SCHEMA-BASED PARTITIONING
FOR OPTIMIZED QUERYING OF
XML DATA

Master’s Thesis, September 10, 2012

Johar A. Syed
Schema-Based Partitioning for Optimized Querying of XML Data

THESIS
Submitted in the partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
COMPUTER SCIENCE
by
Johar A. Syed
Born in Karachi, Pakistan

Web Information Systems
Department of Software Technology
Faculty, EEMCS, Delft University of Technology
Delft, the Netherlands
http://www.ewi.tudelft.nl/en/

X-Hive Corporation B.V.
Aert van Nesstraat 45
3012 CA Rotterdam
The Netherlands
Efficient querying and retrieval of data from XML databases is a challenging task, especially when these DBs contain a large number of documents, potentially millions of them. One way to scale out is by spreading the documents over multiple nodes, but running all queries on all nodes has poor scale-out qualities. Here we propose a solution to this problem whereby the document space is partitioned on the basis of some document characteristic that is referenced by the queries, and can be inferred efficiently for a given query; this allows the query dispatch system to preemptively avoid issuing the query to the nodes that are positively known to contain empty result.

Our partitioning scheme provides an environment for automated placement of data and thus allows the users (i.e., query authors or clients) to operate at a higher abstraction level; in other words, the information about the location of data is not required as an input from the user. Satisfiability of XPath, which determines the emptiness and non-emptiness of its answer at compile time, plays an important role in our strategy of determining data location and query optimization. To further optimize the performance of a query in addition to what is attainable through the process of context identification, indexes are utilized which are built on such XML elements that are important from point of view of expected queries.

The strategies have been designed, implemented and tested on top of xDB. We demonstrate that the performance of distributed XQueries is improved after passing through our context identification phase. The precision of our context identification is 90% on our dataset and query set. We show interesting tradeoffs to alternative approaches in terms of xDB data pages instead of time-based parameters to get highly reliable and consistent results. Large datasets and XPath/XQuery with different syntactic properties have been used which show the applicability of proposed techniques to real world XML database systems.

Graduation Committee:
Prof. dr. ir. G.J. Houben, Faculty EEMCS¹, TU Delft
Dr. ir. A.J.H. Hidders, Faculty EEMCS, TU Delft
Ir. H.J.A.M. Geers, Faculty EEMCS, TU Delft
Michiel Veen, X-Hive Corporation (subsidiary of EMC Corporation), Rotterdam

¹ Electrical Engineering, Mathematics and Computer Science
This document has been produced in the thesis project as part of the Master program in Computer Science at Delft University of Technology (TU Delft). I started working on my thesis project in October 2011 at X-Hive Corporation (fully subsidiary of EMC Corporation).

I would like to take this opportunity to thank my university advisor, Jan Hidders as work on this thesis would have been quite difficult to manage without his guidance. He always gave valuable feedback and advice during group and individual meetings, and guided me in the right direction. I would also like to thank Prof. Geert-Jan Houben for allowing me to perform thesis assignment in the Web Information Systems group. I’d like to convey my thanks to the members of Alpha group (my project group) for their knowledgeable comments and suggestions, and to my thesis committee members for evaluating my thesis work.

I gained a lot of knowledge and practical experience while working at X-Hive with people who have been contributing to the field of XML databases for years. I am very grateful to my advisor, Michiel Veen, who provided me the opportunity of doing thesis project at X-Hive. His support and valuable comments were of great help. Thanks to my direct advisor, Mark Polman, whose advice and constructive criticism helped me to progress in the right direction. I would also like to thank Carla Spruit for her support in understanding the model and implementation of XQuery and its parser in xDB.

Lastly, thanks to my family and friends for their encouragement and support during this thesis project.

Have an enjoyable reading experience!

Johar A. Syed
Delft, the Netherlands
September 10, 2012
## Contents

Acknowledgement ........................................................................................................ iii

Contents ........................................................................................................................... v

List of Figures ................................................................................................................. vii

List of Tables .................................................................................................................... viii

Part I  Introduction and Background Knowledge ....................................................... 1

Chapter 1: Introduction ............................................................................................... 3

1.1 Research Questions ................................................................................................. 4

1.2 Research Strategy .................................................................................................... 5

1.3 Thesis Outline .......................................................................................................... 5

Chapter 2: Extensible Markup Language .................................................................... 7

2.1 Schema Definition .................................................................................................. 8

2.2 Distributed Computing and its Use in XML Databases ........................................ 10

2.3 XPath ...................................................................................................................... 10

2.4 XQuery ................................................................................................................... 11

2.5 Related Work .......................................................................................................... 13

Chapter 3: xDB ............................................................................................................. 15

3.1 xDB's Architecture Overview ................................................................................ 15

3.2 Indexes in xDB ....................................................................................................... 16

Part II  Methodology and Design ................................................................................. 17

Chapter 4: Data Partitioning, Identification of Query Context and Auto-Indexing Framework ............................................................................................................. 19

4.1 Problem Description ............................................................................................... 19

4.2 Data Partitioning .................................................................................................... 25

4.3 Auto-Indexing ....................................................................................................... 33

4.4 Query Set ............................................................................................................... 35

4.5 Data Partitioning Scheme and its Limitations ....................................................... 37

4.6 Summary ............................................................................................................... 38

Chapter 5: Implementation of Data Partitioning ...................................................... 39

5.1 Architecture Overview ......................................................................................... 39

5.2 Partitioning Algorithm ......................................................................................... 40
List of Figures

Figure 1: Sample of XML Document ................................................................. 7
Figure 2: XML Document (Tree Representation) .................................................. 8
Figure 3: Document Type Definition (Sample) .................................................... 9
Figure 4: XML Schema Definition (Sample) ..................................................... 9
Figure 5: XQuery Processing Model ............................................................... 12
Figure 6: Federation in xDB ........................................................................... 15
Figure 7: Maximum CPU Usage with 250,000 Data Documents ......................... 20
Figure 8: Maximum CPU Usage in Multiple Parallel Queries with 50,000 documents.. 21
Figure 9: Time to Calculate Data Location ...................................................... 24
Figure 10: Response Time Difference in Similar and Dissimilar Queries .............. 25
Figure 11: XML Schema .............................................................................. 26
Figure 12: Context Identification Process ....................................................... 28
Figure 13: Schema s1 about Book .................................................................. 32
Figure 14: Schema s2 about Book .................................................................. 32
Figure 15: Query Statistics ........................................................................... 34
Figure 16: Data Partitioning Model ................................................................ 40
Figure 17: XML schema .............................................................................. 43
Figure 18: Query Context Identification Model ............................................... 51
Figure 19: Auto Indexer Model .................................................................... 57
Figure 20: Updated Statistics ....................................................................... 60
Figure 21: Random Partitioning .................................................................... 63
Figure 22: Query's Performance without Context Identification ......................... 70
Figure 23: Addition in Response Time of Query 1 due to Parallel Queries (ms) .... 72
Figure 24: Addition in Response Time of Query 1 due to Parallel Queries (%) .... 72
Figure 25: Additional Response Time of Q1 due to Parallel Queries (%) in 250,000 Docs... 73
Figure 26: Additional Response Time of Q1 due to Parallel Queries (ms) in 250,000 Docs.. 73
Figure 27: Schema Cluster-Based v/s Random Partitioning ............................. 74
Figure 28: xDB (Physical Structure) ............................................................... 90
Figure 29: xDB (Logical Structure) ............................................................... 91
Figure 30: xDB (Embedded Architecture) .................................................... 92
Figure 31: xDB Client (Client-Server Architecture) ......................................... 92
Figure 32: xDB Server (Client-Server Architecture) ....................................... 93
Figure 33: Execution Time of XQuery .......................................................... 94
Figure 34: Speed of Data Update ................................................................. 94
Figure 35: Schema (Author) ........................................................................ 96
Figure 36: Schema (Book) .......................................................................... 97
Figure 37: Schema (Movie) ......................................................................... 98
Figure 38: Schema (Publisher) ..................................................................... 99
Figure 39: Schema (TVShowInfo) .............................................................. 100
List of Tables

Table 1: Experiment for measuring Weight of Cost Function (250,000 Documents) ..........23
Table 2: Experiment for measuring Weight of Cost Function (210,000 Documents) ..........23
Table 3: False Positives........................................................................................................69
Table 4: Precision of Context Identification (Summarized Result).................................70
Table 5: Empty/Non-Empty Result Set.............................................................................75
Table 6: Performance Gain through Value Index...............................................................76
Table 7: Characteristics of xDB’s Different Indexes............................................................95
Part I
Introduction and Background Knowledge
As the world becomes increasingly digital, businesses in general and big enterprises in particular are generating and storing large amounts of data, intended for consumption by existing and potential applications of various kinds. However, a lot of computing resources are required for performing data manipulation efficiently. High-end computing machinery is expensive and unaffordable by most consumers. According to Moore’s Law [1], “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase”.

Gordon Moore made this prediction in the 1960s and it continues to hold true till this day. But on the other hand, growth in the amount of data is happening at a higher speed than the advancement in computational capabilities as predicted by Moore’s law. To handle such huge amounts of data, it is not viable to continue the use of computing resources for its processing in sequential or non-parallel manner. So the question arises: how to retrieve Big Data in an efficient way with contemporary technology? One viable and effective way of dealing with the enormity of the data is to store it in a distributed fashion and process it in smaller chunks using multiple nodes.

Besides the need for getting cheap alternative of computing from data distribution, there is also a need by applications to create and manage data across multiple organizations. Apart from scalability, data could also be distributed for local availability reasons. There are many challenges that may arise due to data distribution and need for interdependence of data. In this situation, effective and efficient design principles of data distribution, querying and transaction management are essential.

Another reason behind opting for data distribution is to bring it closer to the way it is being accessed over Web. After the advent of Web and recent trends of communication through social networking sites like Twitter and Facebook, enormous amounts of data are being produced such that the data is also more and more distributed in nature [2].

In distributed database management systems, data is distributed to take advantage of parallelism. In the relational model of databases, vertical fragmentation or sharding of tables is a frequent phenomenon for getting desired data distribution and there exist various specific methods within those structuring techniques. Distribution of other kinds of databases, e.g., object oriented databases, is partly inspired by those techniques [3], [4]. This is also the case with XML databases.
There are basically two areas where the processing of queries can be made efficient: (i) how the data is stored and (ii) which execution plan will be better for optimal query performance; the focus of this thesis is closer to the former aspect. The optimal processing of queries over collections of XML documents depends on the used physical data model and the access methods [5]. In the distributed environment, right allocation of data is very important, which is also tightly linked to the workload or nature of the queries that are frequent in the system being considered, as well as the size and structure of the XML data in question.

A number of choices are available in designing the strategy for data distribution and allocation, including e.g., schema-based distribution and value-based distribution. For example, suppose there are two sets of documents d and d', containing /a/b and /c/e xml element paths respectively, where ‘a’, ‘b’, ‘c’ and ‘e’ are element names. In schema-based distribution, d and d’ may end up on different physical locations if their allocation is based on the criteria that no element is common between them (there could be other criteria as well within schema-based distribution). In value-based distribution, value of elements could be a candidate for deciding about the allocation of d and d’. Let’s say that ‘b’ in ‘/a/b’ (d) has a range of values from 0-1000. Documents in d can be divided into two sets: d_1 and d_2. d_1 contains documents in which ‘b’ has values from 0-499 and d_2 comprises of documents in which ‘b’ has values in the ranges of 500-1000. In this thesis we’re concerned with only the schema-based classification of data when devising our strategy for data partitioning.

Our main objective is to reduce the response time of XQuery and XPath in a distributed environment. As mentioned earlier, right data partitioning is important so that the query load could be reduced, e.g., by avoiding having to send the query to the nodes that cannot serve it due to the absence of the desired data. This works because there’s a data locality model built by pre-processing the incoming documents at the time of their allocation to DB node. Thus, in order to minimize the cost of query evaluation (see section 4.1.3 for the cost function), we need to focus on two aspects: appropriate data partitioning and accurate data location modeling.

1.1 Research Questions
The following research questions have been addressed in this thesis:

1. Can the performance of XQuery or XPath be improved through data partitioning in a distributed system?

A partitioning scheme that divvies up the workload to multiple nodes is expected to improve overall performance. In which case, the next questions would be:

a) What basis should be used for document partitioning?

We explore some similarity based document partitioning schemes and compare them with certain naive approaches for partitioning. We then further explore the schema based partitioning method we propose.

b) Given a schema based partitioning method, how to deal with documents that are similar but do not have exactly the same schema?
2. Can the determination of emptiness or non-emptiness of the results of an XQuery or XPath at compile time be exploited to improve query performance?

Distribution of data at multiple physical sites or separate and independent storage places enables the exploitation of inter-query parallelism and intra-query parallelism (only the former is considered here). If emptiness of the results of an XQuery (i.e., whether no documents in a set match the given query) can be determined at a preprocessing stage, then there is an opportunity to avoid the expensive operation of its full evaluation using actual data documents. We explore how this optimization can be materialized. The process of performance optimization produces the following two derived questions:

a) How to compute the emptiness or non-emptiness of the results of an XQuery and XPath using schema at compile time?

b) Can the performance of a query that is deemed satisfiable at compile time (but may or may not produce actual results) be further optimized, using the empirical or statistical models constructed against the document schemas?

1.2 Research Strategy

In this thesis, we want to explore and build a computationally efficient mechanism of publishing and querying distributed data. Different alternative approaches of distributed storage and querying of XML data have been designed, implemented and tested for a comparative analysis of their performance. The main performance metric or evaluation criterion is the response time of the query. Different datasets were used and several classes of queries were tested to capture the performance difference and to highlight the best approach for a given class of problem. Results are then analyzed to demonstrate how our proposed approach is better than the baseline approaches.

1.3 Thesis Outline

The remainder of this thesis is organized into four parts or nine chapters. Chapter 2 describes some core concepts about XML and related technologies, which serves as the technology reference for the rest of the thesis. Chapter 3 shares some knowledge about xDB (a native XML database), which was used for the implementation and performance analysis of our data partitioning scheme, query context identification model and the indexing technique.

Chapter 4 describes the problem and defines our thesis objective in more detail: it gives an overview of the strategies of data partitioning of XML documents, determination of location of data (context identification) and indexing techniques, defines cost function and criteria of evaluating the design alternatives and lays down the detailed syntax of the query set (used in our experiments). Chapter 5 describes the architecture of our data partitioning scheme and explains algorithms for its implementation. Chapter 6 covers the implementation details and algorithms of context identification and auto-indexing mechanisms. Chapter 7
presents the design and implementation of the baseline approaches and Chapter 8 shows empirical results and provides interpretation and analysis.

In the last part of the thesis, Chapter 9 presents a summary and describes the future work, highlighting the potential improvements in the proposed model, along with a list of further possibilities for enriching the features.
Extensible Markup Language, abbreviated XML, is now considered a widely popular standard for representation, exchange and storage of data over the Web. The rapid gain in its popularity can be attributed to the fact that XML is human-readable and can be processed by machines [6]. From its origin, it has been ensured that the standard of XML meets greater usability and it should be easy to understand, making it convenient for researchers and developers to interact with it, and enabling the user community to develop several applications on top of it with the capability of performing simple or even complex tasks, efficiently and effectively.

XML describes a class of data objects called XML documents [7]. An XML document is a primary unit for data representation. XML is a subset or a restricted form of SGML, the Standard Generalized Markup Language [7]. The two most important entities in an XML document are called ‘element’ and ‘attribute’. Elements and attributes provide semantics to the data of XML document. Each XML document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag [7]. Starting element of a document is called ‘root’ element. Each element (root or non root) has an empty or a nonempty set of attributes and can hold child elements. Each attribute is described by its name which contains a value. The concept of “ID” and “IDREF” adds some more value to the expressive power of XML. They specify references between two elements and these reference links can be internal (within same document) or external (between documents). Figure 1 shows an example of a typical XML document.

```
<?xml version="1.0" encoding="UTF-8" ?>
<book bookid="1">
   <name>Harry Potter and the Philosopher's stone</name>
   <author>J.K. Rowling</author>
   <publisher>Bloomsbury Publishing</publisher>
   <preface />
   <datePublished>June 30th 1997</datePublished>
   <chapters>
      <chapter>
         <title>The Vanishing Glass</title>
         <body>The Dursleys had everything they wanted, but they also had a secret, and their greatest fear was that somebody would discover it.</body>
         <summary>Dursleys had everything</summary>
      </chapter>
      <chapter>... ...</chapter>
   </chapters>
</book>
```

Figure 1: Sample of XML Document
In Figure 1, `<book>` is the root element of XML document. It has six children i.e. `<name>`, `<author>`, `<publisher>`, `<preface>`, `<datePublished>` and `<chapters>`. Element `<book>` has an attribute “bookid” with a value of “1”.

The hierarchical nature of data representation in XML brings it closer to how objects are represented in the real world. This hierarchical nature enables us to represent XML data model in tree format (see Figure 2 for tree representation of an XML document). It gives us better understanding, especially in case of large and complex XML documents.

![Figure 2: XML Document (Tree Representation)](http://w3schools.com/xml/xml_tree.asp)

In Figure 2, ‘parent’, ‘child’ and ‘siblings’ describe relationships between elements. Elements ‘parent’ and ‘child’ are pretty much self explanatory where elements in ‘siblings’ represent children at the same level from hierarchical perspective.

The XML data model allows the insertion of multiple elements or multiple attributes with the same name within a single document at different hierarchical positions. Thus such elements or attributes represent different information because of their presence in different contexts.

2.1 Schema Definition

There are two main standards to define the schema of XML documents. XML schema can be defined through Document Type Definition (DTD) and XML Schema Definition (XSD) [8], explained in section 2.1.1 and 2.1.2 respectively.

2.1.1 Document Type Definition (DTD)

DTD defines the hierarchical structure of elements and attributes of XML document. Though data type of elements and attributes can be defined but it is limited, compared to
its successor XSD (described in section 2.1.2). Specifications of root element and relationships like child, parent, siblings between elements can be described through DTD. A given XML document can be validated for the correctness of the format against the grammar defined in DTD. Figure 3 shows an example of a typical DTD document.

![Figure 3: Document Type Definition (Sample)](image)

### 2.1.2 XML Schema Definition (XSD)

XSD is considered as a replacement or a successor to DTD and it has some differences and benefits over DTD. XSD provides a richer set of tools to specify the structure and the constraints of elements and attributes. It is relatively easier to define data types and restrict data patterns of values of elements and attributes in XSD. For example, the exact number of occurrences of a certain element can easily be specified. Same convenience is available for indicating order (sequence, choice or all) of sub-elements. XSD can also provide the specification of additional document information, such as normalization and defaulting of attribute and element values [8]. XSD has the same syntactic structure as used by XML documents, which also makes it easy for the developers to build processing tools for them. Figure 4 shows a sample of XSD schema:

![Figure 4: XML Schema Definition (Sample)](image)
2.2 Distributed Computing and its Use in XML Databases

Nowadays, the need of distributed processing or computing is felt like never before because of growing demand for reliability, fault tolerance, and responsiveness by the user. It is also important from service provider’s perspective because distribution of tasks provides an economical alternative solution for processing large amount of data. The idea of distribution in information handling has significantly raised the level of performance because computing resources like CPU, I/O services are delegated to different processing sites and they can perform their jobs independent of each other in parallel.

