LEVEL BASED LABELING SCHEME FOR EXTENSIBLE MARKUP LANGUAGE (XML) DATA PROCESSING

A Thesis Submitted to
The Graduate School of Engineering and Sciences of
İzmir Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Beray ATICI

September 2010
İZMİR

We approve the thesis of **Beray ATICI**

Assist. Prof. Dr. Belgin ERGENÇ
Supervisor

Assist. Prof. Dr. Tolga AYAV
Committee Member

Assist. Prof. Dr. Murat Osman ÜNALIR
Committee Member

23 September 2010

Prof. Dr. Sıtkı AYTAÇ
Head of the Department
of Computer Engineering

Assoc. Prof. Dr. Talat YALÇIN
Dean of the Graduate School of
Engineering and Science

ACKNOWLEDGEMENTS

I would like to thank all those people who have through their support enabled me to complete this thesis. Firstly, I would like to thank my supervisor Assist. Prof. Dr. Belgin Ergenç for her guidance and especially her distance support throughout the course of this thesis.

I am thankful to my ex-colleague Res. Assist. Barış YILDIZ for all his help and guidance throughout my studies.

Special thanks to my friend Onur Can ULAŞ for all his vocabulary support, patience and many helpful suggestions.

Last but not least, I wish to express my gratitude to my family for their encouragement support and patience during my studies.

ABSTRACT

LEVEL BASED LABELING SCHEME FOR EXTENSIBLE MARKUP LANGUAGE (XML) DATA PROCESSING

With the continuous growth of data in businesses and the increasing demand for reaching that data immediately, raised the need of having real time data warehouses. In order to provide such a system, the ETL mechanism will need to be very efficient on updating data. From the literature surveys, it has been observed that there are many studies performed on efficient update of the relational data, while there is limited amount of study on updating the XML data.

With the extensible structure and effective performance on data exchange, the usage of XML data structure is increasing day by day. Like relational databases, real time XML databases also need to be updated continuously. The hierarchic characteristic of XML required the usage of tree representations for indexing the data since they provide necessary means to capture different relationships between the nodes.

The principal purpose of this study is to define and compare algorithms which label the XML tree with an effective update mechanism. Proposed labeling algorithms aim to provide a mechanism to query and update the XML data by defining all relations between the nodes. In the experimental evaluation part of this thesis, all algorithms is examined and tested with an existing labeling algorithm.

ÖZET

GENİŞLETİLEBİLİR İŞARETLEME DİLİ (XML) VERİ İŞLEMEDE KULLANILACAK SEVİYE TABANLI ETİKETLEME ÇERÇEVESİ

Gittikçe artan veri miktarı ve kurumların karar alma sistemlerinde güncel veriye sahip olmak istemeleri gerçek-zamanlı veri ambarları ihtiyacını doğurmuştur. Gerçek-zamanlı veriambarları sayesinde değişen verinin anında veri ambarlarına gönderilmesiyle yığınsal yüklemelerden kurtulma planlanmıştır. Bu konuda ise en büyük yük ETL işlemine düşmektedir. Güncellenen veriyi anında hedef veriambarına yüklemek için iyi performanslı ETL gerekmektedir. Bu tez kapsamında yürütülen çalışmalar sırasında; XML yapıdaki verinin aktarılmasında kullanılan ETL işlemi üzerinde odaklanılmıştır.

XML veri yapısı, esnetilebilir oluşu ve veri aktarımı açısından sağladığı avantajlar sayesinde birçok uygulama tarafından kullanımı hızla artmaktadır. Günümüzde özellikle web tabanlı uygulamalar kullanan birçok kurum, organizasyonel verisini XML biçiminde saklamaya başlamıştır. Bu tipteki verinin de ilişkisel ortamlarda olduğu gibi güncellenme gereksinimi bulunmaktadır. XML verinin hiyerarşik oluşu ve ağaç yapısında tutulabiliyor olması XML datanın güncellenmesi konusunda yürütülen çalışmalara ilişkisel veri tabanlarında olduğundan daha farklı bir boyut kazandırmıştır. Ağaç üzerindeki her bir düğümün birbirleriyle olan hiyerarşik ilişkilerini tutarak istenilen veriye çok daha hızlı erişebilme yolları bulunmuştur.

Bu çalışmada XML ağaç yapısını etiketlemede veri güncellemesini etkin olarak yapabilen algoritmalar sunulmuştur. Önerilen algoritmalar, XML ağaç yapısında bulunan veriyi, düğümler arasındaki ilişkilerin bulunabileceği şekilde etiketleyerek, istenen veriye daha kolay ulaşabilmeyi hedeflemektedir. Çalışma sonunda, önerilen algoritmaların başarımları mevcut bir etiketleme algoritması ile karşılaştırılarak yapılmıştır.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

With the use of data warehouse concept in business area, researches were started to develop different ETL (extract-transform-load) methods. ETL has been providing to load the of the organizations transactional data to data warehouses with daily, weekly or monthly bulk loads. But the increasing amount of data was a big bottleneck for these bulk loads. Both the required storage and the time during the loads were increasing. On the other hand, the organizations were looking for ways to detect business events in production systems as they happen because they use those events to trigger a response in another system. This introduced real time (Baer, 2004). Besides, the internet platform was started to search a data structure which could keep up with its continuous development

The lack of structured data over web was a bottleneck on accessing the valuable information. As a solution to this problem, a new data format called eXtensible Markup Language (XML) was developed (W3C). XML is a semi-structured data format for information exchange over World Wide Web. The structure of XML makes it usable as a semantic preserving data exchange format on the web. With the internet's broader use within time, it became a global data exchange platform and the interest in XML has grown.

For many years, there was a problem of the enterprises while they were trying to extract the useful, concise and handy information from the entire data stored in their complex information systems. After the use of internet in their business and communication channels, the data changed its format into XML. Thus, the importance of integrating XML data to data warehouse environment is becoming increasingly higher. Now, some data warehousing and ETL tools support extraction of XML data from source to feed the warehouse.

Trying to update the XML data is an important problem while the XML data warehouse wanted to be kept up-to-date. To resolve these issues, many researches are going on. One of the solutions is about effectively labeling XML data trees. Labels define the type of relationships presented among nodes and are important blocks for

structural join algorithms and important complements of structural indexes. Choosing a suitable labeling scheme requires different factors to be taken into consideration which are storage, nature of data, query type and efficiency of maintaining that labeling scheme. For a document tree, a labeling scheme is a structural summary of a specific set of tree relations. Each node in the tree is assigned a typically unique node label, so that any of these relations between the nodes can be inferred from their labels.

A labeling scheme supporting dynamic XML data should be able to keep computational cost of labeling, label size and required re-labeling with inserts and deletes at minimum while providing several relationships. Briefly; a labeling algorithm should have quick indexing and easy retrieval while using minimum space for labels.

To supply these requirements, many labeling schemes have been proposed. XML labeling approaches can be classified into two categories. The first one is prefix based labeling algorithms (Sans et al., 2008). The algorithms in this group label a node with using its parent's label as prefix of its label. This property of this group causes a bottleneck on the size of labels. The second category is range based approaches (Ko et al., 2006, Zhang et al., 2001). This group of schemes generally focus on the position of the node in the XML document and they always re-generate the labels for the XML tree. This has a degrading effect on the performance in an update intensive environment.

In this thesis project a labeling scheme called Level-Based-Labeling (LBL) is proposed in three versions. All versions aim to determine four basic relationships; Parent-Child (P-C), Ancestor-Descendant (A-D), sibling and ordering relationships; among nodes while requiring inexpensive computation for construction of a label, minimum re-ordering with inserts and deletes and reasonable label length with increasing level and fan-outs. Each version is good at some of these requirements. The first version, Basic LBL, aims to have minimum label size with keeping the all the nodes in order. This ordering the data issue causes a performance bottleneck on updating data. To handle this issue second version, Single Linked LBL, and third version, Double Linked LBL, aims to avoid relabeling the nodes after each insert, but this increases the size of the labels. Besides, these two versions are efficient on determining the relations between nodes. At this point Double Linked LBL has much better performance on determining the backward ordering relation than Single Linked LBL.

Experimental study of this project consists of 4 different test cases which compare the performance of all versions of the LBL with another labeling scheme;

containment (Zhang et al., 2001). Containment is chosen since it is popular and similar to LBL with its node structure.

The remainder of the report is organized as follows. Chapter 2 gives some background information on data warehousing and ETL structures on both relational and hierarchical data types. Chapter 3 discusses the related work on XML labeling. Chapter 4 proposes new labeling scheme, called Level-Based XML labeling scheme with three versions. Chapter 5 contains the detailed performance study and analysis comparing the test results of the three version of Level Based labeling scheme. Chapter 6 sums up the work and discusses the results reached.

CHAPTER 2

BACKGROUND INFORMATION

2.1. Introduction

Recent advances in computing, communications, and digital storage technologies, together with the development of high-throughput data-acquisition technologies, have made it necessary to gather and store incredible volumes of data. At this point, data integration come into picture which is the process of combining data residing at different sources and providing the user with a unified view of these data.

There are many approaches to combine distributed data and access them through a unified view. One of these approaches is virtual data integration. This integration technique provides an access to distributed data over a mediated schema with a query interface and do not replace data physically. Another data integration approach is data warehousing which provides extracting the historical data from operational databases and loading them to data warehouses.

By time, the business needs grew, data volumes in operational data stores, such as online transaction data, inventory data, and customer information became greater in size. The larger the data volumes become, the more resources and time are required by the ETL processes. Also the standard architecture for a traditional data warehouse is based on periodic batch extracts from the source data, which then flows through the system. Reporting was done from warehouses which were updated on a daily or weekly basis. When the real-time nature of the data warehouse load becomes sufficiently urgent, the batch approach breaks down.

Real time business intelligence is the process of delivering information about business operations without any latency. Real time means delivering information in a range from milliseconds to a few seconds after the business event. While traditional business intelligence presents historical information to users for analysis, real time business intelligence compares current business events with historical patterns to detect problems or opportunities automatically.

At the same time XML data type is rapidly becoming a widely used data format. Soon, it can be expected that large volumes of XML data will exist. Large amount of data needed for decision-making processes are stored in the XML data format, which is widely used for e-commerce and internet-based information exchange. Thus, as more organizations view the web as an integral part of their communication and business, the importance of integrating XML data into data warehousing environments become increasingly higher by time.

In all the real time business approaches, an important point which is the design model of the data sources is escaped from observation. They mostly are deliberated as relational database systems but rapidly increasing usage of XML raised a need to be integrated in an ODS system too. When situation is concerned, many researches has been started to try to find out a solution approach on storing and querying XML data. But the storing and querying is not enough for a fully efficient real time XML data warehouse. The main problem is how to design and query such a data mart that carried out starting directly from an XML source with a good performance.

In this chapter the concepts of data integration, data warehousing and ETL will be explained and how XML data is integrated in these systems will be discussed.

2.2. Data Integration

Data integration is a huge topic for IT because ultimately IT aims to make all systems as they are working together. In many cases, serious data integration must take place between the primary transaction systems of the organization and user queries. Generally, this data integration is complete, unless the organization's decision-making systems have settled on a single system, all important enterprise resource planning system and transaction-processing systems settled apart.

Data integration is the process of combining data residing at different sources and providing the user with a unified view of these data. This process emerges in a variety of situations. The increasing data volume and need of sharing existing data on all systems of an enterprise discloses the need of data integration.

The problem of combining heterogeneous data sources under a single query interface is not a new one. The rapid adoption of databases after the 1960s naturally led

to the need to share or merge existing repositories. This merging can be done at several levels in the database architecture.

Two popular approaches for data integration are Virtual Data Integration and data warehousing. The idea behind Virtual Data Integration, which is first mentioned by Wiederhold in 1992 (Wiederhold et al., 1992), is to provide a uniform query interface over a mediated schema. This query is then transformed into specialized queries over the original databases. This process can also be called as view based query answering because we can consider each of the data sources to be a view over the (nonexistent) mediated schema (Figure 2.1). This data integration solution may address to many problems by considering these external resources as materialized views over a virtual mediated schema, resulting in "virtual data integration". Even though there has been much progress in this area, in Kiani et al. (2007) it is realized that more work is required to overcome the challenges such as lacking of a generic model for virtual data integration, availability of the global schema or impossibility of updating through the mediator based integration systems.

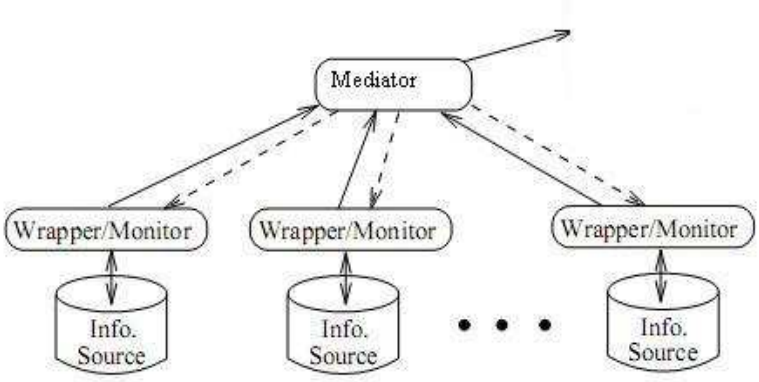


Figure 2.1. The architecture of mediator based information integration system

In Data Warehousing, data from several sources are extracted, transformed, and loaded into a common source and can be queried with a single schema. This idea was first mentioned in 1988, by IBM researchers Barry Devlin and Paul Murphy (Devlin et al., 1988). This can be perceived architecturally as a tightly coupled approach because the data reside together in a single repository at query time.

A data warehouse is an integrated collection of aggregated, historical data from internal and external sources grouped into a common subject matter, such as a business area or business function. As seen on Figure2.2, data from various operational

applications and other sources are selectively extracted and organized on the data warehouse database. The data warehouse then becomes a source for use by analytic applications and user queries.

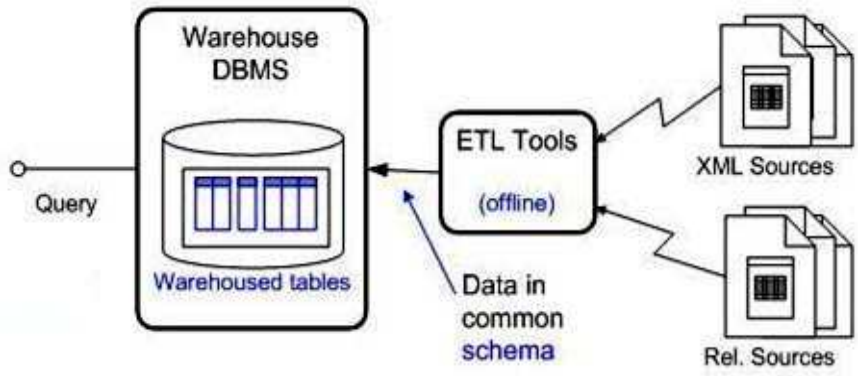


Figure 2.2. The architecture of data warehouse system

A data warehouse provides a common data model for all data of interest, regardless of the data's source. This makes it much easier to report and analyze information rather than using multiple data models from disparate sources in order to retrieve information such as sales invoices, order receipts, general ledger charges, etc.

2.2.1. Real-Time Data Warehouses

Traditionally data warehouses do not contain up-to-date data. They are usually loaded with data from operational systems at most weekly or in some cases nightly, but are in any case a snapshot of the past. As today's decisions in the business world become more real-time, the systems that support those decisions need to keep up. It is only natural that Data Warehouse, Business Intelligence, Decision Support, and OLAP systems quickly begin to incorporate real-time data.

A strict definition of real-time in (Baer, 2004) implies that any data change occurring in a source system is automatically and instantaneously reflected in the data warehouse. This would mean that all changes in the data warehousing environment take place simultaneously with the change in the source system – something that is only achievable when both changes are part of the same atomic transaction. Every mechanism that does not adhere to this rule is in reality only ‘near real-time’ and always

shows some delay between the source system's transaction and the equivalent entries in the data warehousing system.

2.2.2. Need for Real-Time Data Warehouses

ETL (extract, transform and load) is the process that enterprises use to build the consolidated data stores required for effective Business Intelligence (BI). Traditionally, ETL processes have been run periodically, on a daily, weekly or monthly basis, and use a bulk approach that moves and integrates the entire data set from the operational source systems to the target data warehouse. While this approach was acceptable for enterprises over the years, current business conditions require a new way of integrating data - in real time and in an efficient manner.

The demand for real-time integration explained in (Ankorion , 2005) and (IBM , 2008) as;

- **Business globalization and 24x7 operations.** In the past, enterprises could stop online systems during the night or weekend, to provide a window of time for running bulk ETL processes. Today, running a global business with 24x7 operations means smaller or no downtime windows.
- **Need for up-to-date, current data.** Customer demand, competitive pressure and improved decisions require timely information. To make the most of BI in today's ever-accelerating business climate, managers should not be working with last week's or yesterday's data. Today, decision-makers need data that is updated a few times a day or even in real time.
- **Data volumes are increasing.** As time passes and the business grows, data volumes in operational data stores, such as online transaction data, inventory data, and customer information, become larger. The larger the data volumes become, the more resources and time are required by the ETL processes. This trend challenges the bulk extract windows that are getting smaller and smaller.
- **Cost reduction.** Bulk ETL operations are costly and inefficient, as they require more processing power, more memory and more network bandwidth. In addition, as bulk ETL processes run for long periods of time, they also require more administration and IT resources to manage.

- **Growing need to detect and react to business events as they happen.** Many organizations are looking for ways to detect business events in production systems and have those events trigger a response in another system. For example, a cell phone company would like to send a text message to a customer running low on minutes asking if him if he would like to purchase more.
- **The need to track all changes for auditing purposes.** Organizations need to comply with regulations, which often require them to continuously track all changes to data and not just the net result of those changes.
- **Increasing need to keep data in sync across the enterprise.** Customers want up-to-the-minute access to order, payment and inventory data so they can buy products, pay bills and check delivery status online. Employees need much of the same so they can better service customers and make wise business decisions. To accomplish this, eCommerce data needs to be in sync with business applications and data needs to flow in real-time across the enterprise.

