EFFICIENT PROCESSING OF XML DOCUMENTS

WANG WENQIANG

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgement

First of all, I would like to express my deepest gratitude to my supervisor, Professor Ooi Beng Chin, for his continuous guidance from the days since I was an undergraduate student. If not for his continuous encouragement and help, I would not even have the chance to pursue this Ph.D degree. I thank him for his kind sharing of knowledge and experience, not only in the academic area, but also in work and life.

I would like to thank Dr.Barbara Catania, co-author of most of my research work during my Ph.D candidature. It is my great pleasure to be able to work with her, and I thank her for her hospitality during my visit to the University of Genova.

I am very grateful to Dr. Lee Mong Li, who guided my first research work in the area of XML document processing. I would also like to thank Professor Elisa Bertino and Dr. Wang Xiaoling for their valuable suggestions on my research work.

I thank my classmates and friends in the Database Lab. It is really good to get to know all of them.

Last but not least, I thank my parents for their love and support at the time when I need them most.

Contents

List of Figures

List of Tables

Summary

In this thesis, we advocate storing XML documents in a relational DBMS, and address the related challenges. In particular, we set out to address the issues of mapping, indexing and updating XML documents.

The first challenge is how to store XML documents. We propose XStorM, a mapping scheme that maps XML documents to a relational DBMS. Our experiments demonstrate that XStorM gives good query performance, uses minimal space requirement and is scalable.

The second challenge is how to handle branching path (*twig*) queries efficiently. Inspired by the *join index* proposed in the relational context, we propose *XJoin Index*, a simple yet efficient indexing approach to shrink twigs before applying structural join algorithms. Our experiments show that the *XJoin Index* efficiently reduces the number of structural joins, thus improving overall query performance.

The third challenge is how to handle XML updates efficiently. XML updates can be modeled as inserting/removing small XML *segments* into/from an existing XML database. On this premise, we propose a new *lazy* approach to handle XML updates. This approach avoids relabeling existing elements after updates. Our experiments show that the lazy approach is much more efficient in handling updates than using immutable labeling; at the same time, it improves the performance of the structural join algorithm by taking advantage of segments.

Chapter 1

Introduction

The eXtensible Markup Language, XML[4], was initiated by the World Wide Web Consortium (W3C) as a simplified form and subset of the Standard Generalized Markup Language (SGML)[7]. The key features of XML include the ability for information providers to define new tags and attribute names at will, the nesting of document structures to any level of complexity, and the provision of Document Type Declaration (DTD)[8] and XML Schema[11] for constraining the structure and data values of a class of XML documents.

XML has reduced a fair amount of redundant features of SGML, making it much easier to manage and process than SGML. Another great advantage of XML over SGML is that XML is free from any intellectual property restriction while SGML products are proprietary. Compared with its closest sibling, HTML[5], XML is much more powerful in terms of extensibility. XML is in fact not a markup language, as its name suggests, but a metamarkup language. It is malleable, allowing different users to create their own markup-languages based on it. In contrast, HTML is quite limited. It only understands a set of predefined tags which are mainly used to format web pages. Because of all these advantages, although XML was originally designed to meet the challenge of large electronic publishing, it is also rapidly becoming a standard for data representation and exchange over the Internet and in various database applications.

XML is fundamentally different from relational and object-oriented data. The key distinction is that XML is not rigidly structured. In relational and object-oriented models, every data instance has a schema, which is separated from and independent of the data. In XML, the schema exists with the data. Thus, XML data is self-describing. Although W3C has developed DTD and XML Schema along with XML, they are mainly used to validate or to create XML documents. Both are not essential to understanding the contents of the documents. Because XML is self-describing, it can naturally model irregularities that cannot be modeled by the relational or object-oriented data model. For example, data items may have missing elements or multiple occurrences of the same element; elements may have atomic values in some data items and structured values in others; also, collections of elements can have heterogeneous structures.

Figure 1.1 shows an XML instance extracted from DBLP XML record [1]. It clearly shows the irregularity of the XML data model. For example, the first `inproceedings` element has three `author` child elements while the second `inproceedings` has only one. The second `inproceedings` has `cite` child elements, but the first `inproceedings` has none. We can see that the order of `inproceedings`'s child elements is not constant either.

XML data is normally generated as plain text files, and it is necessary to have a graphical representation of the data for efficient processing. Various models can be used for XML data, e.g.,the *Document Object Model (DOM)*[2] and the *Object exchange Model (OEM)*[77]. The work presented in this thesis is mainly based on the DOM model. Under the DOM model, the graphical representation of an XML

```
<dblp> ...
    <inproceedings mdate="2002-06-04"key="conf/ah/MunOW02">
        <author>Hyeonjeong Mun</author>
        <author>Sooho Ok</author>
        <author>Yongtae Woo</author>
        <title>An Automatic Rating Technique Based on XML Document.</title>
        <pages>424-427</pages>
        <year>2002</year>
        <crossref>conf/ah/2002</crossref>
        <booktitle>AH</booktitle>
        <ee>http://link.springer.de/...</ee>
        <url>db/conf/ah/ah2002.html#MunOW02</url>
    </inproceedings>

    <inproceedings mdate="2002-01-03" key="conf/er/Schuldt86">
        <author>Gary Schuldt</author>
        <title>ER-Based Access Modeling.</title>
        <pages>233-251</pages>
        <year>1986</year>
        <booktitle>ER</booktitle>
        <url>db/conf/er/er86.html#Schuldt86</url>
        <crossref>conf/er/86</crossref>
        <cdrom>er86/ER86-P233.pdf</cdrom>
        <ee>db/conf/er/Schuldt86.html</ee>
        <cite label="deMarco">...</cite>
        <cite label="Flavin">...</cite>
        ...
    </inproceedings>
</dblp>
```

Figure 1.1: Example of an XML document

document is usually in form of a tree structure. Figure 1.2 shows a partial graphical representation of the XML document in Figure 1.1. Each ellipse node represents an XML element. The text inside the ellipse is the type/class of the element, and the text below it is the value of the element. For simplicity, attributes of an element are sometimes treated as child elements. Also, it is common in practice that every node is assigned a unique label (identifier), usually based on the position of the element it represents within the XML document, for the convenience of document processing.

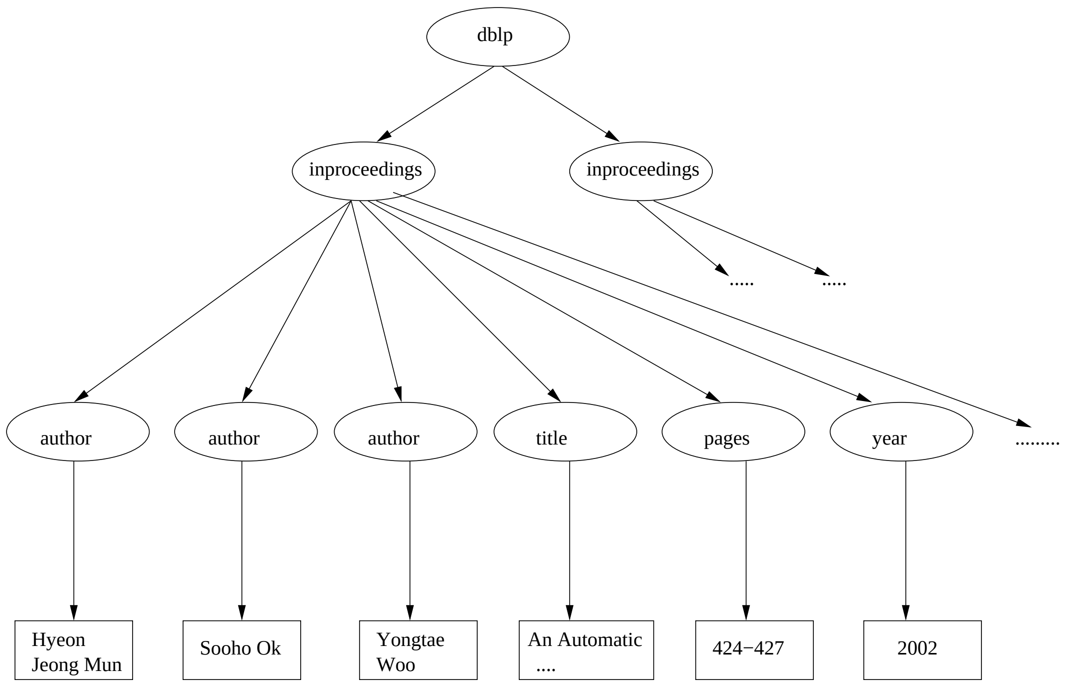


Figure 1.2: Partial graphical representation of the XML document in Figure 1.1

There are various possible ways to store and query XML data. [32, 37, 55, 88, 89, 92, 17, 87, 103, 28, 60, 74, 23, 102] propose mapping techniques to store XML data into a relational database. [18, 33, 36, 72, 45, 63, 69, 85, 86, 57] propose various ways to translate XML queries into SQL. The obvious advantage is that mature RDBMS technologies, e.g.,indexes, concurrent control and transaction management, can be exploited without much change. However, it is also a challenge to accommodate the irregular structure of an XML document into a rigid relational table.

[89, 35, 59, 84, 3, 58] implement an XML database on top of an object-oriented database. Object-oriented databases have richer data modelling capabilities than RDBMS, which are useful for clustering XML elements and attributes. However,the current generation of object-oriented database systems is not fully developed to process complex queries on large databases. Besides, the object-oriented data model is essentially a fixed schema model and it also suffers from the extreme irregularity of XML data as the relational model does.

The research community has shown increasing interest in building native XML databases in recent years. The word “native” here means that XML data is stored directly, preserving its original tree-like structure. Though some components of traditional DBMS, e.g., transaction management functions, can be applied to native XML database with no change, most other components need to be modified to accommodate the new data model and query language. Theoretically, native XML databases should perform much better than simply mapping the XML model to traditional DBMSs, as it is specially designed for XML. However, it takes time for native XML databases to be mature enough to compete with traditional DBMSs. Some early research projects [70, 71, 80] built native XML databases on top of semi-structured databases. Natix[50, 49] has been developed as a storage manager for XML data and its main focus is on efficient physical page management of tree-structured data. Timber[44] is so far the most comprehensive attempt of building practical native XML databases. The Timber system is based on a bulk algebra for manipulating trees, and stores XML directly. It has also developed new access methods, a cost estimation mechanism, and query optimization techniques for query processing.

1.1 Motivation

Mapping XML data into relational database remains the main trend of storing XML data so far, as relational stores are effective in providing multiple distinct logical views on the same data with very good scaling and transactional characteristics.

There are many ways to map XML data into relational tables. Among them, Oracle 8*i* lets the user or system administrator decide how XML elements are stored in relational tables. [88] infers from the DTDs of the XML document how the XML elements should be mapped into tables. *STORED* [32] analyzes the XML data and expected query workload to obtain a set of relational schemas. Any data that cannot be accommodated in these schemas are stored in overflow graphs. This involves integration of the relational storage with a semistructured overflow, raising yet to be resolved system issues. Furthermore, if the data instance has a very irregular structure, then the schema extracted may not cover a large percentage of the data and a lot of overflow graphs will be generated, leading to performance degradation. [37] takes the graph representation of an XML document and studies various schemes to map the edges and nodes into relational tables. Among them, the *binary* approach gives the best experimental performance. The *binary* approach creates a relational table for each XML tag and stores the value accordingly, which is similar to the binary storage scheme proposed for storing semistructured data in [95]. There are as many binary tables created as there are different subelement and attribute names in an XML document. The values of the attributes can be stored together (inlined) in the same table. Unfortunately, the number of join operations needed to answer a query is proportional to the number of attributes involved, which becomes very costly when reconstructing large XML documents.

We note that XML elements that present entities in the real world (objects) are differentiated from XML elements that represent properties of entities (attributes). If we can capture the general structure of an *object* in the XML data, we will be able to generate a relational table to store the *object* together with the majority of its attributes. The motivation of our work on XML storage is therefore to develop a new scheme that maps XML data into relational tables based on this finding.

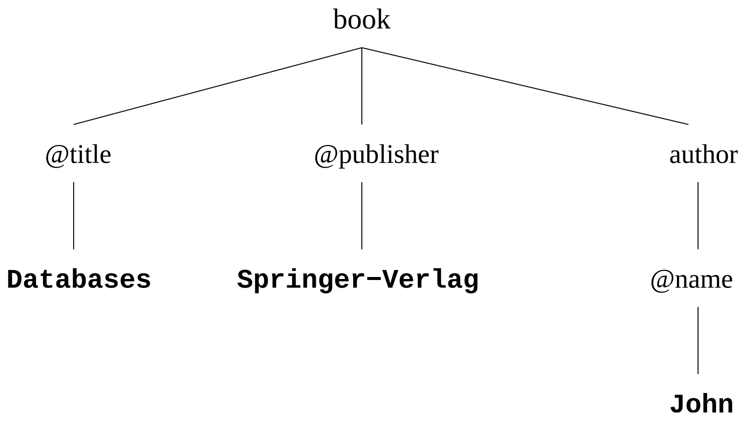


Figure 1.3: Graphical Representation of a Branching Path Expression

The new scheme should overcome the drawbacks of the existing mapping schemes, i.e., with the scheme, no excessive fragmentation is generated and data integrity is guaranteed.

Regardless of whether the XML database is built on top of an existing database management system or built specially for XML data (the native XML database), query evaluation is one of the most important aspects of XML processing. W3C has developed XQuery[10] as a standard to solve queries on XML data while in most cases, the core operation of solving an XML query is to solve the XPath[9] expression within the query. A typical query for XML documents specifies selection predicates for multiple elements, related by some tree structured relations. For example, the query: `book[@title = ‘Databases’ AND @publisher = ‘Springer Verlag’]/author[@name = ‘john’]` matches `author` elements whose name is `john` and that are children of `book` elements, whose title is `Databases` and whose publisher is `Springer Verlag`. Figure 1.3 shows the graphical representation of the above query. Expressions such as this are known as *branching path* expressions because their graphical representations contain branch(es) and correspond to a small tree (a *twig*).

To solve a branching path expression, two main approaches can be applied. Under the first approach, the tree-based representation of the whole XML dataset is scanned; thus a naive tree traversal strategy is used. Summary indexes can be used for this purpose[81, 26, 39, 73, 29, 38, 53, 75]. The main limitation of this approach is that, when the expression contains '//', i.e., requires the evaluation of an ancestor-descendant relationship, it may require the whole dataset to be scanned even if there are only a few matches. Under the second approach, elements matching each single node are first determined. Then, the sets obtained are joined with the use of *structural join* algorithms. Such algorithms take advantage of specific labeling schemes (for every element/attribute in the document) to efficiently check ancestor-descendant and parent-child relationships among elements/attributes. One relationship for each twig edge has to be evaluated and the results have to be merged.

A number of structural join algorithms have been proposed. The results from these algorithms are typically pairs of element/attribute labels, which are later used to evaluate other path query expressions. A more recent attempt tries to reduce the size of intermediate results by using holistic algorithms [48]. Most of these algorithms rely on the usage of some indexing techniques to more efficiently perform the join operation and they can be used to either reduce the number of elements before the structural join algorithm is applied[66], or during the application of the algorithm itself, to skip descendants[25] or descendants and ancestors[47] without matches. These indexing techniques do not vary the number of joins to be executed for branching path expressions. Rather, they provide support for efficient join processing.

The motivation of our work on efficient XML query processing is therefore to find a way to reduce the number of structural joins required to solve a branching path query. This is as important as making each individual structural join more efficient.

Updating XML documents is also a major challenge in the area of XML document processing. As we have mentioned, every element/attribute in an XML document is normally assigned a unique label based on its location in the XML document to facilitate query processing, particularly structural join. The correctness of the structural join algorithm completely depends on these labels. However, this identifier does cause problems when updates take place. The problem is that after the original XML document has been updated, i.e., new elements have been inserted or existing elements have been removed, we may need to update the labels of possibly a large number of elements in order to maintain the correct relationship between elements, which is the foundation of the structural join algorithms. This relabeling process could make the update operation very inefficient. Figure 1.4 illustrates the relabeling scenario. In this figure, each node represents an XML element and we use *(start_position:end_positon)* pairs to uniquely identify them, i.e., labeling each node with a *(start_position:end_positon)* pair. When a new element (the black node) is inserted between nodes 3 and 4, the start positions of 4, 5 and 6 need to be updated and the end positions of 1, 2, 4, 5 and 6 need to be updated as well. So only node 3's label remains unchanged. In general, the I/O cost of update is $O(N)$, where N is the total number of elements in the XML document.

Previous attempts to solve this problem basically rely on various labeling schemes [66, 27, 76, 92, 101, 90, 61, 62]. [66] is an extended interval-based scheme where additional space is reserved for future insertions. This scheme fails if the space required to hold the inserted nodes exceeds the reserved space. Prefix labeling [27, 76, 92, 61, 62] allows each node to inherit its parent's label as the prefix of

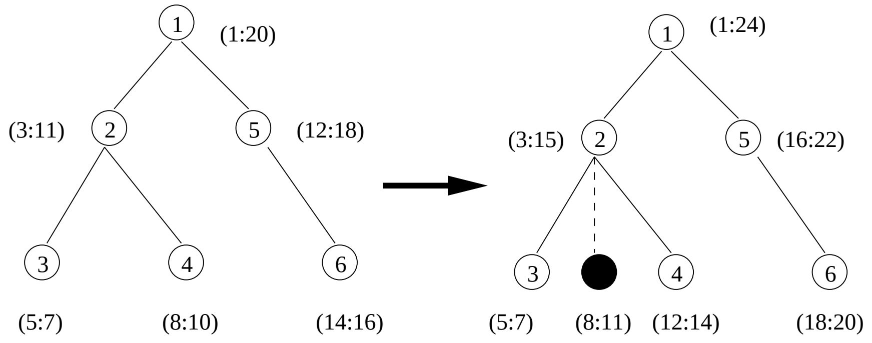


Figure 1.4: Relabeling Caused by Update

its own label so that inserting new nodes does not affect the labels of the existing nodes, i.e., labels are immutable. Unfortunately, the results presented in [27] establish that any immutable labeling scheme requires $\Omega(N)$ bits per label, where N is the size of the document, thus incurring high storage overhead. Moreover, structural join algorithms using a prefix labeling scheme are less efficient than those using an interval-based labeling scheme because determining the ancestor-descendant relationship between two elements using prefix comparison is slower than using simple integer comparison. The prime number labeling scheme [101] overcomes some problems of prefix-labeling by assigning to each node a product of prime numbers as its label and the containment relationship of two elements can be determined by the properties of prime numbers. The order of each element is preserved by maintaining a table of simultaneous congruences of element label sets and element order sets. Heavy computation is required for the insertion of a new element since computing simultaneous congruences is costly. Most recently, a different approach to cope with updates while guaranteeing good query performance was proposed in [90]. In this approach, a dynamic, thus mutable, labeling scheme is used together with specific data structures that provide a good trade-off between query and update costs.

We observe that, in real world scenarios, XML document updates tend to be done in batch manner, i.e., multiple XML elements are inserted (or removed) together. As an example, consider the DBLP XML database. It contains many articles, books and proceedings, and new items need to be added into the DBLP database almost every day. Due to the high frequency of update operations, updating the database after each single request of element insertion/deletion is not a feasible solution. Another example is represented by an on-line registration system. In such a system, once a user submits a registration form, an automatically generated XML document containing information about the user's identification, name, occupation, etc., is inserted into the system. In this case, multiple XML elements are inserted instead of a single element. In both examples, instead of inserting/deleting each element when requested, it seems more reasonable to generate XML *segments* corresponding to a set of elements that must be inserted (deleted) into (from) the whole database together and then update the database once for each segment.

