VICTORIA UNIVERSITY

ACCESS CONTROL MODEL AND LABELLING SCHEME FOR EFFICIENT QUERYING AND UPDATING XML DATA

A Thesis Submitted by

MAGGIE DUONG

BSc. (Hons), Melbourne

For the degree of

Doctor of Philosophy

2010

School of Computer Science & Mathematics

VICTORIA UNIVERSITY

Abstract

As XML continues to be utilised in various fields of information technology, an urgent need for managing XML documents is being felt. Considering day-to-day operations such as updating, accessing and/or querying XML documents, these tasks need to be accurate, quick to carry out, easy to use and more importantly safe from unauthorized accesses. This requires an integrated scheme that can facilitate query processing and determine what kind and/or part of information can be displayed, updated and/or modified by different types of users.

In order to facilitate query processing for XML data, several path indexing, labelling, and numbering schemes have been proposed. However, if there is frequent demand for XML data to be updated, most of these approaches will need to re-compute existing labels, which is rather time consuming. Some other existing approaches tried to solve this problem by reserving spaces or reserving codes to minimize the cost of re-labelling. However, when all reserved spaces or reserved codes are used up, re-labelling will undoubtedly be required. Likewise, many existing access controls use node filtering or querying rewriting techniques. These techniques require rather time-

consuming processes such as parsing, labelling, pruning and/or rewriting queries into safe ones each time a user requests a query or takes an action.

In this thesis, we make two major contributions that help the tasks of querying, updating, and managing XML data become quick and safe. First, we propose a new labelling scheme for dynamic XML data that supports the representation of the ancestor – descendant relationship and sibling relationship between nodes. Our unique way of labelling codes can also help users easily determine the depth (level) of the XML tree. Moreover, it also supports the process of updating XML data without the need of re-labelling existing labels, hence facilitating fast update. Some experimental works have been conducted to show its effectiveness.

Secondly, for XML security purposes, we propose a fine-grained access control model, named SecureX, which supports read and write privileges. With our novel access control concept, various access types are introduced, including those for determining if a user has the right to change XML structure. SecureX ensures that, crucial information will be accessible only to authorized entities. Furthermore, SecureX can be integrated well with a dynamic labelling scheme to eliminate repetitive labelling and pruning processes when determining a user view. This brings about advantages of speeding up searching and querying processes.

When comparing to a traditional node filtering technique, our integrated access control model takes less processing steps. Experiments have shown effectiveness of our approach.

Keywords: Access control, XML query, XML update, labelling scheme, path index, query processing.

PUBLICATION

Duong, M. and Zhang, Y. (2008): Dynamic Labelling for XML Data Processing. To be appeared in the Proceedings of the 7th International Conference on Ontologies, DataBases, and Applications of Semantics. (ODBASE'08), 2008.

Maggie Duong and Yanchun Zhang (2008). An Integrated Access Control for Securely Querying and Updating XML. In Proceedings of the Nineteenth Australasian Database Conference (ADC2008), Wollongong, Australia. Conferences in Research and Practice in Information Technology, Vol. 75.

Maggie Duong and Yanchun Zhang (2005). LSDX: A New Labelling Scheme for Dynamically Updating XML Data. In Proceedings of the 16th Australasian Database Conference, Jan 31 – Feb 3, 2005, Newcastle, Australia. Conferences in Research and Practice in Information Technology, Vol. 39.

CERTIFICATION OF RESEARCH

I, Maggie Duong, declare that the PhD thesis entitled "Access Control Model and Labelling Scheme for Efficient Querying and Updating Xml Data" is no more than 100,000 words in length including quotes and exclusive of tables, figures, appendices, bibliography, references and footnotes. This thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated, this thesis is my own work.

_____ _____

Signature Date

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Professor Yanchun Zhang for his guidance, help and great encouragement during the course of my PhD. His patience, constructive criticism, professional guidance and the ability to draw the best results out of his students have been integral to the success of this work and to my education as a researcher.

I also would like to thank many anonymous referees for their comments on our papers which are the basics of this thesis.

I am grateful to the School of Computer Science and Mathematics for providing the finance support to travel to conferences and for offering me a research scholarship. Thanks also go to Postgraduate Research Office for giving me a scholarship to do PhD degree and Faculty of Health, Engineering and Science for giving me an extended scholarship.

Finally, I would like to express gratitude to my husband, Danh, for his support and encouragement throughout the course of my study. To him, I dedicate this thesis.

Table of Contents

List of Figures

List of Tables

1

Introduction

1.1 Overview and Motivation

With the advanced characteristics of XML, more and more XML documents have been used to display and exchange information on the web. XML (eXtensible Mark-up Language) is used to describe and to store information in such a way that data can be exchanged easily over the internet. Since data is stored in various computer systems and in different, incompatible formats, exchanging these data over the Internet can take a great deal of time. XML is a perfect answer for this problem. A significant characteristic of XML is platform independent. It does not make any difference if users use Windows, Macintosh or UNIX. Data written in XML can be exchanged among computers without having any problem. Information written in XML can also be transmitted regardless of software and hardware used.

As XML continues to be utilized in various fields of information technology, an urgent need for managing XML documents is being felt. Considering day to day operations such as updating, accessing and/or querying XML data,

these tasks need to be accurate, quick to carry out, easy to use and more importantly safe from unauthorized accesses. For instance, electronic commerce transactions require clarity and enforcement of security controls, ensuring that crucial information will be accessible only to authorized entities. This requires a method to determine what kind and/or part of information can be displayed, updated and/or modified by different types of users. Another good example to illustrate the importance of this topic is to think about an organization in which sensitive information should only be accessed by authorized people.

```
<employee>
<dept name="CompSci">
    <staff>
     <name>
          <lastname>Wilkinson</lastname>
          <firstname>John</firstname>
     </name>
     <address>12 Lynn Marree St</address>
     <DOB>24 May 1966</DOB>
     <h_phone>98664356</h_phone>
     <office>D544</office>
     <extension>4566</extension>
     <email>Wilkinson.John@vu.edu.au</email>
```

```
      <job_title>Assoc. Professor</job_title>
      <salary>59300</salary>
   </staff>
   <staff>
      <name>
           <lastname>Bestor</lastname>
           <firstname>Angela</firstname>
      </name>
      <address>188 Princess St</address>
      <DOB>24 May 1954</DOB>
      <h_phone>98664444</h_phone>
      <office>D644</office>
      <extension>4522</extension>
      <email>Bestor.Angela@vu.edu.au</email>
      <position>Lecturer</position>
      <salary>42500</salary>
   </staff>
</dept>
…
</employee>
```

Figure 1.1. Staff.xml

Let us consider a University `staff` database, (see Figure 1.1), where general information such as `name, office, email`, etc are available to public. `Home address, home phone` and `DOB` are confidential information and should only be available to administrators. `Salary` is a protected field and only available to authorized people.

Now considering two users, a student and an administrator with access levels `public` and `private` respectively. When searching for staff details, these two users will have different views.

Student's view:

```
<staff>
  <name>
     <lastname>Wilkinson</lastname>
     <firstname>John</firstname>
   </name>
   <office>D544</office>
   <extension>4566</extension>
   <email>Wilkinson.John@vu.edu.au</email>
   <position> Assoc. Professor </position>
  </staff>
  <staff>
     ....
```

```
</staff>
```

Administrator's view:

```
<staff>
   <name>
        <lastname>Wilkinson</lastname>
        <firstname>John</firstname>
   </name>
   <address>12 Lynne Maree St</address>
   <DOB>24 May 1966</DOB>
   <h_phone>98664356</h_phone>
   <office>D544</office>
   <extension>4566</extension>
   <email>Wilkinson.John@vu.edu.au</email>
   <position> Assoc. Professor </position>
</staff>
<staff>
   ....
</staff>
```

A student should only receive general information of staff but not `home phone`, `home address` etc. Similarly, an administrator can obtain general information of a staff, as a student does, plus other private information.

However, the administrator is unable to view `salary` due to possessing insufficient access level.

Considering when a staff has changed his/her name after a marriage or moving to a new place, this information needs to be updated. Clearly, public and private fields like `name`, `address`, `phone`, `email` etc should be able to be updated by that staff or by an administrator. Of course, s/he cannot alter protected fields like `salary`. If staffs have access to private fields, then, there will be an issue emerging. It is conceptually right that one can access his/her private information. However, no staff would want other colleagues seeing their confidential information. Thus, an explicit rule is needed to state that one can access one's own confidential details but not those of others.

To defend confidentiality for XML data, several access control models have been proposed. For instance, De Capitani di Vimercati, Marrara, and Samarati (2005), Yokoyama, Ohta, Katayama and Ishikawa (2005), Lee, Lee and Liu (2003), Yu, Srivastava, Lakshmanan, and Jagadish (2002), Fan, Chan and Garofalakis (2004), Bertino and Ferrari (2002), Damiani, De Capitani di Vimercati, Paraboschi and Samarati (2001), Bertino, Castano and Ferrari (2001), Damiani, De Capitani di Vimercati, Paraboschi and Samarati (2000) support read privileges.

Damiani, Fansi, Gabillon and Marrara (2007, 2008), Gabillon (2004), Damiani, De Capitani di Vimercati, Paraboschi and Samarati (2002), Kudo and Hada (2000) support both read and write privileges. However, these models cannot explicitly define access rules for the above issue. On the other hand, models that support write privileges do not consider or define clear rules in which XML structure and DTD may be changed due to updating operations.

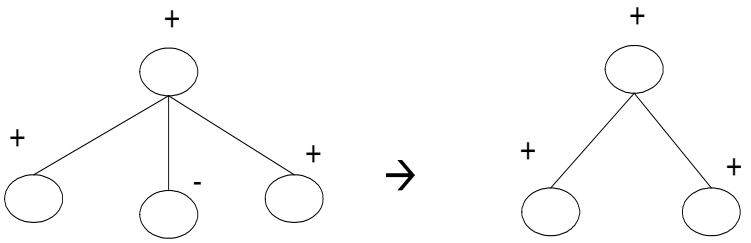


Figure 1.2 Pruning example

In general, these approaches either use node filtering and/or query rewriting techniques. These require repetitive node labelling, then pruning processes and/or rewriting query each time a user sending a query to determine a user view. This degrades the query performance. Examples of node labelling and pruning can be found at Figure 1.2. [Bertino, Castano and Ferrari 2001]. Access authorizations are determined by labelling tree nodes

with a permission (+), or a denial (-) then pruning trees are based on associated signs.

Since queries navigate XML data via path expressions, indexes can be used to accelerate queries. A well constructed index will allow a query to bypass the need of scanning the entire table for results. Although, there are a number of indexing methods which have been proposed to facilitate query processing, these methods do not consider data confidentiality. Likewise, existing XML access controls have not been developed to integrate with existing indexing methods to speed up the search.

To the best of our knowledge, Yu, Srivastava, Lakshmanan and Jagadish (2002) use a compressed accessibility map to quickly determine if a user has the right to access an XML data item. However, drawbacks of this work are that it only supports a single user and an access type at a time; it is space inefficient due to separate Compressed Accessibility Map (CAM) is needed for each user and access type; and it only supports read action.

There have been lots of works focused on XML query processing; some other works developed labelling schemes which play an important role in speeding up query processing. These techniques vary from path indexing to numbering schemes. (Details of these techniques will be discussed in

Related Works). For instance, the works by O'Neil, O'Neil, Pal, Cseri, Schaller and Westbury (2004), Amato, Debole, Rabitti and Zezula (2003), Grust (2002), Cooper, Sample, Franklin, Hjaltason, Shadmon (2001), Meuss and Strohmaier (1999) used path indexing for searching XML data.

Two other approaches are employed in *Numbering Schemes*. One is called *region-based numbering scheme* [Wu, Lee and Hsu (2004), Amagasa, Yoshikawa and Uemura (2003), Li and Moon (2001), etc] and the other one is *prefix-based numbering scheme* [Li and Ling (2005), Cohen, Kaplan and Milo (2002), Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang (2002), and Kaplan, Milo and Shabo, etc]. However, most of proposed labelling schemes are costly due to the need of re-calculating or re-labelling existing nodes whenever XML documents being updated.

The major problem:

If deletion and/or insertion occur regularly in the XML data, these techniques would need expensive re-computing of affected labels. (Details of these approaches will be discussed in chapter 3).

In our view, an effective labelling scheme needs to be:

(i) Compact; total lengths of labels are as small as possible.

(ii) Dynamic, being able to update XML data dynamically without re-labelling or re-calculating value of existing nodes.

(iii) Last but not least, facilitating the identification of various relationships between nodes.

Remark:

Prior to querying the database, another significant issue must be considered, i.e. security. Certain data sets in any database need to be protected so that these data can only be accessed by authorised people. This is a challenging research problem. The focus of current schemes is either to facilitate query performance or to control access to XML documents. As far as accessing control concerns, these schemes are not practical. It is because when an XML document is queried, it is necessary to determine if that particular user has access authorisation for the XML document being queried. The access rules may vary for different users, different XML files in the database, and/or different levels, nodes/elements in the XML documents.

Although one can use one scheme on top of the other, however, every time a user sends a query, access authorization has to be checked first (from accessing rules store in a file or a scheme). Once access is allowed, an index/labelling scheme is used to execute the query process and return the result. A bottleneck in the system performance is created if these steps are

repeated every time a user sending a query. As a result, a perfect scheme is needed to manage dynamic XML documents effectively. For this purpose, we considered the following features in our design:

- An index algorithm that facilitates XML query processing.
- A static labelling scheme – There is no need to re-label even if the XML document needs to be updated.
- Updating actions can be carried out in an effective manner which reduces computation cost and querying response time, hence facilitating fast update actions, including add, modify, and delete.
- Controlling accessing to XML document - sensitive information shall be protected and could only be accessed by authorised people.
- Improve query performance – The query performance shall be improved by combining indexing algorithm and accessing rules of an XML document in one labelling scheme. This combined labelling scheme will help query optimiser to bypass the tasks of verifying access rule and searching for result in two different files or even two difference places every time a user sends a query.

1.2 Claims of the Thesis

Frequently re-computing large amount of elements each time XML data is updated will take time and will reduce performance. It will also raise the

cost of renumbering. Although, there are a number of indexing methods which have been proposed to facilitate query processing, these methods do not consider data confidentiality. Likewise, existing XML access controls have not been developed to integrate with existing indexing methods to speed up the search.

This thesis covers and makes these major contributions to the research community as describe in the following areas.

First, we propose a labelling scheme called LSDX that supports updating XML data dynamically without the need of re-labelling existing nodes, hence facilitating fast update. This dynamic labelling scheme supports all important axes in XPath such as parent, child, ancestor, descendant, previous – sibling, following – sibling, previous nodes, following nodes.

Secondly, we proposed an improved version of LSDX, namely Com-D labelling scheme. In this labelling scheme, we develop a new technique to label XML tree to make it small and more compact. The total lengths of labels are reduced significantly comparing with existing dynamic labelling schemes. Moreover, it does not matter where new nodes should be inserted or how many of new nodes are added. It is guaranteed that none of existing

nodes needs to be re-labelled and no re-calculation is required. These will facilitate fast update as well as enhancing query processing.

Thirdly, we propose a fine-grained access control model, which supports read and write privileges. Our model supports various access types. Cases such as user's information is available to self-access only is also managed sensibly, thus, the task of defining access rules can be done judiciously. We also consider update-types that help to determine if a user has the right to change XML structure. In addition, our access control model can be integrated well with an indexing or labelling scheme to eliminate the repetitive labelling and pruning processes when determining a user view. This brings about the advantage of speeding up searching and querying processes.

To demonstrate, we integrate our access control model with a dynamic labelling scheme and compare its processing steps with a traditional access control model. We show by analysis that, our model requires less processing steps and provides a shortcut for the task of evaluating if a node is accessible for a particular user. Our experiments show that, our model is superior in term of query performance when comparing with exiting works that use node filtering techniques.

In details, our contributions to web-based applications and web users are effective indexing algorithms that provide following advantages:

- Space efficient – Indexing storage space is relatively small.
- Control access to XML documents – Sensitive information shall be protected and only be accessed by authorised entities.
- Support dynamic XML documents – In term of both querying and controlling access to XML documents.
- Speed up updating actions (add, update, delete) can be carried out in an effective manner, which shall reduce computation cost and querying response time, hence facilitating fast update, by eliminating the need of re-indexing even if the XML document needs to be updated.
- Improve query performance – By combining index algorithm and access rules in one labelling scheme, it will be the help for query optimiser to bypass the need of checking for access rule and search for result in two difference files or even two difference places every time a user sends a query.

Summary:

In summary, we have developed innovative labelling schemes for managing dynamic XML documents that makes the tasks of accessing/querying and updating XML data faster, and safe from unauthorized access.

- Our labelling schemes will control access to XML documents; sensitive information shall be protected and only be accessed by authorised entities.

- Support dynamic XML documents. There is no need to re-label even if the XML document needs to be updated. Thus, it will be in the help of speeding up updating actions.

- Add, update, delete operations can be carried out in an effective manner, which shall reduce computation cost and querying response time, hence facilitating fast update.

- Last but not least, it improves query performance. The utilisation of indexing algorithm and accessibility rules in one labelling scheme helps query optimiser bypassing the need of scanning the entire table for results.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents the basic concept necessary for understanding XML, how indexing works and how it can facilitate query processing. Overview of XML, XPath and XQuery are also presented in the subsections.

In chapter 3, we shall briefly discuss about some techniques that have recently been developed by a number of researchers. Those techniques vary from access controls for XML data, path indexing, numbering schemes, etc. We shall examine how they work and identify existing problems. The first part, section 3.1 discusses about related works in XML access control. The second part, section 3.2 discusses about related works in labelling schemes.

Chapter 4 provides an overview of our proposed labelling scheme, the LSDX. How its labelling technique works and how it supports the representation of ancestor – descendant relationships and sibling relationships between nodes. In this chapter, we will also describe how our LSDX can determine the depth (level) of the data tree. Finally, experiments for LSDX labelling scheme is presented.

Chapter 5 provides an overview of our compact labelling scheme, the Com – D and its advantages overs the existing dynamic labelling schemes. Experiments for Com-D labelling scheme is then followed. Experiments works were carried out in terms of label length, storage size, querying/insertion/updating performance, and re-labelling and response time.

Chapter 6 presents our access control model, SecureX for XML data. We shall demonstrate how it can be integrated with an XML labelling scheme. Then, our analysis will show how our access control can be better off comparing to traditional node filtering techniques. In section 6.3, we introduce write privileges of our access control model. Section 6.5 deals with conflict or undefined rules. Finally, in section 6.7, experimental works for SecureX, the access control for XML is presented. Experiments on queries performances, number of node scan versus number of node retrieved, response time, etc are also carried out.

Finally, Chapter 7 presents the conclusions and possible future work.

2

XML Background

In this chapter, we give a brief overview of XML, how indexing works and how it can facilitate XML query processing. In later subsections, we describe the basic concept necessary for understanding XML, XPath and XQuery.

```
<?xml version="1.0" encoding="UTF-8"?>
<Student_List>
   <Student>
      <Stud_Id>1234567</Stud_Id>
      <Name>Jennifer Fall</Name>
      <Address>123 Footscray Rd</Address>
      <Phone>98765432</Phone>
      <Year>2008</Year>
      <Course>
         <Course_Code>ABC4</Course_Code>
         <Course_Title>Bachelor of Science</Course_Title>
      </Course>
   </Student>
```

```
   <Student>
      <Stud_Id>2345678</Stud_Id>
      <Name>Matt Yang</Name>
      <Address>123 Spencer Rd</Address>
      <Phone>96894321</Phone>
      <Year>2008</Year>
      <Course>
         <Course_Code>ABC4</Course_Code>
         <Course_Title>Bachelor of Science</Course_Title>
      </Course>
   </Student>
</Student_List>
```

Figure 2.1. An XML data - newly added elements in shade.

2.1 XML Overview

XML stands for eXtensible Mark-up Language. It is a meta-mark-up language, similar to HTML (Hyper-Text Mark-up Language). Nevertheless, XML and HTML were designed for different purposes. "XML was designed to describe data and to focus on what data is while HTML was designed to display data and to focus on how data looks." -W3Schools, XML Tutorial.

XML is used to describe, to structure and to store information in such a way that data can be exchanged easily over the internet. In XML, tags are not predefined. XML allows users to create their own tags and to structure their own data in their preferred ways.

Since data is stored in various computer systems and in different, incompatible formats, exchanging these data over the Internet can be timed consuming and costly. XML is a perfect answer for this problem. Significant characteristics of XML can be described as follows:

- It is platform independent. It does not make any difference if users use Windows, Macintosh or UNIX. Data written in XML can be exchanged without having any problem.

- Information written in XML can also be transmitted regardless of software and hardware used.

XML documents use a self-describing and simple syntax. An example is shown in Figure 2.1.

In XML, all elements must have an opening tag and a closing tag. However, empty element is an exception which can end with "/>". Elements must be properly nested within each other. An example for this is given below.

```
<Name>Matt Yang</Name>
```

All elements can have sub elements (child elements). Sub elements must be correctly nested within their parent element:

```
<Course>
        <Course_Code>ABC4</Course_Code>
        <Course_Title>Bachelor of Science</Course_Title>
</Course>
```

The correct nesting of XML data is an aspect of *well-formedness*. However, well - formedness does not mean that data is correct. For instance, consider the following XML fragment:

```
<Student>
   <Stud_Id>2345678</Stud_Id>
</Student>
<Name>Matt Yang</Name>
```

Although the above XML data fragment has the correct XML syntax and is well - formed, it is incorrect. <Name> should belong to a particular <Student>. Thus, it should be nested inside <Student> element to describe its meaning correctly.

2.1.1 Tree Terminology

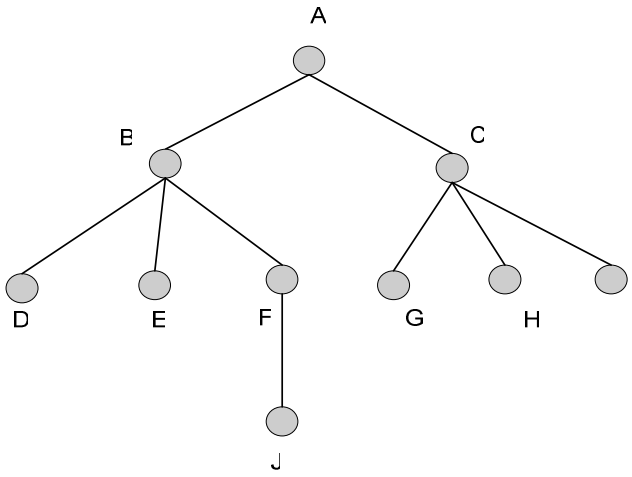


Figure 2.2. An example of a tree.

Figure 2.2 above is used to illustrate the tree terminology. Explanations are as follows:

- A is the root node (in XML known as document element).
- B is the parent of D, E and F.
- C is following sibling of B.
- D is preceding sibling of E.
- D, E and F are children of B.

- D, E, G, H, I and J are external nodes or leaves (has no children).
- A, B, C and F are internal nodes or non-leaves (has child/ren).
- The depth (level) of C is 1, H is 2.
- Height of tree is 3 (maximum depth).

a. In-order Traversal

In-order traversal is illustrated in Figure 2.3. Rule for in-order traversal can be described as follows:

- First visit left sub tree.
- Then visit root node.
- Then visit right sub tree.

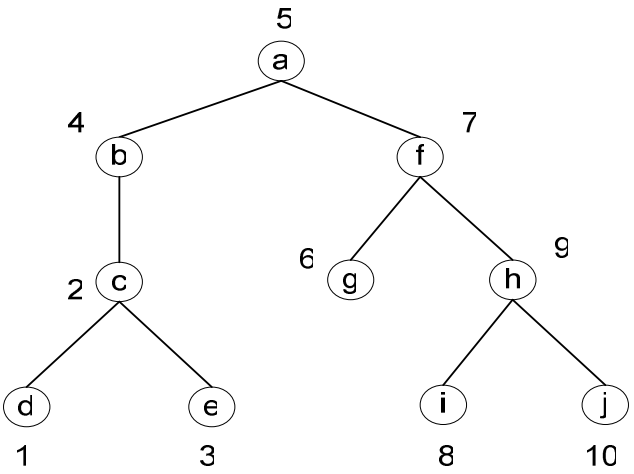


Figure 2.3. An example of in-order traversal

b. Pre-order Traversal

Pre-order traversal is illustrated in Figure 2.4. Rule for pre-order traversal can be described as follows:

- First visit root node.
- Then visit left sub tree.
- Then visit right sub tree.

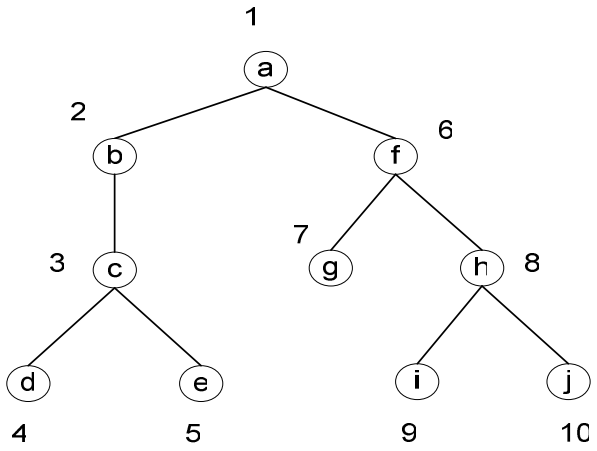


Figure 2.4. An example of pre-order traversal

c. Post-order Traversal

Post-order traversal is illustrated in Figure 2.5. Rule for post-order traversal can be described as follows:

- First visit left sub tree.
- Then visit right sub tree.
- Then visit root node.

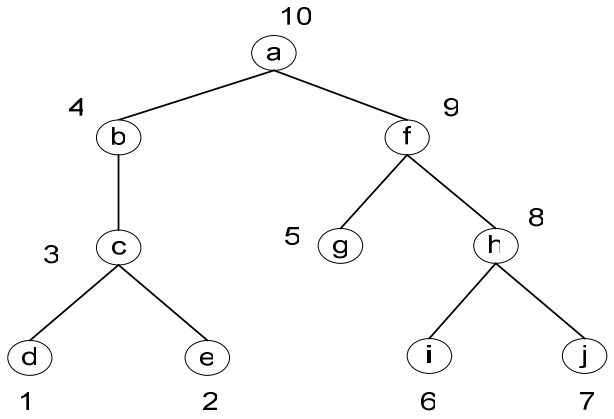


Figure 2.5. An example of post-order traversal

2.2 How Indexing Works

The traditional and most beneficial way to improve query performance requires the creation of an effective indexing scheme. A well - constructed index will allow a query to bypass the need of scanning the entire table for results. –[Kaelin, (2004)]. To make this possible, a unique label is assigned for each node in an XML tree. It is also quite essential to label nodes in such a way that can clearly show relationship between any two given nodes (such as ancestor – descendant relationship or sibling relationship). Once this is done, structural queries can be answered by only using the developed index. There is no need to access the actual documents. –[Lu and Ling (2004)].