2.2.3. Data Warehouse Structure

The main approach for storing data in a data warehouse is the dimensional approach. In a dimensional approach, transaction data are partitioned into either "facts", which are generally numeric transaction data, or "dimensions", which are the reference information that gives context to the facts. The facts are the measurement processes. A measurement is a real-world observation of a previously unknown value. Measurements are overwhelmingly numeric, and most measurements can be repeated over time, creating a time series. A single measurement creates a single fact table record. Conversely, a single fact table record corresponds to a specific measurement event on dimension tables (Kimball et al., 2004). In Figure 2.3, illustrates this structure of data warehouses.

A fact table (WarehouseArchitect, 1999) stores variable numerical values related to aspects of a business. For example, sales, revenue, budget. These are usually the values you want to obtain when you carry out a decision support investigation. A fact table is at the intersection of dimension tables in a star schema.

A dimension table (WarehouseArchitect, 1999) stores data related to the axis of investigation of a fact. For example, geography, time, product. A warehouse model can

have any number of dimension tables. Dimension tables are connected to a central fact table. The primary key in the dimension table migrates as a foreign key in the fact table. Dimension tables can also be connected to other dimension tables to form a hierarchy of dimensions.

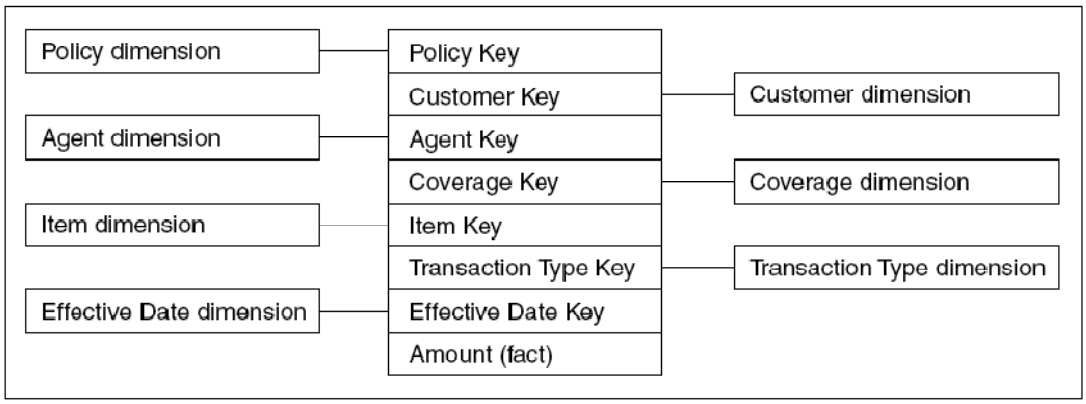


Figure 2 3. An example of a data warehouse model
(Source: WarehouseArchitect, 1999)

With the usage of XML data in business intelligence area and storing it in data warehouses, many studies are going on to form the XML data to load into this dimensional model. Before discussing these approaches, the structure of XML data, the approaches that propose where to store it and how to query it will be explained in the next section.

2.3. XML Data Management

In response to the need for a more powerful language for modeling web information, the eXtensible Markup Language, XML, was proposed by the World Wide Web Consortium (W3C), in 1997. XML is a subset of Standard Generation Markup Language (SGML) which is the meta-language of HTML simplified upon the requirements of web applications.

XML is a hierarchical data format for information exchange in the World Wide Web. An XML document consists of nested element structures, starting with a root element. Element data can be in the form of attributes or sub-elements. Figure 2.4 shows an XML document that contains information about a book. In this example in (Shanmugasundaram et al., 1999), there is a book element that has two sub-elements,

booktitle and author. The author element has an id attribute with value “Dawkins” and is further nested to provide name and address information.

```
<book>
   <booktitle> The Selfish Gene </booktitle>
   <author id = “dawkins”>
            <name>
                <firstname> Richard </firstname>
                <lastname> Dawkins </lastname>
            </name>
            <address>
                <city> Timbuktu </city>
                <zip> 99999 </zip>
            </address>
   </author>
</book>
```

Figure 2.4. An example of XML document
(Source: Shanmugasundaram et al., 1999)

2.3.1. How are XML Data Stored?

In general, numerous different options to store and query XML data exist. In addition to a relational database, XML data can also be stored in a file system, in an object-oriented database or in a special-purpose (or semi-structured) system. It is still unclear which of these options will ultimately find wide-spread acceptance.

Text Files

A file system could be used with very little effort to store XML data, but a file system would not provide support for querying the XML data (Florescu et al., 1999).

Native XML Databases

As defined by the XML DB consortium, the formal definition of a Native XML Database (Nicola et al., 2005) states that; a Native XML Database defines a (logical) model for an XML document and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Native XML Database has an XML document as its fundamental unit

of logical storage, just as a relational database has a row in a table as its fundamental unit of logical storage.

"Native" means that XML documents are stored on disk pages in tree structures matching the XML data model. This avoids the mapping between XML and relational structures, and the corresponding limitations.

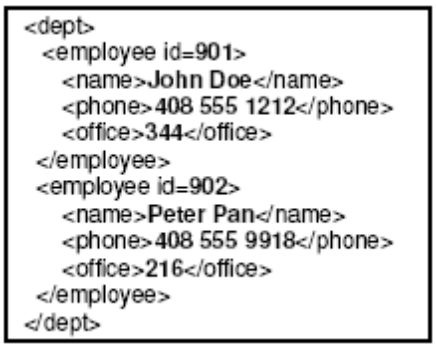


Figure 2.5. An XML document
(Source: Nicola et al., 2005)

To insert XML data into the database, client applications send XML documents in their textual representation to the server. The server uses a parser to check incoming documents for wellformedness and to perform optional validation. The parser events are converted into a hierarchical representation of the XML document. For the sample document in Figure 2.5, this hierarchy looks similar to the document tree in the upper part of Figure 2.6.

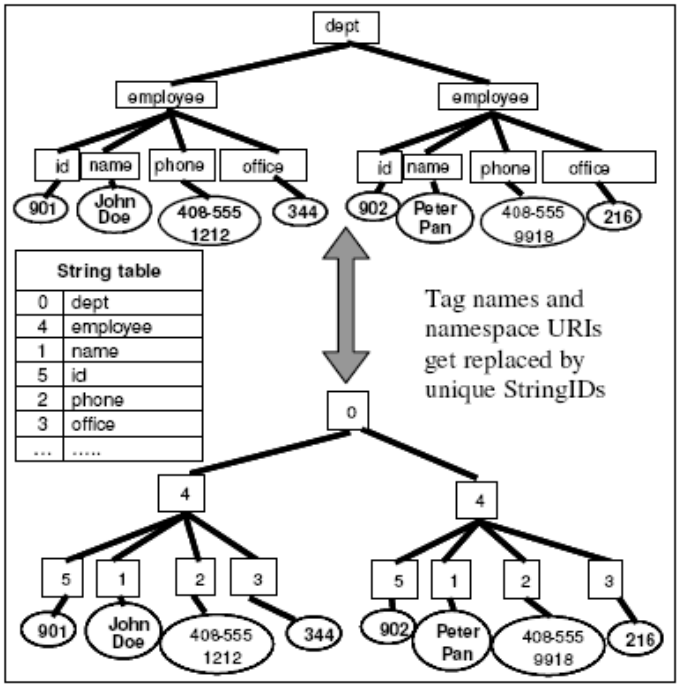


Figure 2.6. An XML document tree
(Source: Nicola et al., 2005)

Converting to different data types

There are various ways to solve the problem of effective, automatic conversion of XML data into and out of relational databases. Database vendors such as IBM, Microsoft, Oracle, and Sybase have developed tools to assist in converting XML documents into relational tables.

Generally XML document elements are modeled as a collection of nested tables or they are stored either as BLOB-like objects or as decomposed into a set of tables. Some vendors use a data type as OPENXML which is also used by Microsoft.

Mapping XML files into RDBMS tables

When using an RDBMS, there are many different ways to store XML data (Florescu et al., 1999).

One option is to infer from the DTDs of the XML documents how the XML elements should be mapped into tables. It is indeed possible to use standard commercial relational database systems to evaluate powerful queries over XML documents. The key that makes this possible is the existence of Document Type Descriptors (DTDs) or an equivalent, such as DCDs or XML Schemas. A DTD is in effect a schema for a set of XML documents. Without DTDs or their equivalent, XML will never reach its full

potential, because a tagged document is not very useful without some agreement among inter-operating applications as to what the tags mean. To apply this approach to query XML documents; first a DTD is processed to generate a relational schema. Second, XML documents are parsed for conforming to DTDs and loading them into tuples of relational tables in a standard commercial DBMS. Third, semi-structured queries over XML documents are translated into SQL queries over the corresponding relational data.

Another option is to analyze the XML data and the expected query workload. This technique is for using a RDBMS to store, query and manage semi-structured data. Semi-structured data can always be stored as a ternary relation, since the data is an edge-labeled graph, but this is no better than storing the schema with the data. Instead, this technique relies on an aggressive mapping from the semi-structured data model to the relational model.

This technique can be used 1) to store and manage efficiently existing semi-structured data sources, and 2) to convert relational sources into a semi-structured format, such as XML.

2.3.2. How are XML Data Queried?

Many XML querying languages are used to query data from different XML storage platforms.

XSLT

The Extensible Stylesheet Language for transformation is an official recommendation of the World Wide Web Consortium which published in 1999. It provides a flexible, powerful language for transforming XML files and uses XML syntax to define transformation rules that are applied to an input XML document to result in a text document that has not to be an XML document. This result can be an HTML document, another XML file, PDF, SVG, java code or a text file (DuCharme, 2001).

XSU

Oracle created the XML SQL (XSU) Java API to convert XML to SQL, and vice versa. Before such mapping is performed, the table, to which the XML document will

be mapped, must be created. XSU maps XML elements to specified database table columns.

XPath

XPath is a language for addressing parts of an XML document which is a standard recommended by W3C. XPath defines a library of standard functions but is not itself written in XML because it defines how to locate parts of an XML document, forms the basis for a query language on XML, such as XSLT or Xquery. XPath models an XML document as a tree of nodes of which there are different types, including element nodes, attribute nodes and text nodes.

XQuery

XQuery is currently still under development by the W3C (XQuery 1.0), and is also known as W3C XML Query. The purpose of XQuery is extracting data from entire XML documents, collections of XML documents, or only document fragments. XQuery is derived from an XML query language called Quilt, which in turn borrowed features from several other languages, including XPath 1.0, XQL, XML-QL, SQL, and OQL. XQuery 1.0 is the superset of XPath 2.0 both in syntax and semantics.

With the growing importance of XML documents as a mean to represent data in the World Wide Web, there has been a lot of effort on devising new technologies to process queries over XML documents. The major relational database systems have been providing XML support for several years, predominantly by mapping XML to existing concepts such as LOBs (Large Object Data) or (object) relational tables. This causes limitations with these approaches in research and industry as functional constraints and performance constraints. Generally, storing XML data as large objects allows for fast insert and retrieval of full documents but suffers from poor search and extract performance due to XML parsing at query execution time. Because of these drawbacks, many researches are proposing to store XML data in native XML databases, query this data with appropriate query language. In decision supporting systems XML data is queried with that appropriate language and load into XML data warehouses. The structure of these warehouses is the same as the dimensional data warehouses', consist of facts and dimensions. But XML data which will be loaded to data warehouse is processed slightly different than relational data. The definition of its facts and

dimensions is formed from the DTD graph of the XML data. The nodes are extracted and loaded to target warehouse according to this form as an XML tree.

The next section will describe the ETL process and its steps which relational and hierarchic data take.

2.4. ETL with Relational Data

The back room and the front room of the data warehouse are physically, logically, and administratively separate. In other words, in most cases the operational source database and analytic target database are on different machines, depend on different data structures. Preparing the data, often called data management, involves acquiring data and transforming it into information.

There are four staging steps found in almost every ETL process of a data warehouse. A properly designed ETL system extracts data from the source systems, enforces data quality and consistency standards, conforms data so that separate sources can be used together, and finally delivers data in a presentation-ready format so that application developers can build applications and end users can make decisions.

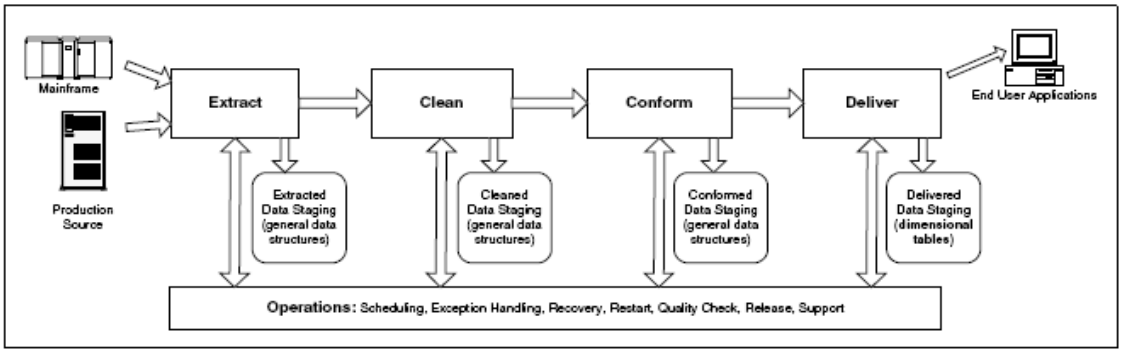


Figure 2.7. The four staging steps of a data warehouse

2.5. ETL with Hierarchical Data

XML data sets are not generally used for persistent staging in ETL systems. Rather, they are a very common format for both input to and output from the ETL system. The hierarchical capabilities of XML need a different method to be integrated more deeply with the data warehouse queryable tables.

Figure 2.8. A XML document
(Source: Golfarelli et al., 2001)

A DTD (Document Type Definition) defines elements and attributes and the nesting and occurrences of each element in an XML document. A document Type Declaration defines the constraints on the sequence and nesting of element tag and attributes. There are four kinds of declarations in DTD; elements, attribute lists, entities and notations.

```
<!DOCTYPE webTraffic [
   <!ELEMENT webTraffic (click*)>
   <!ELEMENT click (host, date, time, url)>
   <!ELEMENT host (category | nation)>
   <!ATTLIST host
      hostId ID #REQUIRED>
   <!ELEMENT category (#PCDATA)>
   <!ELEMENT date (#PCDATA)>
   <!ELEMENT time (#PCDATA)>
   <!ELEMENT url (site, fileType,
urlCategory+)>
   <!ATTLIST url
      urlId ID #REQUIRED>
   <!ELEMENT site (nation)>
   <!ATTLIST site
      siteId ID #REQUIRED>
   <!ELEMENT nation (#PCDATA)>
   <!ELEMENT fileType (#PCDATA)>
   <!ELEMENT urlCategory (#PCDATA)>
]>
```

Figure 2.9. A DTD document
(Source: Golfarelli et al., 2001)

Semi-automatic approach (Golfarelli et al., 2001) can be used for building the conceptual schema of a data mart starting from the XML sources. Starting with the assumption that the XML document has a DTD and conforms to it, the methodology consists of the following steps:

1. Simplifying the DTD
2. Creating a DTD graph
3. Choosing facts
4. For each fact:
    4.1 Building the attribute tree from the DTD graph
    4.2 Rearranging the attribute tree
    4.3 Defining dimensions and measures

These steps are applied to a XML document as following. First the DTD of the XML document should be simplified because it may have been declared in a complicated and redundant way. After simplifying the DTD, a DTD graph representing its structure can be created. Its vertices correspond to elements, attributes and operators in the DTD. Attributes and sub-elements are not distinguished in the graph since, they are considered as equivalent nesting mechanisms. Then one or more vertices of the DTD graph are chosen as facts; each of them becomes the root of a fact schema. Some further arrangements should be made to the attribute tree which is derived from the DTD graph. And finally to normalize a star schema of XML data warehouse, dimensions and facts are defined.

After ability of loading the XML data in data warehouse, the problem of the need of accessing to the data in real time is raised also in XML warehouses. The main problem is to query and extract the XML data from data source easily and to reflect the data changes to the target at the same time with the update occurs on the source. One of the alternative solutions for these problems is indexing the XML tree with labels. And several researches have been started in this area.

2.6. Conclusion

The chapter's overall goal was to provide an overview of the literature in the areas covering this research: the concept of data warehousing and real time ETL processes, the structure of XML data, XML data warehouses and their ETL systems.

Two popular data integration approaches have been introduced; virtual data integration and data warehousing. The growing need to detect and react to business events as they happen, increasing need to keep data in sync across the enterprise and need for up-to-date, current data increased a demand for real time data warehouses.

Also with the migration of business systems into web, the data changed its form as XML. In this chapter all those data integration methods, data warehouse structures and real time demand is explained to prepare a background for the rest of this project. The background information has been given because the proposed XML labeling algorithm is recommended to use in all XML updating systems including the ETL system of XML data warehouses.

In this master thesis, all these XML labeling approaches which focused on to supply a better update performance on XML data are studied. In the next chapter these approaches will be explained briefly and their advantages and disadvantages will be discussed.

CHAPTER 3

XML LABELING SCHEMES

3.1. Introduction

The extensible structure of XML data makes itself a popular data format on many data interchange areas. With the usage of XML data type in several areas such as decision making processes, e-commerce and internet based information exchange; and storing XML data in several platforms as text files, relational databases, object oriented databases or special purpose systems have brought performance problems on querying and updating XML data with them. Many researches on this topic find out that XML data could be kept on a XML tree with labels, and each changed data could be transferred with its label to the target database.

For a document tree, a labeling scheme is a structural summary of a specific set of tree relations. Each node in the tree is assigned a typically unique node label, so that any of these relations between the nodes can be inferred from their labels. Edges in XML data trees represent structural relationships between data nodes. The basic relationships to be determined in XML query processing are ancestor-descendant (A-D), parent-child (P-C), sibling and ordering relationships. The main purpose of all schemes is to satisfy all these relationships in order to support effective indexing mechanisms for querying. Each scheme uses different methods to provide them.