The motivation of our work is therefore to develop a new scheme for XML updates based on the batch update nature of XML documents. This new scheme should solve the update problem mentioned above and in the mean time, it must not affect the efficiency of query processing, i.e., this new scheme must not incur any significant processing overhead for structural join algorithms.

1.2 Contributions

For this thesis, we have designed an architecture for storing and processing XML documents in its natural form. Our work is built around this architecture, and in particular, we address three important problems. Figure 1.5 shows a general structure of our work. In the next three subsections, we summarize our contributions.

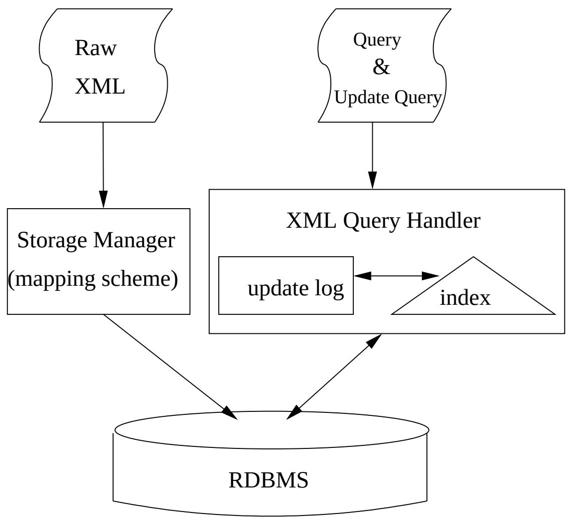


Figure 1.5: General Structure

1.2.1 XStorM Mapping Scheme

To overcome the drawbacks of existing mapping schemes that map XML data into relational tables, we propose a new scheme, XStorM, that has the following features:

- XStorM discovers frequent patterns in an XML dataset by exploiting data-mining algorithms. Based on the frequent patterns discovered, it identifies real-world objects in the XML dataset. These objects are then stored in a *core* relational table together with the majority of its attributes to avoid excessive fragmentation.

- XStorM stores data that deviates from the core schemas in separate *overflow* tables.

- XStorM embeds structural information of an XML document in the names of overflow tables and some attribute names for the fast reconstruction of the original XML document.

- XStorM guarantees data integrity as entire XML data instances are stored in the relational database.

We present the procedure to generate XStorM mapping and we compare it with other mapping schemes on both space occupancy and query performance. Our experimental results show that XStorM yields good query performance and consumes the least storage. More importantly, it is scalable.

1.2.2 XJoin Index

In the relational context, *Join index*[94] has been proposed for efficient join processing. It basically pre-joins some relations and the actual joins benefit from the results of these pre-computed joins. This gives us the inspiration for building a "join index" to solve branching path expressions in the XML context. Our aim is to reduce the number of joins to be executed, and we propose a simple yet efficient join indexing approach to shrink the twig before applying any structural join algorithm. The index technique, which we call XJoin Index, pre-computes some (semi-)join results, thus reducing the number of joins to be computed. Precomputed (semi-)joins correspond to both content and structure information and they support the following operations: (i) attribute selections, possibly involving several attributes; (ii) detection of parent-child relationships; and (iii) counting selections, such as "*Find all books with at least 3 authors*".

The main features of the proposed technique can be summarized as follows:

- *Simplicity*: Unlike other approaches, based on specialized data structures, the XJoin Index is entirely based on B^+-trees [82], constructed over specific tuples of values.

- *Flexibility*: The XJoin Index can be coupled with any structural join algorithms that have been proposed so far. Moreover, given a branching path expression, several execution plans can be defined by the query processor, based on the usage of the XJoin Index.

In this thesis, we first present the XJoin Index and we show that, even with the duplication of some element information in the index, the space required is linear to the number of elements and attributes appearing in the XML dataset. We then present search and update algorithms for the XJoin Index. Our approach differs from other approaches in that any additional attribute or counting predicate inside a query condition does not correspond to a new application of a structural join algorithm. Rather it corresponds to a simple set intersection. Next, we show which query execution plans a query processor can define, based on the usage of the XJoin Index. Such plans differ in the number of joins to be executed to solve the original branching query. Experimental results are presented for the search and update operations of the XJoin Index. These results show that the XJoin Index can process twig queries by up to an order of magnitude faster than traditional index approaches.

1.2.3 Lazy update scheme

Based on the observation that updates tend to be done in batch manner and in the form of segments, we present a new approach to dealing with XML updates in this thesis. We call it *lazy* since segments are used to avoid computations during both updating and querying.

First, we model the whole XML database as a single *super document* by simply adding a dummy root to all the existing XML documents; thus update operations correspond to inserting (or removing) XML segments into (or from) the super document. In this model, each element has two positions. The first is its *local position* with respect to the XML segment it belongs to. The second is its *global position* in the super document. The local position label never changes once it is assigned to an element, but it is not unique. On the other hand, the global position label is unique but it changes if an update occurs. From these considerations, it naturally follows that if a local position label is used as the key (or part of the key) in the element index, we can avoid updating the existing element labels after an update. However, since local labels are not unique, they cannot be directly used in structural join.

The key point here is that the number of inserted (or removed) segments is likely to be significantly less than the number of XML elements these segments contain. For example, an XML document corresponding to a registration form may contain 20 to 30 XML elements. This gives us the inspiration to build an in-memory *update log* to record the information of every segment. The update log must satisfy the following requirements:

- The *update log* must maintain sufficient information to support structural join between segments
- The *update log* must allow us to identify the structural information of the segments given only the global start positions and lengths of the segments. These two values are likely to be the only information we know when a segment is inserted (removed) in real scenarios.
- The *update log* can be integrated easily into existing structural join algorithms. However, segment-aware query processing techniques can be defined to reduce query processing costs.

In this thesis, we also present update algorithms for the proposed update log, assuming that each operation takes as input the position in the super document where the segment has to be inserted/deleted and the length of the segment. We further present a structural join algorithm that works with our *lazy* approach. The algorithm is developed based on the stack-based structural join algorithm proposed in [12] to deal with segments. The results of our experimental study on the update and structural join operations show that the lazy approach is significantly more efficient than existing labeling approaches for updates; additionally, it improves query processing performance.

1.3 Organization

The remainder of this thesis is organized as follows:

- Chapter 2 presents the research works that are closely related to this thesis.
- Chapter 3 introduces the XStorM mapping scheme that maps XML data into relational tables.
- Chapter 4 introduces XJoin Index for efficient branching path query processing.
- Chapter 5 presents our *lazy* approach to handle XML updates.
- We conclude our work in Chapter 6 with a summary of our contributions. We also discuss some limitations and directions for future work.

We acknowledge that the work in Chapter 3 is published in [99, 100] and is a continuation of eailier works as on the honors year project , the work in Chapter 4 is published in [16], and that in Chapter 5 is published in [20].

Chapter 2

Related Work

In this chapter, we review research work closely related to this thesis. Various mapping schemes that map XML document into relational database will be reviewed in Section 2.1. Background information of XML element labeling schemes, which is considered as one of the foundations of XML document processing, is presented in Section 2.2. In Section 2.3, we present an overview of XML query processing, introduce *structural join*, which is considered as the core operation of solving path expressions, and review several indexing techniques designed for facilitating structural join. We will also review several proposals to solve XML query that are independent of structural join algorithms. In Section 2.4, we will show the state of the art on the topic of XML update. We summarize in the last section.

2.1 Relational Mapping Scheme

Ever since the launch of XML, the database research community has been working on the efficient and effective storage of XML documents. Among all the approaches proposed so far, mapping XML data into existing relational databases receives most attention. A number of mapping schemes have been proposed in recent years.

[17, 74, 88] focus on how to define a "good" relational schema from given XML schemas. [22] proposed XFDs, which is a constraint definition to capture structural and semantic information of XML documents, and an mapping scheme called RRXS based on an algorithm that computes the reduced set of given XFDs. [92] proposed several order encoding methods so that ordered XML processing can be supported by relational databases. [37, 103] proposed fixed relational schema for storing the XML data and algorithms were also presented for query translation. STORED [32] was proposed to generate relational schema which is decided based on the XML data itself. In this section, we will take a closer look at STORED, the mapping schemes proposed in [37] and XRel[103] as they are compared with our proposed mapping scheme in our experiments.

STORED is in fact a declarative query language used to express a mapping scheme that maps semistructured data, e.g., XML, to relational schemas. It relies on a data mining algorithm proposed in [97] to identify frequent patterns in the data instance and generate relational schemas bases on these patterns. The process of generating relational schemas involves the following steps:

1. **Computing minimal path prefixes**. In this step, all prefixes [1] l_1, l_2,...,1_k with support greater than or equal to a minimal support are generated. These prefixes identify the collection of objects that become the root objects for the data mining algorithm in the next step.

2. **Data mining**. The data mining algorithm is applied in this step to identify all frequent K patterns, where K stands for the number of leaf nodes in the tree-like pattern. Also in this step, all paths with high support are identified

[1]The prefix of a node is simply a chain of its ancestor nodes starts from root node and ends at its parent node.

and retained.

3. **Selecting $K0$ patterns**. In this step, by using a greedy algorithm, the frequent K patterns are checked to find those $K0$ patterns that best cover the high support paths identified in the previous step.

4. **Selecting required attributes**. In this step, the sub-patterns of each $K0$ pattern selected in the last step are checked to identify which attributes are to be included in the final schemas

5. **Generating STORED queries**. This is a straightforward step to generate relational schemas based on the results from previous steps.

Data that cannot fit the identified relational schemas is stored in external $Overflow$ graphs, thus ensuring that the STORED mapping is lossless. But if the XML data instance has a very irregular structure, the schemas extracted may not be able to cover a large percentage of the data. Hence a lot of overflow data structures will be generated, leading to performance degradation.

[37] takes a graphical representation, i.e., OEM [77], of XML document. In this model, each outgoing edge models an attribute of the object. Edges are labeled with attribute names and each object has a unique identifier. The leaves of this model are labeled with data value (e.g., integers, strings, etc.). Schemes of mapping both attributes and values are proposed and compared with their performances over the same data instance.

The simplest scheme for mapping attributes is to store all attributes in a single

Edge table, which has the following structure:

$$Edge(source, ordinal^2, name, flag, target)$$

The key of *Edge* table is $\{source, ordinal\}$. An index on *source* column and a combined index on the $\{name, target\}$ columns can be established for forward and backward traversals, respectively.

The second scheme for mapping attributes is to group all attributes of the same name into one *Attribute* table, which actually corresponds to a horizontal partitioning of the *Edge* table in the previous scheme. There are as many *Attribute* table as different attribute names in the data instance and each *Attribute* table has the following structure:

$$A_{name}(source, ordinal, flag, target)$$

The key of *Attribute* table is $\{source, ordinal\}$. An index on the *source* column and an index on the *target* column can be built for forward and backward traversal, respectively.

The third scheme is to generate a single *Universal* table to store all the attributes, which corresponds to the result of an outer join of all *Attribute* tables. The structure of *Universal* table is as follows, suppose n_1,...,n_k are the attribute names in the XML instance,

$$Universal(source, ordinal_{n_1}, flag_{n_1}, target_{n_1}, ..., ordinal_{n_k}, flag_{n_k}, target_{n_k})$$

It is obvious to see that the *Universal* table is not normalized. Therefore, a

[2]the ordinal of an attribute is simply its sequence number among all attributes of its parent

normalized $Universal$ approach, $UnivNorm$, was proposed by storing multi-valued attributes in separate overflow tables. The structure of the $UnivNorm$ table and the $Overflow$ tables is as follows, suppose n_1,...,n_k are the attribute names in the XML instance,

$$Universal(source, ordinal_{n_1}, flag_{n_1}, target_{n_1}, ..., ordinal_{n_k}, flag_{n_k}, target_{n_k})$$

$$Overflow_{n_1,...,n_k}(source, ordinal, flag, target)$$

The key of the $UnivNorm$ table is *source* and the key of an $Overflow$ table is $\{source, ordinal\}$. The $flag$ is set to “m” if the attribute is multi-valued. Index can be built on the *source* and the *target* column(s) for both $UnivNorm$ and $Overflow$ tables.

There are two possible ways to store values in the leaves of an XML document tree:

1. Storing values in separate $Value$ tables with the structure of the following form:

   $$V_{type}(vid, value)$$

   The *vids* of the $Value$ tables depend on the implementation of the mapping schemes. Index on both *vid* and *value* column can be built on the $Value$ table.

2. Storing values and attributes in the same table. The table corresponds to an outer join of the $Edge(Attribute, Universal, UnivNorm, Overflow)$ table and the $Value$ tables. This approach is also known as *inlining* as one column is needed for each data type.

Since there are four approaches for storing attributes and two approaches for storing values, there are totally eight different mapping schemes. According to the results of the experiments conducted in [37], among all the eight schemes, storing attributes in separate *Attribute* table and values inline, which is also known as *Binary* approach, yields the best performance. One obvious disadvantage of the *Binary* approach is that the number of join operations needs to answer a query is proportional to the number of attributes involved. This becomes very expensive when answering complicated path queries or reconstructing large XML documents.

XRel[103] also defines fixed schemas to store XML document in relational database. Compared with Binary scheme, XRel is more efficient to solve path queries involving "//" or with long lengths because it embeds path information in the tables, thus string comparison can be used to reduce the number of joins to be performed. The basic structure of XRel scheme consists of four relational schemas, as shown below:

Element(docID, pathID, start, end, index, reindex)

Attribute(docID, pathID, start, end, value)

Text(docID, pathID, start, end, value)

Path(pathID, pathexp)

In the above relational schemas, the database attributes $docID$, $pathID$, *start*, *end* and *value* represent document identifier, simple path expression identifier, start position of a region, end position of a region, and string value, respectively. *index* and *reindex* in the relation *Element* represent the occurrence order of an element node among the sibling element nodes in document order and reverse document order, respectively. *pathexp* in the relation *Path* stores simple path expressions.

XRel stores elements and values in separate tables and stores all elements in one big Element table. Therefore, if the path query does not contain "//" or if the element/value tables are too big, binary scheme may outperform it as joining small tables is likely to take less time than joining large ones.

2.2 Labeling Schemes

Labeling schemes play very important role in XML document processing. The main purpose of labeling XML elements is to allow fast identification of relationships between elements, particularly the ancestor-descendant relationship, which is in fact the core operation of any structural join algorithm. Also, most research work on the topic of XML update focus on developing dynamic labeling schemes that cope with updates. In this section, we will first introduce some classical labeling schemes, including Dietz's labeling scheme [34] and some of its variations, and the well-studied prefix based labeling schemes[92, 27, 61, 62]. Several newly proposed labeling schemes[54, 76, 93, 101, 13, 90, 98, 24, 67], especially the prime number labeling scheme[101], which we used to compare with our approach in Chapter 5, will be subsequently presented.

2.2.1 Dietz's Scheme

Dietz's scheme [34] is the first labeling scheme used to determine the ancestor-descendant relationship between any pair of tree nodes by tree traversal order. The proposition given in the paper is as following:

Proposition 2.1 *For two given nodes x and u of a tree T, x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal.*

For example, consider the left tree in Figure 2.1 where the nodes are labeled by Dietz's labeling scheme. Each node is labelled with a pair of preorder and

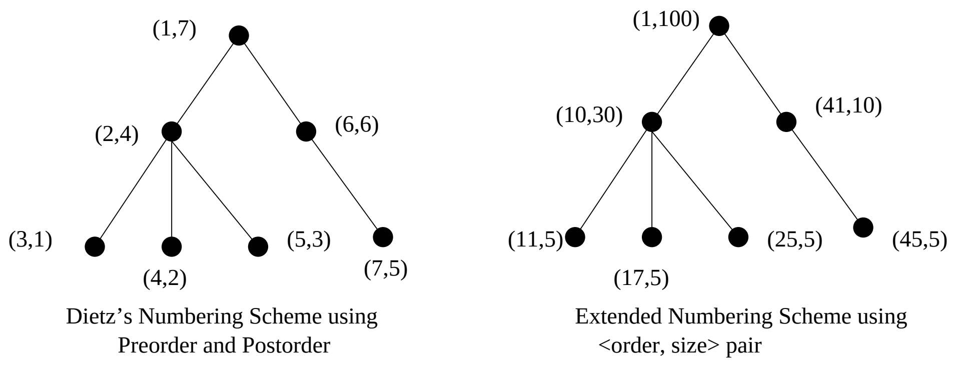

(1,7)
(2,4)
(6,6)
(3,1)
(4,2)
(5,3)
(7,5)
Dietz's Numbering Scheme using Preorder and Postorder
(1,100)
(10,30)
(41,10)
(11,5)
(17,5)
(25,5)
(45,5)
Extended Numbering Scheme using <order, size> pair

Figure 2.1: Numbering Scheme Examples

postorder numbers. In the tree, we can see that node (1,7) is an ancestor of node (4,2), because node (1,7) comes before node (4,2) in the preorder traversal(i.e., $1 < 4$) and after node (4,2) in the postorder traversal(i.e., $7 > 2$). The original Dietz's labeling scheme allows constant time identification of ancestor-descendant relationship between two nodes. However, if a new node is inserted into the tree, the preorders and postorders of many nodes may need to be recomputed.

To overcome this shortcoming, [66] proposed an extended preorder labeling scheme based on original Dietz's labeling scheme by reserving extra space for future insertions. The scheme associates each node with a pair of numbers $<$*order, size*$>$ as follows.

- For a tree node y and its parent x, *order(x)* $<$ *order(y)* and *order(y)* $+$ *size(y)* $\leq$ *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 preorder traversal, *order(x)* $+$ *size(x)* $<$ *order(y)*.

Then, 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 sum of *sizes* of all current descendants of x, which allows to accommodate future insertions gracefully.

This extended labeling scheme also allows constant time identification of ancestor-descends relationship as the original Dietz's scheme does. The lemma given is:

Lemma 2.1 *For two given nodes x and y of a tree T, x is an ancestor of y if and only if order(x)* $<$ *order(y)* $\leq$ *order(x) + size(x).*

For example, consider the right tree in Figure 2.1, a node (25,5) is contained in both (10, 30) and (1,100). Hence, the node with order 25 is a descendant of nodes with order 10 and 1. The extended Dietz's labeling scheme is obviously more flexible than the original Dietz's scheme because it can deal with dynamic updates as long as there is pre-reserved space available. But it is also obvious that this scheme does not solve the dynamic insertion problem completely simply because that it will fail if there is no pre-reserved space available.

A similar labeling scheme uses *(start, end)* pairs as the labels of elements, like the one shown in Figure 1.4. This variation is more widely applied in practice [104, 44, 47, 42, 48, 31], as only one pass of the document is required to generate these labels. The *start(end)* here refers to the *starting (ending)* position of the XML element in the whole XML document, in terms of bytes or words. An element x is an ancestor of an element y if and only if x has a smaller starting position than y AND a larger ending position than y. This scheme can determine ancestor-descendant relationship between elements in constant time as the original Dietz's scheme does. However, it faces the same problem when new elements are inserted.

2.2.2 Prefix Labeling Schemes

Dewey labeling scheme[92] is based on Dewey Decimal Classification developed for general knowledge classification. With Dewey label, each node in a tree is assigned a vector that represents the path from the root to the node. Each component of the path represents the local order of an ancestor node, as illustrated in Figure 2.2. Dewey label is “lossless” because each path uniquely identifies the absolute position of the node within the document. The ancestor-descendant relationship between two nodes can be identified by prefix comparison. For example, node with label “*1.1.2.4.3*” must be a descendant of node with label “*1.1.2*”, but cannot be a descendant of node with label “*1.1.3*”. In case of insertion, only the right siblings (of the inserted node) and their descendants may need to be relabeled.

