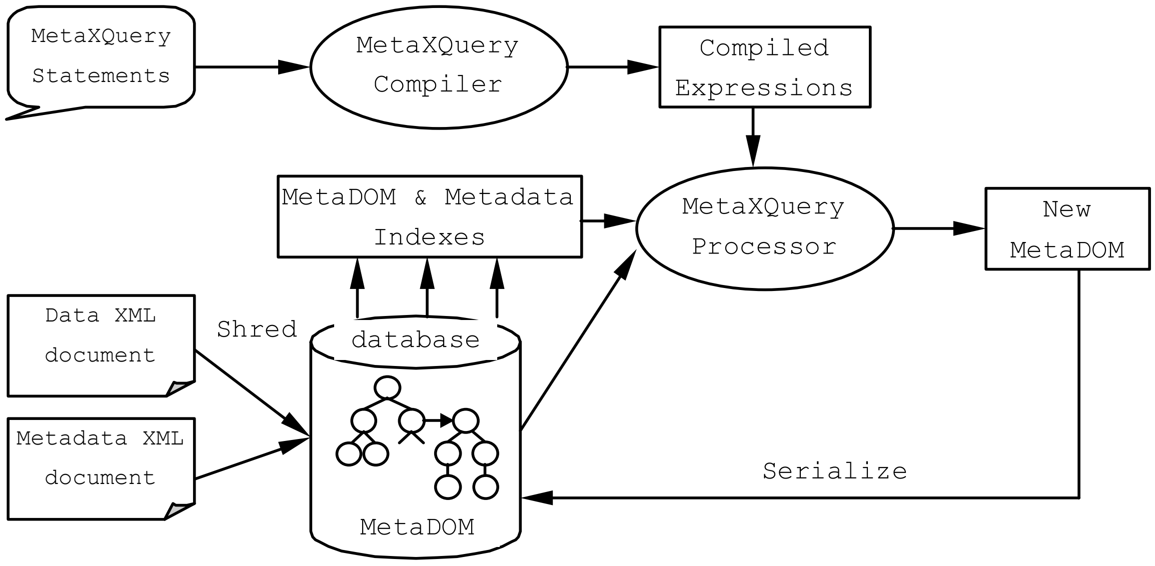


Figure 24 MetaDOM and MetaXQuery persistent implementation architecture

The key difference from the in-memory model is that the data and metadata documents are first shredded into the database and additional data structures like the indexes are built at the same time to facilitate queries. The queries are executed against the data model and the index data structures at the same time instead of the data model alone in the in-memory version. From now on, we will mostly talk about the persistent version of our implementation.

5.3 IMPLEMENTATION CHALLENGES

The overarching goal of implementing MetaXQuery is reusing existing standards and technology. For instance, MetaXQuery is upwards-compatible with XQuery, as is MetaDOM with DOM. Ideally, few changes will have to be made to a native XML DBMS (XDBMS) to implement MetaXQuery. But there are two key implementation challenges. First, MetaXQuery introduces *data scopes* into the data model. In the data model, metadata must be (logically) separate from the data so that wildcard queries (e.g., a descendent axis) explore only within the intended scope. Only the **meta** axis can bridge scopes. Unfortunately, XDBMSs do not support separate scopes for data. The second important challenge is supporting the **filterByPerspective** function. The function applies additional constraints to nodes identified by *every* path expression in a query. There is one check that must be performed for each kind of metadata. Additional levels of metadata add even more constraints. So efficient implementation of **filterByPerspective** is critically important to building support for (proscriptive) metadata into an XDBMS.

5.4 EXECUTION PLANS

The algebra outlined in Section 4.2 to support metadata is relatively straightforward to implement in a native XML DBMS (XDBMS). XDBMSs store XML documents in a back-end database or flat file. Usually, one or more indexes are created to improve query evaluation efficiency. Storing metadata is straightforward since the metadata is also an XML document. Each chunk of metadata is identified by a "`metaID`" attribute in the metadata document. An element in the data document subsequently references a chunk of metadata with a "`metaRef`" attribute. Typically, many elements will share the same metadata or have no metadata, so the metadata will be much smaller in size than the data.

Many XDBMS query evaluation engines use a path index to efficiently evaluate a query. A path index locates nodes for a given path expression, which saves on the (prohibitive) cost of traversing the data model to find the nodes. The result of the path index lookup is then combined with other index lookups (e.g., a text or word index) to process additional search conditions in a query, but in this section we will focus on a path index to illustrate how we plan to incorporate metadata in query evaluation.

Let's use the data model in Figure 25 as an example. In the data part of Figure 25, the path index would map "`/books/book`" to the list of nodes with ID 2 and 4, as shown in Table 3.

Data Path	**Node List**
`/books/book`	(2, 4)

Table 3 Original XML path index

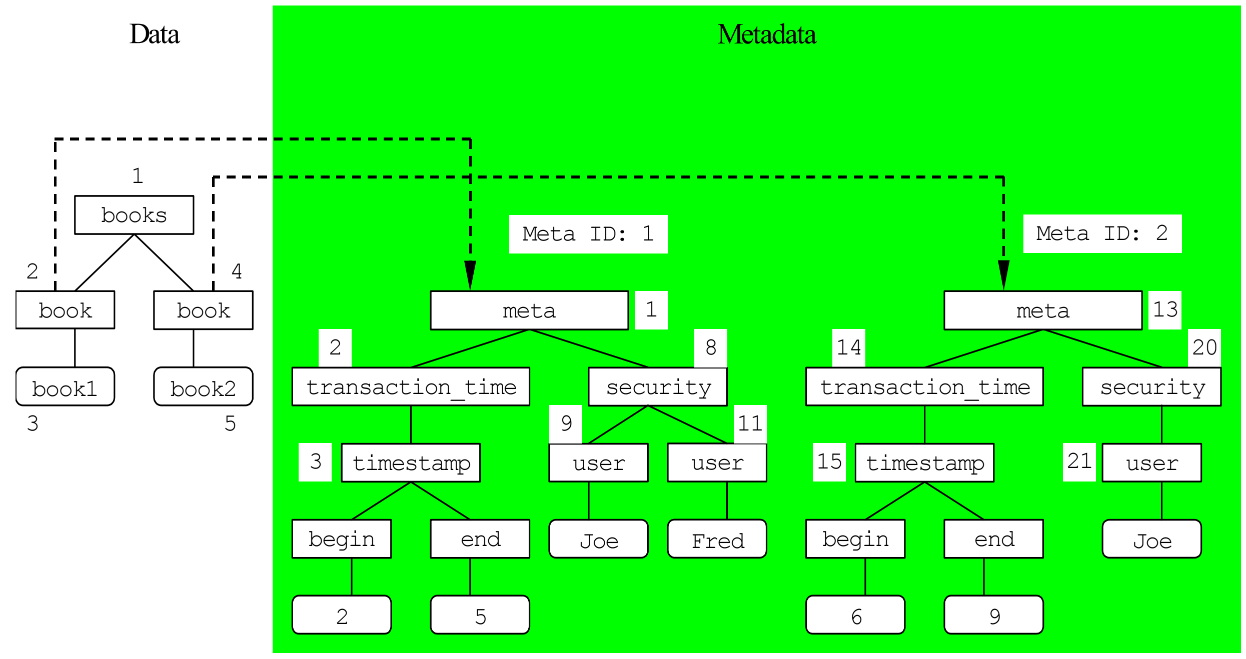


Figure 25 An example MetaDOM for indexing

We extend the path index with an additional column to record the MetaRef value for a list of nodes, as shown in Table 4. Figure 25 shows that the metadata chunk with a Meta ID of 1 is associated with the data node with ID 2.

Data Path	Node List	Meta Ref
`/books/book`	(2)	1
`/books/book`	(4)	2

Table 4 New XML path index

A perspective includes constraints on the metadata. For instance, a user might query from a transaction time perspective of 3 (i.e., rollback the database to time 3) and a security of Joe. Additional indexes are constructed to efficiently search for chunks of metadata that satisfy a specific constraint. Table 5 shows an index for transaction time metadata, while Table 6 illustrates one for security metadata. The Node ID column identifies the source of the metadata in the metadata document. Since meta-metadata could be present, each row in the index includes a Meta-meta Ref column. The data model in Figure 25 has no meta-metadata so that column contains NULLs.

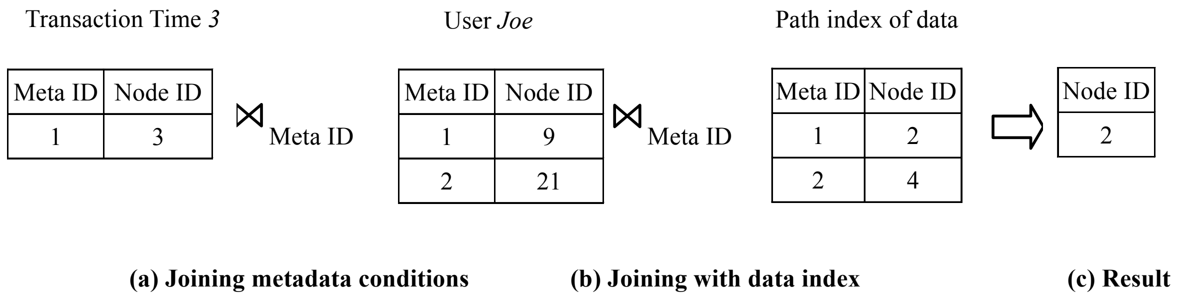
Time	Meta ID	Node ID	Meta-meta Ref
`[2,5]`	1	3	NULL
`[6,9]`	2	15	NULL

Table 5 Level 1 index on transaction time

Now let's demonstrate the use of the indexes with an example. Suppose we have the following query: "*Find book data that is available to the user Joe and exists in the database at time 3*." The steps in the query execution plan for this query are shown in Figure 26. The transaction time index is used to find intervals that include time *3*. Similarly the security index is used to find metadata chunks for the user *Joe* (Figure 26(a)). The path index is then used to locate nodes that match the path expression "`//book`" (Figure 26(b)). The results of the index lookups are joined on the Meta ID column (the join order is determined during query optimization) generating a result (Figure 26(c)).

Security	Meta ID	Node ID	Meta-meta Ref
Joe	1	9	NULL
Fred	1	11	NULL
Joe	2	21	NULL

Table 6 Level 1 index on security



Meta ID	Node ID
1	3

Meta ID	Node ID
1	9
2	21

Meta ID	Node ID
1	2
2	4

Node ID
2

Figure 26 Use of indexes to solve the example query

5.5 IMPLEMENTATION IN EXIST

eXist [44] is an open-source, native XML DBMS. We chose to modify eXist both because the source is available, and also eXist outperformed other systems in our benchmark system [36].

5.5.1 Storage

eXist stores an XML document by adding it to a *collection*, which can hold a set of XML documents. A document is serialized and stored in a paged, data file when it is added to the collection. We use a "`MetaDocRef`" processing instruction in the data document to identify its metadata. When the data document is parsed, the metadata document is also parsed and added to a separate metadata collection. Information about the data and metadata is also placed into the following indexes [44].

- `collections.dbx` – This index maps a collection name (several XML documents can be stored in a single collection) to the Collection object.
- `dom.dbx` – This index maps the unique node identifiers to a page location in the serialized, stored set of documents in the collection where the raw data about the node is

located.

- `elements.dbx` – An index that maps a element and attribute names to a list of unique node identifiers that correspond to all of the elements or attributes of that name.
- `words.dbx` – Maps a word or phrase to the unique node identifiers of the elements or attributes that contain the string in a value or as part of the text content.
- Several metadata indexes – As discussed in the previous section there is an index for each kind of metadata, e.g., a security index. The index is built when a metadata document is parsed and stored. We used eXist's B^+-tree index classes for each metadata index rather than using a specialized index, e.g., a temporal index.

5.5.2 Meta Axis

The implementation of meta axis is straightforward. We had to modify the XQuery parser to recognize the `meta::` axis. The XQuery parser in eXist is created by ANTLR (ANother Tool for Language Recognition), so it is easy to modify. The parsing rules in eXist closely follow the EBNF defined in the W3C XQuery standard. We extended the `axisStep` rule to accept a `meta::` axis step. We also added a class to evaluate the step, and added to the abstract syntax tree so that the new class would be invoked when the meta axis was used.

5.5.3 Perspective

In order to help the users access and query the metadata, we extended eXist with the **filterByPerspective** function. Essentially the function implements the query evaluation plan discussed in Section 4.2.

CHAPTER SIX
BENCHMARKING XML QUERY PROCESSORS

Throughout the history of data management, benchmarks have played a critical role in helping to assess new techniques, compare existing systems, and gain a better understanding of query evaluation efficiency and tradeoffs. Benchmarking XML-based systems will help to assess and compare the abilities of XML query processing tools.

A benchmark is a quantitative comparison of system performance. The system being measured could be a hardware component, a software package, or a combination of both. The performance can be measured in a variety of metrics, from CPU cycles to memory usage. Various benchmarks have been proposed to compare different systems, and components within systems. Benchmarks help users to evaluate systems, allow customers to choose the most desirable product, and enable vendors to claim advantages for their systems. The range of benchmarks includes hardware benchmarks (e.g., SPEC CPU [69] and WinBench [73]) operating system benchmarks (e.g., HBench-OS [66]), programming language benchmarks, web server benchmarks, and database management system benchmarks (e.g., TPC [71]).