Besides, the aim of providing an efficient mechanism for querying XML data, update performance of the labeling scheme with dynamic XML data intensive environments should be considered too. To enhance update performance a group of schemes predefine extra labels for potentially existing nodes. With this property, they aim to never re-order the XML tree. Another group of scheme never allocates extra space but they reorder the XML tree after every update. Several labeling schemes have been introduced to develop an optimized retrieval, since they provide a quick way to determine the type of relationships that are present among the nodes. In this chapter some of these XML labeling schemes will be explained.

3.2. XML Labeling Approaches

There has been a great diversity of labeling schemes. In Su-Cheng et al., (2009) the labeling schemes are classified into 4 categories; sub-tree labeling, prefix-based labeling, multiplicative labeling, hybrid labeling. Xu et al., (2005) analyses labeling schemes with respect to their top-down/bottom-up propagation patterns regarding renumbering when XML updates occur. Besides all, the most popular classification (Sans et al., 2008; Wu et al., 2004; Ko et al., 2006; Lu et al., 2004) divides the labeling algorithms into two categories; prefix based labeling schemes and range (interval) based labeling schemes.

3.2.1. Prefix-Based Labeling Schemes

Prefix-based schemes directly encode the father of a node in a tree, as a prefix of its label using for instance a depth-first tree traversal. In the prefix labeling scheme, the label of a node is that its parent's label concatenates its own label (self_label). For any two nodes u and v, u is an ancestor of v iff label(u) is a prefix of label(v). Node u is a parent of node v iff label(v) has no prefix when removing label(u) from the left side of label(v) (Hu et al., 2006).

Simple Prefix Labeling

Cohen et al., (2002) proposed a prefix based labeling scheme where each label inherits its parents label as prefix of its own label (Figure 3.1). The first child of the root is labeled with "0", the second child with "10", followed by the third and fourth with "110" and "1110" respectively. For any node L(v) denoting the label of v, the first child of v is labeled with L(v) "0", the second child of L(v) "10", and the ith child with L(v) "(1...1)i-10". This labeling scheme does not need to be regenerated for any arbitrarily heavy update such as deletion or addition of nodes or subtree to each right side of a subtree. The limitation of this technique is that the size of simple prefix is often too huge (Su-Cheng et al., 2009).

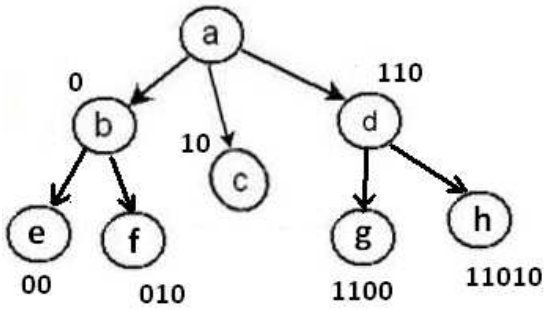


Figure 3.1. Simple Prefix Labeling

Dewey ID Labeling

Dewey ID (Tatarinov et al., 2002) is based on the Dewey Decimal Classification System which is widely used by librarians. The Dewey ID labeling is very similar to tree location address, except that dot separators are present in Dewey ID labeling to differentiate each label inherited from each level of their ancestors (Figure3.2). With Dewey Order, each node is assigned a vector that represents the path from the document's root to the node. Each component of the path represents the local order of an ancestor node. Using this labeling scheme, structural relationships between elements can be determined efficiently (Hu et al., 2006).

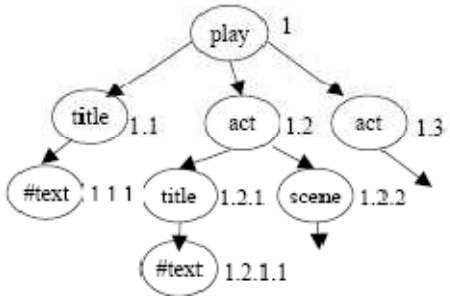


Figure 3.2. Dewey ID Rabeling Scheme

P-Prefix encoding

The main idea to determine the P-C relationship is that the parent index of a node is stored together with the index of this node (similar to P-Containment), called P-Prefix-II.

To facilitate the ancestor-descendant relationship determination, based on P-Prefix-II, the second_prefix_label is indexed for every certain number depth, called

P-Prefix-III index. Based on PPrefix-III index, the A-D relationship can be determined at a higher level firstly, then at a lower level (Li et al., 2005).

P-Prefix-I

To reduce the redundancy of the prefix scheme, the prefix_labels and self_labels are separated, the duplicated prefix_labels are removed and appeared later, and a unique index number (called P-Prefix-I) is given to each unduplicated prefix_label. It gives the sibling determination as, node u is a sibling of node v iff P-PIndexI(u). = P-PIndexI(v), where PPIndexI means the P-Prefix-I index; and ordering determination as, node u is before (after) node v in document order iff 1) PPIndexI(u) < (> resp) P-PIndexI(v); or 2) P-PIndexI(u) = PPIndexI(v) and self_label(u) < (> resp) self_label(v).

P-Prefix-I guarantees that the sibling relationship determination is only one comparison and the ordering relationship determination is at most two comparisons no matter how deep the XML tree is.

P-Prefix-II

The main idea to determine the P-C relationship is that the parent index of a node is stored together with the index of this node, called P-Prefix-II. If the parent index is built on the labels instead of the prefix_labels, the parent-child relationship determination only needs one comparison (in P-Prefix-I). But in that way, the sibling and ordering relationship determinations are expensive when the XML tree is deep.

P-C determination in P-Prefix-II is; node u is a parent of node v iff P-PIndexI(u) = P-PParentIndexI(v) and self_label(u) = second_self_label(v), where P-PParentIndexI means the parent PPrefix-I index.

This property of the scheme guarantees that the P-C determination is only two comparisons no matter how deep the XML tree is and the sibling and ordering determinations are still the same as P-Prefix-I.

P-Prefix-III

To facilitate the ancestor-descendant relationship determination, based on P-Prefix-II, every certain number depth is indexed with a second prefix label called P-Prefix-III index. Based on P-Prefix-III index, the A-D relationship can be determined at a higher level firstly, then at a lower level.

ORDPATH Labeling

ORDPATH (O'Neil et al., 2004) encodes the P-C relationship by extending the parent's ORDPATH label with a component for the child. The main difference between ORDPATH and Dewey ID is that even numbers are reserved for further node insertions in ORDPATH (Su-Cheng et al., 2009). So it supports insert/update efficiently without changing any existing label. It is not suitable for deep trees because the label's length scales up quadratically as number of fan-out and level increases. The labels are shown in Figure 3.3.

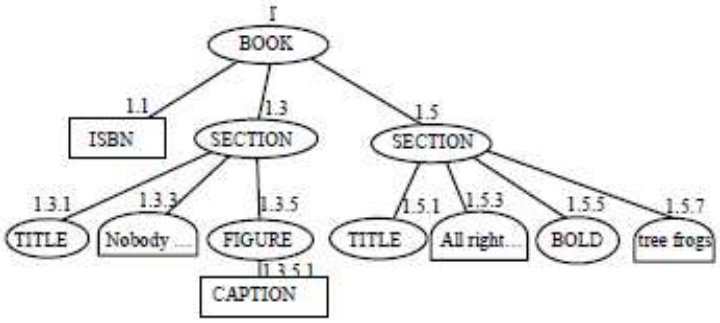


Figure 3.3. ORDPATH labeling

Binary String Based Labeling Scheme

A binary string based labeling scheme (Ko et al., 2006) is developed by the need to efficiently support queries and updates in ordered XML trees. The proposed scheme can allocate any number of new labels without modifying already allocated labels, and does not need to re-label existing nodes or re-calculate any values when inserting order-sensitive nodes into the XML tree. It illustrated in Figure 3.4.

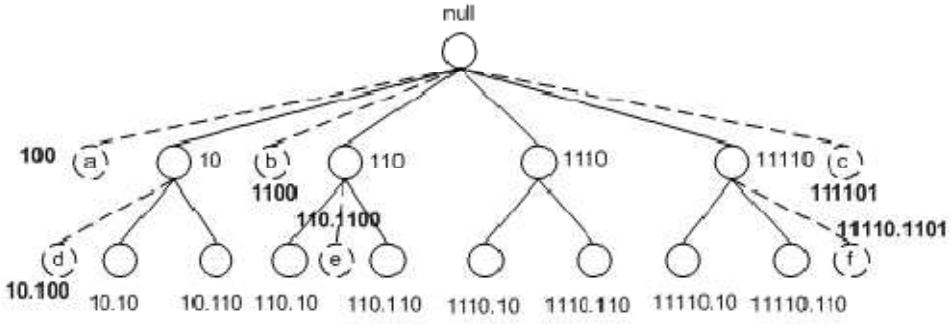


Figure 3.4. Order-sensitive update of binary string based scheme
(Source: Ko et al., 2006)

In prefix based labeling schemes, the nodes inherit their parents' labels as the prefix to their own labels. New nodes can be inserted without affecting thelabels of the existing nodes. Also this allows one to determine the existence of an ancestor-descendant relationship by simply examining whether the prefix relationship exists in the labels of the two nodes. By the way, it has a drawback on the XML trees which have large depth size and fan-out size. With the growth of the tree the labels reach to uncontrollable sizes.

3.2.2. Range Based Labeling Schemes

In range (interval) based labeling schemes, a depth-first traversal of the XML tree is carried out to assign to each node a pair of values that cover the range of values in the labels of its descendant nodes (Ko et al., 2006). These kind of labeling schemes require re-labeling of the entire XML tree when frequent insertions and deletions of nodes occur.

Tree Location Address Labeling

In this approach, each identifier of an ancestor node is a prefix of its descendant (Figure 3.5). A node id (nid) is the concatenation of the nid through the path from the root to the respective node. For example, node 1112 means the second child of the first child of the first child of the root. With this, a P-C relationship can be easily detected. However using this method requires variable space to store the identifiers (Su-Cheng et al., 2009).

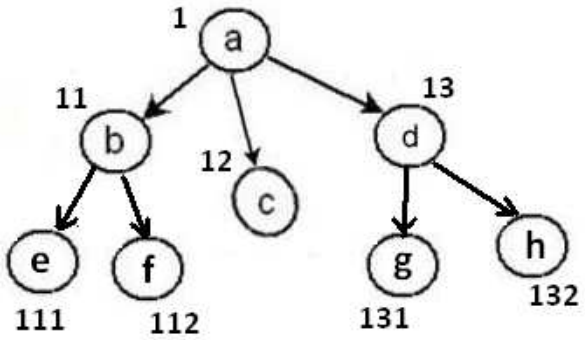


Figure 3.5. Tree Location Address Labeling

Extended Pre-order Traversal

In this approach (Li et al., 2001), each node in the XML tree is labeled with a pair of numbers <order, size> (Figure 3.6). A global reordering is necessary when all the reserved space have been consumed. Moreover it is not clear how one can assign a large enough value for “size”.

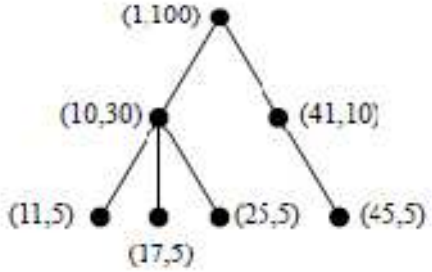


Figure 3.6. Extended Pre-order Traversal

Interval Encoding

The earliest labeling scheme proposed is Interval Encoding (Santoro et al. 1985) which is a well-known technique for post order traversal and numbering of rooted trees.

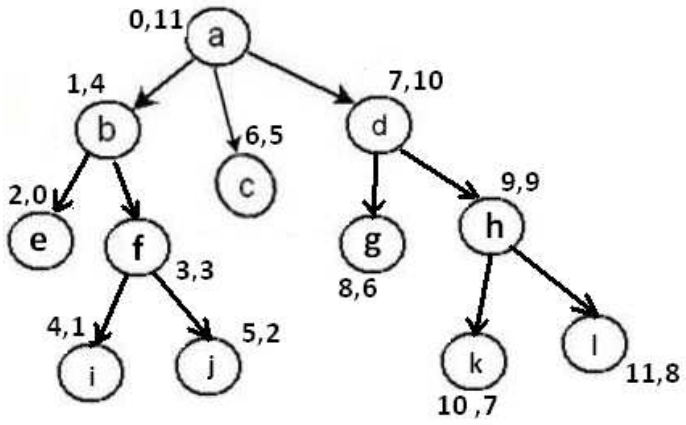


Figure 3.7. Tree Traversal Encoding

One kind of this interval encoding method is Tree Traversal encoding. In this scheme, each node is relabeled with a pair of unique integers consisting of preorder and postorder traversal sequences (in Figure 3.7).

K-ary Complete Tree Scheme

K-ary Complete Tree scheme was introduced in (Lee et al., 1996) which uses tree traversal order to determine the ancestor-descendant relationship between any pair of nodes in the tree. As reported in (Li et al., 2001) and (Xu et al., 2005), one problem of this scheme is that it requires a large numbering space when the k-ary and height of the complete tree are getting large.

This technique enumerates nodes using a k-ary tree, where k is the maximal fan-out of nodes. Here, each internal node is supposed to have the same number of fan-out k. Thus, virtual nodes are created to balance the number of fan-outs. Starting from each level, each node is assigned a label starting with integer 1 from top to bottom and from left to right as depicted in.

The virtual nodes created are shown only until level 3, due to space constraint. In other labeling schemes two already known identifiers are used to determine the parent-child relationships, where the UID technique has an interesting property for the parent node to be determined, based on the identifier of the child node. Given a node having the identifier i, the parent id can compute as in equation (1):

$$\text{parent}(i) = \frac{(i-2)}{k} + 1 \qquad (1)$$

Nevertheless, the major drawbacks are explained in (Li et al., 2001) and (Su-Cheng et al., 2009) for this labeling scheme. Firstly, this labeling scheme requires recursive queries to retrieve A-D relationship. Secondly, when a new node is inserted, a

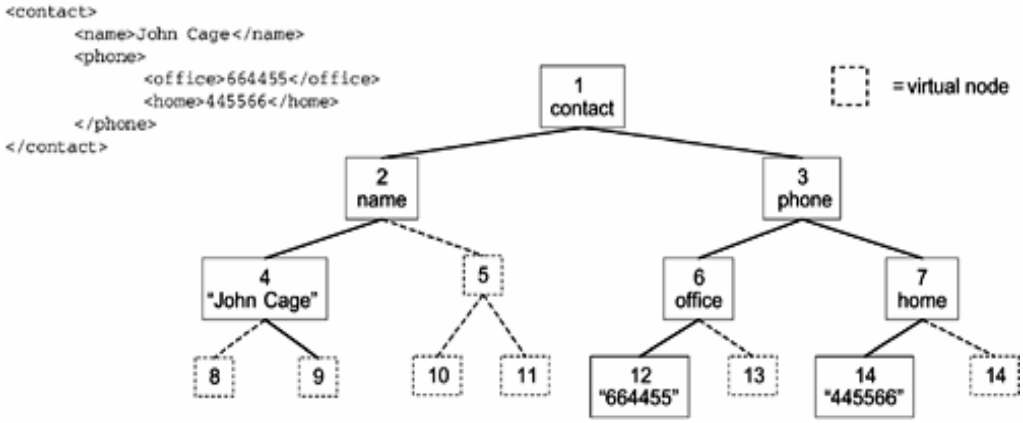


Figure 3.8. K-ary complete tree labeling scheme

new identifier must be assigned to the node. Due to its enumeration method, which begins from left to right, all sibling nodes to the right of the inserted node are increased by one. In addition, all the descendant nodes will need to be changed. Finally, the number of children nodes inserted is restricted by the predefined value k.

Containment Scheme Encoding

In containment labeling scheme the labeling mechanism considers both levels of the nodes and ancestor-descendant relations (Zhang et al., 2001). Labeling scheme is used in which every node is assigned three values: “start, end, level”. For any two nodes u and v, u is an ancestor of v if u.start < v.start and v.end < u.end. In other words, the interval of v is contained in the interval of u. Node u is a parent of node v if u is an ancestor of v and v.level – u.level = 1 (Hu et al., 2006). The labels are shown in Figure 3.9.

Although the containment scheme is efficient to determine the ancestor-descendant (A-D) relationship, the insertion of a node will lead to a re-labeling of all the ancestor nodes of this inserted node and all the nodes after this inserted node in document order. This problem may be alleviated if the interval size is increased with some values unused . However, large interval size wastes a lot of numbers which causes the increase of storage, while small interval size is easy to lead to re-labeling.

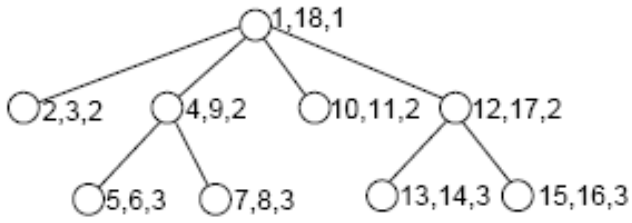


Figure 3.9. Containment Labeling Scheme

P-Containment Encoding

Different from the traditional containment scheme (Zhang et al., 2001), the “parent_start” value is stored rather than the “level” value. The “parent_start” value of a node is the “start” value of its parent. With this scheme determining the parent-child relationship is faster, and determining the sibling relationship is much faster (Li et al., 2005).

For two different nodes *u* and *v*, node *u* is a parent of node *v* iff the "parent_start" value of node *v* is equal to the "start" value of node *u* based on P-Containment.

For two different nodes *u* and *v* which are not the root of the XML tree, node *u* is a sibling of node *v* iff the "parent_start" value of node *u* is equal to the "parent_start" value of node *v* based on P-Containment.

The ancestor-descendant and ordering relationship determinations based on P-Containment are the same as the traditional containment scheme.