Since queries navigate XML data via path expressions, it can be accelerated using an index. For this reason, many researchers have been trying to

develop the most efficient way to index XML data. Next, we show an example that requires the presence of an effective index.

In Figure 2.1, we have an XML document that stores details of students. (Ignore for the shaded elements for now). Each `Student` element represents a student. Every `Student` contains `Stud_Id` which corresponds to student ID, `Name` is student name, `Address` is address of the student, `Year` is enrolment year and `Course`, `Course_Code`, `Course_Title` represent course details.

If we want to get names of all students, the simple ways we will need to do is just using XPath expression and specifying the path of the element we want to retrieve. In this case, we can use the following XPath expression:

```
/Student_List/Student/Name
```

Nevertheless, if this document stores thousands of student records, it will take a great deal of time to traverse all elements in this document to find what we are looking for. Thus, indexing is extremely useful to bypass all unnecessary elements. We could directly access to those elements that we need.

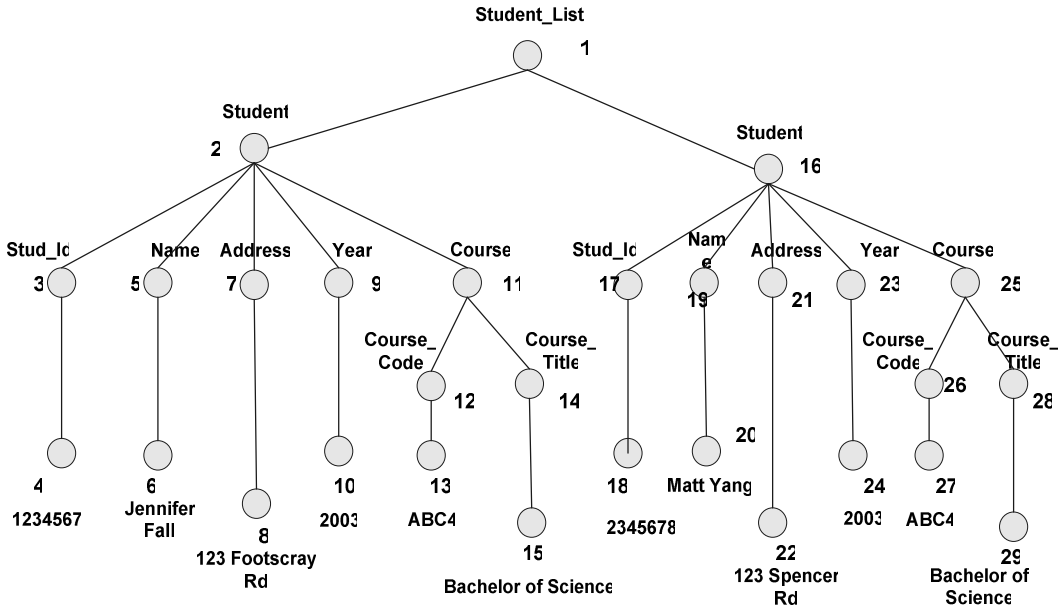


Figure 2.6. An example of indexing technique based on an XML in Figure 2.1 - Amato, Debole, Rabitti and Zezula (2003)

To demonstrate how indexing works for the above example, we use the indexing method from Amato, Debole, Rabitti and Zezula (2003) to label each element in our XML document. They assign '1' for the first element, '2' for second element, '3' for the third element and keep increasing 'number' when adding new element in that order. Figure 2.6 shows how this indexing method is represented in the data tree.

```
/Student_List-> {1}

/Student_List/Student-> {2, 16}

/Student_List/Student/Stud_Id-> {3, 17}
```

```
/Student_List/Student/Name-> {5, 19}

/Student_List/Student/Address-> {7, 21}

/Student_List/Student/Year-> {9, 23}

/Student_List/Student/Course-> {11, 25}

/Student_List/Student/Course/Course_Code-> {12, 26}

/Student_List/Student/Course/Course_Title-> {14, 28}
```

Figure 2.7. Path lexicon based on Figure 2.1

Based on the above indexing, we can have a Path lexicon as Figure 2.7.

This path lexicon table shows that all `name` elements are labelled as node '5' and node '19'. Student's `course` elements are labelled as node '11' and node '25'. Student's `course titles` are labelled as node '14' and '28'. When we want to access to any element of the XML document, we can go directly to the labelled node using this indexing. Thus, we shall reduce the time needed to traverse all unnecessary nodes in the XML document, hence facilitate query processing.

Remark:

While the above indexing technique can facilitate query processing, problems still exist. XML data have an intrinsic order. That means XML data

orders its nodes corresponding to the order in which a sequential read of the textual XML would encounter the nodes. – [Grust, (2002)]. Let us take an example to highlight one of problems of the mentioned indexing technique.

Suppose that our student details need to be updated, for example, school now wants to store student's contact number. This means we need to add a new element in our student data file for each student. To ensure that the newly added element is correctly described, we must add it inside the `<Student>` element.

At this point, problem emerges because children and/or siblings nodes of the `<Student>` element will need to be re-indexed. In particular, if we add it after the `<Address>` element and before the `<Year>` element, all elements from the first `<Year>` element downward need to be reindexed.

Now considering Figure 2.1 with the updated XML data, newly added elements are shaded. An updating problem is presented in a data tree shown in Figure 2.8. Newly added element is represented as dot lines. All nodes that need to be reindexed are marked with a cross.

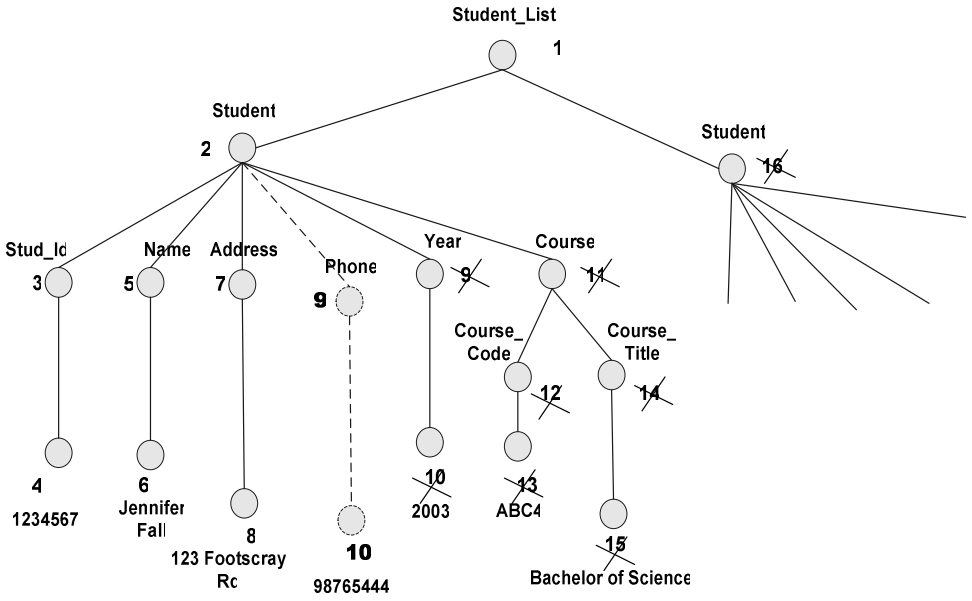


Figure 2.8. Updating problem

The above problem occurs quite often because data usually need to be updated. A query optimiser may use various techniques such as indexing, numbering or labelling schemes to improve query processing. However, if these techniques are not designed with the advantage of being capable of dynamically updating XML data in mind, when XML data need to be updated frequently, reindexing or relabelling will probably have to be done. This is a challenge for researchers who want to solve this problem.

2.3 An Overview of XPath

XML has emerged as a universal format for data representation and exchanging information on the web. Naturally, XML requires a query

language so that users can extract, convert and maintain XML data.

A few years ago, the World Wide Web Consortium developed a query language for XML documents called XPath. XPath is a declarative query language for XML. It provides a simple syntax for addressing parts of an XML document. XPath provides a decent selection capability.

XPath is a set of syntax rules used to define parts of an XML document. It helps to address and/or to navigate to those parts. XPath is a W3C Standard. Current version is 2.0. XPath uses path expressions to identify nodes in an XML document. These path expressions look like traditional file paths in a computer file system. An absolute location path starts with a forward slash (/). It denotes the document element (root node). A relative path does not start with any slash. In both cases, the location path consists of one or more location steps. Each location step is separated by a slash.

The XPath expression below is an absolute path. It will select all the Student elements of the `Student_List` element.

```
/Student_List
```

An example of a relative path is given below. It will return all students contact number.

```
Student/Contact_No
```

As XML content is stored in a hierarchy structure, XPath is employed to locate various parts of this hierarchy when needed. The way in which XPath operates requires users to specify a context node, e.g., `Student` node in the example above. From the context node, it proceeds in one of several directions in the hierarchy. These directions are called axes. It gets to desired node through a number of location steps.

The XPath expression below selects all the Name elements of all the Student elements of the `Student_List` element:

```
/Student_List/Student/Name
```

XPath defines a library of standard functions for working with strings, numbers and Boolean expressions. The XPath expression below selects all the Student elements that enrolled in the Year 2008:

```
/Student_List/Student[Year="2008"]
```

A path beginning with two slashes (//) means all elements in the document that meet a specified criteria will be selected even if they are at different levels in the XML tree. Consider the following XPath expression:

```
//Student
```

This will select all Student elements in the document.

Wildcard (*) can be used to select unspecified XML elements. For example, the following XPath expression selects all the child elements of all the Student elements of the `Student_List` element:

```
/Student_List/Student/*
```

However, XPath only supports Boolean, String, and Numeric data types and does not work effectively with case-insensitive strings or regular expressions. One cannot select part of a node or combine different results to produce new nodes. Finally, XPath does not provide the ability to build new data. –[Esposito et al. 2001].

To overcome these limitations, the World Wide Web Consortium then proposed a more refined version of XPath, which is called XQuery. XQuery is built on XPath expressions. XQuery improves on XPath's selection capabilities by adding support for more data types and by adding the ability to consider externally linked documents as a sub-tree of the existing document. –[Esposito et al. 2001].

2.4 An Overview of XQuery

XQuery is a query language developed by World Wide Web Consortium group for querying XML data. The current version of XQuery is 1.0. XQuery is built on XPath expressions. XQuery 1.0 and XPath 2.0 share much of the same expression syntax, the same data model and the same functions. Originally, XQuery is derived from an XML query language called Quilt. As its name suggests, Quilt borrowed features from several other languages, including XPath 1.0, XQL, XML-QL, SQL, and OQL. –[Boag, Chamberlin, Fernández, Florescu, Robie and Siméon (2004)]. XQuery attempts to utilise strength from several query languages and to take full advantage of their versatility. –[Robie, Chamberlin and Florescu (2000)].

In general, XML Query Language's ability is superior to XPath. XQuery improves on XPath's selection capabilities by adding support for more data. XQuery also provides a set of predefined namespace prefixes that can be

used in any query without an explicit declaration. For example, there is distinct syntax for XML Query, the syntax FLOWR:

- This name comes from the five principal instructions that are used: FOR, LET, ORDER-BY, WHERE and RETURN.

XML Query is intended to make such a query easier to work with, and to produce simple node-set, text, and data results on the spot.

Let us introduce some examples of XQuery to illustrate how it works. The following examples will use XML data shown in Figure 2.1. This XML data will be saved with the name: "StudentList.xml".

- Example 1: XQuery uses XPath node paths to extract data from XML documents.

The following query:

```
doc("StudentList.xml")/Student_List/Student/Name
```

Will return all student names:

```
<Name>Jennifer Fall</Name>
```

```
<Name>Matt Yang</Name>
```

In the example above, the function doc ("StudentList.xml") is used to open the XML document. The XPath node path `/Student_List/Student/Name` is used to extract all the `Name` elements. (`/Name` selects the `Name` element, `/Student` selects all the `Student` elements under the `Student_List`).

Recall on XPath, the above location path `(/Student_List/Student/Name)` can be rewritten as `//Name` to select all student names.

Next, we will use FLOWR syntax to query XML data.

- Example 2: We want to select all students enrolled in 2008, display their names and course titles.

The query for this example will be:

```
FOR $stud IN doc(StudentList.xml)//Student,
    $course IN $stud/Course
WHERE $stud/Year = "2008"
```

```
RETURN
    <Student>
      <Name>{$stud/Name}</Name>
      <Course_Title>{$course/Course_Title}</Course_Title>
    </Student>
```

In the above example, the FOR clause generates a list of bindings in which $stud is bound to individual Student elements in the document found at the given URL; `$course` is bound to individual `Course` elements that are descendants of `$stud`. The WHERE clause retains only those records in which `Year = "2008"`, and the RETURN clause generates the result of the query operation and returns a set of XML nodes. The following is the result of this query:

```
<Student>
    <Name>Jennifer Fall</Name>
    <Course_Title>Bachelor of Science</Course_Title>
</Student>
<Student>
    <Name>Matt Yang</Name>
    <Course_Title>Bachelor of Science</Course_Title>
</Student>
```

- Example 3: We want to select all student names that have student ID greater than 1000000. Names will be ordered ascendingly.

```
FOR $stud IN doc(StudentList.xml)//Student
WHERE $stud/Stud_Id > 1000000
ORDER BY $stud/Name
RETURN $stud/Name
```

The above query will return the following nodes:

```
<Name>Jennifer Fall</Name>
<Name>Matt Yang</Name>
```

In the example above, the FOR clause selects all Student nodes into a variable called $stud. The WHERE clause selects only the `$stud nodes` (Student nodes) with `Stud_Id` elements have value greater than 1000000. The ORDER BY clause orders the `$stud nodes` (Student nodes) by Name elements (Name nodes). The RETURN clause returns the Name nodes.

In general, XQuery attempts to utilise strength from several query languages and take full advantage of their versatility. XPath and XQuery are both

strongly typed as declarative queries. They use path expressions to traverse XML data irregularly.

3

XML Labelling and Access Control

In this chapter, we shall discuss about some existing approaches that have recently been developed in XML Access Control and XML Labelling Scheme. Techniques such as Node Filtering, Query rewriting for XML access controls, to path indexing, numbering/labelling schemes to facilitate query processing, etc. We shall examine how they work and identify existing problems. Section 3.1 discusses about related works in XML access control. Section 3.2 will discuss about related works in XML labelling scheme.

3.1 Existing Approaches in XML Access Control

A number of different approaches have been proposed to secure XML information in a Web system [Damiani, Fansi, Gabillon and Marrara (2007, 2008), Mohan, Sengupta, Wu and Klinginsmith (2005), Wang and Osborn (2004), Lee, Lee and Liu (2003), Damiani, De Capitani di Vimercati, Paraboschi and Samarati (2002), Kudo and Hada (2000), etc]. For instance, Damiani, De Capitani di Vimercati, Paraboschi and Samarati (2000) proposed an access control model based on structure and content of XML

documents. Authorizations can be specified on a single XML document or on the DTD. Signs of authorization can be either positive (+) for permission or negative (-) for denial.

The limitation of this model is that it does not provide access control modes specific to XML documents, but only provides the read access mode. This work was then extended by Damiani et al. 2002 by enriching the authorization types supported by the model, providing a complete description of the specification and enforcement mechanism. Presented in a five-tuple of the form: {subject, object, action, sign, type}, where:

- ***subject:*** is the subject to whom the authorization is granted.
- ***object:*** is either a URI in Obj or is of the form *URI:PE*, where *PE* is a path expression on the tree of document *URI*.
- ***action*** = read is the action being authorized or forbidden.
- ***sign*** {+, -} is the sign of the authorization, which can be positive (allow access) or negative (forbid access).
- ***type*** {LDH, RDH, L, R, LD, RD, LS, RS} is the type of the authorization (Local DTD Hard, Recursive DTD Hard, Local, Recursive, Local DTD, Recursive DTD, Local Soft, and Recursive Soft, respectively).

Table 3.1 Example of Damiani et al. 2002 Access authorizations.

Subject user/group,IP,domain	**Object (path expression)**	**Action**	**Sign**	**Type**
Public, * ,*	/department/@name	Read	+	L
Public, * ,*	/department/division	Read	+	L
Administrative, *, *.hospital.com	/department//name	Read	+	LDH
Administrative, *, *.hospital.com	/department//address	Read	+	RDH
Administrative, 159.101.80.5, *	/department/medical staff//salary	Read	+	LDH
Administrative, 159.101.80.5, *	/department/patient//cost	Read	+	LDH
Public, *, *	/department/medical_staff/ /salary	Read	-	LDH
Public, *, *	/department/patient//cost	Read	-	LDH
Public, *, *	/department[./@name="medicine"]/medical_staff	Read	+	R
Public, *, *	/department[./@name="medicine"]/medical_staff//address	Read	-	R

Public, *, *	/department[./@name="medicine"]/medical_staff//salary	Read	-	L
PhyC, *, *	/department[./@name="medicine" and ./division="cardiology"]/patient	Read	+	R
Public, *, *	/department[./@name="medicine" and ./division="cardiology"]/patient	Read	-	R
MedicalStaff, *, *	/department[./@name="medicine"]/research	Read	+	R
Public, *, *	/department[./@name="medicine"]/research	Read	-	R
PhyC, 159.*, *	/department/research/project[./@type="private"]	Read	+	R
*, *, *	/department/research/project[./@type="private"]	Read	-	R
NurseC, *, *	/department/patient//illness	Read	+	LS
NurseC, *, *	/department/patient//name	Read	+	L

NurseC, *, *	/department/patient//drug	Read	+	R
NurseC, *, *	/department/patient/room	Read	+	R

Soft authorizations are authorizations that apply to the document unless otherwise stated at the DTD level (intuitively, a department can state that its documents can/cannot be accessed unless the organization states otherwise). Hard authorizations allow an organization to specify authorizations that must be enforced in all instances of a DTD, no exceptions. A similar framework was implemented in the Author-X project [Bertino, Castano and Ferrari (2001)].

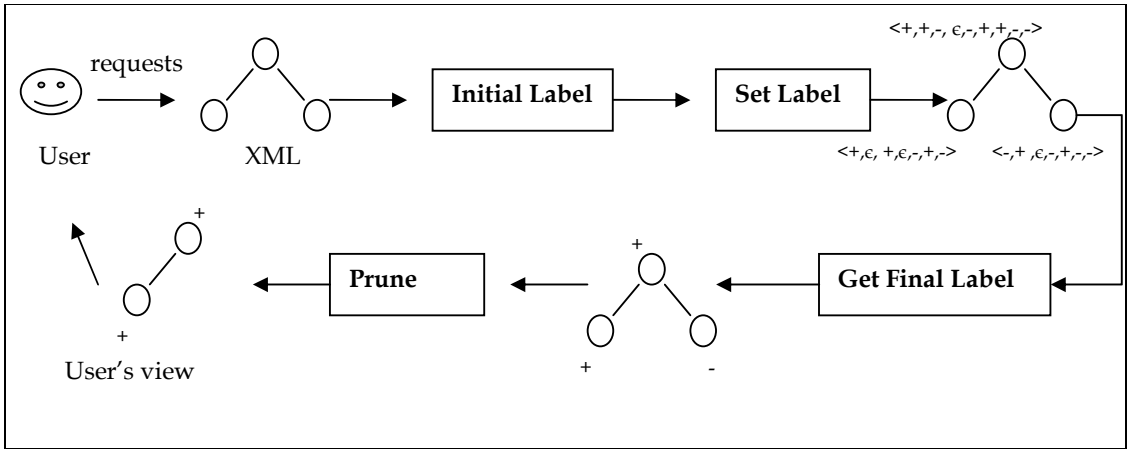


Figure 3.1 – Execution steps of the Compute-view algorithm.- Damiani et al. 2002.

Then work from Damiani et al. 2000 and Damiani et al. 2002 were further developed by De Capitani di Vimercati, Marrara and Samarati (2005) to

define access control rules at the schema level. This model requires query-rewriting technique and view computation for each user/query. This computation process consists of four steps: initial labelling, conflict resolution, propagation, and pruning.

Fan, Chan and Garofalakis (2004) enforced access control polices based on the notion of security views. Access specifications are enforced during the process of deriving the security view, which is based on the user view DTD and a function defined via XPath queries. To avoid the overheads of view materialization, they also employ query-rewriting technique and query optimization, which transform a query over a security view to an equivalent query over the original document, and prune query nodes by using the structural properties of the DTD in conjunction with approximate XPath containment tests.

Mohan, Sengupta, Wu and Klinginsmith (2005) introduce a Security Specification Language for XML (SSX) in the form of a set of primitives. Each primitive takes an XML schema tree as input, and outputs an XML schema tree.

Together with a set of rules to rewrite user queries to enforce security constraints, SSX is used to produce a security view schema for each user.

The annotation algorithm takes a schema and a SSX sequence as input (dealing with one operator at a time), and creates a Security Annotated Schema which can be used for rewriting user queries.

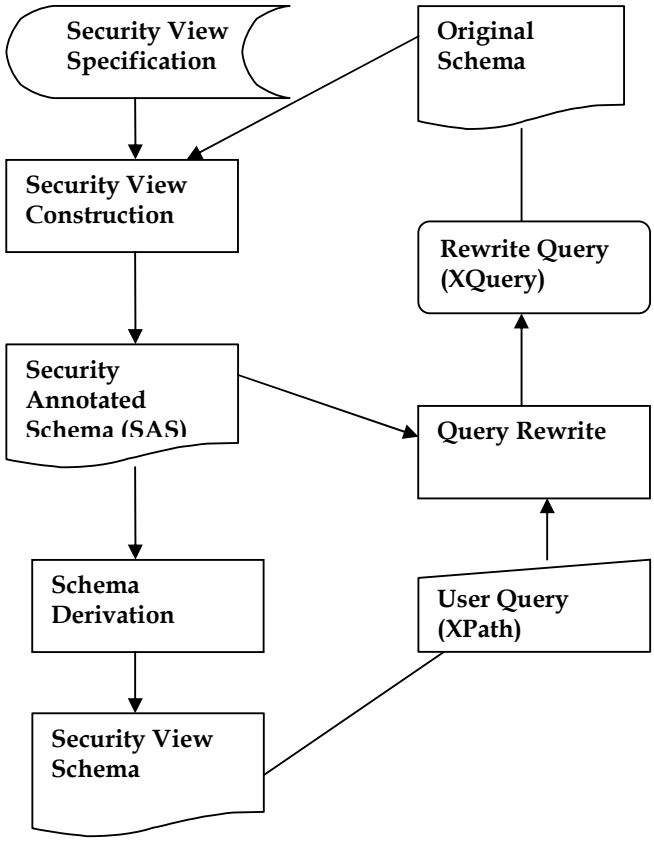


Figure 3.2: The Infrastructure of the Query Rewriting System. –Mohan et al. 2005.

The main drawback of this work is in the SSX, because using primitives such as copy or delete are inefficient for a large-scale access control policy specification. As shown in its experiments, on average, the approach has a similar performance comparing to materialized views.

Lee, Lee and Liu (2003) proposed a model for securing XML documents by enforcing XML data into an underlying relational database. This requires XML data slices into the relational table. As stated by Fundulaki and Marx (2004), since relational data greatly differs from XML data, access control for XML also differs from existing approaches in relational databases, which only support table or column level access controls. One cannot control an access to a row or cell in relational database.

```
<AllDepts>
  <Dept dname='CS'>
      <Manager eid='m10'>
          <Name>Tom</Name>
          <Addr>110 Foster Ave.</Addr>
          <Salary>70K</Salary>
      </Manager>
      <Staff eid='e10'>
          <Name>Jane</Name>
          <Addr>J54 Union St.</Addr>
          <Salary>45K</Salary>
      </Staff>
      <Proj pname='XML''>
          <Year>2003</Year>
          <Budget>100K</Budget>
      </Proj>
      <Proj pname='Stream'>
          <Year>2002</Year>
          <Budget>300K</Budget>
      </Proj>
  </Dept> ...
</AllDepts>
```

Subject	Object	Action	Sign	Type
Public	/AllDepts/*/@dname	Read, write	-	L
Public	//Dept/*/Name	Read	+	L
Manager	//Dept/Staff	Read	+	R
Manager	//Dept/Proj/Budget	Read, write	+	L
Staff, Manager	//Dept/Project/Budget	Read	+	L

Figure 3.3 Lee et al. 2003 AC model

Yokoyama, Ohta, Katayama and Ishikawa (2005) proposed a wrapper program for the XML repository system, namely, SAXOPHONE, which used a prefixed labelling scheme to identify user accounts. (See Figure 3.4). With

only two access policies: *Denial* (access is prohibited) and *Annotation* (descendant and user local annotation) and five Events: Start element, Attribute, Text, End element, and NULL event.

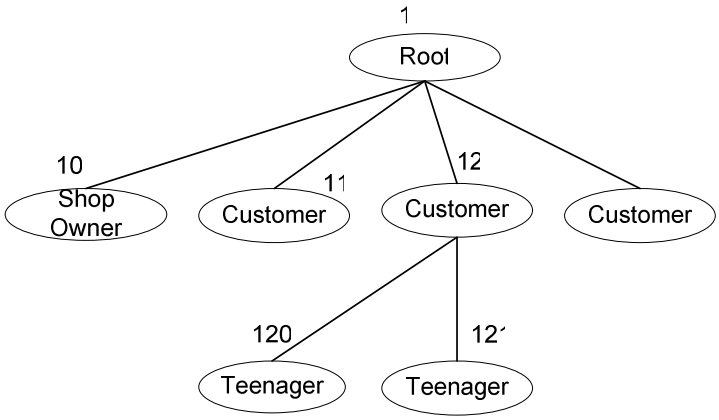


Figure 3.4 Yokoyama et al. 2005 uses prefix-labelling to identify account user

```
<Kiosk>
  <Cigarettes>
     <Name>Peter Jackson</Name>
     <Cost>7.40</Cost>
     <Price>14.95</Price>
   </Cigarettes>
   <Drink>
     <Name>Orange juice</Name>
     <Cost>0.80</Cost>
     <Price>2.20</Price>
   </Drink>
   …
</Kiosk>
```

eID	aID	Type	Property
1	1	startElement	Kiosk
2	1	startElement	Cigarettes
3	1	Text	Peter Jackson
..	..	…	…
6	1	startElement	Cost
..	..	..	..
6	12	NULL	
..	..	..	..
10	12	Text	14.95
..	..	..	..
2	120	NULL	
3	120	NULL	
4	120	NULL	
..	..	..	..
..	..	..	..