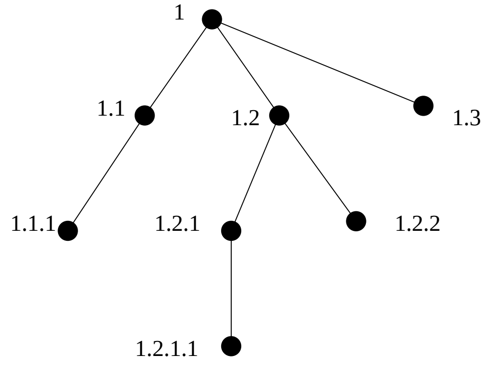


Figure 2.2: Dewey Order Example

Binary prefix labeling scheme has been thoroughly discussed in [27], both in the static case, where the full document is given in advance and in the dynamic case, where no information about the document is known beforehand.

Static prefix schemes typically work as follows. The outgoing edges of each node are assigned a set of prefix-free binary strings (a set of strings is *prefix-free* if no string in the set is a prefix of another), and then, starting from the root and going down, the label of each node is defined to be the concatenation of its parent label and the string assigned to the edge leading to the node. For example, consider a node v with three children v_1, v_2, v_3. Strings “0”, “10”. and “11” can be assigned to the three edges (v, v_1), (v, v_2), and (v, v_3), respectively. So the labels of v1, v2, and v3 are $L(v_1) = L(v) \cdot 0$, $L(v_2) = L(v) \cdot 10$, and $L(v_3) = L(v) \cdot 11$. The above scheme is similar to the Dewey labeling scheme, except it uses binary strings to code prefixes. Therefore, this scheme encounters similar problem in a dynamic setting as well. For example, if a new child v_4 is added to v,there is no string that can be attached to new edge(v,v_4). This is because any string would have one of the strings 0, 10, and 11 as a prefix.

A more flexible prefix scheme for the dynamic situation works as follows. The root of the tree is labeled with an empty string. The first child of the root is labeled with "0", the second child with "10", the third with "110"(rather than the "11" in the static labeling example), the forth with "1110", etc. Similarly for any node v the first child of v is labeled with $L(v) \cdot$ "0", the second child of v is labelled with $L(v) \cdot$ "10", the third with $L(v) \cdot$ "110", and the i^{th} child with $L(v) \cdot$ "$111^{i-1}0$". It is easy to see that for all pairs of nodes v, u, $L(v)$ is a prefix of $L(u)$ if and only if v is an ancestor of u. Also, by induction it is easy to prove that the length of the maximum label is at most i-1 after inserting i nodes including the root. So for any n-node tree the maximum label length is at most n-1 without any need to know n in advance.

The following theorem[27] shows that no labeling scheme (regardless if it is prefix based, range based, or uses any other labeling type) can achieve better bound on the labels length.

Theorem 2.1 *For every deterministic labeling scheme S = <p, L> there is an insertion sequence of length n such that S assigns a label of length at least n - 1 for some node in the sequence.*

The above theorem assumes no restrictions on the tree structure. It works for the case that a node can have arbitrary number of children. But for XML documents, the DTD and XML Schema may restrict the number of children, e,g. the total number of children is bounded by some constant Δ. In this case, following theorem[27] gives a slightly weaker lower bound of the length of label.

Theorem 2.2 *For every deterministic labeling scheme S and every constant Δ, there is an n-node insertion sequence constructing a tree of maximum degree Δ on which S assigns a label of length at least $n\log_2(1/\alpha)$ - O(1), where α is a root of $x + x^2 + ... + x^\Delta = 1$.*

The above theorem shows that even if for binary trees ($\Delta = 2$), any deterministic labeling scheme will have some label of size Ω(n), or, more precisely, of size at least $0.69n$ - O(1) ($\alpha = 0.618$, for $\Delta = 2$).

In practice, XML documents tend to have a relatively low depth, i.e., the trees are balanced with relatively high degrees. A more suitable labeling scheme can be developed for such trees. The children of a node v have label of v concatenated with the string attached to their incoming edge, similar as previous scheme. The string $s(i)$ for the i^th child is defined such that

$$s(1), s(2), s(3), ... = 0, 10, 1100, 1101, 1110, 11110000, ...$$

Namely, to obtain $s(i{+}1)$ the binary number represented by $s(i)$ is increased by 1 and if the representation of $s(i) + 1$ consists of all ones, the length of the label is doubled by adding sequence of zeros.

The heuristics of this scheme is that a node with more children is more likely to have additional children when update takes place. So rather than allocating for the new child the shortest possible available prefix-free string (as done in the previous scheme), a longer one is given instead. The investment is likely to pay off as it will shorten the labels of forthcoming siblings. In the previous scheme, for each new child, the length of the assigned prefix free string grows by exactly one bit. In contrast, in this scheme, the length may grow by several bits at once. But then can stay the same for several future coming nodes (until it needs again to grow).

The length of labels of this scheme is given in the follow theorem[27].

Theorem 2.3 *The maximum length of a label using this scheme is at most $4d\log(\Delta)$, d being the maximal depth of the tree and Δ the maximum outdegree of a node.*

2.2.3 Recent Works on Labeling Scheme

The research community has proposed quite a number of labeling schemes in recent years. [54] expresses the coordinate of an XML element based on the region of its parent element and proposed an index structure based on this coordinate, which requires only a small portion of index file to be updated in case of updating. ORDPATH[76, 93] is a hierarchical labeling scheme, which uses a "careting-in" scheme to support dynamic insertions of element. Prime number labeling scheme [101] labels elements with prime numbers and allows ancestor-descendant relationship identification by properties of prime numbers. [13] uses floating numbers, instead of integer numbers, to label elements so that there is more space between labels, which favorites insertion. But the number of distinct values is still limited by the number of bits used in representation. The BOX method [90] proposed a dynamic labeling scheme, used together with specific data structures (W-BOX and B-BOX), to provide a good trade-off between query and update costs. W-BOX uses weight-balanced B-trees to reduce the relabeling overhead, obtaining a logarithmic amortized update costs and constant worst-case lookup cost, whereas B-BOX further reduces update costs, resulting in a constant amortized update time and logarithmic worst-case lookup cost, by avoiding the storage of labels, that can however be reconstructed starting from the proposed data structure, a variant of B-tree. [98] proposed a new labeling scheme called PBiTree coding, which also allows efficient identification of ancestor-descendant relationship between elements. New algorithm for solving containment query was subsequently proposed based on this

PBiTree coding, which does not require sorting/indexes. [24] presented a labeling scheme that combines P-labeling (for processing suffix path queries efficiently) with D-labeling (for processing queries involving the descendant axis). Most recently, [67] proposed an powerful *extended Dewey* labeling scheme such that from the label of an element alone, all element names along the path from the root to the element can be derived. This labeling schemes is used as the base of its holistic twig join algorithm[67].

The prime number labeling scheme[101] was proposed to overcome some problems of prefix-labeling schemes. This scheme determines ancestor-descendant relationship between elements by properties of prime numbers.

Property 2.1 *If an integer A has a prime factor which is not a prime factor of another integer B, then B is not divisible by A.*

In XML trees, if a node A has a descendant which is not a descendant of another node B, then A cannot be a descendant of node B. Therefore, if the leaf nodes in XML tree are labeled by prime numbers and the non-leaf nodes are labeled as a product of the labels of its child nodes, then the ancestor-descendant relationship can be easily determined by using the above property of prime numbers. But it is obvious that this bottom-up approach can quickly result in relatively large numbers being assigned to nodes at the top of XML tree.

An alternative of bottom-up approach is a top-down approach, as shown in Figure 2.3. The label of a node is divisible only by its ancestor's label. In the top-down scheme, each non-leaf node is given a unique prime number and the label of each node is the product of its parent node's label (parent-label) and its own label(self-label). In Figure 2.3, the "parent-label" is 2 for the node whose label is "10" while its "self-label" is 5. When a new node is inserted, it is easy to simply assign a prime number that has not been assigned before as the self-label for the newly inserted node. No re-labeling is required.

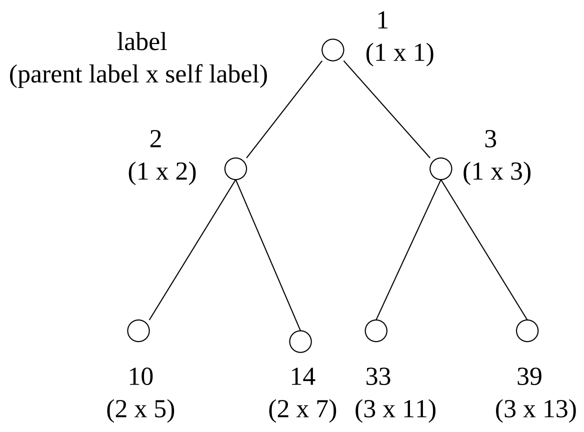


Figure 2.3: Top Down Prime Number Labeling Scheme

Several optimization techniques can be applied to overcome the shortcoming of the top-down labeling scheme, which is still the relatively large label size. These techniques include (1) reserving small prime number for upper level labels, (2) making use of the only even prime number “2” to label leaf nodes and (3) merging paths to reduce redundancy if ordering is not important.

A major advantage of prime number labeling scheme is its capability to preserve orders of elements when updates take place, which makes it capable to answer order-sensitive queries. The Chinese remainder theorem is applied to maintain order in the prime number labeling scheme.

Definition 2.1 ***Chinese Remainder Theorem****: Let $M = [m_1, m_2, ..., m_k]$ and $N = [n_1, n_2, ..., n_k]$ be two lists of integers. If the GCD$(m_1, m_2, ..., m_k) = 1$, then the simultaneous congruence SC $(M, N) = x$ satisfies*

$$\begin{cases} x \;\; mod \;\; m_1 = n_1 \\ x \;\; mod \;\; m_2 = n_2 \\ \ldots \\ x \;\; mod \;\; m_k = n_k \end{cases}$$

and there is exactly one solution x between 0 and C, where $C = \Pi_k^{i=1} m_i$

The simultaneous congruence of two sets of integers can be computed using the following Euler's quotient function:

$$X = (\Sigma_k^{i=1} (\frac{C}{m_i}) \times n_i \times \phi(m_i)) \; mod \; C,$$

where C = $\Pi_k^{i=1} m_i$ and $\phi(x)$ is Euler's totient function [14] which is defined as the total number of integers which are smaller than x and relatively prime to x.

For example, given a list of prime numbers P = [3,4,5], and a list of integers I = [1,2,3], the Chinese remainder theorem states that there exists a number x = 58, where

$$\begin{cases} x \;\; mod \;\; 3 = 1 \\ x \;\; mod \;\; 4 = 2 \\ x \;\; mod \;\; 5 = 3 \end{cases}$$

The Chinese Remainder Theorem makes it possible to generate a one-to-one mapping between the elements in P and I. When the prime numbers in P are self-labels of the nodes in an XML tree, the integers in I represents the ordering of these nodes. The number SC(P,I) = x can therefore be used to capture the global ordering for an XML document. Figure 2.4 shows an XML tree that has been labeled using the prime number labeling scheme. The integer inside the node represents the order of the element within the XML document. The SC value which is generated from self-labels and the order numbers is 29243. Thus, the global ordering for each node can be subsequently derived from the formula: SC mod (self-label). For example, the order number for the node whose self-label is 5 is 3, i.e., 29243 mod 5.

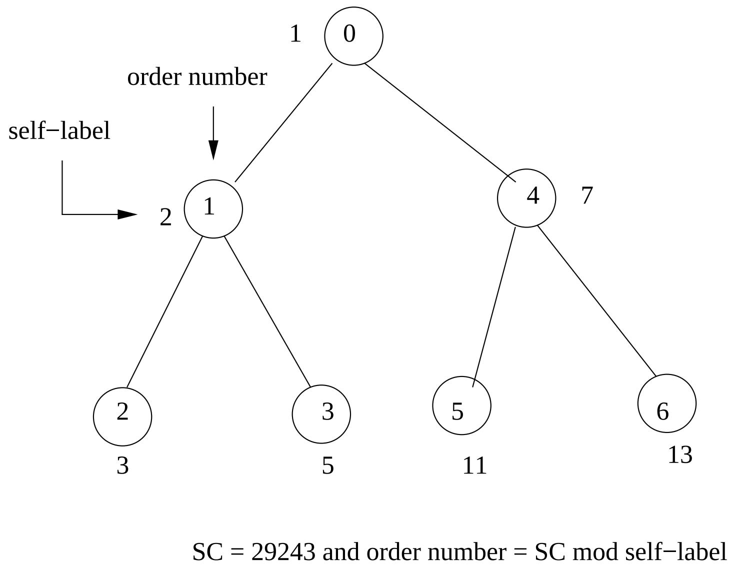


Figure 2.4: Capturing order by an SC value

In practice, huge SC value may be generated for large XML tree. An alternative way is to use a list of SC value instead of a single SC value. Each SC value maintains the global ordering of a subset of the nodes in the XML tree. In Figure 2.5, two SC values are used to capture the ordering of the nodes in the XML tree in Figure 2.4.

The update operation of the prime number labeling scheme is as follows. Suppose a new node with self-label 17 and order number 3 is inserted into the XML tree in Figure 2.4. The corresponding SC value for this record is also updated to

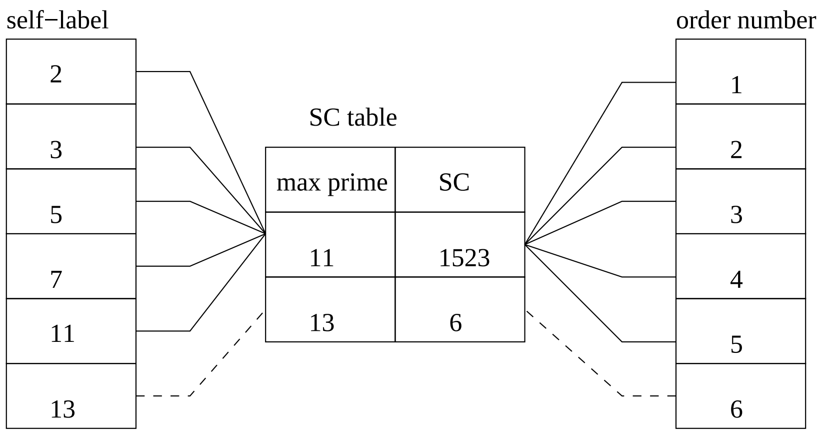


Figure 2.5: SC table for XML tree in Figure 2.4

the new simultaneous congruence value that satisfies the following two equations:

$$\begin{cases} x \ mod \ 13 = 7 \\ x \ mod \ 17 = 3 \end{cases}$$

The order numbers for the nodes that comes after the newly inserted node will be increased by 1. Thus, the SC values associated with these nodes need to be updated accordingly, in this example, the first record of the SC table contains the order number that need to be changed. The new simultaneous congruence value for this record is computed according to the following equations:

$$\begin{cases} x \ mod \ 2 = 1 \\ x \ mod \ 3 = 2 \\ x \ mod \ 5 = 4 \\ x \ mod \ 7 = 5 \\ x \ mod \ 11 = 6 \end{cases}$$

Figure 2.6 shows the updated SC table. Since an SC value can capture the order numbers for several nodes in an XML tree, updating the ordering information of these nodes can be performed by updating the SC value.

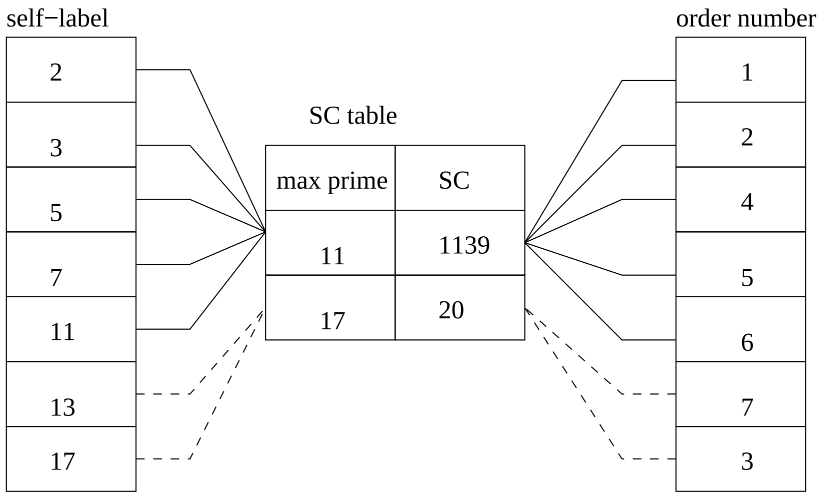


Figure 2.6: Updated SC Table

The major drawback of the prime labeling scheme, comparing with other labeling schemes, is the heavy computation cost for SC value, as our experiments shows in Chapter 5. Moreover, insertion of a single node could cause up to O(N) nodes in the XML tree having their order numbers changed, which means it is possible that all SC values in the SC table require re-computation. The update efficiency could be seriously decreased in this scenario.

2.3 XML Query Processing

In this section, we first introduce the related work on *Structural Join* as it is considered as the core operation of solving XML path query and most recent research efforts on XML index are aimed to make *structural join* more efficient. Then we introduce those research works on efficient XML query processing that are independent of structural join.

2.3.1 Structural Join

Structural join is basically a process to generate a set of element pairs (a, b), where a is an ancestor of b, and these pairs are later merged or joined to produce final result of the path query. It may be called in different ways, for example, *Multi-predicate Merge Join* in [104], *EE-Join* in [66], etc, but the underline idea remains the same. In the section, we will first introduce several structural join algorithms that do not exploit indexes to increase join performance (though they may use index to construct the input lists). Then we introduce several indexing schemes that facilitate structural join processing.

[104] proposed a "Multi-predicate Merge Join" algorithm to solve containment queries, which is generally considered as the first structural join algorithm. The containment relationship referred in [104] is essentially the same idea as ancestor-descendant relationship which we mentioned earlier in this thesis. A pair of numbers (*docno*, *begin:end*, *level*) is assigned to every element in the XML document, where *docno* refers to the identifier of the document, *begin* is the starting position (in term

of words) of the XML element, *end* is the ending position (in term of words) of the element, and *level* refers to the level where the element is in the corresponding XML tree. The Containment Property is defined as follows[104]:

Definition 2.2 ***Containment Property:*** *An occurrence of an XML element ($docno_1$, $begin_1$:end_1, L_1), contains an occurrence of another XML element ($docno_2$, $begin_2$:end_2, L_2), if and only if: (1) $D_1 = D_2$, and (2)$begin_1$:end_1 nests $begin_2$:end_2. For example, (1,1:23,0) contains (1,9:13,2).*

The inputs to Multi-predicate Merge Join algorithm are two lists of elements, which are in fact the inputs to most structural join algorithms. Each list contains the XML element with same label and is sorted by *docno* and *begin* position. The algorithm is presented in Figure 2.7

The whole join process contains two logical steps. In the first step, the equality predicate on *docno* is used to produce pairs of rows whose *docno* values match. In the second step, the inequality predicates are applied on these matching rows. The Multi-predicate Merge Join is essentially a form of nested-loop join, except that seeking is not done on an index, but directly on data records. The "nested-loop" join is performed in the following way: for each outer row, a seek is done on the inner rows until a "start record" is found, then a record scan is conducted and each row during the scan is attempted to join with the outer row; the record scan ends at a "stop record". The next seek does not need to start from the first record, but instead can start from the beginning of the last scanned record. A merge join can be done this way since both the inner rows and the outer rows are sorted.