There are mainly two aspects of distributed computing. One is about the distribution of function or processing and the second one is related to the distribution of data to different physical sites. This distinction is also applied in other domains including Distributed Databases. The concept of distributed databases is not only limited to the world of relational databases but its doors are opened to other types of database management systems including XML database. The distribution of functionalities or data also eliminates single point of failure.

2.2.1 Types of Query Parallelism

Parallelism in distributed databases can be achieved through two kinds of techniques. We can exploit “inter-query parallelism” by executing multiple queries at the same time on different nodes. “Intra-query parallelism” can be exploited by issuing a query to multiple nodes where node will act as an independent processing center and will send back the result to the query-originating node. Both techniques are good in providing the benefits of distributing computing but when it comes to select one, it mainly depends on the type of queries as well as on the degree of workload. We believe that one of the most important factors contributing towards the potential benefits of parallelism exists in the form of a smart partitioning scheme and this is where our partitioning approach comes into play to prove our hypothesis.

2.2.2 Native XML Databases

There exist various open source XML database systems like proposed in [9], [10] as well as some proprietary systems. With the idea of having native XML database, researchers and application developers started to work on achieving high level of efficiency in storing and retrieving XML documents. Currently XML database management systems offer several features like indexing, views, etc. As a user, we are used to them in Relational database but from the research point of view, these techniques of optimization cannot be applied directly in an XML database because of its inherent differences with relational databases in data modeling.

2.3 XPath

XPath is a query language which operates on the abstract, logical structure of an XML document called data model described in XDM [11]. Its basic unit is ‘expression’ and nested expressions can be inserted into it. XPath 2.0 is the current version of standards for XPath language, according to W3C’s recommendation. It supports more data types as compared to its previous version (i.e. XPath 1.0). XPath 2.0 is a superset of XPath 1.0 with backward compatibility, and nearly all XPath 1.0 expressions deliver the same result with XPath 2.0 [11].
As mentioned earlier, XML data model represents data in hierarchical, tree like structure. So it was required to have a language which is capable of navigating through a certain path for accessing data in this structure. Keeping this objective in mind, XPath was recommended by W3C and navigation of data nodes is considered as an important feature of XPath [11]. Data nodes can be represented by elements, attributes. Because of convenient data navigation and accurate pattern matching, other languages of XML such as XQuery and XSLT have to rely heavily on XPath to retrieve nodes or atomic values from XML documents. Several path expressions can be used to access same node of an XML document which increases flexibility for the query expression. For example, ‘/book/name’ and ‘//book/name’ are equivalent path expressions which can access name elements of root element book from XML document (see Figure 1 for XML document).

XPath also plays an important role in making accessibility operation very convenient. Each XPath can have two parts: data we want to access or update and the context of it. Context helps users in traversing to the node or literal value they want to access or update. For example, a user wants to access names of an employee working in a company. This information can be retrieved through XPath: ‘employee/name’, where ‘employee’ is the context which helps database management system to look for those ‘name’ elements which are children of ‘employee’.

### 2.3.1 Abbreviated and Unabbreviated Path Expressions
Path expressions can be written in two forms: abbreviated and unabbreviated. Unabbreviated form of a path like ‘/child::book/child::name’ can be expressed in abbreviated XPath as ‘/book/name’. Since abbreviated form is easier and is used commonly in specifying a query, therefore we use it in this thesis.

### 2.3.2 Predicates in XPath
Predicates can be embedded in path expressions to provide selectivity in data retrieval and other kinds of manipulation with XML documents. To retrieve name element from document (in Figure 1) whose author is ‘J.K.Rowling’ can be done through: ‘/book[author = ‘J.K.Rowling’]/name’.

### 2.4 XQuery
XQuery is a query language designed to query a broad range of XML information sources, including both databases and documents. There are numerous methods which can make the processing of XQuery very efficient, especially in a distributed environment. XML documents can be fragmented and distributed to different nodes and each node can act as an independent processing node. Some proved methods of fragmentation in XML have basically been originated from Relational Databases, include horizontal, vertical and hybrid fragmentation. The authors in [12] have shown the methods of how XML data can be fragmented and how query can be executed in distributed models. In this way, queries can be sent to specific nodes and thus they can get significant performance improvement by division of task and by parallelizing the execution of sub-tasks.

Generally, the processing model of XQuery can be described in four phases [13], shown in Figure 5.
The first three stages of XQuery processor are performed at compile time whereas last one is part of runtime process.

**Parser:** This module ensures that the XQuery is syntactically correct with respect to the stated grammar. For example, XQuery: “let $x$ in //books” is syntactically incorrect and parser in this case should raise an error.

**Normalization:** Changes like removal of data redundancy is performed in this phase.

**Static semantics checker:** This module has the responsibility of identifying violation of type matching. For example, comparison between string type and number type is illegal and its checking is part of semantics’ integrity at compile time.

**Dynamic semantics checker and XQuery Execution:** The execution of XQuery is performed in this stage. Any error like out of heap space is checked in this phase.

### 2.4.1 Indexing

Indexing plays a crucial role in speeding up the query processing irrespective of whether database architecture is based on distributed or centralized ground. Indexes are also used in relational and in other types of databases. Their widespread use proves the point that they are important part of query preparation and execution.

In relational databases, there are various types of index structure. B+ index is considered the most popular one. In it, non-leaf nodes can consist of up-to ‘k’ keys. Left child contains keys less than the parent node and right child has keys greater than the one with parent node. Leaf nodes contain pointers to the data values. Indexes in native XML database have been further explained in Section 3.2.
2.5 Related Work

The existing work in the area of data partitioning and processing of XQuery in distributed environment mostly lacks the treatment of performance data partitioning perspective.

The technique of data partitioning proposed in this thesis utilizes the concept of document clustering based on XSD schema. Prior research has proposed several methods for clustering that can be divided into two main categories:

- Clustering of XML documents using schema
- Schema-less clustering of XML documents

**Clustering of XML documents using Schema:** [14], [15], [16], [17], [18], [19] advocate the determination of XML tree edit distance between two documents. This distance is used to find structural similarity between two XML documents. Tree editing involves two operations: insertion and deletion at different nodes of a XML document. Research presented in [14] is considered the first one to use edit distance. [14], [15] proposed the idea of applying edit operation at leaf or non-leaf nodes. The authors in [16], [17] support different approach and calculate edit distance using leaf nodes only. The proposed technique in [18] offers optimization in the calculation of edit distance by using structural summaries of two trees. The approach in [20] is innovative because it brings collaborative clustering in which each node in the network accesses a portion of a given document collection and communicates with all the other nodes to perform a clustering task in a collaborative fashion.

**Schema-less clustering of XML documents:** Another option for doing clustering of documents is to use documents rather than schemas. Though many of XML documents follow schemas [21] but such kind of structural and semantic information aren’t available for all XML documents, so schema-less clustering methods will be beneficial for those XML documents that do not have schemas but this is not necessary as schema from data document can be generated. [22], [23], [24] have presented and supported the idea of schema-less clustering.

The authors of [24] claim that semantic information is preserved in their approach. According to them, each document is decomposed into macro-path sequences in which the semantic and structural information are maximally preserved. After computing the similarity value among these macro-path sequences, the similarity matrix of the document collections is constructed. Finally, the hierarchical clustering technique is used to group these documents on the basis of the generated similarity matrix.

**Commonality of XPaths of Schemas in Data Partitioning:** We have modeled our clustered based partitioning algorithm on the idea of finding common XPath expressions in the two schemas being compared. To the best of our knowledge, no one except [25], [26] have applied XPath commonality in building the clusters of XML documents. According to [25], XML document contains XML-sequences where each sequence corresponds to a set of elements ordered by decreasing level of tree hierarchy and is labeled by a sequence-id. XML sequence is contained by another sequence if it is a subsequence of that XML-sequence. In a set of XML-sequences, an XML sequence is maximal if it is not contained by any other XML-sequence. This maximal XML-sequence is referred as common XPath (CXP). The authors in [26] utilize paths of schema (DTD) to determine the common paths in them.
They propose the strategy of building paths using child relation between element tags of a schema for comparing them with the paths of other schemas.

**Shipement of Query or Data:** In the domain of how query and its data should be transmitted in distributed environment, [27] proposes XQuery extension that equip XQuery with function shipping style distributed querying abilities. According to them, XRPC is applied to each sub-query so that it can be executed over heterogeneous data sources in distributed environment. But they seem silent on whether system will transform an XQuery into XRPC XQuery or user will specify query in XRPC format. The authors of [28] also present similar kind of extension (XQueryD) to bring query shipping capabilities to XQuery. In XQueryD, user has to specify the location of data nodes into the query.

**Determination of Data Location and Satisifiability of XPath:** Much attention hasn’t been paid towards design specification of model of a system which determines itself where query data is located, despite the complexity presence due to richer set of queries possible in XQuery 1.0. In this thesis, we use Satisfiability of XPath in schema to determine the location of data. Work in [29] proposes the satisfiability of XPath in DTD to check whether the answer of a query is nonempty on an XML tree of a document. Our algorithm of satisfiability and context identification expand this principle into XQuery in the presence of XML schema definition instead of DTD. According to [30], the satisfiability problem for XPath expressions allows the detection of erroneous expressions and optimizes query performance by removing expressions that return an empty result. In this thesis, this concept is applied differently because it is used to determine the emptiness/non-emptiness of XQuery’s result set instead of removal of individual path expressions (having empty result) from XQuery.
Chapter 3

xDB

xDB is a native XML database management system for storing XML documents in an integrated manner. It provides high scalability and high performance for handling large quantities of documents. Applications can be built on top of it to access data using application programming interface (API). xDB supports various standards recommended by W3C. In DOM, xDB implements an extended DOM level 3 interface for updating content, structure and style of documents. It includes navigation and retrieval of nodes within XML documents. Documents can be stored in libraries, which are implemented as DOM nodes. Libraries can be attached with other libraries in a hierarchical manner, similar to the nested structure of directories or folders within a file system. A database has exactly one root library which is also the topmost library in the hierarchy. All operations can be performed on documents as well as on libraries. XQuery is also supported by xDB which can address any type of information in an XML document. Selections of XML nodes can be made through it while retrieving data or constructing new data. XPath and XPointer are also supported by xDB.

3.1 xDB’s Architecture Overview

There are two different architectures of xDB: Embedded and Client-Server architecture, shown in Appendix B. Our prototype uses Embedded architecture in which same JVM hosts client and xDB server. xDB can contain one or more databases and all these databases can be grouped under xDB federation. A federation provides a server connection and applications can connect to database through it. Figure 6 shows an inner architecture of a federation containing two instances of database.

Figure 6: Federation in xDB
xdb database consists of one or more segments. A segment is a logical storage location within an xDB database which has one or more files and each file consists of one or more pages. Files can be added to a database segment using the administration client or the xDB API. Our prototype uses xDB API to add new XML documents in the default segment. Under the hood, xDB extends DOM implementation for retrieving data from documents and storing new ones. Retrieval of data can be made with the help of document ID, document name, XQuery and XPath.

3.1.1 Data Page and its Size
An XML document can be stored in one or more pages. Page size can be specified to the power of 2, with a range from 512 to 65536. Since our prototype is designed, implemented and tested on Windows platform, therefore the page size is set to 4096 bytes, same as the page size of Windows operating system.

3.2 Indexes in xDB
As we discussed in section 2.3, an index improves query efficiency. xDB also utilizes indexes to speed up query processing. In xDB, an index is stored in a pair (key-value), where key is a string and value is a number type and a node set value. The indexes are updated automatically when the corresponding data is updated. Indexes can be defined for libraries and documents. xDB currently supports the following indexes:

- **Path indexes** - index the value of elements and attributes.
- **Full text indexes** - index elements by element text values or an attribute value.
- **Multi path indexes** - index multiple elements without requiring the explicit configuration of every single index path.
- **Value indexes** - index elements by an element value or attribute value.
- **Metadata indexes** - index the value of a metadata entry and points to individual documents.
- **Library indexes** - index the content of a library.
- **ID attribute indexes** - index elements by their unique element ID.
- **Element name indexes** - index elements by name.

In this thesis project, we use value indexes to build index list which can contain multiple value indexes. Value indexes are live indexes that are automatically updated when elements or attributes are inserted, replaced, or removed. Value indexes can be created for a library or a document.
Part II
Methodology and Design
Chapter 4

Data Partitioning, Identification of Query Context and Auto-Indexing Framework

As mentioned in the previous chapters, query efficiency can be improved through different methods of data partitioning (like fragmentation) or through query optimization methods (like choosing best evaluation plan or indexing). We mainly focus on the aspect of how data partitioning is linked to the optimization of XQuery or XPath in the distributed environment. In addition, we also explore how our data partitioning assists in the automatic context identification for getting location of data (explained in sections 4.2.1, 6.2).

This chapter describes the problem in more detail and defines some concepts necessary to understand our strategies for partitioning and locating data and provides an overview of those strategies including an indexing technique. It also describes the evaluation criteria used to assess the effectiveness of our strategies at solving the said problem.

A prototype of the proposed strategies has been built on top of EMC Documentum xDB (see section 3.1 for xDB) using Java jdk 1.7 (Eclipse 3.7) on Windows 7. One instance of xDB server runs in the client whose job is to maintain Data-Dictionary (defined in section 4.2), to partition and allocate new set of documents, and to identify query context. Query context is defined as the identity of the node or a set of nodes and the identity of set of documents on those nodes that are relevant to the query. Each processing node that receives the query for evaluation has a separate xDB server running on it and its job is to prepare the result of the query and send the result back to the client node.

4.1 Problem Description

XML databases can contain huge amounts of data and retrieving a certain piece of information efficiently from a large database is a challenging task. Efficiency is usually characterized by either the computational cost or the response time of the query. The latter includes the time taken in processing the query and that in returning its results back to the user. The processing time is mainly determined by the number of data pages accessed by the database server (details later in this section). We would like to minimize this component of the query response time by use of a data partitioning scheme in combination with a precise identification of the query context (for locating data) and an indexing technique.

Let the set of schemas representing the entirety of documents be $S = \{s_1, s_2, ..., s_m\}$. Then the set of documents can be represented as $D = \{D_{s_1}, D_{s_2}, ..., D_{s_m}\}$ where $D_{s_1}$ is the subset of documents conforming to schema $s_1$, $D_{s_2}$ to schema $s_2$ and so on. Let the set of processing nodes $N$ be $\{N_1, N_2, ..., N_n\}$ where $n$ is the number of nodes. Each element of $N$
runs an instance of xDB server. XML documents and XQuery/Xpath are input at the client for data storage and data retrieval respectively. We do not consider the update operation. There are multiple clients which are linked with multiple processing nodes. Each XML document is assumed to be of the size of 1 page or 4096 bytes.

**Response time** is the key criterion used to measure the performance of a query evaluation, so let’s look at how useful it is for assessing the effectiveness of the proposed techniques in this thesis. Response time can be decomposed into two main components: Processing Time and Network Time.

### 4.1.1 Processing Time

This is the time spent in tasks such as transferring pages from disk to memory and searching them to prepare the required result at xDB server. We observed in our experiments (Experiment # 1 and Experiment # 2) that some portion of the computing resources remains idle regardless of the number of queries running in parallel on one processing node. CPU usage doesn’t go beyond 90% despite heavy workload (measured using SIGAR API\(^3\)). This means that the processing of results is bound by the I/O capacity because the disk access (like 1Mbps for 7200 rpm disk) is much slower than the processing speed of the contemporary processors. This idea is also supported by observation that the response time of a query is higher when xDB server has to access and process large number of pages. Response time of a query is thus directly proportional to the number of pages accessed in serving that query.

**Experiment # 1**: We run seven trials, starting from the processing of one query to the parallel execution of seven queries. There are 250,000 documents in the database and each query accesses 50,000 documents to get the required result. Q1-Q5 access different sets of documents but Q6, Q7 read the same documents as accessed by Q1. CPU activity increases with increase in the workload but 10% of its capacity remains available even in the presence of heavy workload, due to the process being I/O bound. It means that the response time of a query is bound by the amount of pages accessed and read to satisfy a query rather than the capacity of processor because of the mismatch between the speeds of two components. It means that we need to concentrate on devising such techniques which would reduce the number of data pages fetched for execution of the query, resulting in better response time (results are shown in Figure 7). 1 (on horizontal-axis) indicates that Q1 is being executed, 2 (on horizontal-axis) indicates that Q1 and Q2 are in execution and so on.

![Figure 7: Maximum CPU Usage with 250,000 Data Documents](http://www.hyperic.com/products/sigar)

\(^3\) [http://www.hyperic.com/products/sigar](http://www.hyperic.com/products/sigar)
**Experiment # 2:** This experiment also measures maximum CPU activity when multiple queries are run in parallel. Experimental environment is the same as in Experiment # 1 except that each query accesses 10,000 docs instead of 50,000 docs and there are 50,000 data documents in the database. This experiment also shows that the maximum CPU usage remains in the range of about 90% as shown by Figure 8.

![CPU Usage (Max)](image)

*Figure 8: Maximum CPU Usage in Multiple Parallel Queries with 50,000 documents*

### 4.1.2 Network Time

This is the time taken in sending the query’s result back from processing node to the client. Rate of data transmission is dependent on the network speed, and throughput of network these days has improved considerably. Another important determination of communication time is its direct proportionality to the amount of transmitted data. Large size of query result constitutes large amount of data and thus affecting communication time accordingly. In conclusion, due to high network speed, time consumption in communication of data between two or more nodes is marginal.

### 4.1.3 Latency Estimation Function (LEF)

The breakdown of the processing and network times above suggests that in order to evaluate our data partitioning and indexing techniques accurately, the most relevant and critical factor is number of pages accessed in serving a query. Hence, the cost function of query q is:

\[
C(q) = Ctx + P(q) + W \times \sum_{i=1}^{m} P(Q_i)
\]

where:

- \( Q \) = set of queries (excluding q) submitted at a specific node n at time t
- \( Q_i \) = \( i^{th} \) query in \( Q \)
- \( P(q) \) = number of data pages accessed in serving query q
- \( P(Q_i) \) = number of data pages accessed by \( Q_i \)
- \( m \) = number of queries (excluding q)
- \( W \) = weight indicating the influence of queries \( \in Q \) on ‘q’, (see section 4.1.3.1 for ‘W’)
- \( Ctx \) = time to calculate context of the query for execution
4.1.3.1 **Empirical model of weight \( (W) \) in LEF**

In this section, we empirically quantify weight that determines the influence of parallel queries on one another due to the fact that a node’s computing resources are shared among multiple queries.