Figure 3.5 Yokoyama et al. 2005 Access rules

This system uses relational databases for XML document storage. Since authorization is managed by the relational database management system, XML data has to be translated and stored in relational tables. Similarly, XML queries have to be translated into SQL statements.

Nevertheless, relational data greatly differ from XML data; access control for XML also differs from existing approaches in relational databases for a number of reasons [Fundulaki and Marx, 2004]:

- The hierarchical nature of XML.
- The semi-structured nature of XML documents - In relational tables, the structure is known ahead of time.
- The dependence of a node to its ancestors - Relational tables exist as stand-alone entities, where an XML node stay alive with respect to its ancestors and its children are dependent on the node itself.

Yu, Srivastava, Lakshmanan and Jagadish (2002) proposed an access control model that employs compressed accessibility map (CAM). Data items that have the same accessibility are grouped on a per-user basis in a CAM. A benefit of this is, given a user and a concerned XML data item, this model can quickly determine if the user has the right to access that data item.

However, there are a few drawbacks. It restricts to a single user and an access type at a time; it requires more storage spaces due to the fact that separate compressed accessibility map is needed for each user and access type; and it only supports read action.

For write privilege, Gabillon (2004) proposed an approach based on Xupdate, a non-standard XML update language developed by Laux and Martin (2000). A new *position* privilege is used to acknowledge the existence of a node regardless of its content. Nodes tagged with a position privilege are shown with a restricted label.

Bertino and Ferrari (2002) and Bertino, Castano, Ferrari and Mesiti (2000) supports browsing (read) and authoring (write) privileges. Authorizations can be specified at the DTD - level or at the instance - level. Propagation options are specified along with authorizations and may propagate to all the indirect and/or direct sub-elements or not propagate at all. In Bertino and Ferrari (2002), their model supports information push, which is used for massive distribution of data to subscribers. It is also capable of describing various protection granularity levels, content-based access control and conflict resolution issues. A rule is declared in a tuple of the form: (subject, object, privilege and propagation).

Another model considering write privilege was proposed by Kudo and Hada (2000). This model allows provisional authorization, which indicates an action that a user has to perform before obtaining a given privilege. This model supports several access modes: read, write, create, delete. Three types of propagation policy are employed, no propagation, propagation up and propagation down. Propagation up refers to an access authorization of an element is propagated to all its parent elements. Similarly, propagation down refers to an authorization is propagated to all its sub elements.

```
<?xml version="1.0"?>
 <schema
xmlns="http://www.w3.org//2001/XMLSche
ma">
  <element name="showroom">
   <complexType>
   <sequence>
    <element name="vehicles"
maxOccurs="unbounded" minOccurs="1" >
   <complexType>
   <sequence>
   <element name="available"
maxOccurs="unbounded" >
   <complexType>
   <sequence>
    <element name="model"
type="string"/>
    <element name="color"
type="string"/>
    <element name="price"
type="string"/>
    <element name="accessory"
maxOccurs="unbounded">
   <complexType>
   <sequence>
   <element name="description"
type="string"/>
   <element name="price"
type="string"/>
 </sequence>
 </complexType>
 </element>
 </sequence>
 </complexType>
 </element>
<element name="sold"
maxOccurs="unbounded" >
```

```
<?xml version="1.0"?>
<schema
xmlns="http://www.w3.org//2001/XMLSche
ma">
 <element name="showroom"
access="allow" dirty="true">
  <complexType>
   <sequence>
    <element name="vehicles"
maxOccurs="unbounded" minOccurs="1"
access="allow" dirty="true" >
     <complexType>
      <sequence>
       <element name="available"
maxOccurs="unbounded"
access="allow" dirty="true"
condition="C">
       <complexType>
       <sequence>
      <element name="model"
type="string" access="allow"/>
       <element name="color"
type="string" access="allow"/>
       <element name="price"
type="string" access="allow"/>
      <element name="accessory"
maxOccurs="unbounded"
access="allow" condition="C1">
       <complexType>
        <sequence>
         <element name="description"
type="string" access="allow"/>
         <element name="price"
type="string" access="allow"/>
       </sequence>
      </complexType>
      </element>
```

```
 <complexType>
  <sequence>
   <element name="model"
type="string"/>
   <element name="color"
type="string"/>
   <element name="price"
type="string"/>
   <element name="buyer"
maxOccurs="unbounded">
   <complexType>
   <sequence>
   <element name="name"
type="string"/>
   <element name="address"
type="string"/>
   <element name="city"
type="string"/>
  </sequence>
...
  </sequence>
  <attribute name="city" type="string"
use="required"/>
 </complexType>
 </element>
</schema>
```

(a)

```
     </sequence>
    </complexType>
    </element>
    <element name="sold"
maxOccurs="unbounded" access="deny">
   <complexType>
    <sequence>
     <element name="model"
type="string"/>
     <element name="color"
type="string"/>
     <element name="price"
type="string"/>
     <element name="buyer"
maxOccurs="unbounded">
      <complexType>
       <sequence>
         <element name="name"
type="string"/>
        <element name="address"
type="string"/>
        <element name="city"
type="string"/>
       </sequence>
...
   </sequence>
   <attribute name="city"
type="string" use="required"/>
   </complexType>
  </element>
</schema>
```

(b)

Figure 3.6. Example of a Schema (a) and the corresponding annotated Schema (b). - Damaiani et al. 2007, 2008.

Damiani, Fansi, Gabillon and Marrara (2007, 2008) proposed an access control model for querying and updating XML data. The presenting access control model is an extended and combined version of previous works. The purpose of this combination is to unite two common techniques in XML security, which are node filtering and query rewriting techniques. It then

adds a query-rewriting rule based on Finite State Automata and uses XPath --, a subset of the XPath to rewrite unsafe queries into safe ones.

In general, the rewriting procedure consists of the following three steps:

- The annotated XML schema is transformed according to the policy that applies to each role. According to the user's role, the user is provided with the view of the schema (in short Sv) that s/he is entitled to see. Then, s/he can write his/her query using information available on Sv. Henceforth, unless stated otherwise, the term 'view' will refer to the view of the schema and not to the view of a source document.

- The annotated schema is translated into an automaton which represents the structure of Sv. Each state within Sv contains some security attributes that will further serve us while rewriting the user request.

- The user query is rewritten using the finite state automaton.

For query rewriting technique, to make it simple, let us consider this example. A user, Carolyn makes a query such as:

```
//vehicles/*
```

Her request is to view all vehicle details, such as model, colour, price, description, etc. This is considered an unsafe query, because she does not have permission to view all vehicle details. Information about cars that has been sold and all related information about the car owners are private. Thus, they are not available for public viewing. Hence, her query will be rewritten as:

```
/showroom/vehicles
except (/showroom/vehicles/sold
Union /showroom/vehicles/
available[not(C)] Union /showroom/vehicles/available[C]/
accessory [not(C1)])
```

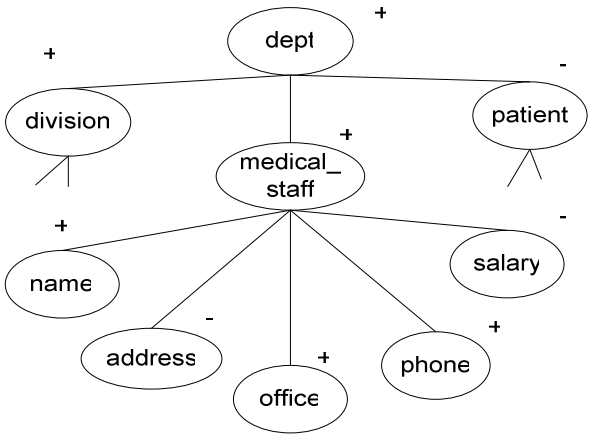


Figure 3.7(a). Example of labelling based on Access Authorization for Public view. ('+' for permission, '-'for denial)

Nevertheless, coming to that safe query involves many stages of evaluations, rules and analysis steps. It needs to be checked against Access Authorization or Annotated Schema (see the one in Figure 3.6(b)). Furthermore, these steps need to be repeated for every query that a user requests.

In general, existing approaches either use node filtering and/or query rewriting. In node filtering, access authorizations are determined by labelling tree nodes with a permission (+), or a denial (-) then pruning trees are based on associated signs. Examples of node labelling and pruning can be found at Figure 3.7 (a) and 3.7(b). The drawbacks of the node filtering technique are that it requires repetitive node labelling, then pruning processes to determine a user view.

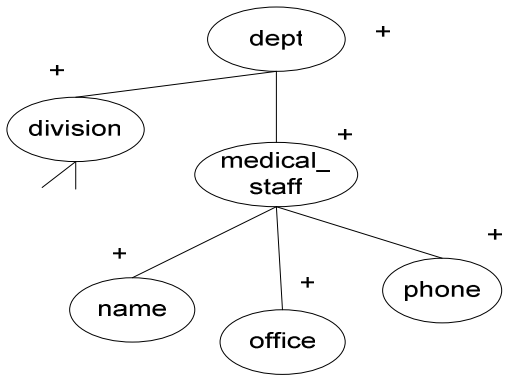


Figure 3.7(b). After pruning process, only node labelled with '+' is shown.

Similarly, in query rewriting technique, every time a user sending a query, it needs to rewrite unsafe queries into safe ones to ensure it returns only accessible elements. This degrades the query performance because costly run-time security checking is still required for unsafe queries. Besides, rules of existing proposals are not well defined for cases in which all users are allowed to access specific fields of their own information but not those of others. Likewise, rules of existing proposals for write privileges are not well defined and do not consider cases of violating DTD and changing XML structure when updating operations occur.

3.2 Existing Approaches in Labelling Scheme

The traditional and most beneficial technique for increasing query performance is the creation of an effective indexing. A well - constructed index will allow a query to bypass the need of scanning the entire table for results. (Kaelin, 2004). To make this possible, a unique label is assigned for each node in the XML trees in such a way that can clearly show relationship between any two given nodes (such as ancestor – descendant relationship or sibling relationship). Once this is done, structural queries can be answered by only using the index. There is no need to access the actual documents. (Lu and Ling, 2004).

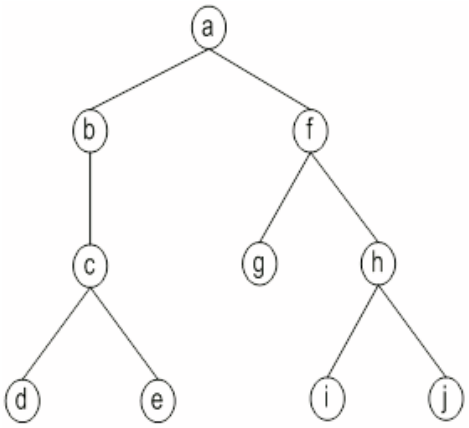
In term of facilitating query processing for XML data, there have been several proposed approaches such as path indexing, tree labelling and numbering schemes. For example, the works by Amato, Debole, Rabitti and Zezula (2003), Grust (2002), Cooper, Sample, Franklin, Hjaltason, and Shadmon (2001), Meuss and Strohmaier (1999) used path indexing for quicker searching XML data.

Moreover, the works by Li, Ling and Hu (2006), Li and Ling (2005), Yu, Luo, Meng and Lu (2005) and Cohen, Kaplan and Milo (2002) used *prefix-based labelling* schemes. On the other hand, the works by Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang (2002) used *Dewey prefix-based numbering* scheme. In addition to that, the works by Li and Moon (2001), Yoshikawa and Amagasa (2001) used *region-based numbering* scheme.

In this section, we shall discuss each of these techniques in more details. Firstly, in section 3.2.1, we will use a different example to illustrate Path Indexing technique. Next, in section 3.2.2, we shall discuss about *Prefix-Based* labelling scheme. Then in section 3.2.3, we shall give an overview of *Region-Based* labelling scheme. Finally, details of *complete k-ary tree-based* numbering scheme shall be discussed in section 3.2.4.

3.2.1 Path indexing

In chapter 2.1, we discussed indexing technique by Amato, Debole, Rabitti and Zezula (2003). Now, we shall examine Grust (2002)'s indexing method.




```
<a>
  <b>
    <c>
      <d> </d>
      <e> </e>
    </c>
  </b>
  <f>
    <g> </g>
    <h>
      <i> </i>
      <j> </j>
    </h>
  </f>
</a>
```

Figure 3.8(a). Data tree based on (b).-XML data

Suppose we have an XML data like the one shown in Figure 3.8(b) and its data tree is represented in Figure 3.8(a). The inner nodes: `a, b, c, f, g, h` represent XML element nodes. The leaf nodes: `d, e, g, i and j` represent either element nodes or attribute nodes.

The idea of Grust (2002)'s work is to find an index structure which guarantees that, for any given context node, one can "determine the set of nodes in the four document partitions specified by the major axes. The further XPath axes (parent, child, descendant-or-self, ancestor-or-self,

following-sibling, and preceding-sibling) determine specific supersets or subsets of these node sets, which are easy to characterize." –[Grust (2002)].

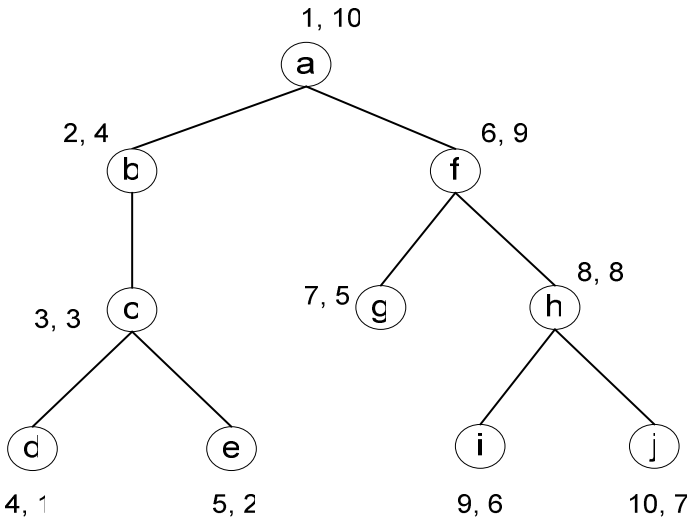


Figure 3.9. Data tree Labelled by <pre-order, post-order> traversal. - Grust 2002.

The concept of this work is to use pre-order traversal and post-order traversal of the data tree (For more information on tree terminology and different orders of tree traversal, please refer to Chapter 2.2) to label nodes. It is developed because XML data orders its nodes corresponding to the order in which a sequential read of the textual XML would encounter the nodes. –[Grust (2002)]. This order is determined by a pre-order traversal of the data tree. In a pre-order traversal, a tree node v is visited and assigned its pre-order rank, `pre(v)` before its children being recursively traversed from

left to right. For the example shown in Figure 3.8(a), the document order is as follows:

```
a < b < c < d < e < f < g < h < i < j
```

Therefore:

```
pre(a) = 0, pre(b) = 1 . . . . , pre(j) = 9
```

In pre-order traversal, 0 – 9 are the codes that are used to label ten elements appeared in Figure 3.8(b).

A post-order traversal is the dual of pre-order traversal, a node v is assigned its post-order rank, `post(v)` after all its children have been traversed from left to right. Again, with the above example, it gets:

```
post(d) = 0, post(e) = 1 . . . . . post(a) = 9
```

Based on the above concept, a theory emerges that:

```
v' is a descendant of v
```

If `pre(v) < pre(v') ^ post(v') < post(v)`

This may be explained as follows: During a sequential read of the XML data, we first see the opening tag `<v>` before `<v'>` and the closing tag `</v'>` before `</v>`. In other words, the element corresponding to v' is contained in the element corresponding to `v`.

In general, based on pre-order: `pre(v)` and post-order traversals: `post(v)` of the data tree, all four major axes relationships between nodes can be determined.

Remark on re-indexing costs:

Indexing based on the traversal order [Grust, (2002)] of the data tree or techniques such as the work of Amato, Debole, Rabitti and Zezula, (2003) still require re-indexing when the XML data is updated. It is because they assign the code for each node in the order where the XML data is entered, (post-order is reversed). Thus, once the XML data is changed, all the codes of related nodes also need to be changed.

An example of this problem is illustrated in Figures 3.10 (a) and 3.10(b). Figure 3.10(a) is an updated XML data. Updated fields are shaded. Figure 3.10(b) shows updating problems where dot lines are.

```
<employee>
     <person>
          <name>Lisa McCathy</name>
          <phone>98765432</phone>
     </person>
     <person>
          <name>Michael Kain</name>
          <address>12 Moon Street</address>
          <phone>98765332</phone>
     </person>
     <person>
          <name>Jennifer Fall</name>
          <phone>93729898</phone>
     </person>
</employee>
```

Figure 3.10(a). Updated XML data

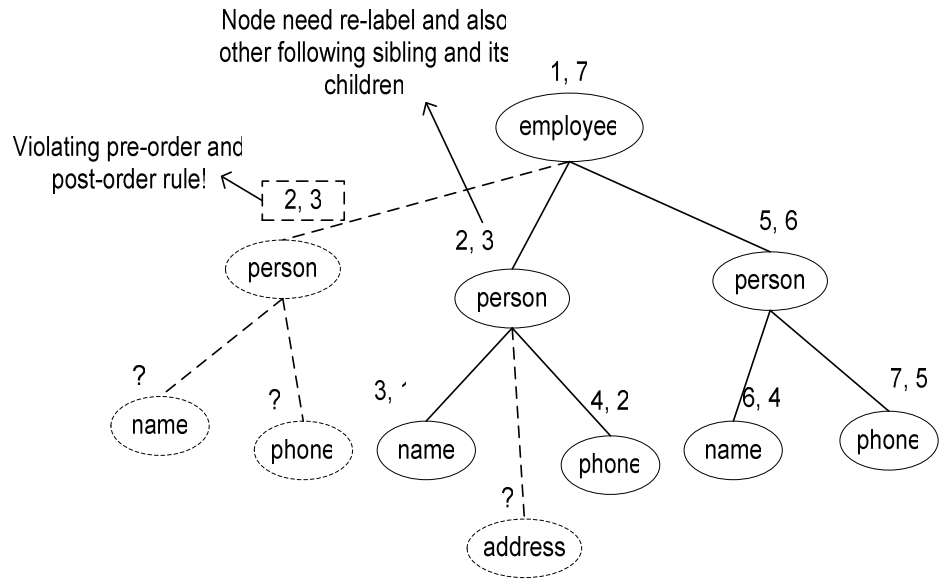


Figure 3.10(b). Example of updating problem. –Labelling Scheme by Grust, 2002.

3.2.2 Prefix-Based Labelling

Prefix-based labelling schemes differ from other schemes as it gives the label of the parent node to the child nodes thus we called it prefixed label. To make it simple, the string before the delimiter, usually is a dot "." or a commas "," is called a prefix-label, the string after the delimiter is called a self - label.

Li, Ling and Hu (2006) and Li and Ling (2005) proposed prefix labelling schemes, which uses binary string to label nodes. In general, it works as follows.

The root node is labelled with an empty string. The self-label of the first (left)

child node is "01". The self-label of the last (right) child node is "011". The purpose of choosing to use "01" and "011" as the first and last sibling self-labels is because they want to insert nodes before the first sibling and after the last sibling. Moreover, in the expectation of dynamically supporting XML update without re-labelling any existing nodes. Nevertheless, there is a conflict in labelling nodes here. Let's look at its labelling rule first.

Once they have assigned the left and right self-labels, they label the middle self - label using these two rules:

Case (a): IF left self-label size ≤ right self-label size.

When adding the middle self – label, they change the last character of the right self - label to "0" and concatenate one more "1".

Case (b): IF left self-label size > right self-label size.

They directly concatenate one more "1" after the left self label.

Thus, they label the middle child node, which is the third child, i.e. [(1 + 5) / 2] = 3. The size of the 1st (left) self-label ("01") is 2 and the size of the 5th (right) self label ("011") is 3 which satisfies Case (a), thus the self label of the third child node is "0101". ("011" → "010" → "0101").

Next, they label the two middle child nodes between "01" and "0101", and between "0101" and "011". For the middle node between "01" (left self-label) and "0101" (right self-label), i.e. the second child node [(1 + 3) / 2] = 2, the left self-label size 2 is smaller than the right self label size 4 which satisfies *Case (a)*, thus the self label of the second child is "01001". ("0101" → "0100" → "01001").

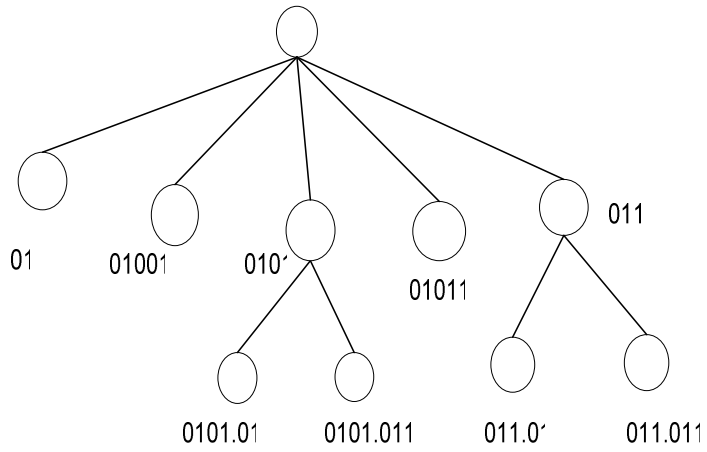


Figure 3.11 ImprovedBinary Scheme

For the middle node between "0101" (left self-label) and "011" (right self-label), i.e. the fourth child [(3 + 5) / 2] = 4, the left self-label size 4 is larger than the right self-label size 3 which satisfies *Case (b)*, thus the self label of the fourth child is "01011". ("0101" + "1" → "01011").

This prefix labelling scheme uses following theorems:

- The sibling self-labels of ImprovedBinary are lexically ordered.
- The labels (prefix-label + delimiter + self-label) of ImprovedBinary are lexically ordered when comparing the labels component by component.

For example, self-labels of the five child nodes of the root in Figure 3.11 are lexically ordered, i.e. "01" < "01001" < "0101" < "01011" < "011" lexically. Similarly, "0101.011" < "011.01" lexically.

Let us try to add some more nodes to the existing nodes, say we want to add a child node to the node "01", as it is the first (left) child node, the code for the new node is "01.01". It looks fine. Then if we add further child node to this node, the code of the new node will be "0101.01". This causes a conflict with the existing node.

O'Neil, O'Neil, Pal, Cseri, Schaller and Westbury (2004) uses Dewey-like numbering scheme (ORDPaths) to label each nodes. The difference is that it starts with the odd numbers for initial load. Such as 1.1, 1.3, 1.5, 1.3.1, 1.3.3 etc. When new nodes are inserted, it uses even "careting-in" between sibling nodes without re-labelling. However, when comparing sizes of labels with our Com-D labelling scheme, result of ours is much smaller and more compact than theirs.

Cohen, Kaplan and Milo (2002) proposed two *prefix-based labelling* schemes to assign a specific code to each child of a node v. The first approach is *one-bit growth*. For instance, the first child's code of the root is "0" which is labelled as L(v).0. The second child's code of root is "10" which is L(v).10. The third child's code is "110" which is L(v).110. Hence, the i^{th} child's code is repeated with "1" for each child's code that ends with "1", together with a "0" attached at the end.

The second approach is *double-bit growth.* Given that u_i's code is L(v).L'(u_i) where L(v) is its direct parent code. It assigns its children as L'(u_1) = 0, L'(u_2) = 10, L'(u_3) = 1100, L'(u_4) = 1101, L'(u_5) = 1110, L'(u_6) = 11110000, etc. In general, it increases the binary code represented by L'(u_i) by 1, that means to assign L'(u_{i+1}). However, if the representation of L'(u_i) + 1 consists of all ones, it doubles its length by adding a sequence of zeros.

Similar technique is used by Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang (2002). It uses *Dewey prefix-based numbering* scheme, which is presented as follows:

> Given n children of a node ν, coded as L(ν), u_1, u_2, u_3, ... u_n . The code of u_i is L(u_i) = L(v).i.

An example of this technique is given in Figure 3.12(a).

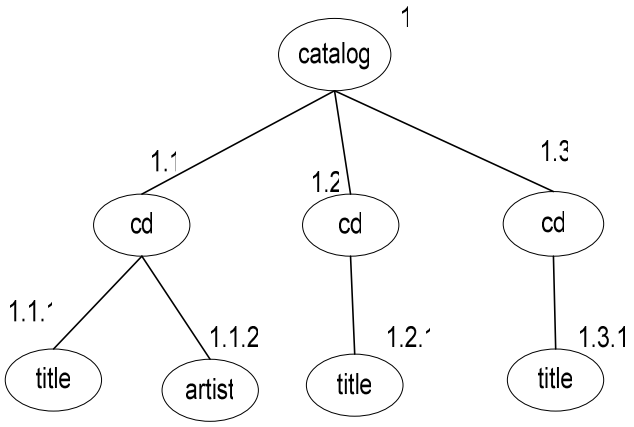


Figure 3.12(a). Dewey Prefix-Based by Tatarinov et al. 2002.

Figure 3.12(b) shown below is the XML document of the mentioned example above. It also contains updated elements, which are shaded. If we use *Dewey prefix-based numbering* scheme by Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita and Zhang (2002) to label this XML data, we will have problems with updating represented by dot lines shown in Figure 3.12(c).

```
<catalog>
     <cd>
          <isbn>43222111</isbn>
          <title>Empire Burlesque</title>
          <artist>Bob Dylan</artist>
```

```
            <price>10.90</price>
      </cd>
      <cd>
            <title>Hide your heart</title>
      </cd>
      <cd>
            <title>Something here</title>
      </cd>
</catalog>
```

Figure 3.12(b). A running example of XML data. Updated fields are shaded.

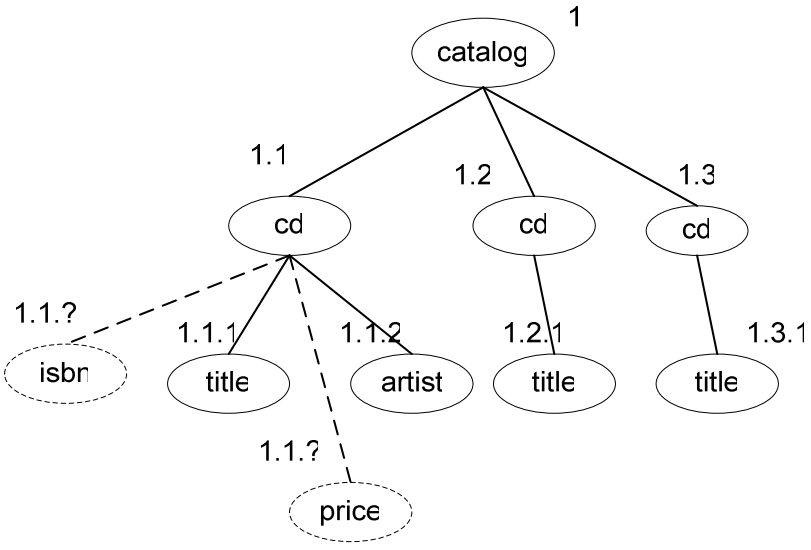


Figure 3.12(c). Dewey Prefix-Based Updating Problem

It came to our knowledge that the work by Yu, Luo, Meng and Lu (2005) dynamically supports updating XML data. It proposes a *prefix-based PBi Tree* scheme, which uses preserved codes between every two-child nodes to reduce the possibility of renumbering all siblings and their descendants when updating is needed. This technique works as follows:

> Suppose a node v has n child nodes, u_1, u_2, u_3 ... u_n. A unique child code is assigned to u_i using m-bits such that $2^{m-1} < n <= 2^m$, indicated $L'(u_i)$. Let the parent node v's code be L(v), then the code of u_i is $L(u_i) = L(v).L'(u_i)$ where "." is concatenation operator.