Benchmarks can be classified as either *generic* or *application-specific*. A generic benchmark measures general system performance, independent of an application. An application-specific benchmark on the other hand is tailored to synthesize a workload for a particular application domain. Generic benchmarks are useful because of the prohibitive cost of implementing and measuring a specific application on many different systems. A limitation of generic benchmarks though, is that no single metric can measure the performance of systems on all applications. Depending on the application domain, the performance of a system could vary enormously and systems designed for a specialized domain may have weaker performance in other domains.

At this early stage of XML development, there's still no commonly agreed standard of XML application scenarios. So we have built a generic benchmark platform. The benchmark focuses on measuring the cost of query processing. XPath and XQueries are evaluated against a tree-like data model. Queries typically traverse part of the tree-like data model. The efficiency of the tree-traversal has a major impact on the cost of query processing. The tree can vary in depth, density, size, and the kind of information in each node. We designed an XML document generator which generates XML documents that conform to several factors which control the shape and size of the tree. By varying only one of the control factors (e.g., tree depth) and keeping the other factors constant the benchmark is able to isolate the impact of that factor on query performance. The benchmark also includes a suite of query templates that can be instantiated to produce a set of benchmark queries. Overall, the benchmark is designed to assess the impact of trees of different sizes and shapes on query performance. This will help query engine developers understand and evaluate implementation alternatives, and also help users to decide which query engine best fits their needs.

6.1 HISTORY OF BENCHMARKS

In the area of database benchmarks, the TPC family of benchmarks is the most well-known and widely used. The Transaction Processing Performance Council (TPC) created a series of benchmarks beginning in 1988. The earliest were TPC-A and TPC-B. The TPC-A benchmark primarily focuses on the online transaction processing (OLTP) environments emphasizing update-intensive database services [71]. The metrics used in TPC-A are throughput measured as *transactions per second* (*tps*), and the associated *price-per-tps*. The TPC-B benchmark uses the same metrics as TCP-A, but for a different environment, one that does not require online processing.

The Internet, and in particular the web, is an integral part of many businesses. Consequently, benchmarks of Internet and web servers are becoming more important and widely discussed. Several benchmarks have been proposed to measure the performance of servers, such as SPECWeb99 [70], httpperf [47], WebBench [72], InetMonitor [67], and WebStone [68].

There have been several benchmarks proposed for evaluating the performance of XML data management systems. XMark [53] is a benchmark for XML data stores. The benchmark consists of an Internet auction application scenario and twenty XQuery challenges designed to cover the essentials of XML query processing. XOO7 [49] is an XML version of the OO7 benchmark [10], which is a benchmark for OODBMSs. The OO7 schema and instances are mapped into a Document Type Definition (DTD) and the corresponding XML data sets. Then the eight OO7 queries are translated into three respective languages for query processing engines: Lore, Kweelt, and an ORDBMS. XMach-1 [6] developed at the University of Leipzig, Germany, is based on a web application that consists of text documents, schema-less data, and structured data. XMach-1 differs from other benchmarks in this area insofar as it is a multi-user benchmark, and it is based on a web-oriented usage scenario of XML data, not just the data store. The performance metrics that XMach-1 evaluates are throughput and cost effectiveness, so it's closer to the metrics that TPC-A and TPC-B use. The Michigan benchmark [52] is a *micro-benchmark* for XML data management systems. A micro-benchmark targets the performance of basic query operations such as selections, joins, and aggregations. The data set of the Michigan benchmark is generated by a synthetic XML generator, rather than from a particular application scenario.

The primary difference between our benchmark and all the above benchmarks is that we are most interested in how the basic properties of an XML document and XPath query affect the performance of XPath query execution. So we not only evaluate the performance of XPath queries in persistent XML data management systems, but also of in-memory XPath query processors. This difference also makes our definition of "basic query operations" different because all the previous benchmarks start from a database perspective. While they refer to the basic operations in database queries such as selection, projection, join, and aggregation, we focus on the basic properties of an XML document and the path traversal model of XPath.

In most benchmarks, each test case consists of a set of fixed parameters that are chosen to test the performance of the system under a particular (usually typical) configuration. We took a different approach to design the test cases. In each of our test case, we allow one parameter to vary. The purpose is to enable us to study how this single factor affects the query performance. So our test cases are tendency tests rather than fixed value tests.

Another important difference is that our benchmark is not domain-specific. XMark, XOO7, and XMach-1 are domain-specific benchmarks. Our benchmark is similar to Michigan's micro-benchmark in the sense that we are studying the atomic properties that affect general cases of XPath query execution. We use a synthetic XML document generator rather than generating documents that conform to an application specific XML schema.

6.2 Benchmark Data Set - XML Document Generator

This section describes the XML document generator. The generator is used to create each of the data sets in this benchmark. Since the focus of this dissertation is on benchmarking the performance of XPath evaluation engines, we will describe the document generator in terms of the kinds of "trees" that are induced by the generated documents. An XML document is parsed into a tree-like data model prior to evaluation of an XPath query. Changing one or more factors that control the document generator has the effect of creating different kinds of trees. For example, when the number of nested levels in the XML document is increased, the effect that is achieved is an increase in the depth of the tree created when that document is parsed. From this point onwards in this dissertation, we will tend to use the terms "document" and "tree" interchangeably.

Some benchmarks focus on including as much natural language as possible when generating documents for testing, e.g., the Michigan benchmark [52] and XMach-1 [6]. Our approach is different. We generate documents with randomly generated text values and selected element names. The reason is that XML query evaluation is based on data syntax rather than semantics. The XML document generator has a number of *control factors* that manipulate the shape and content of the tree data model. For example our benchmark provides a control factor for the length of text values. The document generator randomly generates text values of the specified length. The choice of these control factors is critical to the benchmark because each benchmark test depends on the control factors as described further in Section 6.2.1. We have carefully chosen a set of factors that we think are important properties of an XML document and hence, may have a significant impact on the performance of XPath queries. These factors represent the most common and influential properties of an XML document in the context of XPath query evaluation. We also chose control factors that are *basic* and do not depend on other factors, or combinations of factors. With these control factors, we are able to precisely control the document generation and isolate the impact of an individual factor on query evaluation.

6.2.1 Control Factors

There are eleven control factors in all. They are listed in Table 7. The factors are divided into two groups. The first group, the *tree shape group*, controls the general shape of the XML tree model. The second group, the *tree data group*, consists of factors that are more relevant to the content of the tree, such as the number of attributes, and length of text values. Though the number of attributes also affects the shape of the tree, since attribute nodes are on a special axis, we decided to place that control factor in the *tree data group*. The factors are described in more detail below.

Group	Control factors
Tree shape	Number of root children, Depth, Bushiness, Density.
Tree data	Number of attributes, Magic level, Selectivity, Magic position, Magic length, Text length, Random name.

Table 7 XML document generator control factors

The generated document has a single root element named "`Root`". The number of elements in the level below the root is controlled by the *root children* factor. Specifying a large number of root children will force the tree to broaden quickly. Since this is the first level containing real data, we call it level one. Names of elements in level one begin with the letter '`A`'. Below level one, the number of children for a node is specified by the *bushiness* control. Note that the bushiness does not apply to the first level since the root children are a special case. In general, element names at level *n* begin with the *n*th letter in alphabetical order. The reason for using this naming scheme is that it allows us to construct XPath queries that descend to a desired level. For instance to descend to level five we might use the query '`//E`'. Each level also has some embedded *magic* elements. A magic element has a fixed suffix of '`Magic`' in order to distinguish it from the non-magic elements at that level. The *selectivity* factor controls the percentage of magic elements. By setting a low selectivity, only a small percentage of the elements in a level will be magic. A query that locates these elements, e.g., '`//Emagic`', will generate a small query result. Figure 27 depicts how some of the factors impact the shape and content of the generated tree. The number of root children factor controls the level below the root. The bushiness controls the number of children of other interior nodes. The depth is the total number of levels. In the figure, some nodes at the leaf level are magic.

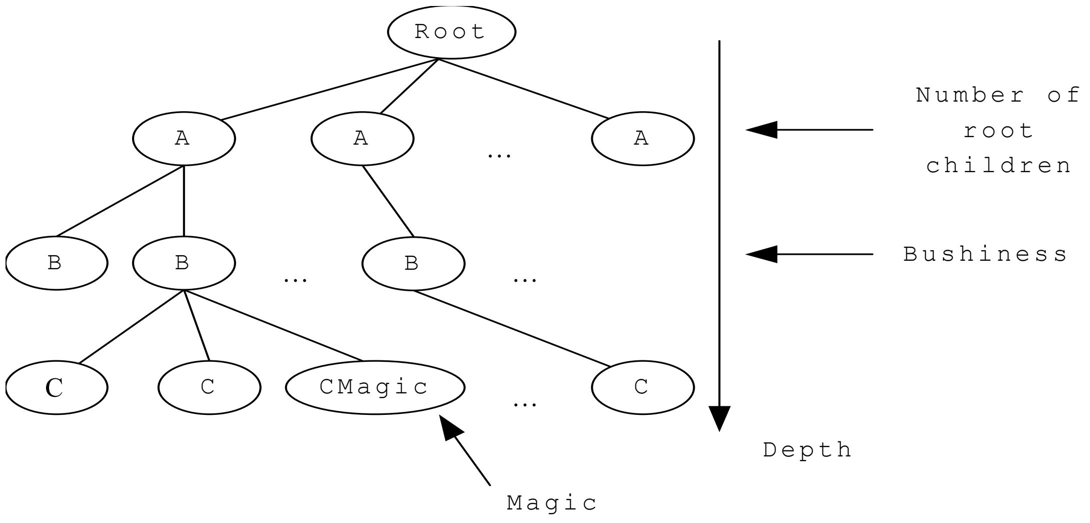


Figure 27 XML document generator model

The following list gives a detailed description of the eleven control factors in the XML document generator.

1) Number of root children — The number of root children controls the number of sub-trees under the root element node. A tree with only a few root children will have only a few nodes in level one, and by extension only a few nodes in the top few levels. By increasing the number of root children, the tree will acquire immediate breadth at the top.

2) Depth — The depth is the maximum number of levels in the tree, or the maximum depth of sub-element nesting in the generated document. The count starts from the level beneath the root, so the root is at level 0, the level with elements named 'A' is at level 1, etc.

3) Bushiness — The bushiness is the number of children of each element node, that is, the number of sub-elements in each element in the generated document. The parameter is used to control the width of the generated tree.

4) Number of attributes — Each element can have attributes, this parameter controls how many.

5) Density — Using the first three control factors (number of root children, depth, and bushiness), the total number of elements in the tree can be pre-determined for a complete tree. But we would like some trees to be sparse. The density is the percentage of nodes generated relative to the number of nodes in a complete tree. For instance a density of 50% will generate a tree only half as full as a complete tree of the same depth, bushiness, and root children.

6) Magic level — The magic level is the level in the tree at which magic nodes appear. Distinguishing between magic and normal nodes controls the size of a query result. For instance, setting the magic level to 5 will generate some 'EMagic' nodes (how many depends on the selectivity). By setting the level to 0 magic nodes are inserted at all levels.

7) Selectivity — The selectivity is the percentage of magic nodes at a level, specified by the magic level control factor. A selectivity of 20 means that 20% of the nodes (randomly selected) at the magic level will be magic nodes.

8) Magic Positions — XPath queries can locate not only the element nodes, but also attribute and text nodes. This control factor specifies where magic nodes are positioned. There are four options: 1) element name, 2) attribute name, 3) attribute value, and 4) the text.

9) Magic length — By default, element and attribute names are a single character in length. Magic nodes have the string "Magic" appended. The magic length control factor adds a suffix of the specified length to the magic nodes. Since the magic nodes are the ones queries often select, this parameter can help to evaluate the impact of matching long vs. short strings in a query.

10) Text length — This parameter controls the length of the text nodes in the document. The length can range from 0, which means no text nodes to any selected length.

11) Random name — By default, all of the non-magic elements in a level have the same name. The random name control factor increases the diversity of element names. When this control factor is selected, a random number is appended to the end of its regular name, so nodes will have different names (which impacts element indexes).

6.3 Benchmark Tests

There are many control factors in the XML document generator, each with a wide range of possible settings. Exhaustively testing all combinations is infeasible, so in this section we elaborate a small, but *representative* suite of test cases. Each test case in the suite is intended to gauge the impact of a control factor or factors on the performance of XPath queries. The test is based on a hypothesis about the impact of that control factor. The experimental methodology is to vary the single, control factor while keeping the others constant so that we can isolate the impact of this single factor and determine how it affects the performance of XPath queries in various implementations. Our tests are tendency tests unlike some other XML benchmarks [6] [49] [53]. The advantage of tendency tests is that it becomes possible to compare how well a system can handle a particular property of the XML document or the XPath query. For example, in an application scenario where the depth of the XML tree model is an important varying factor, not only can we tell the performance of a query engine in dealing with trees of a particular depth, but also which query engine scales well as the tree depth increases. In many cases, this information is much more important than just a single data point.