[66] proposed the decomposition method to solve XML path queries. A complex path expression can be decomposed into several simple path expressions. The

```
Multi-predicate Merge Join (list1, list2)

begin
    set cursor1 at beginning of list1
    set cursor2 at beginning of list2
    while (cursor1 <> end of list 1 and
           cursor2 <> end of list 2) do
        if (cursor1.docno < cursor2.docno) then
            cursor1 ++
        else if (cursor2.docno < cursor1.docno) then
            cursor2 ++
        else
            mark = cursor2
            while (cursor2.position < cursor.position and
                   cursor2 <> end of list2) do
               cursor2 ++
               if (cursor2 == end of list2) then
                  cursor1++
                  cursor2 = mark
               else if (cursor1.val contains cursor2.val) then
                  mark = cursor2
                  do
                    merge cursor1 and cursor2 values
                    cursor2 ++
                  while (cursor1 value contains cursor2 value and
                         cursor2 <> end of list2)
                  cursor1 ++
                  cursor2 = mark
               endif
             endwhile
         endif
      endwhile
end
```

Figure 2.7: The Multi-predicate Merge Join Algorithm

intermediate results from these simple path expressions are combined or joined together to obtain the final result. For example, a regular path expression of the form $E_1/E_2/E_3/E_4$ can be decomposed to E_1/E_2 and E_3/E_4. Then, the intermediate results from E_1/E_2 and E_3/E_4 are joined together to produce the final result. In general, a regular path expression can be decomposed into the following basic subexpressions:

1. a subexpression with a single element or a single attribute,
2. a subexpression with an element and an attribute,
3. a subexpression with two elements,
4. a subexpression that is a Kleene closure(+,*) of another subexpression,
5. a subexpression that is a union of two other subexpressions.

EA-Join, EE-Join and KC-Join were designed to respectively process subexpression 2, 3 and 4. Both EE-join and EA-join can be viewed as structural join. EE-join is principlely similar to Multi-predicate Merge Join mentioned above. To determine parent-child relationship between attributes and elements, we can use either containment comparison, as that is used to determine relationship between elements, or we can directly retrieve the information from the index.

The EA-Join algorithm joins two intermediate results from subexpression, which are a list of elements and a list of attributes. The skeleton of algorithm is shown in Figure 2.8. Since the element (or attribute) index maintains the element (or attribute) records in a sorted order by document identifiers and then position values, the join of the intermediate results can be obtained by a two-step sort-merge operation without sorting. In the first step, elements and attributes are merged according to their document identifers. In the second step, every pair element list and attribute list with the same document identifier is merged by examining the parent-child relationship based on containment comparison, or just the information retrieved from index.

```
Input :
$\{E_1, ..., E_m\}$: $E_i$ is a set of elements having a common document identifier;
$\{A_1, ..., A_m\}$: $A_j$ is a set of attributes having a common document identifier;
Output:
A set of $(e,a)$ pairs such that the element $e$ is the parent of attribute $a$;
//Sort-merge $\{E_i\}$ and $\{A_j\}$ by doc. identifier.
foreach $E_i$ and $A_j$ with the same did do
    //Sort-merge $E_i$ and $A_j$ by PARENT-CHILD relationship
    foreach e $\in E_i$ and a $\in A_j$ do
        if (e is parent of a) then output (e,a);
    end
end
```

Figure 2.8: The EA-Join algorithm

The EE-Join algorithm , shown in Figure 2.9, joins two lists of elements. Like EA-Join, it performs join by a two-step sort-merge operation sorting. But unlike EA-Join, two sets of elements with a matching document identifier cannot be merged in a single scan by EE-Join algorithm. For example, for a pair of elements a and b, their ancestor-descendant is determined by examining whether the order(b) is contained in [order(a), order(a) + size(a)]. Since a descendant element can have more than one ancestor element, it may be necessary to scan the b element list more than once.

[12] proposed a stack-based tree-merge algorithm for structural join. It has been proved to be more efficient than traditional tree-merge algorithm as it avoids scanning descendant list multiple times. The algorithm takes a (*docID*, *startPos:endPos*, *levelNum*) representation of elements, which is similar to the (*docno*, *begin:end*, *level*) representation of elements in [104]. According to the order of output list, the stack-based structural join algorithm family has two members. The first one

```
Input :
{$E_1$, ...,$E_m$} and {$F_1$, ...,$F_n$}: $E_i$ or $F_j$ is a set of elements having a common
document identifier;
Output:
A set of ($e$,$f$) pairs such that the element $e$ is an ancestor of element $f$;
//Sort-merge {$E_i$} and {$F_j$} by doc. identifier.
foreach $E_i$ and $F_j$ with the same did do
    //Sort-merge $E_i$ and $F_j$ by ANCESTOR-DESCENDANT relationship
    foreach e $\in E_i$ and f $\in F_j$ do
        if (e is an ancestor of f) then output (e,f);
    end
end
```

Figure 2.9: The EE-Join algorithm

is called “Stack-Tree-Desc”, where the output list [(a_i,d_j)] is sorted by (docID, d_j.startPos, a_i.startPos). The second one is called “Stack-Tree-Anc”, where the output list is sorted by (docID, a_i.startPos, d_j.startPos).

The algorithm “Stack-Tree-Desc” is presented in Figure 2.10. The basic idea is to take the two input operand lists, AList and DList, both sorted on their (docID, startPos) values and conceptually merge (interleave) them. As the merge proceeds, the ancestor-descendant relationship between the current top of the stack and the next node in the merge, which is in fact the node with the smallest value of startPos, is determined. Based on the comparison, the stack is manipulated and the output is produced. The stack at all times has a sequence of ancestor nodes and each node in the stack is a descendant of the node below it. When a new node from the AList is found to be a descendant of the current top of the stack, it is simply pushed on to the stack. When a new node from the DList is found to be a descendant of the current top of the stack, then it is a descendant of all the nodes in the stack. Also, it is guaranteed that it will not be a descendant of any other node in AList. Hence, the join results involving this DList node with each of the AList nodes in the stack are output. If the new node in the merge list is not a descendant of the current top of the stack, then it is guaranteed that no future node in the merge list is a descendant of the current top of the stack, so the stack may be popped, and the rest is repeated with the new top of the stack.

```
Algorithm Stack-Tree_Desc(AList, DList)

/* Assume that all nodes in AList and DList have the same docID */
/* AList is the list of potential ancestors,in sorted order of
startPos */

/* DList the list of potential descendants in sorted order of
startPos */

a = AList->firstNode; d = DList->firstNode; OutputList = NULL;
while (the input lists are not empty or the stack is not empty) {
    if((a.startPos > stack->top.endPos) &&
      (d.startPos > stack->top.endPos)) {
      /* time to pop the top element in the stack */
      tuple = stack->pop();
    }
    else if (a.startPos < d.startPos) {
        stack->push(a)
        a = a ->nextNode
    }
    else {
        for (a1 = stack->bottom; a1 != NULL; a1 = a1->up){
            append (a1,d) to OutputList
        }
        d = d->nextNode
    }
```

Figure 2.10: Algorithm Stack-Tree-Desc

The algorithm Stack-Tree-Anc is presented in Figure 2.11. It is not straightforward to modify Algorithm Stack-Tree-Desc to produce results sorted by ancestor because of the following: if node a from AList on the stack is found to be an ancestor of some node d in the DList, then every node a' from AList that is an ancestor of a (and hence below a on the stack) is also an ancestor of d. Since the startPos of $a^{'}$ precedes the start position of a, the output of the join pair (a,d) must be delayed until after $(a^{'},d)$ has been output. But there remains the possibility of a new element $d^{'}$ after d in the DList joining with $a^{'}$ as long $a^{'}$ is on stack, so the pair (a,d) cannot be output until the ancestor node $a^{'}$ is popped from stack.

The solution is to associate two lists with each node on the stack: the first, called *self-list* is a list of result elements from the join of this node with appropriate DList elements, the second, called *inherit-list* is a list of join results involving AList elements that are descendants of the current node on the stack. When a new node from the AList is found to be a descendant of the current top of the stack, it is simply pushed on to the stack. When a new node from the DList is found to be a descendant of the current top of the stack, it is simply added to the self-lists of the nodes in the stack. If no new node (from either list) is a descendant of the current top of the stack, then it is guaranteed that no future node in the merge list is a descendant of the current top of the stack, so the stack can be popped and test is repeated with the new top of the stack. When the bottom element in the stack is popped, its self-list is output and then its inherit-list. When any other element in the stack is popped, no output is generated. Instead, its inherit-list is appended to its self-list, and the result is appended to the inherit-list of the new top of the stack.

Various index schemes, and structural join algorithms that make use of these indexes, have then been proposed. We introduce some major research work below.

[66] has first proposed an index structures *XISS* for structural join algorithms. The purpose of these index structures is to allow fast retrieval of elements that will participate the proposed EE/EA-joins. The XISS index structure consists of

```
Algorithm Stack-Tree_Anc(AList, DList)

/* Assume that all nodes in AList and DList have the same docID */
/* AList is the list of potential ancestors,in sorted order of
startPos */

/* DList the list of potential descendants in sorted order of
startPos */

a = AList->firstNode; d = DList->firstNode; OutputList = NULL;
while (the input lists are not empty or the stack is not empty) {
    if((a.startPos > stack->top.endPos) &&
      (d.startPos > stack->top.endPos)) {
      /* time to pop the top element in the stack */
      tuple = stack->pop();
      if (stack->size == 0) { /* the elements were popped. */
        append tuple.inherit-list to OutputList
      }
      else{
        append tuple.inherit-list to tuple.self-list
        append the resulting tuple.self-list to
        stack->top.inherit-list
      }
    }
    else if (a.startPos < d.startPos) {
        stack->push(a)
        a = a ->nextNode
    }
    else {
        for (a1 = stack->bottom; a1 != NULL; a1 = a1->up){
            if (a1 == stack->bottom)
                append (a1,d) to OutputList
            else
                append (a1,d) to the self-list of a1
        }
        d = d->nextNode
    }
```

Figure 2.11: Algorithm Stack-Tree-Anc

three major components: element index, attribute index and structure index. Both the element index and attribute index are implemented as a B^+-tree using name identifiers (*nid*) as keys. Each entry in a leaf node points to a set of fixed-length records for elements (or attributes) having an identical name string, grouped by document they belong to. The element index allows quick retrieval of all elements with the same name string. Each element record includes the extended preorder label and other related information of the element, and the element records are in a sorted order by the *order* values as shown in Figure 2.12. The attribute index has the similar structure as the element index.

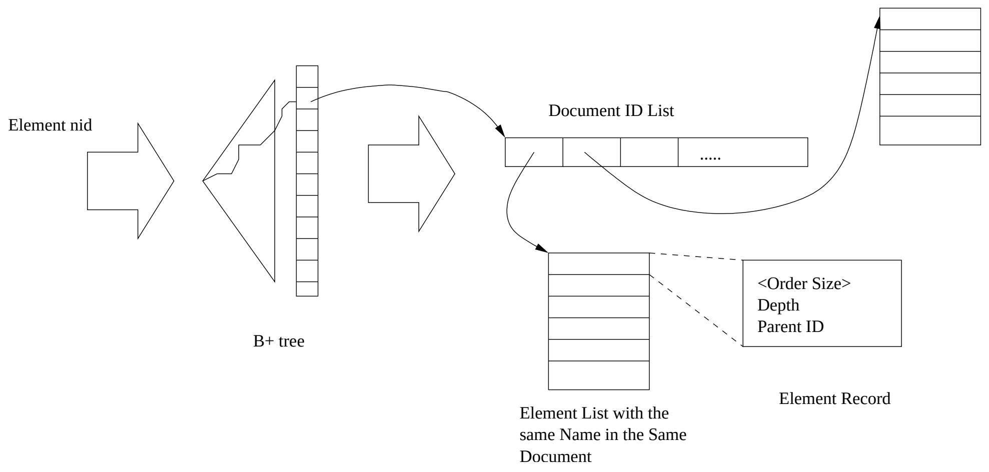


Figure 2.12: XISS element index structure

[25] proposed a structural join algorithm *Anc-Des-B+*, base on the use of B^+-trees, to skip descendants that do not match the considered structural relationship. The algorithm Anc-Des-B+ is shown in Figure 2.13. Initially, variables a and d denote the first elements (having the smallest start position) of the two sorted lists A and D. Then the algorithm systematically moves a and d forward and performs the join until one of the lists becomes empty. During the execution, a stack of

element from list A is maintained. This is similar to the Stack-Tree algorithm in [12]. However, in algorithm Anc-Des-B+, steps 11 and 15 utilize the B^+-tree to skip elements from the A and D lists, respectively.

```
Algorithm Anc-Des-B+ (List A, List D)
1.      Let a, d be the first elements of A and D;
2.      while (not at the end of A or D) do
3.         if (a is an ancestor of d) then
4.            Locate all elements in A that are ancestors of d
              and push them into stack;
5.            Let a be the last element pushed;
6.            Output d as a descendant of all elements in stack;
7.            Let d be the next element in D;
8.         else if (a.end < d.start) then
9.            Pop all stack elements which are before d;
10.           Let l be the last element popped;
11.           Let a be the element in A(locate using B+-tree)
              having the smallest start that is larger than l.end;
12.        else /* a is after d, or a is a descendant of d */
13.           Output d as a descendant of all elements in stack;
14.           if (ancestor stack is empty) then
15.              Let d be the element in D (locate using B+-tree)
                 having the smallest start that is larger than a.start;
16.           else
17.              Let d be the next element in D;
18.           endif
19.        endif
20.     endwhile
```

Figure 2.13: Algorithm Anc-Des-B+

[25] also proposed an enhancement that further improves the join performance. This enhancement relies on the use of *Containment Forest* (*C-forest*). A *Containment Forest* (*C-forest*) is a structure linking the elements from the same tag. Each element corresponds to a node in the structure and is linked to other elements for the same tag via *parent*, *first-child* and right-sibling pointers. These pointers are defined as following.

- Given nodes n and n_p from the same tag, node n_p is called the *parent* of node

n iff: (a)n_p is n's ancestor in the document tree; and (b) there is no other ancestor node n_a of n from the same tag, such that n_p is an ancestor of n_a. n is therefore called a child of n_p.

- Given same-tag nodes n and n_c, node n_c is called the *first-child* of n iff: (a) n_c is a child of n; and (b) there does not exist another same-tag node that is a child of n which is before n_c (node n_1 is before n_2 iff n_1.end < n_2.start).
- Given same-tag node n and n_s , n_s is the *right-sibling* of n iff: (a) n and n_s have the same parent; and (b) there is no same tag node between them which has the same parent (n_2 is between n_1 and n_3 iff n_1.end < n_2.start and n_2.end < n_3.start).

A C-forest has the following properties:

- The (start,end) interval of each node contains all intervals in its subtree (hence the name containment forest)
- The start numbers in the forest follow a preorder traversal.
- The start(end) numbers of sibling nodes are in increasing order.

A C-forest for a given tag can be easily embedded within the B^+-tree that indexes the tag's elements, which is accomplished by adding C-forest parent and next-sibling pointers among the leaf records of the B^+-tree. First-child pointers are implicit because an element and its first child are always stored subsequently in the B^+-tree. The *Anc-Des-B+* join algorithm also works when the B^+-tree is enhanced with the C-forest pointers. One difference appears in step 11 that finds the element a_{new} which has the smallest start position larger than a.end. Consider the embedded C-forest, the relationship between a_{new} and a is as follows: if a has a right-sibling then $a_{new} = a$.sibling. If a does not have a right-sibling, but a's parent has a sibling, then $a_{new} = a$.parent.sibling, and so on. If neither a nor any of a's ancestors have right-sibling, then the join algorithm completes since no other A elements need to be examined.

The index structure XR-tree, proposed in [47], further improves the idea presented in [66] to skip not only descendants but also ancestors. The basis of this index scheme, *stab* and *stab list* are defined as following.

Definition 2.3 *Given a key k and an element with region $E_i(s_i, e_i)$, E_i is said to be stabbed by k, or k stabs E_i, if $s_i \leq k \leq e_i$. Given a set or ordered keys, $k_j (0 \leq j < n)$, where $k_x < k_y$ if $x < y$, and an element $E_i(s_i, e_i)$, E_i is said to be primarily stabbed by k_j, or k_j primarily stabs E_i, if (1)$s_i \leq k \leq e_i$, and (2) for all l, $l < j$, $k_l < s_i$, that is, k_j is the smallest key that stabs E_i.*

Definition 2.4 *Given a set or ordered keys,$k_j (0 \leq j < n)$, where $k_x < k_y$ if $x <y$, and a set of elements $\varepsilon = \bigcup_i (s_i, e_i)$, the stab list of a key k_j is the list of elements in ε that are stabbed by k_j, denoted as SL_j or SL_{k_j}. The primary stab list of a key k_j is the list of elements in ε that are primarily stabbed by k_j, denoted as PSL_j or PSL_{k_j}.*

Definition 2.5 *The start and end positions, ps_j, pe_j of key k_j with primary stab list PSL_j are defined the start and end positions of the first element of PSL_j when $PSL_j \neq \emptyset$, and (nil, nil) if $PSL_j = \emptyset$*

An XR-tree is essentially a B^+-tree with a complex index key entry and extra stab lists associated with its internal nodes. It is defined as following[47],

Definition 2.6 *An XR-tree for a set of region-encoded XML elements is a tree with the following properties:*

1. *An XR-tree is a balanced tree.*

2. *An internal node contains m key entries in the form of (k_i, ps_i, pe_i), with $k_0 < k_1 < \cdots < k_{m-1}$, and d leq m leq 2d, where d is the degree of the XR-tree.*

3. *An internal node with m keys also contains m + 1 pointers p_j, ($0 \leq j \leq m$), pointing to the nodes in the next level of the tree, such that all keys in the node pointed by p_i are less than k_i, and all keys in the node pointed by p_{i+1} are greater than or equal to k_i, respectively.*

4. *An internal node n is associated with a stab list, SL(n), which holds all elements E_i, such that E_i is stabbed by at least one key in n but not stabbed by any key of any ancestor of n. Each element in SL(n) is in the form of (s,e,pointer), where (s,e) is the region of the element and pointer points to the data entry of the element.*

5. *SL_j,PSL_j for the set of all keys k_j in an internal node n are defined on the list SL(n) by Definition 2.4. Each pair of (ps_j, pe_j) of k_j is defined by Definition 2.5.*

6. *Leaf nodes contain element entries, in the form of (s,e,InStabList, pointer), where (s,e) is the region of the element, and s is the index key. InStabList is a flag indicating whether the element is included in any stab list of internal nodes, and pointer points to the data entry of the element.*

7. *Leaf nodes are linked from left to right.*

To process structural join with XR-tree index, two basic operations are first introduced. The first one is *FindDescendants*, i.e., given an element E_a, find all its descendants in an element set indexed by an XR-tree. The second one is *FindAncestors*, i.e., given an element E_d, find all its ancestors in an element set indexed by an XR-tree.

For a given element(s_a, e_a), finding all its descendants is to find all elements E_i such that $s_a < E_i.start < e_a$, which is a simple range query over the start position of elements which are indexed in the backbone of XR-trees. The algorithm is rather straightforward and there is no need to access stab lists when searching for descendants.