An example of this technique is given in Figure 3.13(a).

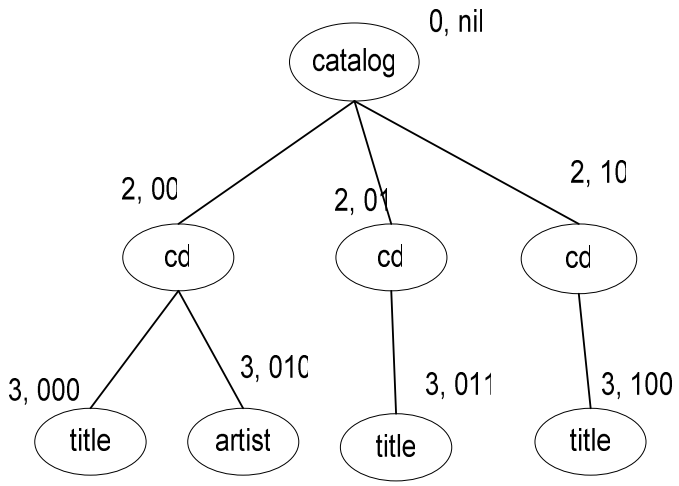


Figure 3.13(a). P-PBiTree by Yu, Luo, Meng and Lu 2005

Although this approach supports updating XML data, this technique is not flexible because codes must be reserved before hand. Moreover, when all reserved codes are used up, renumbering has to be done again. See Figure 3.13(b).

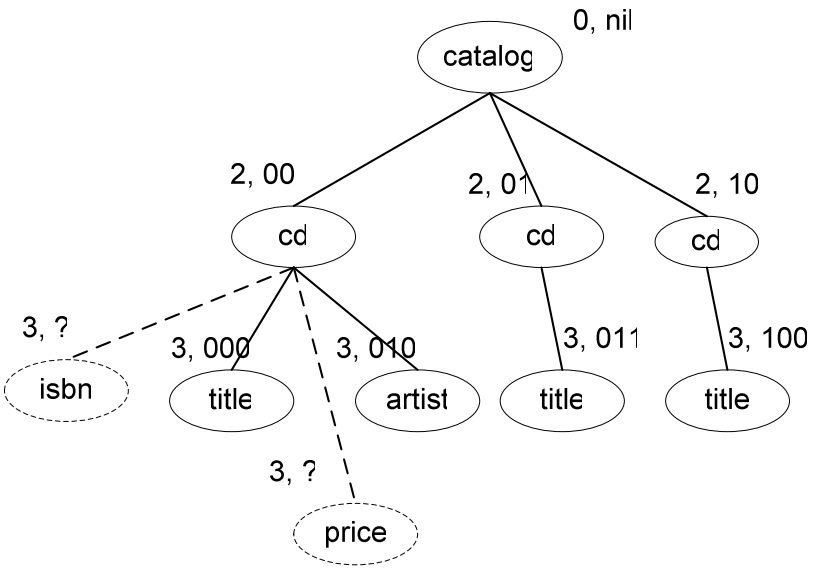


Figure 3.13(b). P-PBiTree Updating Problem

Remark on re-labelling costs:

In *prefix based* numbering scheme, due to the ways it assigns bits as a prefix to a node, sometimes renumbering is still required. For instance, when a new node v is added as the i^{th} position, the code for those nodes originally at v_i and v_{i+1}, v_{i+2} … v_n need to be reallocated by one position. Therefore, all nodes in the sub trees rooted at v_i, v_{i+1}, v_{i+2} …v_n need to be renumbered.

3.2.3 Region-Based Numbering

In Li and Moon (2001)'s work, the authors discovered the benefit from Dietz's numbering scheme in which the ancestor-descendant relationship between any pair of tree nodes can be determined by examining the pre-order and post-order numbers of tree nodes. However, the authors realised the limitation of this inflexibility approach which is "the pre-order and post-order may need to be recomputed for many tree nodes, when a new node is inserted".

To get around this problem, they proposed a new numbering scheme that still uses pre-order and a range of descendants to reserve additional number space for future insertions. The proposed numbering scheme associates each node with a pair of numbers `<order, size>` that can be described as follows:

- Given that a tree node `y` and its parent `x`, then `order(x) < order(y)` and `order(y) + size(y) <= order(x) + size(x)`. In other words, interval `[order(y), order(y) + size(y)]` is contained in interval `[order(x), order(x) + size(x)]`.

- For two sibling nodes `x` and `y`, if `x` is the predecessor of `y` in pre-order traversal, then `order(x) + size(x) < order(y)`.

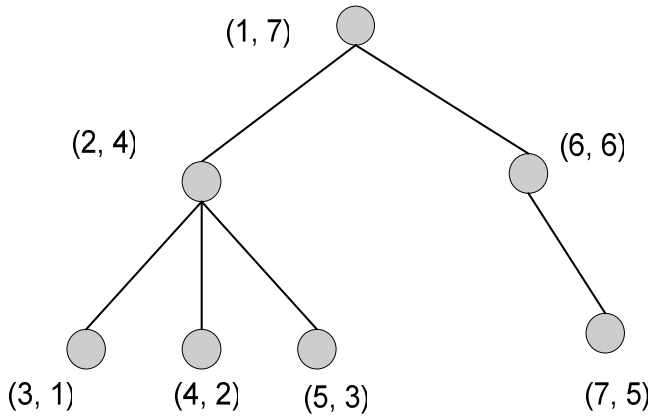


Figure 3.14. Dietz's Numbering Scheme using Pre-order and Post-order

The way in which it reserves space for future inserted elements can be described as follows:

> For a tree node *x*, size(x) $\geq \sum_y$ size(y) for all y's that are a direct child of *x*.

Thus, `size(x)` can be an arbitrary integer larger than the total number of the current descendants of `x`. This would accommodate future insertions.

In Figure 3.15, each node is labelled by a `<order, size>` pair that defines an interval. The interval of a node is appropriately contained in the interval

of its parent node. For instance, a node `(25, 5)` is contained in both `(10, 30)` and `(1, 100)`. Therefore, the node with order 25 is a descendant of nodes with order 10 and 1.

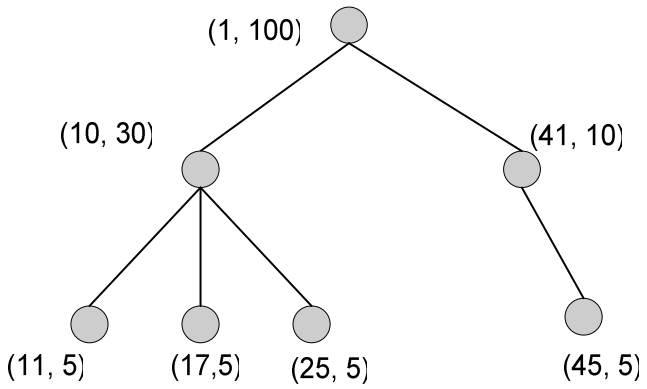


Figure 3.15. Numbering Scheme using <order, size> pair by Li and Moon

A similar technique is used by Amagasa, Yoshikawa and Uemura (2003), in which nodes are labelled with `<start>` and `<end>` of the interval by using floating point values. When a new node is inserted, this technique may not need to re-label existing nodes due to the available values between two floating point numbers. However, there is finite number of values between any two floating point numbers. When all the available values are used up, re-labelling has to be done. Thus, this technique can not quite solve the problem.

Wu, Lee and Hsu (2004) use *Prime* numbers to label XML nodes. The label of

a node is the product of its parent label (a prime number) and its self label (the next available prime number). Ancestor – descendant relationship between two nodes is determined if one can exactly divide the other. Such as node `u` is ancestor of node `v` if and only if `label(v) mod label(u) = 0`.

In order to maintain document order, Prime employs a table of Simultaneous Congruence (SC) values to keep order for each element. Thus, the order of a node is calculated by SC mod self-label. For instance, the document order of the node labelled as '5' is 3, which is calculated by 29243 (a SC value) mod 5.

Prime does not need to re-label any existing nodes when new nodes are inserted in XML tree. However, it needs to re-calculate the SC value to keep the new ordering of the nodes. Furthermore, Prime has to skip a lot of numbers to obtain a prime number, as the result, products of primes can become quite big. Additionally, re-calculating Simultaneous Congruence values every time a new node is inserted is quite time consuming.

Remark on re-numbering costs:

Comparing to Dietz's scheme, Li and Moon (2001) numbering scheme is more flexible and can somehow deal with the issue of dynamic updating XML data. This can be possible because extra spaces are reserved before

hand. Nevertheless, when all the reserved spaces are used up, renumbering affected nodes shall be required. On the other hand, Wu, Lee and Hsu (2004) needs to re-calculate values of affected nodes when order-sensitive nodes are updated.

3.2.4 Complete k-ary Tree-Based Numbering

PBiTree is proposed by Wang, Jiang, Lu and Yu (2003). PBiTree is a numbering scheme called *Perfect Binary Tree*. As its name suggests, the perfect binary tree contains internal nodes (non-leaf nodes) and external nodes (leaf nodes). Each internal node has two children. An external node has no children and all of external nodes are at the same level. Each node is encoded with a number (of the in-order of traversal of the tree) which is called the code of that node. PBiTree is supported by two properties, which can be summarised as follows:

- For a given node n_i of a perfect binary tree, its ancestor n_j at a given height h_j can be directly calculated by $F(n_i, h_j) = 2h_j+1. \lfloor n_i / 2^{hj+1} \rfloor + 2^{hj}$.

- Given the code of a node `n`, its height `height(n)` is the position of the rightmost `'1'` bit in its binary representation. The level of a node, therefore, can be obtained by `H - height(n) - 1`.

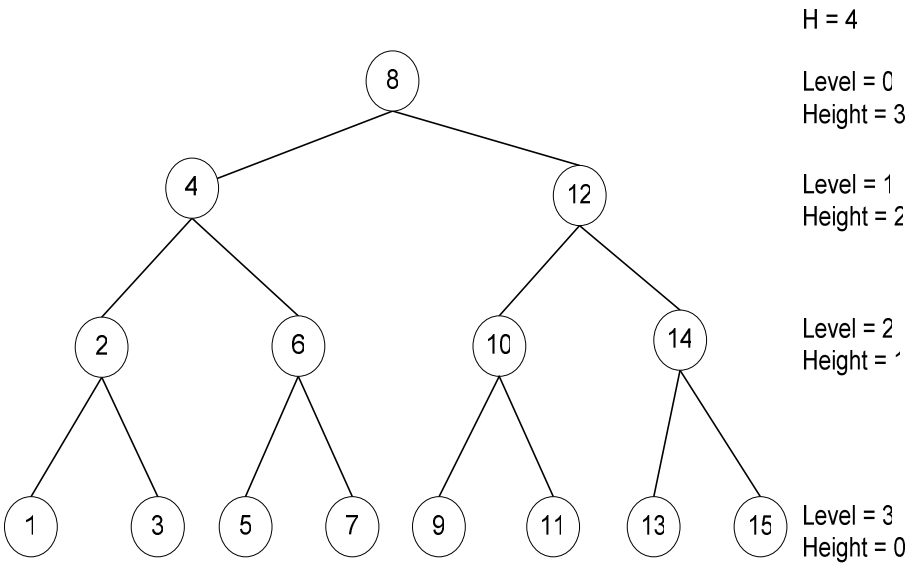


Figure 3.16. An example of PBiTree

Figure 3.16 shows how nodes are coded in PBiTree. The height of the tree, `H`, is `4`. Height of leaf level nodes is equal to `0`. The level of root is `0`. For instance, in Figure 3.15, a node that coded as `10` is at height `1`, its level shall be `4 – 1 – 1 = 2`. This result is obtained by using the formula described in the second properties: `(H – height(n) – 1)`.

Based on the two properties above, the following lemmas are given:

- Given two nodes n_i and n_j in a PBiTree, n_i is an ancestor of n_j if and only if $n_i = F(n_j, height(n_i))$.

- For any node *n* in the PBiTree, let `l` be the level of `n` and *∂* be the zero-based position index of element nodes from left to right, i.e. $\alpha \varepsilon [0, 2^l - 1]$, then $n.Code = Ģ(\alpha, l)$, where $Ģ(\alpha, l),) = (1 + 2\alpha) * 2^{H-l-1}$.

- Given a node `n`, let `H` be the height of the PBiTree and *h* be the height of *n*, i.e., $h = height(n)$, $(n - (2^h - 1), n + (2^h - 1))$ can serve as the region code of *n* in the form of `(Start, End)`.

- Given a node `n`, let `H` be the height of the PBi-Tree and *h* be the height of `n`, i.e., $h = height(n)$, the binary representation of $n >> h$ can serve as the prefix code of *n* (>> is the right shift operator).

However, the tree-structured data shown in Figure 3.16 is usually not modelled as a perfect binary tree. To make use of the properties of perfect binary trees, they embed the original data into a corresponding PBiTree. They call the process of embedding a data tree in a PBiTree as binarization. The PBiTree code for each data tree node can be obtained during the binarization process. The relationship between the PBiTree and the original data tree can be described using an injective function h, such that:

- $h(u_i) = h(u_j)$ if and only if $u_i = u_j$.

- $h(u_i)$ is an ancestor of $h(u_j)$ in the PBiTree if and only if u_i is an ancestor of u_j in the original data tree.

As reviewed and revealed by Yu, Luo, Meng, and Lu (2004), disadvantages of this work are: PBiTree needs to use virtual nodes which do not exist in original data tree and need "to binarize XML trees in an extra pre-processing step". –[Yu et al (2004)].

Lee, Yoo and Yoon (1996) proposed a *k-ary complete tree scheme*, which assigns each element according to the level-order tree traversal to determine the ancestor-descendant relationship between nodes in the data tree. The problem of this approach is identified by Li and Moon (2001):

> "When the arity and height of the complete tree are getting large, the identifier may be a huge number. For example, for a 10-ary complete tree with a height of 10, the total node number will be around 11 billion, which is too large to store in a four-byte word integer. This makes the approach unrealistic for large XML documents." –[Li et al (2001)].

Lu, Ling, Chan and Chen (2006)'s work is not suitable for the update because the labels of all nodes and the finite state transducer must be reconstructed after data insertions. Similar problem exists in Catania, Ooi, Wang and Wang (2005)'s work, when new insertions occur, the global position and the length of each segment must be relabelled.

3.3 Summary

In general, those above mentioned path indexing and labelling scheme can facilitate query processing to a great extent. However, eliminating the problem of re-computing existing labels when XML data need to be updated remains a challenge. Furthermore, these schemes do not consider data confidentiality. Likewise, existing XML access controls have not been developed to integrate with existing indexing methods to speed up the search.

In the next chapter, we will present our new labelling schemes that will support updating for dynamic XML data without the pain of re-labelling, hence facilitating fast update. It does not matter where new nodes should be inserted or how many of new nodes are added. It is guaranteed that none of existing nodes needs to be re-labelled and no re-calculation is required.

Moreover, as proved in our experiments, the total lengths of labels of our Com-D labelling scheme is space efficient, more compact than other labelling schemes. In addition, our new labelling schemes can help ones easily determine the ancestor - descendant relationships and sibling relationships between nodes. The depth of the tree is also represented in our labelling scheme.

4

Dynamic Labelling Schemes

In this chapter, we shall discuss about XML labelling schemes and what makes a labelling a persistent/dynamic labelling scheme. We then introduce our first labelling scheme, LSDX; how it works and its advantages over other static labelling schemes. Finally, our experimental works is presented to show its effectiveness.

4.1 Introduction to Dynamic Labelling

A persistent labelling scheme differs from all other labelling schemes because it supports updating XML data dynamically without the need of re-labelling existing nodes. It is also known as dynamic labelling scheme.

We will present two persistent labelling schemes that will efficiently support updating XML data. It does not matter where new nodes shall be; there is no need to re-label existing nodes. Our labelling schemes also support the representation of the ancestor - descendant relationships and sibling relationships between nodes, such as parent, child, ancestor, descendant,

previous – sibling, following – sibling, previous and following.

In the following subsections, we will introduce the primitive persistent labelling scheme. The improved one will be presented in the next chapter. We call the primitive scheme a loose LSDX labelling scheme and the improved one, a compressed labelling scheme.

4.2 Overview of LSDX Labelling Scheme

Instead of using numbers, letters, or binary string like Li, Ling and Hu (2006), Li and Ling (2005), Silberstein, He, Yi and Yang (2005), Yu, Luo, Meng and Lu (2005), O'Neil, O'Neil, Pal, Cseri, Schaller and Westbury (2004), Yu, Luo, Meng and Lu (2004) or using letters like Wang, Jiang, Lu and Yu (2003) to label each node, we combine numbers and letters in our labelling scheme to label XML trees. This combination makes our new labelling scheme persistent and brings about the following advantages:

a. When XML data is required for updating, there is no need to re-label affected nodes, hence facilitating fast update.
b. It supports the representation of the ancestor-descendant relationships and sibling relationships between nodes.
c. In addition, LSDX also indicates the depth of the tree.

d. Its unique way of labelling will be of help for quick insert, delete, update and capable of easily gaining access to any arbitrary node.

These operations can be implemented using *TreeMap* in Java which algorithms are adapted from *red-black tree* developed by Cormen, Leiserson, Rivest and Stein (2001). In brief, a red-black tree is a binary search tree which inserts and removes nodes wisely, ensuring the tree is reasonably balanced. This shall guarantee that these basic operations (search, insert, delete) take log(n) time in the worst case, where *n* is total number of elements in the tree. Figure 4.1(a) shown below shall give an overview of our LSDX.

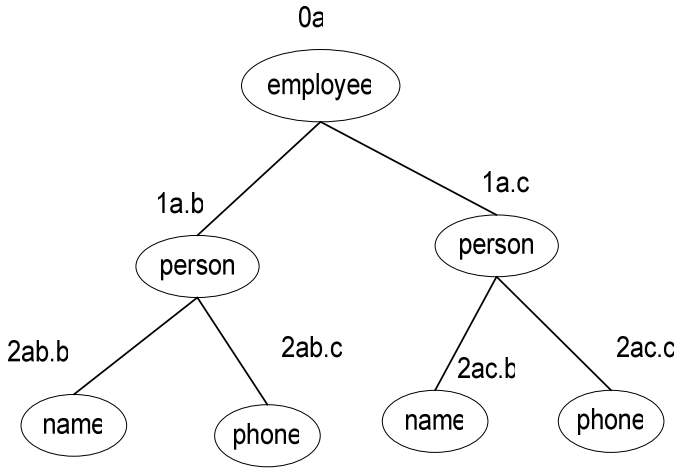


Figure 4.1(a). An example of our LSDX

/employee -> 0a
/employee/person -> 1a.b, 1a.c

/employee/name -> 2ab.b, 2ac.b

/employee/phone -> 2ab.c, 2ac.c

Figure 4.1(b). Path lexicon of the above example

4.3 LSDX and Its Solutions

The loose labelling scheme is demonstrated in Figure 4.1(a) shown above. Let us start with the document element (employee). We first give it an "a". As there is no parent node for the document element, we assign "0" at the front of that "a". "0a" is the unique code for the document element (employee). For the children nodes of "0a", we continue with the next level of the XML tree which is "1" then the code of its parent node which is "a" and a concatenation ".". We then add a letter "b" for the first child, letter "c" for the second child, "d" for the third child and so on. Unique codes for children nodes of "0a" shall be "1a.b" for the first child, "1a.c" for the second child, and "1a.d" for the third child, etc

From level 1 of the XML tree and downwards, we choose to use letter "b" rather than "a" for the first child of the document element because we want to save codes for any InsertBefore operation that might be required in the future. The concatenation "." is employed to help users figure out relationship between ancestor and descendant nodes. For example, just by

looking at node "1a.b", one will realize that it is a descendant of node "0a".

There is a little difference in using the concatenation "." at the third (level 2) and other lower levels of the XML tree. As shown in Figure 4.1 the unique code for the first child of node "1a.b" is "2ab.b" rather than "1a.b.b". We intentionally decide to name it that way because we do not want to confuse users with too many "." In general, it works as follows:

> Given a node v with n child nodes: u_1, u_2, u_3 ... u_n, a unique code for u_1 is a combination of its level + code of its parent node + "." + "b". The unique code for u_2 is its level + code of its parent node + "." + "c". The unique code for u_3 is the code of its parent node + "." + "d". The labelling continues for the rest of child nodes in alphabetical order. When it gets to "z", attaches "b" at the end and continues as mentioned above.

4.4 LSDX Updating Support

For updating and future node insertions, Li and Moon (2001) reserve extra number spaces and Yu, Luo, Meng and Lu (2004) preserve codes. These techniques work well. However, when all reserved spaces and reserved codes are used up and the need for updating continues to arise, affected

nodes will need to be recomputed. To overcome this problem, we use a combination of numbers and letters to create unique codes for XML data.

The concept of using letters is similar with Figure 2.1(b). Using letters from `a` to `z` to label from the root to the child nodes shall keep order and shall give a faster and easy access to a specific node. The difference is the flexible way we use to generate unique codes for every existing node and new nodes, which shall be needed for future updating. The rule for generating unique codes for new nodes is described below.

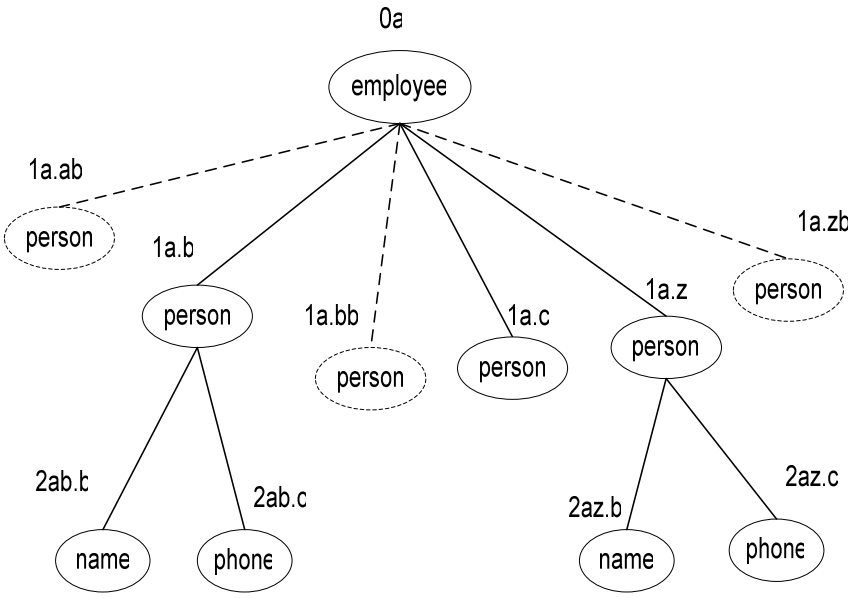


Figure 4.2. An example to illustrate Rule for Generating Labels

Rule for generating labels for new nodes: If there is no node standing before the place that a new node shall be added, get the code of the node standing after the new node and insert "a" after the ".". Otherwise, keep counting from the code standing before it so that the code for the new node will be greater than the code of its preceding sibling and less than the code of its following sibling (if have) in alphabetical order. If the code of its preceding node ends with "z", attach "b" at the end.

Figure 4.2 shows an illustration of our Rule for Generating Labels for New Nodes. Generated labels for new nodes are represented with dot lines.

LSDX supports four operations used for updating XML data. These operations are InsertBefore, InsertAfter, Delete, and Update. More details on these operations are described below.

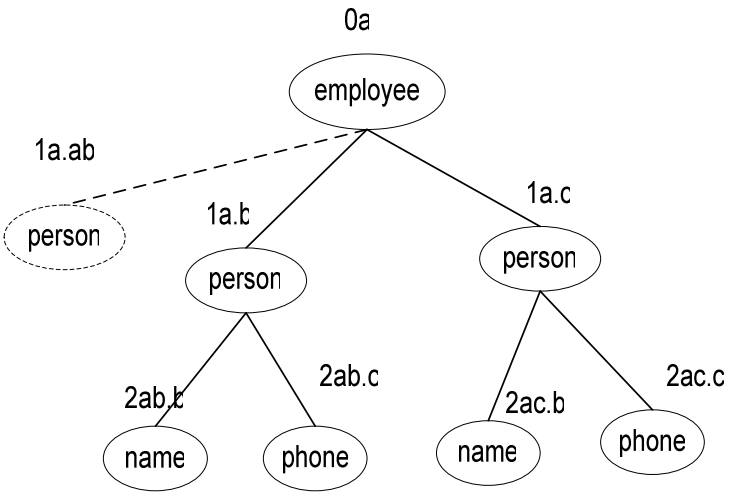


Figure 4.3(a). Inserting Before

- **InsertBefore** (ST, N). Insert a node/sub tree "ST" into an existing XML tree before the node "N". See Figure 4.3(a) – (d).

With InsertBefore operation, one can insert a node or a sub tree before any given node. For instance, Figure 4.3(a) - (d) shows inserted nodes with dot lines. If we want to add a node before the node "`1a.b`", we will just follow the rule for generating labels for new nodes to do this. In this case, there is no node before "`1a.b`" thus we get the code of this node, then add "`a`" after the "`.`". Thus, the code for the new node shall be "`1a.ab`". See Figure 4.3(a).