6.3.1 Overview of Test Cases

The benchmark has fifteen test cases divided into three groups. Each of these test cases isolates a single property of an XML document or XPath query and evaluates the impact of this property on the performance of XPath query processing. The first two groups deal with the how different XML documents impact the performance of XPath queries. Among them, the *tree shape group* focuses on the aspects that change the shape of a tree structure, such as tree depth and width, while the *tree data group* focuses on the aspects that relates to the content of an XML document, such as the magic level, selectivity, and text length. These two groups are similar to the groups of control factors for the benchmark's XML document generator. The third group, the *XPath property group*, focuses on the different location paths and functions that XPath provides. The name and description of each test is listed in Table 8, and the values of the control factors in the test case are listed in Table 9. In Table 8 the Test Case column lists a short, descriptive name for each test. The Description column provides a short description of the test. The remainder of this section describes each test case in more detail. In Table 9 the explicit settings for the control factors are given. The "*Vary*" value stands for the control factor that varies.

There is one test case for each control factor, except for the density. The reason is that navigating an incomplete tree is similar to navigating a fraction of a complete tree. By using magic nodes, we are able to control how large a fraction of the tree is traversed in a query so the density in all of the test cases is 100%. Although there are no tests currently in the benchmark that vary the density, we include density as a control factor for future extensions. In particular being able to set the density to a very low percentage is useful for creating very deep trees because deep, complete trees can exceed memory capacity.

6.3.2 Descriptions of the Test Cases

The following sections describe each test case in more detail.

6.3.2.1 Fat, flat tree (text value) test

The first test case evaluates the impact of the number of root children on the performance of XPath queries. This test constructs trees with a varying number of root children and queries the root children. This kind of XML document and query are fairly common in real-world applications, for instance in a phone book there will likely be many people listed, but each listing has a very shallow tree. A typical document may be a long list of elements similar to the one depicted in Figure 28.

```
<Person>
  <Name>Joe Smith</Name>
  <Address>P.O. Box 111</Address>
  <Phone>888-8888</Phone>
</Person>
```

Figure 28 Sample XML fragment

Test Case	Description
Tree Shape Group	
Fat, flat tree (text value)	Varies the number of root children to create trees with “broad shoulders”, queries access the text values in the leaves of the tree.
Fat, flat tree (attribute value)	Varies the number of root children to create trees with “broad shoulders”, queries access the attribute values in the leaf elements.
Tree depth	Varies the depth of the XML document tree.
Tree width	Varies the width of the tree.
Tree Data Group	
Magic level	Varies the level at which magic nodes are placed, with queries descending through the magic nodes.
Selectivity	Varies the percentage of magic nodes, controlling how many nodes are selected by a query.
Text length	Varies the length of text values in the document.
Random name	Runs the same XPath query on documents with random element names vs. fixed element names.
Number of attributes	Varies the number of attributes in each element, with queries selecting the attributes.
Magic length	Varies the length of the magic suffix, creating elements with long names.
Magic position	Varies the position of magic nodes (element name, attribute name, attribute value, and text/element value), queries are tailored to locate the magic.
XPath Property Group	
XPath query type	Tests XPath queries on different axes.
Short-circuit evaluation	Tests whether the query engines have the capability of short-circuit evaluation in predicates.
Steps vs. predicates	Trades predicates for steps (evaluating efficiency of steps vs. predicates).
String function	Tests efficiency of various XPath string functions.

Table 8 Benchmark tests and descriptions

Root	**Depth**	**Bush-iness**	**Attrs.**	**Magic Level**	**Select-ivity**	**Magic Position**	**Magic Len.**	**Text Len.**	**Random Name**	**XPath**
Tree Shape Group										
Fat, flat tree (text value)										
Vary	2	5	0	1	1%	T	3	10	No	Desc.
Fat, flat tree (attribute value) test										
Vary	1	1	5	1	1%	AV	10	0	No	Desc.
Tree depth test										
100	*Vary*	4	0	*Vary*	10%	E	0	0	No	Child
Tree width test										
100	4	*Vary*	0	4	10%	E	0	0	No	Child
Tree Data Group										
Magic level test										
100	7	4	0	*Vary*	10%	E	0	0	No	Desc.
Selectivity test										
100	7	4	0	7	*Vary*	E	0	0	No	Desc.
Text length test										
100	5	4	0	5	10%	T	1	*Vary*	No	Child
Random name test										
100	7	4	0	All	30%	E	0	0	*Vary*	Child
Number of attributes test										
100	5	4	0	5	10%	AN	0	0	No	Desc.
Magic length test										
100	5	4	0	5	10%	E	*Vary*	0	No	Child
Magic position test										
100	5	4	5	5	10%	*Vary*	0	20	No	Child
XPath Property Group										
XPath query type test										
100	5	4	0	4	30%	E	0	0	No	*Vary*
Short-circuit evaluation test										
100	5	4	0	5	30%	E	0	0	No	*Vary*
Steps vs. Predicates test										
100	5	4	0	All	30%	E	0	0	No	*Vary*
String function test										
100	5	4	0	5	30%	T	5	100	No	Pred.

Abbreviations:
Magic position: E (element name), AN (attribute name), AV (attribute value), T (text).
XPath query: Desc. (descendant axis), Child (child axis), Pred. (predicate).

Table 9 Benchmark tests parameters

Queries on these kinds of documents often search for a very small amount of data (one person typically) so the selectivity is very low. To model the anticipated query behaviour this benchmark test has a selectivity of just 1%. The depth and bushiness parameters are also very small. Since the query type is usually searching directly for a person's name, in XPath, we'll use the descendant axis. The number of root children is the varying factor. The test increases it from a small to a large value as depicted in Figure 29, the number of `A` elements varies. The number of root children are 1000, 5000, 10000, 50000, and 100000.

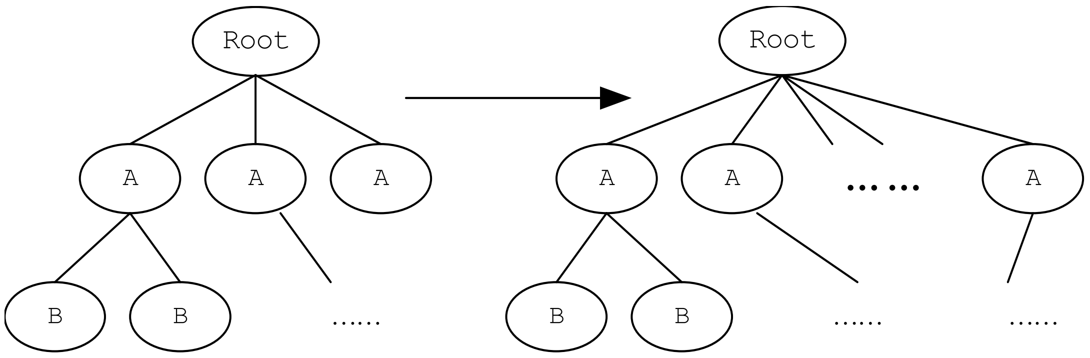


Figure 29 Fat, flat tree tests

6.3.2.2 Fat, flat tree (attribute value) test

Another common way to store long lists of elements is to use attributes, rather than subelements. An example is given below.

```
<Person Name="Joe Smith" Address="P.O. Box 1111" Phone="888-8888" />
```

The experiment settings are otherwise similar to the previous test. In particular, the varying factor is the number of root children.

6.3.2.3 Tree depth test

This test tests the ability of an XPath query engine to scale as tree depth increases. Some XML documents may have a very flat structure, like in the previous two tests, while some may be deeply nested, resulting in a very deep tree. This test increases the depth of the tree from 1 to a maximum of 7, as depicted in Figure 30. It keeps constant all of the other factors, except the magic level. The magic level is kept at the leaf level. The depth values included in this test case are 1, 3, 5, and 7.

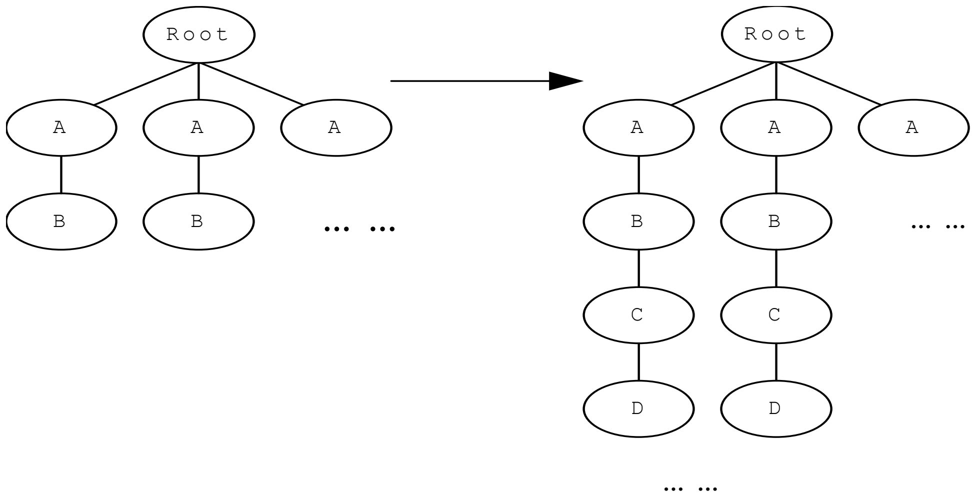


Figure 30 Tree depth test

6.3.2.4 Tree width test

This test tests the ability of an XPath query engine to scale as tree width increases. The bushiness varies while the other control factors are kept constant in this test. Figure 31 shows that a tree with two children is made bushier by increasing the number of children for each node while keeping the depth the same. Since the number of root children is also kept constant the total size of the tree increases exponentially as the bushiness increases linearly. The bushiness values included in this test case are 1, 5, 10, 15, and 20.

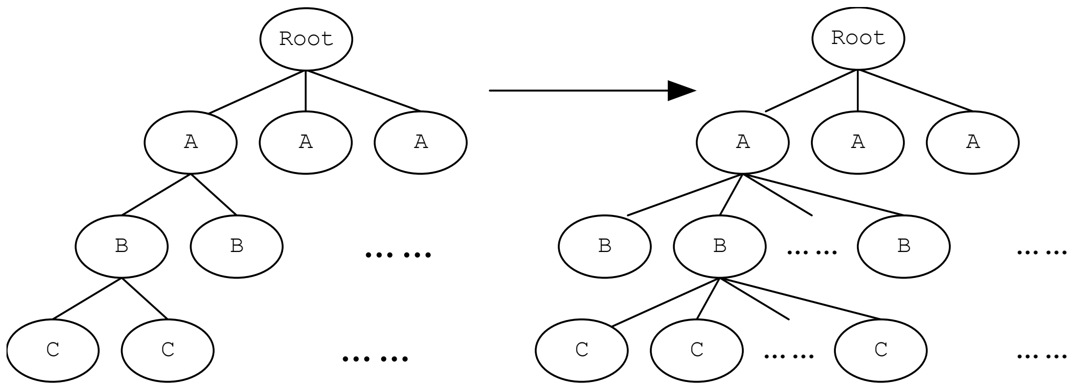


Figure 31 Tree width test

6.3.2.5 Magic level test

In this test, we increase the level at which magic nodes appear in a tree so that the queries have to descend further. The goal of the test is to determine how increasing the search space impacts query performance. Figure 32 sketches the general idea of placing the magic nodes at different levels. The magic nodes appear high in the tree initially and gradually are pushed to deeper levels in the tree. Queries in this test will descend through the magic nodes to the leaves. Only 10% of the nodes in the magic level are magic nodes. For example, when the magic level is 1, the first step of the query "`//Amagic//G`" searches all the nodes in the first level and the second step searches only 10% of the subtrees beneath the magic nodes (when the selectivity is 10%). When the magic level is 3, the first step of the query "`//Cmagic//G`" would have to search all the nodes in the top three levels and the second step still only searches 10% of the subtrees under level 3; and finally when the magic level is at the leaf, the query "`//Gmagic`" would have to search the entire tree. In general, the search space of queries will increase in size as the magic level is pushed deeper in the tree, while keeping the size of the tree constant and the size of the query result, thereby ensuring that the performance of the XPath queries can be compared fairly (they won't be skewed by the varying tree or result sizes). The depth of all the trees in this test case is 7, and we use 1, 3, 5, and 7 as the magic level parameters.