In the experiments carried out to compute the value of \( W \), we find that it:

- Remains close to 0.20 when the size of \( Q \) is less than the number of processing cores
- Switches to 0.30-0.34 when the size of \( Q \) is equal to or greater than the number of processing cores

The first range (i.e., about 0.20) exists because each processing core has its own buffer and its computing capacity is exploited by only one query at a time. Computing resources are shared when the size of \( Q \) levels or exceeds the available processing units and thus the latency of a query gets more influenced by other simultaneously running queries. The weight factor \( W \) has been calculated using the formula:

\[
W(q, Q, T, T') = \frac{(T'q_1q_2 - T_q)}{(T_q + T_{q1} + T_{q2} + ... + T_{qm})}
\]

where:

- \( W \) = weight indicating the effect of queries \( \in Q \) on the performance of ‘q’ when all of them are running in parallel
- \( Q \) = set of queries (excluding q) = \{q1, q2, ..., q\_m\}
- \( T \) = time taken by the queries when run individually = \{T\_q, T\_q1, ..., T\_q\_m\}
- \( T' \) = time taken by the queries when they are run in parallel like \( T'q_1 \rightarrow q_2 \), when q, q1, q2 are run and so on.
- \( m \) = total number of queries running in parallel (excluding q)

4.1.3.2 **Testing the empirical model of \( W \)**

The values of weight \( W \) are established by running a number of different experiments (see below). The number of documents of different schemas is varied in each trial to achieve better confidence in the resulting model. Results are consistent and repeatable in the various trials, which shows the accuracy of the empirical given earlier (section 4.1.3.1). Results are shown in Experiment # 3 and 4. These two experiments use the following queries:

Q1: for $x$ in /book let $y := $x[name = "Harry Potter and the Philosopher's stone"]/author return $y$
Q2: for $x$ in /publisher let $y := $x[name = "John Publishing"]/authors return $y/published_author$
Q3: for $x$ in /author let $y := $x[name = "J.K.Rowling"]/book_titles return $y/bookTitle$
Q4: for $x$ in /movie let $y := $x[name = "Speech"]/director return $y$
Q5: for $x$ in /TV let $y := $x[name = "Million Dollar Baby"]/seasons return $y/season$

**Experiment # 3:** Table 1 shows the result of our first experiment regarding the calculation of the weight and its experimental verification. It shows the response time of 5 queries (size of \( Q = 4 \)). Node N has 250,000 documents in its database with 50,000 documents of a given
schema. Each query accesses documents of different schemas. The value of weight $W$, computed as the influence on $q$ of each of $Q$'s queries is shown in Table 1.

<table>
<thead>
<tr>
<th>Response Time (ms)</th>
<th>q</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Weight (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Query Run</td>
<td>630</td>
<td>652</td>
<td>656</td>
<td>658</td>
<td>649</td>
<td></td>
</tr>
<tr>
<td>q, Q1 run in parallel</td>
<td>739</td>
<td>787</td>
<td></td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>q, Q1, Q2 run in parallel</td>
<td>1034</td>
<td>814</td>
<td>1070</td>
<td></td>
<td></td>
<td>34%</td>
</tr>
<tr>
<td>q, Q1, Q2, Q3 run in parallel</td>
<td>1201</td>
<td>1185</td>
<td>1304</td>
<td>1244</td>
<td></td>
<td>31%</td>
</tr>
<tr>
<td>q, Q1, Q2, Q3, Q4 run in parallel</td>
<td>1463</td>
<td>1566</td>
<td>1502</td>
<td>1570</td>
<td>1531</td>
<td>31%</td>
</tr>
</tbody>
</table>

Table 1: Experiment for measuring Weight of Cost Function (250,000 Documents)

Experiment # 4: In another experiment, there are 210,000 documents in the database of Node N1. We are going to observe the behavior and change in the value of $W$ due to the influence of other queries on $q$. This time, the number of documents (accessed by Q5) stored at Node N1 is reduced from 50,000 to 10,000. The response time of $q$ increases when 5 queries ($q$, Q1, Q2, Q3, Q4) are run instead of 4 queries ($q$, Q1, Q2, Q3) but that increment takes place at a lower rate as compared to the one occurring in experiment # 3, because the number of documents accessed by Q4 is reduced by 80%. In other words, the numerator of formula in section 4.1.3.1 will be smaller in this experiment. Likewise, the denominator will also be reduced by same degree because time taken by Q4 (only Q4 by itself) will be smaller as compared to Experiment # 3 when Q4 is run in the single query execution mode. Thus, no difference in weight factor $W$ will occur and the resulting value will remain in the same range (i.e. 0.30 - 0.34 or about 0.20). Table 2 shows the result of this experiment.

<table>
<thead>
<tr>
<th>Response Time (ms)</th>
<th>q</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Weight (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Query Run</td>
<td>624</td>
<td>585</td>
<td>656</td>
<td>640</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>q, Q1 run in parallel</td>
<td>717</td>
<td>655</td>
<td></td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>q, Q1, Q2 run in parallel</td>
<td>1045</td>
<td>781</td>
<td>1077</td>
<td></td>
<td></td>
<td>34%</td>
</tr>
<tr>
<td>q, Q1, Q2, Q3 run in parallel</td>
<td>1201</td>
<td>1185</td>
<td>1217</td>
<td>1201</td>
<td></td>
<td>31%</td>
</tr>
<tr>
<td>q, Q1, Q2, Q3, Q4 run in parallel</td>
<td>1249</td>
<td>1241</td>
<td>1234</td>
<td>1225</td>
<td>297</td>
<td>31%</td>
</tr>
</tbody>
</table>

Table 2: Experiment for measuring Weight of Cost Function (210,000 Documents)

Experiment 3 and 4 show that change in the number of documents accessed by Q4 doesn’t make any difference on the value of weight ‘$w$’ and one of the main purposes of data partitioning is to influence the number of documents that constitute the search space of a certain query. Thus our experiment results support the claim that $W$ conforms to a certain empirical model given a certain scheme of data partitioning.

4.1.3.3 Cost of Computing Query Context (Ctx)
The context computational cost is dependent on the number of various path expressions in the given query and the number of schemas in the system. But in practice, this operation is
so small (after optimization) that it takes 60-70 ms in all of our experiments. Optimization here means that the full paths of every schema are built at the time of its allocation and they are stored as its metadata in the Data Dictionary (see Definition 4.1 below) so that pre-built full paths can be retrieved quickly to identify the context. With no optimization it takes 250-300 ms for calculating the context of the query. The following graph shows the cost of identifying the context (with optimization) for the queries given above.

![Figure 9: Time to Calculate Data Location](image)

The above graph shows that the time taken in identifying the location of nodes and libraries is nominal and thus using it at compile time saves a lot of computing resources of different nodes which can be used to execute other queries in an efficient manner (at the nodes that contain the relevant or target data).

### 4.1.4 Difference in Response Time between Similar and Dissimilar Queries

**Similar queries** are defined as those that may ask for different kind of information but require the same set of documents, whereas **dissimilar queries** are those that target different sets of documents.

It was observed in our experiments that multiple *similar* queries are quicker in response (irrespective of the result size) as compared to multiple *dissimilar* queries (see Figure 10), because in the former case, data pages are not replaced in the working memory. It can also be noted in Figure 10 that queries running in parallel slow down one another even if they access the same sets of documents.

### 4.1.5 Characterization of Workload

The system is expected to receive diverse sets of queries. Diversity here means that the number of queries accessing different schemas is fairly even. For example, if schema S1 contains /a/b and schema S2 contains /c/d, then the system is expected to receive the queries containing /a/b or /c/d evenly. Please note that there is nothing special we are assuming about the form or structure of the queries. All queries that refer to XPaths would benefit from our data partitioning scheme.
4.2 Data Partitioning

As described earlier, our main goal is to reduce the number of pages accessed by xDB in serving a query in order to improve response time. Our strategy is to use a two phase approach to achieve this target: In the first phase, which is about the data partitioning, we place documents of similar schema on the same processing node where a separate library is allocated for documents of each schema. The second phase uses value indexes which speeds up query evaluation (more details in section 4.3).

The reason we would like to have a schema-node affinity model whereby we place similar documents on the same node (the so-called schema clustering based partitioning scheme) is two-fold:

- To minimize inter-query interference: Note that the higher the difference between the nature of the documents targeted by the queries, the higher the interference (see section 4.1.4)
- For better fault tolerance: Minimize the impact of network partitions or node failures on the servicing of the queries: if a node serves only a limited set of schemas, it is required to be available by only those queries that target documents of the said schemas

Some important concepts are defined in the following (Definitions 4.1, 4.2, 4.5, 4.6) that would help in understanding and designing the strategies of data partitioning, context identification and auto-indexing. The concepts (defined in Definitions 4.3, 4.4) are used as the evaluation parameters for describing the completeness and correctness of our context identification process.

**Definition 4.1:** Data Dictionary (DD) is a database that contains a map of document allocation. An xDB library is assigned a set of structurally similar schemas, so the
metadata of a schema contains a location ID of the form (Node ID, Library ID), where Node ID identifies the processing node where the schema is hosted and Library ID identifies xDB library containing XML data documents of that schema.

**Definition 4.2**: Perfect Match: Schema S1 and S2 are considered a perfect match if they are identical, i.e., describe data about the same kind of objects (in terms of elements and sub-elements). Similarity coefficient (Definition 5.2) in this case will be 1 on a scale from 0 to 1. Any difference in the structural elements or in their ordering would decrease the similarity score or perfectness of the match between S1 and S2.

**Definition 4.3**: False Positive: If the context identification process indicates non-emptiness of the result of query Q at the preprocessing stage but the query returns an empty result set from node N against the documents of schema S.

**Definition 4.4**: False Negative: The context identification process declares empty result from node N containing documents of schema S but N contains data targeted by query Q.

**Definition 4.5**: Sub-Exp(Pi, Pj): If Pi is a prefix of Pj where Pi and Pj are XPaths, then Sub-Exp(Pi, Pj) is true; otherwise it is false (non-Sub-Exp). For example, Sub-Exp(/a/b, /a/b) is true and Sub-Exp (/a/b, /b/a/b) is false.

**Definition 4.6**: FP(s) := \{FP; FP is a unique full path generated from schema s, starting with RTs and ending with NTi\}

where:

- Ts is an XML tree representing Schema s where each node (NT) of Ts is an XML element, NT contains zero or more attribute(s) and an attribute doesn’t have child node in Ts.
- NTi is a leaf node in Ts (i.e., an xml element)
- RTs is the root node of Ts

**Example**: FP(s) of schema in Figure 11 is:

 )))author/name/first_name, /author/name/family_name, /author/education)))

![Figure 11: XML Schema](image-url)
4.2.1 Query Context Identification

In this section, we describe the strategy of the process of query context identification (results regarding its correctness and completeness are presented in section 8.3.1). In the context identification step, we consult Data Dictionary (Definition 4.1, above) at the dispatching client to determine at compile time the set of nodes that would have data relevant to the query so that query can be sent only to those nodes, not to irrelevant nodes. Context identification is mainly dependent on two components: the satisfiability of the path expressions (excluding data values) present in the query by an XML schema, and the mapping between the schema and the location of corresponding data. Satisfiability indicates whether or not there are any documents in the database that would return a nonempty result.

In determining satisfiability, we consider elements and attributes, and two hierarchical relationships between elements: (1) child or ‘/’ and (2) self-or-descendants or ‘//’. Handling of other hierarchical relationships in XPath is left as a part of the future work. For checking the satisfiability of XPath in predicates, we trim/ignore constant values from predicates and where-expressions because data documents are not hosted by the client. For example, satisfiability of “/name/family_name” (instead of “/name[family_name = ‘Michael’]”) in XQuery: “for $x$ in /name[family_name = ‘Michael’] return $x$” can be checked against the schema, which amounts to static or compile time analysis. There are two reasons for not having data documents at the client: 1) There could exist millions of data documents which would cost a hefty amount of space and it also defeats the purpose of storing data in a distributed fashion in the first place and 2) Checking satisfiability on data documents is equivalent to the evaluation of query and the required computing will go wasted if its result set comes out empty and we don’t detect emptiness at compile-time. So instead of determining satisfiability of path expressions in data documents, we instead use non-recursive schema documents that correspond to those data documents. As we mentioned earlier, the drawback of bypassing literal values is that path expression with data values which returns a nonempty result set at compile time could still return an empty result set at run-time (a false positive), which hurts the optimality of performance; nevertheless, the opposite situation is not possible (i.e., a false negative), a quality that is critical for correctness. Detailed algorithm of context identification is included in section 6.2.2.1.

In the context identification of XQuery, we consider the FLWOR expression, and the Conditional (if-then-else clause) and Logical (Union operation) expressions within the FLWOR expression. Schema s is defined by an XML tree Ts and FP(s) defines the set of full paths from Schema s (Definition 4.6, above); Satisfiability of the query Q is determined by checking the satisfiability of Q’s path expressions in FP. Q’s syntax tree is defined by a set of nodes, i.e. AST(Q) = {nd₁, nd₂, ..., ndₘ}. AST(Q) is a simplification of the real AST as defined by xDB query parser and is defined by five properties, i.e., Np(ndᵢ) = <tᵢ, xᵢ, pᵢ, ptᵢ, chdᵢ>, where:

- tᵢ is the type of the query clause (for XQuery 1.0 we consider the values of {FLWOR | for | let | where | order | return | if | then | else | union})
- xᵢ is a variable or ε (will not be ε or empty if tᵢ = {for | let} ), e.g., ‘$x$’ in “let $x := /a/b$”
- pᵢ is an XPath, e.g., ‘/a/b’
- ptᵢ is the parent node of ndᵢ. If ndᵢ is the root node of AST(Q), then ptᵢ will be ε
- chdᵢ is a set of nodes of AST(Q) which are children of ndᵢ.
• nd_i is the parent node of zero or more nodes of \{nd_{i+1}, ..., nd_m\} (parent is indicated by the symbol $\prec$)

The following gives definitions of a few operations used later in the analysis:

• $S(p, s)$ is a Boolean operation which returns true if and only if $\text{Sub-Exp}(p, p_i) = \text{true}$, where $p_i$ is any XPath satisfiable in $s$ (Definition 4.5, above)
• $Ve(x_i, p_k)$ is a Boolean operation which returns true if and only if $\text{Sub-Exp}(p_i, p_k) = \text{true}$ for some $p_i$ where $p_i$ and $p_k$ are XPaths such that: $p_k$ isn’t given by $Np(nd_i)$, and $p_i, x_i$ are given by $Np(nd_i)$. For example, if $x$ is ‘$x’$ and $x/a/b$ is ‘$p_k’$ in ‘let $y := x/a/b’$, then $Ve(x_i, p_k)$ is true.
• Descendant $(nd_i)$ returns descendent nodes of $nd_i$.

**Example:** Here we illustrate AST($Q$) and five properties of its nodes using an example query, $Q := \text{“for } x \text{ in } /a/b \text{ return } x/c”$. AST($Q$) is shown in tree representation in Figure 12.

In Figure 12, $nd_1$ has two children i.e. $nd_2$ and $nd_3$, presented with its properties in the associated table. This notion of parenthood is represented as $nd_1 \prec (nd_2, nd_3)$ in text.

**4.2.1.1 Optimization in Determining XQuery Satisfiability**
While identifying the location of data for a query, we first determine its satisfiability by searching path expressions referenced in the query against the schema. It is possible that the schema satisfies some of the path expressions in the query but their satisfiability does not necessarily signify the satisfiability of the query itself. In this section, we describe
which constructs of XQuery comply with this behavior and how we can improve the accuracy of the context identification.

The pre-processing step of identifying the query context can be optimized for certain constructs of XQuery (explained later) where unsatisfiability of one path expression (p₁) signifies the unsatisfiability of certain other path expressions (p₂, p₃, ..., pₙ) in the query.

**FLWOR Expression**

Schema s satisfies p (an XPath) when all elements and attributes in p are reachable from the root of Ts (defined in Definition 4.6) in the right order. Please note we use schema (not data documents) to determine the satisfiability of the XPaths of a query for determining the location of required data, so it is possible that our context identification process produces false positives (results and detailed analysis are presented in Chapter 8). The computation of satisfiability can be optimized to varying degrees for the different constructs of FLWOR expression in the query. There are four main scenarios in this regard:

1. **FLWOR Expression (Scenario 1):** If tᵢ is of type ‘for’ and pᵢ is unsatisfiable in s where tᵢ, pᵢ are given by Np(ndᵢ), then there is no need to check the satisfiability of ∃p ∈ Np(Descendant(ndᵢ)) because emptiness or non-emptiness of p’s answers will be irrelevant to the final result of Q. We will remain in scenario 1 or will enter into scenario 3 for ndᵢ₊₁ (depending upon the value of tᵢ₊₁) where ndᵢ₊₁ is child of ndᵢ, if pᵢ is satisfiable. If pᵢ is unsatisfiable, then we have scenario 2.

2. **FLWOR Expression (Scenario 2):** It is still needed to check the satisfiability of p where p is given by Np(nd) and nd∈(AST(Q)) is parent-or-its-descendant or sibling-or-its-descendant of ndᵢ. Then, there are two sub-cases in this scenario:

   - **Case 2a:** If Sub-Exp(pᵢ, p) is true, then emptiness of p can be determined without checking its satisfiability.
   - **Case 2b:** If Sub-Exp(pᵢ, p) is false, then we have scenario 1 or 3 depending on the type of unexplored nodes of AST(Q).

3. **FLWOR Expression (Scenario 3):** If tᵢ = (let | where | return) and pᵢ is unsatisfiable in s where tᵢ, pᵢ are given by Np(ndᵢ), then there is no need to check the satisfiability of p∈Np(Descendant(ndᵢ)) if Sub-Exp(pᵢ, p) is true; satisfiability of p will be checked otherwise. If pᵢ is satisfiable, then we will remain in scenario 3 or will enter into scenario 1 for ndᵢ₊₁ (depending on the value of tᵢ₊₁) where ndᵢ₊₁ is a child of ndᵢ.

   After exploring all nodes of AST(Q), if S(p, s) is empty where ∀s∈(Sc | Sc is the set of schemas stored on Nᵢ) and (∀p∈Np(AST(Q)) : p is of type ‘return’), then result of Q is empty.

4. **FLWOR Expression (Scenario 4):** If p contains a predicate in which a ‘fn:not’ operator is applied on p’ which is represented by fn:not(p’) where fn:not is a unary operator (has higher precedence than binary operator like ‘|’ or union) and fn:not(p’) is contained in the predicate which is represented by p[fn:not(p’)], then satisfiability of p’ will imply the unsatisfiability of p[fn:not(p’)] and unsatisfiability of p’ will imply the satisfiability of p[fn:not(p’)].
Conditional Expressions (the if clause)
If \( p = p' \land p'' \) where \( p, p', p'' \) are XPaths in ‘if’, ‘then’ and ‘else’ clauses respectively, then there are three simple propositions:

- \( P1 := p' \) is non-empty
- \( P2 := p'' \) is non-empty
- \( P3 := p \) is nonempty

If \( P1 \lor P2 \) is false, then determination of \( P3 \) is irrelevant to the emptiness/non-emptiness of the result set of XQuery \( Q \) or XPath \( (p_{i-1}) \) where \( p_{i-1} \) is given by \( Np(nd_{i-1}) \), \( nd_{i-1} \) is a parent node of the node \( nd_i \), and \( p', p'' \) and \( p \) could be the result of FLWOR expression. \( p_{i-1} \) is XPath with \( t_i = \{FLWOR | for | let | where | return\} \) and thus \( p_{i-1} \) will be treated as explained in four different cases of **FLWOR expression**. In conclusion, here we can optimize our context identification process by not checking satisfiability of \( p \)'s answer unless \( P1 \lor P2 \) is true.