For a given element (s_d, e_d),finding all its ancestors is to find all elements E_i such that $E_i.start < s_d < E_i.end$. In other words, it is to search for all elements stabbed by s_d.The basic idea is, during the navigation from the root to the leaf page, the stack lists of internal nodes is searched to collect elements stabbed by s_d. After reaching the leaf page, those elements stabbed by s_d but not included in the stab lists of internal nodes is output. The *FindAncestors* algorithm is presented in Figure 2.14.

```
Algorithm: Find Ancestors

Description: Find all ancestors of $E_D = (s_d, e_d)$ in XR-tree T.

Let node be the root of T;
Search non-leaf pages
while node is not a leaf page do
        Retrieve all elements stabbed by $s_d$ in its stab list, by calling SearchStabList.
        Find the largest key $k_i$, such that $k_i < s_d$.
        If found, let node be $k_i$.rightChild; otherwise, let node be the left child of the
        first index entry.

[Search within the leaf page] Search from the first element of node to output ele-
ments that are stabbed by $s_d$ but with InStabList flags being no, until an element
whose start position is greater than $s_d$ is encountered.
```

Figure 2.14: Algorithm FindAncestors

The algorithm for finding stabbed elements in a stab list is shown in Figure 2.15. Assume that s_d falls in $[k_i, k_{i+1})$. It is clear that s_d cannot stab any element in PSL_j, where $j > i + 1$ because all such elements have their *start* values larger than k_{i+1}, i.e., these elements are "behind" s_d. Therefore, only PSL_c of k_c, where $c \leq i + 1$, needs to be checked.

```
**Algorithm**: SearchStabList
**Description**: Search the stab list of an internal node I for all elements stabbed by $s_d$.

Let $k_i$ be the key in I, such that $k_i \leq s_d < k_{i+1}$;
for c = $i$ + 1 to 0 do
    if $ps_c \neq nil$ and $ps_c < s_d < pe_c$ then
        Scan $PSL_c$ and output the scanned elements until an element not
        stabbed by $s_d$ is encountered;
    endif
endfor
```

Figure 2.15: Algorithm SearchStabList

The structural join algorithm using the XR-tree to skip ancestors or descendants is outlined in Figure 2.16. According to the results presented in [64], the performance of XR-tree highly depends on the buffer size mainly because of the additional scans on the stab-lists. If the buffer size is small, XR-tree is likely to perform less efficient than the B$^+$-tree index approach presented in [25]. If the buffer size is big(100kb, as in their experiments), XR-tree can outperform the B$^+$-tree index approach presented in [25] because stab-list pages can be pinned in memory. Also according to the results presented in [64], XR-tree has higher update cost because additional maintenances are required for the stab-lists.

```
**Algorithm**: Stack-based Structural Joins with XR-trees
**Input**: A is the ancestor set and D is the descendant set.

CurA := First(A);
CurD := First(D);
stack := $\emptyset$;
while CurA$\neq$Endof(A) and CurD$\neq$Endof(D) do
    if stack $\neq \emptyset$ then
        pop all elements that are not ancestors of CurD;
    endif
    if CurA.start < CurD.start then
        $A_d$ = FindAncestors(A,CurD.start);
        for each $a_j \in A_d$, if $a_j \notin stack$, put it on the stack;
        output pairs ($a \in stack$, CurD);
        CurA := first element in A whose $start > CurD.start$;
        CurD := next element in D after CurD;
    else
        if stack $\neq \emptyset$ then
            output pairs $(a \in stack, CurD)$;
            CurD := next element in D after CurD;
        else
            CurD := first element in D whose $start > CurA.start$;
        endif
    endif
endwhile
```

Figure 2.16: Stack-based Structural Join Algorithm with XR-trees

2.3.2 Non-SJ based Query Processing

Quite a number of other techniques were proposed to speed up XML query processing other than structural join algorithms. Some work are based on building summary indexes, usually in the form of a labelled directed graph, of XML document [81, 26, 39, 73, 29, 38, 53, 75]. The basic idea is to extract paths and summary nodes from the original XML data. Such an index typically consists of a structural summary, together with a mapping from summary nodes to actual data nodes. It can be used to evaluate path expressions directly by pruning search space. [29] encodes paths as strings and uses a Patricia trie[56] based index called Index Fabric to index these strings. A major drawback of these kinds of approaches is that they cannot efficiently solve expressions that contain "//",where ancestor-descendant relation is involved, as it may be necessary to scan all stored paths to find possible matches.

Many other techniques have been proposed recently to solve XML query. Among them, the technique presented in [40], based on the employment of R-trees [41] and B-trees, is able to support any kind of structural relationship, according to XPath [9] specification. Holistic twig join [19] was proposed to match XML twig patterns without decomposition. [46] proposed holistic algorithms to solve twig queries with OR predicates. The technique proposed in [51] extends the notion of covering index to deal with branching path expressions. ViST [96] proposed a B-tree base approach to directly solve branching queries without applying structural join algorithm. [83] transforms both XML document and twig query patterns into sequences of labels by $Prüfer's$ methods[43] and then identifies the occurrence of twig pattern by performing subsequence matching on these $Prüfer's$ patterns.

Set-based techniques [19, 21, 67, 68] use a chain of linked stacks to compactly represent partial results to root-to-leaf query paths, which are then composed to obtain matches for the twig pattern. The algorithm presented in [19] makes use of the (*docID, startPos:endPos,level*) representation of elements. Each node q is query twig pattern is associated with a stream T_q, The stream contains the positional representations of the database nodes that match the node predicate at the twig pattern node q. The nodes in the stream are sorted by their (docID, startPos) values. A stack S_q is also associated with each query node q. Each data node in the stack consists of a pair: (positional representation of a node from T_q, pointer to a node in $S_{parent(q)}$). At every point during the computation, (i) the nodes in stack S_q (from the bottom to top) are guaranteed to lie on a root-to-leaf path in the XML database, and (ii) the set of stacks contain a *compact encoding* of partial and total answers to the query twig pattern, which can represent in linear space a potentially exponential (in the number of query nodes) number of answers to the query twig pattern. Figure 2.17 illustrates the stack encoding of answers to a path query for a sample dataset. The answer [A_2, B_2, C_1] is encoded since C_1 points to B_2, and B_2 points to A_2. Since A_1 is below A_2 on the stack S_A, [A_1,B_2,C_1] is also an answer. Finally, since B_1 is below B_2 on the stack S_B, and B_1 points to A_1, [A_1, B_1, C_1] is also an answer. [A_2,B_1,C_1] is not an answer, since A_2 is above the node (A_1) on stack S_A to which B_1 points.

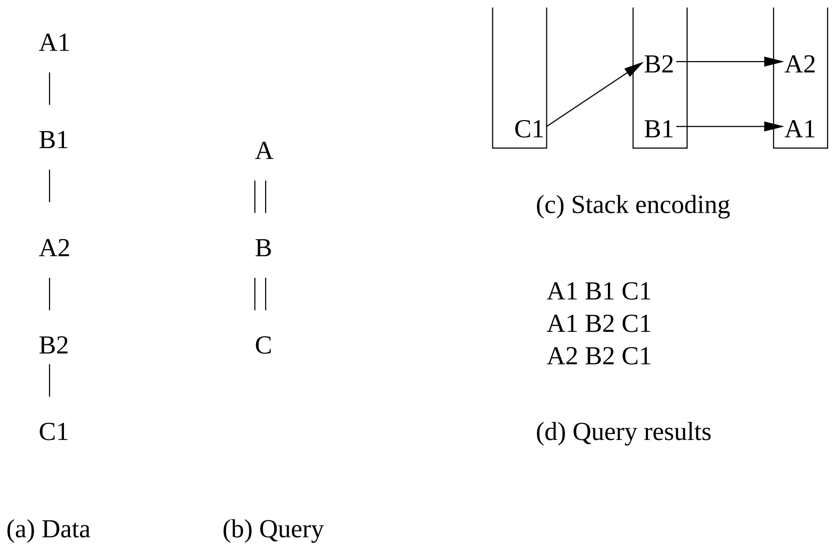


Figure 2.17: Compact encoding of answers using stacks

Algorithm TwigStack[19], which computes answers to a query twig pattern, is presented in Figure 2.18. It operates in two phases. In the first phase, some (*but not all*) solutions to individual query root-to-leaf paths are computed. In the second phase, these solutions are merge-joined to compute the answers to the query twig pattern. Before a node, h_q, from the steam T_q is pushed on its stack S_q, TwigStack ensures that: (i) node h_q has a descendant h_{q_i} in each of the streams T_{q_i} for $q_i \in$ children(q), and (ii) each of the nodes h_{q_i} recursively satisfies the first property. Thus, when the query twig pattern has only ancestor-descendant edges, each solution to each individual query root-to-leaf path is guaranteed to be merge-joinable with at least one solution to each of the other root-to-leaf paths. This ensures that no intermediate solution is larger than the final answer to the query twig pattern.

```
**Algorithm** TwigStack(q)
    //Phase 1
    while ¬end(q)
        $q_{act}$ = getNext(q)
        if (¬isRoot($q_{act}$))
            cleanStack(parent($q_{act}$, nextL($q_{act}$))
        if (isRoot($q_{act}$) ∨ ¬empty($S_{parent(q_{act})}$))
            cleanStack($q_{act}$,next($q_{act}$)
            moveStreamToStack($T_{q_{act}}$,$S_{q_{act}}$, pointer to top($S_{parent(q_{act})}$))
            if (isLeaf($q_{act}$))
                showSolutionWithBlocking($S_{q_{act}}$,1)
        else advance ($T_{q_{act}}$)
    //Phase 2
    mergeAllPathSolutions()
Function getNext(q)
    if (isLeaf(q)) return q
    for $q_i$ in children(q)
        $n_i$ = getNext($q_i$)
        if ($n_i \neq q_i$) return $n_i$
    $n_{min} = minarg_{n_i}$ nextL($T_{n_i}$)
    $n_{max} = maxarg_{n_i}$ nextL($T_{n_i}$)
    while(nextR($T_q$ < nextL($T_{n_{max}}$))
        advance ($T_q$)
    if (nextL($T_q$ < nextL($T_{n_{min}}$)) return q
    else return $n_{min}$
```

Figure 2.18: Algorithm TwigStack

A more efficient variation of TwigStack algorithm is TwigStackXB[19], which uses a variation of B^+-tree, the XB-tree, to speed up the algorithm by skipping parts of the input list without miss any matches. One of the limitations of TwigStackXB is that the effectiveness of skipping data depends on the distribution of the matches in the input lists. Also, both TwigStack and TwigStackXB algorithm solve twig pattern involving parent-child relationship less efficiently than ancestor-descendant relationship. The algorithms may need to check unnecessary matches that are not included in the final result.

2.4 XML Update

Compared with XML query evaluation, the topic of updating XML documents has received much less attention from the research community. Current research efforts mainly focus on how to update XML index efficiently, so that no rebuilding of the index is required. For the indexes making use of labeling schemes, most work concentrates on the development of various dynamic labeling schemes that cope with updates, which we have already introduced in Section 2.2. In this section, we will introduce some other works on this topic.

In [91], a set of basic update operations for XML data has been proposed and the XML query language, XQuery, is extended to incorporate these update operations. Assume there is the presence of a path-expression-matching operation that binds variables to objects within the input XML document and returns tuples of references to the selected objects. One of these bindings will be the *target* of the sequence of operations, and is assumed implicit in the specification below. The update operations also take a set of parameters. The proposed primitive operations of an update are described as:

- **Delete**(*child*): if the *child* is a member of the *target* object, it is removed.

Valid types for *child* include PCDATA, attribute, IDREF within an IDREFS list, and element. If the *child* is a reference within an IDREFS, only the single entry is removed - the remainder of the IDREFS is preserved.

- **Rename**(*child*, name): if the *child* is a non-PCDATA member of the target object, it is given a new name, Note that an individual IDREF cannot be renamed within an IDREFS, such a rename operation will rename the entire IDREFS.

- **Insert**(content): inserts new content, which can be PCDATA, element, attribute, or reference, into *target*. An attempt to insert an attribute with the same name as an existing attribute fails. An attempt to insert a reference with the same name as an existing IDREFS adds an extra entry into the IDREFS. In an ordered execution model, all non-attribute insertions are defined to occur at the end, i.e., the new contend is appended.

- **InsertBefore**(*ref*, content): (defined only for ordered execution). If *ref* is a child element of target or PCDATA, then content must be an element or PCDATA, and it will be inserted directly before *ref* in *target*'s list of children. If *ref* is an entry in an IDREFS, then content must be an ID and it it inserted directly ahead of *ref* in the IDREFS. **InsertAfter**(ref, content) is defined analogously.

- **Replace**(*child*, content): atomic replace operation, equivalent to InsertBefore(*child*, content) followed by Delete(*child*) in the ordered model, or (Insert(content), Delete(*child*)) under unordered execution.

- **Sub-Update**(*patternMatch*, *predicates*, *updateOp*); starting at the *target* element, invokes a new pattern-matching operation over the input, returning bindings that are filtered by the *predicates*. For each valid combination of bindings, recursively invokes the update operation. This allows expressing updates at multiple levels within a complex XML structure.

A full update-operation may consist of several of these sub-operations that execute in sequence. Therefore, several additional restrictions are added to the semantics of the operations in order to prevent ill-defined semantics. All bindings within **Sub-Update** operations are made over the input *before* any updates take place. Likewise, *content* is evaluated for each *target* before the sequence of updates is executed. Finally, a binding that has been deleted cannot be used by any operations later in the sequence (except as *content*).

The work in [52], based on the notion of graph bisimilarity, analyzes two kinds of updates - the addition of a subgraph, intended to represent the addition of a new document to the database, and the addition of an edge, to represent a small incremental change. The details of the first on is shown below as it is more related to our proposed scheme in Chapter 5.

The notion of graph bisimilarity[52] is defined as below.

Definition 2.7 *A symmetric, binary relation $\approx$ on V_G is called a bisimulation if, for any two data nodes u and v with $u \approx v$, we have that*

1. *u and v have the same label, and*
2. *if u' is a parent of u, then there is a parent v' of v such that u'$\approx$v', and vice-versa*

Two nodes u and v in G are said to be bisimilar, denoted by $u \approx^b v$, if there is some bisimulation $\approx$ such that $u \approx v$

The partition of V_G induced by $\approx^b$ defines an index graph, refereed to as *Bisim*(G) or simply "the 1-index [73]". Suppose there is a database of XML documents on which the 1-Index is already built and a new document is added to the database (suppose there are no inter-document references). The problem is how to find the new 1-index without having to recompute it from scratch. Let the data graph corresponding to the database before the addition of the new file be G, the 1-index be I_G and let the addition of the new file correspond to the addition of a new subgraph H under the root. The following theorem[52] enables computation of the modified index from the old index and the index on the new file, without having to look at the whole of the current data.

Theorem 2.4 *Let G be a data graph, Let Bisim(G) be the 1-Index constructed from the bisimulation relation and Bisim*$^{ref}(G)$ *be the index graph constructed from any refinement of Bisim(G), Then Bisim(Bisim*$^{ref}(G)$*) = Bisim(G). Here, graph equality means isomorphism.*

```
procedure subgraph-add(G,H)
//Graph H added under root of G

1.      Let I ← 1-Index of G
2.      Compute the 1-Index of H, Let it be $I_H$
3.      Add $I_H$ as a subgraph under the root of I. Let this graph be I'
4.      Treat I' as a data graph and compute its 1-index. Let it be $I_{new}$
5.      Set the extents of the nodes of $I_{new}$ by blowing up the current extents
6.      return $I_{new}$
end
```

Figure 2.19: Algorithm Addition of a subgraph

The algorithm to find new 1-index without recomputing it from scratch is outlined in Figure 2.19. While computing I_{new} from I', the index extents have nodes of I'. Therefore, the nodes of I' , together with their respective extents that consist of data nodes, needs to be “blow up” in order to obtain the original data nodes corresponding to an extent.

2.5 Summary

In this chapter, we have reviewed research works in the area of XML document processing. We first reviewed several mapping schemes which map XML documents to relational DBMS. Mapping XML documents into relational DBMS is the current trend of storing XML document, yet it is facing many challenges to handle irregular structures of XML documents. Building native XML DBMS, where XML documents are stored as plain documents, is attracting more and more attentions, but it still needs time to meet industry requirements. We then reviewed various labeling schemes which play important roles in XML document query processing. Labeling XML elements allows us to quick determine ancestor/descendant relationships between elements, but it also incurs processing overhead, especially when update operations take place. Next, we reviewed technologies used to process XML queries. Structural join was proposed to match XML elements with ancestor/descendant relationships, which is generally considered as the core operation to solve XML path queries. Some other technologies that are independent of structural join were also proposed to overcome the major disadvantage of structural join, i.e., generation of possibly large intermediate results. We reviewed research works regarding XML update at last. Most research efforts on this topic focus on developing various dynamic labeling schemes. Besides that, there are also research works focusing on dynamic inserting/removing sub-graphs to/from original XML trees.

Chapter 3

The XStorM Mapping Scheme

3.1 Introduction

There are various possible ways to store and query XML data: ranging from a primitive file system, a relational database, an object-oriented database to a special-purpose system (the "native" XML database). The file system is the most straightforward option, but it lacks support for querying the XML data. An object-oriented database system has rich data modeling capabilities, which are useful for clustering XML elements and sub-elements. Unfortunately, the current generation of object-oriented database systems is not fully developed to process queries with heavy workload on large databases. Native XML database systems should work best theoretically as they are specially designed to work on XML data. However, such systems are not yet so common and take time to get mature enough to be able to compete with relational database systems.

Therefore, the main trend is to leverage the robust and widespread technology by using a relational database system. Relational stores are great at providing multiple distinct logical views on the same data with very good scaling and transactional characteristics. Oracle 8_i lets the user or system administrator decide how XML elements are stored in relational tables. [17, 60, 88, 93] generate relational schema(s) based on the given XML DTD or schema. STORED [32] determines relational schemas based on a data-mining algorithm and any data that cannot be accommodated in these schemas is stored in external data structures, which are called overflow graphs. This scheme faces the problem of integrating the relational storage with external data structures and in case there are too many overflow graphs due to highly irregular data instances, its query performance is likely to degrade significantly. [37] studies various schemes to map edges and nodes in the graphical representation of XML document into relational tables. The *Binary* approach, which has the best experimental performance among all these schemes, creates a relational table for each XML tag and stores the value accordingly. There are as many *Binary* tables as there are different tags. The values of the attributes are stored together (inlined) in the same table. The problem of this approach is that number of operations required to answer a query is proportional to the number of attributes involved, which makes the query processing very inefficient.

We address the above mentioned drawbacks and propose a new mapping scheme, XStorM, to store XML data in relational database. We distinguish elements that represent real world entities (*objects*) from elements that represent properties of objects (*attributes*). We avoid excessive fragmentation by mapping each *object*, together with the majority of its *attributes*, into a *core* relational table. And for the overflow data that cannot be accommodated into *core* table, we store them in separate relational tables, or *overflow* tables. We also embed structural information into the attribute/table name for the purpose of fast reconstruction of the original XML document. Our experiments show that XStorM gives good query performance, uses minimal space requirements and is scalable.

The rest of this chapter is organized as follows. We present both table structure of XStorM and the procedure to generate these tables in Section 3.2. Section 3.3 gives the performance study results and we conclude in Section 3.4.

3.2 XStorM Mapping Scheme

In this section, we will introduce our proposed XStorM mapping scheme, including both its table structure, and the procedure to generate these relational tables.

3.2.1 The table structure

As we have mentioned earlier, we observe that there are two types of XML elements: one that denotes *objects* or entities in the real world and one that denotes *attributes* or the properties of entities. Note that in XML, an attribute can also be defined within the start tag of an element, for example, in the following element

$$< book\ btype = \text{``}textbook\text{''} >\ \ myBook\ \ < /book >$$

where *btype* is an attribute of the *book* element. Such attributes bear textual information instead of structural information and we have to mark these attributes when we store them in a relational database so that we can reconstruct the XML data instance correctly. In addition, we can differentiate between collections of attributes versus collection of objects. Figure 3.1 shows how $authors$ becomes a collection of objects when the XML document records more information about an author such as *institution*, *country*, etc. in addition to *name*. Note that $authors$ can also be a mixed collection of attributes (author just have *name*) and objects (author has *name*, *institution* and *country* or just *name* and *institution*) as shown in Figure 3.2.