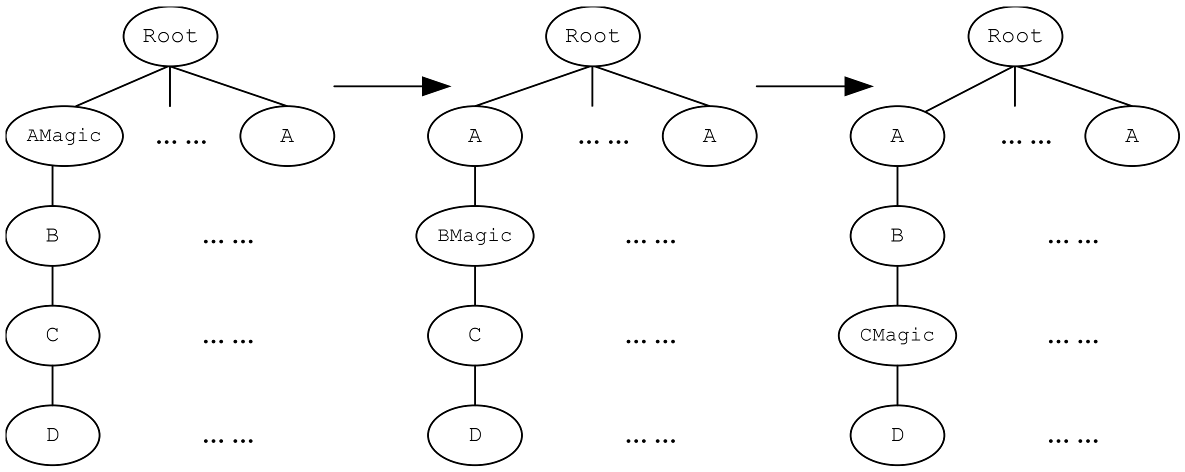


Figure 32 Magic level test

6.3.2.6 Selectivity test

The selectivity test increases the selectivity (the percentage of magic nodes). Queries in the test fetch the magic nodes, so as the selectivity increases, queries will have to fetch larger and larger result sets. The test generates a large tree with the magic level set to be the leaf level. The varying factor is the selectivity. The selectivity test measures how well a query evaluation engine scales as the result size gets larger (and all else remains the same). The values of selectivity we choose in this test case are 10%, 20%, 30%, and 40%.

6.3.2.7 Text length test

Whether an XML document has text, the length of the text fragments may be an important factor in query performance. In this test, we increase the length of the text values to gauge the effect that text length has on performance. Queries retrieve the text content of the document. The text length is increased in all levels, including the magic level. The text length values in this test case are 0, 10, 100, 500, and 1000.

6.3.2.8 Random name test

Some XPath query engines might build a path index or Dataguide [25]. One way to uncover this optimization technique is to randomize the element names within a level. Figure 33 shows the effect of turning on the random name control factor in the document generation. Dataguides will be more costly to construct with random element names, since there are far more "different" paths in the document. We set the magic level to all levels in the tree and use queries that traverse the "magic" paths as follows: "`/Amagic/Bmagic/Cmagic`".

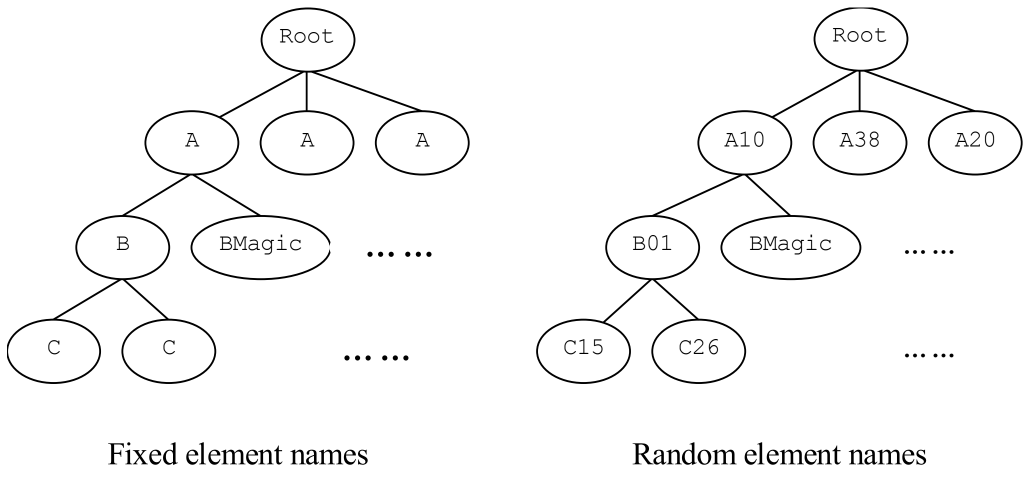


Figure 33 Random name test

6.3.2.9 XPath query type test

Different XPath queries will sometimes retrieve the same result. In this test we create documents that will return the same result on a select set of different queries. This enables us to determine differences in query construction. The different XPath queries are listed below.

- Using the child axis - `/A/B/C/Dmagic`
- Using the descendant axis - `//Dmagic`
- Using the ancestor axis - `//D/ancestor::A/B/C/Dmagic`, it's possible that the result set is smaller than the magic set, but generally not much
- Using predicates - `//Dmagic[./E]`
- Using the union operator - `//Dmagic | //Dmagic[./E]`
- Using the preceding axis - `//Dmagic[./preceding::D]`, it's also possible that the result set is smaller than the magic set

6.3.2.10 Short-circuit evaluation test

This test determines if an XPath package uses short-circuit evaluation of predicates. When a query has multiple predicates, there is an implicit logical AND between each predicate. In short-circuit evaluation, if a predicate fails then there is no need to evaluate the remaining predicates. Predicates are side-effect free, so short-circuit evaluation should be used by every package. The test consists of three "atomic" queries to determine the cost of one or two predicates, and two combined queries that might benefit from short-circuit evaluation. The three atomic queries are listed below (following page). Note that all of the queries are the "same" in the sense that they return exactly the same result set.

- `/A/B/C/D[./Emagic]`
- `/A/B/C/D[./following::*/B/C/D/Emagic]`
- `/A/B/C/D[./ancestor::A][../..//Emagic]`

The two combined queries are as follows.

- `/A/B/C/D[./Emagic][./following::*/B/C/D/Emagic]`
- `/A/B/C/D[./Emagic][./following::*/B/C/D/Emagic][./ancestor::A]`
  `[../..//Emagic]`

6.3.2.11 Steps vs. Predicates test

This test explores the cost of querying by steps *vs.* the cost of querying by predicates. All queries essentially explore the same portion of the tree, which consists of all the paths from the root to a leaf that use only magic nodes. In order to do this, the magic nodes should appear in all levels, and the XPath queries include the following.

- `/Root/Amagic/Bmagic/Cmagic/Dmagic/Emagic`
- `/Root/Amagic/Bmagic/Cmagic/Dmagic[./Emagic]`
- `/Root/Amagic/Bmagic/Cmagic[./Dmagic[./Emagic]]`
- `/Root/Amagic/Bmagic[./Cmagic[./Dmagic[./Emagic]]]`
- `/Root/Amagic[./Bmagic[./Cmagic[./Dmagic[./Emagic]]]]`

Although the size of each result set could be different, the portion of the tree explored by each query is similar.

6.3.2.12 Number of attributes test

This test examines the impact of adding more attributes in each magic element on the performance of the XPath query engines. The numbers of attributes in each magic element are 1, 3, 6, or 10. Each element will have the same number of attributes, and benchmark queries in this test will access all of the attributes among the magic elements.

6.3.2.13 Magic length test

A node test matches the name of the node against a string in the query. Several such tests are performed in many queries, so the length of the name may have an impact on the cost of queries. This test measures that impact by choosing the length of the name to be 0, 10, 50, 100, or 500. Magic nodes are added to every level and queries decscend through each level along the magic nodes.

6.3.2.14 Magic position test

This test compares how different query engines behave when the same information is placed into different positions in the document. This can yield insights into where data should be placed to improve performance. The possible positions of magic nodes include *element name*, *attribute name*, *attribute value*, and *text/element value*. Benchmark queries for this test will locate the magic information in each position.

6.3.2.15 String functions test

String processing and matching functions play an important role in many queries. In this test, the performance of queries with different string functions is measured to see how well each XPath query engine does string processing. The document remains the same for every case in this test, but the queries differ. The following queries test XPath's string functions.

- Using full string match: `//D[text()='Dmagic']`
- Using the "*starts-with*" function: `//D[starts-with(text(), 'Dmagic')]`
- Using the "*substring*" function: `//D[substring(text(), 2, 5)='magic']`
- Using the "*contains*" function: `//D[contains(text(), 'magic')]`
- Using the "*substring-after*" function:

`//D[substring-after(text(),'D')= 'magic']`

- Using the "*string-length*" function: `//D[string-length(text())=6]`

6.4 BENCHMARK RESULTS AND ANALYSIS

In this Section, we present a general overview and analysis of the benchmark tests we did and what they tell us about the factors that impact the performance of XML query processors.

6.4.1 Benchmark Packages

Many XPath query packages are available in the market and as research prototypes. We chose several popular query packages to evaluate against the benchmark. The packages fall into two broad categories: 1) the in-memory packages and 2) the XML database packages. In-memory packages can use either SAX or DOM parsing to construct a tree-like data model (a DOM) in memory. After parsing completes, XPath queries are evaluated on the constructed data model. The XML database packages parse and save an XML document in a persistent data store. XPath queries are then run on the persistent data store, i.e., as database queries. XML database packages are much more likely to create secondary data structures like indexes or dataguides to improve query performance. The in-memory packages can be further divided by language into packages that use Java vs. C++. Based on the above categorizations, we set out to find the products that are commonly used and representative of the technology for each category. The XPath packages that we chose are summarized in the following list.

- **Java packages**
  - **Saxon** – Saxon [41] is an open-source XSLT and XQuery processor written by Michael H. Kay. The XML document parser of Saxon is a slightly improved version of the Ælfred from Microstar, so it's based on the SAX API.
  - **Xalan-Java** – Xalan-Java [58] is an open-source XSLT processor for transforming XML documents into HTML, text, or other XML document types developed by the Apache XML project. It fully implements XSLT version 1.0 and XPath version 1.0.
  - **DOM4j** – DOM4j [62] is another open-source framework for processing XML. It is integrated with XPath and fully supports DOM, SAX, JAXP and the Java platform such as Java 2 Collections. DOM4j works with any existing SAX parser via JAXP, and/or DOM implementation. So in our benchmark, we'll use both the default SAX parser of DOM4j and a DOM parser, just like what we do for Jaxen.
  - **Jaxen** – Jaxen [63] is an open-source Java XPath Engine from the Werken Company. It's a universal object model walker, capable of evaluating XPath expressions across

multiple models. Jaxen is based on SAXPath, which is an event-based model for parsing XPath expressions. Currently, it has implemented the XPath engine for DOM4j and JDOM, two popular and convenient object models for representing XML documents. Of course, W3C DOM is also supported.

- **C++ packages**
  - **MSXML** – Microsoft® XML Core Services (MSXML) [65] is a collection of tools that helps customers to build high-performance XML-based applications. It fully supports XPath version 1.0 in its XSLT processor.
  - **Xalan-C++** – Xalan-C++ [59] is just the C++ version of Xalan-Java.
- **XML database packages**
  - **eXist** – eXist [44] is an open-source XML native database featuring efficient, index-based XPath query processing. The database is lightweight, completely written in Java, and can be run as either a stand-alone server process, inside a servlet, or directly embedded into an application. Its Java API completely comforms to the XML:DB API, which provides a common interface to access XML database services.
  - **Xindice** – Xindice [61] is another open-source XML product developed by the Apache XML project. It is also a native XML database using XPath as its query language. Xindice also implements the XML:DB API.
  - **COR** – COR is the name of a leading commercial database with extensions to support XML by shredding a document into a back-end object-relational database. Due to a licensing agreement for the commercial package, we cannot disclose the actual names of the package, so we will just refer to it as **COR**.

6.4.2 Results and Analysis

This section provides an analysis of trends that are present across the many individual tests in the benchmark. Where possible, we also draw infererences about the behavior of the various packages.

Saxon is generally the fastest Java package that we tested. Saxon uses an innovative tree structure [42], a subset of the DOM data model to represent an XML document, which contributes to its good performance. The nodes in this tree structure are represented as integer arrays rather than as objects. Saxon does not provide a complete DOM interface. For instance, DOM update is not supported. Though we focused exclusively on query performance, and Saxon supports all of the benchmark queries, the lack of full DOM support indirectly enhances Saxon's performance because it reduces memory consumption. So for read only applications in Java, Saxon is a very good choice.

In the C++ group, MSXML performs much better in all cases than Xalan-C++, and also much better than all the Java packages. MSXML however is only supported on a Windows platform (currently). But for Windows-based applications that need fast performance, MSXML is the best choice among the packages we tested.

In the XML database group, eXist is a good choice because of its excellent performance. However, eXist does not even support all of the queries in XPath. Some simplifications were made in the design of eXist leading to improved performance at the cost of full functionality, just like Saxon. eXist fails to handle some uncommon axes and string functions. Xindice, on the other hand, provides very stable and uniform performance in all test cases, although it's slower than eXist.

Increasing the depth or the nesting level of the XML document comes at a very high cost, not surprisingly. To reduce depth, use attributes rather than subelements with text values. Alternatively, if possible, expand the tree horizontally rather than vertically. Both alternatives can generally yield better performance. Of course, whether these alternatives are possible largely depends on the schema of the XML document and the application scenario. Another factor to keep in mind is that querying an attribute value is slightly more expensive than querying a text value (assuming both are at the same depth in the tree). So if attributes can shrink a tree, they improve performance, but otherwise, use text nodes.