Logical Expression (Union operation)
Logical path expression: \( p = p' \mid p'' \), where \( p', p'' \) are operands of union operation and \( p \) is the result of union operation. Using the above three propositions, we have the conclusion that \( P3 \) is false if \( P1 \land P2 \) is false and it is true otherwise.

4.2.1.2 Importance/Benefit of Context Identification
If a query is not passed through context identification process, then it will be sent to every node in the system. Some of the recipient nodes will also be busy in processing such queries for which they have no data and those nodes will return empty result for them. Thus, empty result queries will slow down the non-empty result queries because all of them will share CPU resources as the data pages are loaded for empty result queries as well. Thus context identification plays important role in improving the overall performance of a processing node.

Note that some empty-result queries may still pass through our filtering system because of the absence of data values in the Data Dictionary (DD). Those queries can be classified as ones with data values in ‘predicates’ or ‘where-expression’, contained for the purpose of selection of objects on certain data criteria. For example, ‘for $x$ in /movie[name = ‘The Dark Knight’]/director return $x$’ will pass through context identification process, that will be sent to node ‘n’ which contains the ‘movie’ database but assuming none of the documents at node ‘n’ contain the value ‘The Dark Knight’ on element ‘name’, the result of the query would be returned from ‘n’ as empty.

4.2.2 Schema Similarity
Placement of documents requires the calculation of similarity coefficient between schema \( s' \) (representing the set of new documents \( d' \)) and schema \( s \) (representing the set of existing documents \( d \)). If similarity co-efficient between \( s \) and \( s' \) is 1 (Definition 2, above), then \( d' \) will be placed with \( d \) on the same node and within the same library. We will see later what can be done if similarity coefficient is less than 1.

Not all of the documents have schema but it is required for the data partitioning and query context identification (for locating data) in our approaches, therefore schema is needed to be extracted or reverse-engineered from the XML documents in that case. There exist ways of
doing that and a third party application could be used for this purpose. We have used an open-source tool Trang\(^4\) for building XML schema definition from XML documents.

Earlier we mentioned that our system stores documents of similar schemas to the same node; these documents may have semantically similar but not entirely identical structure, e.g., due to various optional elements in the underlying schema. They may also have syntactic differences in the structure, such as a different order or cardinality of certain elements. The process of schema inference could therefore produce multiple different schemas even if the source documents have the same underlying or originating schema; in other words, the process of schema inference is inherently lossy. In order to handle this situation and allocate such documents on the same node nevertheless, we calculate structural similarity between new documents and existing documents. Suppose that some new documents arrive into the system such that they contain the same elements as contained by the documents already in the system but in different order or with different cardinality. The new group of documents would produce a slightly different schema compared to the one for the first group. Since both groups describe same type of information, a single query can get its desired data from both groups. If a schema similarity measure is not incorporated into the system, then documents and schema of the new group may go to a node different from the one where the first group lives, while we would like them to be published to the same node.

The path indicates the position of an element in addition to its tag name and these two pieces of information give important hierarchical and meta-data information. Thus, the basic building block of our data partitioning is full paths (see Definition 4.6, above) which protect the loss of structural and semantic information. When a new document or a set of new documents arrives, full paths of their schema are built, which are then used to construct path expressions. For example, ‘/a//c’ is a path expression generated from full path: ‘/a/b/c/d’. The reason of using path expressions is their use in the queries and since we want to send the queries looking for similar type of information to the same node because of reasons mentioned earlier (section 4.2), therefore the similarity coefficient between the new and the existing schemas is (defined to be) directly proportional to the number of common XPath expressions between them. The notion of common XPath is that it should be common to both schemas being compared. If new and existing schemas are found similar (similarity coefficient ≥ 0.5, see section 5.2.2 for details), the new schema is placed in the same library (representing a processing node) in DD where the existing similar schema is already placed and new documents along with their schema are sent to the processing node(s) where documents of similar existing schema are stored. Details of this algorithm can be found in Chapter 5.

Path expressions with wildcards at every position (e.g., ‘/*/*’) are excluded from the schema similarity computation, because of two reasons: Similarity coefficient between two totally different schemas will always be greater than 0 if we consider wildcards at every position as well. If two schemas have some elements in common, then similarity score will be higher due to score added by wildcards. The choice of node for storing a set of documents is decided after comparing their schema with all of the schemas already existing in the system, meaning selection is based on a relative score. Thus inclusion of wild cards will not make any difference in this relative score. Secondly, the occurrence of wildcards at every position

\(^4\) http://www.thaiopensource.com/relaxng/trang.html
of XPath is rarely found in practice, as it usually doesn’t help in getting any meaningful information.

The similarity computation described above involves enumerating the path expressions (subset of full paths) that are common in the XML schemas being compared. To do this, path expressions are generated from full paths of a schema. Suppose there are two schemas s1 and s2 where new documents belong to s2. If the two schemas are a Perfect Match, then we assign documents of s1 and s2 to the same library on the same node. If the similarity coefficient is less than 1 but still greater than a certain threshold (0.5 in our experiments), then we allocate new documents on the same node but to a different

**Example:** Two schemas s1 and s2 (shown in Figure 13 and Figure 14 respectively) should be placed together on the same node, under the same library because they are a Perfect Match (Definition 4.2, above). In the two schemas s1 (Figure 13) and s2 (Figure 14), the only difference is the presence of attribute “bookid” and unbounded occurrence of element “author” in schema s2 whereas s1 doesn’t have any attribute node and doesn’t specify any constraint on the cardinality of element “author”. Since in this thesis we consider common path expressions involving only elements, and child and descendent relationship between them, therefore similarity coefficient between s1 and s2 comes out to be 1. Therefore documents of both schemas will be allocated to the same node N_i and the same library “book”. Future work could take some other information like attributes into account in the similarity calculations because currently, the search space of queries containing search on attributes (“bookid” in the current example) will be the document set stored in the library “book”, which is not optimal as some of the documents (at least those belonging to schema s1) will return an empty set for queries involving the said attributes.

![Figure 13: Schema s1 about Book](image)

![Figure 14: Schema s2 about Book](image)
4.3 Auto-Indexing

As we described in the previous section (4.2.1), queries which are detected with non-empty result at compile time can be found as empty at runtime but detection of emptiness of the result set at runtime is an expensive operation because all relevant documents (relevancy is determined in the context identification process) are required to be searched, which increases response time of other parallel queries (should be avoided specially when their result is going to be non-empty). The operation cost of detection of emptiness of query result at runtime can be minimized to large extent by utilizing indexes. These would be value indexes (see section 3.2), as provided by xDB, since values or document data is the part missing at compile-time.

Apart from the detection of emptiness of query’s result at runtime, there is another benefit of using element-value index. As we mentioned in current and previous chapters, the strategy of data partitioning using schema similarity and context identification optimizes the number of pages required to satisfy a query; this can also be thought of as a meta-index based approach. Value indexes can further reduce the cost (as measured in terms of page access or response time). The challenge with these indexes is to decide upon what elements to index to improve query performance most substantially. Building indexes on such elements, attributes or paths that are not important from the querying point of view would be a waste of storage and processing resources (note that the index update overhead is significant). One way of determining the importance is to analyze the access patterns of past queries, specifically the data pages accessed. The information so gathered is cumulative in that there’s very little information initially when the system has not seen very many queries, but the accuracy of the model grows as the system experiences more queries.

To recap, the overall approach proposed in this thesis can be seen as consisting of two phases: (a) data partitioning (described earlier), which works off of schemas and doesn’t depend on the historical data about queries; and (b) auto-indexing (described next), which gets more effective as the query patterns are better understood by the system.

4.3.1 Element Value Index

An element value index is built using element and its values. Building indexes on all elements would not be very effective for optimization due to indexing overheads. Our context identification strategy helps in incorporating information about schema differences into indexing of the comprising elements. That way, we can filter out those elements from indexing that are not important from the querying point of view. Data partitioning provides an enabling environment for constructing indexes using filtered elements and only on those documents that are accessed most frequently. As we know that documents are divided with respect to schemas and queries are also expected to have some level of schema affinity, our data partitioning, indexing and querying work synergistically.

4.3.2 Steps of Auto-Indexing Process

Three main steps are involved in building indexes on element values:
4.3.2.1 Collection of Statistics

Statistics are collected from the queries submitted to the system for processing. Since an element needs to be identified to create an element value index, we go through a selection process in deciding which of the elements serve best in speeding up the queries expected in a system. Since indexing is value-based, firstly we extract path expressions from the predicates and ‘where’ expression. Secondly, extracted path expressions are parsed to differentiate between leaf elements and non-leaf elements in a query. Normally last element in the path expression in a predicate is assumed to be the leaf element of XML tree but we ensure it by searching it against every leaf node of every relevant schema. Relevancy of schema is determined in context identification process by checking satisfiability of non-predicate path expressions in the schemas stored at DD. Each leaf element is characterized by:

**Definition 4.7**: For the purposes of value based indexing, a leaf element is represented as 

\((LE) := <\text{Name}, S, N, A, C>\)

where:

- **Name** = Name of the leaf element
- **S** = schema which contains the leaf element
- **N** = Node which contains S
- **A** = access frequency of the leaf element
- **C** = flag (true or false) indicating whether a value index is already built on the said leaf element

As we mentioned earlier, leaf elements can be common in different schemas on name similarity basis but above definition will label them as different leaf elements because of the different context (different schema and node). A snapshot of query statistics in Figure 15 shows different leaf elements.

![Figure 15: Query Statistics](image)
As we can see in Figure 15, there exist two elements with same name ‘name’ but have difference in their access frequency and number of pages accessed by past queries referencing the element (either in predicate or in ‘where’ expression’ or both). They are also labeled under different nodes and libraries. The first element (on which index is built already) is part of the documents stored in library ‘book’ on node ‘Node1’ while the second leaf element ‘name’ is an element of the documents stored in a different library, i.e., ‘publisher’ on ‘Node1’. This differentiation is desirable because there could be many different leaf elements in a typical database and not all of them are used in the queries with data selectivity (see algorithmic details in chapter 6). Our context identification mechanism enables this differentiation between the elements.

4.3.2.2 Statistics Analysis
In the second step of the process, we analyze information (such as presented in Figure 15) to determine which of the leaf elements (according to Definition 4.7) have been referenced by the queries that accessed highest number of pages in the past. A list of leaf elements is generated, ranked by the number of pages accessed and the top M% of them are selected for index creation (M could be customizable by the admin). These indexes can now be utilized to dramatically reduce the number of pages accessed during the processing of a typical query.

4.4 Query Set
The optimization strategies presented in this thesis were tested and empirically analyzed using various data retrieval queries. In order to distinguish between different scenarios and to highlight where our approach provides biggest gains in performance, we used queries of different forms to evaluate and to measure the performance difference between our technique and alternative approaches. Currently there is no well-known standard or benchmark workload defined in the literature or by the industry against which we can measure different techniques of publishing and retrieving data. The sets of queries and data used by this thesis for the said purpose could potentially be adopted as the first such standard for future work.

In general, joins and aggregate functions are not considered in these queries (left as possible future work). Queries include XPath/XQuery with and without selective conditions (i.e., predicates and the ‘where’ clause), along-with a variation in the result size of each query.

xDB supports current standards of XPath and XQuery. In the phase of context identification (see algorithm in section 6.2), we have addressed two relationship operators (/, //). There are three types of wildcards in XQuery: *, node0, and @* [31]. We are considering the first two of these wildcards, i.e., ‘*’ and node0, while extension @* is left as a part of future work.

4.4.1 Syntax of Queries Used for Testing
The following two sections describe the various XPath and FLWOR expressions that were used to test our data partitioning approach and compare it with certain naive alternatives.
4.4.1.1 Category 1 (XPath expression)

Type 1 (XPath expression with/without predicate):

Query ::= ( Op (E | '@'A) )+ ( [ (E | @A)+ Co-op (a-zA-Z0-9) | (Op (E | @A)+ ] )? (Op (E | '@'A) )* 

where:

- Op = {/, //
- E = Elements’ tags of leaf or non-leaf nodes of an XML document. E is categorized by [a-zA-Z0-9_]*
- A = Attributes formed using words and/or letters
- Co-op = (= | < | > | ≤ | ≥ | eq | le | ge | lt | gt)

Example: Q := /books//book[name = “Jones”]

Example: Q := /books//@name

Type 2 (XPath expression containing fn:not with/without predicate):

Query ::= fn:not ( (Op (E | '@'A) )+ ( [ (E | @A)+ Co-op (a-zA-Z0-9) | (Op (E | @A)+ ] )? (Op (E | '@'A) )* ) ) 

Example: Q := fn:not(/note)

4.4.2.2 Category 2 (FLWOR expression)

Type 3: Query with/without a predicate, without ‘where’ expression and with mandatory ‘for’ and ‘return’ clauses

Query ::= ‘for’ ‘$’(a-zA-Z0-9)+ in (Op (E | '@A')+ ( [ (E | @A)+ Co-op (a-zA-Z0-9) | (Op (E | @A)+ ] )? (Op (E | '@A) )* ‘return’ (‘<open_tag’>result’<’closing_tag’>’) | result )

where:

- open_tag = (a-zA-Z0-9)+
- closing_tag = (a-zA-Z0-9)+
- open_tag and closing_tag must be same
- result = either node (including its child nodes) or scalar constant

Example: Q := for $x in /a/b[c = ‘teacher’] return $x

Type 4: Query with a ‘where’ expression and with ‘for’ and ‘return’ clauses

Query ::= ‘for’ ‘$’(a-zA-Z0-9)+ in (Op (E | '@A')+ ( [ (E | @A)+ Co-op (a-zA-Z0-9) | (Op (E | @A)+ ] )? (Op (E | '@A) )* ‘where’ (‘$’(a-zA-Z0-9)+) | (Op (E | @A)+ Co-op (a-zA-Z0-9) | (Op (E | '@A) )+ ‘return’ (‘<context’>result’<’context’>’) | result )
**Example:** \( Q = \text{for } x \text{ in } /a/b[c = \text{teacher}] \text{ where } x/d = \text{Peter} \text{ return } x \)

**Type 5:** Query with ‘for’, ‘where’ and ‘return’ clauses and containing fn:not in a predicate.

Query ::= ‘for’ ‘$’(a-zA-Z0-9)+ in ( Op (E | ‘@’A) )+ ( fn:not( (Op(E | ‘@’A))+ (Co-op ( (a-zA-Z0-9) | (Op(E | ‘@’A))+))+ )? ( Op (E | ‘@’A) )* ‘where’ ( (”$’(a-zA-Z0-9)+) | (Op(E | ‘@’A))* )+ Co-op ( (a-zA-Z0-9) | (Op(E | ‘@’A))+ ) ‘return’ ( (’<context’>’result’</context’>) | result )

**Example:** \( Q := \text{for } x \text{ in } /a/b[\text{fn:not}(.c = \text{‘teacher’})] \text{ where } x/d = \text{‘Peter’} \text{ return } x \)

### 4.5 Data Partitioning Scheme and its Limitations

There are certain use cases where queries would be more efficient if the documents of each schema are evenly divided among the nodes (\( N_s \)) rather than on the basis of their schema because some of the nodes could be idle at any given time. They can be utilized to provide results quickly because each node then has to process smaller number of data pages. At off-peak times, this gain is less likely to occur in a schema cluster-based partitioning system (as proposed in this thesis). Figure A and B show this difference pictorially, where there are two processing nodes (labeled \( N_{s1} \) and \( N_{s2} \)) and ‘Q1’ is a query.

In schema cluster-based partitioning (Figure B), node \( N_{s2} \) cannot contribute to the evaluation of Q1 despite being idle because it doesn’t have data for Q1 and \( N_{s1} \) has to process Q1 on its own. In comparison, both \( N_{s1} \) and \( N_{s2} \) potentially contribute toward the evaluation of Q1 in a system where documents are randomly distributed (Figure A). Note however that in heavily loaded systems, which is exactly where performance matters most, running all queries on all nodes means longer query backlogs and therefore, higher query latencies.
Aside from the expected latency differences at high query load, an even more important difference between the two approaches is that of network partition tolerance or node failure. In a distributed system where the documents are placed randomly over all the nodes without any regard to the structural makeup or schema of those documents, a single partition in the network or a single node failure would affect the correctness or completeness of the response of nearly all queries. On the other hand, in a schema based partitioning system, only a small subset of queries that actually target the data residing on the failed node(s) would fail or be affected.

4.6 Summary
This chapter covered three areas: Firstly, a description of the main objective of our thesis, i.e., ways to optimize the performance of XQuery and XPath. Secondly, the concepts behind and an overview of the proposed strategies of data partitioning (schema based or schema-cluster based partitioning), identification of query context, i.e., location of node(s) where query relevant data resides, using XPath satisfiability and the method of element-value indexing. It was explained how these three techniques can contribute toward the goal of improving response time of a query. We will present algorithms for these strategies in Chapters 5 and 6, and will interpret those results in Chapter 8 to determine the usefulness of our techniques using a set of queries whose syntax or form was defined in this chapter. The performance numbers for the proposed algorithms are compared with some baseline strategies that will be introduced in Chapter 7.

Lastly, an evaluation criteria or latency estimation function (response time in terms of data pages) and a weight measure (the influence factor that parallel queries exert on one another) were defined in this chapter which help in the interpretation and analysis of results in order to gauge the effectiveness of the strategies proposed in this thesis.
Chapter 5

Implementation of Data Partitioning

In the previous chapters, we were introduced with the concepts that form the foundation of the strategies of our data partitioning and techniques of identification of data location (context identification). In addition, we also gave an overview of the framework of data partitioning, context identification and indexing techniques. In this chapter, we go one step further and describe the details of implementation of data partitioning i.e., algorithms of how we are going to partition the data into different portions so that they can be allocated to different processing nodes and separate libraries.

If two schemas S1 and S2 share paths and a large percentage of path expressions are found common in them, then S1 and S2 are highly similar and thus they (and their data documents) should be placed on the same node as described in Chapter 4. As a user asking a query in usual case, he will ask same type of information and therefore its constructed query will contain such path expressions which will be present on a single node or at least reduced set of nodes. In this way, other users asking different kind of information will be able to get computing resources exclusively from different set of nodes. This will greatly improve user responsiveness.

Our approach also supports storage of XML documents (validated by same schema) in multiple batches. The only requirement is to construct XML schema from documents before storing process starts. It means that new set of documents (validated by existing schema) is allocated to the same node every time.

5.1 Architecture Overview

Our data partitioning design works for the objective of reducing the processing of data pages in satisfying a query (details in section 4.1). Figure 16 shows pictorial view of data partitioning architecture.