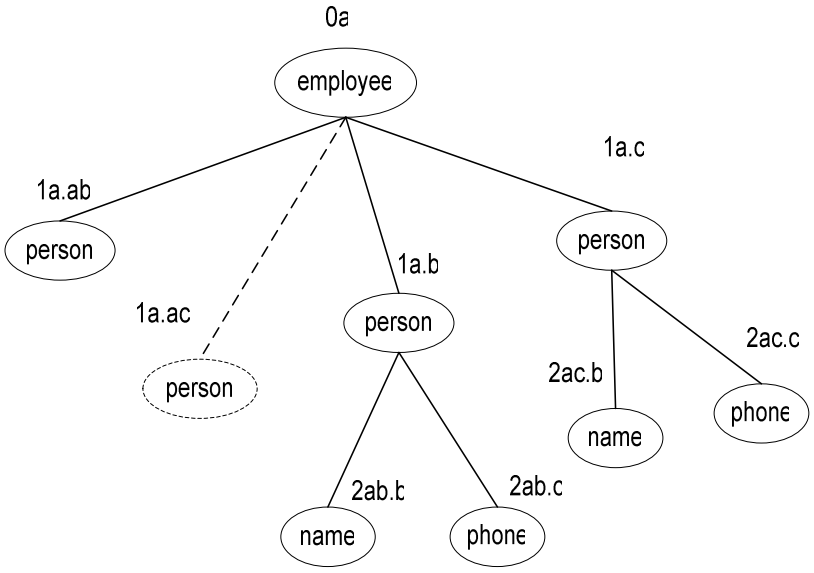


Figure 4.3(b). Another example of Inserting Before

If there is a node standing before the place that we want to insert the new node. See Figure 4.3(b). Apply the rule to generate labels for new nodes. In this case, we keep counting from the preceding code, which is "`1a.ab`" to generate the new code which shall be greater than "`1a.ab`" and less than the code of its following sibling "`1a.b`" in alphabetical order. Thus, the code for the new node will be "`1a.ac`".

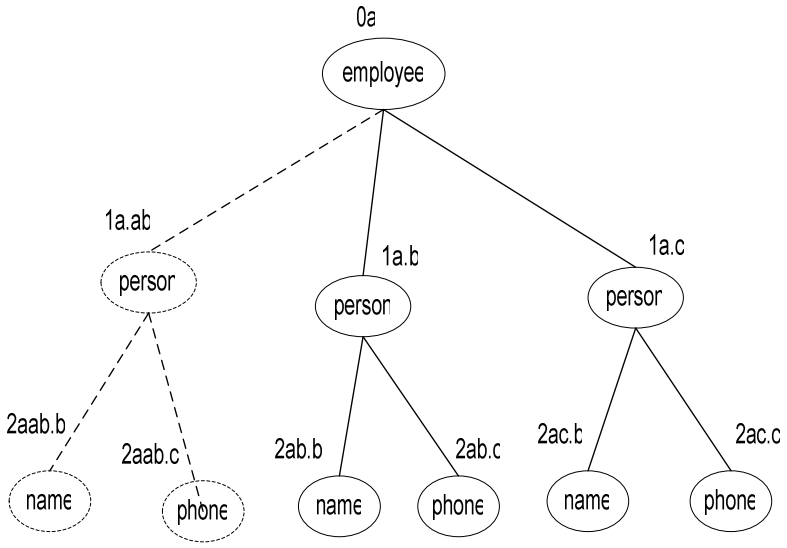


Figure 4.3(c). An example for inserting sub tree

If this is a case of inserting a sub tree, all children of the new node will have "`2aab.`" attached at front then a letter "`b`" for the first child, "`c`" for second child, "`d`" for the third child and so on. See Figure 4.3(c).

The need for future insertions might continue to arise. For example, if we need to insert another new node before the node "`1a.ab`", the unique code

for the new node will be "`1a.aab`". See Figure 4.3(d). Nodes from "`1a.aab`" to "`1a.aaz`" can be used when more insertions are needed. This technique can be utilized over and over again.

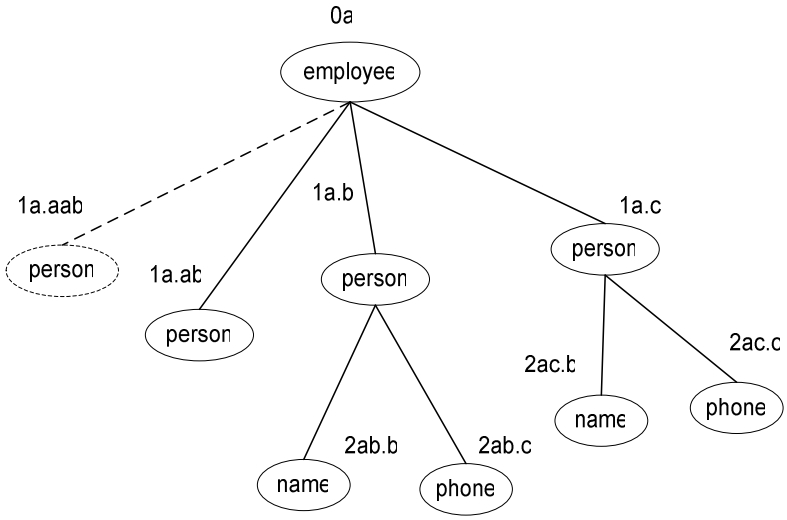


Figure 4.3(d). Another example of Inserting Before

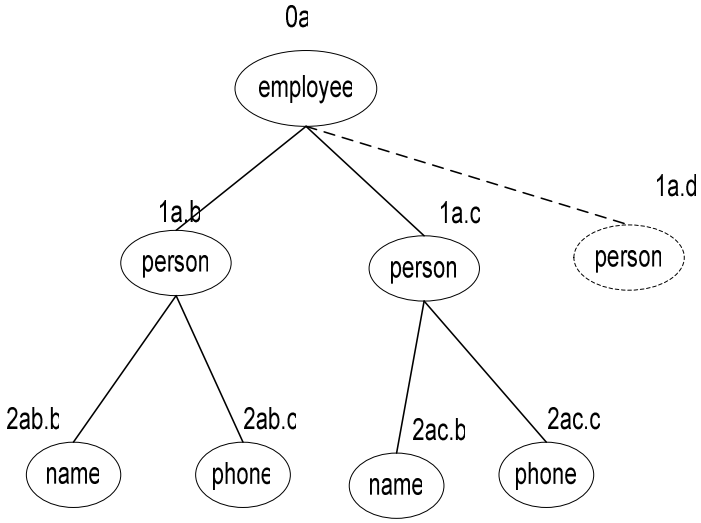


Figure 4.4(a). Inserting After

- **InsertAfter** (ST, N). Insert a node/sub tree "ST" into an existing XML tree after the node "N". See Figure 4.4(a) – (d).

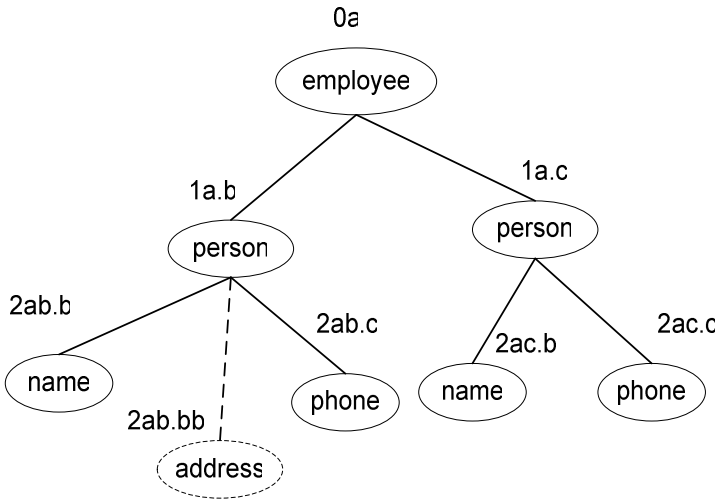


Figure 4.4(b). Example of Inserting After

With the same spirit as InsertBefore operation, InsertAfter operation can be used to insert a node or a sub tree after any given node. Example is given in Figure 4.4(a) with dot lines. If we want to add a new node after the node "`1a.c`", we just follow the *Rule for generating labels for new nodes* to generate the unique code for it. In this case, the preceding node is "`1a.c`". There is no following node. Thus, we need to continue counting from "`1a.c`" to generate a code, which shall be greater than "`1a.c`" in alphabetical order. The code for the new node will be "`1a.d`". If another new node is needed

for insertion after "`1a.d`", its code will be "`1a.e`" and shall continue up to "`1a.z`", "`1a.zb`" to "`1a.zz`", etc.

Figures 4.4(b) and 4.4(c) provide some more examples for InsertAfter operation.

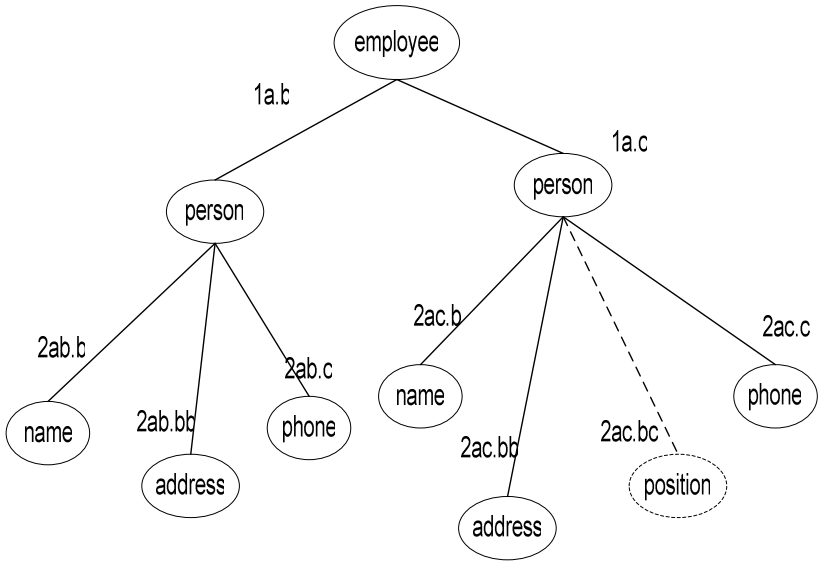


Figure 4.4(c). Another example of Inserting After

Inserting a sub tree after a given node is done in a similar way and demonstrated in Figure 4.4(d) shown below.

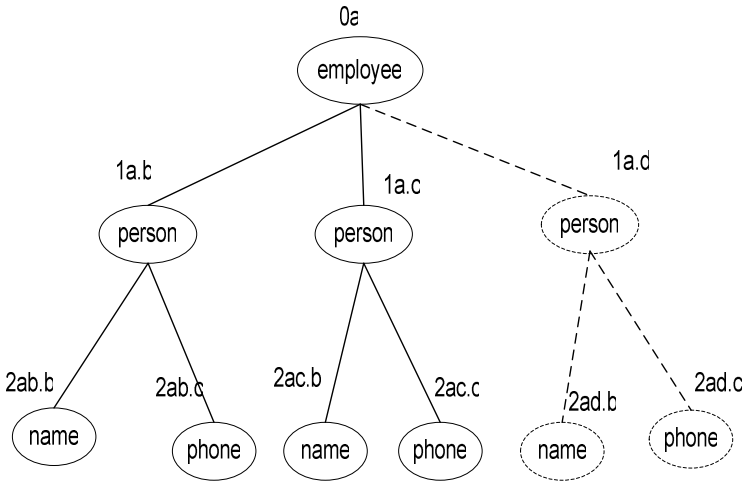


Figure 4.4(d). An example of inserting a sub tree

- **Delete** (ST). Delete a node/sub tree "ST" from the existing XML tree. See Figure 4.5.

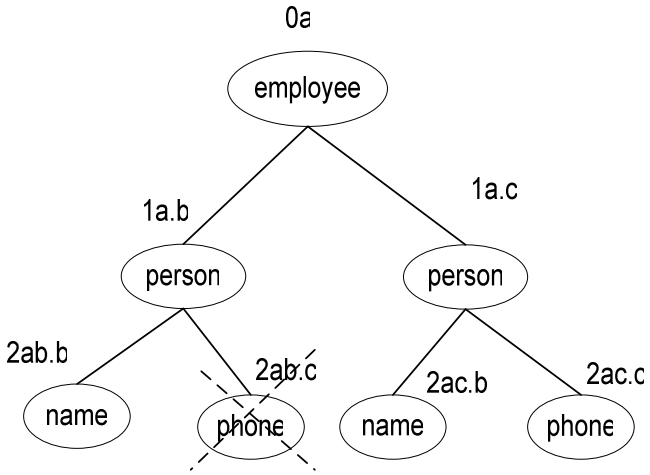


Figure 4.5. Deleting

This operation can be used to delete a node or a sub tree from the existing XML tree. Deleting node/sub tree is quite simple comparing to inserting

node/sub tree because generating code/s is not needed. Code/s of deleting node/sub tree can be used again when a new node/sub tree is inserted in its place.

- **Update** (V, N). Update the content of the node "N" with the value "V". See Figure 4.6.

The content of a node can be updated using this Update operation. An example of this is shown in Figure 4.6 for changing the surname. This operation does not require the need for generating labels.

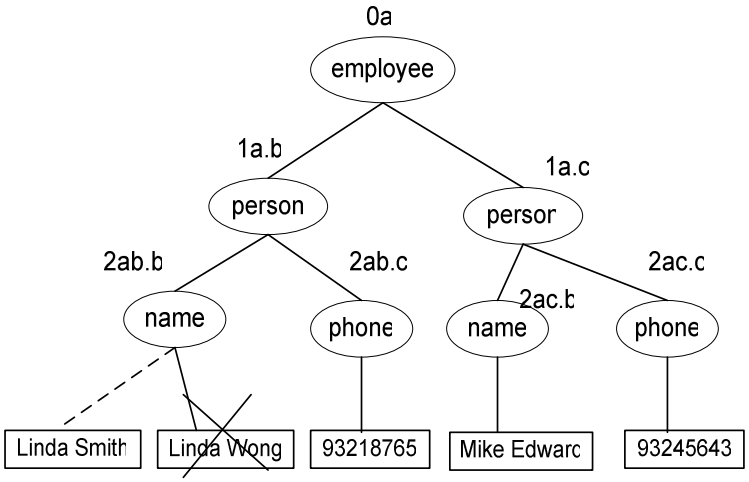


Figure 4.6. Updating

4.5 Ancestor - Descendant Relationship

By adding codes of the parent nodes to the codes of child nodes, it helps us to determine the ancestor – descendant relationships and the sibling relationship between nodes. For instance, in Figure 4.6, by knowing a node called "`1a.b`", we can understand that its parent is "`0a`" and all the nodes beginning with "`1a.`" are its siblings.

Precisely, for all other nodes that start with "`1a.`" and the remaining letters of their codes (after the "`.`") is less than "`b`" in alphabetical order, those nodes are preceding - siblings of node "`1a.b`". If the remaining letters of their codes are greater than "`b`" in alphabetical order, they will be following-siblings of node "`1a.b`". All children nodes of node "`1a.b`" shall have "`2ab.`" attached at front.

4.6 Depth of Tree

Another helpful feature in our new labelling scheme is that it can show the depth (level) of the tree. This can be possible because we attach the level number as the first character when we assign unique codes for each node. For example, if we want to find out the level of node "`1a.b`", we only need to look at the first character of its unique code which is "`1`". Thus, we can say that the level of node "`1a.b`" is 1. In other words, the first character of the unique code of a node always tells the level of that node in the data tree.

Moreover, when this labelling scheme is implemented, using level numbers as explained shall help one quickly gain access to a specific level. Similarly, using our unique codes shall help one easily get to a specific node. Therefore, our labelling scheme will help to reduce the number of nodes that would otherwise need to be accessed to carry out tasks such as retrieving, inserting, deleting or updating XML data. Consequently, these advantages shall make those tasks a lot easier and help to save time.

4.7 LSDX Experiments

We have implemented our proposed first LSDX labelling scheme in Java and used SAX from Sun Microsystems as the XML parser. For the database, we used XMark datasets [Schmidt, Waas, Kersten, Carey, Manolescu and Busse (2002)] to generate XML documents. We used various scaling factors (0.005 – 0.5) to create from 100KB to 57000KB of data. We chose XMark dataset as it generates a standard, balanced XML document and it is typically encountered in real-world scenarios.

An advantage of our proposed labelling scheme is that it supports updating XML data dynamically without the need of re-labelling existing labels. We ran some experimental works and compared our works with the works by

Lu and Ling (2004) and Cohen, Kaplan and Milo (2002) whose labelling techniques are known to support dynamic XML data. Our experiments were performed on the Pentium IV 2.4G with 512MB of RAM running on windows XP with 25G hard disk.

Below is our java program developed for testing our experiment work. In the future, this program will be extended to work as an XQuery tool, which applies our new labelling scheme to facilitate query processing.

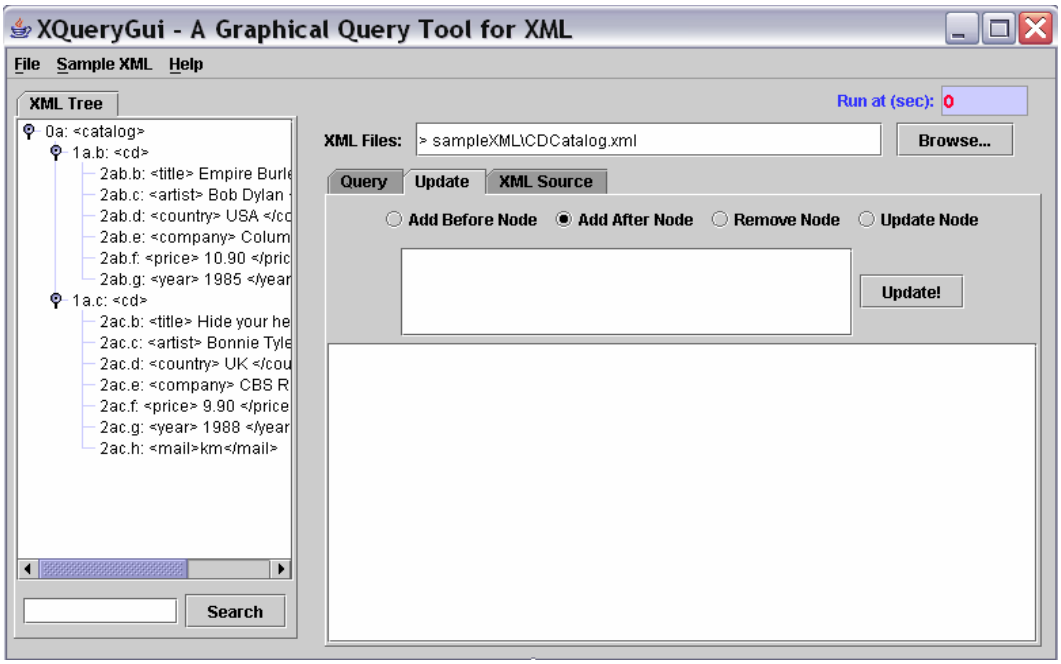


Figure 4.7. A Java application tool used for experiment works

Table 4.1. Total length of labels - A comparison between our LSDX, GRP and SP labelling schemes

XML Doc (MB)	No of Nodes (K)	Total length of labels (MB)		
		LSDX	GRP scheme (Lu, 2004)	SP One bit (Cohen, 2002)
1.2	17	**0.17**	0.64	0.72
2.3	33	**0.41**	1.20	1.46
3.4	50	**0.76**	2.00	2.27
4.7	68	**1.17**	2.72	3.12
5.6	84	**1.63**	3.44	3.86
6.9	100	**2.21**	4.24	4.66
8.0	118	**2.91**	5.04	5.54
9.2	134	**3.60**	5.92	6.32
10.3	151	**4.43**	6.80	7.1
11.4	167	**5.29**	7.60	7.91

4.7.1 Length of Labels

The first experiment for our LSDX labelling scheme was carried out to compare sizes of our proposed labels to those of some other researchers. We studied the experiments of GRP by Lu and Ling (2004) and SP by Cohen, Kaplan and Milo (2002), which support dynamic XML data. In this experiment, we used XMark to generate ten XML documents based on scaling factors (0.01 – 0.1), same as those used by Lu and Ling (2004). We

used SAX to parse different sizes of those ten XML documents. We then generated unique codes for every node in each XML documents and saved them to the file. Eventually, we made comparisons in term of total length of labels. See Figure 4.8. We discovered that our LSDX labelling scheme could be two times smaller compared to GRP and SP schemes. Detailed figures are presented in the Table 4.1.

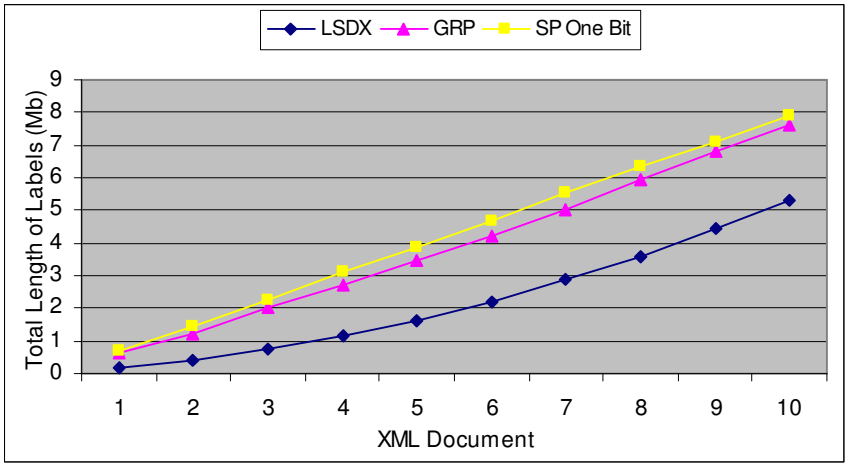


Figure 4.8. Total Length of Labels

4.7.2 Time Used to Generate Labels

Our second experiment was to test how long it takes to generate labels for various datasets that were generated above. We passed each XML document in our Java program, using SAX as a parser, and then concurrently generated labels for it. We noticed that time needed for generating labels varies from 1 second for 1.2MB of data to 28 seconds for 50MB of data.

Approximately, it will take one minute to generate 100MB of data. Some results of this experiment are displayed in Table 4.2.

Table 4.2. Time used to generate labels using LSDX

Scaling Factor	XML Doc (MB)	# of Nodes (K)	Time (Sec)
0.01	1.2	17	**1.09**
0.02	2.3	33	**1.56**
0.03	3.4	50	**2.25**
0.04	4.7	68	**2.79**
0.05	5.6	84	**3.28**
0.06	6.9	100	**3.90**
0.07	8.0	118	**4.54**
0.08	9.2	134	**5.09**
0.09	10.3	151	**5.59**
0.1	11.4	167	**6.18**

4.7.3 Insertion and Deletion Time

We did some experiments on inserting single nodes and inserting sub trees to the mentioned above XML documents. In this experiment, we first generated a unique label for each of the nodes, and added it to the XML tree. We then updated the XML document by saving changes to the file

immediately. While committing all the changes can be done at last to save a great deal of time, we chose to save each change individually merely for the purpose of finding out how much time is needed to commit inserting node/s and sub tree/s operation to the database. Results of time used are shown in Figure 4.9.

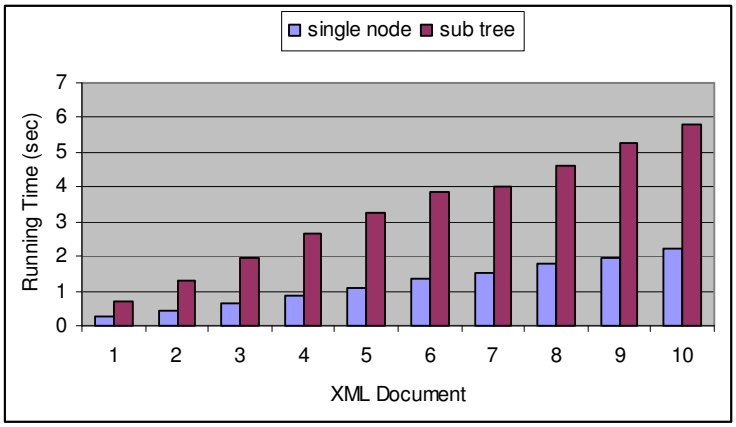


Figure 4.9. Time used to insert nodes.

The performance of inserting single nodes is spectacularly quick. Inserting sub trees took a little bit longer compared to inserting single nodes. A comparison of time used for these two operations is shown in Figure 4.9.

We also did some experiments on deleting single nodes and deleting sub trees from the XML documents above. We first removed the node/s from the XML tree and then updated the XML file so that it is permanently

removed. The running time for deleting a single node and deleting a sub tree are not of much difference. Results are shown in Figure 4.10.

As mentioned previously, we are only interested in finding out actual time taken to permanently remove nodes. For that reason, we did the commit straight away so that deleting nodes will be immediately removed. Our experiments show that deleting nodes/sub trees took less time than inserting nodes/sub trees because generating labels is not required for this operation.

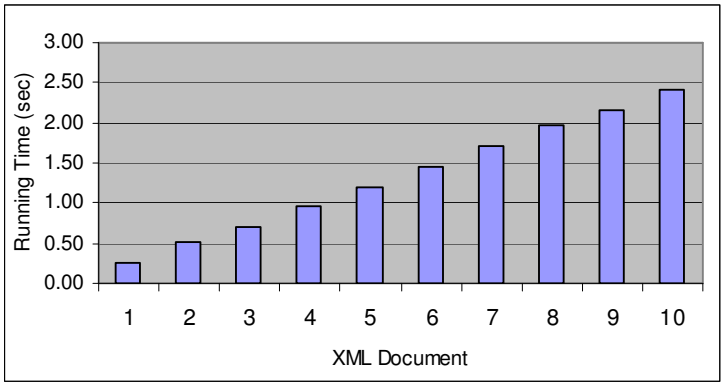


Figure 4.10. Time used to delete nodes

4.8 Summary

Loose LSDX labelling scheme is a dynamic labelling scheme. When XML data is required for updating, there is no need for LSDX to re-label affected nodes, hence facilitating fast update. It can also identify all important axes in

XQuery and XPath, such as ancestor – descendant relationships, sibling relationships between nodes. However, regarding to the length of labels, there's room for improvement for this labelling to become more compact and space efficient. We will discuss this in the next chapter.

5

Com-D Labelling Scheme

In this chapter, we shall introduce another dynamic labelling scheme, the Com-D labelling. We will show how it works and discuss about how it can be better off comparing to existing dynamic labelling schemes. It does not matter where new nodes should be inserted or how many of new nodes are added, with Com-D labelling scheme, there is no need to re-label or re-calculate existing nodes. Then, we will demonstrate how Com-D labelling supports order-sensitive queries. Finally, our experimental work is presented to show its effectiveness.

5.1 Introduction to Com-D

We develop a new technique to label XML tree to make them smaller and more compact. We name our improved version of the loose LSDX as a compressed dynamic labelling scheme or Com-D for short. In brief, Com-D is superior to LSDX in term of its compact labels. Section 5.4 shows the experimental results when comparing these two schemes.

The basic concept for adding new nodes of our Com-D scheme works similarly to the primitive scheme. However, there are slight differences in this technique to make this scheme more compact, space efficient than the primitive version.