If possible, use the child or descendant axis in your location paths, rather than one of the other axes. Try to avoid the preceding and following axes, and to a lessor extent the ancestor axis, because evaluating one of these axes can be a “gotcha”, resulting in a much greater cost than a more common axis.

The structure of the XML document always plays a more important role than the data itself in query performance. In other words, if you want to manipulate the XML document to get better query performance, try to reduce the depth, bushiness or other shape properties of the document. Shortening element names, for instance, generally won’t help to improve query performance.

CHAPTER SEVEN
PERFORMANCE TESTS

We designed a series of experiments to evaluate the performance of our prototype. The overarching goal of the experiments is to ascertain the scalability of modeling metadata in MetaDOM and querying it with MetaXQuery. Support for metadata will add some overhead since it is new functionality; but our chief concern is that such support might significantly degrade performance, especially as the size and amount of metadata increases. So we developed several experiments, each of which increases a typical aspect of the problem space, to elicit insights on the scalability of our approach.

We performed each experiment on a single-processor 1.7 GHz Pentium IV machine with 1GB of main memory. The machine was running Windows XP and Sun JDK version 1.4.1.

7.1 IN-MEMORY PERFORMANCE TESTS

First, we implemented MetaDOM and the MetaXQuery certifying, sanitizing and grouping functions by extending Apache's Xerces2 Java Parser platform [60]. We let it parse the XML document and build the MetaDOM data model in memory. We then apply these functions to the data model nodes.

We designed four experiments to evaluate the in-memory processing model. The first experiment tests the metadata functions on documents of increasing size by increasing the level of nesting in the document (i.e., increasing the depth of the MetaDOM). Each element in the document is annotated with five kinds of metadata. The experiment increases the amount of data, but the amount of metadata per data element is kept constant. The second experiment fixes the size of the data, but increases the number of metadata properties per data element, from no metadata to five different kinds of metadata. The third experiment keeps the size of the data constant, but increases the amount of metadata per element by varying the number of values for a single metadata property. The fourth experiment also keeps the size of the data document constant, but increases the level of metadata (meta-metadata, …). If all four experiments exhibit linear cost as the problem size increases, then our approach, while adding some overhead, should scale.

For all of the experiments we used our own XML document generator described in Section 6.2 to generate the data documents. The configuration of relevant parameters for each of the experiments is shown in Table 3 (the bushiness or number of children is set to 4). For each test case, we decided to evaluate the *worst-case* scenario. In the worst-case every element in the data document points to an identical piece of metadata. It is the worst case because every metadata function has to process all of the metadata. For instance, meta-match would have to test every piece of metadata. We used up to five kinds of metadata: transaction time, valid time, security, language, and reliability. The number of metadata values specifies how many values there are in each of the properties.

Test Cases, What Varies	**DOM Depth**	**Metadata Properties**	**Metadata Values**	**Metadata Levels**
1) data document size	2 to 7	5	1	1
2) # of metadata properties	4	0 to 5	1	1
3) # of metadata values	4	1	5 to 30	1
4) # of metadata levels	4	1	1	1 to 5

Table 10 In-memory performance tests parameters

7.1.1 Varying data document size

In this experiment, we increased the amount of data to determine how the performance changes. The results are depicted in Figure 34. The x-axis plots the depth of the generated data document. Since the number of nodes in the document increases exponentially as the depth increases, the y-axis plots the log, base 4, of the time taken to perform the experiment. The graph shows linear, or sub-linear, growth in the time as the document size increases, essentially, adding a constant factor to the cost of traversing the DOM. Merge costs slightly more since it is constructs nodes during the merge.

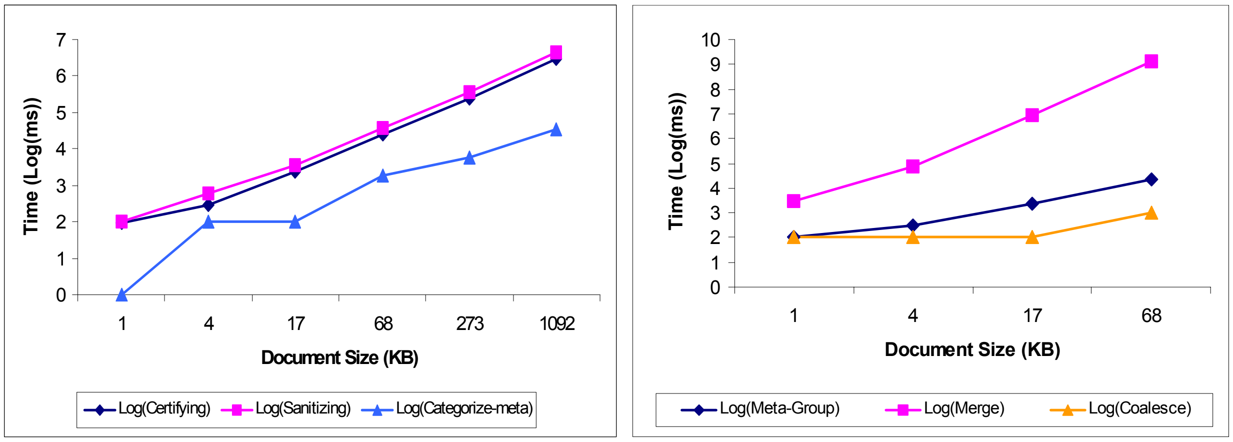


Figure 34 Results of in-memory experiment 1, varying the data document size

7.1.2 Varying the number of metadata properties

In this experiment, the size of the data is kept constant while the number of metadata properties increases from no metadata (0 properties) to 5 kinds of metadata. Figure 35 shows the results. The performance is linear in the number of metadata properties. There are small fluctuations in performance because different metadata properties have different representations and semantics, so the time to process each kind varies. Merge is once again more expensive because it constructs nodes.

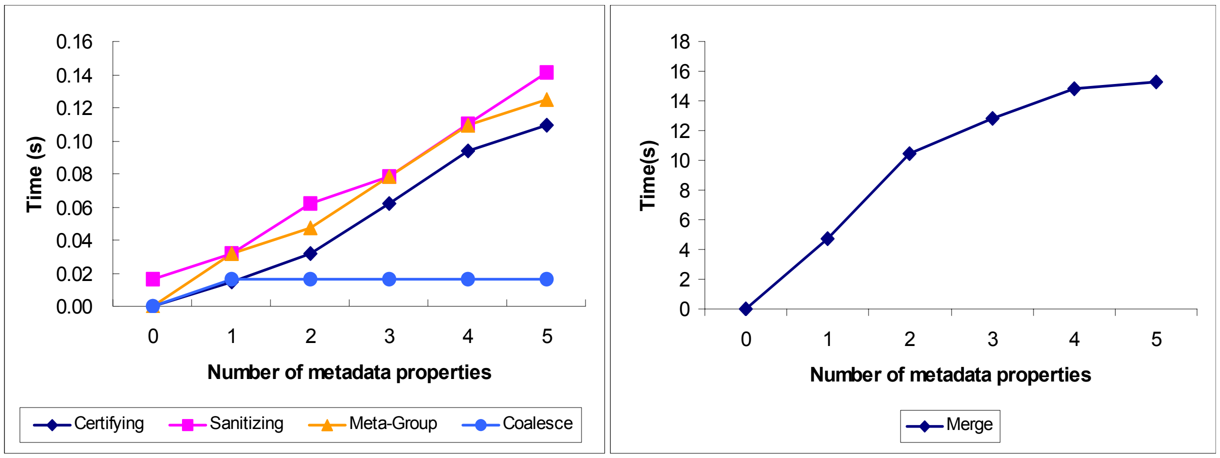


Figure 35 Results of in-memory experiment 2, varying the number of metadata properties

7.1.3 Varying the number of metadata values

This experiment fixes both the size of the data document and the number of metadata properties (we use only one metadata type, security, in this experiment). We start with 5 users, and increase it to 30 users in increments of 5. The result of this test is shown in Figure 36. Again, the time grows linearly in the amount of metadata. Merge is much more expensive once again (because of the node construction), so it is plotted separately.

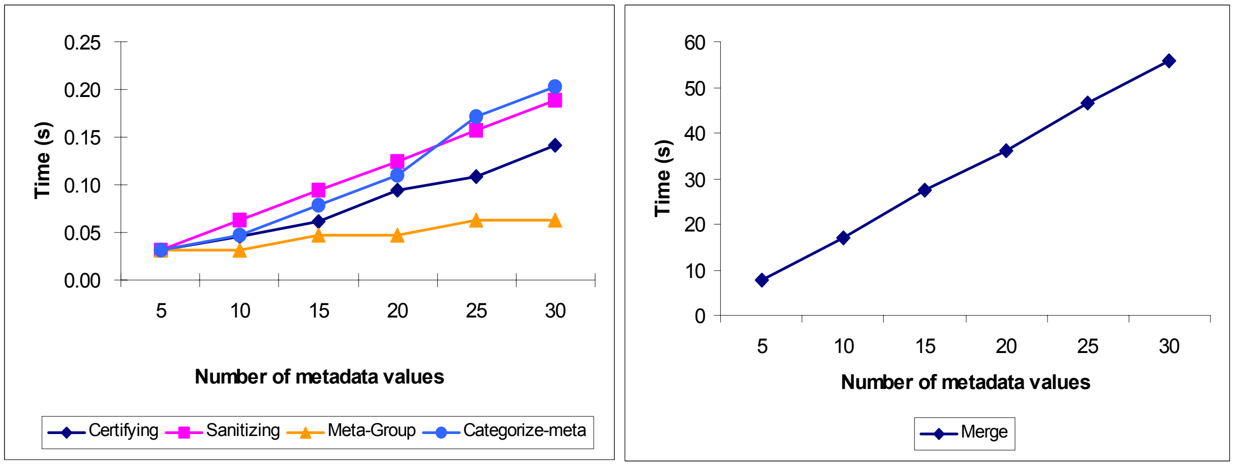


Figure 36 Results of in-memory experiment 3, varying the number of metadata values

7.1.4 Varying the number of metadata levels

This experiments fixes the size of the data document and the number of metadata properties and values. We increase the nested level of metadata from 1 to 5.

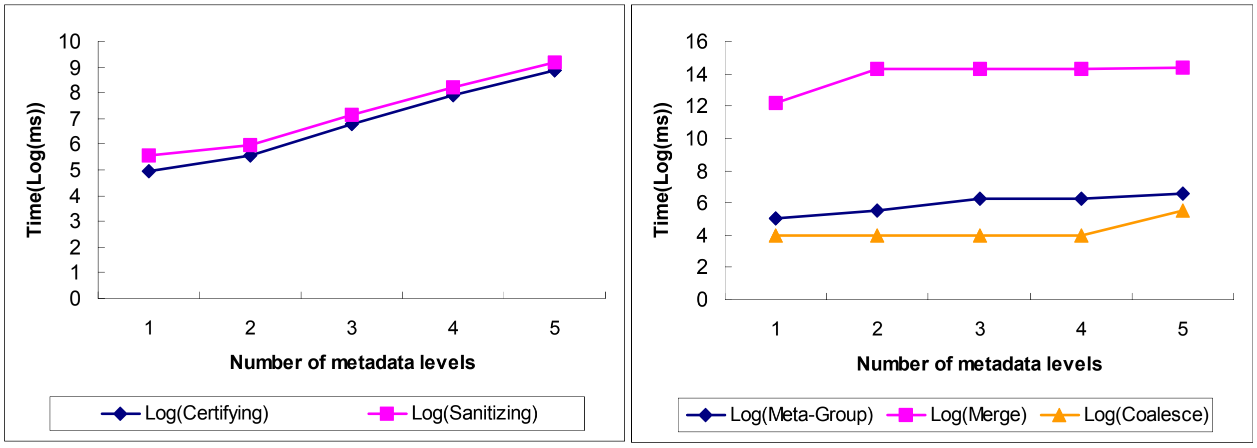


Figure 37 Results of in-memory experiment 4, varying the number of metadata levels

The result of this test is shown in Figure 37. When we increase the nested level of metadata, the performance would actually increase more than linearly because for every node, we have to do the evaluation recursively to the last level. That's why we use the logarithmic scale in this figure. As we see, the log lines increase linearly or even close to constantly, so the performance still scales elegantly within the reasonable upper bound.

In summary, the in-memory experiments indicate that as the amount of data or metadata increases, the cost also increases. But the growth is largely linear. Hence, we anticipate that the prototype will scale to large, real-world applications.

7.2 PERSISTENT PERFORMANCE TESTS

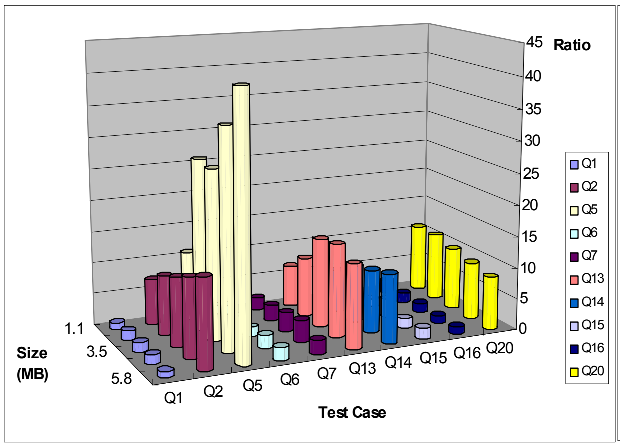
For our persistent version of implementation, we also designed a series of experiments to evaluate its scability, compare the performance of the new metadata-enabled queries with the original data queries running on eXist, as well as to compare the performance of different query strategies mentioned above. We chose to use a different data and query set from the in-memory version, namely these from XMark [53], so that we'll have a more comprehensive and better understanding of its performance.