In Figure 16, “Docs Allocator” has three modules which communicates with a UI, three modules and three xDB servers for each processing node, and a xDB server for DD at client. The specifics of these components are:

**Path Builder:** Its function is to build full paths (defined in Definition 4.6) from a set of schemas which include new schema and existing schemas.

**DD Lookup Module:** It retrieves existing schemas from DD and provides them to Path Builder.
Docs Allocator: It is the main module of our data partitioning method whose job is to calculate similarity coefficient between existing set of schemas and new schema with the help of other modules as shown in Figure 16. Algorithms of Docs Allocator are described in section 5.2.

DD Storage Module: It stores new schema into a library at DD as suggested by Docs Allocator.

![Data Partitioning Model](image)

**Figure 16: Data Partitioning Model**

### 5.2 Partitioning Algorithm

We describe our data partitioning algorithm in pseudo-code format. Algorithm 1 (fullPathBuilder) is used to build full paths (defined in Definition 4.6) of two schemas (si is a new schema to be allocated and sj is an already allocated schema) and then they are compared to decide whether we should place si and sj on the same node or not. Data structures stack \( S_e = [S_e[1], S_e[2], ..., S_e[n]] \) and sibling queue \( SB[n] = \{S_{b1}, S_{b2}, ..., S_{bn}\} \) for
∀Se[n] ∈ Se, are used in algorithms fullPathBuilder to build full paths. Se and SBN for each Se[n] are defined in Definition 5.3. After building two sets of full paths, each from si and sj, they will be used in Algorithm 2a – 2d to determine how similar they are. All of these algorithms (1 and 2a-2d) are repeated for each of the existing schemas in the system.

The following three definitions help in understanding the algorithm of determining schema similarity:

**Definition 5.1 (Common Path Expression):** A common path expression is an expression matched in P1 and P2, where P1 and P2 are two full paths from schemas being compared.

**Definition 5.2:** The equation representing similarity coefficient (using Jaccard coefficient) between Schema sx and Schema sy is:

\[
Ce(s_x, s_y) = \frac{|GPE(s_x) \cap GPE(s_y)|}{|GPE(s_x) \cup GPE(s_y)|}
\]

where:

- GPE(x) = set of path expressions generated by the elements of FP(x), i.e. GPE(x) = GPE(FP1) ∪ ... ∪ GPE(FPn) if FP(x) = {FP1, ..., FPn}. FP(x) is defined in Definition 4.6
- |X| denotes the number of elements in the set X
- |GPE(sx) ∩ GPE(sy)| = |GPE(sx)| + |GPE(sy)| - |GPE(sx) ∪ GPE(sy)|

In Definition 5.2, the construction of set \{GPE(sx) ∩ GPE(sy)\} can be optimized if we pick the smaller full path in the two full paths FP1 and FP2 from schema sx and sy respectively. The path expressions will be generated from FP1 and FP2 and all those generated path expressions will have the maximum length equal to the length of that smaller full path. In this way, we can avoid generating and comparing such path expressions which cannot be satisfied in both schemas i.e. sx and sy. For example, FP1 in schema sx is “/a/b” and FP2 in schema sy is “/b/c/d”. While computing the set \{GPE(sx) ∩ GPE(sy)\}, we will generate path expressions from FP1 and FP2 having length from 1 to 2 (2 is the length of smaller full path i.e. FP1). Though it is possible to generate the path expressions of length 3 from FP2 like /b//c/d but it is not satisfiable in both schemas sx and sy (not satisfiable in sx) and therefore it cannot be a member of \{GPE(sx) ∩ GPE(sy)\}. Algorithm 2c (section 5.2.4) applies this technique to get the desired optimization.

**Definition 5.3 (Element Stack):** Element node in the stack structure at i\textsuperscript{th} index is denoted by Se[i]: = \langle t_v, LE, SBi, C \rangle and the complete stack is denoted by Se: = [ Se[1], Se[2], ..., Se[n] ]

where:

- t_v = text value of Se[i]
- range of Leaf element (LE) = \{0,1\}, 0 => Se[i] is a leaf element node in the schema; 1=>otherwise (default)
- SBi is the set of sibling nodes of element node referenced by Se[i]
- C is a Boolean flag, 0 => if sibling nodes of Se[i] haven’t been added to SBi, 1 => otherwise

Implementation of Data Partitioning 41
5.2.1 Algorithm 1 (FullPathBuilder)

This algorithm builds full paths (defined in chapter 4, above) from a schema Sc.

**Input:** A schema denoted by Sc.

**Output:** A set of full paths FP = {FP₁, FP₂, ..., FPₘ} as generated by FP(Sc) (Definition 4.6, above).

**begin**

FP := ∅
m := 0  // m indicates the total number of full paths i.e. size of FP
Sₑ := []  // n indicates the number of elements in stack Sₑ
n := 0

Push root element of Sc on Sₑ[n] and increase n by 1

while (Sₑ ≠ ∅) {
add all sibling nodes of Sₑ[n] to SBⁿ if they haven't been added
if (LE of Sₑ[n] = 1) {  // if Sₑ[n] contains non-leaf node (element)
  add first child (in document order) of Sₑ[n] to Sₑ[n+1] and increase n by 1
  if (LE of Sₑ[n] = 1)
    jump to while loop  // to keep adding child of top element of Sₑ
} else {
  if Sₑ[n] contains leaf element, form full path from contents of Sₑ
    FPᵐ := “”  // FPᵐ is a full path at mᵗʰ index of FP
    for (j := 1 to n)
      FPᵐ := FPᵐ ◊ '/' ◊ tᵥ of Sₑ[j]  // ◊ => concatenation
    FP := FP ∪ FPᵐ
    m := m + 1
} end-if

if (Sₑ[n] has no sibling left)
  pop Sₑ[n] and decrease n by 1
else replace Sₑ[n] with its sibling
jump to while loop
} end-while
return FP

**end**

**Example:** Figure 17 shows an XML schema describing the structure of an author’s data. First node element will be added to Sₑ, Sₑ[1] = <author, 1, ∅, 0>. Step of adding siblings of Sₑ[1] will be skipped because Sₑ[1] has no sibling. Since LE of Sₑ[1] = 1, therefore Sₑ[2] = <name, 1, ∅, 0> will be added to Sₑ. SBₑ is ‘education’ so Sₑ[2] will become <name, 1, {education}, 1>. Then Sₑ[3] = <first_name, 0, ∅, 0> is entered into Sₑ. SBₑ is ‘family_name’ so Sₑ[3] will become <first_name, 0, {family_name}, 1>. Since LE of Sₑ[3] = 0, thus contents (only tᵥ) of Sₑ will be concatenated and saved into FP₁ where FP₁ ∈ FP. FP₁ = ‘author/name/first_name’. Sₑ[3] will be replaced by <family_name, 0, ∅, 0> because SBₑ is ‘family_name’. FP₂ = ‘author/name/family_name’ will be formed. Since Sₑ[3] has no sibling and LE of Sₑ[3] is 0, therefore Sₑ[3] will be popped from Sₑ. Sₑ[2] will be replaced by <education, 0, ∅, 0>. FP₃ = ‘author/education’ will be added to FP because LE of Sₑ[2] is 0. Sₑ[2] will be popped from Sₑ because Sₑ[2] has no sibling and LE of Sₑ[2] is 0. Sₑ[1] = {author, 1, ∅, 0} won’t be used to form FP₄ because LE of Sₑ[1] is 0. Sₑ₁ has no sibling, therefore Sₑ₁ will be popped from Sₑ. Algorithm stops as Sₑ becomes ∅. Now, the output will be:
FP = {'/author/name/first_name', '/author/name/family_name', '/author/education'}

![XML schema](image)

**Figure 17: XML schema**

### 5.2.2 Algorithm 2a: SimilarityCoefficientFinder

This algorithm calculates the similarity coefficient between two schemas with the help of algorithms (2b – 2f).

**Input:** A pair of two XML schemas $S = <A, B>$ with $FP(A) = \{FP_{A_1}, FP_{A_2}, ..., FP_{A_n}\}$ and $FP(B) = \{FP_{B_1}, FP_{B_2}, ..., FP_{B_m}\}$ where $n$ and $m$ is the number of full paths (defined in Definition 4.6, above) in Schema A and Schema B respectively.

**Output:** SimilarityCoefficient

```
begin
    shorterPath  // shorter path of the two full paths of Schema A and Schema B
    SPL := 0     // length of the shorter path
    commonPathsSchA_SchB := 0   // contains the number of common paths in the two schemas (numerator of the similarity coefficient function presented in Definition 5.2)
    denominator := 0            // contains the denominator value of the similarity coefficient function presented in Definition 5.2
    for (i:=1 to n) {
        for (j:= 1 to m) {
            shorterPath = getShorterPath(FP_{Ai}, FP_{Bj})
            pathExps_{FP_{Ai}} := generatePathExp(FP_{Ai}, SPL, genPaths_SchA)
            pathExps_{FP_{Bj}} := generatePathExp(FP_{Bj}, SPL, genPaths_SchB)
            genPaths_SchA := genPaths_SchA ∪ pathExps_{FP_{Ai}}
            genPaths_SchB := genPaths_SchB ∪ pathExps_{FP_{Bj}}
        }
    }
    commonPathsSchA_SchB := getCommonPaths(genPaths_SchA, genPaths_SchB)
    denominator := nr_genPaths_SchA + nr_genPaths_SchB – commonPathsSchA_SchB
    SimilarityCoefficient := \frac{commonPathsSchA_SchB}{denominator}
    return SimilarityCoefficient
end
```
Example: (See example after the example of Algorithm 2d)

5.2.3 Algorithm 2b: getCommonPaths
This algorithm calculates the number of path expressions common between two schemas.

Input: genPaths_SchA and genPaths_SchB, generated path expressions (obtained through Algorithm 2c) from elements of full paths of Schema A and Schema B respectively.

Output: commonPathsSchA_SchB, number of common paths among genPaths_SchA and genPaths_SchB

begin
    commonPathsSchA_SchB := 0
    for (i:=1 to n) { //n is the nr. of path expressions in genPaths_SchA
        for (j:=1 to m) { //m is the nr. of path expressions in genPaths_SchB
            if (genPaths_SchA[i] = genPaths_SchB[j]) {
                commonPathsSchA_SchB++
                jump to outer for loop
            }
        }
    }
    return commonPathsSchA_SchB
end

Example: (See example after the example of Algorithm 2d)

5.2.4 Algorithm 2c: generatePathExp
This algorithm generates path expressions from FP (input). SPL (input) is the length of shorter path of the two full paths from their respective schemas. It sets the limit on the maximum length of the path expressions that can be generated from the FP (details in Definition 5.2). Third input (genPaths_Sch) is a set which contains the path expressions (no duplicates) that have been generated from schema S during previous iterations of this algorithm where FP ∈ S. The output (genPaths_FP) contains total path expressions generated from the FP with length from 1 to SPL (inclusive). The input genPaths_Sch and genPaths_FP are used to stop the generation of a path expression twice in order to get path expressions of S without any duplicates. The function “allocateTokens(SPL, FP, m)” in this algorithm is used to allocate tokens to the elements of the FP where the number of tokens allocated to an element E indicates the number of generated path expressions in which E can be included and ‘m’ denotes the number of elements in the FP. The function “distributeTokens(elementSet, SPL, genPaths_FP)” in this algorithm is used to decrement the token of an element E of the FP to track the generation of one path expression from the FP containing E. Once the number of tokens of E allocated by “allocateTokens(SPL, FP, m)” reaches zero, E will not be used to generate further path expressions from the FP. The function “genExp_WildCard(pathExpr, SPL, genPaths_FP, genPaths_Sch)” generates path expressions with wildcards using the path expression pathExpr provided by Algorithm 2c and also generates and uses binary representation of numbers from 1 to 2^{SPL} minus 1. This binary number representation is used to suggest the position where element’s tag name in
Implementation of Data Partitioning

pathExpr should be replaced with wildcard (*) to generate a new path expression with wildcard. For example, pathExpr ("/a//b") has two elements. The binary numbers possible from 2 numbers (decimal format) are \{00, 01, 10, 11\}. ‘1’ in these binary numbers helps in replacing elements of “/a//b” with “*”. ‘00’ and ‘11’ aren’t used because ‘00’ indicates that none of the elements in “/a//b” will be replaced with “*” because “/a//b” has already been added into genPaths_FP by Algorithm 2c and ‘11’ is not used because we aren’t inserting wildcards at every position of a path expression. Thus, ‘01’ and ‘10’ will be used to generate two path expressions from pathExpr which will be: {"/a/*" and "/*//b"}.

**Input:** FP, SPL, genPaths_Sch  
**Output:** genPaths_FP

```
begin  
genPaths_FP := Ø  
if (SPL = 1) {  
  for (j:= 1 to 2^SPL) {  
    for (k:= 1 to m) { // m is the number of elements in FP  
      pathExpr := "" // will contain generated path expression  
      if (k = 1 or (position of FP[k] - position of FP[k]) = 1) {  
        pathExpr := "/" ◊ FP[k] // ◊ => concatenation  
        if (pathExpr not in genPaths_Sch  
          && pathExpr not in genPaths_FP) {  
          genPaths_FP := genPaths_FP \cup pathExpr  
        }  
      } else {  
        pathExpr := "//" ◊ FP[k]  
        if (pathExpr not in genPaths_Sch  
          && pathExpr not in genPaths_FP) {  
          genPaths_FP := genPaths_FP \cup pathExpr  
        }  
      }  
    }  
  }  
} else {  
  pathExpr := "//" ◊ FP[k]  
  if (pathExpr not in genPaths_Sch  
    && pathExpr not in genPaths_FP) {  
    genPaths_FP := genPaths_FP \cup pathExpr  
  }  
}  
end-if  
end-for  
if (SPL > 1) {  
  elementSet := allocateTokens(SPL, FP, m)  
  while (true) {  
    elem_WorkingSet = distributeTokens(elementSet, SPL, genPaths_FP)  
    if (size of elem_WorkingSet = 0) exit  
    for (j:= 1 to 2^SPL) { // for gen. path expressions with child/descendant rel.  
      pathExpr := "" // will contain generated path expression  
      for (k:= 1 to SPL) {  
        currentElemPosition_inFP = elem_WorkingSet[k]  
        previousElemPosition_inFP = elem_WorkingSet[k-1]  
        if (k=1 or (element position of elem_WorkingSet[k] in FP - element position of elem_WorkingSet[k-1] in FP) = 1) {  
          pathExp := pathExpr ◊ "/" ◊ (element at elem_WorkingSet[k])  
        }  
      }  
    }  
  }  
end-if  
```
Example of Algorithm 2c (section 5.2.4):

FP₁ := /a/b
FP₂ := /a/c

The first iteration of Algorithm 2c for FP₁ will generate the following path expressions of length from 1 to 2: {/a, //a, //b, /a/b, //a/b, //a/b, //a/*, //a/*, //a/*, //a//*}. The next iteration of Algorithm 2c for FP₂ will generate the following path expressions of length from 1 to 2: {/*c, //a/c, //a/c, //a/c, //a/c, //a/c, //a/c, //a/c, //a/c, //a/c, //a/c, //a/c, //a/c} and path expressions {/a, //a, //a/*, //a/*, //a/*, //a/*, //a//*} will not be generated because they have already been generated in the first iteration.

5.2.5 Algorithm 2d: allocateTokens

This algorithm allocates the number of tokens for each element of FP and returns elementSet having all the elements of FP containing the allocated tokens. The number of tokens is directly proportional to SPL (input) and m (input) as shown below, where SPL and m are defined in Algorithm 2c.

Input: SPL, FP, m
Output: elementSet

begin
tokens := 0
elementSet := ∅
for (k:= 1 to m) {
tokens := (m-1)C(SPL-1)  // C denotes the operation of combination
add info regarding tokens to newElement
elementSet := elementSet ∪ newElement
}
end

return elementSet
Example of Algorithm 2d (section 5.2.5):

FP1:= /a/b/c
Let’s assume that we want to generate the path expressions of length 2 from FP1 i.e., SPL = 2. Thus, the path expressions (without wildcards, without descendants relationship) will be{/a/b, /a/c, /b/c}. As we can see that each element in these generated path expressions have been used twice, therefore the number of tokens for each element of FP1 to generate path expressions of length 2 will be 2 and the above formula i.e., \((m-1)C(SPL-1)\) helps in calculating this value of token. Please note that the tokens are not used to generate path expressions having wildcards because the function “genExp_WildCard(pathExpr, SPL, genPaths_FP, genPaths_Sch)” (see algorithm 2c) generates them from each of the path expressions in the set {/a/b, /a/c, /b/c} (in this example) and the reason of not generating path expressions containing descendants relation using tokens is also similar because they are generated by Algorithm 2c for each member of {/a/b, /a/c, /b/c} (in this example).

Example
This example illustrates the logic implemented in the above algorithms (1, 2a-2d) whose purpose is to find out the similarity between two schemas. Let new schema \(s_i\) (to be allocated) is characterized by two full paths {/a/b, /a/c} and existing schema \(s_j\) (already allocated) by two full paths {/b/c, /b/d}. So our universe of path expressions would be:

<table>
<thead>
<tr>
<th>Path expressions from (s_i)</th>
<th>Path expressions from (s_j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ /a, //a, /b, //c, /a/<em>, //a/</em>, /a//<em>, //a//=, <em>/b, //</em>/b, //</em>/b, /<em>//b, <em>///b, /a/b, //a/b, /a//b, //a//=b, /a/c, //a/c, /a//=c, //a//=c, /</em>/c, //</em>/c, /*//c, *///=c }</td>
<td>{ /b, //b, /c, //d, /b/<em>, //b//=, //b//=, <em>/c, //</em>/c, //</em>/c, //<em>/c, /</em>//=c, //<em>/d, /</em>//=d, //<em>///=d, <em>///=d, //</em>///=d, //</em>///=d }</td>
</tr>
</tbody>
</table>

In total this gives us 42 (counting common path expressions once) different path expressions and this will be used as a denominator to calculate similarity coefficient. The condition of similarity is met when path expressions are common in two schemas (\(s_i\) and \(s_j\) in current example) and when it happens, the numerator will increase by 1 for each common path expression. In this example, 6 path expressions are common in the schemas \(s_i\) and \(s_j\). So after executing the algorithms (Algorithm 1 and 2a-2d), which took time of 18 ms, we find the similarity coefficient between \(s_i\) and \(s_j\) to be:

\[ Ce(s_i, s_j) = \frac{6}{42} = 0.1428 \]

Since \(Ce(s_i, s_j)\) does not meet threshold of 0.5, therefore \(s_i\) will be not be placed on the node where \(s_j\) is hosted. If the similarity coefficient between every pair of schemas comes up in the range of 0-0.5 magnitude, then documents can be allocated to any node. But the problem with this randomness is large amount of data or large number of high processing queries may end up on same node, making other nodes idle and thus resulting in imbalance.
Implementation of Data Partitioning

in workload distribution. To avoid this imbalance, documents are allocated to node with least workload. Statistics available as the metadata of schema in DD tells about the number of data pages accessed on each node. The nodes are ranked and documents are allocated to the node which has processed the least number of pages so far. This helps in making even distribution of work among processing nodes and thus can improve query’s response time.