Prime-number Labeling Scheme

(Wu et al., 2004) proposed using prime numbers to label the XML tree via top-down and bottom-up approaches (Figure 3.10). As bottom-up approach, a prime number is assigned to each leaf node. Then, for each subsequent level, the parent's label becomes a product of their child labels. This approach has two drawbacks. Firstly, it will cause relatively large numbers to be assigned to the nodes at the top. Secondly, it is not possible for nodes with a single child only (Hu et al., 2006).

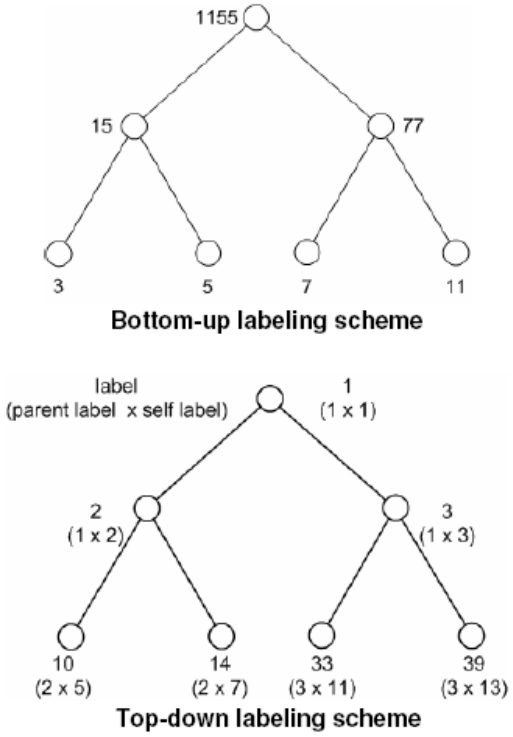


Figure 3.10. Bottom Up And Top Down Approaches Of Prime-number Labeling Scheme (Source: Wu et al., 2004)

Another disadvantage of the prime number labeling scheme is that each prime number can only be used once. Thus, the size of the label increases as it reaches the bottom of the tree (Su-Cheng et al., 2009).

Behind these disadvantages, it supports dynamic updates, requires no updates.

Region-based numbering schemes are the most popular numbering schemes. They can determine the ancestor–descendant relationship between two elements efficiently. While such a numbering scheme can greatly improve query performance, renumbering large amount of elements caused by updates becomes a performance bottleneck if XML documents are frequently updated. Insertion or deletion of nodes into a labeled XML tree may result in a total re-labeling of the XML tree.

3.3. Conclusion

The chapter's overall goal was to provide an overview for the related work in this research. Each algorithm that mentioned in this chapter provides a labeling algorithm for XML trees. All aim to reach to the queried node in an efficient way, but to do this they use different methods which have several advantages and disadvantages. Each node in the tree is assigned a typically unique node label, so that any of these relations between the nodes can be inferred from their labels.

Several labeling schemes have been introduced to develop an optimized retrieval, since they provide a quick way to determine the type of relationships as Parent-Child (P-C), Ancestor-Descendant (A-D), sibling and ordering relationships that are present among the nodes. A labeling approach should consider the characteristic of the data, the maximum depth and fan-out of the XML trees, space requirement of the labeling algorithm, label size and requirement of relabeling need.

Range based labeling schemes hold the starting or ending positions of an element in a document to identify the element so that the ancestor–descendant relationship between two elements can be determined by merely examining their codes. While such a numbering scheme can greatly improve query performance, renumbering large amount of elements caused by updates becomes a performance bottleneck if XML documents are frequently updated. Prefix based labeling schemes assign each node a vector that represents the path from root to the node itself. However such a labeling can

determine the structural relationships efficiently, the size of labels increases while the depth of the tree increases.

Here, by focusing on these drawbacks, a new XML labeling algorithm, Level Based Labeling (LBL), is proposed. Three versions of LBL are proposed. All LBL versions are flexible on the child number of the nodes. Second and third versions also reduce the need of relabeling after each insert/delete when compared to containment labeling scheme. Also all versions have fix-size labels, in spite of the depth of the tree increases; it still uses the same size of labels. In the next chapter all these properties of LBL will be explained.

CHAPTER 4

LEVEL BASED LABELING SCHEMES

4.1. Introduction

Labeling schemes provide the type of relationships present among nodes and are important block for structural join algorithms and important complement of structural indexes. Choosing a suitable labeling scheme requires different factors to be taken into consideration as storage, nature of data, query type and efficiency of maintaining that labeling scheme. A labeling scheme supporting dynamic XML data should be able to keep computational cost of labeling, label size and required re-labeling with inserts and deletes at minimum while providing Parent-Child (P-C), Ancestor-Descendant (A-D), sibling and ordering relationships. From this point of view, it is observed that there are many researches on labeling XML data. The proposed approaches in these works present many different methods. Some provide effective performance by defining relationships, while the others provide this performance gain by avoiding from relabeling the nodes.

In this thesis, three versions of a labeling scheme called Level-Based-Labeling (LBL) are proposed. All of the versions of it provide all four basic relationships among nodes. Basic LBL requiring inexpensive computation for construction of a label while Single Linked LBL and Double Linked LBL require minimum re-ordering with inserts and deletes. They have a reasonable label length with increasing level and fan-outs.

LBL is focused on to determine all 4 relationships to provide effective query performance. This scheme handles parent-child, ordering, ancestor-descendant and sibling relations by requiring reasonable space and relabeling with updates. In this thesis, three different versions of LBL, which can handle these relationships and have an efficient update on XML trees, are proposed. Each LBL version is efficient on a different characteristic of a XML document. The first version Basic LBL is effective on labeling the less frequently updated XML trees, while Single and Double Linked LBL are effective on frequently updated XML trees. However, they are both less effective on initial labeling of the trees. The main feature of the third version is that it is effective for

reversal ordering scans.

In this chapter the new labeling schemes are proposed, the structures of them are defined and the relationship determination and updating mechanism are explained with some examples.

4.2. Basic Level Based Labeling

Each node of Basic Level Based Llabeling (B-LBL) scheme is labeled with three values, <level, order, parent-order>. As it can be seen in Figure 4.1, the numbers in node labels represent the level of the node, the order of the node at the same level and the order of the node's parent node respectively.

The proposed labeling scheme does not uses pre-allocated labels for potentially insertion of data. This provides for the algorithm to use an optimized space for labeling. Nevertheless, in some cases insertion or deletion of the nodes requires relabeling. Whereby using level and orders of the nodes, this relabeling is processed only on necessary nodes.

B-LBL is appropriate for less frequently updated XML files. B-LBL keeps its computational cost at minimum. The initial labeling of an XML tree is at minimum level with this approach. Besides it has reasonable label length without considering the depth of the XML tree or fan-out of the nodes.

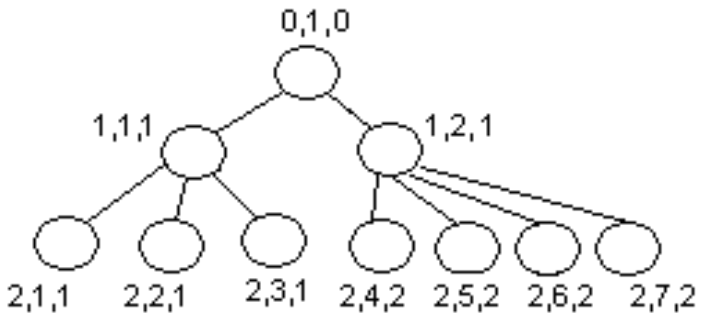


Figure 4.1. Basic Level-Based Labeling Scheme

4.2.1. Determining the relationships

Since every node in XML tree is assigned values <level, order, parent-order>, every node of a child holds the order number of its parent**.** This provides access to parent node directly.

For any nodes u and v, to satisfy Parent-Child (P-C) relation we can say, iff u.level+1=v.level and u.order=v.parent-order then u is parent of v.

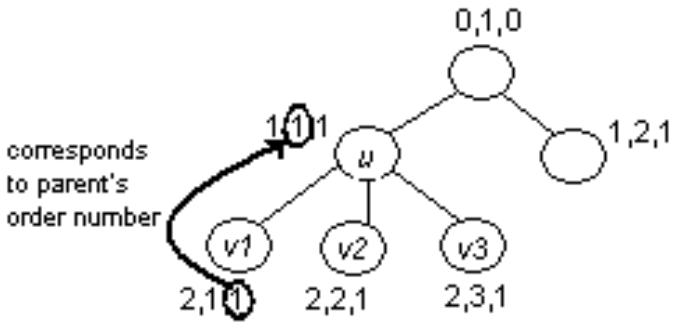


Figure 4.2. Parent Child Relationship in B-LBL

Similarly by knowing the level of the node and the order of its ancestor, Ancestor-Descendant (A-D) relationship is satisfied and can be stated as, for any nodes node u and v, iff v.level = u.level-1and v.parent_order=u.order then v is the ancestor of u (Figure 4.3).

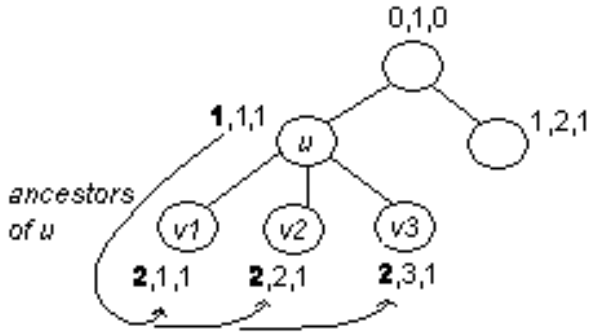


Figure 4.3. Ancestor-Descendant Relationship in B-LBL

In this scheme to determine the sibling and order relationships, the label holds the order number of the node. This order number represents the order of the node on its

level.

For any nodes v_1 and v_2, iff v_1.parent=v_2.parent, v_1.level=v_2.level and v_1.order≠v_2.order then v_1 and v_2 are siblings. Iff v_1 and v_2 are siblings and v_1.order < v_2.order then v_1 is in previous order then v_2. (Figure 4.4)

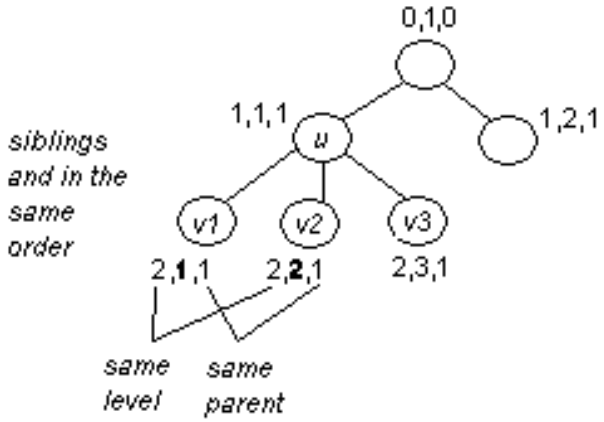


Figure 4.4. Sibling and Order Relationships in B-LBL

4.2.2. Updating data

LBL scheme is an optimized class labeling scheme. One property of this labeling scheme is that it does not relabel all the nodes after an update occurs. Because of holding the order numbers of nodes on a label, an update may effect on the order of other nodes. Addition or deletion of a node increases or decreases the number of nodes on a label. So this update requires renumbering of only the order numbers of the nodes.

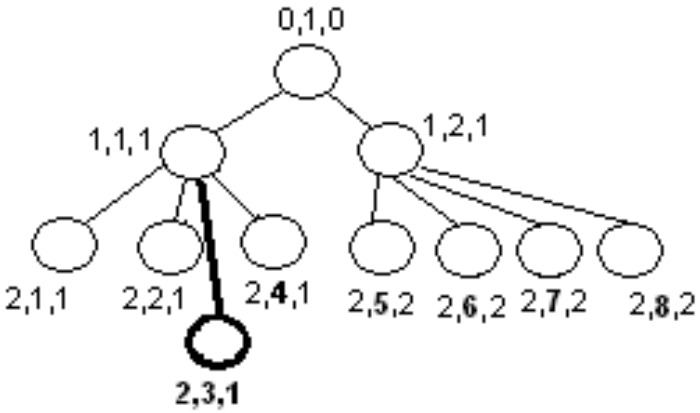


Figure 4.5. Insertion of the node labeled as <2,3,1>

In B-LBL, when an update occurs, only the following nodes on the same level and parent-order part of the descendant nodes should be relabeled in this scheme. Figure 4.5 and Figure 4.6 illustrates an example for this kind of relabeling.

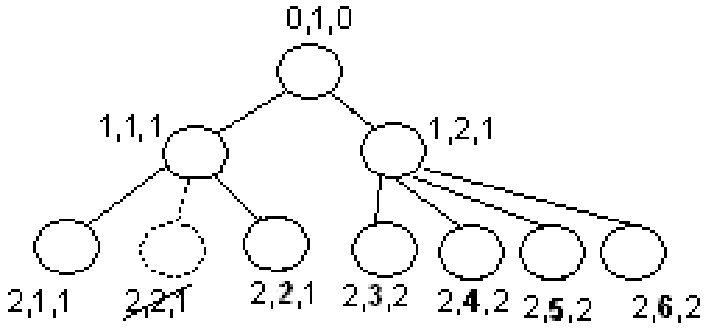


Figure 4.6. Deletion of the node labeled as <2,2,1>

If the reordered nodes have child nodes, the parent-order part of their labels should be updated as shown in Figure 4.7 and 4.8, because the parent-order represents the order number of the parent node.

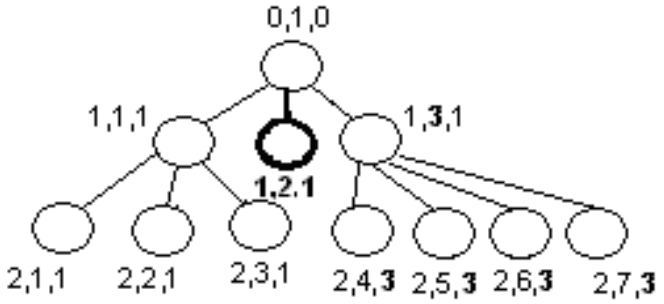


Figure 4.7. Insertion of the node labeled as <1,2,1>

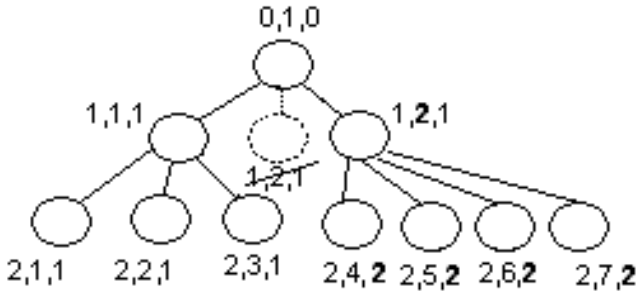


Figure 4.8. Deletion of the node labeled as <1,2,1>

4.3. Single Linked Level Based Labeling

Different from the first version, Single Linked LBL (SL-LBL) has four-part labels. The first, second and third parts are the same as in B-LBL; level, order and parent-order. The fourth part holds the order number of the right-side node (Figure 4.9).

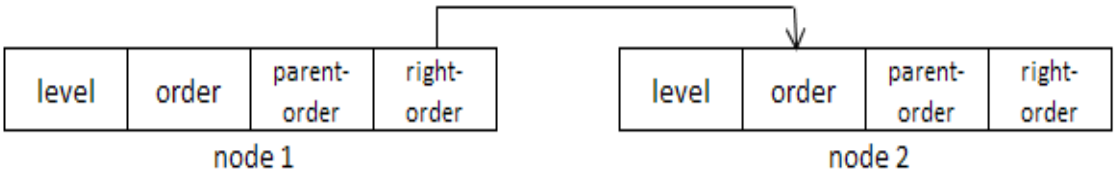


Figure 4.9. Label parts of SL-LBL

By holding the right-order, SL-LBL aims to avoid from relabeling of the nodes which are at the same level. Inserting a new node between node1 and node2 requires relabeling of only the right-order part of node1. After insertion, right-order part of node1 is equal to order number of the new node, and the new node's right order is equal to order of node2. The best result of this linking to the right node is that child nodes are not affected from insertion as in B-LBL.

In SL-LBL, the number of relabeled nodes after an insert or delete is always 1. By means of the links between nodes, no reordering is required, because a new inserted node gets "*the maximum order number of the level + 1*" as order number and takes its place between the required nodes by *right_order* links. Also these links supply an efficient querying on the XML tree.

When it is compared with B-LBL, one of the drawbacks of SL-LBL is its space requirement. Although it has fixed-size labels, it has one more part than B-LBL. So it consumes 1,33 times of space of Basic LBL.

Another drawback is its initial label construction time. Organization of the links of the labels consume some more time than the first version.

4.3.1. Determining the Relationships

In this second version, SL-LBL still provides the 4 relations between nodes. The P-C, A-D and sibling relations are the same as B-LBL.

The linking mechanism provides a better performance on querying ordered nodes. When querying the forward order of a node, following the right-order node numbers brings an easy way to determine order relationship.

In the Figure 4.10 the ordering relation between the nodes on the 4th level is illustrated. The 7th node is a node that is inserted between the 4th and 5th nodes on the 4th level of the tree after its initial labeling. After that update the node 4 shows the node 7 and the node 7 shows the node 5 as their next orders.

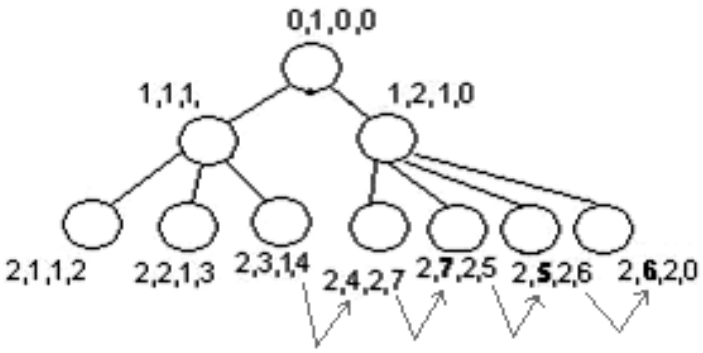


Figure 4.10. Order Relationship in SL-LBL

4.3.2. Updating Data

The Figure 4.11 explains this updating mechanism. In the figure, a new node is inserted between <1,1,1,2> and <1,2,1,0>. The maximum order number of level 1 is 2. When a new node is inserted it takes 3 as the order number. So the node <1,1,1,2>'s right-order is relabeled as 3. New label of node 1 is <1,1,1,3>. Node 3 takes 2 as right-order, and its label is <1,3,1,2>. Order of node 2 has not been changed, so the parent-order numbers of its child nodes are not affected from this insertion. The relabeled node number is always equals to 1.