7.2.1 Varying the data document size and optimization technique

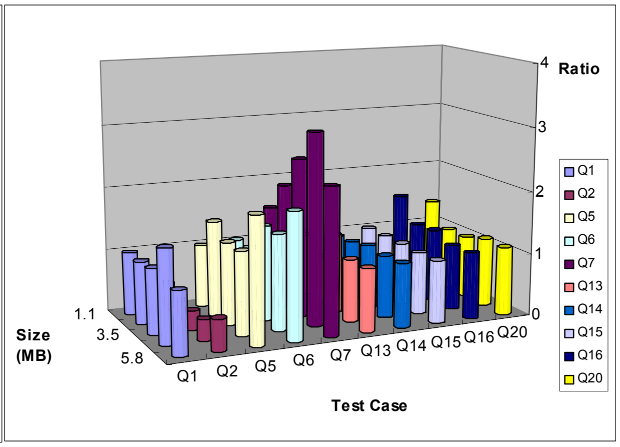
The first experiment tests the scalability of the MetaXQuery implementation with increasing data document sizes as well as the effect of using applying the filterByPerspective functions in the query. We generated the data documents with XMark benchmark factor that increases from 0.01 to 0.05 (document size increases from 1.1 MB to 5.8 MB). We first ran the benchmark queries in eXist (with no metadata extensions). We then added the same metadata value to all the elements in the data document. We chose to use the same metadata value everywhere because that gives the filter the most work to do. Every time the filter processes the data nodes, it can't throw any of them away. That way, we also guarantee that the query result after applying the filters is exactly the same as the data query with no metadata extension. Table 5 shows the parameters for the data and metadata for experiment one. Finally, we performed two variations of the experiment. In the worst-case variant, we applied a filterByPerpective function to every path expression in the benchmark query. That is obviously a very expensive strategy. A smart query optimizer can remove some of the redundant filters so that we don't have to do it for every path expression in the query. So we created an optimized worst-case scenario. We manually optimized the queries to so that they have about the same work and result size as the worst-case queries and also have minimum number of filterByPerspective necessary. These optimized queries in the experiment are shown in Table 12. We have eliminated the name space and the second parameter of the filterByPerspective function, the perspective node for less space. But both of them are all the same for all the queries. For the original XMark queries, please refer to [53].

Experiment	Factor	Document Size (MB)	# Metadata Properties	# Metadata Trees
1	0.01, 0.02, 0.03, 0.04 0.05	1.1, 2.3, 3.5, 4.8, 5.8	2	1
2	0.01	1.1	0 to 3	# of elements
3	0.01, 0.02, 0.03, 0.04 0.05	1.1, 2.3, 3.5, 4.8, 5.8	2	1

Table 11 Persistent experiments parameters



(a) Worst Case Queries



(a) Optimized Worst Case Queries

Figure 38 Results of persistent experiment 1, varying the data document size and optimization technique

Figure 38 shows the result of the first experiment. The data is plotted as a performance ratio showing the overhead of MetaXQuery. A value of 1 indicates that eXist and eXist-MetaXQuery give the same performance. A value greater than 1 means that eXist-MetaXQuery performed worse than eXist. For instance, MetaXQuery was at most 1.5 times slower on Q1 (worst-case), but often had no slowdown (the height of each bar is near 1). As we see from the optimized queries chart in Figure 38 (b), adding the smart support for perspective in

MetaXQuery doesn't add a great amount of overhead on XQuery evaluation in exist, and it scales pretty well as the data document gets larger. But that depends heavily on the rewriting strategy. In the worst case queries shown in Figure 38 (a), the performance is much worse because there's a filter for every path expression and even more importantly, it is called in every iteration of the FOR loop. That's why the ratios are much higher than the optimized case. So obviously the optimization of the filters in the MetaXQuery system is crucial to the performance.

7.2.2 Varying the number of metadata types

Experiment 2 tests the scaling factor of the MetaXQuery processor with increasing types of metadata. The experiment fixes the size of the data and metadata documents, and increases the types of metadata from 0 to 3. The queries used in this experiment are the optimized ones and we use the same number of metadata fragment as the total number of element in the data document. All of these fragments have different values so that the effect of index is maximized. Figure 39 shows the performance of selected queries in experiment 2. We only choose some of the queries here to make the graph clearer. All of the queries show a sub-linear increase in time with the increasing number of metadata types, which indicates that the system will scale as the number of metadata types increases.

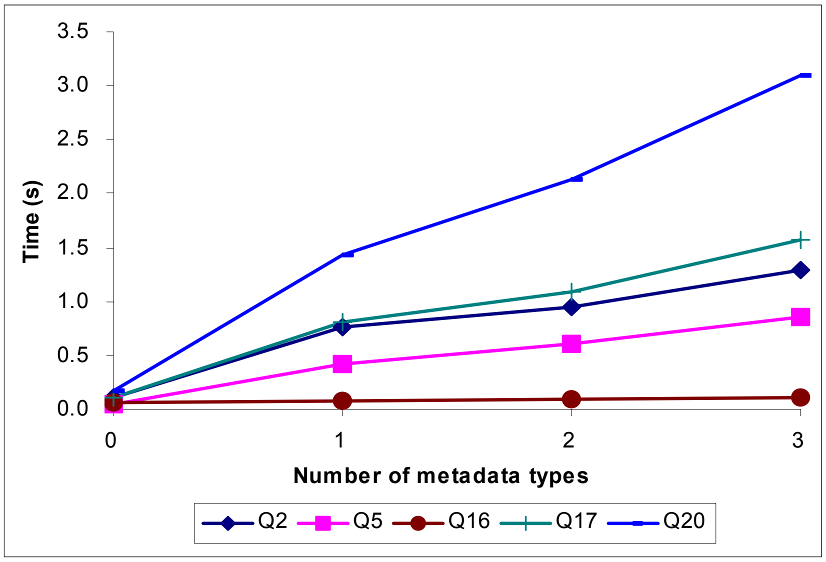


Figure 39 Results of persistent experiment 2, varying the number of metadata types

7.2.3 Varying the implementation strategy

Experiment 3 compares the different query execution strategies of the filterByPerspective function. As mentioned in Section 4.2, there could be a naïve implementation of the filterByPerspective function that iterates through every data node and checks if its metadata matches the perspective (as shown in Figure 19). We suggested that a much better strategy would be to use the index to find all the metadata fragments that match the perspective first and then join them with the data nodes (as shown in Figure 20). In this experiment we compare these two query evaluation strategies. We call the first strategy the traversal method and the second the index join method. Like the first experiment, we will use a series of data documents from factor 0.01 to 0.05 and we also use the same metadata value for all the elements in the data document. For the traversal method, it is pretty much the same as using different metadata fragment or value because it checks the metadata value of every data node. For the index join algorithm on the other hand, the index doesn't help much so it's mostly the cost of joins. This is actually reasonable because with the help of data structures like B+-trees, index lookup is always a very fast operation. So the dominant part of the query execution is still going to be the joins. We use the same experimental setup as the previous experiments so that the results are the same as the original data query. The queries used in this experiment are the optimized queries.

Figure 40 shows the performance of the experiment. As we expected, the index join method outperforms the traversal method in most of the test cases. And furthermore, it shows much better scalability. The cost of the traversal method increases linearly with the data size, but the indexes and joins (a merge-join) show sub-linear increase. The results justify our work in developing the more efficient metadata association join algorithm in filtering data with metadata conditions.

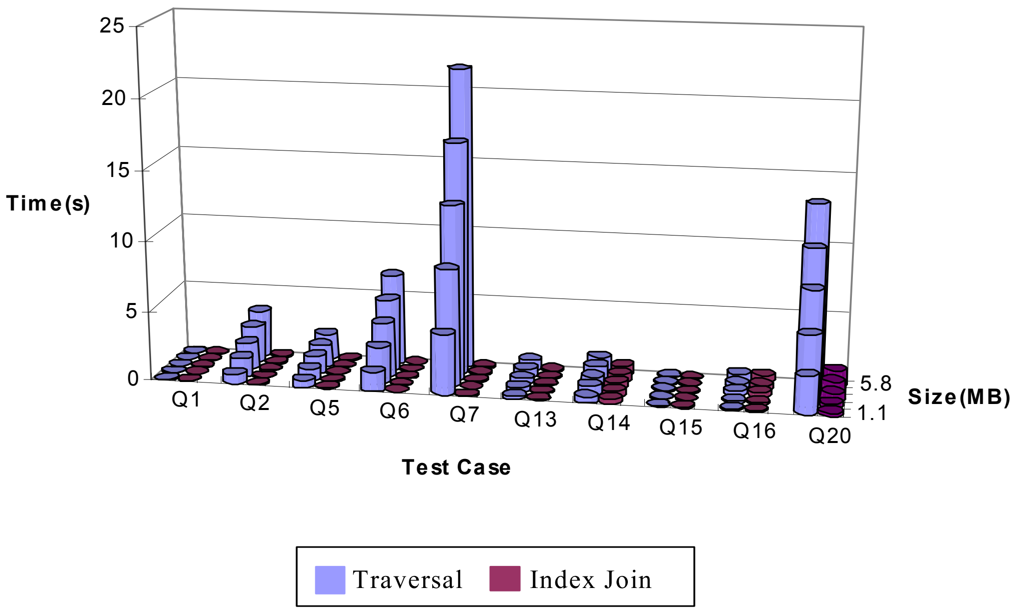


Figure 40 Results of persistent experiment 3, varying the implementation strategy

Query	Optimized Queries
1	filterByPerspective(FOR $b IN /site/people/person[@id="person0"] RETURN $b/name)/text()
2	<increase> {filterByPerspective(FOR $b IN /site/open_auctions/open_auction RETURN $b/bidder[1])/text()} </increase>
5	count(filterByPerspective(FOR $i IN /site/closed_auctions/closed_auction WHERE $i/price/text() >= 40 RETURN $i/price))
6	count(filterByPerspective(FOR $b IN /site/regions RETURN $b//item))
7	FOR $p IN /site RETURN count(filterByPerspective($p//description)) + count(filterByPerspective($p//annotation)) + count(filterByPerspective($p//email))
13	FOR $i IN filterByPerspective(/site/regions/australia/item) RETURN <item name="{$i/name/text()}"> {$i/description} </item>
14	filterByPerspective(FOR $i IN /site//item WHERE contains($i/description,"gold") RETURN $i/name)/text()
15	<text> {filterByPerspective(FOR $a IN /site/closed_auctions/closed_auction/annotation/description/parlist/listitem/text/emph/keyword RETURN $a)/text()} </text>
16	FOR $s IN FOR $a IN /site/closed_auctions/closed_auction WHERE not (empty ($a/annotation/description/parlist/listitem/text/emph/keyword/text())) RETURN $a/seller RETURN <person id="{filterByPerspective($s)/@person}" />
20	<result> <preferred> {count (filterByPerspective(/site/people/person/profile[@income > 100000]))} </preferred> <standard> {count (filterByPerspective(/site/people/person/profile[@income < 100000 and @income >= 30000]))} </standard> <challenge> {count (filterByPerspective(/site/people/person/profile[@income < 30000]))} </challenge> <na> { count (filterByPerspective(for $p in /site/people/person where empty($p/@income) return $p)) } </na> </result>

Table 12 Optimized worst case queries

CHAPTER EIGHT
RELATED WORK

There are few papers on systems to manage metadata in native XML databases, but lots of research in related areas. In this chapter we consider related work in representing and querying metadata on the web, systems that support metadata, research on special kinds of metadata, and papers relating to query evaluation in native XML databases.

Generally there are two methods to represent metadata: the *embedded* method and the *annotated* (or *superimposed*) method. In the embedded method, the metadata is stored together with the data or referenced directly from the data. We have already seen this method in the examples in Figure 21. The advantage of this method is that it's easy and quick to access metadata and ascertain its relationship with the data. The disadvantage though, is that the schema of the data is different from its original version, so the application needs to be adapted for that change. On the other hand, the annotated method keeps the original version of the data intact and annotates the data with external metadata. The most widely used language on the web for annotating a document with metadata is the Resource Description Framework (RDF) [78], which has also become an important language for supporting the Semantic Web [4]. RDF consists of three object types: the *resources*, the *properties*, and the *statements*. A statement is a specific resource together with a named property plus the value of that property for that resource, and these three individual parts of a statement are called, respectively, the *subject*, the *predicate*, and the *object*. So generally, an RDF statement can be formatted in XML as "<*subject*> HAS <*predicate*> <*object*>". Discussions are ongoing about accessing and querying RDF data [77] but in general RDF query languages use a very different data model than the family of XML query languages. An RDF document can be used to describe any resource that can be located by a URI. An instance of the RDF data model is created when an RDF document is parsed. One drawback of using RDF to represent the metadata is that the data and metadata have different data models. The RDF data model is a directed, labeled graph, unlike an XML data model such as DOM. This forces users to switch data models and query languages as they move from metadata to data (or vice-versa). Query language implementers also face a challenge in optimizing queries that jump between the data models. A third drawback is that the RDF data model supports only descriptive metadata; it has no support for enforcing the semantics of proscriptive metadata like security. Several strategies for unifying the representation of XML and RDF have been proposed [31] [54] , but query languages have largely targeted either RDF or XML. There have been several RDF query languages proposed in the literature including RQL [40], SeRQL [7], and TRIPLE [55]. For a comparison of these RDF query languages, please refer to [30].