Let's consider a large XML document with a big fan-out tree, which may contain hundreds of siblings in a level. There are chances that some part of the node label will be repeated. For example, for a node labelled "`0101010111`", "`01`" is repeated four times, for node "`0110110011`", "`011`" is repeated two times. Node "`bcbcbc`" contains `3(bc)`. Similarly, nodes "`bbbc`", "`abcabcddd`" and "`bacbacbacee`" all have repetitive labels.

Therefore, we can make our loose labelling more compact by rewriting these labels using the following technique. Below are some of the examples that labels can be rewritten.

Example 1. "aaaaabbbcdde" → "5a3bc2de"

Example 2. "aaabbcddddddddee" → "3a2bc8d2e"

Example 3. "kkkkkkkkkkokkkkkkkkkkobbbbb" → "2(10ko)5b"

Example 4. "eaabcbcbcaabcbcbcx" → "e2a3(bc)2a3(bc)x" → "e2(2a3(bc))x"

Example 5. "eeebbbbcbbbbcbbbbcbbbbcbbbbcbbbbcbbbbcbbbbc" → "3e7(4bc)"

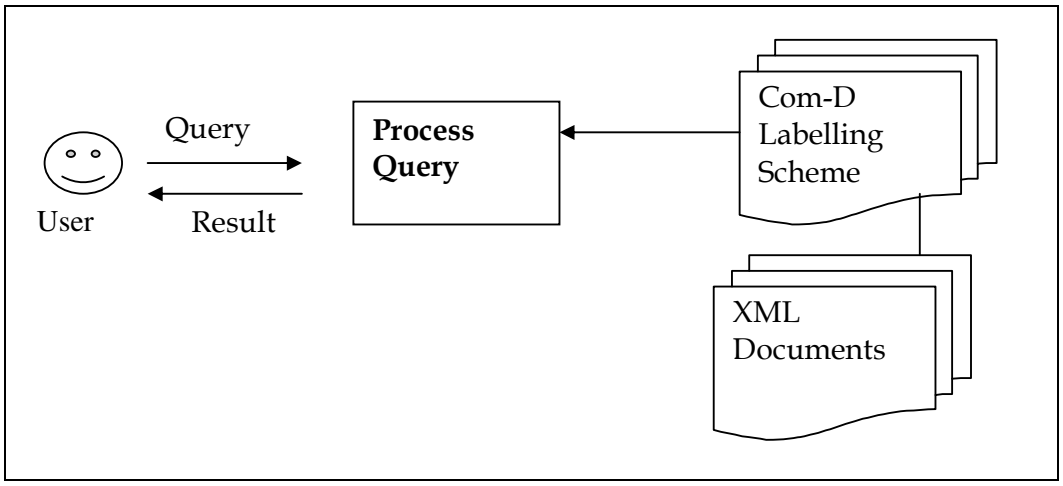


Figure 5.1. A query process using Com-D labelling scheme.

We start with empty string "0" for the root document. For the children of the root document, we start with its tree level + "," + a letter "b" for the first child, letter "c" for the second child, "d" for the third child and so on. Unique codes for children nodes of root document shall be "1,b", "1,c", and shall continue up to "1,z", "1,zb" to "1,zz". Since there are repetitive letters, "zz", we can replace them by number of occurrence of "z" and the letter "z" itself. That means "zz" shall be replace by "2z". It can be continued with "2zb" to "3z". Repetitive letter is not counted if the

concatenation "." stand in middle. For example, "b.bc" is considered as having no repetitive letter.

Similarly, for all child nodes of node "b", the unique code for the first child is its tree level + "," + code of its parent node concatenating with the "." and a letter "b" for the first child, letter "c" for the second child, "d" for the third child and so on. For instance, the first child of node "b" is "1,b.b", the second child of "b" is "1,b.c", and the third child of "b" is "1,b.d" and so on.

It is important to keep in mind that when generating unique code for child nodes, the "." from the code of parent node shall be removed. This is done to minimize the number of "." needed while maintaining its advantage in showing relationship between nodes. In general, generating unique codes works as follows:

Suppose node u is the first child of node v. Rule for generating unique code for node u will consist of the following three steps:

1. Get the code of node v, remove "." if have, and check for repetitive letters. If any letter appears more than once, it shall be

accumulated and replaced by number of its occurrence + the letter itself.

2. Add concatenation dot "."
3. Add "b" if it is the first child of node v. Add "c" for the second child and add "d" for the third child of node v. The labelling continues for the rest of child nodes in alphabetical order. If any repetitive letter occurs again, it shall be replaced by number of its occurrence + the letter itself.

Algorithm 1 below shows how to generate a unique label for each node. There are three separate functions for three different situations. Function generateLabel() is called first to generate code for each node. This function will only process codes for root document and the first child of the root document. If this is not the case, it will determine and call other functions such as firstChildNode()to generate label if parent code is known with no previous sibling present. Alternatively, if previous sibling node is present, function getLabel() will be called to continue calculating label from previous sibling code.

Algorithm 1

Method: generateLabel

```
Parameter: parent code, previous sibling
```

```
Return: Unique code for each node
generateLabel (parent code, previous sibling){
   if (parent code == null)
      return “”;// dummy root
   else if (parent code == "" AND previous sibling ==
null)
      return "b"; //this is first child of root
   else if (previous sibling == null)
      return firstChildNode(parent code);
   else
      return getLabel(previous sibling);
   end if
}
End function
```

Method: firstChildNode

```
Parameter: parent code
Return: Unique code for every first child node
firstChildNode(parent code){
   temp []← parent code.split(".");
   parent Code ← temp[0];
   node label ← parent code + ".b";
   return node label;
}
```

```
End function

Method: getLabel continue calculate code from previous
node
Parameter: previous sibling
Return: Unique code for each node
getLabel(previous sibling){
   char[]temp ← label.toCharArray();
   char last ← temp[temp.length-1];
   if (last == 'z')
      label ← label.concat("b");
   else{
      temp[temp.length-1] ← ++last;
      label ← String.valueOf(temp);
   }
   end if
   return label;
}
End function
```

Algorithm 2 below shows how to compact label of each node for the first round. By just using a few programming lines, a shorter, more compact code can be assigned for each node.

Algorithm 2

```
count: number of repetitive letter

curChar: current character

nextChar: next character

label: uncompressed label code

finalLabel: compressed label code


count ← 0, curChar ←  null, nextChar ← null, finalLabel
← null
while( label != null)
    outer:
    for  (i ← 0; i < label.length(); i++)
          curChar ←  label.charAt(i)
          count++
          if ((label.length()-2) ≥ i)
          if (curChar == line.charAt(i+1))
               continue outer;
          end if
          switch (count)
          case 0:
          case 1:
          finalLabel ← finalLabel + curChar
          break
          default :
```

```
                finalLabel ← finalLabel + count + curChar
        end switch
        count ← 0
    end for
end while
```

5.2 Updating

For updating XML data, our labelling scheme can generate unique code for every new node without re-labelling existing nodes. It does not matter where new nodes shall be added. The rule for generating unique codes for new nodes is described below.

Updating rule: If there is no node standing before the place that a new node shall be added, unique code of new node is the code of its following sibling node minus one value from the last letter. If the last letter of the code of the new node is "a", attach "b" at the end.

Otherwise, keep counting from the code of its preceding sibling so that the code for the new node will be greater than the code of its preceding sibling and less than the code of its following sibling (if have) in alphabetical order. If the code of its preceding node ends with "z", attach "b" at the end.

To get into more details of this compact labelling, let us use some insertions to show how it works. We categorize two insertion situations, one is to insert a new node that has no preceding-sibling and the other, has preceding-sibling. We call these two situations as Insert Before and Insert After operations respectively.

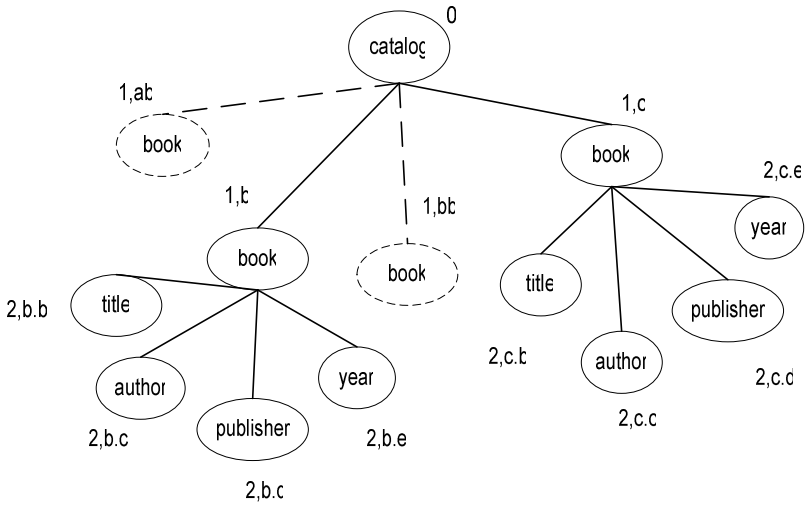


Figure 5.2. Insert a node before a given node.

5.2.1 Insert Before

Insert before is inserting a node/sub tree before any given node which have no preceding - sibling. For instance, Figure 5.1 shows inserted node with dot lines. If we want to add a node before the node "`b`", we will just follow the updating rule. In this case, there is no node standing before the node "`1,b`", thus we get code "`b`" minus one value, which is "`a`". As our rule said, if the

last letter is "a", attach "b" at the end. Thus, the code for the new node shall be "1,ab". See Figure 5.2.

All children of the new node of "1,ab" will have "2,ab." attached at front, then a letter "b" for the first child, "c" for second child, "d" for the third child and so on. See Figure 5.3.

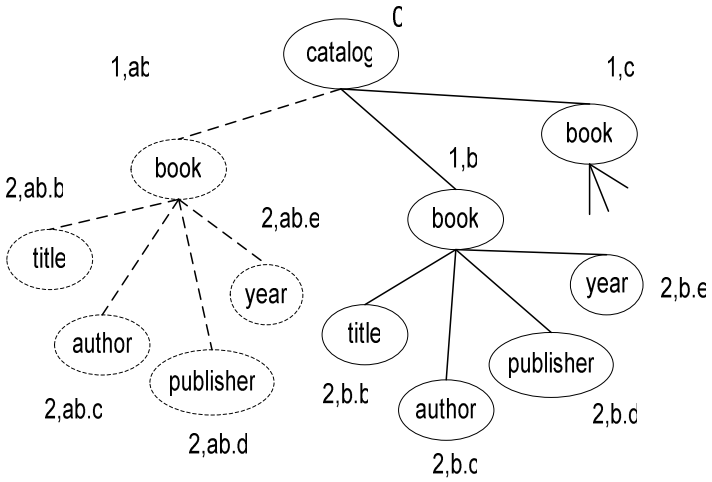


Figure 5.3. Insert a sub tree before a given node.

The need for more insertions might continue to arise in the future. Just simply apply updating rules to generate unique label for each new node. For example, if we need to insert another new node before the node "1,ab", the unique code for the new node will be "1,aab", or "1,2ab" after compression. Nodes from "2ab" to "2az" can be used when more insertions are needed. This technique can be utilized repeatedly.

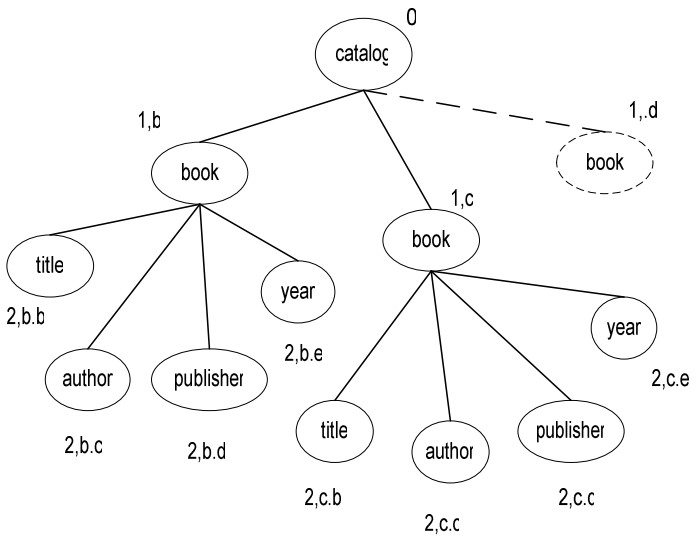


Figure 5.4 Example of inserting a node after a given node.

5.2.2 Insert After

Insert After is inserting a new node after any given node. Insert after differs from insert before because there must be a preceding node before the space that is intended for Insert After operation. However, there might be no following sibling at all. Example is given in Figure 5.4 with dot lines.

If we want to add a new node after the node "1,c", in this case, the preceding node is "c". There is no following node. Thus, we need to continue counting from "c" to generate a code, which shall be greater than "c" in alphabetical order. The code for the new node will be "d". If another new node is needed for insertion after "d", its code will be "e" and shall

continue up to "z", "zb" to "2z", etc.

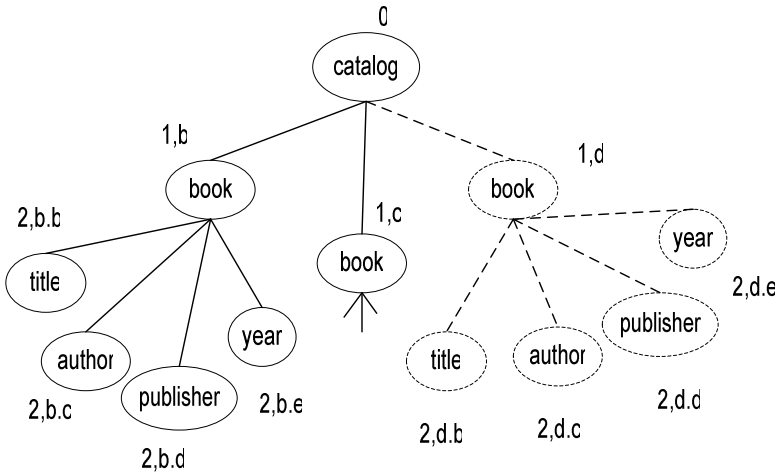


Figure 5.5 Inserting a sub tree after a given node.

Suppose now we want to store two more fields, `price` and `cost` of each book in our database. These new two fields are intrinsic ordered, say after the `Title` and before the `Author` node. Using our rules for adding new nodes, unique codes can be generated for the two new elements without re-labelling enormous nodes already existing in data file and still maintain the order of data. Figure 5.6 illustrates this situation.

5.3 Order-Sensitive Queries

Com-D labelling scheme can be used in all kinds of ordered queries. Ordered queries like Position = n, Preceding, Following, Preceding - sibling and Following-sibling can be answered by evaluating labels of nodes. For

instance, the query "/play/act[3]" can be retrieved by first selecting all act nodes that are descendants of "play", followed by returning the third act.

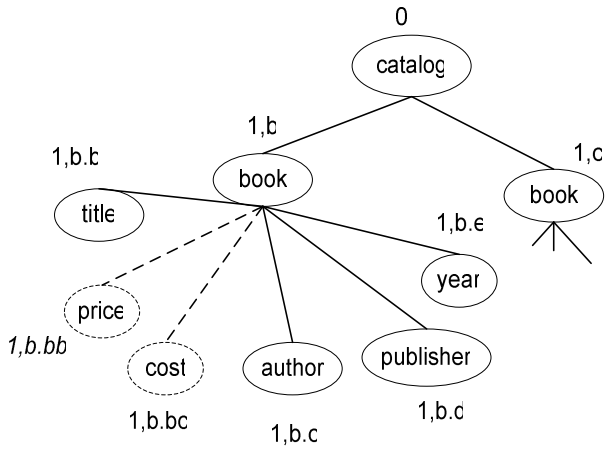


Figure 5.6 Order-sensitive updates – Adding new elements.

Preceding and Following queries like "/play/act[3]/preceding::*" or "following::*" can be answered by comparing the order of all node labels occur before or after with the act[3] node label respectively, descendants of act[3] are ignored.

Preceding-sibling and Following-sibling queries such as "/play/act[3]/following-sibling::act" or "preceding-sibling::act" retrieve all acts that are sibling of act[3] and then output all nodes after act[3] or before act[3] respectively in document order.

5.4 Experiments

We have conducted experimental works to compare our proposed Com-D labelling scheme with some other labelling schemes such as ORDPaths, Dewey, LSDX, GRP, SP Double bit and SP single bit schemes to observe its performance. All of our experiments are performed on the Pentium IV 2.4G with 512MB of RAM running on windows XP with 25G hard disk.

Table 5.1 - Documents used in experiments.

Doc	File Size (MB)	Total Number of Nodes
D1	1.2	17132
D2	2.3	33140
D3	3.4	50266
D4	4.7	67902
D5	5.6	83533
D6	6.9	100337
D7	8.0	118670
D8	9.2	134831
D9	10.3	151289
D10	11.4	167865
D11	22.8	336244
D12	34.0	501498

We use Java and SAX from Sun Microsystems as the XML parser. For the database, Schmidt, Waas, Kersten, Carey, Manolescu and Busse (2002) provides a balanced XML document which usually comes across in real -

world situations. We use XMark datasets to create various sizes of data for experimental purposes. Table 5.1 above shows the size of the XML data files and the total numbers of nodes in each file that were used for the experiment.

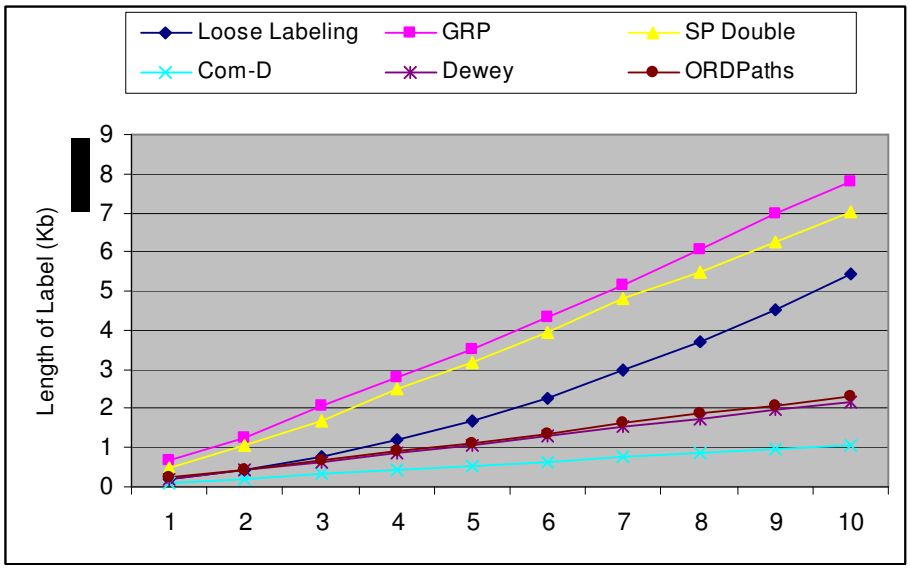


Figure 5.7. Comparison of length of labels among

Com-D Labelling, Loose Labelling, GRP, Dewey, SP double bit and ORDPath.

To start with, we ran an experiment to compare code length between ORDPaths, Dewey, LSDX, GRP, SP Double bit and Com-D schemes using those XML files in Table 5.1. The result of this experiment is shown in Figure 5.7. Among these six schemes, Com-D and ORDPath have shortest code length while GRP and SP double bit have longest code length.

To follow, we carry out three sets of experiments to evaluate the performance of six labelling schemes. The first set compares the storage requirements of four schemes. The second set examines the query performance and the last set investigates the order-sensitive update and study the numbers of nodes, which might require re-labelling.

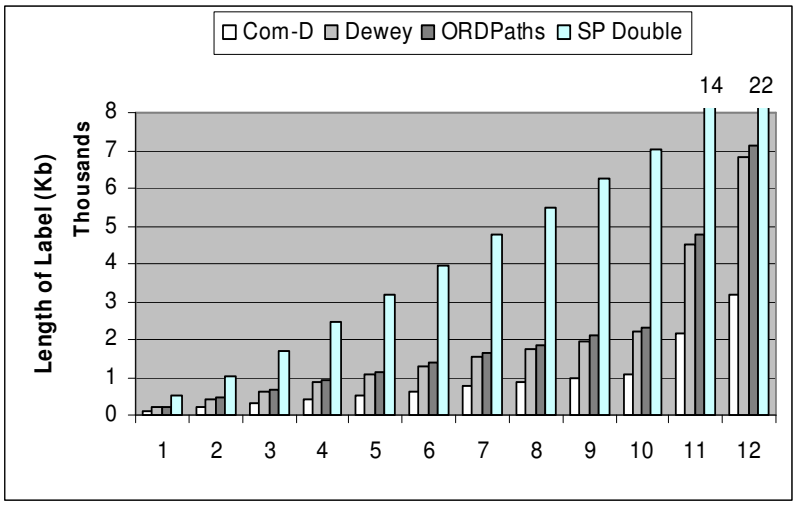


Figure 5.8 Space requirements for each labelling scheme

5.4.1 Storage requirement

Our next experiment is to compare storage space of labels with some other labelling schemes such as ORDPaths, Dewey and SP double bit growth labelling. These experiments indicate that our Com-D labelling scheme is superior to all ORDPaths, SP one bit and double bit growth, GRP, LSDX and Dewey labelling. Results of these experiments are showed in Figure 5.8. To

avoid graph clustering, SP one bit, GRP and LSDX results are not included in this graph.

Table 5.2. Query performance

	Test Queries	Number of nodes returned	Response Time (ms)
Q1	/play/act[5]	185	16
Q2	/play/act/scene[2]preceding::scene	855	15
Q3	/play/act	925	0
Q4	/play/act/scene/speech[4]	3545	250
Q5	/play/act/scene	3740	0
Q6	/play/act/scene/speech[3]preceding-sibling::speech	7280	266
Q7	/play/act/scene/speech/line[2]	85445	1343
Q8	/play/act/scene/speech[2]following-sibling::speech	147275	412
Q9	/play/act/scene/speech	154665	110
Q10	/play/act/scene/speech/line	534410	422

5.4.2 Query Performance

In this experiment, we test the query performance using our new labelling scheme. We use Shakespeare's play dataset from Niagara Project for this

purpose. In order to see its real performance on huge XML data, we increase the Shakespeare's play dataset 5 times. Ten queries used in this experiment are shown in Table 5.2 with number of nodes returned by these queries and theirs response time.

5.4.3 Update Performance

For this experiment, we run several updates to an XML file to measure order-sensitive update performance among several labelling schemes. We use the Dream XML file in Shakespeare's play since all elements in the file are order-sensitive. Dream contains 5 acts; we add a new act before and between existing acts. We then calculate number of nodes that need to be re-labelled for each case. Figure 5.9 shows the number of nodes that require re-labelling.

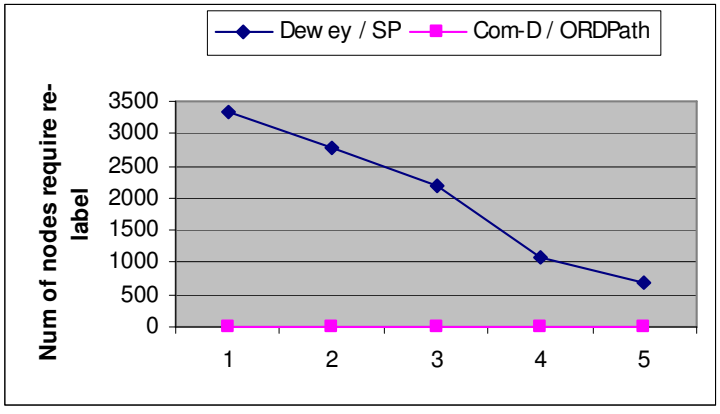


Figure 5.9 Numbers of nodes need re-labelling

In all 5 cases, Com-D and ORDPaths labelling schemes need not to re-label any existing nodes. Dewey and SP schemes need to re-label a huge amount of nodes. Figure 5.9 shows the result of this experiment.

For Prime labelling scheme, there is no node that needs to be re-labelled, however, in order to maintain the order-sensitive of XML nodes, there is a number of nodes required to recalculate SC value. Since the performance of Prime is reported in Wu, Lee and Hsu (2004), we omit it in this experiment.

5.5 Summary

The advantages of our Com-D labelling scheme over the existing labelling schemes is that it is a dynamic labelling scheme for XML data and it is compact. It does not matter where new nodes will be inserted or how many of new nodes are added, as it guarantees that none of existing nodes needs to be re-labelled and no re-calculation is required. These will facilitate fast update as well as enhancing query processing. In addition to those advantages, our Com-D labelling scheme also supports the representation of the ancestor - descendant relationships and sibling relationships between nodes.

6

XML Access Control

In this chapter, we shall introduce SecureX, our XML Access Control model. SecureX supports both read and write privileges. We will show how SecureX can be integrated with a dynamic labelling scheme to speed up searching and querying processes. We then will analyse and compare processing steps between our access control model with traditional node filtering techniques. Finally, experimental results are shown to prove the effectiveness of our approach.

6.1 Introduction to SecureX

Many existing access controls use node filtering or querying rewriting techniques. These techniques require rather time-consuming processes such as parsing, labelling, pruning and/or rewriting queries into safe ones each time a user requests a query or takes an action. In the next section, we present our access control model for read privilege. Write privilege will be presented in the following section. For clarity, we use XML Access Authorization (XAA) file to declare access level details for each element in

XML data. XML Schema Authorization (XSA) file states access control if rules are declared in schema level for a set of documents. XML Group Authorization (XGA) file is used to define the access authorization for individual user or a group of users. We make an assumption that, the person who creates XML files will also have the right to create access rules for those files.

```
<XAA doc="Employee.xml">
....
<rule object="//name" access="+" type="R"/>
<rule object="//address" access="-" />
<rule object="//DOB" access="-" />
<rule object="//h_phone" access="-" />
<rule object="//office" access="+" />
<rule object="//extension" access="+" />
<rule object="//email" access="+" />
<rule object="//position" access="+" />
<rule object="//salary" access="#" />
</XAA>
```

(a)

Figure 6.1 (a)XAA

Considering a staff database of a University that stores all staff information, such as name, office, extension, home address etc. Suppose this university has three levels of access policy defined as public, private and protected levels, denoted as '+', '−' and '\#' respectively. Therefore, its access rules are defined in XAA in Figure 6.1(a).

```
<XAG doc="Employee.xml">
....
<group entity="student" access-type="+" />
<group entity="staff" access-type="+" />
<group entity="admin" access-type="-" />
<group entity="CEO" access-type="#" />
....
</XAG>
```

(b)

```
.....
<group entity="staff" access-type="$,+" />
<group entity="admin" access-type="$, -" />
.....
```

(c)

Figure 6.1(b)XGA. - (c)Updated XGA.