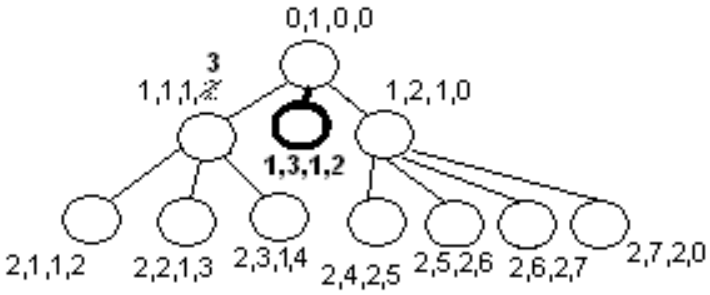


Figure 4.11. Insertion of the node labeled as <1,3,1,2>

4.4. Double Linked Level Based Labeling

The third version of LBL is Double Linked LBL (DL-LBL). It differs from the other versions by its label parts. This version uses five-part labels. The first 3 part of the label is as the same as B-LBL. The last two parts hold the order numbers of right and left nodes (Figure 4.12).

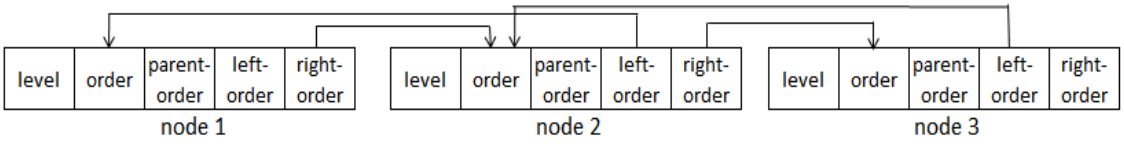


Figure 4.12. Label parts of DL-LBL

By holding the left-order and right-order, DL-LBL aims to avoid from relabeling that B-LBL has to be done. Inserting a new node between node1 and node2 requires relabeling of only two nodes. The right-order part of its pre-node and left-order of its next-node are updated as the new node's order. This fifth part of the label gains no more update performance than SL-LBL but it gains performance on backward order querying. If the XML database is queried much for backward order of a node then this version may be more appropriate for labeling.

Each update requires relabeling only on two nodes; the right and the left side nodes of the inserted one. This gains performance for continuously updated XML trees.

As its disadvantage space requirement issue can be considered. DL-LBL is the worst between all LBL versions. It uses one more part than SL-LBL to hold the backward links. This one more part means one more integer value for each element. Totally, it consumes1.66 times of space of B-LBL and 1.25 times of space of SL-LBL.

The initial labeling of the DL-LBL spends more time. As in SL-LBL, its initial label construction time is more than the first version.

4.4.1. Determining the relationships

The 3 relations; P-C, A-D and sibling; between nodes are provided as the same as B-LBL. The *level*, *order* and *parent_order* parts of the labels provide determination of these relations. DL-LBL also provides a forward ordering relationship with *right_order* parts of its labels, as in SL- LBL. In addition to that this version determines a backward ordering relation between the same level nodes with the help of *left_order* parts of its labels. Figure 4.13 illustrates this linking mechanism of DL-LBL.

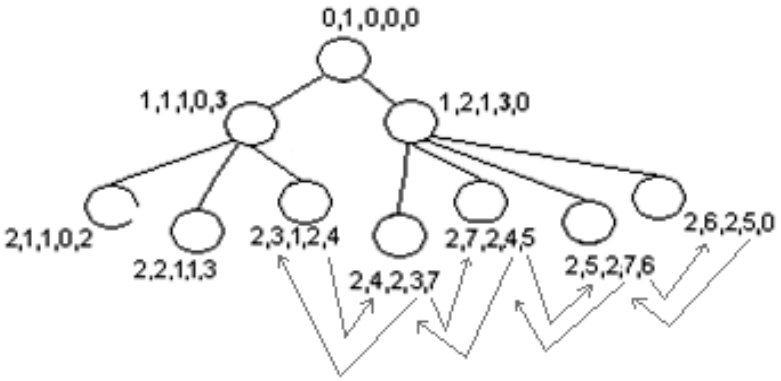


Figure 4.13. Order Relationship in DL-LBL

4.4.2. Updating Data

In the Figure 4.14, a new node is inserted between <1,1,1,0,2> and <1,2,1,1,0>. The maximum order number of level 1 is 2. When a new node is inserted it takes 3 as the order number. So the node <1,1,1,0,2>'s right-order is relabeled as 3. New label of node 1 is <1,1,1,0,3>. Node <1,2,1,1,0>'s left order is relabeled as 3 and its new label is <1,2,1,3,0>. Node 3 takes 1 as left order and 2 as right-order, and its label is <1,3,1,1,2>. Orders of node 1 and 2 have not been changed. So this insertion is not affected on any other nodes. The relabeled node number is always equals to 2.

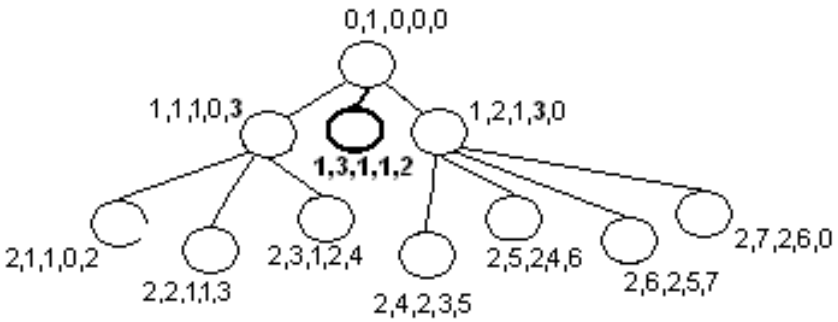


Figure 4.14. Insertion of the node labeled as <1,3,1,1,2>

4.5. Conclusion

The chapter's overall goal was to introduce the proposed labeling algorithms. All versions of the LBL labeling algorithm are explained with all their updating and relabeling mechanisms.

As introduced in the chapter, B-LBL labeling scheme determines all the four relationships with its three-part fix-size labels. All the labels on the tree do not need to be regenerated after each insert; only the required nodes should be relabeled. Also the depth or fan-out number of the tree does not effect on the size of the labels. From the labeling point of view, the scheme is not flexible on a continuously updating XML tree. The requirement of relabeling causes a performance bottleneck on insertion. But for less frequently updated XML trees this version is the most appropriate version of LBL, because of its initial labeling and space requirement.

SL-LBL uses four-part fix-size labels. This fourth part provides a link between the nodes on the same level. By the help of this link SL-LBL avoids relabeling. It requires relabeling only on 1 node after each update. As B-LBL, the depth or fan-out number of the tree does not effect on the size of the labels. All 4 relationships are provided on this version. In addition it provides a forward ordering mechanism with its right-order links. Although it consumes more space than the B-LBL and more time for initial labeling, its updating time is considerably better than the first version. So this version is more appropriate for continuously updating XML trees.

DL-LBL is a five-part fix-size labeled labeling scheme. This version aims to meet the deficiency of backward ordering of SL-LBL. In this version nodes are connected to each other with forward and backward links. If the XML database is permanently updated and also queried for backward ordered nodes, then this version

may be preferred.

In the performance tests, LBL versions will be compared with containment labeling scheme which is also a range based scheme. Containment labeling scheme is one of the most popular XML labeling schemes. It also uses three-part fixed-size labels, which hold level of the node with one of their parts. In the next chapter, 4 test cases are going to be performed and performance results are going to be evaluated.

CHAPTER 5

PERFORMANCE EVALUATION

5.1. Introduction

In this chapter the performance of three versions of LBL schemes are evaluated and compared with another level based labeling scheme. This labeling scheme is containment labeling scheme. Containment (Zhang et al., 2001) is a relabel-dependent labeling scheme which labels with 3-part labels. The first part of the label holds start number, the second part holds end number and the third part holds the level of the node. This labeling determines A-D, P-C and level relations. The first version of LBL is an optimized labeling scheme which has 3-part labels. It can determine the A-C, P-C, order and sibling relationships. The second version of LBL uses 4-part labels. After updates it requires relabeling on one node. And the third version of LBL has 5-part labels, which supply both forward and backward ordering while requiring minimum relabeling after each update.

The performance tests are made for 4 different cases; labeling performance tests, space requirements of all 4 algorithms, querying and updating performance on XML datasets. There were 3 test datasets with different characteristics that are formed from the real world data and be used in the performance tests. The details of these performance tests are explained in the Appendix A.

This chapter includes the labeling performance, space requirement, updating and querying performance comparison tests on 4 labeling approaches.

5.2. Experimental Evaluation

All of the schemes are implemented in Java and all the experiments are carried out on a Intel Core 2 Duo 2.53 GHz processor with 3 GB RAM running Windows XP Professional.

Table 3.1 shows the characteristics of the test datasets. All 3 datasets are all real-

world XML data (University of Washington). These datasets are chosen because they have different characteristics, i.e. their file size, fan-out, depth, and total number of nodes.

Table 5.1. Test datasets

filename	**description**	**file size**	**elements**	**attributes**	**max-depth**	**avg-depth**	**max fan-outs**
lineitem.xml	Line items	30 MB	1022976	1	3	2,94117	60175
nasa.xml	Astronomical Data	23 MB	476646	56317	8	5,58314	2435
treebank_e.xml	Partially-encrypted treebank	82 MB	2437666	1	36	7,87279	56384

5.2.1. Labeling Performance

Labeling performance of all algorithms is tested on all the datasets. All XML files in a given dataset is read and placed in a XML data trees with node labels.

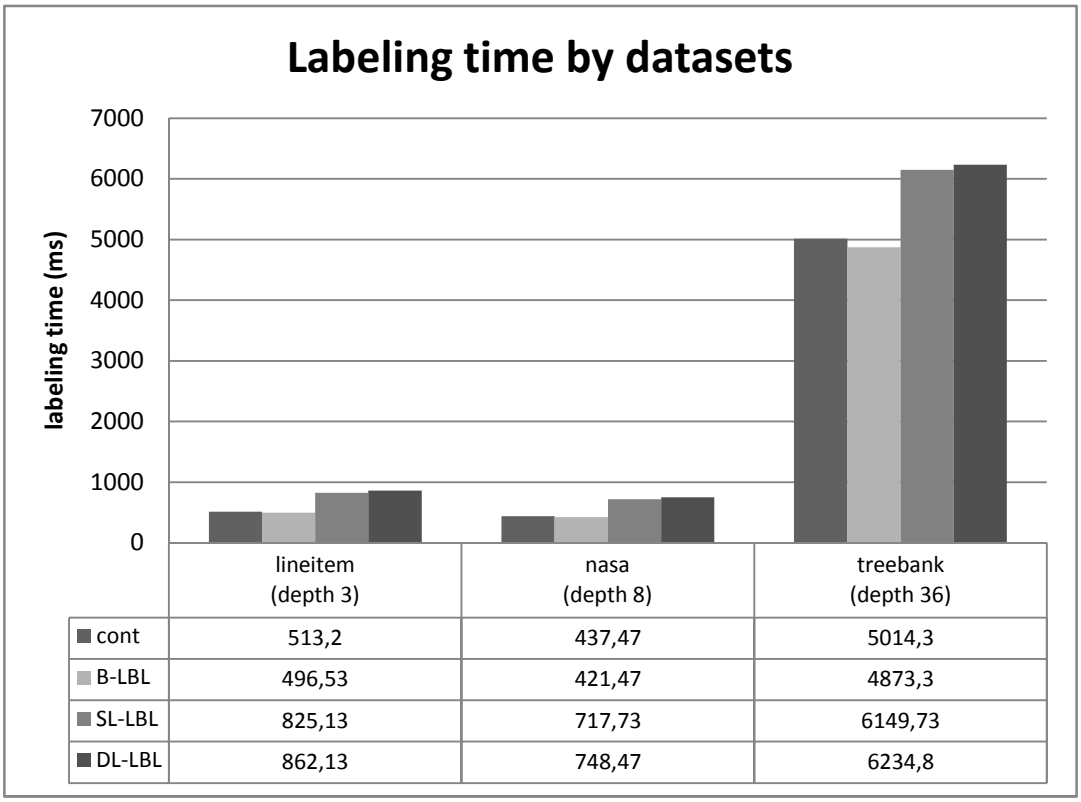


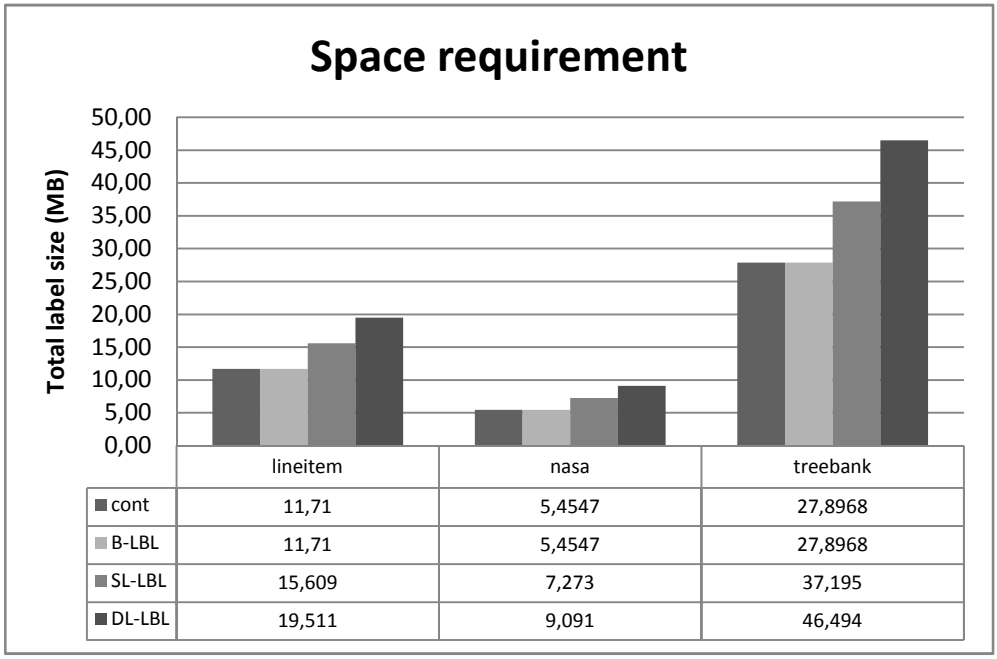
Figure 5.1. Labeling performance

In the test, each algorithm is executed on three data sets, for 15 times. The values on the graph of Figure 5.1 show the average execution times of these repeated results.

The execution time of these labeling algorithms differs by the structure of the tested xml files. In all cases of labeling files with different depths, it has been observed that B-LBL algorithm performs better than the containment scheme and other two LBL versions. The initial labeling time of SL-LBL and DL-LBL are more than other two cases. This is because in these two versions the labels are connected to its next or previous nodes with their *right_order* or *left_order* parts. This initial organization of labels consumes much time but with the help of these links, these algorithms require minimum relabeling.

5.2.2. Space Requirements

The performance study to understand the space requirement while storing the labels is done on lineitem, nasa and treebank datasets. All the xml files in a dataset are labeled with containment and LBL algorithms respectively. The graph in Figure 5.2 shows the total space of all the labels that allocated for each data set.



	lineitem	nasa	treebank
cont	11,71	5,4547	27,8968
B-LBL	11,71	5,4547	27,8968
SL-LBL	15,609	7,273	37,195
DL-LBL	19,511	9,091	46,494

Figure 5.2. Space requirements

In this test, all algorithms used integer values to label the XML trees. Containment uses 3 integers for each of its labels. B-LBL is also uses 3 integers for a label of an element. Because SL-LBL uses 4-part labels, its labels consume 16 byte (4 integers) space for each element, while DL-LBL uses 20 byte (5 integers) for its 5-part labels.

The graph in Figure 5.2 shows that DL-LBL uses the most space for labeling the datasets. Because of containment and B-LBL uses 3-part labels they consume the same space for the same datasets.

The range of the numbers which all LBL versions use is smaller. Containment scheme labels the elements with both a start value and an end value, so it can label maximum max_value_of_datatype/2 elements. All LBL schemes use level or order values. So they can label maximum max_value_of_datatype elements. If it is assumed that only integer values are used while labeling, it can be observed that all LBL schemes can label a XML tree with more depth and fan-out. The maximum depth could be 1073741823 while labeling with containment labeling scheme where LBL schemes could label a 2147483647 depth tree.

5.2.3. Query Performance

In the experimental studies of this paper, the query performance is evaluated with three versions of LBL and containment labeling scheme. In this test, relationship querying performance is compared, so this test is completed in five parts; querying P-C, A-D, sibling, forward ordering and backward ordering.

P-C Querying

In this part of the test, the children nodes of a given parent node are queried. The results showed that SL-LBL and DL-LBL performs the best performance on querying P-C relations. B-LBL and containment schemes have slower performance on querying the child nodes. However, B-LBL performs far better performance than containment scheme. The main reason of this better performance is the *parent_order* part of the labels of LBL schemes.