After calculating the similarity coefficient and deciding where new set of documents should be allocated, XML documents are imported from disk which are required to be parsed first. The operation of parsing is performed using DOM Load and Save specification [32], which provides standard ways for parsing.

5.3 Complexity Degree of Determining Similarity Coefficient

In the worst case, the time complexity of determining the similarity coefficient between two schemas si and sj is O(Nsi x Nsj) where Nsi and Nsj denote the number of path expressions generated from the elements in the full paths of schema si and sj. As mentioned in Definition 5.2, our strategy picks the smaller path in the full paths of the respective schemas to reduce the comparison between path expressions.

5.4 Summary

In this chapter, we described algorithms of finding the similarity between new schema (Si) and a set of schemas (S) already allocated in the system. The list of similarity coefficients is used to decide where set of XML documents belonging to new schema should be allocated. List containing similarity coefficient between Si and Sj are ranked in descending order of similarity coefficient where ∀Sj∈S. Top entry suggests allocation of Si (and all of its XML data documents) to processing node N’ because similarity score (MaxScore) between Si and Sj’ is highest in the list where Sj’∈S. If MaxScore ≥ τ (threshold set by admin), then we proceed with the allocation of Si and all of its documents to N’, otherwise to that node which has the least workload.
Chapter 6

Implementation of Context Identification & Auto-Indexing

Our scheme of data partitioning provides enabling environment for determining the location of data (context identification) and auto-indexing. Therefore in the previous chapter, we saw how data documents are partitioned into separate libraries on different nodes on the basis of differences in their XML schemas (if differences in schemas exist), otherwise on the same libraries in the same node. In this chapter, we describe the implementation of the strategies of context identification in the first part and present implementation of an indexing technique later on. Accurate context identification is very important to avoid the execution of query on irrelevant nodes (not containing data required by query) or irrelevant libraries (not containing data required by query) within relevant nodes so that the response time of a query can be optimized as suggested by cost function (see section 4.1.3 for cost function). To detect empty result queries quickly at runtime and for further optimizing overall query performance, we also present technique of building indexes in this chapter.

Two approaches can be used for data access when data is distributed over several servers on different physical sites. Either data is fetched to the site where the query is issued or the query is sent to the nodes where required data is hosted. Former is called ‘Data Shipping’ and later is known as ‘Query Shipping’. Our method follows the concept of “Query Shipping” in which query is sent to different nodes where it is processed and result is fetched back to the node that issues query.

For achieving a higher abstraction level for the user in the evaluation of the query, one option could be in the form of sending the same query to all of the processing nodes. Those nodes which have required data will reply with non empty result and the remaining nodes will respond with empty result because required data do not exist there.

On the other hand, our technique for sending the query does some optimization over the approach mentioned in the previous paragraph. Optimization can be gained as query is sent to only those nodes which have data for the query and not to every node which saves lot of computing resources which can be used to evaluate queries (for which they have data), resulting in their better response time. The further optimization is gained through stopping empty queries from issuing to processing nodes at compile time. As we also mentioned in earlier chapters, one of the goals of database management systems is to hide structural details of data model from the users. Same thing is desired about the information of data location in any type of database architecture in general and in case of distributed database
in particular. Therefore, our method automatically translates context unaware query into a context aware query which knows location of data where it should be executed. Detailed algorithm is presented in the section 6.2.2.

The identification of context of query is determined by analyzing the content of Data Dictionary (DD). Each sub-library of root library of DD represents a node of the system and it contains XML schema documents whose associated data documents are stored on the respective node. The analysis of content (schema only) in Data Dictionary (Definition 4.1) can produce two results: Positive Match (Definition 6.1) and Negative Match (Definition 6.2), and that determines whether we should send the query for execution or not.

**Definition 6.1**: Positive Match: Existence of mapping entry of path expression from query to the set of full paths in the schema or set of schemas on a node. There could be multiple positive matches.

**Definition 6.2**: Negative Match: No mapping entry of path expression from query to the set of full paths in the schema or set of schemas on a node.

In section 6.1, the model of context identification is presented, followed by the description of its algorithms in section 6.2. This chapter ends with the presentation of model and algorithms for indexing technique (section 6.3) and a summary of this chapter.

### 6.1 Architecture of Query Context Identification

Database on one processing node can contain millions of documents and loading them all into memory for serving a query is definitely not a good idea as it will require large number of pages loading into the memory and many of them could be completely useless and could be avoided easily at compile time. In order to ensure that queries access minimum number of pages, it is run only on such libraries of the database which contain schemas whose structural properties match with the path expressions of a query. The process of context identification consists of three stages: XPath extraction, perform XPath satisfiability in the schema at DD and determine the location of data.

So before sending queries to the processing nodes, we extract path expressions from them which is also the first step of context identification. Extracted path expressions are searched in DD and upon finding them in a schema, meta-data of it provides identification of the relevant node as well as identification of library containing data documents. Figure 18 shows the architecture of our XQuery processing mechanism.

In the Figure 18, different modules contribute in determining satisfiability of XPath and identifying context of submitted query. The functionalities of the modules are defined as follows:

**Path Extractor**: It takes XQuery and extracts path expressions from it with the help of xDB parser and gives them to Predicate Filtering Module if data value is present in one or more path expressions.

**Predicate Filtering Module**: This module drops data literals from each of the path expressions (as shown in Figure 18) and gives the resultant path expressions to Context Identifier.
**Context Identifier:** It is the main module whose job is to check the satisfiability of XPath as provided by Predicate Filtering module. If satisfiability of XPath becomes successful in a schema or a set of schemas (see section 4.2.1 for the strategy of XPath satisfiability), then DD tells where the data documents of that XPath can be found and query is sent to the processing node where relevant data documents are hosted.

![Figure 18: Query Context Identification Model](image)

### 6.2 XPath Satisfiability Algorithm

In this section we see that how the task of context identification can be implemented. In this thesis, we handle queries with two kinds of relations between elements. One operator is child (/) and other is self-or-descendant (/a/).

#### 6.2.1 Characteristics of an Element in context of its Path in a Query

Each query may contain one or multiple path expressions. Each path expression is composed of a set of elements possessing a hierarchical relationship (mentioned in previous paragraph) between each other. Characteristics of each element (Ec) with respect to its path (let’s call it relevant path) are defined as:

**Definition 6.3:** $Ec = \{En, Hs, Pe\}$
In Definition 6.3, 'En' is the name of element, Hs is the hierarchical symbol (/ or //), Pe is the accumulated position of the element within the relevant path. The reference point for measuring accumulated position is the first element of relevant path. For example, ‘a/b’ is a path expression contained in input query. first element i.e. ‘a’ (same with ‘b’) is name of an element. Pe of ‘a’ in its relevant path i.e. in /a/b is 1 whereas Pe of ‘b’ in /a/b is equal to 2. Hs of ‘a’ or ‘b’ is same i.e. '/'. Take another path expression represented by '/a//b'. In it, Pe of ‘a’ is 1 but this time, Pe of ‘b’ is equal to or greater than 2. The information possessed by Hs and Pe plays an important role in speeding up the seeking operation of path expression in XML schema. This role is described in Algorithm 3b.

6.2.2 Algorithms

In order to find the location of data required by the query, we have designed and have implemented some algorithms. Design details of them are explained below.

When the query is submitted to xDB client, it goes through a number of phases before it reaches the stage of execution. In the first phase, the query is parsed. Then it is processed for its optimization and goes through other phases of query preparation so that it can be executed efficiently by xDB engine. We look into the phase of parsing only because other phases are beyond the scope of this thesis. xDB parser helps in identifying a set of path expressions of a query which is an input for the process of our context identification.

In the algorithm 3a “XQueryContextFinder” (see section 6.2.2.1), there are two inputs. First input is a set of path expressions X = {X1, X2, ..., Xn}, where ‘n’ is the number of path expressions in the XPath/XQuery ‘q’ and X is obtained with the help of xDB parser. The parser provides path expressions in unabbreviated form which is converted into abbreviated form (abbreviated/unabbreviated XPath is defined in section 2.3.1) and then searched in XML schema. Second input is a set of schemas (S) stored in DD. In the algorithm 3a, ‘DL’ contains the list of processing nodes where the query will be executed. This list may contain duplicate entries and in that case, we remove duplicates. The output of this algorithm is a set of node locations L = {l1, l2, ..., lm}, with m is the number of nodes where query could be sent and executed. If DL is empty after matching each member of X with all of the paths of all schemas, then it means that no data is found in the database according to the submitted query. Out of ‘m’, there may be zero or more nodes returning empty result, i.e. our algorithm may produce false positives in that case (results in section 8.3.1). Algorithm 3a does not execute findXPathInSchema(X'[i+1],Sj) if findXPathInSchema(X'[i],Sj) (algorithm in section 6.2.2.2) returns empty result where Sub-Exp< X'[i+1],X'[i]> is true where X'[i], X'[i+1] are abbreviated path expressions of AST of XQuery. For example, if ‘a/b’ in “for $x in /a/b return $x/c” is not satisfiable in any of the schemas of system, then the operation of checking satisfiability of $x/c is not performed.

6.2.2.1 Algorithm 3a: XQueryContextFinder

Input: X, S
Output: L

begin
    for (i:= 1 to n) {

| 52 | Implementation of Context Identification & Auto-Indexing |
\[
x_i' := \text{get abbreviated form of } X[i]
\]
for (j := 1 to m) {
    \[
    N_j := \text{findXPathInSchema}(X_i', S_j) \quad \text{// Sj is the set of schemas allocated to node } N_j \text{ with location info } L_j
    \]
    if (Nj is not \( \emptyset \))
        add Nj to DL where Nj contains ID of node and library which contains S.
} end-for
DL' := filter out nodes from DL which were added because XPath in ‘FLWOR’ expression is non-empty except emptiness of path expression in ‘return’ clause (see details in section 4.2.1).
L := buildUniqueList(DL') // one Node with same node and library ID can be present twice in DL' if more than 1 member of X is satisfied by the same schema.
return L
end

6.2.2.2 Algorithm 3b: findXPathInSchema
This algorithm takes two inputs: 1. X'[i], an i-th abbreviated path expression of the submitted query; 2. S, a set of full paths of a schema stored in DD. Output is denoted by N, location of the node where the query will be executed. Each element is defined according to Definition 6.3. The element pointer, EPj(X'[i]) gives reference to the j-th element of X'[i]. X'[i] is satisfiable in a Schema S if X'[i] is a subset of at-least one of the full paths of S. To find out this subset, there is a filtering mechanism consisting of some steps. First condition for deciding about subset is to count the number of elements in X'[i]. If it is not less than or equal to number of elements of a full path of S, then no comparison is needed between elements of the two and we can skip to next path in S. In the opposite situation, we can move on to next step of the algorithm. This involves elements matching of two entities: X'[i] and a path of S (SP). If EPj(X'[i]) points to a wildcard (*), then EP moves to (j+1)th position of X'[i] because every element name of S is fine with ‘*’ in X'[i] and therefore comparison at this element position can be ignored. If EPj(X'[i]) points to a non wildcard (*) entry, then its hierarchical symbol (Hs) is checked. Hs(EPj(X'[i])) = '/’ means that EPj(X'[i]) can only be found in SP at element position indicated by Pe(EPj(X'[i])). If outcome is negative, then it is safe to move on to the next full path of S. Hs(EPj(X'[i])) = ‘//’ suggests that EPj(X'[i]) can occur at any position in SP, starting from one supplied by Pe(EPj(X'[i])) and ends until last element of SP is visited because EPj(X'[i]) is a descendant of EPj-1(X'[i]) and same relationship should exist in S. If element matching result comes out as negative, then search is performed with next member of S. While comparing elements of a certain SP with X'[i], if algorithm never goes into negative condition, then increment in ‘j’ keeps continue until its value matches the number of elements in X'[i]. This means that all the elements of Xi’ are matched at the required hierarchical position and thus it is satisfied by Schema S. IDs of processing node and library of S (both IDs are in the metadata of S) in DD is added to the execution context of the submitted query. If all the full paths of S are explored and we can’t meet the condition regarding j’s value (i.e. not a single path expression of query is satisfied in a certain S), then this algorithm is repeated for other unexplored schemas.

Input: X_i', S \{n is number of full paths in S, S[k] is k-th full path in S\}
Output: N containing <ID of node (Nn), ID of library (N_l)>
begin
k := 0
N is empty
pathFound := false
m is nr. of elements in Xi'

// An(x) retrieves attribute name of associated element x
while (k < n) {
    j := 0
    if (nr. of elements in S[k] ≥ nr. of elements in X'[i]) {
        while (j < m) {
            if (En(X'[i][j]) != '*') { // X'[i][j] is j'th element in i'h XPath
                if (Hs(X'[i][j]) = '/') {
                    y := Pe(X'[i][j])
                    if (En(y' element of S[k]) != En(X'[i][j]))
                        jump to outer 'while' loop
                }
            } else {
                y := Pe(X'[i][j])
                for (z := y to nr. of elements in S[k]) {
                    if En(z' element of S[k]) != En(X'[i][j])
                        elementNotFoundCounter++;
                }
            }
            if (elementNotFoundCounter = (nr. of elements in S[k] - y + 1))
                jump to outer 'while' loop
        } end-if
    } j := j+1
} end-while
if (j = nr. of elements in X'[i])
pathFound := true;
}
if (pathFound = true) {
    Nn := get ID of node containing S[k]
    Nl := get ID of library containing S[k]
    add pair <Nn, Nl> to N
}
return N
end

Example: Schema book_details and author_info are characterized by the following full paths generated by Algorithm 1 (chapter 5) according to Definition 4.6:
Let's say that user query (q) is:

```plaintext
for $x in /book
where $x/name = 'The Hobbit'
return $x/author/first_name
```

The xDB parser parses q and gives three path expressions which we transform into abbreviated XPath: `/book`, `$x/name` and `$x/author/first_name`. Path expression in ‘for’ clause (‘/child::book’) is translated into abbreviated version ‘/book’. Hs of ‘/book’ is ‘/’ and Pe (accumulated position) of it is 1. Algorithm starts comparing first full path i.e. ‘/book/name’ of schema ‘book details’ with ‘/book’ of the query at position 1 (Pe = 1). Match is successful and we will move to second path expression of q i.e. ‘$x/name’. Ve<$x,'$x/name'> is true, therefore first it will be checked that ‘$x’ has been satisfied in ‘book_details’ or not. Since it has been satisfied, therefore algorithm will continue. Pe and Hs of ‘name’ in ‘$x/name’ is 2 and ‘/’ respectively. Since Hs = ‘/’, therefore we will compare ‘name’ with full paths of ‘book_details’ at second position only (no need to search at other positions of elements in any of full paths). Luckily, it matches with first full path and thus ‘$x/name’ is also satisfied in schema ‘book_details’. Finally we come to check the satisfiability of ‘$x/author/first_name’. It is also satisfied in ‘book_details’ due to 2nd full path. All the matches are successful, therefore associated node and library details of schema ‘book_details’ is added to execution context that will receive query q.

The algorithm will also match path expressions of query q with other schema ‘author_info’ at DD because we want to avoid any false negative. Since none of the three XPaths of q will be satisfied, therefore library ID of ‘author_info’ won’t be added to the execution context of query q.
Let’s consider another query \( q' \) to see different behavior of algorithm \( 3b \):

```plaintext
for $x$ in /book//chapter
where $x/title$ = ‘The first morning’
return $x/summary$
```

Abbreviated path expressions of \( q' \) are: ‘/book//chapter’, ‘$x/title’, ‘$x/summary’. \{En, Hs, Pe\} of ‘/book’ is \{book, ‘/’, 1\} whereas \{En, Hs, Pe\} of ‘//chapter’ is \{chapter, ‘//’, 2\}. \{book, ‘/’, 1\} tells that book can be present at first position of any of the full paths of a schema. This condition is met in full paths 1 through 9. Then element pointer advances to the second element of ‘/book//chapter’ i.e. ‘//chapter’. Its element information \{chapter, ‘//’, 2\} suggests that ‘chapter’ should be found at 2nd position or at later position (because Hs = ‘//’) of a schema’s paths which fulfills in 7th, 8th and 9th path, but element pointer is moved to next element right after having a successful match in 7th path. Likewise, other two XPaths i.e. ‘$x/title’ and ‘$x/summary’ are satisfied in ‘book_details’ as they are matched with its 7th and 9th full path. No XPath of \( q' \) is satisfied in ‘author_info’ schema and therefore node and library details of only ‘book_details’ schema are added to execution context where query \( q' \) will be sent and executed.

### 6.2.3 Complexity Degree of Context Identification Process

\( P_Q \) is a set of unique XPaths in query \( Q \) whose size is represented by \( P_N \). In the worst case, the comparison operation of XPaths in the query with schemas is \( O(S_N \times P_N) \) irrespective of type of query (cross-node join or not), where \( S_N \) is the total number of full paths of all the schemas in Data Dictionary. Due to optimization possible in checking the satisfiability of an XPath (see section 4.2.1.1), all XPaths in \( P_Q \) may not be required to compare with the schemas that will optimize context identification process.

### 6.3 Auto-Indexing Algorithms

**6.3.1 Architecture Overview**

We have already explained the purpose and contribution of having element-value index. Those queries are considered for element-value index that have been accessing highest number of pages and have been in high frequency. It is important to distinguish between high and low number of data pages because queries accessing high number of pages consume more processing power. If index is built on elements which are involved in those queries, then positive impact on the performance of query by utilizing that index will be bigger. The queries accessing low number of pages will get outcome (empty or non-empty result) in relatively short amount of time, lowering the importance of elements present in those queries. In general, it speeds up every query if a leaf-element (on which index is already built) is a subset of path expression(s) of the submitted query. Figure 19 describes the model of ‘Auto Indexer’ with the help of an example query (FLWOR expression) and we can see that five modules are participating to build element-value index.
In order to understand how Auto Indexer works, we should go through the functionalities offered by its different components. These components have the following responsibilities, presented in the order in which they participate in the process.

**Path Extractor:** Its function is same as described in section 6.1. In Figure 19, path expression in the predicate i.e. "/a/b/c = ‘data value’" is the outcome of Path Extractor which is further processed by Auto-Indexer to build element-value index.

**Context Identifier:** It is the central component of the model of query context identification process (section 6.1).

**Leaf Element Filtering Module:** The mining of leaf elements from path expression is performed in two steps. The first step is executed in this module which is the extraction of leaf elements from the path expression. The last element of path expression in the query could be the potential candidate to become member of a set of leaf elements in a query so this module extracts last element to form set of potential candidates. Auto Indexer refines this set with the help ‘DD look-up module’ to ensure each member of this set is a leaf element by removing non-leaf elements (if exist).

**DD look-up Module:** The use of this module by Auto-Indexer is to check that recipient element is leaf or non-leaf node by looking into XML schemas included in context set (determined by Context Identifier) rather relying only on last elements of path expressions as provided by ‘Leaf Element Filtering Module’.
**DD storage Module:** Statistics in the Data Dictionary are updated when one or more leaf elements are accessed through a query. This process is performed by Auto-Indexer with the help of this module. The detail algorithm of building and updating statistics of query is explained in section 6.3.2 and 6.3.3 respectively.