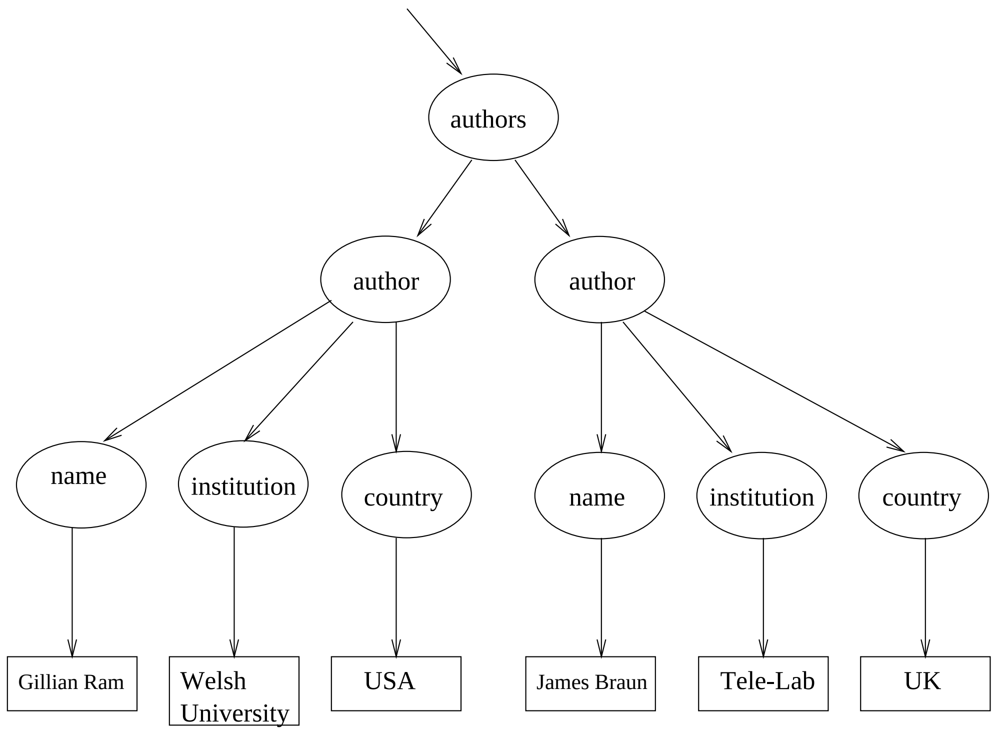


Figure 3.1: Example of Authors as a collection of objects

It is important that different types of objects should be stored in separate tables since they are likely to be referenced somewhere else. For example, if *author* is a complex object consisting of *name, institution, country*; then it should be stored in a separate table since a person is likely to write more than one article and repeating his information for each article leads to redundancy and updating anomalies. However, we have a number of options when *authors* is just a collection of author names. For example, if the articles almost always have two authors, then we can consider storing all the attributes of article including $author_1$ and $author_2$ in the same table. In practice, however, this may not be possible as the XML data is often irregular. An article A may have 5 authors while another article B may have only 2. If we map the article element together with all its attributes (including fields $author_1$,$author_2$,$author_3$,$author_4$,$author_5$) to a relational table, then all the author fields for article B will be NULL except for $author_1$ and $author_2$. This

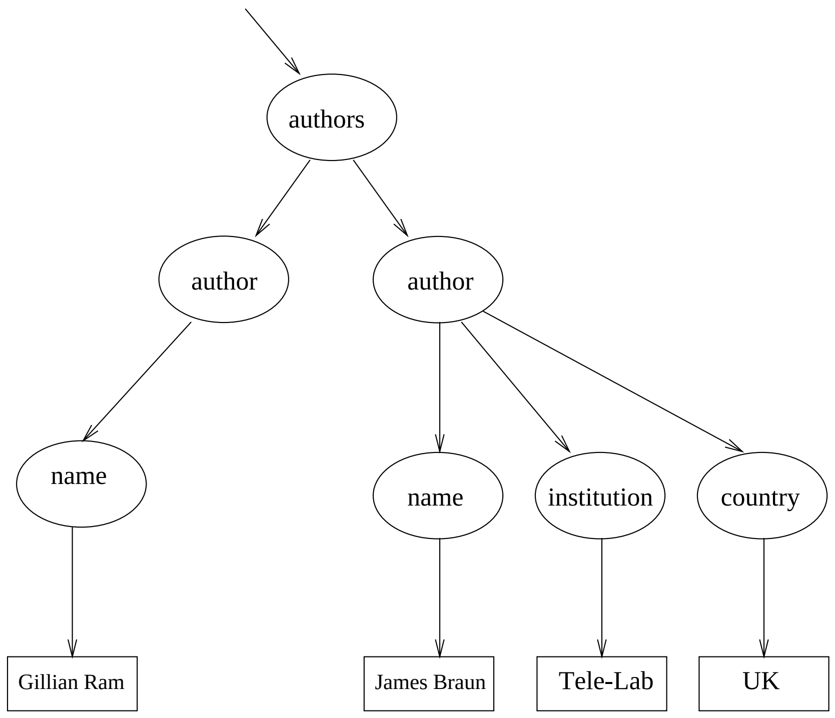


Figure 3.2: Example of Authors as a mixed collection of attributes and objects

situation is worsened if the majority of the articles have two authors. Adding more authors to an article becomes problematic without expanding the table.

One solution is to store collections of attributes in a separate table. Hence, we can have a *core* relational table for an object containing all the single-valued attributes and separate *overflow* tables for collections of attributes. This scheme resembles how multi-valued attributes are handled in relational data model. However, this may not work too well if an XML element has many different small collections of attributes, leading to many overflow tables and subsequent joins for heavy queries.

On the other hand, suppose we know that the majority of the articles have two authors, with a few outliers that have more than two authors or even one author. In this case, we can incorporate $author_1$ and $author_2$ into the *core* relational table for *article*. Additional authors will be stored in an overflow *author* table. If an article has only one author, then we simply set the $author_2$ field to NULL.

Table 3.1 shows the core relational table obtained by assuming that the majority of the articles have two authors and Table 3.2 shows the corresponding overflow table. Note that we have embedded the structural information of the XML document in the attributes authors.$author_1$ and authors.$author_2$ and overflow table name article.*authors.author*. The subscripts are used to ensure uniqueness of names.

Core_Article							
articleID	issueNo	title	initPage	endPage	authors.author1	authors.author2	abstract
1	16	Inter...	365	376	Gillian Ram	James Braun	...
2	18	Parallel...	123	133	Jacob Linz	Paul Tan	...

Table 3.1: Core table example

Overflow_Atricle.authors.author		
articleID	index	author
2	1	Kelvin Tan

Table 3.2: Overflow table example

3.2.2 The mapping procedure

In this section, we describe the procedure to map XML data into relational tables by our proposed XStorM mapping scheme. The steps involved are:

1. Identification of XML objects.

2. Identification of frequent tree patterns in XML graph.

3. Generate core relational tables.

4. Generate overflow tables.

We will elaborate on each of these steps in the following subsections.

Object identification

As noted earlier, element nodes in DOM can be differentiated into object nodes and attribute nodes. The goal of this step is to find all the object nodes in an XML data instance. For example, if we have an XML data instance containing 1000 articles, then we need to identify all nodes that represent the object *article*. Depending on whether the DTD of the XML instance is presented or not, we have two ways to identify these object nodes.

If the DTD of the XML instance is given, it is rather straightforward to find all object nodes we are interested at. We first extract all the paths that leads from the root node to the object node from the semantic information presented in the DTD. Then we use a depth-first search algorithm to traverse the XML instance to find all the object nodes which these paths lead to. Figure 3.3 gives the algorithm for finding the objects with a path extracted from DTD. We shall note that the paths extracted from DTD may not be explicit, i.e., they may contain wildcards. Therefore, we also need to include matching wildcards while we are comparing the current path and object path in line 2 of the algorithm. The cost of the algorithm is $O(N)$, where N is the total number of nodes in the document.

However, if the DTD of the XML instance is not presented, to identify nodes that represent a specific object is a daunting task. Without any semantic information of the XML instance, it is difficult, if not impossible, to know whether a node represents an object or not. We adopt a three-step approach in our work. First, we determine the number of path corresponding to a prefix. We shall refer to this as the *support* of the prefix. Next, we identify the minimal prefix which is

```
IdentifyObjectWithPath(objPath)
1.  Identify(docRoot,emptyPath)

Identify(root,curPath)
1.  curPath := curPath + root.tag
2.  If(curPath = objPath)
3.  Then
4.      Add root to object node set
5.  ELSE
6.      For every child node i of root
7.          Identify(i,curPath)
```

Figure 3.3: Algorithm to identify object nodes with a path extracted from DTD

the shortest prefix whose support is greater than or equal to some certain predetermined threshold. Finally, the node at the lowest level of the minimal prefix is the target object that we are looking for. This scheme can be implemented efficiently using the well-known breadth first search (BFS) algorithm as shown in Figure 3.4. Suppose the XML tree is a balanced tree, which is true for most cases, the cost of the algorithm is *O(NlogN)*, where N is the total number of nodes in the document.

Figure 3.5 shows the partial tree of a sample XML document, which we use as an example to illustrate the process of object identification when the DTD is not presented. Suppose the object we are interested in is "article", then the path we need to identify is $SigmodRecord \rightarrow issue \rightarrow article$. We first carry out a BFS on the data tree. During the search process, we will discover the prefixes. If the support of a prefix exceeds a certain threshold or minimal support, then we record it. Nodes that support this prefix will not be expanded, i.e., their children will not be pushed into the queues in the algorithm. We set the minimal support to be 3, which is a heuristic value. The prefix we will discover first is: $SigmodRecord \rightarrow issue$. However, the support of this prefix is only 2 (i.e., only 2 "issue" tags), which is less than the minimal support. The next prefix found is: $SigmodRecord \rightarrow$

```
IdentifyObjectWithoutPath()
1.  IdentifyPath(minSupport)
2.  For every path $p$ in identified path set
3.      IdentifyObjectWithPath($p$)

IdentifyPath(minSupport)
1.  Enqueue($Q$,docRoot)
2.  While $Q$ is not empty
3.  Do $node$ = Head($Q$)
4.      If ($node$.prefix exists in identified path set)
5.      Then addSupport(node.prefix)
6.      Else Add node.prefix to identified path set
7.      If $node$.prefix.support $\geq$ minSupport
8.          add $node$.prefix to identified path set
9.      Else For every child $i$ of $node$
10.             Enqueue($Q$,$i$)
11.     Dequeue($Q$)
```

Figure 3.4: Algorithm to identify object nodes without predefined path

issue $\rightarrow$ *article*, The support of this prefix is 4 and it is greater than the minimal support, so we record this prefix. As mentioned previously, the children of *article* node will not be pushed into the queue. The search process therefore stops. The prefix we recorded is then used as the path to identify the four *article* objects in the sample. Note that choosing an appropriate minimal support value is crucial. In the above example, if the minimum support value we choose is 2, then the object identified will be *issue*, not *article*.

Frequent tree pattern identification

Given the "schema-less" semi-structured XML data, it is very difficult to find a general schema that covers the whole XML data instance. [32] showed that generating a storage schema for semistructured data that minimizes computational cost is NP-hard with respect to the size of the input. Dynamic programming algorithms

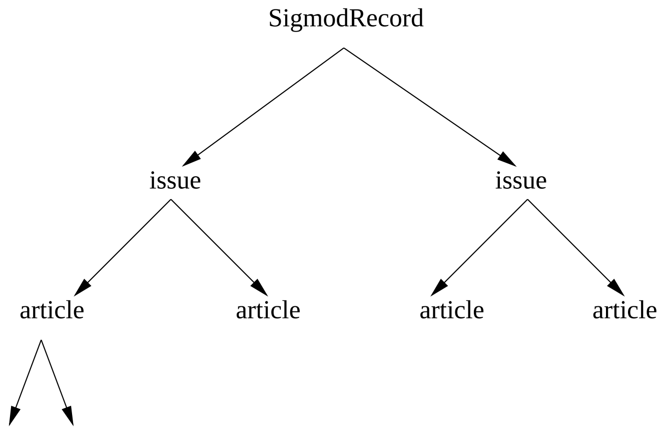


Figure 3.5: Example of Object Identification

are not feasible in this case. Instead, we use a data-mining algorithm to identify frequent tree patterns in an XML graph. This enables us to generate a schema that covers a major portion of the data. Our aim is to incorporate as many small attribute collections into an object's core relational table as possible in order to minimize the number of overflow tables. Query performance is improved when excessive fragmentation is avoided because the number of joins is reduced.

We adopt the data-mining algorithm for semistructured data described in [97] for our purpose here. First let us review some concepts used in the algorithm. A *tree-expression* is a tree-like structure for representing patterns in the DOM graph. A *k-tree-expression* is a tree-expression containing k leaf nodes. A k-tree expression, $k \geq 1$, can be constructed by "gluing" a sequence of k 1-tree expressions that are not prefixes of each other. A 1-tree expression is actually a simple path from a root node to a leaf node. For example, suppose we have three 1-tree expressions p_1, p_2, p_3 as shown in Figure 3.6. Then, a 3-tree expression p_4 can be constructed from p_1, p_2, p_3 and the sequence is $< p_1, p_2, p_3 >$. We say that $p_4 =< p_1, p_2, p_3 >$. Note that if the sequence of the 1-tree-expression is different, then a different k-tree expression is constructed. Let p_i denote a 1-tree-expression, for $i \geq 1$. The k-tree-expression $< p_1, p_2, ..., p_k >$ is constructed from two *(k-1)*-tree-expressions

$< p_1, p_2, ..., p_{k-2}, p_{k-1} >$ and $< p_1, p_2, ..., p_{k-2}, p_k >$. We call these two *(k-1)*-tree-expressions a *matching pair*. This property of k-tree expression is very useful as it prunes our search. We do not need to consider a k-tree-expression if it has some "subtree-expression" that is known to be infrequent.

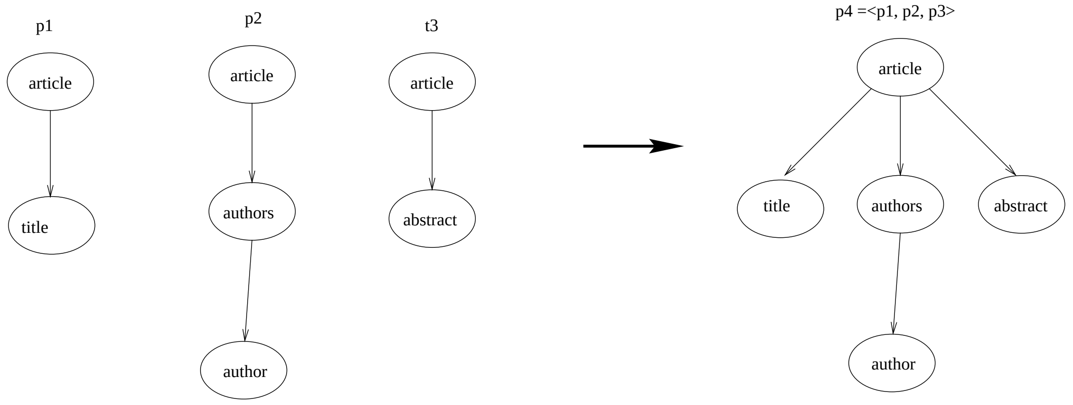


Figure 3.6: Example of how a k-tree-pattern can be constructed from k 1-tree-expressions

In order to determine frequent k-tree-expressions, we use the depth first search algorithm to discover all the 1-tree-expressions starting from the object nodes found in step 1. The support of these 1-tree-expressions are tracked. Frequent 1-tree-expressions are used to generate 2-tree-expressions; frequent 2-tree-expressions are used to generate 3-tree-expression, and so on. Finally, the algorithm will generate frequent k-tree-expressions for a given k. A large k should be used to find a schema with maximal data coverage. Figure 3.7 gives the details of the algorithm. Theoretically, for each frequent k-tree-pattern, we need to scan the whole XML data once. The cost of this step is therefore O(αN), where α is the number of frequent k-tree-expressions discovered and N is the total number of nodes in the XML data tree. We are aware that there are more efficient methods to discover frequent k-tree-expressions. But as our main focus of this piece of work is to develop an XML-to-Relational mapping scheme, we still use this algorithm for it is sim-

```
**Frequent Tree Patterns (k)**
/* find frequent 1-tree-expressions */
1. $T_i$, $1 \geq i \geq k$: sets of i-tree-expressions, initially empty;
2. $F_i$, $1 \geq i \geq k$: set of frequent i-tree-expressions, initially empty;
3. For each object node n do
4.         Depth-First-Traverse(n);
5.         For each 1-tree-expressions t found do
6.                 If (t IN $T_1$)
               /* increase the support of t */
7.                         addSupport (t);
8.                 Else
               /* add t to the list of 1-tree-expressions */
9.                         $T_1 = T_1$ UNION {t};
10.For each t in $T_1$ do
11.         If (getSupport(t) $\geq$ THRESHOLD)
12.                 $F_1 = F_1$ UNION {t};
/* generating k-tree-expression */
13.For i = 2 to k do

/* generate i-tree-expression from matching (i-1)-tree-expression */
14.         $T_i$ = combineMatchingParis($F_{i-1}$);
15.         For each t in $T_i$ do
16.                 If (getSupport(t) $\geq$ THRESHOLD)
17.                         $F_i = F_i$ UNION {t};
18.END
```

Figure 3.7: Algorithm to find frequent tree patterns

ple to implement and it is quite effective to discover frequent k-tree-expressions. "THRESHOLD" is a user-specified value and may vary on different XML data instances.

Generate core relational tables

The frequent k-tree-expression(s) obtained in step 2 creates schema(s) for the XML data. SQL statements that create relational tables can now be generated from these schema(s). Root nodes in the k-tree-expressions represent objects and each object node n in the schema is mapped to a core relational table R. Leaf nodes in the tree rooted at n become attributes of R. In addition, R has an attribute that stores the object identifiers, for example, *articleID* in Table 3.1. The XML data can now be loaded into these relational tables. Nulls are used for any missing data.

Figure 3.8 gives the algorithm for generating a relational schema from a frequent tree expression. The algorithm first uses the Depth-first search algorithm to traverse every leaf node of the tree pattern and convert the tag name of these leaf nodes to the attributes of the final relational schema. If there are other nodes between the root node and a leaf node, the resulting attribute name corresponding to this leaf node will then be defined as the path from the root node to this leaf node(including the left node name), instead of the name of the leaf node alone. Moreover,if there are leaf nodes with identical names, we need to append index numbers to the corresponding attribute names so that they can be differentiated. After the Depth-first traverse, we will get a list of attribute names. Then we can write SQL statement to create relational table based on this attribute list. Note that we need to add one more attribute, which is *object ID*, to the attribute list before we write the SQL statement. The types of these attributes can be determined by the user according to his knowledge or just character string if it can not be decided. The cost of the algorithm is O($MlogM$), where M is the number of tree nodes in the frequent k-tree-pattern.