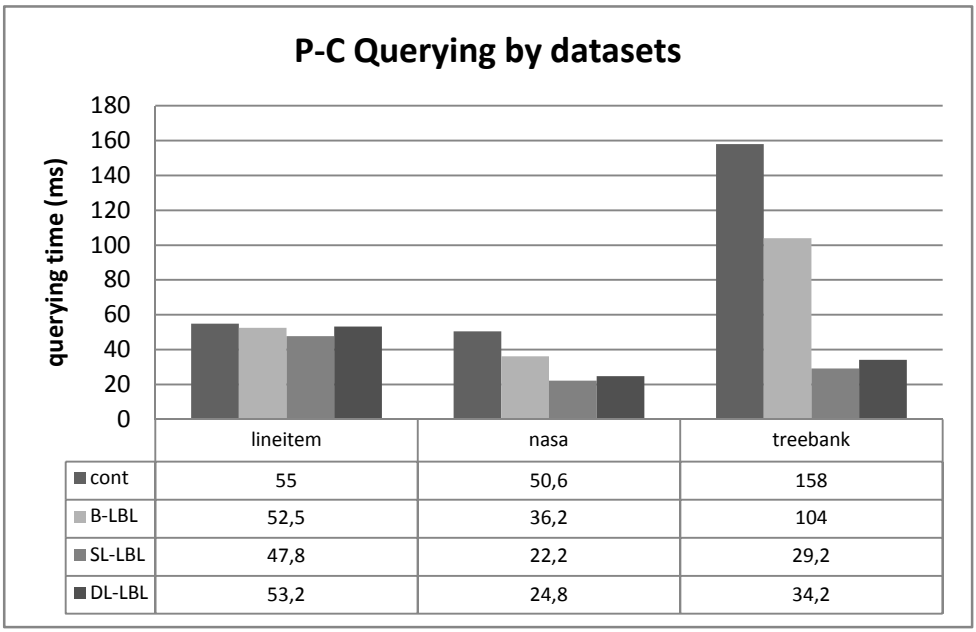


Figure 5.3. P-C Querying performance

A-D Querying

In this part of performance tests, the ancestors of a given element are queried. Since containment scheme is a range based labeling scheme it performs better performance on A-D relationships. B-LBL queries the ancestor nodes in a recursive way which causes worse performance than others.

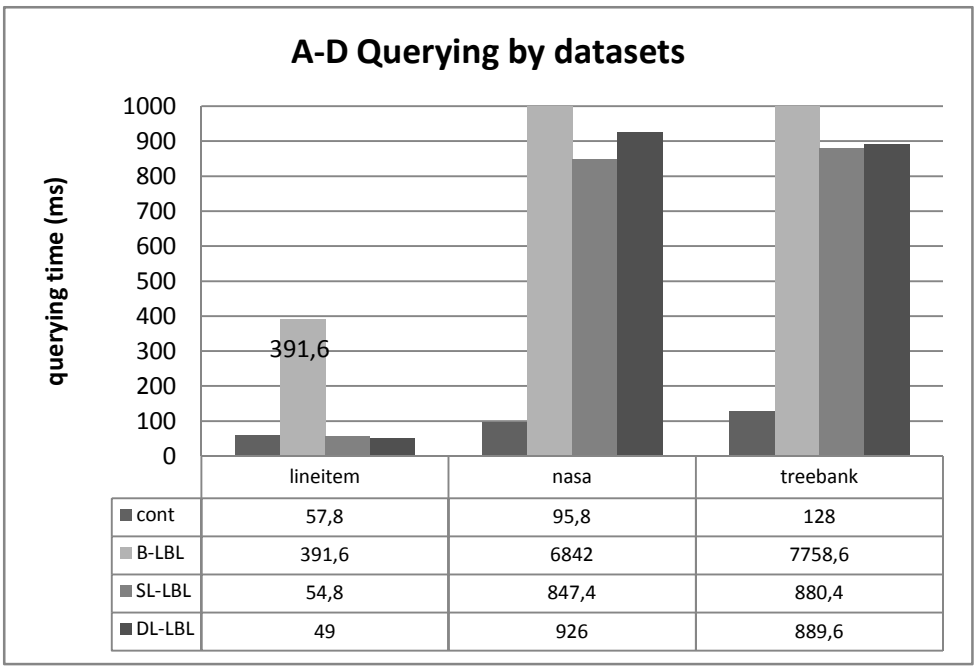


Figure 5.4. A-D Querying performance

Sibling querying

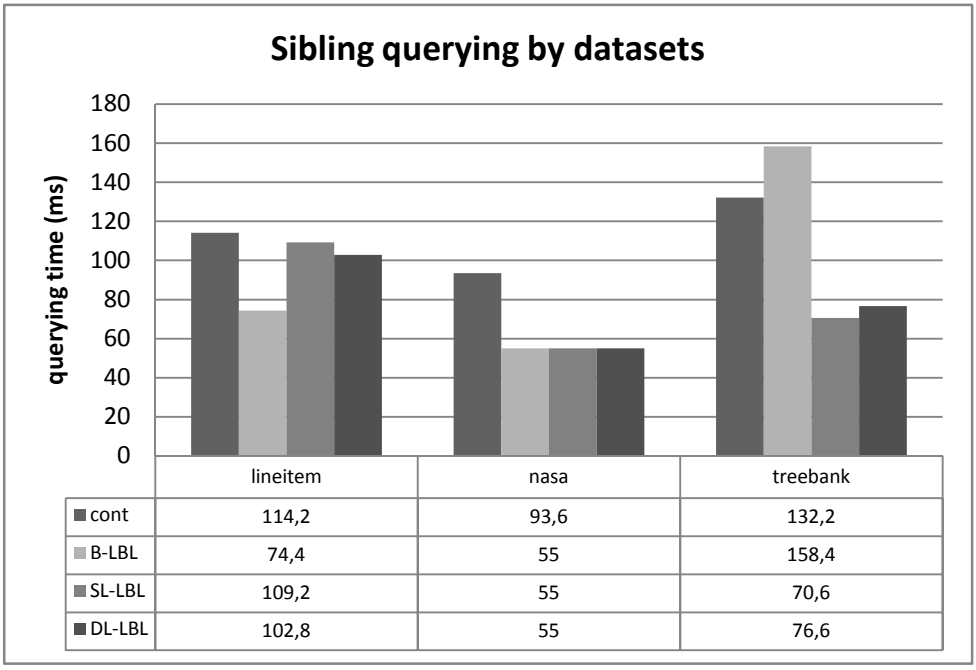


Figure 5.5. Sibling querying performance

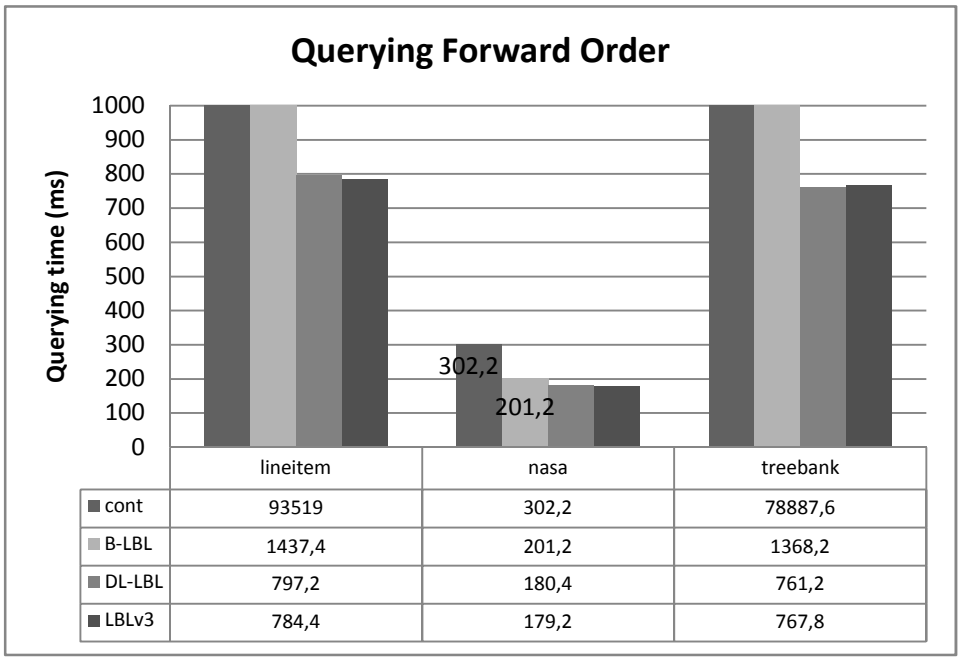
While querying the sibling of a node, the total element number effects on the total query time for containment and B-LBL. SL-LBL and DL-LBL uses links for detecting the siblings of nodes. So only the fan-out number effects on the performance on querying the siblings of a node.

Order querying

SL-LBL and DL-LBL uses *right_order* parts of their labels to detect forward orders of a node. Also these two schemes search the following orders only on the nodes which are on the same level.

Relabeling characteristic of B-LBL keeps all nodes in order with a sequential order number. This pre-ordering provides a better performance than containment scheme on detecting the document order.

After the graph on the Figure 5.6 is analyzed, it can be noticed that SL-LBL and DL-LBL have very close results on querying the forward order nodes. This result is much longer for B-LBL and many times longer for containment labeling scheme. It can be said that, DL-LBL is %7460 better than containment scheme, while SL-LBL is %7422 better and B-LBL is %4141 better than containment labeling scheme.



	lineitem	nasa	treebank
cont	93519	302,2	78887,6
B-LBL	1437,4	201,2	1368,2
DL-LBL	797,2	180,4	761,2
LBLv3	784,4	179,2	767,8

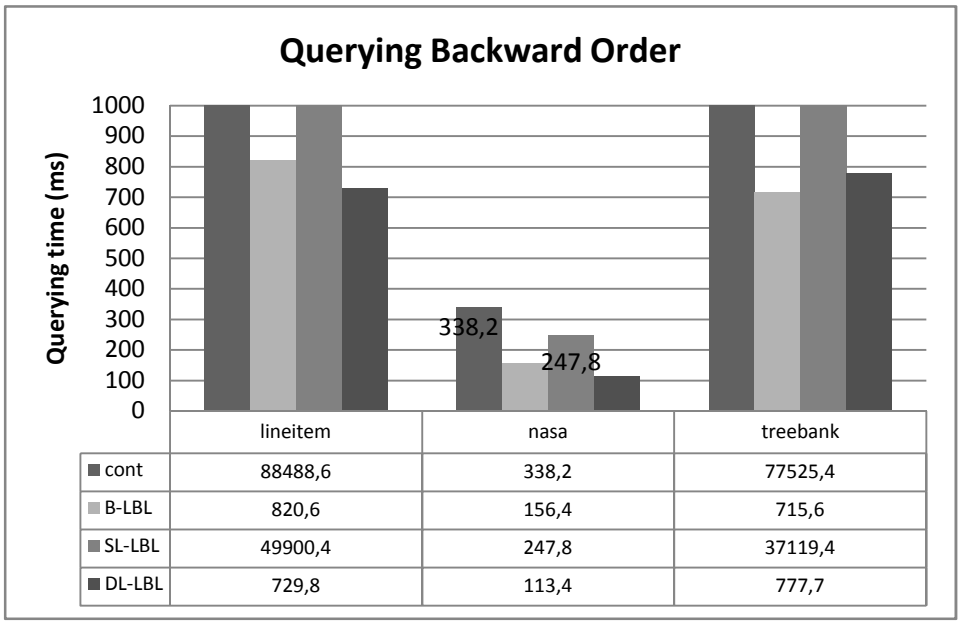
Figure 5.6. Forward order querying performance

For detecting the backward order nodes, DL-LBL offers a *left_order* part on its labels. This part holds the order of the node which is in the previous order. With the help of this part, DL-LBL performs much better performance on backward order detecting. Also B-LBL has a good performance on querying the backward order. Since it keeps the node order numbers in order by relabeling after each update, this scheme can detect the previous orders by finding the order numbers on the descending order.

When the time performances on Figure 5.7 are analyzed, the results are; B-LBL queries the backward order nodes %7283 better than containment scheme; SL-LBL queries with %173 better performance and DL-LBL queries with %7471 better performance than containment scheme.

SL-LBL may not be a preferable labeling scheme for an XML database which is also queried for the backward order, because it does not keep the order numbers in order and does not have any label part to hold previous nodes.

Containment scheme is a range based labeling scheme but this property is not enough for detecting the document order. With the help of *level* part of containment labels, the order of the children of a node can be queried. However, its performance is the worst when it is compared with LBL schemes.



	lineitem	nasa	treebank
cont	88488,6	338,2	77525,4
B-LBL	820,6	156,4	715,6
SL-LBL	49900,4	247,8	37119,4
DL-LBL	729,8	113,4	777,7

Figure 5.7. Backward order querying performance

5.2.4. Update Performance

In the update performance tests, 3 cases are evaluated. The first case performs a uniform insertion. For every dataset new nodes are inserted after every 50th node on the same level. The aim of this case is to observe the update performance of the algorithms against uniformly insertions.

Second case inserts 250 new nodes to a particular point. This test is evaluated to observe how algorithms respond to insertions that are occurred between two particular siblings.

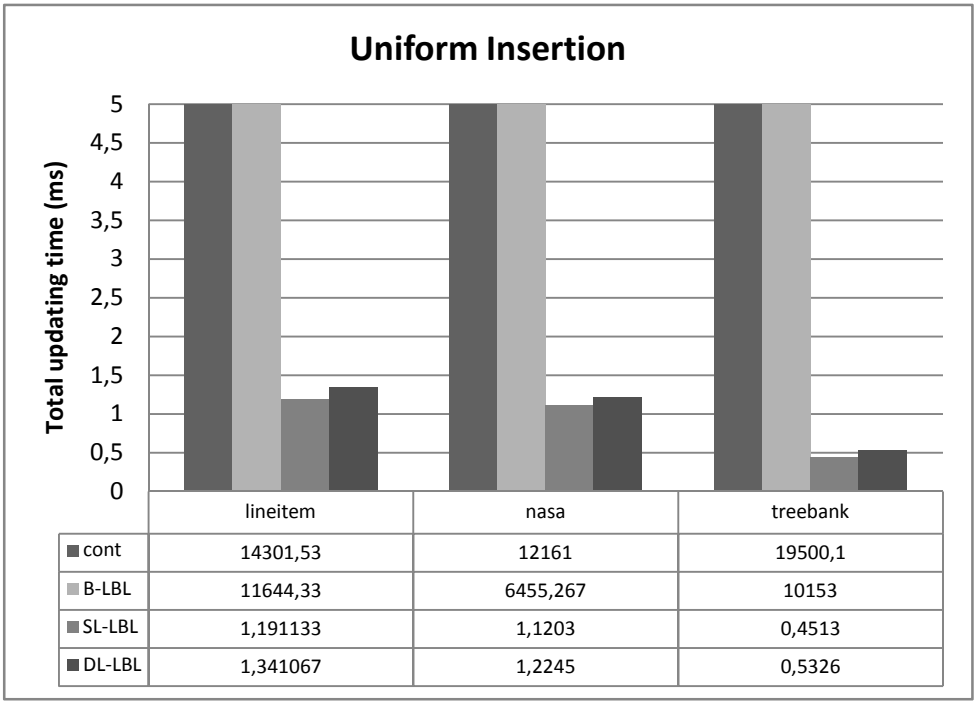
Third case is the union of the first two cases. 600 new nodes are inserted to different three points of the tree.

Since containment labels keeps both start and end values, this scheme needs to re-label the existing nodes at each time when a node is inserted into the XML. This causes a bottleneck on updates.

B-LBL re-labels only the nodes that are on the same level with the inserted node. The aim of the algorithm is to keep the nodes in order based on their *order* numbers. So relabeling is required after any update on the same level. In SL-LBL to avoid this relabeling requirement, *right_order* links are used. So the updating process relabels only one node for each insert; the previous node. This empowers the update performance of this algorithm. We can observe the same performance gain for DL-LBL. It uses

right_order and *left_order* parts to avoid relabeling. It overcomes the relabeling issue with these two parts.

The graphs 5.8, 5.9 and 5.10 show the total update time of the algorithms on these test cases respectively.



	lineitem	nasa	treebank
cont	14301,53	12161	19500,1
B-LBL	11644,33	6455,267	10153
SL-LBL	1,191133	1,1203	0,4513
DL-LBL	1,341067	1,2245	0,5326

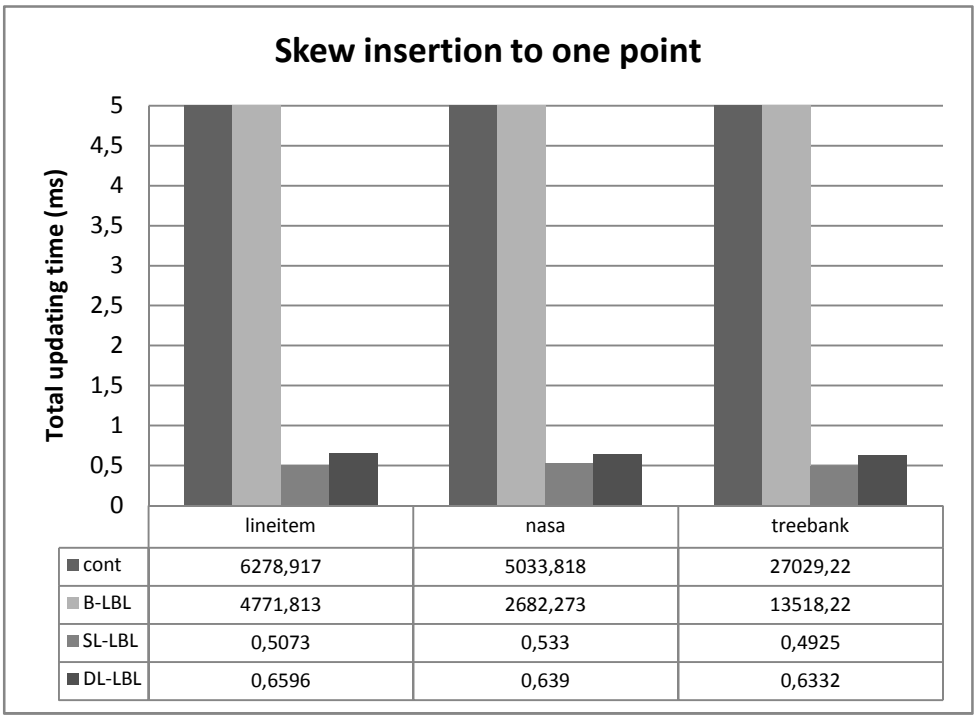
Figure 5.8. Uniform insertion performance

In the Graph 5.8, the total updating time differences can be seen obviously. According to this graph B-LBL has %167 better performance than containment scheme where SL-LBL has approximately %2206966 and DL-LBL has approximately %1914428 better performance than containment labeling scheme.