### 6.3.2 Statistics Collection (Algorithm)

This algorithm builds statistics containing information regarding the leaf elements of path expressions of submitted query so that value-index can be built on them. Input of the algorithm is LE, which is a set of leaf elements, extracted from path expressions in the query. xDB provides AST of the query and our algorithm traverses AST to get a set of path expressions, extracts last element of such path expressions and adds them to LE. A leaf element le ∈ LE (input of addQueryPredicateInfo) doesn’t have the context information (Definition 4.7) of leaf element i.e. the node and schema ID of le is unknown which are obtained during context identification of the query (see section 6.1). Each member of LE’ is a leaf element node, with context information.

#### 6.3.2.1 addQueryPredicateInfo()

**Input:** LE = {LE1, LE2, ..., LEn}

**Output:** LE’ is added to DD

**begin**

\[
\text{LE'} := \text{removeNonLeafElements(LE, FP)} \quad // \text{FP is a set of full paths of a schema} \\
\text{for } (i := 1 \text{ to } m) \{ \\
\quad \text{elementName} := \text{getElementName(LE'}[i]) \\
\quad \text{libraryName} := \text{getLibraryName(LE'}[i]) \\
\quad \text{nodeName} := \text{getNodeName(LE'}[i]) \\
\quad \text{if } \text{nodeName exist in predicateAccessInfoDoc in DD} \{ \\
\quad \quad \text{if } \text{libraryName exist as a sub-element of nodeName in predicateAccessInfoDoc} \{ \\
\quad \quad \quad \text{if } \text{elementName is sub-element of libraryName in predicateAccessInfoDoc} \\
\quad \quad \quad \quad \text{increase 'access_freq' by 1} \\
\quad \quad \quad \text{else} \{ \\
\quad \quad \quad \quad \text{create sub-element of libraryName for element with name 'elementName'} \\
\quad \quad \quad \quad \text{access_freq} := 1 \\
\quad \quad \quad \quad \text{create sub-element ‘isCreated’} \\
\quad \quad \quad \quad \text{isCreated} := 0 \quad // 0 => element index is not created \\
\quad \quad \} \} \}
\quad \text{else} \{ \\
\quad \quad \text{create sub-element of nodeName for library with name ‘libraryName’} \\
\quad \quad \text{create sub-element of libraryName for element with name ‘elementName’} \\
\quad \quad \text{access_freq} := 1 \\
\quad \quad \text{create sub-element ‘isCreated’} \\
\quad \quad \text{isCreated} := 0 \quad // 0 => element index is not created \\
\quad \} \} \}
\]

**end**
else {
  create element for node with name 'nodeName'
  create sub-element of nodeName for library with name 'libraryName'
  create sub-element of libraryName for element with name 'elementName'
  access_freq := 1
  create sub-element ‘isCreated’
  isCreated := 0 // 0 => element index is not created
}
} end-if
} end-for
end

Example: Query Q:= for $x$ in /publisher where $x/name = 'Martin Ravalec' return $x/authors; Statistics in Figure 15 are changed after running Q. Access frequency of element ‘name’ in ‘publisher’ library will be changed from 15236 to 25294 (Q accessed 10058 data pages).

6.3.3 Creation of element value index

In creating value index on elements, top n% of leaf elements ranked by access frequency of data pages are determined from predicate access statistics. In algorithm (section 6.3.3.1), we have used n% as 10% (n% is customizable). Elements of set LE and LE’ are defined according to Definition 4.7. We can differentiate between elements on the basis of their context (node and schema ID) if element name of two or more elements of LE’ are same.

6.3.3.1 buildIndex

Input: LE
Output: indexCreated {will be 0 if index is not created, otherwise 1}

begin
  if (size of LE = 0)
    indexCreated := 0
  else {
    LE’ := 10% of LE
    if (size of LE’ < 1)
      LE’ := first element of LE
  }
  endif

  for (i := 1 to n) {
    createIndex(LE’[i])
    isCreated := 1
  }
} endfor
indexCreated := 1
return indexCreated
end

Example: When index is built using statistics shown in Figure 15, LE will have {name, name}. 10% of it is less than the absolute value of 1, therefore LE’ := {name}. Thus, algorithm will build index on ‘name’ element of ‘book’ schema and won’t build index on
‘name’ element of ‘publisher’ schema because of lower number of data pages accessed. After creating index, if the query accesses information from XML documents of ‘book’ schema, then query containing ‘name’ element in its path expression will access only those pages which contain the text value that have been asked as text value of ‘name’ in the query. Figure 20 shows updated statistics.

![Figure 20: Updated Statistics](image)

6.4 Summary
In this chapter, we saw how the emptiness and non-emptiness of query’s result set can be determined at compile time using our context identification process. This process also helps in optimizing the search space of the query so that query is executed only on relevant set of documents. Our algorithm of indexing speeds up not only the detection of queries which are declared non-empty at compile time (using our context identification procedure due to inaccessibility of data documents by it) and eventually come out with empty result but also helps in further optimization of all the queries.
Part III
Baseline Approaches and Evaluation
Chapter 7

Introduction to Baseline Approaches

The strategies and implementation of our main approaches for data partitioning, context identification and index building have been the focus in previous part of this thesis. In this chapter, we describe some baseline approaches of data partitioning (section 7.1) and context identification (section 7.2). The purpose of these baseline techniques is to compare their performance with sophisticated approaches (see chapter 8 for the comparison).

7.1 Basic Approach of Data Partitioning (Random Distribution)

In this approach, data documents (to be stored) do not go through the phase of our data partitioning and thus, schema and all of its documents are allocated to a node on the random basis. The process of data allocation starts when one of the clients in the system takes input in the form of XML data documents and their schema. Some notations regarding this scheme of data allocation are:

- Ci = the i\textsuperscript{th} client of system S, with i has a range from 1 to m
- m = total number of clients interacting with the system S
- Nj = the j\textsuperscript{th} processing node, with j varies from 1 to n
- n = total number of processing nodes working in the system S

Figure 21 shows the model of baseline partitioning where new schema and its documents are allocated to Node N2 on random basis:

[Diagram of data allocation]

7.1.1 Allocation of XML Documents to Nodes

In this approach of data partitioning, set of documents can be stored to a random node, let’s call it Nj where j varies from 1 to n. All documents irrespective of their schema will be stored in a single library (root library) on Nj. Selection of node is random, and all of the
nodes have equal probability of receiving new schema and whole set of its documents. Distribution of workload, schema similarity or any other criteria is not considered for storing new documents. In Figure 21, documents’ set (denoted by ‘Documents’) and its schema are allocated to N2.

7.1.2 Query Processing
Since the allocation of schemas is purely random, each node has equal chance to be selected for storage of schema and all of its documents. It is also possible that the documents of a same schema are distributed over more than one node because schema similarity is not calculated before allocating documents in this random approach. Due to this, each query will be sent to all the nodes for processing. For example, there is a query q which is sent to three processing nodes (total nodes = 3). Since documents are stored in the root library of the database, all documents will be searched in each of the three nodes in order to calculate the result of q. This will cause performance degradation (more response time) of its own as well as of other queries running in parallel (if any). This situation is unlikely to happen if data partitioning is based on clustering of schemas (see more analysis in section 8.4.3).

7.2 Basic Approach of Context Identification
The baseline approach of context identification will be compared with optimal approach in order to highlight their different behavior in different scenarios (different constructs of XQuery). Because both of them require context identification, therefore utilization of schema cluster-based data partitioning is needed to provide DD (pre-requisite of context identification process) in order to do comparison between them.

The baseline approach considers all path expressions of the query to get information regarding the location of data. This approach will avoid the occurrence of any false negatives but on the other side, it will produce more false positives as compared to optimal approach which is described in chapter 4 (algorithmic details in chapter 6). The reasons behind occurrence of higher false positives are different due to variance in constructs of XQuery or XPath which is explained as follows:

7.2.1 FLWOR expression
The result of a FLWOR expression is empty when path expressions in every ‘return’ clause are empty. The result of a FLWOR expression may be empty when path expressions in every ‘for’ clause are empty but it is not necessary to occur (see details in section 4.2.1.1). Since baseline approach considers path expressions in every clause of FLWOR expression and sends query for execution when at least one path expression of the query is satisfied. This design will produce more false positives (comparing to sophisticated approach) when path expression in ‘for’, ‘let’ or ‘where’ clause is non-empty but ‘return’ path expression is not satisfiable in any of the schemas. For example, all path expressions of query q “let $x := /a/b for $y in $x/c return $y/d” are checked whether they are satisfiable in any schema. Let’s assume that ‘/a/b’ and ‘$x/c’ are satisfiable in schema s and thus baseline approach sends q to node n which hosts schema s and its documents. So far so good, but the result of q is empty (because ‘$y/d’ in q is empty) and thus query should not be sent to n for unnecessary processing. Using baseline approach, query will be sent because two path expressions of q are present in the schema and this doesn’t happen in optimal approach as explained in section 4.2.1.
7.2.2 Logical (union) And Conditional (If-then-else) Expressions

As mentioned in chapter 4, logical expressions can be present as a subset of FLWOR expression in a query. The baseline approach performs as good as the optimal approach if path expression p is logical expression but if p is a sub-expression of a FLWOR expression in the query q (if q contains FLWOR expression), then difference between baseline approach and sophisticated approach becomes visible due to difference caused by FLWOR expression as mentioned in section 7.2.1. For example, query: “(/a/b | /c)” will behave same in baseline and sophisticated approaches but baseline approach may incur false positives in another query: “let $x := (/a/b | /c) return $x/d if ‘$x/d’ is not satisfiable. This will not happen in the sophisticated approach. More analysis on this difference between two approaches is presented in section 8.3.

If path expressions in ‘then’ and ‘else’ clauses are empty and path expression in ‘if’ clause is non-empty, then false positives can occur in baseline approach. For example, let $x := /a/b return if $x/c = ‘Morning’ then $x/g else $x/d” will be declared non-empty in baseline approach if p is non-empty where $p \in \{/a/b, ‘$x/c’, ‘$x/g’, ‘$x/d\}$ in current example.

7.3 Summary

In this chapter, baseline approaches for data partitioning (i.e. Random Partitioning) and context identification (Baseline Approach) have been presented. In baseline approach of context identification, all path expressions of the query are checked. In this way, two things are determined: 1). Yes or No answer of “does path expression satisfy in the set of available schemas?” and 2). If such schema(s) exist, then it also gives the location of data belonging to that schema so that query can be sent to right place for execution. In terms of accuracy of context identification, baseline approach produces more false positives than optimal approach in certain constructs of FLWOR expression (see chapter 8 for detailed empirical analysis).
Chapter 8

Evaluation

Through previous chapters, we presented strategies and algorithms of data partitioning, context identification and value based indexing. In this chapter, we present comparative analysis of algorithms with the help of experiments and their results.

8.1 Parameters of Evaluation
In chapter 4, we argued that the response time of a query is heavily dependent on how many pages are searched by xDB. This direct proportionality between response time and number of data pages concludes that we should measure the performance of our strategies through the number of pages accessed in satisfying a query. It is also an accurate measure as compared to some other time-based parameters whose accurate determination is difficult to establish.

8.2 Experimental Setup
The specification of the system used for the evaluation of our prototype is: (CPU clock speed: 2.70 GHz; RAM: 8 GB; OS: Windows 7 64-bit). There are two processing nodes and one client being used in our experiments (except the experiments of section 8.3 where we use three nodes for interesting analysis). The number of processing nodes can be expanded easily. In order to produce reliable results, several XML documents belonging to different schemas have been created using xTest and used as a dataset in our experiments. The xTest Test Framework is a test framework designed to quickly and easily set up performance tests and other long-running tests against xDB and other systems\(^5\). We have taken multiple readings by running each algorithm 20 times and have taken the median of the series to remove aberrant figures to get a final result that is statistically significant.

8.3 Effect of Context Identification
The performance of a query gets improved if we accurately identify the node and the library which constitute the context of query execution. In this section we try to support this claim with the help of experimental results.

8.3.1 Precision of Context Identification Process
In this section, we distinguish between approaches for context identification in terms of false positives and false negatives. False negative is not occurred in any of the approaches

presented in this thesis. Before delving into the results of experiments, here is a brief recap of baseline approaches and optimal approach.

**Baseline:** The underlying partitioning technique in this approach of context identification is schema cluster-based. The location of data is identified by extracting all path expressions of the query and then checking which schemas satisfy them. The host nodes which contain those schemas where path expressions of the query are satisfied, are added to the context of the query.

**Baseline – All Nodes:** This approach sends the query to all nodes of the system because data partitioning is random.

**Optimal Approach:** IDs of those processing nodes are included in the query context set which are obtained by extracting and analyzing emptiness/non-emptiness of path expressions and finally by considering only result oriented path expressions (conceptual details in chapter 4, algorithmic details in chapter 6).

**Example:** The following queries have been used to evaluate the performance of context identification using optimal approach and to discuss its benefits over baseline and baseline (all nodes):

1. `/book//chapters/chapter`
2. `/publisher[@name]`
3. `/book[@name = 'The rising sun']`
4. `for $x in /book for $y in $x[publisher = "Million Dollar Baby"] return $y/name`
5. `for $x in /book[fn:not(name)] return $x`
6. `/book/[./publisher = 'Martin Ravalec']/author`
7. `let $y := /book[name = "Harry Potter and the Philosopher's stone"]/publisher return $y`
8. `for $x in /book let $y := $x[name = "Harry Potter and the Philosopher's stone"]/author return $y/name`
9. `for $x in /book for $y := $x[name = "Harry Potter and the Philosopher's stone"]/chapter return $y`
10. `for $x in /publisher for $y := $x[name = "Harry Potter and the Philosopher's stone"]/book/name return $y`
11. `let $x := /publisher//author_name let $y := //publisher//author return ($x | $y/name)`
12. `(book/title or /book/name)`
13. `for $x in doc('book0.xml')/book let $y := $x[name = "Harry Potter and the Philosopher's stone"]/author return $y`
14. `for $x in /movie_record/novel_info/novel_based return $x`
15. `for $x in /movie_record/novel_info/novel_based where $x = 'yes' return $x/movie`
16. `let $x := /movie_record for $y in $x/movie where $x/novel = 'The Next Target' return $y`
17. `let $x := for $y in /TVshow/director return $y/show_title return $x`
18. `let $x := /movie_record for $y in $x/movie where $x/novel = 'The Next Target' return $y`
19. `for $x in /movie_record where $x/novel_info/novel = 'Dreams' return $x/movie`
20. `let $x := //book/name let $y := //movie return ($x | $y/novel)`
As we explained in chapter 7, baseline will extract and search the set of all path expressions $P$ of the query in the database (DD) and will add the location of all those nodes which contain the schema where at least one path expression of $P$ is satisfied. This approach will not miss any information but the query would be issued against irrelevant nodes too. The baseline approach (all nodes) will issue query against all processing nodes and some of them will return empty result set that do not contain required data. Though both baseline approaches will incur no false negative but the number of false positives will be higher than optimal approach. Optimal approach is better than the other two approaches because false positives is smaller in number and will be zero for specific constructs of XQuery (explained in chapter 4). False positive is difficult to eliminate because DD is used to identify the location of data using schema and it doesn’t contain data documents.

In Table 3, $o$ => neither false positive nor false negative; $(n)p$=>false positives incurred on ‘$n$’ nodes where $(n)p$ is worse than $(n-1)p$ and ‘$o$’ is best. Table 3 shows the results of the above queries to indicate which is better in terms of precision of context identification when they were run on xDB containing documents and their schemas (schemas are shown in Appendix E).

<table>
<thead>
<tr>
<th>Query</th>
<th>Baseline (All Nodes)</th>
<th>Baseline</th>
<th>Optimal Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2p</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>2</td>
<td>2p</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>3</td>
<td>2p</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>4</td>
<td>2p</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>5</td>
<td>2p</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>6</td>
<td>2p</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>7</td>
<td>2p</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>8</td>
<td>3p</td>
<td>1p</td>
<td>o</td>
</tr>
<tr>
<td>9</td>
<td>3p</td>
<td>1p</td>
<td>o</td>
</tr>
</tbody>
</table>
As we can see in Table 3, the baseline (all nodes) approach performs poorly in terms of precision of context identification and causes false positives in case of all the queries. Baseline and optimal approach are better than baseline (all nodes). Difference between optimal and baseline approach lies in those constructs of XQuery when the satisfiability of one or more path expressions is true but some of the path expressions which contribute towards the result of the query are not satisfiable. For example, query q: “for $x$ in /a/b return $x/c$” has empty result because $x/c$ is not satisfiable in any schema whereas converse holds for /a/b. Optimized approach of context identification can handle this case correctly but in the baseline approach, q will be sent to the processing node because of non-emptiness of ‘/a/b’. We have already mentioned earlier in this chapter and in chapter 4 as well that optimal approach produces false positives only when the targeted database doesn’t have documents containing data values required by the query and not due to structural properties of XML document like elements’ tag name or attributes. Table 4 shows the summarized results of Table 3.

Baseline(all nodes) sent all of 20 queries to all the processing nodes (3 in our experiment) and Table 4 shows that all 3 nodes returned empty result for 8 queries, 2 out of 3 nodes
returned empty result for 10 queries and 1 node out of 3 nodes returned empty result for 2 queries. In general, performance of a query in baseline (all nodes) approach will be affected adversely to large extent because nodes will be receiving such queries as well which could have been stopped before issuing through context identification process. On the other side, accuracy of baseline approach of context identification is better i.e. 60% and optimal approach comes at top with 90% accuracy which shows that optimal approach is the best approach available among the three approaches. The negative effect on the query performance is analyzed in detail in section 8.4.

<table>
<thead>
<tr>
<th></th>
<th>Baseline(all nodes)</th>
<th>Baseline</th>
<th>Optimized Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>3p</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2p</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1p</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>o</td>
<td>0</td>
<td>12</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4: Precision of Context Identification (Summarized Result)

8.4 Differentiation in Query’s Performance in Different Approaches

In the previous section, we saw the precision of context identification in different approaches. Now it is time to analyze the performance of the queries in different approaches.

8.4.1 Performance of a Single query

This experiment shows the difference in data pages accessed by baseline (all nodes) approach and optimal approach. In baseline (all nodes) approach, no DD is available and thus system doesn’t know where query q1: “for $x$ in/book return $x/publisher” will get non-empty result set from, so it will be sent to all nodes Ns (i.e. 2). There will be no cost to calculate context (Ctx = 0, defined in cost function, section 4.1.3.3). Since we don’t know the context of query, all documents will be searched in the database, hosted on each node of the system.