Generate overflow relational tables

Since XML is semistructured, not all the data can fit into the core tables. In contrast to STORED which uses overflow graphs in external storage, we store the extraneous data in overflow tables in the relational database as shown in Table 3.2. Figure 3.9 describes the algorithm for storing XML data into relational tables according to our XStorM scheme. Overflow tables are created on-the-fly while

```
GenerateSchema(treePattern)
1. GenerateAttribute(treePattern.root)
2. add objectID to attributeList
3. generate (attributeName, attributeType) pairs
4. generate relational table schema.

GenerateAttribute(root)
1. If root has no child /* leaf node */
2. Then
3.     If (root.path = treePattern.root.name) AND (Index(root.name) = 0)
4.     Then
5.         add root.name to attributeList
6.     Else If (root.path ≠ treePattern.root.name) AND (Index(root.name) = 0)
7.     Then
8.         add (root.path + root.name) to attributeList
9.     Else If (root.path = treePattern.root.name) AND (Index(root.name) > 0)
10.    Then
11.        add (root.name + Index(root.name)) to attributeList
12.        IncreaseIndex(root.name)
13.    Else
14.        add (root.path + root.name + Index(root.name)) to attributeList
15. Else /* non leaf node */
16.    For every child i of root
17.        If (Index(root.path + root.name) > 0)
18.        Then
19.            IncreaseIndex(root.path + root.name)
20.        GenerateAttribute(i)
```

Figure 3.8: Algorithm to create a relational schema from a tree expression

we map the XML instance to the core relational tables. The mapping process starts from each object element that has been identified in the first step of XStorM mapping scheme. For each data tree rooted at a object element, we use the Depth-first search algorithm to traverse every leaf node. If the path of a leaf node is contained in one of the frequent k-tree-expressions which are discovered in the second step of XStorM mapping scheme, then the value of this leaf node shall be stored in the corresponding core relational table. If the path of a leaf node is not contained in a frequent k-tree-expression, then it is considered as *overflow* data and shall be stored in an *overflow* table. The overflow table names embed the XML structural information necessary for pattern matching queries and reconstruction of the XML document. More precisely, the name of a overflow table is defined as the root-to-leaf path of the overflow leaf node. For example, the overflow table name for the *author* object is *overflow_article.authors.author*, indicating the path in the XML graph. Note that there is an additional attribute, *objectID*, in the overflow tables that contains the identifier of the object to which these overflow data belongs. Apparently, we need to traverse the whole XML instance to store every leaf node into core or overflow relational tables. The cost of determining a leaf node's path is bounded by the height of the XML tree. The cost of the algorithm is therefore bounded by O(N + $mlogN$), where N is total number of nodes in the XML tree and m is the total number of leaf nodes.

3.3 Performance study

We implemented XStorm in Java and carried out a series of experiments on a PC with Pentium 3GHz CPU and 1GB RAM to evaluate the performance of it. The RDBMS we choose to use is Oracle 9i, which is generally considered as the leading DBMS product. Two metrics are used: the size of the relational tables generated and the response time of different classes of queries. We compared our storage scheme with the binary approach in [37], STORED in [32] and XRel in [103].

3.3.1 Experiment Setup

We created synthetic XML documents based on the ACM SIGMOD Record XML data for our experiments. The datasets have the following characteristics:

```
XML_To_RDBMS()
1.  For every object element obj
2.      Let p be the corresponding frequent k-tree-pattern for obj
3.      mapData(obj, p)

mapData(currentRoot, pattern)
1.  If currentRoot is a leaf node
2.  Then
3.      If currentRoot.path exist in pattern
    /* this leaf node should be mapped to core relational table
4.      Then
5.          add currentRoot.value to the table record of obj
6.      Else
    /* this leaf node should be mapped to overflow table */
7.          Let table_name = “overflow_” + currentRoot.path
8.          If table_name exists in database
        /* corresponding overflow table exists, insert data only */
9.          Then
10.             add currentRoot.value to corresponding overflow table
11.         Else
        /* corresponding overflow table does not exist */
12.             create new overflow table with table_name
13.             add currentRoot.value to the new overflow table
14. Else
15.     For every child i of currentRoot
16.         mapData(i)
```

Figure 3.9: Algorithm to map XML data to Relational DBMS

- Five sets of XML documents with varying sizes are used: 1MB, 10MB, 20MB, 40MB, and 100MB. They contains 1274, 11500, 22890, 44200, and 113754 articles, respectively.

- Each article has the following information: issue number, title, starting page, ending page, a number of authors, and a short description. The number of authors varies from 1 to 17 and majority (over 80%) of the articles have 2 or 3 authors.

Query	Description	Figure
Q1	Retrieve information to reconstruct XML object	Select by object id
Q2	Find objects that have attribute a_1 with value in certain range	Select by value
Q3	Find objects that have attributes a_1 and a_2 with certain values	Two predicates
Q4	Find objects that have a_1 and a_2 with certain value or just a_1 with certain value	Optional predicates
Q5	Find objects that have a_1 or a_2 or a_3 with certain value	Predicate on attribute name
Q6	Find objects that match a certain pattern	Pattern matching

Table 3.3: Benchmark query templates

Table 3.3 describes the query templates used in our experiments. These query templates test a variety of features, including simple selections by object identifiers (*oid*) and attribute values, optional predicates, predicates on attribute names and pattern matching. All the attribute values are stored as strings in the database. For predicates involving numerical comparisons, we convert string values to numbers using the *to_number*() function provided in SQL. We translate all queries into corresponding SQL queries, which are shown in the appendix of this thesis. We shall note that in XRel Scheme, an element is uniquely identified by its document id and start position in the document, while in other three schemes, an element is identified by its object id which is assigned automatically by the system.

In order to obtain reproducible experimental results, we carry out all the benchmark queries as follows: Each query is run once to warm up the database buffers and then ten times subsequently to get the average response time of the query. Warming up the buffers will have an impact on queries that operate on data which fits in the main memory although queries with heavy workload are not likely to be affected.

3.3.2 The impact of frequent *k*-tree-patterns identified

First of all, we study how the frequent k-tree-patterns, which are identified in step 2 of Section 3.2.2, affect the resulting relational schemas. We use the 100MB dataset as described in Section 3.3.1 in this set of experiments. By choosing different thresholds as indicated in Figure 3.7, we get a number of k-tree-patterns, which are then converted to core relational schemas with different number of *author* attributes.

To study how the *threshold* value affects the storage space, we measure two values in the experiment. One is the overflow table size, the other is the space wasted in core relational table, which is caused by *author* attributes with empty value. Figure 3.10 gives the results. We can clearly see that the size of wasted space increases as the *threshold* value decreases, while in the mean time, the sizes of overflow tables decrease. The reason is that if we choose a large *threshold* value, we will get a k-tree-pattern with a small k value, which means that few space in the core relational table is wasted. However, a large number of *author* attributes are then stored in overflow tables. If we choose a small *thresholdvalue*, then a lot of spaces in the core relational table are wasted because a lot of *author* attributes are empty. But in this case, the size of overflow tables will also be small because the core relational table is able to cover most of the data.

Figure 3.11 shows how different *threshold* values affect the response time of queries involving overflow tables. In this experiment, we choose query Q3 and Q4 in Table 3.3 because they are typical queries that involve joins between core relational table and overflow tables. We can see from the results that as the *threshold* value decreases, the query response time for both Q3 and Q4 decreases as well. This is because that we get smaller overflow tables as the *threshold* value decreases, which has been elaborated in the last paragraph.

Generally speaking, choosing small *threshold* value helps increase query efficiency because most data can be stored in the core relational table. However, a large k-tree-pattern identified by choosing small *threshold* value will cause significant waste of disk space. In the given 100MB dataset of our experiment, we can see that when the *threshold* value is set to be 65%, we can strike a balance between the size of wasted space and the size of overflow tables. Meanwhile, the decrease of query performance is also acceptable. In this case, the k value is three, which means we store three *author* attributes in the core relational table.

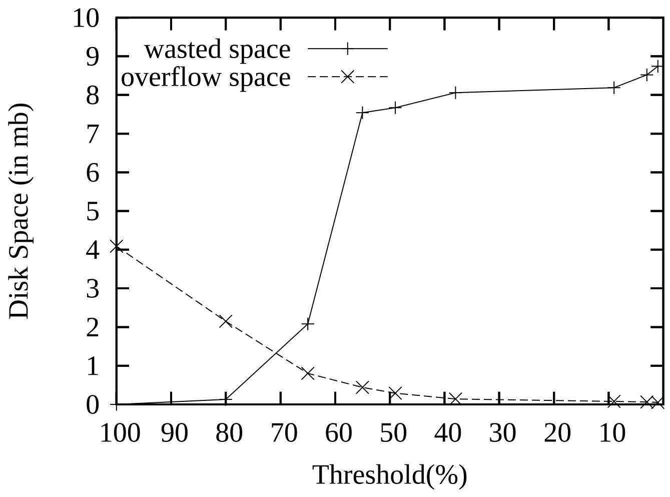


Figure 3.10: Resulting disk space by varying *threshold* value

3.3.3 Storage Requirements

We investigate the amount of storage required by the various mapping schemes to store XML data in this sub section. Table 3.4 shows the size of the XML document and the resulting relational databases. The binary and XRel approach produce much larger relational databases compared with STORED and XStorM. Indeed, the binary and XRel approach require double of the original document size for storage, while STORED and XStorM require a storage space of about 70%-80%

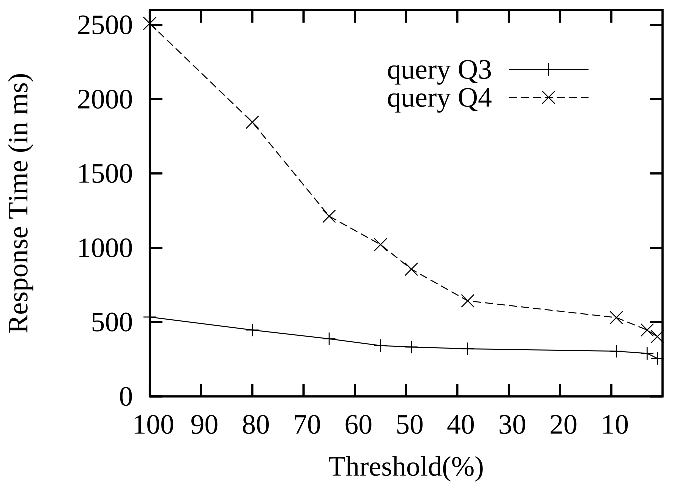


Figure 3.11: Resulting query response time by varying *threshold* value

of the original XML documents. The high storage space requirements by the binary and XRel approach are because that they store the structural information for each object, even when most objects have similar structure. On the contrary, both STORED and XStorM only store common structures extracted from the documents. The XRel scheme requires slightly more space than the binary approach because in XRel, values of elements are stored in separate "Text" tables, which in general, incurs more overhead. While both STORED and XStorM use about the same amount of storage for the base data, the overflow data obtained in XStorM is much smaller than in STORED. Furthermore, XStorM keeps both the base data and overflow data in the relational database instead of using auxiliary data structures as in STORED. Apart from consistency, keeping the overflow data in the database is amenable to faster reconstruction.

3.3.4 Query Response Time

Query running time is a very important measure when evaluating these mapping schemes. We will elaborate the results in the following subsections.

	XML(MB)	Binary(MB)	XRel(MB)	STORED(MB)	XStorM(MB)
			dataset 1		
Base Data	1.1	2.25	2.41	0.84	0.84
Overflow Data	-	-	-	0.1	0.01
Total	1.1	2.25	2.41	0.94	0.85
			dataset 2		
Base Data	10.1	20.2	21.4	7.6	7.6
Overflow Data	-	-	-	0.85	0.08
Total	10.1	20.2	21.4	8.45	7.68
			dataset 3		
Base Data	20.2	40.3	41.2	15.2	15.2
Overflow Data	-	-	-	1.7	0.16
Total	20.2	40.3	41.2	16.9	15.36
			dataset 4		
Base Data	40	82.7	84.6	30.4	30.4
Overflow Data	-	-	-	3.4	0.32
Total	40	82.7	84.6	33.8	30.72
			dataset 5		
Base Data	100	202.2	204.73	77.4	77.4
Overflow Data	-	-	-	7.4	0.8
Total	100	202.2	204.73	84.8	78.2

Table 3.4: Comparison of database sizes generated by different schemes

Retrieve information to reconstruct XML document

This experiment queries information needed to reconstruct an XML object from the relational database. This query involves selection by object id or element position (for XRel scheme) and for every table, there are indexes build on object id or element position(e.g., oid, source) column because it is the primary key or part of the primary key. Figure 3.12 shows the results. We first observe that the performance of XRel is worse than other three schemes in most cases. XRel performs worse than the binary scheme is mainly because XRel stores elements and values separately, thus more join operations are required to form the final answer. Another reason is that generally speaking, inequality comparison is less effective than equality comparison in RDBMS. In most cases, there is no significant difference among binary, STORED and XStorM mapping schemes since index lookup is very fast in RDBMS. For the 100MB dataset, the binary scheme performs worse because it requires the retrieval of data from all the attribute tables compared with STORED or XStorM which retrieves data from just the core table and overflow graphs/tables.

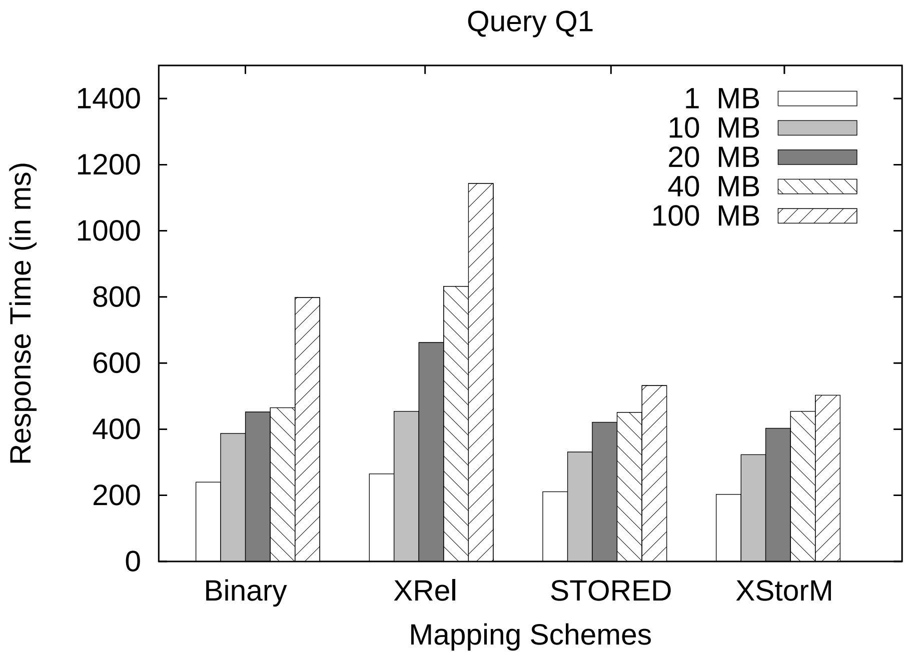


Figure 3.12: Results of reconstructing XML document experiment

Selection queries

In this experiment, we tested queries to retrieve objects that have attribute with values in a certain range. An example of the query used is

Query : Select articles that have "initpage" between 500 *and* 600.

The results are shown in Figure 3.13. There is no significant difference among all four mapping schemes except for the 100MB dataset. XRel performs slightly worse than other schemes as it is the only scheme that requires join operations. For other three schemes, no join operation is needed to answer the query. For the 100MB dataset, the binary scheme performs better since its single attribute table is much smaller than the core tables in the STORED scheme and XStorM scheme (8MB vs 77MB).

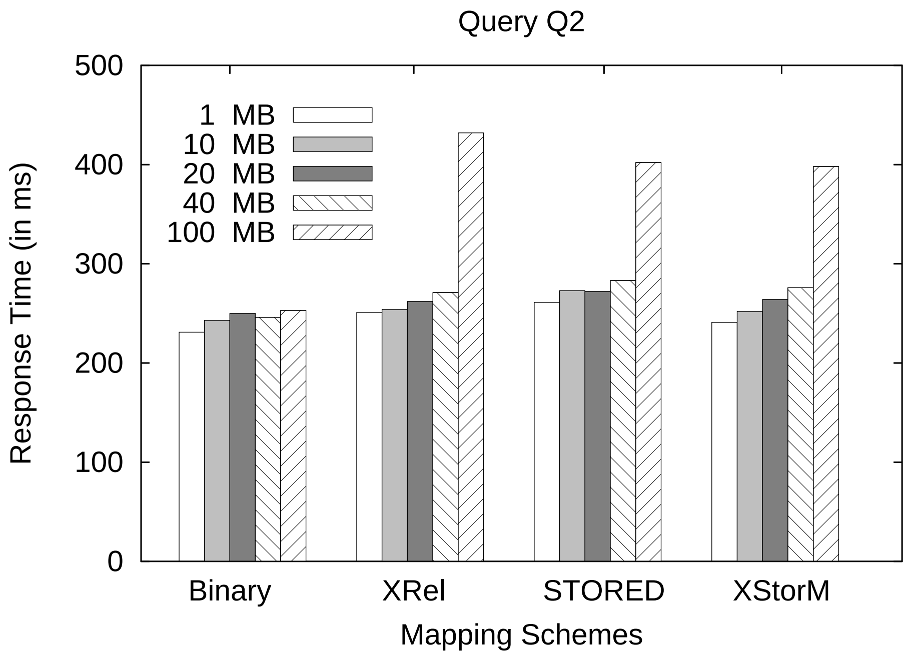


Figure 3.13: Results of selection query experiment

Join Queries

This experiment examines the situation where multiple predicates are involved in a query. A typical query is shown below.

> *Query: Select articles with issueNumber 15 and whose 10th author is 'Pinar Koksal'.*

Note that one of the predicates involves overflow attribute for the STORED and

XStorM schemes. Figure 3.14 shows the results of this experiment. In order to make the bar chart easier to read, we set the upper bound of y axis to 6000. The same setting is also applied to Q4 and Q5. The exact response time of the binary and XRel schemes for the 100MB dataset can be found in the appendix and they are in fact over fifty times of that of XStorM. The differences become large enough to claim that XStorM performs the best among all four schemes when size of datasets increase. The reason is simple: joining a much smaller overflow table is much faster than joining two large attribute tables (or large element and or text table) as required in the binary approach (or XRel scheme). The STORED scheme performs poorly in the 100MB dataset because searching for values in overflow graphs is getting more and more inefficient as the data size increases.

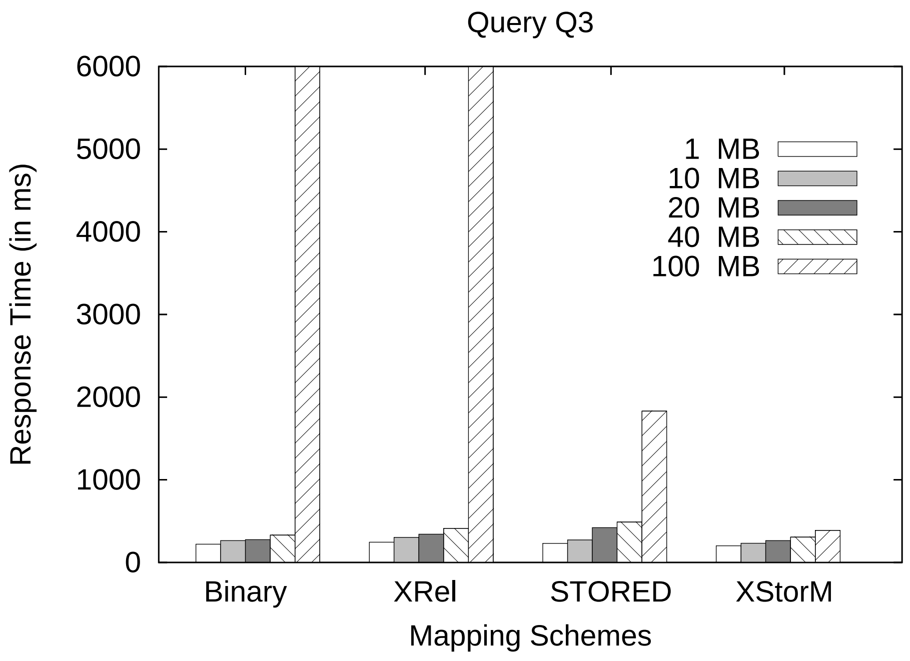


Figure 3.14: Results of join query experiment

Queries with optional predicates

We tested the performance of queries containing optional predicates and predicates involving overflow data on the various storage mapping schemes. For instance,

Query: Select articles that have first author 'Dallan Quass' AND 7th author 'SvetlozarNestorov' OR just first author 'Kenneth A. Ross'.