We have chosen to use a uniform XML data model for both data and metadata instead of using RDF to capture metadata. There could be difficulties in such approach when we want to convert an existing RDF model into our MetaDOM because of the mismatch of data models. RDF uses directed edge graph, but our MetaDOM consists of multiple trees. Without special constructs in the tree nodes (e.g., ID and IDRef), it can't capture cycles in a graph. But in general cases, an RDF data model doesn't contain cycles either so it's relatively a minor issue for our framework.

There are several systems that support metadata similar to MetaXQuery. Mihaila et al. suggest annotating data with quality and reliability metadata and discuss how to query the data and metadata in combination [46]. The SPARCE system wraps or superimposes a data model with a layer of metadata [48]. The metadata is active during queries to direct and constrain the search for desired information. Systems that provide mappings between metadata (schema) models are also becoming popular [45]. MetaXQuery differs from these systems by focusing on XQuery extensions to support metadata, and by building a framework whereby the semantics of individual kinds of metadata can be specified as "plug-in" components.

Support for particular kinds of metadata has been researched. Our approach is to build an infrastructure that supports a wide range of different kinds of metadata in the same vein as our previous efforts with the semistructured data model [22] and XPath data model [23]. Two of the most important and most widely discussed types of (proscriptive) metadata are temporal metadata and security metadata. Temporal extensions of almost every W3C recommendation exist, for instance, *TT*XPath [20], τXQuery [26], and τXSchema [17]. Another important area of time-related research is techniques for storing and retrieving past versions of data [9] [13]. Grandi has an excellent bibliography of time-related web papers [28]. There has also been research on security in XML management systems, e.g., [5] and [18]. MetaXQuery differs from all of the above papers because it builds a single, extensible framework to support the many kinds of metadata rather than just one kind of metadata.

One of the particular area of problems we tackled is grouping and restructuring XML data with metadata. Though grouping is important it has not received much attention in the research community. Paparizos et al. showed that the lack of an explicit grouping construct in XQuery forces users to employ inefficient strategies when grouping [50]. They proposed improving the efficiency of grouping by adding an explicit grouping operator to the TAX algebra in the context of the TIMBER [33] native XML database system. We focus instead on grouping with metadata.

In relational databases, the relational algebra is a set of operations that manipulate relations as they are defined in the relational model. Because of their algebraic properties, they are often used in database query optimization as an intermediate representation of a query to which certain rewrite rules can be applied to obtain a more efficient version of the query. In the XML world, people have also designed algebras to represent queries on the tree data model as well as optimization rules to transform the queries into efficient low-level operators. The XML algebras include the Tree Algebra for XML (TAX) [32], the XML Query Algebra [27], and an algebra on a graph structure [3]. Research on query execution plans and especially join algorithms include containment queries [56], structural joins [1] [2] [14], and twig joins [8] [35]. We extended the TAX algebra to support our metadata functionalities and also borrowed ideas from some of the join algorithms for our metadata association join.

There are also several native XML database system in the literature, such as eXist [44], Xindice [61] and Galax [24]. We have chosen eXist because it’s completely implemented in Java (so easier to extend) and also performs the best in our own benchmark [36].

CHAPTER NINE

CONCLUSIONS AND FURTURE WORK

In this dissertation, we first outline an XML data model, called MetaDOM, that supports data annotated with metadata. Different semantics can be given to different kinds of metadata in MetaDOM. It reuses the DOM data model for metadata so each level is an XML data model itself. It also supports recursively nested metadata, i.e., meta-metadata. We show how to extend DOM to implement MetaDOM.

We then present a query language, called MetaXQuery, for the extended data model. MetaXQuery extends XQuery with an additional "`meta`" axis, and functions to certify and sanitize data with regard to metadata, group data and metadata, to merge nodes with metadata, to match an implicit metadata perspective during path evaluation, to restructure metadata by coalescing metadata values, and to output metadata. To execute MetaXQuery expressions, we show how to translate these user-level queries into low-level algebraic operators. The low-level operators are expressed in MetaTAX, which is an extension of TAX. A naïve implementation of MetaXQuery would result in very inefficient query execution plans, so next we discuss how to optimize the processing of metadata-related queries by using index and join algorithms.

We implemented a prototype system of MetaDOM and MetaXQuery, in both an in-memory model and a persistent model. The in-memory model is based on the Apache Xerces2 Java Parser platform, and the persistent model is based on a native XML database, eXist. To evaluate the performance of the prototype system and to compare different implementation strategies, we developed an XML query processing benchmark. We introduce the data set, our XML document generator, and benchmark test cases. We use this benchmark platform to test our in-memory model implementation and another public benchmark, XMark to test our persistent model implementation. Experiments with the prototype show that the cost of supporting metadata is linearly related to the amount of metadata present and scales elegantly with the increase of both data and metadata.

Our metadata extension of DOM and XQuery is clean and minimal. These standards are extended by this research, but not modified. The extensions are upwards compatible because all XPath and XQuery evaluated in a MetaXQuery implementation on a data model without metadata would have the same behavior as an XQuery implementation. We define most of our extensions of XQuery as functions not only because it is easy to access and use metadata, but also because each functional unit can be encapsulated and modularized. So the extensions can be implemented in an XML query processing system as user-defined functions or external functions, and they can also be optimized individually. That way, we achieve high cohesion and low coupling.

We give MetaTAX translations to primitive functions like **getMetadataValues**, **filterByMetadataValues**, and **filterByPerspective** but not the **certify**, **sanitize**, and **grouping** functions because their definitions are much more complex. Their implementation depends largely on the low-level system, so we decide to use just a declarative definition to specify what exactly should happen at the conceptual level, but let the implementation to decide what's the best to do. In the in-memory model, the **certify**, **sanitize**, and **grouping** functions manipulate directly on the DOM data model trees, but they could have a very different implementation in the persistent model by using indexes. We didn't have time to cover that part in our research yet, but it could be an interesting supplement to our current system. Though we reuse the general-purpose index of eXist for all metadata types in the persistent implementation, we have yet to incorporate special-purpose metadata indexes (e.g, temporal indexes) to achieve better performance for some metadata types.

In future we plan to extend XML Schema to specify the layout and validation semantics for both data and metadata. XML Schema currently only specifies the structure of data in the document and (some) constraints on the data (e.g., keys), but with metadata support, there could be additional rules that capture the relationship between data and metadata, and additional constraints on the metadata (e.g., the lifetime of a parent should encompass those of its children).

We need to either extend the rules of the XML Schema or add an additional layer of metadata-specific schema rules to enforce the semantics of the metadata.

Another theoretical issue is the completeness of our extensions to XQuery and TAX. We proposed several extensions that we think are the most obvious and representative, but by no means comprehensive. Though it's not our primary purpose to do so, it would be nice from the theoretical point of view to come up with a set of extensions that are complete in the sense of all possible scenarios. We need to find out the minimal of such set is and prove that it is actually complete. With such theoretical background, we can approach other issues much more easily.

In the three major types of metadata mentioned in the introduction, we have studied *descriptive* and *proscriptive* metadata in our research, but not *interpretive* metadata. Interpretive metadata tells the user or the application how to understand a piece of data. It is very common in scientific data, for instance an integer value that represents a temperature is only meaningful in the context of a particular metric or scale, e.g., Celsius. Different people and countries use very different metrics and scales, so for them to communicate, it is best if an application can automatically and intelligently translate between interpretations. With the help of interpretive metadata, this task should be a lot easier to accomplish. We plan to incorporate interpretive metadata into our framework.

Another avenue of future research is to explore the possibility of using Aspect Oriented Programming (AOP) techniques to model the extended metadata functionalities. AOP is a new programming paradigm that allows programmers to modularize code that exhibits similar behavior but doesn't fit naturally into a single program module. Instead, they are modularized by *crosscut concerns* and *aspects*. Each kind of metadata, e.g., time, roughly corresponds to an aspect. The idea is to weave support for a kind of metadata into the data using AOP techniques. Research is needed however because data weaving will likely be quite different from code weaving. But the paradigm of AOP could potentially help us reduce a lot of redundant work in deploying metadata support into the existing XQuery systems.

Finally, existing XML tools can to be extended to make the use of metadata easier. One such example is XML, DTD or XML Schema editors. Traditionally they are only for editting XML data, but they can be expanded to create an integrated display and editing of data together with metadata, such as hyperlinks between data and metadata, automatic formatting of metadata, and intelligent help to create and modify metadata. Each type of metadata could be treated specially depending on their semantics to help users. For instance transaction time changes could be shaded to indicate age or colors could be used to show differing levels of security. Moreover, an XML query processing tool can be augmented to remember the session of the current user and apply his settings as perspective in all his queries. All these extensions help users to easily handle metadata in such applications.

BIOBILIOGRAPHY

[1] Shurug Al-Khalifa, H. V. Jagadish. “Multi-level operator combination in XML query processing”. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management*, pp. 134-141, McLean, Virginia, November 2002.

[2] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, Divesh Srivastava. “Structural Joins: A Primitive for Efficient XML Query Pattern Matching”. In *Proceedings of the 18th International Conference on Data Engineering* (*ICDE*), pp. 141-152, San Jose, California, February-March 2002.

[3] David Beech, Ashok Malhotra, Michael Rys. “A Formal Data Model and Algebra for XML”. *W3C XML Query working group note*, September 1999.

[4] T. Berners-Lee. “Why RDF Model is Different from the XML Model”, September 1998. `http://www.w3.org/DesignIssues/RDF-XML.html`, current as of July, 2005.

[5] Elisa Bertino, Silvana Castano, Elena Ferrari, Marco Mesiti. “Specifying and Enforcing Access Control Policies for XML Document Sources”. In *WWW Journal*, Volume 3, Number 3, pp. 139-151, 2000.

[6] Timo Böhme, Erhard Rahm. “XMach-1: A Benchmark for XML Data Management.” In *Proceedings of the 9th Datenbanksysteme in Büro, Technik und Wissenschaft* (*BTW*), pp. 264-273, GI-Fachtagung, Oldenburg, März 2001.

[7] Jeen Broekstra, Arjohn Kampman. “SeRQL: An RDF Query and Transformation Language”. Submitted to the *International Semantic Web Conference* (*ISWC*), 2004.

[8] Nicolas Bruno, Nick Koudas, Divesh Srivastava. “Holistic twig joins: optimal XML pattern matching”. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 310-321, Madison, Wisconsin, June 2002.

[9] Peter Buneman, Sanjeev Khanna, Keishi Tajima, Wang Chiew Tan. “Archiving scientific data”. In *ACM Transactions on Database Systems* (*TODS*), Volume 29, Number 1, pp. 2-42, 2004.

[10] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton. “The OO7 benchmark.” In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 12-21, Washington, D.C., May 1993.

[11] Rick Cattell *et al*. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann Publishers, San Francisco, 1996.

[12] Donald D. Chamberlin, Jonathan Robie, Daniela Florescu. "Quilt: an XML Query Language for Heterogeneous Data Sources". In *the World Wide Web and Databases, Third International Workshop WebDB*, pp. 1-25, Dallas, Texas, May 2000.

[13] Sudarshan S. Chawathe, Serge Abiteboul, Jennifer Widom. "Representing and Querying Changes in Semistructured Data". In *Proceedings of the 14th International Conference on Data Engineering* (*ICDE*), pp. 4-13. Orlando, Florida, February 1998.

[14] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, Carlo Zaniolo. "Efficient Structural Joins on Indexed XML Documents". In *Proceedings of the 28th International Conference on Very Large Data Bases* (*VLDB*), pp. 263-274, Hong Kong, China, 2002.

[15] Dublin Core Metadata Initiative (DCMI) Glossary. `http://dublincore.org/documents/usageguide/glossary.shtml`, current as of July 2005.

[16] Dublin Core Metadata Initiative. "Dublin Core Metadata Element Set, Version 1.1: Reference Description", DCMI Recommendation, June 2003. `http://dublincore.org/documents/2003/06/02/dces`, current as of July 2005.

[17] Faiz Currim, Sabah Currim, Curtis E. Dyreson, Richard T. Snodgrass. "A Tale of Two Schemas: Creating a Temporal XML Schema from a Snapshot Schema with τXSchema". In *Proceedings of the 9th International Conference on Extending Database Technology* (*EDBT*), pp. 348-365. Crete, Greece, March 2004.

[18] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati. "Securing XML Documents". In *Proceedings of the 7th International Conference on Extending Database Technology* (*EDBT*), pp. 121-135. Konstanz, Germany, March 2000.