In optimal approach, only Ni (i.e. N1) is the node which contains required data and library Lj (i.e. ‘book’) in Ni contains the documents of a schema which satisfies the path expressions (/book’ and ‘$x/publisher’) of ‘q1’. N1 contains 50,000 documents and Lj contains 10,000 documents. The results are shown pictorially in Figure 22 (y-axis represents number of data pages accessed in serving q1):

![Figure 22: Query’s Performance without Context Identification](image)
In both approaches, required data is hosted on N1. In order to get a better understanding of how the resources are consumed or wasted in executing a query q1 without context identification, we are showing performance differences of two approaches on a single node (i.e. N1 because required data is hosted on N1 in both approaches) with single query execution mode (i.e. q1 only). Blue bar shows number of pages searched if q1 is executed on optimal approach. The degradation in query performance (red bar) takes place in baseline (all nodes) approach and it is easily generalizable to the environment where query q1 will be consuming computing resources of other nodes (Ns-Ni) in the system network in baseline partitioning, slowing down the set of queries (Q') executing there because location of data is not associated with query at compile time. On the other side, Q' will be affecting q1 and other queries at N1 (if multiple queries are running at N1) and this phenomenon is explained in section 8.4.2.

8.4.2 Influence on Query’s Performance from other Parallel Queries

The underlying partitioning scheme is random in this experiment, showing the effect on the performance of a query which is evaluated on a node where other queries are also received but they are returned with empty result. This situation (receiving empty result queries) will mostly occur in baseline (all nodes) approach as explained in section 8.3.1. Like the previous section, we are showing the performance of one query i.e. Query 1: “for $x$ in /book return $x$/publisher”, at N1 (N1 contains 50,000 documents of different schemas) and the behavior of node N2 towards Query 1 is omitted to concentrate fully on one aspect i.e. effect of multiple queries (with empty result) on Query 1. In section 8.4.3, we see how multiple queries perform on multiple nodes with the analysis whether emptiness or non-emptiness of result set makes any difference in terms of processing required by nodes.

We evaluate the performance of ‘Query 1’ at N1 which also receives queries whose XPaths are not satisfied at N1 (as context identification is not utilized) and all queries are assumed to be received at time t. Figure 23 shows the addition in the response time (ms) of Query 1 in five different cases. Case 1 denotes that only Query 1 is run on N1, one query in addition to Query 1 is run in Case 2 and similarly the number of parallel queries keeps increasing in other cases (Case 3, Case 4 and Case 5). Figure 24 shows the same thing in terms of percentage of Query 1 (i.e. Addition in Response Time / Query 1’s response time). For example, in Case 2, Query 1 takes 190 ms in addition to what it takes in Case 1. So, in terms of %, it will be 190/485 = 39.5% where 485 is the response time (ms) of Query 1.

It is difficult to visualize the adverse effect of empty result queries on the performance of non-empty result queries if test results are shown in unit of data pages, therefore in the graph (Figure 23), we use time unit to show the effect of execution of different number of queries (all have empty result except Query 1) on Query 1 in each case. As is clear in Figure 23 and Figure 24, empty result queries deteriorate the performance of Query 1 because we sent query to every node of the system without context identification and node which executes Query 1 is also executing other queries (simultaneously) for which they have no data. Red curve shows addition in the response time of Query 1 in each case. Our cost function (here $Ctx = 0$) explains that the slowness in Query 1 is contributed by third component of the cost function (i.e. effect of other queries) and the current example also suggests weight factor of cost function as about 0.2 for two parallel queries and 0.3 - 0.34 for more than 2 queries in our experimental setup (section 8.2). Context identification helps in filtering out those queries which are going to return empty result sets from nodes. Other nodes in addition to Ni will also face this situation because they will also receive irrelevant
queries. In addition to stop sending queries to certain nodes (which don’t have data for queries), context identification also helps in stopping issue of queries to empty nodes, if any.

In another experiment, we have 250,000 documents stored on the node. Query Q1 = “for $x$ in /publisher where $x/name = 'Ravalec'$ return $x/authors/published_author” is sent to all of the nodes because Q1 is not passed through the step of context identification. The following two figures show the influence of other queries on Q1 at Node N1 which returns nonempty result of Q1 and empty result for all other queries.
Figure 25: Additional Response Time of Q1 due to Parallel Queries (%) in 250,000 Docs

Figure 26: Additional Response Time of Q1 due to Parallel Queries (ms) in 250,000 Docs

The main purpose of experiments in this section is to show that other parallel queries (having empty result set) do influence Q1 and we saw that every query irrespective of whether its result set is empty or non-empty increases response time of Q1. Fortunately false positives in optimal approach occurs less frequently as compared to baseline approach (all nodes), therefore processing nodes will receive such queries (like Q2 through Q5) in lesser quantity in optimal approach that will speed up query execution.
8.4.3 Query Performance in Random and Schema Cluster-based Partitioning

We have already argued that query is executed more efficiently in clustered-based partitioning (optimal approach) as compared to random partitioning because former offers accurate context identification. Unlike previous experiments, we are going to show experiment result of two nodes with mixed number of queries having empty and non-empty result set. In tests (results are shown in Figure 27), queries are passed through the phase of context identification in optimal approach but this is not available in random partitioning. Figure 27 shows the results highlighting the difference of the two approaches of data partitioning in satisfying a query. In Random partitioning, N1 contains two schemas and all of their documents. Likewise, N2 hosts three schemas and all of their documents. In optimal partitioning, fives schemas and all of its documents are allocated to N1 where N2 doesn’t contain any data.

![Schema Cluster-Based Partitioning and Random Partitioning](image)

The bar (green) shows the number of pages accessed by each of the five queries in the data partitioning schema where documents of a schema or perfectly matched schemas (Definition 4.2, above) are allocated in same library on a node (schema cluster-based partitioning). The bars (red and blue) indicates the number of pages when schema and all its documents are assigned randomly to a node and documents of each schema are not organized into separate libraries in the database as all data at a node is stored under root library. It is visible that random partitioning requires all pages of the database to read for preparing result of the query and every query (Q1 to Q5) are executed on all the nodes (N1 and N2 in current experiment) of the system. Schema cluster-based partitioning is beneficial in two aspects as compared to random partitioning: 1). Query is sent to only those processing nodes which are expected to host required data and 2). Query is run only on those documents of the database whose schema satisfies the path expression(s) of the
query. Due to these two reasons, pages accessed (green bar) in schema cluster-based partitioning for preparing query result is low in number.

It is also worth to mention that every query (Q1 to Q5) returned non-empty result from processing node N1 in schema cluster-based partitioning where in random partitioning, situation is quite different. In random partitioning, we got the following results:

<table>
<thead>
<tr>
<th>Query</th>
<th>Node N1</th>
<th>Node N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Non-Empty result</td>
<td>Empty result</td>
</tr>
<tr>
<td>Q2</td>
<td>Empty result</td>
<td>Non-Empty result</td>
</tr>
<tr>
<td>Q3</td>
<td>Non-Empty result</td>
<td>Empty result</td>
</tr>
<tr>
<td>Q4</td>
<td>Empty result</td>
<td>Non-Empty result</td>
</tr>
<tr>
<td>Q5</td>
<td>Empty result</td>
<td>Non-Empty result</td>
</tr>
</tbody>
</table>

Table 5: Empty/Non-Empty Result Set

If we analyze the results presented in Figure 27 and Table 5 together, we can see that xDB has to read data pages for determining the result of a query despite the fact that some of the queries in the experiments of this section obtained empty result from one of the two processing nodes in random partitioning. Similar results are also shown in section 8.4.2.

Random partitioning is not suitable for building value-based indexes either. We need to define element index in terms of its parent library ID (we create element index at library) and elements may be common to documents of different schemas but not all of them are expected in the query. Schema can bring differentiation in these common elements but in random partitioning, all documents irrespective of their schema are stored into a single (root) library within a database on a node. Thus, schema based partitioning is better than random partitioning in two aspects: firstly, it allows allocation of documents into separate libraries according to their schemas which helps queries to run on those documents only whose schema satisfies XPaths of the queries. Secondly, schema based partitioning provides enabling environment for creating element-value index.

### 8.5 Performance gain due to Value based indexing

Queries involving predicates can get benefit of this second phase of query pre-processing before their execution if relevant index exists. Non-predicate or non-selective queries cannot qualify for such performance optimization. All queries irrespective of their type will have to go through the phase of context identification to get context information before entering into this phase because indexes in this thesis are created at the level of library and identification of library is part of context information. Without the awareness of library ID, utilization of indexes is not possible.

We have run the same set of queries under two different experimental conditions. Under the first condition, there exists no index and the second condition enjoys the presence of relevant element-value indexes. Table 6 shows the performance edge attainable to the following queries (Q1 and Q2) in the second environmental condition. Difference between Q1 or Q3 and Q2 is that Q1/Q3 returns non-empty result where Q2 returns empty result.
Example:

Q1: /book[/publisher = 'Martin Ravalec']//author
Q2: /book[/publisher = 'NewHouse']//author
Q3: for $x$ in /publisher where $x$/name = 'John Publishing' return $x/authors

<table>
<thead>
<tr>
<th></th>
<th>No Index is Available</th>
<th>Index is Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10071</td>
<td>7</td>
</tr>
<tr>
<td>Q2</td>
<td>10070</td>
<td>6</td>
</tr>
<tr>
<td>Q3</td>
<td>10067</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6: Performance Gain through Value Index

We can see that if index is already built and query is hit against that index, then the pages processed by xDB is way less than the situation where no index exists. Q2 which is declared as a query containing result by our context identification (Q2’s path expression are satisfied in a schema) has empty result in actual. Fortunately due to value-based indexing, detection of emptiness of Q2’s result set can be done quite quickly and the processing of 10070 data pages is easily avoidable and thus, only 6 data pages are needed to be processed by xDB. Performance improvement is also quite promising in other queries (Q1, Q3 in this experiment).

Elements like ‘name’ or ‘id’ can be common across many schemas but they all might not occurring in queries frequently. In order to bring differentiation between them, our design of Auto-Indexer builds index on element schema wise and our data partitioning scheme support this strategy because it also stores documents by schema.

8.6 Summary

This chapter presents empirical analysis and shows how performance of XQuery and XPath gets better using our approaches of data partitioning and context identification. Our data partitioning scheme helps in reducing search space of the query so that small amount of data pages are processed by xDB.

Though our approach of context identification (Optimal Approach) doesn’t eliminate the occurrence of false positives (occurs due to unavailability of data at compile time) but it produces less number of false positives (10%) as compared to other two approaches (Baseline and Baseline (all Nodes)).

Data pages are loaded and processed by xDB irrespective of emptiness or non-emptiness of the query’s result. In relation to this claim, we observed in this chapter that if query contains empty result set in reality but Optimal Approach declares it as non-empty, then element-value index saves lot of computing resources of certain node(s) which can be used
to serve other queries. In general, it speeds up every query if a leaf-element (on which index is already built) is a subset of path expression of the submitted query.
Part IV

Conclusions and Future Work
Chapter 9

Conclusions and Future Work

In this chapter, we summarize the strategies presented in this thesis, review answers to the research questions we set out to address and suggest some future directions for further development of these ideas.

9.1 Summary

The thesis proposes a computationally efficient method for allocating XML data in a distributed environment with the goal of optimized servicing of queries. Two major research questions were addressed: 1) Can a data partitioning scheme make a difference in query performance (minimize response time) and 2) Whether determination of emptiness/non-emptiness of the results of XQuery or XPath at compile time be used for query optimization.

The results of the proposed partitioning scheme are very promising. Our method allocates new documents to different nodes according to schema similarity: documents of similar schema go to the same node (space permitting). Within that node, documents are organized into separate libraries where each distinct schema has its own library. A typical query is expected to target documents of a single schema (but it’s not mandatory). Therefore most queries would not be executed on irrelevant nodes or irrelevant set of documents. Even if a given query has a set of path expressions that are satisfied in multiple schemas, it will be executed only on those documents that satisfy the referenced path expressions. In contrast, our experiments show that a random partitioning of documents leads to a large number of data pages to be processed in order to prepare the result of a query, since the location of data is not known at all prior to the query execution. This causes unnecessary burden on the processing nodes, thus increasing response time of the queries.

Regarding the second research question, we found that in the absence of some clever optimization scheme, a query has to process all the relevant documents irrespective of the emptiness or non-emptiness of the result set. Through our context identification process, we optimized the detection of the emptiness and non-emptiness of the result set for certain forms of XQuery (containing FLWOR, If-Then-Else and Union expressions in it) at compile time. This way, a query is not sent to those processing nodes that would certainly return an empty result set, given the information the system is maintaining about the schemas hosted at the respective nodes. This approach of context identification (Optimal Approach) does produce some false positives (occurs due to unavailability of data at compile time) but this happens less frequently (10%) than with the other two approaches (Baseline and Baseline (all Nodes)), the latter being the worst of the three approaches.

The false positives occur in the cases where the processing node searches for the data but doesn’t find anything matching the query, causing delays in the execution of other queries.
We build element-value index to address this. Obviously creating indexes on all of the elements would have high cost due to updates to the indexes in response to document updates. In order to avoid this situation, indexes are built on the elements of such path expressions that are referenced by the queries that access a high number of data pages and/or are executed frequently. The higher the number of page accesses, the higher the importance of the elements belonging to the schema containing those paths, from the point of view of the query workload. Indexes on such elements will have a bigger impact on the performance of the queries. The queries accessing a low number of pages will get the outcome (empty or non-empty result) in a relatively short period of time anyway, lowering the importance of elements present in those queries. In general, indexing speeds up every query if a leaf element (on which index is already built) is a subset of path expression of the submitted query. Note that our context identification process optimizes query performance by reducing its search space even if query doesn’t hit the required index.

Another benefit of our context identification technique is the ability to connect the information about the location of data with XQuery and XPath. The query is parsed to determine which nodes need to be involved for processing of the query, avoiding the unnecessary computation by the processing node(s). While the users can operate at a higher abstraction level and are decoupled from the implementation details, such as having to specify the nodes to retrieve the data from.

The optimization in the operation of XPaths’ comparison with full paths in context identification technique is also one of the contributions of the thesis. In addition to the optimization techniques themselves, the thesis also contributes by devising queries and workloads that could be used to benchmark and compare the performance of similar systems, and in general, XML databases.

### 9.2 Future Work

#### 9.2.1 Data Partitioning

- We dealt with data partitioning schemes based on document schema (metadata), while another way of partitioning would be on the basis of document values (data). That’s another area to explore for potential improvement because queries involving filtering of information through predicate or ‘where’ expression require the data pages that lie in the data filtering range. This can be further optimized if data is partitioned based on the value of those elements or attributes that are frequently involved in joining of data. That said, this strategy is challenging to design because at the time of allocating and storing documents, the system doesn’t know the syntax and nature of queries that would be asked of the system. This complexity can be eased out by incorporating the information of unique IDs into the schema because usually join operations occur on these unique IDs and choosing them as the partition keys. A unique ID is an element or an attribute whose value must be unique in an XML document. This property facilitates in identifying a document conveniently, and suggests its importance among other elements and attributes of the document. It is therefore a great candidate to serve as the joining attribute for the purposes of combining information from multiple XML documents of different schema.
• We used the value of 0.5 as the threshold of similarity coefficient. It would be interesting to characterize the queries that would gain better performance by varying the level of this threshold.
• The analysis in this thesis assumes write-once documents, i.e., that they’re never updated in place. The impact of updates on the performance of the proposed methods would make for an interesting study.
• Characterize the minimum query load at which the proposed “smart” document allocation scheme gives significant benefits, particularly in terms of the query response time, over the naïve alternatives, such as a random distribution of documents over the available nodes.
• What happens when a new schema is allocated such that the node that’s the best fit for the schema is already full? Would it be worthwhile to relocate existing schemas in such a case?

9.2.2 Query Context Identification
• Cross-node join queries may contain multiple sub-queries or multiple XPaths whose data is located on different nodes, which makes them interdependent on each other. The result of a sub-query is required to be calculated before retrieving data of other sub-queries so that the final result of the whole query can be calculated. The results of different sub-queries need to be placed together on one node so that they can be joined. If data of different sub-queries or XPaths are independent of each other then further optimization can be achieved by determining such XPaths that have smaller set of result and then transfer their result to the other node where the final result of the query is computed.
• The cost function currently doesn’t account for the cost of inter-node transfer of data. A large result set can take a relatively long time in getting transmitted between nodes, even with high speed networks. This is somewhat related to the earlier point about cross-node joins.
• A larger variety of queries or query forms could be included in the context identification process. This could include some additional XQuery constructs like sorting (ascending and descending) and other aggregate functions.


8. Gao, Shudi (Sandy), Sperberg-McQueen, C. M. and Thompson, Henry S. *W3C XML Schema Definition Language (XSD) 1.1 Part 1*.


<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>SGML</td>
<td>Standard Generalized Markup Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language</td>
</tr>
</tbody>
</table>
Appendix A: Glossary

**Document Object Model**: It is a cross-platform language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents.

**Inter-query Parallelism**: Inter-query parallelism means receiving queries from multiple applications at the same time which are run independently.

**Intra-query Parallelism**: Intra-query parallelism means the simultaneous processing of parts of a single query.

**Query Context Identification**: It means the identifications of processing node(s) and libraries where the required data of the query is hosted.

**Schema Similarity Coefficient**: Schema similarity coefficient varies from 0 – 1, and it is equal to the ratio of common path expressions between two schemas and total path expressions satisifiable in each of them.

**xDB**: xDB is a native XML database

**XPath Satisfiability Problem**: The determination of whether the answer of an XPath is nonempty on an XML document.
Appendix B: xDB Storage Architecture

Figure 28: xDB (Physical Structure)
Figure 29: xDB (Logical Structure)
Figure 30: xDB (Embedded Architecture)

Figure 31: xDB Client (Client-Server Architecture)
Figure 32: xDB Server (Client-Server Architecture)
Appendix C: xDB Performance

**Figure 33:** Execution Time of XQuery

**Figure 34:** Speed of Data Update
### Appendix D: Indexes in xDB

<table>
<thead>
<tr>
<th>Index</th>
<th>xpath</th>
<th>xpointer</th>
<th>xquery</th>
<th>api</th>
<th>live</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library Id index</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Library Name index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Id Attribute indexes</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Element Name index</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Element Value index</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Attribute Value index</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Named elements by attribute value index</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Path indexes</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Full Text indexes</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Metadata indexes</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Multi-path indexes (new in xDB 10.0)</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

*Table 7: Characteristics of xDB’s Different Indexes*
Appendix E: Set of Schemas

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="author">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="book_titles"/>
        <xs:element ref="genre"/>
        <xs:element ref="education"/>
        <xs:element ref="dob"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="firstName"/>
        <xs:element ref="lastName"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="book_titles">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="bookTitle"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element ref="firstName" type="xs:string"/>
  <xs:element ref="lastName" type="xs:string"/>
  <xs:element name="bookTitle" type="xs:string"/>
  <xs:element name="genre" type="xs:string"/>
  <xs:element name="education" type="xs:string"/>
  <xs:element name="dob" type="xs:string"/>
</xs:schema>
```

Figure 35: Schema (Author)
Figure 36: Schema (Book)
Figure 37: Schema (Movie)
Figure 38: Schema (Publisher)
Figure 39: Schema (TVShowInfo)