Our access control model requires the specification of subject, object, access type, action, and propagation type to determine whether an authorization access is allowed. Subject is group/individual identifier; object is URI or XPath expression; access type states access level of a user; action is action on which authorization is defined, propagation type defines how the permissions is propagated, such as local or recursive. Default is local. Group can be a single user. Group and access type are determined after a user logs on. This is described in XGA file in Figure 6.1(b).

***Case 1*:** Imagine that John belongs to a `staff` group. He has a request to view all staff details. Based on rules described in Figure 6.1(b), his access level denoted as `"+"`, thus he can access to all public fields such as `name`, `office`, `email`, etc. of all staff including his own information. However, due to the limitation of his access level, he cannot view private information and salary of other staff. Surprisingly, he cannot even view his private information. This does not seem right because one should be able to view his/her own information for clarity checking or updating purposes. The concept is as simple as you cannot read email of other people but you can read yours.

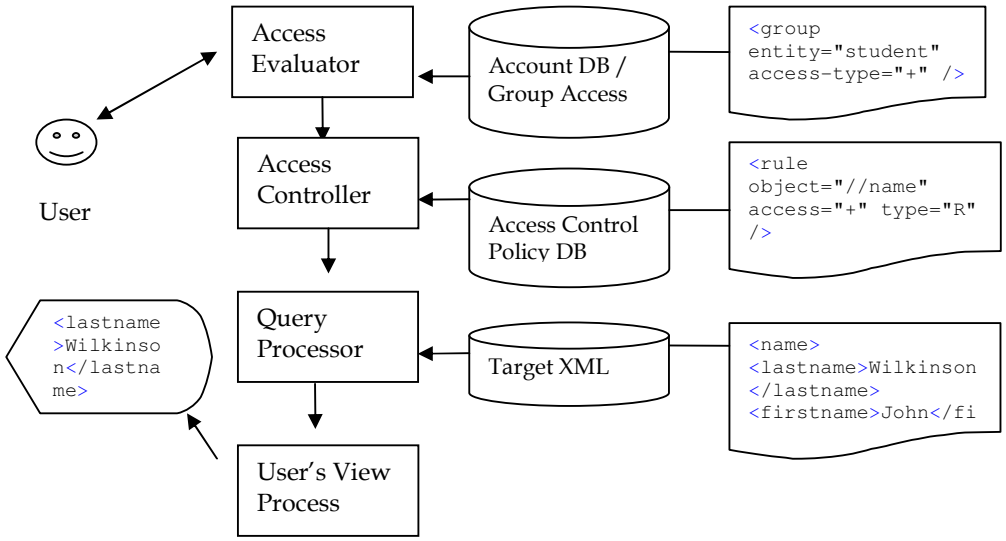


Figure 6.2 XML Access Control Model

To meet this requirement, we introduce a new policy for self access, denoted as `'$'`. For staff group, their access type now is `'$, +'`. See Figure 6.1(c). This is a twofold access rule, which stands for self and public access levels. Conceptually, this rule can be read as this person can access his or her own information and public information of others.

In fact, the above access levels are formed hierarchically. Who has a right to access at a higher level type could obviously access all information of lower levels. For example, user `A` holds an access right of a protected level, he/she can access all public and private information levels. The rule is `public < private < protected`.

Table 6.1. Possible accessibility symbols.

Symbol	Accessibility
+	Public/External viewers
*	All staffs
$	Self information
-	Administrators
&	Managerial staffs
@	Other
%	Other
#	CEO/Top secret

It is worth to mention that, in this case, we only introduce three access levels. These access levels can easily be extended to meet different needs of each organization, see Table 6.1 for a list of possible accessibility levels. For instance, organization `A` sets five security levels for their database, such as level 1 for public view, level 2 for staffs, level 3 for administrators, level 4 for managerial staffs and finally level 5 is top secret which can only be accessed by CEO. Without having any difficulty, this can be implemented using our approach, denoted as `"+, *, –, &, #"` respectively. Moreover, our concepts are flexible to define mix rules for any special case. Examples of these cases are discussed in next sessions.

Case 2: Get back to our University example, considering Lisa is the head of Computer Science Department. Her access level is protected. Obviously, with this access level she can access all information of all staff in all departments in the University. In practice, this is insecure, since she has nothing to do with other staff's private information in other departments. To make it right, she can access to all information of staff of Computer Science Department but only public information of staff of other departments. Therefore, to reflex this circumstance, her access level can be declared as follows.

```
<group entity= "Lisa" access-type= "+">
     <rule object="//dept[@name='CompSci']" access-
     type="#"  type="R" />
</group>
```

6.2 Integrated with XML Labelling

As discussed in Chapter 1, existing access controls mostly focus on node filtering and query rewriting techniques. For query rewriting approach, access control rules are not defined for the XML data. Unsafe queries of a user are translated into safe ones and are evaluated against the original XML dataset. In general, processes of node filtering require parsing XML data to get a DOM tree. Then, based on access right for this user, it labels each node

in the DOM tree with a permission, normally denoted as '+' if accepted or '–' if denied. Finally, pruning is done from the DOM tree. Nodes labelled as denial (–) are removed and nodes labelled with permission (+) are shown to users.

Clearly, these techniques require repetitive parsing and labelling processes for every user. Moreover, if a user requests other actions, these steps will need to be repeated again for the same user. These are time consuming and take a lot of resource for XML parsing, labelling and tree searching. In addition, they do not take advantages of indexing/labelling schemes, which have been developed by many researchers to facilitate query processing.

In this section, we will integrate our access control model with our dynamic labelling scheme, the Com-D. The framework of this integration between these two schemes is demonstrated in Figure 6.3 and Figure 6.4. Figure 6.3 shows initial steps that are needed to carry out before user sending a query. In fact, this process only needs to be carried out once. It generates access authorization and label code for each node in XML data and stores them in an index file. Figure 6.4 shows processes when a user sends a query. Comparing to other approaches, our model makes less processing/preparing steps and interact with less resources to return a user's view.

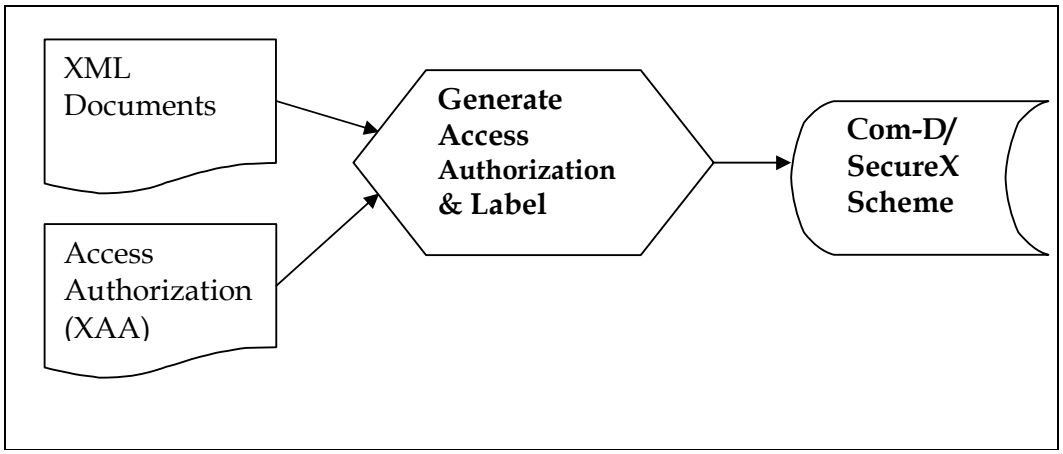


Figure 6.3. Generating access & label codes for each XML Data.

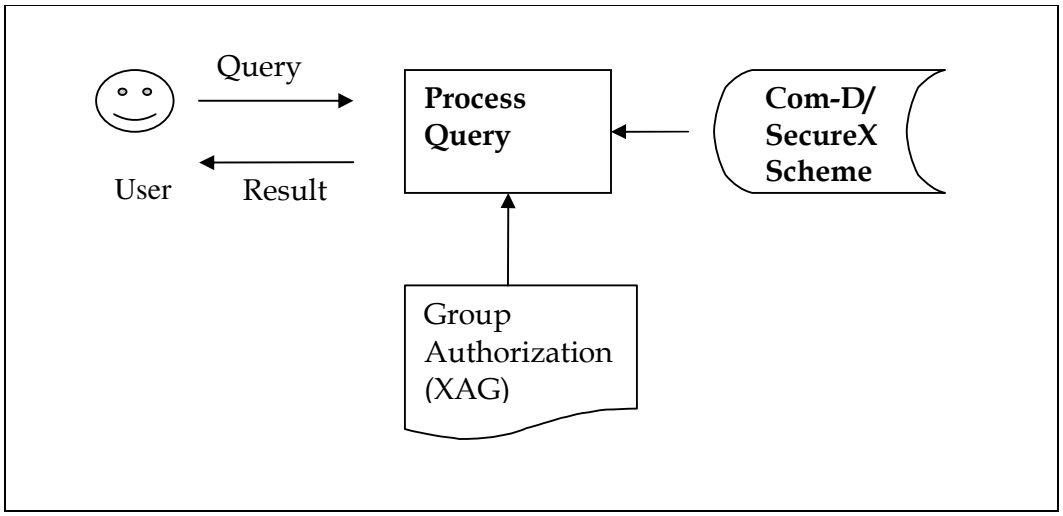


Figure 6.4. Query processing with Integrated SecureX model.

In fact, our SecureX can be integrated well with any indexing/numbering scheme or dynamic labelling scheme that support XML update such as Duong and Zhang (2005), O'Neil, O'Neil, Pal, Cseri, Schaller and Westbury (2004) or Wu, Lee and Hsu (2004), etc. We choose Com-D labelling scheme

because it is superior to all of the above dynamic schemes either in term of shortening label length or no need to relabelling or recalculate values for existing nodes when updating or inserting order sensitive nodes. Moreover, it also supports the representation of the ancestor descendant relationships and sibling relationships between nodes. In general, it works as follows.

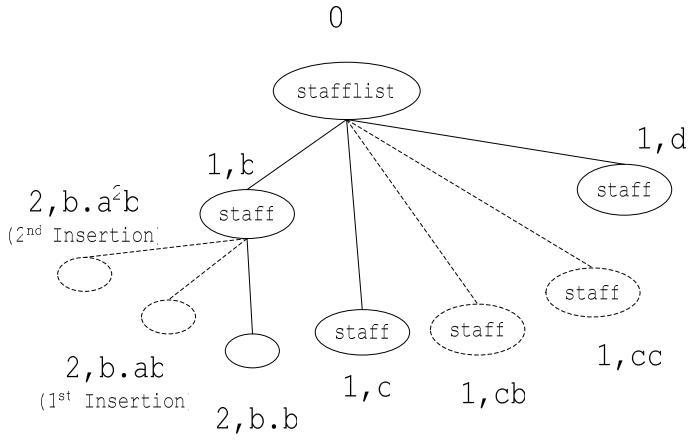


Figure 6.5. Dynamic labelling scheme, dot lines present newly inserted nodes.

Start with tree level "0" for the root document. Children of the root document will be its tree level + letter "b" for the first child, letter "c" for the second child, "d" for the third child and so on. Thus, unique codes for children nodes of root document shall be "1,b", "1,c", and shall continue up to "1,z", "1,zb" to "1,zz". Since there are repetitive letters, "zz", we can replace them by number of occurrence of "z" and the letter "z" itself. That means "zz" shall be replace by "z^2". It can be continued with "z^2b" to "z^3". Repetitive letter is not counted if the

concatenation "." stand in middle. For example, "1,b.bc" is considered as having no repetitive letter.

Similarly for all child nodes of node "1,b", the unique code for the first child is the code of parent node concatenating with the "." and a letter "b" for the first child, letter "c" for the second child, "d" for the third child and so on. For instance, the first child of node "1,b" is "2,b.b", the second child of "1,b" is "2,b.c", and the third child of "1,b" is "2,b.d" and so on. Likewise, child nodes of "1,c" are "2,c.b", "2,c.c", "2,c.d" and so on. The first numeric value is changing according to the tree level.

Inserting new node(s) can be done using the following rules: If there is no node standing before the place that a new node shall be added, unique code of the new node is the code of its following sibling node minus one value from the last letter. If the last letter of the code of the new node is "a", attach "b" at the end. For example, inserting a node before node "2,b.b", minus one value from the last letter ("b"), we get "a", then attach "b" at the end, thus we have "2,b.ab" as the code for the inserting node. Similarly, to insert a new node before node "2,b.ab", minus 1 from "b", we get "a", because of "a", we need to attach "b" at the end, thus we have "2,b.aab" or "2,b.a^2b" as the code for the new inserting node. See Figure 6.5.

To insert a new node between exiting nodes, add 1 value from the code of its preceding sibling node, ensuring that the code for the new node will be greater than the code of its preceding sibling node and less than the code of its following sibling node in alphabetical order. If the value of the new node equals to the code of its next sibling, attach `"b"` instead.

If the code of its preceding node ends with `"z"`, attach `"b"` at the end. For example, inserting a new node after node `"1,c"`, we can not add 1 value to `"c"` because its following sibling is `"1,d"`. In this case, we attach `"b"` at the end, thus we have `"1,cb"` for the new node. On the other hand, code for the new node after `"1,cb"` will get `"1,cc"` because it does not violate the rule. However, when the last value of the code of its following sibling is `"b"`, `"a"` can be used as a special case.

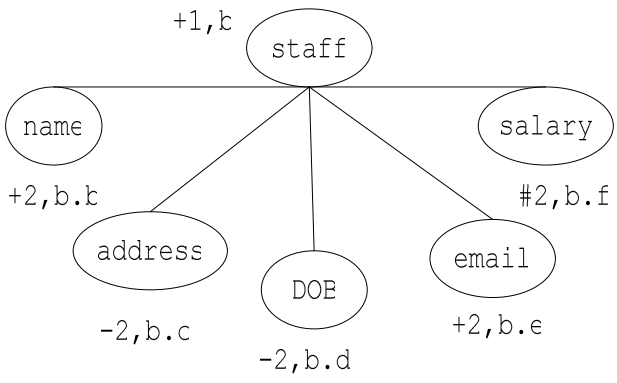


Figure 6.6. Access rules associated with a labelling scheme.

Figure 6.6 shows an example of how our access control integrates with Com-D labelling scheme. This integration will help to bypass the need of repetitive labelling and pruning processes to determine what users can actually view. Furthermore, there is no need to rewrite queries.

***Case 3*:** Suppose that a user belongs to student group, and he/she has a public access level. He/she requests all staff details, such as `//staff/*`. Normally, with existing node filtering approach, this requires an access controller to do schema verification and labelling (with permission or denial) processes to determine what this user can view then pruning tree to display the result.

Imagine when more than one user requesting access to XML data, access controller has to repeat all the same processes for each individual user. This could take enormous times and memory for parsing and searching XML data.

With our hybrid access control model, these unnecessary steps can be eliminated. In the above query, we will look at all staff details, which are ready in our indexes. Only fields that are marked with public access denoted as "+" are returned to this user. Thus, all private and protected fields are hidden from this user.

Case 4: Now, considering Julia, an administrator who has access right as private level. She requests to view all staff salary such as `//staff//salary`. This obviously fails. Keep in mind that access controller is always behind the indexes to determine a user view. When query processor looks for salary, this field requires a protected access level `(#)`. Because Julia does not have sufficient access right, her request is not allowed. In contrast, if the field she requests is private, then she could successfully access to those fields.

It is worth to mention that, because all staff can read all of their own details, when Lisa sends a query to view all of her own details, she is granted access provided that the condition `WHERE staff-id = "her-staff-id"` is met. It is the task of access controller to validate access authorization, which cannot be enforced by the query processor.

Case 5: Let us consider another special case where our access controller interactively works with our indexes. The case in the above subsection states that, Lisa is the head of Computer Science Department. She has the right to access protected fields of Computer Science Department and public fields of other departments. The query processor will first look for indexes of each department. If department is Computer Science, all protected level fields

and those below this level are returned (e.g. those marked with `+, - and #`). On the contrary, for other department, only fields marked with public `(+)` are returned.

6.3 Process Analysis

To demonstrate how our integrated access control model can be better off comparing to a traditional node filtering technique such as [Damiani, Fansi, Gabillon and Marrara (2007), De Capitani di Vimercati, Marrara and Samarati (2005), Damiani, De Capitani di Vimercati, Paraboschi and Samarati (2002, 2000)]. We set up a list of required processing steps when dealing with a user request. See Table 6.2.

To determine a user view, processing steps such as identifying user, checking access authorization, determining user views are required by both SecureX and node filtering technique. For SecureX, the last step is retrieving data using our indexes, and then result will be returned to user. For node filtering technique, more steps need to be done here. Parsing XML step is required next, then creating and labelling the DOM tree with permission `"+"` or denial `"-"` are followed.

Furthermore, pruning process will be carried out to remove all nodes that are labelled by `"-"`. Finally, the DOM tree with all `"+"` is shown to user.

Regretfully, these processes are repeated for every user and every action a user takes. On the contrary, in our model, these timing processes are not necessary and can be bypassed. Consequently, this will speed up the search and query processes.

Table 6.2. Comparison of processes taken upon a user's request

Process Steps	Node Filtering	SecureX
Query executor	1. Checking request 2. Identify user	1. Checking request 2. Identify user
Determine access right of user	3. Check Authorization Schema 4. Check Authorization Document	3. Check access group
Process query based on user's view	5. Parsing XML 6. Tree labelling (with permission "+" or denial "-")	4. Filtering data from index
Prepare user's view	7. Pruning process	
Returning result	8. Display user view	5. Display user view

In addition, Node Filtering technique [Damiani, Fansi, Gabillon and Marrara (2007), De Capitani di Vimercati, Marrara and Samarati (2005), Damiani, De

Capitani di Vimercati, Paraboschi and Samarati (2002, 2000)], cannot explicitly define a rule in which all users are allowed to access their own information but not those of others.

6.4 XML Update Control

Currently, sending queries to request updating XML data is not yet allowed to users and is still a research issue. However, with a strong development of XML updating languages, e.g., Chamberlin, Florescu and Robie (2006), Tatarinov, Yves, Halevy and Weld (2001), Laux and Martin (2000), updating XML will soon be available to users and become standardized. Consequently, the need for managing XML update emerges.

Among XML updating languages, an update facility from W3C (Chamberlin et al. 2006) which is still in working draft, is the one that extends the XML Query language to make persistent changes to instances of the XQuery 1.0 and XPath 2.0 Data Model. Some operations provided by XQuery Update Facility are insertion, deletion of a node, modification of a node by changing some of its properties etc. Complete description can be found at (Chamberlin et al. 2006).

In our update control model, we make use of update operation types from (Chamberlin et al. 2006) and redefine them as four write privileges: Update, Insert, Rename and Delete. Details of these actions are described as follows.

- Update allows the content of a node to be changed.
- Insert operation inserts one or more nodes into a chosen position with respect to a target node. Detail actions of insert operation are as follows:
    - Insert before or after: the inserted nodes become the preceding (or following) siblings of the target node.
    - Insert first or last: the inserted nodes become the first (or last) child of the target node.
- Rename replaces a name property with a new qualified name.
- Delete operation deletes one or more nodes.

Now imagine if a staff sends queries such as `delete /company/customers//*` to request deletion of all customers' details of his/her company, or s/he accesses to the payroll to increase all staffs salary by twenty percent. This is insecure. There must be a rule controlling who can insert, update or delete particular information in a system. This is about data security and data integrity. We have discussed data security in the above section. In the next subsection, we will introduce our XML update control concepts that can deal with the above issues.

6.4.1 Update Control Concepts

Four write privileges mentioned above are described as follows.

- If a user holds an Update privilege on a node *u*, he/she is allowed to update the content of node *u*.
- If a user holds an Insert privilege on a node *u*, he/she is allowed to insert a new node or sub-tree which is a child of node *u*.
- If a user holds a Rename privilege on a node *u*, he/she is allowed to rename node *u*.
- If a user holds a Delete privilege on a node *u*, he/she is allowed to remove node *u* and its sub trees.

When updating is allowed, the structure of XML document may be changed. For example, a sale representative, Tom has read and insert privileges on a list of customers. Instead of adding only required details such as `name, address and contact number`, he inserts additional details such as `vacation-address, emergency contact`, which for him are important. This violates existing DTD and alters XML structure. There must be a more specific rule to determine who can update and/or change XML structure.

In practice, we believe that, the task of changing the structure of database should only be assigned to the owner of data or people whose positions are at the managerial level. Other users can update information but not the structure of data. Should the XML structure needs to be modified, for example, a new field is required to store in database, such action should only be taken by the owner or executives. They first need to update DTD and then defining a new access authorization rule for the new field.

Table 6.3 Update Types

Operation	**Update Type**	**Description**
Update	U	Update content of node, no structure change
Insert	SI	Insert new node, Structure change allowed
	I	Insert new node, structure change is NOT allowed
Rename	SR	Rename node, structure change allowed
	R	Rename node, structure change is NOT allowed
Delete	SD	Delete node, structure change allowed
	D	Delete node, structure change is NOT allowed

In our write privileges, update operation only allows updating content of nodes, thus, XML structure remains the same. Insert, rename and delete

operations may or may not require structure changed. In order to identify who can update data and who can change the structure of XML data, we introduce seven update-types for write privilege in our model. Table 6.3 shows details of these update authorization rules and its description.

Case 6: In case 1 of the above section, John can access all of his information and public information of others. If he moves to a new address, could he update his information by himself or should an authorized person be needed to do this job. This is up to the policy of his organization. Suppose his organization allows staff to update their own details. Then, John needs a write privilege to update his own information.

Although fields like name, address, phone, email etc. can be updated by staff, salary and job position are not allowed for staff to update. To express this circumstance, we now classify update authorization for each field. In XAA, Figure 6.7(a), we can see that, although position is a public field and all users can read it, not everyone is allowed to update it, even for the staff. This is because one cannot promote (or demote) oneself prior to getting the approval from the boss.

In XAG, see Figure 6.7(b), the new update type for each user group is specified. With the updated access rules, apart from having read privilege

(self and public information), staff can now update his/her self private information. Note that, position and salary are protected information, see Figure 6.7(a). Staffs are not allowed to update these details. Thus, we do not need to state this rule again.

```
<XAA doc="Employee.xml">
<rule object="//name" access="+" update= "-" type="R"/>
<rule object="//address" access="-" update= "-" />
<rule object="//DOB" access="-" update= "-"/>
<rule object="//h_phone" access="-" update= "-" />
<rule object="//office" access="+" update= "-"/>
<rule object="//extension" access="+" update= "-" />
<rule object="//email" access="+" update= "-" />
<rule object="//position" access="+" update= "#"/>
<rule object="//salary" access="#" update= "#"/>
......
</XAA>
```

(a)

Figure 6.7(a). Updated XAA with Write privileges.

```
<XAG doc="Employee.xml">
```

```
     <group entity="student" access-type="+" update-
type="denial" />
<group entity="staff" access-type="$, +" update-
type="U$-" /group>
<group entity="admin" access-type="$, -" update-type="U-
, I-" />
<group entity="executive" access-type="#" update-
type="U#, SI, SR, SD" />
…
</XAG>
```

(b)

Figure 6.7(b). Associate XAG.

Case 7: Let us consider Ian, the middle level administrative officer of Computer Science department, who has read privilege to all protected fields in his department. For write privilege, security officer states that he can update protected fields (`position and salary`) for employees in his department, however, he cannot update protected information of the head of school. Below is the rule setting up for him.

```
<group entity="Ian">
```

```
    <rule object="//dept[@name='CompSci']" access-
    type="#" update-type="U#" type="R" />
    <rule object="//staff/position |//staff/salary
    [//staff/position='Head of School']" update-
    type="denial" />
</group>
```

Case 8: Let consider another special case, where a user has an insert privilege such as *update-type= "I#"*, therefore, he/she is allowed to insert protected details of a new staff such as `name, address, phone, email, salary etc`.

Note that these fields must correspond to existing DTD. If he/she inserts a field such as `personal-interest`, the operation will be rejected because he/she is not allowed to violate DTD and change XML structure. Similarly, if he/she inserts an authorized field but not in a right order as in DTD, the operation will also be rejected.

In contrast, imagine a user who is at the managerial level and has an insert privilege such as *update-type= "SI"*. When he/she inserts a new node such as `biography`, this operation is allowed because he/she is allowed to change the structure of XML data. Certainly, updating to DTD is then required and access authorization rule needs to be defined.

6.5 Integrate Write Privilege with XML Labelling

In this subsection, we discuss how to integrate write privilege with a labelling scheme. Similar to read privilege, with an XML data and an associated XAA file, we can create an index for this XML to classify update level of each element. Although both read and write privileges can integrate with a labelling scheme by using `<read-privilege>`, `<node label>`, `<write-privilege>` format. For a clarity reason, we only show labels with write privilege in this subsection.

Let us consider a low-level administrator with update authorization is `"U-"` sending a request to update address of John. The system will first look and retrieve all addresses in the index which are labelled such as `{"-b.c", "-c.c", "-d.c"}` then check these nodes for the address that belongs to John. For example, it returns node `"-b.c"` which is the address of John. Then, the operation is allowed and updating is done. In contrast, if this user requests to update position of John, which labelled such as `"#b.d"`, the system can then determine that this is a protected field, because the update right of this user is private only, `"U-"`, access controller will reject the operation due to insufficient authorization level.

The advantage of combining indexing or labelling scheme with access authorization rule can obviously facilitate query processing. It helps to bypass the need of scanning the whole table to search for result. Searching processes are reduced and frequent checking for access rules from XAA file is not necessary. Furthermore, when a node is inserted, label of the newly inserted node and its associated access rule can be generated on the spot without the need of parsing XML document or relabel existing nodes.

6.6 Dealing with Conflict or Undefined Rule

In our model, we make assumption that access policy exists for every XML file, whoever creates the XML file also creates the access policy for that file. If there is no access rule associated with a particular file, for the security reason, we will consider this file as uncompleted XML data and this file will not be available for users. This prevents the XML owner from forgetting to load the XAC together with the XML file. To make the XML file entirely public, rule can be defined in the XAC file as:

```
<rule object="*" access="+">
```

Similarly, if there is no access rule associated to a field, that field will not be available for users in our model. This is designed to avoid missing rules for private or protected fields.

There could be a case in which multiple or different access rules apply for one subject/object. To solve this conflict, we assign that the most specific subject takes precedence and denials take precedence principles. Our chain of command rule, `deny < public < private < protected` is also implemented.

Finally yet importantly, we would like to bring up an issue from other works that still left open. Consider Anne who requests a deletion of node *u*. When she deletes node *u*, she also deletes its sub-tree. The problem states, some of the nodes, which belong to that sub-tree, are not accessible by Anne. A question is then raised; shall the operation be rejected if some nodes of the deleted sub-tree do not belong to the user's view? This is a typical issue between data confidentiality and data integrity. Existing decisions are based on whether they emphasize the confidentiality or integrity. For those who emphasize on data confidentiality, then delete operation is accepted.