Graph 5.9 indicates the total updating time of skew insertions to one point of the tree. SL-LBL and DL-LBL have noticeable performance gain on this insertion test. SL-LBL is approximately %2563793 more efficient than containment scheme. For DL-LBL this ratio is %2009311 and for B-LBL it is %172.

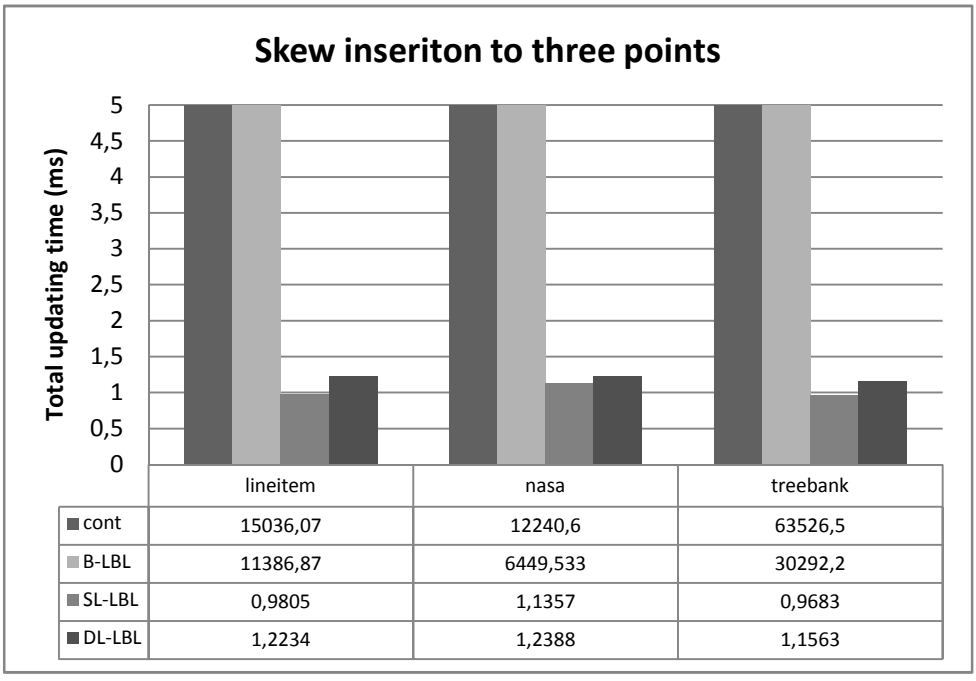
Figure 5.10 illustrates the result of another insertion test case. In this test, skew insertions are performed to three different points of the tree. This graph shows the total insertion time for all three datasets. In this test, SL-LBL is the most efficient labeling algorithm again. It is %3052347 more efficient than containment labeling scheme.

DL-LBL also has a noticable difference than containment scheme. Its performance gain is %2565311 with respect to containment scheme.



	lineitem	nasa	treebank
cont	6278,917	5033,818	27029,22
B-LBL	4771,813	2682,273	13518,22
SL-LBL	0,5073	0,533	0,4925
DL-LBL	0,6596	0,639	0,6332

Figure 5.9. Skew insertion performance



	lineitem	nasa	treebank
cont	15036,07	12240,6	63526,5
B-LBL	11386,87	6449,533	30292,2
SL-LBL	0,9805	1,1357	0,9683
DL-LBL	1,2234	1,2388	1,1563

Figure 5.10. Complex insertion performance

5.3. Discussion on Results and Conclusion

Experimental study of this paper consists of 4 different test cases which compare the performance of versions the LBL schemes with each other and containment labeling scheme. Containment labeling algorithm is chosen because of its range-based and level based labeling property. It is also good at determining the A-D relationship.

The first test case is about labeling the static XML data. In this test we observed that B-LBL scheme has better performance than other algorithms. SL-LBL and DL-LBL consume much time than B-LBL and containment for initial labeling.

The second test case is about space requirements. When the total label spaces are calculated, it is observed that containment and B-LBL consume the same space, while SL-LBL and DL-LBL consumes much more space because of their label length. Also it can be analyzed that containment labeling scheme can label half of the elements that all LBL versions can do with the same label range.

From the querying P-C, sibling and order relations graphs, it can be analyzed that all LBL algorithms provide a good performance on detecting those relations. When LBL versions are compared between each other SL-LBL and DL-LBL respond more quickly to the queries on the tests. Especially on backward order querying DL-LBL has an acceptable performance gain than the others.

Update tests show that SL-LBL and DL-LBL have a noticeable better performance than containment and B-LBL, since they do not need to relabel more than one or two nodes at any update.

CHAPTER 6

CONCLUSION

Motivated by the need to support XML data updates in dynamic XML environments, in this thesis, a new XML labeling scheme, Level-Based labeling (LBL), is proposed in three versions. While the main structure of the algorithm is the same for all versions, each version focuses on different performance requirement. The aim of all versions of the labeling algorithm is to keep label size and required re-labeling with inserts and deletes at minimum while providing several relationships. The first version, Basic LBL focuses on ordering the nodes on the same level and keeping the label size at minimum. Because of its label structure, its computational cost of labeling is at minimum. The second version, Single Linked LBL is an upgrade of Basic LBL on relabeling issue. It uses links between consecutive nodes, and relabels only one node after each update. These links also provides an easy movement on the nodes. Double Linked LBL supplies the deficiency of Single Linked LBL on backward ordering. It uses backward links on its labels.

Basic LBL can be applied widely to less frequently updated XML data integration schemes or other applications to efficiently query the XML data. Single Linked LBL can be applied to continuously updated XML database because of its efficiently updating mechanism. Double Linked LBL can again be applied to dynamic XML data which is frequently queried in the backward order. Experimental study of this project compares the performance of the LBL versions with each other and a labeling scheme; containment labeling scheme. The tests are performed with 4 different cases; labeling the XML data, space requirement, query performance and update performance. Containment is chosen since it is similar to LBL with its label structure and comparison would reveal out labeling, space requirement, query and update performances.

The results of the performance tests confirm that;

i) Basic LBL is more efficient than containment labeling scheme in labeling the xml trees

ii) Basic LBL has the same space requirement to hold the labels with containment scheme because they both use three-part labels

iii) Double Linked LBL consumes the most space while labeling the nodes

iv) Basic LBL's querying performance is better than containment scheme although both of them uses level wised information in the labels

v) Basic LBL needs less renumbering than containment scheme in case of updates done on the XML tree where Single and Double Linked LBL need almost no relabeling with the same update

As a summary we can say that the proposed labeling schemes have strengths as

i) all are flexible to changing fan-outs

ii) each provides level wised information in its labels resulting in better performance for certain types of queries

iii) all have a very simple label logic which does not need heavy computation

iv) second and third versions require reasonable amount of relabeling in case of updates

REFERENCES

Alkhatib R.; Scholl M. H. *Compacting XML Structures Using a Dynamic Labeling Scheme*, BNCOD, **2009**.

Ankorion I. *Change Data Capture – Efficient ETL for Real-Time BI,* DM Review Magazine, January **2005**.

Baer H. *On-Time Data Warehousing with Oracle Database10g - Information at the Speed of your Business*, An Oracle White Paper, March **2004**.

Cohen E.; Kaplan H.; Milo T. *Labeling Dynamic XML Trees*, Proceedings of PODS, **2002**.

Devlin B. A.; Murphy P. T. *An Architecture for a Business and Information System,* IBM Systems Journal, **1988**, *21* (1).

DuCharme B. *XSLT Quickly*, Manning Publications Co., **2001**.

Duong M.; Zhang Y. *Dynamic Labelling Scheme for XML* Data *Processing*, On the Move to Meaningful Internet Systems: OTM, **2008**.

Florescu D.; Kossmann D. *Storing And Querying Xml Data Using An Rdmbs ,* Bulletin Of The IEEE Computer Society Technical Committee On Data Engineering, **1999**.

Golfarelli M.; Rizzi S.; Vrdoljak B. *Data Warehouse Design from XML Sources*, Proceedings of the 4th ACM International Workshop on Data Warehousing and OLAP, **2001**.

Hu M.; Ling T. W.; Li C. *Efficient Processing of Updates in Dynamic XML Data,* Proceedings of the 22nd International Conference on Data Engineering, **2006**.

IBM Information Management software White Paper *Evaluating real-time data integration solutions,* January **2008**.

Kaplan H.; Milo T.; Shabo R. *Compact Labeling Scheme for XML Ancestor Queries,* Theory of Computing Systems, **2007**, *40* (1), 55-99.

Kiani A.; Shiri N. *Generalized Model for Mediator Based Information Integration*, IEEE, **2007**.

Kimball R.; Caserta J. *The Data Warehouse ETL Toolkit*, Wiley Publishing Inc., **2004**, pp 29-52.

Ko H. K.; Lee S. *An Efficient Scheme to Completely Avoid Re-labeling in XML Updates*, WISE, **2006**.

Lee Y. K.; Yoo S. J.; Yoon K. *Index Structures for Structured Documents,* ACM First International Conference on Digital Libraries, March **1996**.

Li C.; Ling T. W.; Lu J.; Yu T. *On Reducing Redundancy and Improving Efficiency of XML Labeling Schemes*, CIKM, **2005**.

Li Q.; Moon B. *Indexing and Querying XML Data for Regular Path Expressions*, Proceedings of the 27th VLDB Conference, Roma, Italy, **2001**.

Lu J.; Ling T. W. *Labeling and Querying Dynamic XML Trees*, Proceedings of 6th Asia Pacific Web Conference, APWeb, **2004**.

Nicola M.; Linden B. *Native XML Support In DB2 Universal Database*, The 31st VLDB Conference, Trondheim, Norway, **2005**.

O'Neil et al. *ORDPATHs: Insert-Friendly XML Node Labels*, SIGMOD, **2004**.

Sans V.; Laurent D. *Prefix Based Numbering Schemes for XML:Techniques, Applications and Performances*, PVLDB '08, August **2008**.

Santoro N.; Khatib R. *Labeling and Implicit Routing in Networks*, The Computer Journal, **1985**, *28* (1).

Shanmugasundaram, J. et al. *Relational Databases For Querying XML Documents: Limitations And Opportunities*, The 25th VLDB Conference, Edinburgh, Scotland, **1999**.

Su-Cheng H.; Chien-Sing L. *Node Labeling Schemes in XML Query Optimization: A Survey and Trends*, IETE Technical Review, **2009**.

Tatarinov et al. *Storing and Querying Ordered XML Using a Relational Database System*, ACM SIGMOD, **2002**.

University of Washington XML Repository Database. http://www.cs.washington.edu/research/xmldatasets/ (accessed July 10, 2010).

WarehouseArchitect User's Guide, PowerDesigner Version 6.1.3, Document number: AA04531, Chapter 14:Defining Multidimensional Objects, Sybase, Inc. and its subsidiaries, Jan **1999**.

Wiederhold G. *Mediators in the Architecture of Future Information Systems*, The IEEE Computer Magazine, March **1992**.

The World Wide Web Consortium (W3C) Homepage, http://www.w3.org (accessed 18 August, 2009).

Wu X.; Lee M.; Hsu W. *A Prime Number Labeling Scheme for Dynamic Ordered XML Trees*, Proceedings of the 20th International Conference on Data Engineering, **2004**.

Xu J.; Luo D.; Meng X.; Lu H. *Dynamically Updating XML Data: Numbering Scheme Revisited*, World Wide Web: Internet and Web Information Systems, **2005**, *8*, 5–26.

Zhang et al. *On Supporting Containment Queries in Relational Database Management Systems*, ACM SIGMOD, **2001**.

APPENDIX A

SOFTWARE IMPLEMENTATION

The test cases are implemented in Java on a Intel Core 2 Duo 2.53 GHz processor with 3 GB RAM running Windows XP Professional. 4 test cases are tested and 3 real world datasets (University of Washington) are used in these test. All data is stored as text files.

These datasets are;

Lineitem.xml file is a 30 MB line item database. It has 1022976 elements at all. The maximum depth of the tree is 3 and maximum fan-out of a node is 60175.

Nasa.xml is a 23 MB XML file. This database has 476646 with maximum 8 depths. The maximum fan-out of the tree is 2435.

Treebank_e.xml is a Partially-encrypted treebankfile. Its size is 82 MB, its maximum depth is 36 and maximum fan-out is 56384. The file has 2437666 elements.

There are 4 test cases; labeling performance, space requirements, query performance and update performance. The following are explaining the software of the tests.

Labeling Performance

In labeling performance codes which are written in Java, the .xml file of the dataset is read from its folder and labeled.

The file is read from the dataset folder as below;

```
File f = new File(xmlFilePath + "/" + xmlFile + ".xml");

DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
DocumentBuilder dbuilder = dbFactory.newDocumentBuilder();
Document doc = dbuilder.parse(f);
Element root = doc.getDocumentElement();
```

A new instance is created as the type of *DocumentBuilderFactory* which is a class of Java API. The *DocumentBuilderFactory* instance is assigned to a *Document builder*. Each *builders* parses the xml file and assigns the first element to a *root* variable. The *Element root* holds the root of the XML data tree.

Afterwards each version of LBL labels the XML tree with its own labeling method.

Basic LBL

Each node starting from the root is assigned to a variable named *root*, is labeled with the numbers *level*, *order* and *pOrder*. '*level*' is the variable that holds the level number for each node. *level* is incremented by 1 after *root* variable is assigned to the child of the current node. '*order*' is the variable that holds the order of the *root* variable. *Order* is incremented by one while the *level* is constant. When *level* is incremented the *order* is assigned to *1* again. The '*pOrder*' holds the order number of the current node's parent. When a node is labeled with the *level, order* and *pOrder* variables, it is put in an array list, *XmlIndexList.* The code of this labeling process is below.

```
public void traverseXMLfile(Element root, int level,
ArrayList<XmlNode> XmlIndexList, int orderList[]) {
      int pOrder=0;
      int order=0;

      // gets the children of the root node
      NodeList children = root.getChildNodes();
      level++;

      // if the node is not a leaf node, it is put in the array list
      with //labels
      if(root instanceof Element){
            pOrder=orderList[level-1];
            orderList[level]++;
            order=orderList[level];
            XmlNode newData = new XmlNode(level,order,pOrder,root);
            XmlIndexList.add(newData);
      }

      for (int i = 1; i < children.getLength(); i=i+2) {
            Node child = children.item(i);

            boolean hasChldNodes = child.hasChildNodes();

// If the root has chidren the 'label' method is called
//recursively with assigning the 'child' element to root.
if (hasChldNodes == true && child.getChildNodes().getLength()>2) {
                  traverseXMLfile((Element) child, level, XmlIndexList,
orderList);
            }
            // If root is a leaf node, it is labeled with its
            //level,order,pOrder then put in the index array.
            else {
                  if (child instanceof Element) {
                        pOrder=orderList[level];
                        orderList[level+1]++;
                        order=orderList[level+1];
```

```
                        XmlNode newData = new
XmlNode(level+1,order,pOrder,(Element)child);
                        XmlIndexList.add(newData);

                    }
                }
            }
}
```

Single Linked LBL

Single linked LBL holds all XML tree as a vector of linked lists. Each level of the tree is placed on a link list and all link lists are connected each other with a vector. Figure A.1 illustrates this structure.