[19] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, Dan Suciu. "A Query Language for XML". In *Computer Networks,* Volume 31, Numbers 11-16, pp. 1155-1169, May 1999.

[20] Curtis E. Dyreson. "Observing Transaction-time Semantics with *TT*XPath". In *Proceedings of the 2nd International Conference on Web Information Systems Engineering* (*WISE*), pp. 193-202, Kyoto, Japan, December 2001.

[21] Curtis E. Dyreson. "Temporal Coalescing with Now, Incomplete Information, and Granularity". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 169-180. San Diego, California, June 2003.

[22] Curtis E. Dyreson, Michael H. Böhlen, Christian S. Jensen. "Capturing and Querying Multiple Aspects of Semistructured Data". In *Proceedings of 25th International Conference on Very Large Data Bases* (*VLDB*), pp. 290-301, Edinburgh, Scotland, September 1999.

[23] Curtis E. Dyreson, Michael H. Böhlen, Christian S. Jensen. "METAXPath". In *Proceedings of the International Conference on Dublin Core and Metadata Applications*, pp. 17-23, Tokyo, Japan, 2001.

[24] Mary F. Fernández. "Implementing XQuery 1.0: The Story of Galax". In *the 11th Datenbanksysteme in Business, Technologie und Web* (*BTW*), pp. 30-47, Karlsruhe, März 2005. `http://www.galaxquery.org`

[25] Roy Goldman, Jennifer Widom. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases." In *Proceedings of the 23rd International Conference on Very Large Data Bases* (*VLDB*), pp. 436-445, Athens, Greece, 1997.

[26] Dengfeng Gao, Richard T. Snodgrass. "Temporal Slicing in the Evaluation of XML Queries". In *Proceedings of 29th International Conference on Very Large Data Bases* (*VLDB*), pp. 632-643, Berlin, Germany, September 2003.

[27] Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, Philip Wadler. "The XML Query Algebra". `http://www.w3.org/TR/2000/WD-query-algebra-20001204/2001`

[28] Fabio Grandi. "Introducing an Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web". In *SIGMOD Record*, Volume 33, Number 2, June 2004.

[29] Jim Gray. *Database and Transaction Processing Performance Handbook.* `http://www.benchmarkresources.com/handbook`, 1993.

[30] Peter Haase, Jeen Broekstra, Andreas Eberhart, Raphael Volz. "A Comparison of RDF Query Languages". `http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query`

[31] Jane Hunter, Carl Lagoze. "Combining RDF and XML Schemas to Enhance Interoperability Between Metadata Application Profiles". In *Proceedings of the 10th International World Wide Web Conference* (*WWW*), pp. 457-466, Hong Kong, China, May 2001.

[32] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. "TAX: A Tree Algebra for XML". In *Proceedings of the 8th International Workshop on Database Programming Languages* (*DBPL*), pp. 149-164, Frascati, Italy, September 2001.

[33] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, Cong Yu. "TIMBER: A native XML database". In *VLDB Journal*, Volume 11, Number 4, pp. 274-291, December 2002. `http://www.eecs.umich.edu/db/timber`

[34] Christian S. Jensen and Curtis E. Dyreson (eds), Michael H. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. K¨afer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in *Temporal Databases:*

Research and Practice, O. Etzion, S. Jajodia, andS. Sripada (eds.), Springer-Verlag, pp. 367–405, 1998.

[35] Haifeng Jiang, Wei Wang, Hongjun Lu, Jeffrey Xu Yu. "Holistic Twig Joins on Indexed XML Documents". In *Proceedings of 29th International Conference on Very Large Data Bases* (*VLDB*), pp. 273-284, Berlin, Germany, September 2003.

[36] Hao Jin and Curtis E. Dyreson. "A Benchmark for XPath Evaluation". Submitted to *WWW* Journal.

[37] Hao Jin and Curtis E. Dyreson. "Grouping in MetaXQuery". In *Proceedings of 5th International Conference on Web Information Systems Engineering* (*WISE*), pp. 688-693, Brisbane, Australia, November 2004.

[38] Hao Jin and Curtis E. Dyreson. "Sanitizing using Metadata in MetaXQuery". In *Proceedings of the 2005 ACM Symposium on Applied Computing* (*SAC*), pp. 1372-1376, Santa Fe, New Mexico, March 2005.

[39] Hao Jin and Curtis E. Dyreson. "Capturing, Querying and Grouping Metadata Properties in XML". Submitted to *IEEE TKDE*.

[40] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, Michel Scholl. "RQL: a declarative query language for RDF". In *Proceedings of the 11th International World Wide Web Conference* (*WWW*), pp. 592-603, Honolulu, Hawaii, May 2002.

[41] Michael H. Kay. Saxon: The XSLT and XQuery Processor. `http://saxon.sourceforge.net`

[42] Michael H. Kay. "Saxon: Anatomy of an XSLT processor." In *IBM DeveloperWorks*, February 2001. `http://www-106.ibm.com/developerworks/library/x-xslt2`

[43] Stephan Kepser. "A Proof of the Turing-completeness of XSLT and XQuery". *Technical Report SFB 441*, Eberhard Karls Universitat Tubingen, May 2002.

[44] Wolfgang Meier. "eXist: An Open Source Native XML Database." `http://exist.sourceforge.net`

[45] Sergey Melnik, Erhard Rahm, Philip A. Bernstein. "Rondo: A Programming Platform for Generic Model Management". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 193-204, San Diego, California, June 2003.

[46] George A. Mihaila, Louiqa Raschid, Maria-Esther Vidal. "Using Quality of Data Metadata for Source Selection and Ranking". In *Informal Proceedings of the Third International Workshop on the Web and Databases* (*WebDB*), pp. 93-98, Dallas, Texas, May 2000.

[47] David Mosberger and Tai Jin. "*httpperf - A Tool for Measuring Web Server Performance*". In *SIGMETRICS Performance Evaluation Review*, Volume 26, Number 3, pp. 31-37, December 1998.

[48] Sudarshan Murthy, David Maier, Lois M. L. Delcambre, Shawn Bowers. "Superimposed Applications using SPARCE". In *Proceedings of the 20th International Conference on Data Engineering* (*ICDE*), pp. 861, Boston, Massachusetts, March 2004.

[49] Ullas Nambiar, Zoé Lacroix, Stéphane Bressan, Mong-Li Lee, Ying Guang Li. "Benchmarking XML Management Systems: The XOO7 Way." *Technical Report TR-01-005*, Department of Computer Science, Arizona State University, 2001.

[50] Stelios Paparizos, Shurug Al-Khalifa, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Divesh Srivastava, Yuqing Wu. "Grouping in XML". In *XML-Based Data Management and Multimedia Engineering - EDBT Workshops*, pp. 128-147. Prague, Czech Republic, March 2002.

[51] Jonathan Robie, Joe Lapp, David Schach. "XML Query Language (XQL)". In *the W3C Query Language Workshop*, Boston, Massachussets, December 1998.

[52] Kanda Runapongsa, Jignesh M. Patel, H. V. Jagadish, Shurug Al-Khalifa. "The Michigan Benchmark: A Microbenchmark for XML Query Processing Systems". In *the Workshop of Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web* (*EEXTT*), pp. 160-161, 2002.

[53] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, Ralph Busse. "XMark: A Benchmark for XML Data Management". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pp. 974-985, Hong Kong, China, 2002.

[54] Peter F. Patel-Schneider, Jérôme Siméon. " "The Yin/Yang Web: A Unified Model for XML Syntax and RDF Semantics". In *IEEE Transaction of Knowledge and Data Engineering* (*TKDE*), Volume 15, Number 4, pp. 797-812. July/August 2003.

[55] Michael Sintek, Stefan Decker. "TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web". In *Proceedings of the Semantic Web Conference*, pp. 364-378, Sardinia, Italy, June 2002.

[56] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, Guy M. Lohman. "On Supporting Containment Queries in Relational Database Management Systems". In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, 2001.

[57] Shuohao Zhang, and Curtis E. Dyreson. "Adding Valid Time to XPath". In *Proceedings of the International Workshop on Database and Network Information Systems* (*DNIS*), pp. 29-42, Aizu, Japan, December 2002.

[58] Apache XML Project. Xalan-Java. `http://xml.apache.org/xalan-j`

[59] Apache XML Project. Xalan-C++. `http://xml.apache.org/xalan-c`

[60] Apache XML Project. Xerces Java Parser. `http://xml.apache.org/xerces-j`

[61] Apache XML Project. Xindice. `http://xml.apache.org/xindice`

[62] DOM4j: The Flexible XML Framework for Java. `http://www.dom4j.org`

[63] Jaxen: Universal Java XPath Engine. `http://jaxen.sourceforge.net`

[64] Simple API for XML (SAX). `http://www.saxproject.org`

[65] Microsoft® XML Core Services (MSXML). `http://msdn.microsoft.com/xml`

[66] HBench-OS Operating System Benchmarks.
`http://www.eecs. harvard.edu/~vino/perf/hbench`

[67] Microsoft® Site Server. InetMonitor.
`http://www.microsoft.com/siteserver/site/DeployAdmin/InetMonitor.htm`

[68] MindCraft® WebStone. `http://www.mindcraft.com/webstone`

[69] Standard Performance Evaluation Corporation. SPEC CPU2000 v1.2.
`http://www.spec.org/cpu2000`

[70] Standard Performance Evaluation Corporation. SPECWeb99.
`http://www.specbench.org/osg/web99`

[71] Transaction Processing Performance Council. TPC Benchmarks. `http://www.tpc.org`

[72] VeriTest®. WebBench. `http://www.veritest.com/benchmarks/webbench`

[73] VeriTest®. WinBench 99 version 2.0.
`http://www.veritest.com/ benchmarks/winbench`

[74] International Organization for Standardization (ISO). *Information Technology-Database Language SQL*. Standard No. ISO/IEC 9075:2003. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.)

[75] World Wide Web Consortium. "Cascading Style Sheets, level 1", W3C Recommendation, Dec 1996, revised Jan 1999.
`http://www.w3.org/TR/1999/REC-CSS1-19990111`

[76] World Wide Web Consortium. “Document Object Model (DOM) Level 3 Core Specification, Version 1.0”, W3C Recommendation, April 2004. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407`

[77] World Wide Web Consortium. RDF Data Access Working Group. `http://www.w3.org/sw/DataAccess`

[78] World Wide Web Consortium. “RDF Primer”, W3C Recommendation, February 2004. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210`

[79] World Wide Web Consortium. “Extensible Markup Language (XML) 1.0 (Third Edition)”, W3C Recommendation, February 2004. `http://www.w3.org/TR/2004/REC-xml-20040204`

[80] World Wide Web Consortium. “XML in 10 Points”. `http://www.w3.org/TR/2000/REC-xml-20001006`

[81] World Wide Web Consortium. “XML Information Set”, W3C Recommendation, February 2004. `http://www.w3.org/TR/2004/REC-xml-infoset-20040204`

[82] World Wide Web Consortium. “XML Linking Language (XLink) 1.0”, W3C Recommendation, June 2001. `http://www.w3.org/TR/2001/REC-xlink- 20010627`

[83] World Wide Web Consortium. “XML Pointer Language (XPointer)”, W3C Working Draft, August 2002. `http://www.w3.org/TR/2002/WD-xptr-20020816`

[84] World Wide Web Consortium. “XML Path Language (XPath) Version 1.0”, W3C Recommendation, November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`

[85] World Wide Web Consortium. “XQuery 1.0: An XML Query Language”, W3C Working Draft, April 2005. `http://www.w3.org/TR/2005/WD-xquery-20050404`

[86] World Wide Web Consortium. “XQuery 1.0 and XPath 2.0 Data Model”, W3C Working Draft, April 2005. `http://www.w3.org/TR/2005/WD-xpath-datamodel-20050404`

[87] World Wide Web Consortium. “XQuery 1.0 and XPath 2.0 Full-Text Use Cases”, W3C Working Draft, April 2005. `http://www.w3.org/TR/2005/WD-xmlquery-full-text-use-cases-20050404`

[88] World Wide Web Consortium. “XML Schema Part 0: Primer Second Edition”, W3C Recommendation, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028`

[89] World Wide Web Consortium. “XML Schema Part 1: Structures Second Edition”, W3C Recommendation, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028`

[90] World Wide Web Consortium. “XML Schema Part 2: Datatypes Second Edition”, W3C Recommendation, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028`

[91] World Wide Web Consortium. “Extensible Stylesheet Language (XSL) 1.0”, W3C Recommendation, October 2001. http://www.w3.org/TR/2001/REC-xsl-20011015

[92] World Wide Web Consortium. “XSL Transformations (XSLT) 1.0”, W3C Recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116

[93] World Wide Web Consortium. “XML Encryption Syntax and Processing”, W3C Recommendation, December 2002.
http://www.w3.org/TR/2002/REC-xmlenc-core-20021210

[94] http://www.xml-benchmark.org

[95] http://www.comp.nus.edu.sg/~ebh/XOO7.html

[96] http://db.uwaterloo.ca/~ddbms/projects/xbench

[97] http://www.eecs.umich.edu/db/mbench

[98] http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html

[99] http://www.hyperdictionary.com