Here, we believe that, this dilemma is caused by inconsistent access authorization rules, and is avoidable. Consider this scenario, Mike asks a gardener to prune his citrus tree. He says that the gardener can prune off branch *A* and branch *B* etc. When the gardener finishes the job, Mike realizes that, he did not want to cut a sub branch of branch *A*. This clearly shows

that, if Mike wanted to keep a sub branch of branch `A`, he should have not allowed the gardener to cut branch `A`.

Similarly, Leanne is a receptionist, who does general duties in an office. One day, her boss asked her to shred confidential documents. Without a doubt, she could view all confidential information of the company that normally she would not be able to see.

In order to keep away from this dilemma, we look at several aspects and employ unambiguous update types when defining access authorization for users.

- We carefully consider their levels in the organization and the confidentiality and integrity of XML data.

- Whether or not the deletion of data is more important than keeping the data. E.g. For a staff/customer who is no longer with company, a low-level administrative should better make inactive remarks rather than removing associated nodes from the system. Should the deletion is required; it should then be done by someone who is at a higher level and has access to the node and its sub tree. It is because delete action may cause changes to XML structure.

- While update and insert new nodes are needed more often and usually can be done by low level administrative people, rename, delete operations or other operations that could change XML structure should only be done by someone who belongs to high level of managerial. Our update types can clearly state these situations. For example, "I" for Insert operation with no structure change allowed. "SI", the insert operation with structure change allowed. Similarly, "D" and "SD" for delete operations, etc. Access right of each user is specified in a simple way yet totally applicable for day-to-day operations.

6.7 Experiments

We implemented our proposed SecureX access control, which integrates with a dynamic labelling scheme Com-D and a typical Node-Filtering technique in Damiani, Fansi, Gabillon and Marrara (2008, 2007), De Capitani di Vimercati, Marrara and Samarati (2005), Damiani, De Capitani di Vimercati, Paraboschi and Samarati (2002, 2000) to observe its query performance. Experiments were carried out on a Pentium IV 2.4G with 512MB of RAM running on windows XP with 25G hard disk. We used Java and SAX from Sun Microsystems as the XML parser.

For the database, we obtained the Department dataset from Niagara Project. (http://www.cs.wisc.edu/niagara/data). In lieu of a larger sized data, we replicated the Department dataset 10 times.

We created a set of query and applied these queries against SecureX and Node-Filtering technique. For each query, we compared number of nodes needed to be scanned to get the result and its responded time.

```
<?xml version="1.0"?>
<!ELEMENT department (deptname, (gradstudent | staff |
faculty | undergradstudent)*)>
<!ELEMENT gradstudent (name, phone, email, address,
office?, url?, gpa)>
<!ELEMENT staff (name, phone, email, office?)>
<!ELEMENT faculty (name, phone, email, office)>
<!ELEMENT undergradstudent (name, phone, email, address,
gpa)>
<!ELEMENT name (lastname?,firstname)>
<!ELEMENT address (city, state, zip)>
<!ELEMENT deptname (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
```

```
<!ELEMENT office (#PCDATA)>

<!ELEMENT phone (#PCDATA)>

<!ELEMENT lastname (#PCDATA)>

<!ELEMENT firstname (#PCDATA)>

<!ELEMENT url (#PCDATA)>

<!ELEMENT gpa (#PCDATA)>

<!ELEMENT email (#PCDATA)>
```

Figure 6.8 The Department DTD

Access rules for this dataset are assumed as follow. `Department`, `Deptname`, `Faculty` and `Staff` nodes are publicly available; they can be accessed by any users. Children of `Staff` and `Faculty` are also public. `Gradstudent`, `Undergradstudent` and their details (all child nodes) are private fields thus can only be accessed by authorized people such as staff or administrators etc.

Furthermore, staff and faculty can access to private fields (graduate and undergraduate students) in their department only. Students can access to public fields and their own information. They cannot access information of other students. The characteristics of queries used in this experiment can be found in Table 6.4.

Table 6.4.Characteristics of Used Queries

	Query	**Access Level**	**Description**
Q1	//*	Public	Only public information will be returned
Q2	//*	Admin	All public and private information are returned
Q3	//*	CS Staff	All public info and only private info in Comp Sci department are returned. Since staff can only view private info in their own department.
Q4	//*	Student	Only public and student self information are returned
Q5	/department/gradstudent //*	Public	Insufficient access level, thus query is rejected.
Q6	/department/gradstudent //*	Admin	All graduate students are returned
Q7	/department[./deptname = "afr"]/staff/phone/	Student	All phone of staff of African Studies dept are returned
Q8	/department/ undergradstudent//*	Admin	All undergraduate students are returned

Q9	/department/faculty/phone/	Staff	All faculty's phone are returned
Q10	/department[./deptname = "cs"]/faculty/email/	Staff	All email of faculty of Comp Sci dept are returned
Q11	/department/ungradstudent/ email/	Admin	All undergraduate students email are returned

In term of measuring query performance, we are only interested in processing time. For example, given a user with an access level and a request query, we will determine the time needed to answer that request and number of nodes that needed to be scanned against number of nodes retrieved. Note that, the pre-checking steps such as validating users are assumed done.

6.7.1 Node Scan Observation

Figure 6.9 represents a comparison between SecureX and a Node-Filtering technique for the total number of nodes needed to be scanned for each query. From this experiment, we found that, Node Filtering technique would have to scan excessive nodes when users with high access levels request little information.

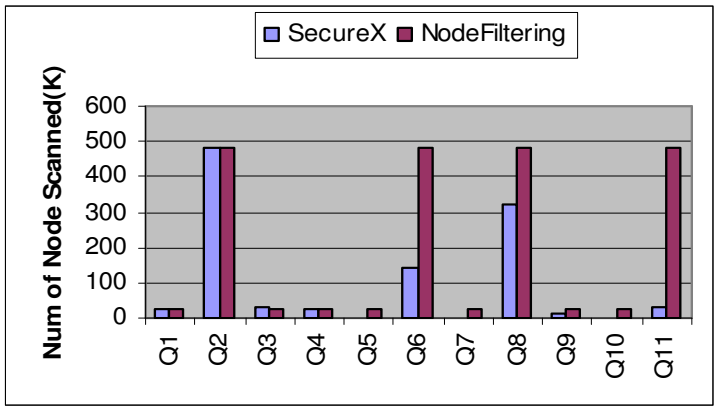


Figure 6.9. Number of Node Scanned

For instance, Q6 and Q11 are requested by an administrator. When Node Filtering technique is used, all information that he/she can access are made available. Even after the tree is pruned, a huge amount of node becoming accessible to this user. However, he/she might only be interested in a piece of information such as students email. To get the result for this simple request, the system has to search the pruned tree excessively for that piece of information. The fact is the larger of the pruned tree, the more nodes the system has to scan.

In contrast, SecureX is very effective in term of searching for result. This achievement is gained because SecureX is designed to integrate with a labelling scheme, thus, unnecessary nodes can be skipped.

On the other hand, when it comes to a query such as Q1 and Q2, which users request to see all nodes, Node Filtering technique works as good as SecureX because the number of scanned nodes and the number of retrieved nodes are identical.

Table 6.5 Number of node retrieved vs. number of node scan

Query	# of Node Retrieved	# of Node Scanned	
		SecureX	Node-Filtering
Q1	24330	24786	24330
Q2	485420	485420	485420
Q3	27710	28142	27710
Q4	24343	24788	24343
Q5	0	0	24332
Q6	140000	140000	485420
Q7	120	120	24330
Q8	321090	321090	485420
Q9	1750	11290	24330
Q10	100	100	24330
Q11	29190	29190	485420

6.7.2 Response Time

Results of this experiment shown in Figure 6.10. Again, SecureX shows its effectiveness comparing to a Node Filtering technique. With a closer observation, for the fastest processes, SecureX took near 0 millisecond in Q5 and Q10 while Node Filtering technique took at least 41000 milliseconds to answer these queries.

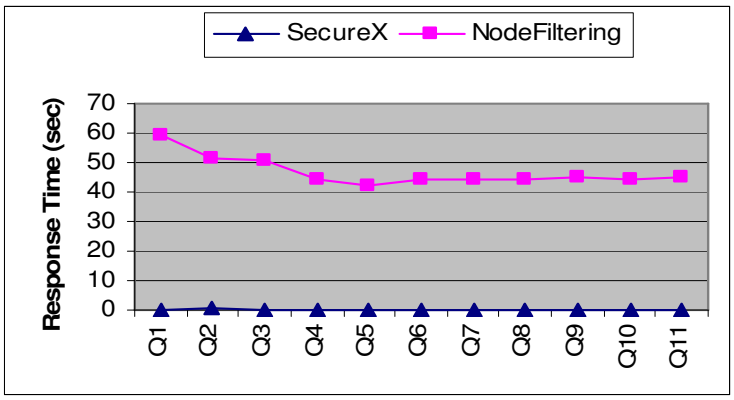


Figure 6.10. Queries Response Time.

In Q2, SecureX took 516 milliseconds while Node Filtering took 52000 milliseconds to answer this query. As analysed in Session 6.3, when a new user sending a request, Node Filtering technique needs to repeat all parsing, labelling then tree pruning processes. This is time consuming. In contrast, SecureX can take advantage of an indexing/labelling scheme to produce quick, yet secured results.

Table 6.6 Queries results.

Query	# of Node Retrieved	# of Node Scanned		Response time (ms)	
		SecureX	Node Filtering	SecureX	Node Filtering
Q1	24330	24786	24330	47	59422
Q2	485420	485420	485420	516	51157
Q3	27710	28142	27710	47	50890
Q4	24343	24788	24343	62	44577
Q5	0	0	24332	0	41798
Q6	140000	140000	485420	110	44406
Q7	120	120	24330	0	44094
Q8	321090	321090	485420	266	44188
Q9	1750	11290	24330	16	45032
Q10	100	100	24330	0	44312
Q11	29190	29190	485420	47	44953

6.8 Summary

Many existing access controls use node filtering or querying rewriting techniques. These techniques require rather time-consuming processes such as parsing, labelling, pruning and/or rewriting queries into safe ones each time a user requests a query or takes an action. In this chapter, we have proposed a fine-grained access control model, named SecureX, which

supports read and write privileges. With our novel access control concept, various access types are introduced, including those for determining if a user has the right to change XML structure.

Furthermore, SecureX can be integrated well with a dynamic labelling scheme to eliminate repetitive labelling and pruning processes when determining a user view. This brings about advantages of speeding up searching and querying processes. When comparing to a traditional node filtering technique, our integrated access control model takes less processing steps. Experiments have shown effectiveness of our approach.

7

Conclusion

In this thesis, after examining advantages and disadvantages of some related works on access control, indexing, numbering and labelling techniques for XML data, we have proposed two dynamic labelling schemes and a fine-grained access control model, SecureX, for securely querying and updating XML data.

With our dynamic labelling schemes, LSDX and Com-D, they are both support updating XML data dynamically without the need of re-labelling existing nodes, hence facilitating fast update. LSDX also supports the representation of the ancestor – descendant relationships and sibling relationships between nodes. Moreover, our LSDX is capable of showing the depth of the data tree.

In addition to that, when these labelling schemes are implemented, its unique way of labelling nodes shall help one quickly gain access to a specific level and a specific node. As a result, when retrieving, inserting, deleting

and updating XML data are required, our labelling schemes will help to reduce the number of nodes that would otherwise need to be accessed to carry out those tasks. Consequently, these advantages shall make those tasks a lot easier and help to save time.

Our experiments show that in term of total length of labels, our LSDX labelling scheme is about two times shorter comparing to GRP (Lu and Ling, 2004) and about 7 - 18 times shorter comparing to SP scheme (Cohen, Kaplan and Milo, 2002). Generating labels for XML documents vary from 1 second for 1.2MB of data to one minute for 100MB of data. In term of permanently storing individual changes in the files, time used for insertion and deletion are considered spectacularly quick. This will be useful when two or more programs need to use the same XML data concurrently.

Furthermore, our Com-D labelling scheme also supports updating XML data dynamically without the need of re-labelling existing nodes, hence facilitating fast update. Moreover, our proposed Com-D labelling scheme is more compact than existing ones. Our experimental works show that, Com-D labelling scheme is superior to all ORDPaths, GRP, Dewey, SP one bit and double bit schemes.

Com-D labelling scheme also supports all important axes in XPath such as

parent, child, ancestor, descendant, previous – sibling, following – sibling, previous, following.

In addition to those advantages, using our dynamic labelling schemes as an index structure shall reduce the number of nodes that would otherwise need to be accessed for searching or querying purposes.

We have also proposed a fine-grained access control model, SecureX, for securely querying and updating XML data. With our novel access control model, we can define access authorization rules for users explicitly. Case such as information of a user, which is available to self access only, is also managed sensibly. Moreover, we have considered update operations made by users and introduce seven update types to determine if a particular user has the right to change XML structure.

In addition, SecureX can easily be integrated with any numbering/labelling scheme to take the advantage of speeding up the search and query processes. We have illustrated how our access control model integrates with a dynamic labelling scheme and performed comparison between SecureX and typical Node Filtering techniques. Our analysis shows that, our proposed model requires less processing steps. Our experiments have

proved its effectiveness. We have also pointed out a shortcut to determine if a node is accessible to a particular user.

Possible Future Work

Some possible means of extending the research presented in this thesis are given below:

- Considering update operations for XML documents, which may not have DTD for verification → May need to investigate and improve update processes by eliminating unnecessary verification steps caused by update operations as possible.

- Conducting more comprehensive experimental works regarding to order - sensitive queries and update performances of the Com-D labelling scheme and compare results with other existing labelling schemes.

- As Com-D labelling scheme has the potential to facilitate query processing, we also hope to develop a query tool for XQuery based on this labelling scheme.

Bibliography

Abitboul, S., Quass, D., McHugh, J., Widom, J. and Wiener, J. *The Query Language for Semistructured Data.*

Ait-Kaci, H., Boyer, R., Lincoln, P. and Nasr, R. (1989): *Efficient implementation of lattice operations*, ACM Transactions on Programming Languages and Systems, 11(1):115-146, 1989.

Alstrup, S. and Rauhe, T. (2002): *Improved Labelling Scheme for Ancestor Queries.* In proceedings of the 13th annual ACM-SIAM Symposium on Discrete Algorithm, 2002.

Amagasa, T., Yoshikawa, M. and Uemura. S. (2003): *QRS: A Robust Numbering Scheme for XML Documents*, ICDE, 2003.

Amato, G., Debole, F., Rabitti, F. and Zezula, P. (2003): *Yet Another Path Index for XML Searching,* in Proceedings of ECDL 2003. Research and Advanced Technology for Digital Libraries, 7th European Conference, Trondheim, Norway, 2003.

Bertino, E. and Ferrari, E. (2002): *Secure and selective dissemination of xml documents.* ACM TISSEC, 5(3):795--825290--331, 2002.

Bertino, E., Castano S. and Ferrari, E. (2001): *Securing xml documents with author-x*. IEEE Internet Computing, 5(3):21--31, 2001.

Bertino, E., Castano S., Ferrari, E. and Mesiti, M. (2000): *Specifying and enforcing access control policies for xml document sources*. World Wide Web Journal, 3(3):139--151, 2000.

Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J. and Siméon, J. (2007): *XQuery 1.0: An XML Query Language*. W3C Recommendation. http://www.w3.org/TR/xquery/, 2007

Bonifati, A. and Ceri, S. (2000): *Comparative Analysis of the Most Representative XML Query Languages*, Dec 2000.

Catania, B., Ooi, B., Wang, W. and Wang, X. (2005): *Lazy XML Updates: Laziness as a Virtue of Update and Structural Join Eciency*. In Proc. of the ACM SIGMOD 2005.

Chamberlin, D., Florescu, D., Melton, J., Robie, J. and Siméon, J. (2008): *Xquery update facility*. http://www.w3.org/TR/xquery-update-10/, 2008.

Chamberlin, D., Robie, J. and Florescu, D. *Quilt: An XML Query Language for Heterogeneous Data Sources*".

Chen, Y., Mihaila, G., Bordawekar, R. and Padmanabhan, S. *L-Tree: a Dynamic Labelling Strucutre for Ordered XML Data*.

Cohen, E., Kaplan, H. and Milo, T. (2002): *Labelling dynamic XML trees,* in Proceedings of PODS 2002.

Connolly and Begg. *Database Systems*, 3rd Edition Ch 15, Ch 29.

Cooper, F. B., Sample, N., Franklin, J. M., Hjaltason, R. G., and Shadmon, M. (2001): *A Fast Index for Semistructured Data,* in Proceedings of VLDB Conference, 2001.

Cormen, H. T., Leiserson, E. C., Rivest, R. L. and Stein, C. (2001) *Introduction to Algorithms.* Ch 13, Second Edition 2001.

Damiani, E., Fansi, M., Gabillon, A. and Marrara, S., (2008): *A general approach to securely querying XML.* Computer Standard & Interfaces vol 30 (2008) p 379-389.

Damiani, E., Fansi, M., Gabillon, A. and Marrara, S. (2007): *Securely querying and updating xml*. Submitted to ICDE 07, 2007.

Damiani, E., Fansi, M. Gabillon, A. and Marrara, S. (2007): *A General Approach to Securely Querying XML*. Proc. of the 5th International Workshop on Security in Information Systems (WOSIS 2007). 12-13 June, 2007 - Funchal, Madeira - Portugal. p 115-122.

Damiani, E., De Capitani di Vimercati, S., Paraboschi,S. and Samarati, P. (2002): *A fine-grained access control system for xml documents*. ACM TISSEC, 5(2):169--202, 2002.

Damiani, E., De Capitani di Vimercati, S., Paraboschi, S. and Samarati. P. (2000): *Design and implementation of an access control processor for xml documents*. Computer Networks, 33(1-6):59--75, 2000.

Damiani, E., De Capitani di Vimercati, S., Paraboschi,S. and Samarati, P. (2000): *Securing xml documents*. In Proceedings of the 2000

International Conference on Extending Database Technology, EDBT 2000, Konstanz, Germany, March 2000.

Damiani, E., De Capitani di Vimercati, S., Paraboschi, S. and Samarati. P. (2001): *Controlling access to xml documents*. IEEE Internet Computing, 5(6):18--28, 2001.

De Capitani di Vimercati, S. Marrara, S. and Samarati, P. (2005): *An access control for querying xml data.* In Proceedings of SWS05 Workshop, 2005.

Derksen, E., Fankhauser, P., Howland, E., Huck, G., Macherius, I., Murata, M., Resnick, M. and Schöning, H. (1999): *XQL (XML Query Language)*. http://metalab.unc.edu/xql/xql-proposal.xml

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1998): *XML-QL: A Query Language for XML*. http://www.w3.org/TR/NOTE-xml-ql/

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. *A Query Language for XML.* See http://www.research.att.com/~mff/files/final.html

Dietz, P. (1982): *Maintaining order in a linked list,* in Proceedings of the 14th Annual ACM Symposium on Theory of Computing, p122–127, San Francisco, California, May 1982.

Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M, Siméon, J.and Wadler, P. (2007): XQuery 1.0 and XPath 2.0 Formal

Semantics. W3C Recommendation. http://www.w3.org/TR/xquery-semantics/, 2007.

Duong, M. and Zhang, Y. (2008): Dynamic Labelling for XML Data Processing. In Proceedings of the 7th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE'08), 2008.

Duong, M. and Zhang, Y. (2008): *An Integrated Access Control for Securely Querying and Updating XML Data.* In 19th Australasian Database Conference (ADC2008), Wollongong, Australia. Conferences in Research and Practice in Information Technology, Vol. 75.

Duong, M. and Zhang, Y. (2005): Lsdx: A new labelling scheme for dynamically updating xml data. In 16th Australasian Database Conference, Vol 39, Newcastle, Australia. pages 185--193, 2005.

Elmasri and Navathe. *The fundamental of Database Systems*, 4th Edition Ch 12, Ch 15.

El-Sayed, M., Dimitrova, K. and Rundensteiner, E. (2003): *Efficiently Supporting Order in XML Query Processing*, WIDM'03, November 7-8, 2003, New Orleans, Louisiana, USA.

Esposito, D. (2001): *The XML Query Language*, Available at http://www.winnetmag.com/SQLServer/Article/ArticleID/19958/SQLServer_19958.html

Fan, W., Chan, C. and Garofalakis, M. (2004): *Secure XML with security views.* In Proceedings of ACM SIGMOD 2004, June 13-18, 2004, Paris, France, 2004.

Fisher, D., Lam, F. and Wong, R. (2004): *Algebraic Transformation and Optimization for XQuery,*APWeb 2004, LNCS 3007, pp 201-210, 2004.

Florescu, D. and Manolescu, L. *Integrating Keyword Search into XML Query Processing*. See http://www9.org/w9cdrom/324/324.html

Fundulaki, I., and Marx, M. (2004): *Specifying access control policies for xml documents with xpath.* In SACMAT'04, June 2-4 2004.

Gabillon, A. (2004): *An authorization model for xml databases*. In ACM Workshop on Secure Web Services, October 29, 2004, Fairfax VA, USA, 2004.

Grust, T. (2002): *Accelerating XPath Location Steps,* in Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, ACM 2002.

Hou, J., Zhang, Y. and Kambayashi, Y. (2001): *Object-Oriented Representation for XML Data*, Proceedings of the 3rd International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'2001), April 23-24, 2001 in Beijing, China, IEEE CS Press.

Kaelin, M. (2004): *Database Optimization: Increase query performance with indexes and statistics*. TechRepublic,

http://techrepublic.com.com/5100-6313_11-5146588.html?tag=search.

Kaplan, H., Milo, T. and Shabo, R: *A Comparison of Labelling Schemes for Ancestor Queries.* http://www.math.tau.ac.il/~haimk/papers/comparison.ps

Krall, A., Vitek, J., and Horspoo, N. (1997): *Near optimal hierarchical encoding of types,* in the European Conference on Object Oriented Programming, ECOOP'97. P 128-145, Finland.

Kudo, M. and Hada, S. (2000): *Xml document security based on provisional authorization.* In Proceedings of the 7th ACM Conference on Computer and Communications Security, Nov. 2000.

Laux, A. and Martin, L. (2000): *Xml update (xupdate) language*. Technical report, XML: DB working draft, September 2000.

Lee, D., Lee, W. and Liu, P. (2003): *Supporting xml security models using relational data-bases: A vision*. In XSym 2003, LNCS 2824, 2003, pages 267--281, 2003.

Lee, K.Y., Yoo, S. J. and Yoon, K. (1996): *Index structures for structured documents*, in ACM First International Conference on Digital Libraries, p 91-99, Bethesda, Maryland, March 1996.

Li, C., Ling, W. T. and Hu, M. (2006): *Efficient processing of updates in dynamic xml data*. In International Conference on Data Engineering, ICDE 2006.

Li, C. and Ling, W. T. (2005): *An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML*. DASFAA 2005, Beijing, 17-20 April 2005.pages 125-137.

Li, Q. and Moon, B. (2001): *Indexing and Querying XML Data for Regular Path Expressions,* in Proceedings of VLDB 2001.

Lu, J., Ling, T., Chan, C. and Chen, T.(2005): *From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching*. Proc. of the VLDB 2005.

Lu, J. and Ling, W. T. (2004): *Labelling and Querying Dynamic XML Trees*, in Proceedings of 6th Asia Pacific Web Conference, APWeb 2004, China.

Meuss, H. and Strohmaier, M. C. (1999): *Improving Index Structures for Structured Document Retrieval.* In 21st BCS IRSG Colloquium on IR, Glasgow, 1999.

Milo, T. and Suciu, D. (1999): *Index Structures for Path Expression*. In proceedings of 7th International Conference on Database Theory, 1999.

Mohan, S., Sengupta, A., Wu, Y. and Klinginsmith, J. (2005): *Access Control for XML - a dynamic query rewriting approach*. In Proceeding of VLDB 2005 Conference, 2005.

Niagara Project. http://www.cs.wisc.edu/niagara/

O'Neil, P., O'Neil, E., Pal, S., Cseri, S., Schaller, G. and Westbury, N. (2004): *Ordpaths: Insert-friendly xml node labels*. In proceedings of the 2004 ACM SIGMOD, Paris, France, 2004.

Robie, J., Chamberlin, D. and Florescu, D. (2000): *Quilt: An XML Query Language*, Mar 2000. Available at http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html

Schmidt, A., Waas, F., Kersten, M., Carey, J. M., Manolescu, I. and Busse, R. (2002): *XMark: A Benchmark for XML Data Management,* in Proceedings of VLDB 2002.

Silberstein, A., He, H., Yi, K. and Yang, J. (2005): *BOXes: Efficient maintenance of order-based labeling for dynamic XML data*. In the 21st International Conference on Data Engineering (ICDE), 2005.

Sheth, S. and Miller, A. J. (1999): *Query Languages and Tools for XML Documents and Databases,* July 1999.

Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C. (2002): *Storing and Querying Ordered XML Using a Relational Database System*, in Proceedings of SIGMOD 2002.

Tatarinov, I., Yves, Z., Halevy, A. and Weld, D. (2001): *Updating xm*l. In ACM SIGMOD, 2001.

Wang, J. and Osborn, S. (2004). *A role-based approach to access control for xml databases.* In SACMAT'04, Yorktown Heights, New York, USA, June 2-4, 2004.

Wang, W., Jiang, H., Lu, H. and Yu, X. J. (2003): *PBiTree Coding and Efficient Processing of Containment Joins.* In 19th International Conference on Data Engineering, 2003 Bangalore, India.

World Wide Web Schools. *XML, XPath, XQuery, DTD and WAP Tutorial* Available at http://www.w3schools.com/

Wu, X., Lee, M. and Hsu, W. (2004): *A prime number labeling scheme for dynamic ordered xml trees*. In proceedings of the 20th International Conference on Data Engineering (ICDE'04), 2004.

Yokoyama, S., Ohta, M., Katayama, K. and Ishikawa, H. (2005): *An access control method based on the prefix labeling scheme for xml repositories.* In 16th Australasian Database Conference. Vol 39, Jan 31 - Feb 3, Newcastle, Australia. Pages 105--113, 2005.

Yoshikawa, M. and Amagasa, T. (2001): *XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases,* ACM 2001.

Yu, T., Srivastava, D., Lakshmanan, L. and Jagadish, H. (2002): *Compressed accessibility map: Efficient access control for xml*. In Proc of 28th VLDB Conference, Hong Kong, 2002.

Yu, X. J., Luo, D., Meng, X. and Lu, H. (2005): *Dynamically Updating XML Data: Numbering Scheme Revisited*. In World Wide Web: Internet and Web Information System, Vol 8, No 1, 2005.