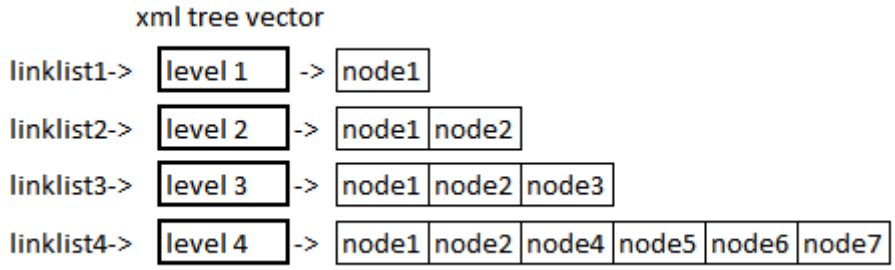


Figure A.1. Link list structure of Single Linked LBL

In this algorithm, the main idea of labeling is the same as Basic LBL, but the structure is different. When a node is read from the tree it is hold by *newNode* variable. It s labeled with *level*, *order*, *pOrder* and *rOrder* numbers. Each node is added to an array list regarding to its *level* number. Then each array list is added to the related indice of the *XmlIndexList* vector corresponding to its level numbers. The source code of this algorithm is below;

```
public void label(Element root, int level,
      Vector<ArrayList<XmlNode>> XmlIndexList, int orderList[][]) {
      int pOrder = 0;
      int order = 0;

      // gets the children of the root node
      NodeList children = root.getChildNodes();

      ArrayList<XmlNode> levelBase = new ArrayList<XmlNode>();
      if (XmlIndexList.size() == 0)
            XmlIndexList.add(level, levelBase);
      level++;
      if (XmlIndexList.size() < level + 1)
            XmlIndexList.add(level, levelBase);
```

```
      // if the node is not a leaf node, it is put in the array list
      with //labels
      if (root instanceof Element) {

            pOrder = orderList[level - 1][0];
            orderList[level][0]++;
            order = orderList[level][0];
            int size = XmlIndexList.get(level).size();
            XmlNode newData = new XmlNode(level, order, pOrder, 0,
root);
            XmlIndexList.get(level).add(newData);
            if (size != 0)
                  XmlIndexList.get(level).get(size - 1).rOrder = order;
      }

      for (int i = 1; i < children.getLength(); i = i + 2) {
            Node child = children.item(i);
            boolean hasChldNodes = child.hasChildNodes();

            // If the root has chidren the 'label' method is called
            //recursively with assigning the 'child' element to root.
            if (hasChldNodes == true &&
child.getChildNodes().getLength() > 2) {
                  traverseXMLfile((Element) child, level, XmlIndexList,
orderList);
            }

            // If root is a leaf node, it is labeled with its
            //level,order,pOrder then put in the index array.
            else {
                  if (child instanceof Element) {

                        ArrayList<XmlNode> levelBaseChild = new
ArrayList<XmlNode>();
                        if (XmlIndexList.size() == level + 1)
                              XmlIndexList.add(level + 1,
levelBaseChild);
                        pOrder = orderList[level][0];
                        orderList[level + 1][0]++;
                        int size = XmlIndexList.get(level + 1).size();
                        order = orderList[level + 1][0];
                        XmlNode newData = new XmlNode(level + 1, order,
pOrder, 0,(Element) child);
                        XmlIndexList.get(level + 1).add(newData);
                        if (size != 0)
                              XmlIndexList.get(level + 1).get(size -
1).rOrder = order;
                  }
            }
      }
}
```

Double Linked LBL

The structure of labeling algorithm of Double Linked LBL is similar with Single Linked LBL. The only difference is Double Linked LBL holds backward links, *lOrder*, within its node structure. The link lists which represent the levels of the tree are double

links lists. As the only difference on the software layer, the previous node's order number is hold and assigned as *lOrder* while labeling the current node. The source code is shown below.

```
public void traverseXMLfile(Element root, int level,
Vector<ArrayList<XmlNode>> XmlIndexList, int orderList[][]) {
      int pOrder=0;
      int order=0;

      // gets the children of the root node
      NodeList children = root.getChildNodes();
      ArrayList<XmlNode> levelBase = new ArrayList<XmlNode>();
      if(XmlIndexList.size()==0)
            XmlIndexList.add(level, levelBase);
      level++;
      if(XmlIndexList.size()<level+1)
      XmlIndexList.add(level, levelBase);
      // if the node is not a leaf node, it is put in the array list
      with //labels
      if(root instanceof Element){

            pOrder=orderList[level-1][0];
            orderList[level][0]++;
            order=orderList[level][0];
            int size=XmlIndexList.get(level).size();
            XmlNode newData = new XmlNode(level,order,pOrder,0,0,root);
            XmlIndexList.get(level).add(newData);
            if(size!=0)
                  XmlIndexList.get(level).get(size-1).rOrder=order;
            if(size>0)
                  XmlIndexList.get(level).get(size).lOrder=order-1;
      }

      for (int i = 1; i < children.getLength(); i=i+2) {
            Node child = children.item(i);
            boolean hasChldNodes = child.hasChildNodes();
            // If the root has chidren the 'label' method is called
            //recursively with assigning the 'child' element to root.
            if (hasChldNodes == true &&
child.getChildNodes().getLength()>2) {
                  traverseXMLfile((Element) child, level, XmlIndexList,
orderList);
            }

            // If root is a leaf node, it is labeled with its
            //level,order,pOrder then put in the index array.
            else {
                  if (child instanceof Element) {
                        ArrayList<XmlNode> levelBaseChild = new
ArrayList<XmlNode>();
                        if(XmlIndexList.size()==level+1)
                              XmlIndexList.add(level+1,
levelBaseChild);

                        pOrder=orderList[level][0];
                        orderList[level+1][0]++;
                        int size=XmlIndexList.get(level+1).size();
                        order=orderList[level+1][0];
```

```
                        XmlNode newData = new
XmlNode(level+1,order,pOrder,0,0,(Element)child);
                        XmlIndexList.get(level+1).add(newData);
                        if(size!=0)
                              XmlIndexList.get(level+1).get(size-
1).rOrder=order;
                        if(size>0)

      XmlIndexList.get(level+1).get(size).lOrder=order-1;
                        }
                  }
            }
      }
```

Space Requirements

Space requirement is calculated after labeling all the datasets. The amount of space that is required for a dataset shows the sum of all nodes of a dataset. For the range of the labels, integer values are used. An integer equals to 4 bytes in Java and the maximum range is +2,147,483,647.

For each scheme the space requirement of the algorithms is calculated as *number_of_label_parts x 4bytes x number_of_elements.*

Containment scheme uses 3parts x 4 bytes = 12 bytes for each element.

Basic LBL is also uses 3parts x 4 bytes = 12 bytes for each element, because it uses 3-part labels as containment labeling scheme.

Single Linked LBL uses 4parts x 4 bytes = 16 bytes for each label.

Double Linked LBL uses 5 parts x 4 bytes = 20 bytes.

Query Performance

The query performance tests are performed on determining the relationships of the algorithms. In this test 5 cases are evaluated; querying P-C, A-D, sibling, forward order and backward order.

Parent-Child Querying

For the parent-child querying the level and order information is taken from the user and the children of the node are listed. In Basic LBL it is coded as;

```
public void queryPC(int rOrder,int rLevel,ArrayList<XmlNode>
XmlIndexList){
            for (int i=0;i<XmlIndexList.size();i++){
                  if(XmlIndexList.get(i).level==rLevel+1 &&
XmlIndexList.get(i).pOrder==rOrder){
```

```
      System.out.println("Child:"+XmlIndexList.get(i).XMLnode.getNodeN
ame());

      System.out.println(XmlIndexList.get(i).level+"."+XmlIndexList.ge
t(i).order+"."+XmlIndexList.get(i).pOrder);
                  }
            }
      }
```

Single Linked LBL and Double Linked LBL get the same level and order information and searches only the next level of the tree for the children of the node. Here the *"vector of link lists"* structure gains performance with reducing the range of nodes to be searched. The determination of this relation is coded in these two versions as below;

```
public void queryPC(int rOrder,int rLevel,Vector<ArrayList<XmlNode>>
XmlIndexList){
      for(int i=0;i<XmlIndexList.get(rLevel+1).size();i++){
            if(XmlIndexList.get(rLevel+1).get(i).pOrder==rOrder)
      System.out.println(XmlIndexList.get(rLevel+1).get(i).XMLnode.get
NodeName());
      }
}
```

Ancestor-Descendent Querying

In the test, the descendants of a given node are queried. The level and the order number of the node are taken from the user as *rOrder* and *rLevel*. Then the all descendant nodes are found with *getAllChildren* method and put in an array list, *children*.

Its source code in Basic LBL is;

```
public void queryAD(int rOrder, int rLevel, ArrayList<XmlNode>
XmlIndexList) {
      Element parent = null;
      for (int t = 0; t < XmlIndexList.size(); t++)
            if (XmlIndexList.get(t).level == rLevel
                        && XmlIndexList.get(t).order == rOrder)
                  parent = XmlIndexList.get(t).XMLnode;
      int s = parent.getChildNodes().getLength();
      if (s > 2) {
            ArrayList<int[][]> children = new ArrayList<int[][]>();
            XmlTreeLabel.getAllChildren((Element) parent, XmlIndexList,
                        children);
            for (int t = 0; t < children.size(); t++)
                  System.out.println("child:" + children.get(t)[0][0] +
"-"
                              + children.get(t)[0][1]);
```

```
	}
}


public static void getAllChildren(Element node,
		ArrayList<XmlNode> XmlIndexList, ArrayList<int[][]>
children) {
	NodeList chList = node.getChildNodes();

	for (int i = 1; i < chList.getLength(); i = i + 2) {
		Node child = chList.item(i);
		int tmp[][] = { { 0, 0 } };
		tmp[0][0] = getLevel(child, XmlIndexList);
		tmp[0][1] = getOrder(child, XmlIndexList);
		children.add(tmp);
		boolean hasChldNodes = child.hasChildNodes();
		if (hasChldNodes == true &&
child.getChildNodes().getLength() > 2) {
			getAllChildren((Element) child, XmlIndexList,
children);
		}
	}
}
```

Its source code in Single Linked LBL and Double Linked LBL is;

```
public void queryAD(int rOrder,int rLevel,Vector<ArrayList<XmlNode>>
XmlIndexList){
	ArrayList<int[][]> children = new ArrayList<int[][]>();

	XmlTreeLabel.getAllChildren(rLevel, rOrder, XmlIndexList,
children);

	for (int s = 0; s < children.size(); s++)
		System.out.println(children.get(s)[0][0] + "-"
				+ children.get(s)[0][1]);
}



public static void getAllChildren(int rootLevel, int rootOrder,
		Vector<ArrayList<XmlNode>> XmlIndexList, ArrayList<int[][]>
children) {
	int childLevel = rootLevel + 1;
	if (XmlIndexList.size() != childLevel)
		for (int i = 0; i < XmlIndexList.get(childLevel).size();
i++) {
			if (XmlIndexList.get(childLevel).get(i).pOrder ==
rootOrder) {
				int tmp[][] = { { 0, 0 } };
				tmp[0][0] = XmlIndexList.get(rootLevel +
1).get(i).level;
				tmp[0][1] = XmlIndexList.get(rootLevel +
1).get(i).order;
				int childOrder = tmp[0][1];
				children.add(tmp);
				if (childLevel + 1 < XmlIndexList.size()) {
					getAllChildren(childLevel, childOrder,
XmlIndexList,children);
				}
			}
```

```
            }
}
```

Sibling Querying

While detecting the sibling relationship, the order and level of the node are taken from the user. The nodes whose *level* and *pOrder* are equal are found out as siblings.

It is coded in Basic LBL as;

```
public void querySib(int order, int level, ArrayList<XmlNode>
XmlIndexList) {
      int pOrder = 0;
      for (int t = 0; t < XmlIndexList.size(); t++) {
            if (XmlIndexList.get(t).level == level
                        && XmlIndexList.get(t).order == order)
                  pOrder = XmlIndexList.get(t).pOrder;
      }
      for (int t = 0; t < XmlIndexList.size(); t++) {
            if (XmlIndexList.get(t).level == level
                        && XmlIndexList.get(t).pOrder == pOrder
                        && XmlIndexList.get(t).order != order) {
                  System.out.println("Child:"
                              +
XmlIndexList.get(t).XMLnode.getNodeName());
                  System.out.println(XmlIndexList.get(t).level + "."
                              + XmlIndexList.get(t).order + "."
                              + XmlIndexList.get(t).pOrder);
            }
      }
}
```

Also in second and third versions, only the same level nodes are searched for siblings. Its code in Single Linked LBL and Double Linked LBL is below;

```
public void querySib(int order,int level,Vector<ArrayList<XmlNode>>
XmlIndexList){
      for(int i=0;i<XmlIndexList.get(level).size();i++){
      if(XmlIndexList.get(level).get(i).order!=order&&XmlIndexList.get
(level).get(i).pOrder==XmlIndexList.get(level).get(order-1).pOrder)
      System.out.println(XmlIndexList.get(level).get(i).XMLnode.getNod
eName());
      }
}
```

Order Querying

The order numbers of the labels make it easy to find the document orders of the nodes. For this issue Basic LBL relabels the order number of the nodes after each insert or delete. So it aims to keep all consecutive nodes in order. For the same issue Single

and Double Linked LBL have a different approach. They aim not to relabel the nodes after inserts and deletes. These two versions give a new, unused order number for a new inserted node. And the arrangement of the order relation is set up with links to next nodes. These links provide a quick detection of ordering. The main difference of Double Linked LBL from its previous version is that, it also arranges backward links between nodes.

In the performance tests, two kind of ordering test are performed; forward order querying and backward order querying. In forward order querying the node is taken from the user and the nodes which are in the following order are returned. While searching the nodes, Basic LBL uses the incremental order numbers. In the code below, the *rOrder* is assigned to the order number which is requested by the user. After finding that node, *rOrder* number is incremented by 1, and then the search is made for the new order number. This loop continues until all sibling nodes are found.

```
public void queryFOrd(int rOrder, int rLevel,
            ArrayList<XmlNode> XmlIndexList) {
      int pOrder = 0;
      for (int t = 0; t < XmlIndexList.size(); t++) {
            if (XmlIndexList.get(t).level == rLevel
                        && XmlIndexList.get(t).order == rOrder)
                  pOrder = XmlIndexList.get(t).pOrder;
      }
      for (int t = 0; t < XmlIndexList.size(); t++) {
            if (XmlIndexList.get(t).level == rLevel
                        && XmlIndexList.get(t).order == rOrder + 1
                        && XmlIndexList.get(t).pOrder == pOrder) {
                  System.out.println("Child:"
                              +
XmlIndexList.get(t).XMLnode.getNodeName());
                  System.out.println(XmlIndexList.get(t).level + "."
                              + XmlIndexList.get(t).order + "."
                              + XmlIndexList.get(t).pOrder);
                  rOrder = XmlIndexList.get(t).order;
            }
      }
}
```

Single and Double LBL uses the *rOrder* parts of the labels. They begin with the order number that the user is requested. It checks the rOrder of the label, then sets the rOrder number as new order number and searches for the new order. It continues to this loop until all the sibling nodes are listed.

```
public void queryFOrd(int rOrder,int rLevel,Vector<ArrayList<XmlNode>>
XmlIndexList){
```

```
      int nextOrder=XmlIndexList.get(rLevel).get(rOrder-1).rOrder;
      int parent=XmlIndexList.get(rLevel).get(rOrder-1).pOrder;
      while(nextOrder>0){
            if(XmlIndexList.get(rLevel).get(nextOrder-
1).pOrder==parent){

      System.out.println(XmlIndexList.get(rLevel).get(nextOrder-
1).level+"."+XmlIndexList.get(rLevel).get(nextOrder-1).order+"-
"+XmlIndexList.get(rLevel).get(nextOrder-1).XMLnode.getNodeName());
                  nextOrder=XmlIndexList.get(rLevel).get(nextOrder-
1).rOrder;
            }
      }
}
```

For backward querying only Double Linked LBL handles the performance bottleneck and provides a backward ordering relationship with its backward links. Here double link lists are used for each label. The code below is used for backward querying in Double Linked LBL;

```
public void queryBOrd(int rOrder,int rLevel,Vector<ArrayList<XmlNode>>
XmlIndexList){
      int preOrder=XmlIndexList.get(rLevel).get(rOrder-1).lOrder;
      int parent=XmlIndexList.get(rLevel).get(rOrder-1).pOrder;
      while(preOrder>0){
            if(XmlIndexList.get(rLevel).get(preOrder-
1).pOrder==parent){
            System.out.println(XmlIndexList.get(rLevel).get(preOrder-
1).level+"."+XmlIndexList.get(rLevel).get(preOrder-1).order+"-
"+XmlIndexList.get(rLevel).get(preOrder-1).XMLnode.getNodeName());
                  preOrder=XmlIndexList.get(rLevel).get(preOrder-
1).lOrder;
            }
      }
}
```

Update Performance

To insert a node to the XML, the parent node and the sibling node are asked to the user. Then the new data, *parent_level+1* and new *order* is sent to insert method to convert it to a new node with its label. Basic LBL gives *sibling_order+1* as the new node's order number and relabels the following nodes. Single and Double Linked LBL gives a new order number to the node and assigns the new node's *order* number to its sibling's *rOrder* or *lOrder* parts.

The source codes of this insertion part for each version are as below;

Basic LBL

```
public void insertToXml(int newLevel, int newOrder, int pOrder,
            Element newNode, ArrayList<XmlNode> XmlIndexList) {

      XmlNode newData = new XmlNode(newLevel, newOrder, pOrder,
newNode);
      XmlIndexList.add(newData);
      for (int i = 0; i < XmlIndexList.size(); i++) {
            while (i < XmlIndexList.size()) {
                  if (XmlIndexList.get(i).level == newLevel
                              && XmlIndexList.get(i).order >= newOrder
                              && XmlIndexList.get(i).XMLnode !=
newNode)
                        XmlIndexList.get(i).order =
(XmlIndexList.get(i).order) + 1;

                  if (XmlIndexList.get(i).level == newLevel + 1
                              && XmlIndexList.get(i).pOrder >=
newOrder)
                        XmlIndexList.get(i).pOrder++;
                  i++;
            }
            break;
      }
}
```

Single Linked LBL

```
public void insertToXml(int newLevel,int pOrder, Element
newNode,Vector<ArrayList<XmlNode>> XmlIndexList, int orderList[][],int
sOrder) {
      int rOrder=0;
      orderList[newLevel][0]++;
      int order = orderList[newLevel][0];
      if(sOrder!=0){
            rOrder=XmlIndexList.get(newLevel).get(sOrder-1).rOrder;
            XmlIndexList.get(newLevel).get(sOrder-1).rOrder = order;
      }
      else{
            rOrder=1;
            orderList[newLevel][1]=order;
      }
      XmlNode newData = new XmlNode(newLevel,
order,pOrder,rOrder,newNode);
      XmlIndexList.get(newLevel).add(newData);
}
```

Single Linked LBL

```
public void insertToXml(int newLevel, int pOrder, Element newNode,
            Vector<ArrayList<XmlNode>> XmlIndexList, int orderList[][],
            int sOrder) {
      int lOrder = 0;
      int rOrder = 0;
      orderList[newLevel][0]++;
      int order = orderList[newLevel][0];
      if (sOrder != 0) {
            rOrder = XmlIndexList.get(newLevel).get(sOrder - 1).rOrder;
```

```
            XmlIndexList.get(newLevel).get(sOrder - 1).rOrder = order;
            lOrder = XmlIndexList.get(newLevel).get(sOrder - 1).order;
            XmlIndexList.get(newLevel).get(sOrder).lOrder = order;
      } else {
            rOrder = orderList[newLevel][1];
            XmlIndexList.get(newLevel).get(orderList[newLevel][1]-
1).lOrder = order;
            orderList[newLevel][1] = order;
      }
      XmlNode newData = new XmlNode(newLevel, order, pOrder, lOrder,
rOrder,newNode);
      XmlIndexList.get(newLevel).add(newData);
}
```