Figure 3.15 contains the results of the experiment. XStorM again has the best overall performance. The response time of the binary and XRel scheme increases rapidly as the dataset size increases and becomes over forty times of that of XStorM scheme for the 100MB dataset. Similarly, STORED performs poorly for the 100MB dataset for the same reasons given in join query experiments,i.e.,Q3.

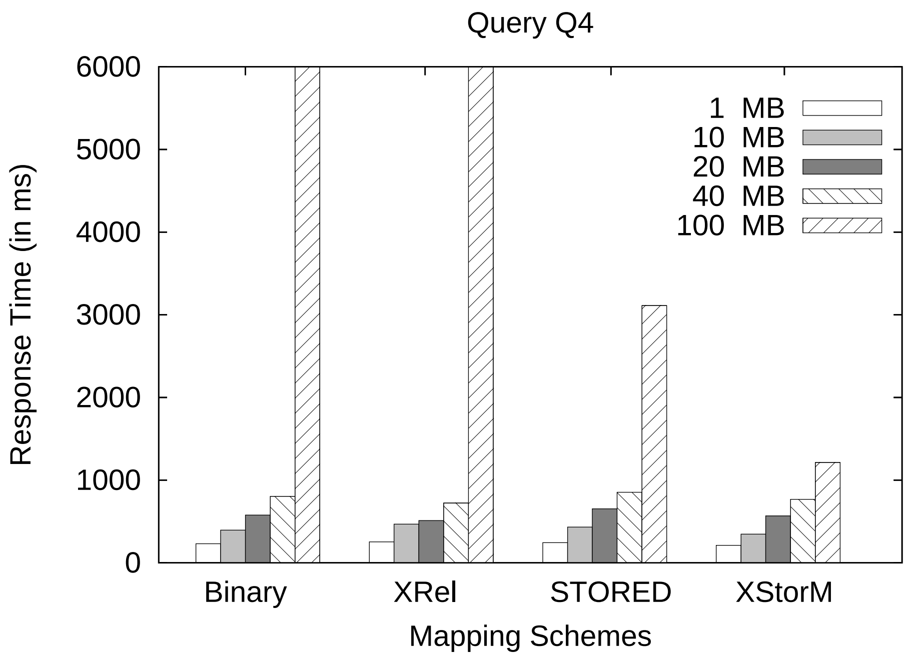


Figure 3.15: Results of optional predicate query experiment

Queries with attribute predicates

This experiment evaluates the performance of queries involving predicates on attribute names. Figure 3.16 shows the results when the following query is issued.

> *Query: Select articles that have initpage = 388 or endpage = 2 or 7th author 'Svetlpzar Nestorov'*

For this query, differences among all schemes become obvious in the 100MB dataset. The binary scheme performs poorly because we need to search matching tuples in three attribute tables and then take the union of the tuples returned. The XRel scheme performs even worse as it requires additional join operations between element table and value table. Although the STORED scheme only searches one table and XStorM searches two tables, XStorM performs better than STORED because accesses to the overflow graphs in STORED takes more time than retrievals from the overflow tables in XStorM, especially in the case of 100MB dataset.

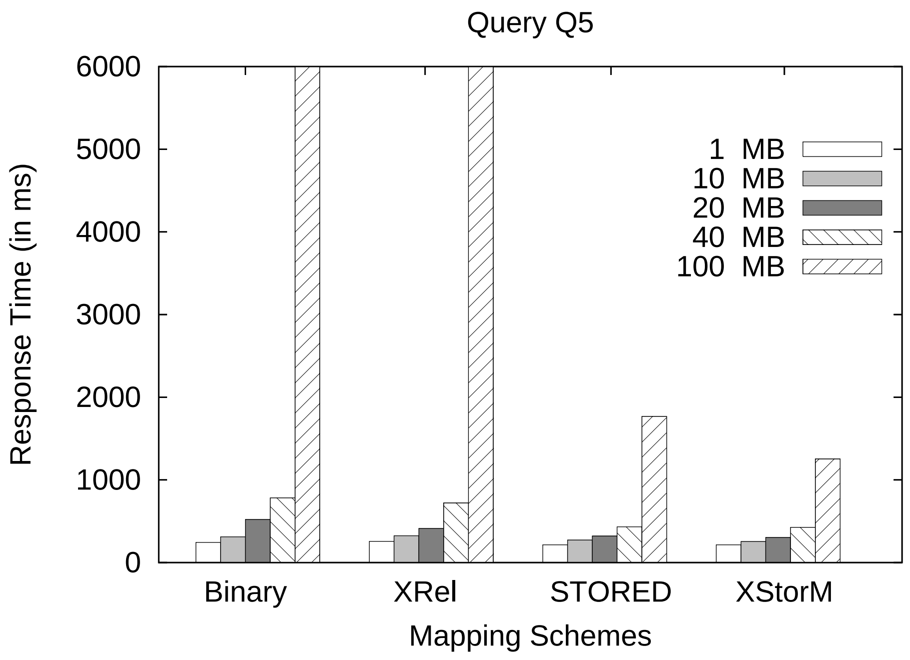


Figure 3.16: Results of query with attribute predicates experiment

Pattern matching queries

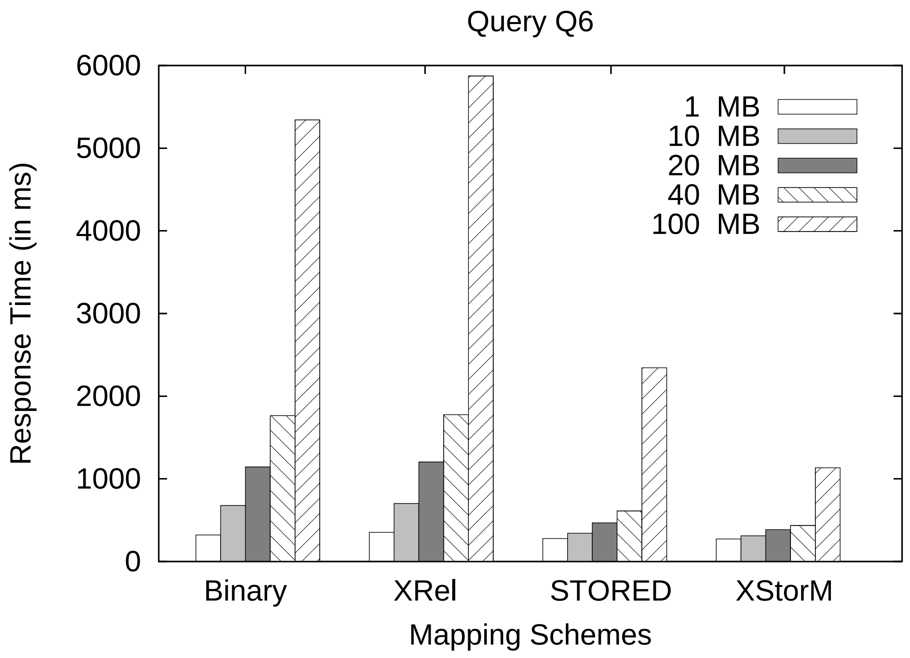


Figure 3.17: Results of pattern matching query experiment

Figure 3.17 shows the performance results of the following pattern matching query:

Query: Select articles that have attributes issuenumber, title, initpage, and 9 authors.

Again, XStorM has the overall best performance. The binary scheme performs poorly in this query because it needs to join many attribute tables to find the matching tuples, especially when the pattern involves many attributes. The same reason applies to the XRel scheme as well. The STORED scheme performs much better than the binary scheme as most of the attributes are contained in the core table. But retrieving data from disk-resident overflow graphs is still slower than retrieving data from database tables in most cases.

Discussion

Theoretically, schemes that map XML data into relational model based on schematic information should work better than just storing XML data in attribute tables. The most expensive operation in query processing is join operation and to answer most queries, we need information about several attributes of an object. In the binary scheme, if we want information from several attributes, we have to join the corresponding attribute tables to form the query result. If the attribute tables are large, the join operation will be expensive. The situation is even worse for the XRel scheme as it has only one single element table and additional joins with a single value table are often required to process a query. On the other hand, storing XML data according to the schema not only saves disk space, but also reduces the number of join operations needed to answer a query.

For example, let us consider the query to find objects with attributes a_1 and a_2 with certain values. For binary scheme, to answer this query we need to join two attribute tables: table a_1 and table a_2. For the XRel scheme, we need to join the element table “Element” with itself and with value table “Text”. On the other hand, STORED and our proposed scheme, XStorM, only search one table for tuples that satisfy the selection condition. While in most cases selection operation is much faster than the join operation, there are situations that involves overflow data. Suppose attribute a_2 is not included in the schema of the table. In this case, we will need to join the core table with the overflow table that stores a_2 to get the complete answer. The cost of this join operation is tolerable because overflow tables are typically much smaller than the core table. In addition, storing overflow data in relational tables have a better query performance than storing overflow data in external data structure. The reason is because relational databases have very powerful query optimizers that will find the optimal plan for most queries.

If we store overflow data on external data structure, we cannot make use of this conventional tool and have to find a way to efficiently retrieve and update them. Furthermore, if the size of overflow data is too large to fit the main memory, then we have to fetch them from local disk, which is also a time consuming process.

When the XML dataset is small, for example, 1MB, we observe that there is not much significant difference in the query response time for all three schemes. However, when the sizes of datasets increase, the performance gain in XStorM becomes obvious. The experimental result on the 100MB dataset clearly presents the advantage of XStorM scheme in term of scalability.

XRel is mainly designed to efficiently process queries involving long or undetermined paths. When there's few such paths involved in the query, the performance of XRel is likely to be worse than the binary scheme. XRel stores elements and values separately and all elements are stored in a single large element table while [37] has already shown that this scheme is less efficient than the binary scheme. Moreover, inequality comparisons are also involved when we perform self-join operation on the element table and join operations between element and value table, which are also less efficient than equality comparisons in most cases.

3.4 Concluding Remarks

In this chapter, we have examined how XML data can be stored using a relational database. The semistructured feature in XML introduces complications in the mapping. Our proposed scheme, XStorM, overcomes the problems by making a distinction between XML elements that represent entities in the real world, i.e., objects, and XML elements that represent properties of entities, i.e., attributes. A breadth-first-search algorithm is used to identify objects in the XML data.

XStorM avoids excessive fragmentation of XML data by mapping each object together with the majority of its attributes to a core relational table. A data mining algorithm is exploited to find frequent patterns in the XML dataset. These frequent patterns are used to generate the core relations for objects. Irregularities or data instances that deviate from the core schemas are stored in separate overflow tables. The names of these overflow tables also contain the structural information of the XML document for fast reconstruction. Our performance study has demonstrated that XStorM gives good query performance, minimizes storage space and is scalable.

Chapter 4

The XJoin Index

4.1 Introduction

Efficient query evaluation is the core operation of XML document processing. Different from traditional SQL query designed for relational data, a typical query for XML documents specifies not only selection predicates for elements/attrbutes, but also specifies the structural relationship between these elements/attributes. If the graphical representation of the query is not a trivial tree(a null tree or a single linked list), then we call the query a *branching path query*. A branching path query consists of two parts, one is the *value predicate(s)*, and the other is the structural components which we call *branching path expression*. Let's take the query: `book[@title = 'Databases' AND @publisher = 'Springer Verlag']/author[@name = 'John']` as an example. In this query, the '=' signs represent *value predicates*, and the structural information can be seen in Figure 1.3. To solve this query, we need to match those `author` elements whose `name` attributes are of value `John` and that are children of `book` elements, whose `title` attributes and `publisher` attributes have value `Databases` and `Springer Verlag`, respectively. A *branching path query* is also know as a *twig* query.

Besides the traditional approaches which make use of summary index to quickly scan the whole data tree, *structural join algorithms* are used to first evaluate binary path expressions, i.e., those path expressions of the form $A//D$, or A/D, in the query and then the results for these binary path expressions are merged to form the final answer. Quite a number of index structures have been proposed to speed up the structural join process by the research community in recent years.

We note that such indexing techniques do not vary the number of structural joins to be executed. Instead, they aim to make each individual structural join more efficient. On the other hand, the *join index* proposed in relational context pre-joins some relations and makes the joins more efficient by the semi-joined results. The idea of trading space occupancy for query efficiency inspired us to build a *join index* in the XML context.

In this chapter, we present our approach to indexing XML data for solving twig queries with the aim of reducing the number of joins to be executed. We propose a simple yet efficient join index (*XJoin Index*) approach to shrink the twigs before applying structural join algorithms. The proposed XJoin Index pre-computes some (semi-) join results that help reduce the number of structural joins to be executed. These pre-computed joins correspond to both value and structural information and support following operations: attribute selections involving multiple attributes, parent-child relationship detection and counting selections such as *Find all books with at least 3 authors*. The XJoin Index is simple as it is completely based on B^+-tree and no specialized data structures are required. The XJoin Index is also flexible since it can be coupled with other structural join algorithms and several execution plans can be defined based on the usage of the XJoin Index.

The rest of this chapter is organized as follows. Section 4.2 introduces the concept of branching path expressions in detail, extended with counting predicates. Section 4.3 presents the structure of the XJoin Index. We show that even if some element information is replicated in the XJoin Index, the space occupancy is still linear with respect to the total number of elements/attributes in the original XML documents. Search and update operations of the XJoin Index are presented in Section 4.4. We show that to solve attribute or counting predicate inside a query conditions, we do not need to perform structural join operations. Instead, we only need simple selection and set intersection. In Section 4.5, we show how different query plans can be defined based on the XJoin Index. Such plans differ in the number of joins to be executed. Experimental results are shown in Section 4.6, The results show that by using the XJoin Index, we can process twig queries much faster than traditional index approaches. We conclude this chapter in Section 4.7.

4.2 Preliminaries

In this section,we provide and recap on the background information about XML document model and also some basic concepts about XML branching query expression, which is essential to describe our XJoin Index.

4.2.1 XML documents

We assume each XML document is represented as a rooted, ordered and labeled tree where each node corresponds to an element, an attribute, or a value. Edges represent ancestor-descendant, element-subelement, element-attribute, element-value, or attribute value relationships.

We assume that a position is assigned to each element or attribute node according to some labeling scheme. In the following, we assume to use a numbering scheme that univocally identifies each node and that supports the detection of ancestor-descendant relationships, for example, the Dietz's or the Dewey labeling schemes. We use the labeling scheme to assign a unique identifier to each node, that we call *position*. In this chapter, for the sake of simplicity, we assume that the XML dataset corresponds to a single document, eventually generated by adding a dummy root to all documents in the document collection. This limitation can be easily overcome by inserting the identifier of the document the element or the attribute belongs to as part of their position.

4.2.2 Branching path expressions

We consider a subset of branching path expressions according to the syntax present in Figure 4.1. With respect to other approaches dealing with branching path expressions, we consider an additional predicate that allows us to define conditions over the number of child-elements of a given node, with a certain name. Examples of these queries are *Find all books with 3 authors, Find all companies with at least 100 employees*, etc. We call these predicates *counting predicates*. While no other technique have been proposed to deal with counting queries, they are quite relevant in all XML applications that require computations, for example, in XML data warehousing applications. Counting queries can be represented in XPath as follows:

```
books[author][last() = 3]
company[employee][last() >= 100].
```

For the sake of simplicity, in the following part of this thesis, $a[b][last()\theta n]$ is simply denoted by $a[b(\theta n)]$.

According to our definition, a branching path expression can always be seen as a sequence of *base expressions*, connected by '/' or '//'. Base path expressions represent conditions applied to elements and can be classified as follows (in the

```
V:= value strings

A:= attribute names

E:= element names

N:= natural numbers

OP := >|<|=|>=|<=

ACOND:= @A|@A=v

ECOND:= E|E(OP N)| E = v

COND:= ACOND|ECOND|COND AND COND|COND OR COND

CPE:= E|E[COND]|CPE/CPE|CPE//CPE
```

Figure 4.1: Syntax of branching path expressions

following a, b, and e are element names, c and d are attribute names, and v and u are values):

- *Simple selections*: they represent all expressions generated by $ECOND$ and $ACOND$ productions, excluding the expressions of the form *E(OP N)*. a,$a = v$, $@c$, $@c = v$ are examples of simple path expressions. The result of evaluating a simple path expressions is the set of elements (attributes) with tag name $a(c)$, satisfying the equality condition, if required.

- *Attribute selections*: they represent expressions generated by $E[ACOND]$ production. $a[@c]$, $a[@c = v]$ are examples of attribute expressions. Expressions inside brackets are called *attribute predicates.* The result of evaluating an attribute expression is a set of elements which satisfy the attribute predicate.

- *Counting selections*: they represent expressions generated by the $E[ECOND]$

production. $a[b(=5)]$ and $a[b(>6)]$ are examples of counting selections. Expressions inside brackets are called *counting predicates.* The result of evaluating a branching selection is a set of elements which satisfy the counting predicate.

- *Complex selections*: they represent expressions generated by the $E[COND]$ production such that $COND$ contains at least two predicates. $a[@c = d$ AND $b(<6)$ OR $b(=4)]$ is an example of a complex selection. Expressions inside brackets are called *complex predicates.*

We then call *navigational expressions* those expressions generated by productions E/E and $E//E$. The first is called *direct navigation*, the second *indirect navigation.* a/b, and $a//b$ are examples of navigational expressions. The result of evaluating a navigational expression is a set of pairs (n, m) where n is the parent(in the direct case) or the ancestor (in the indirect cast) of m, and n is an element with tag name a, m is an element with tag name b. Each base path expression, except simple ones, represent semi-join operations whereas navigational expressions correspond to join operations. In both cases, relations to be joined are sets of element/attribute positions with a certain tag name and the join predicate imposes a structural relationship between them.

According to our definition, a branching path expression can always be represented as a tree, where each node represents either an element tag, an attribute tag, or a value. This tree is also called *twig.* Conditions like $b(\theta_n)$ are represented in the tree with a node labeled b^{θ_n}. Edges represent (semi-)join operations, corresponding to ancestor-descendant, element-subelement, element-attribute,element-value, or attribute value relationships. Given an XML document D and a branching path expression Q, the result of evaluating Q against D is a mapping from the nodes of Q to nodes in D such that content and structural relationships between query nodes are satisfied by the corresponding document nodes.

4.3 XJoin Index: the Structure

In this thesis, we are interested in defining an index structure for efficiently supporting the evaluation of base expressions. The proposed index structure, that we call *XJoin Index*, pre-computes some (semi-) joins, similarly to the join index defined in the relational context [94]. Any index access corresponds to directly solving one relationship represented as an edge inside the query tree. Thus, it allows the query processor to shrink the twig before applying the structural join algorithm. The index is flexible enough to support a large variety of query processing strategies, that we discuss in more details in Section 4.5.

In the relational context, the join index over two relations R and S is the set

$$JI = \{(r_1, r_2) | f(t_1.A, t_2.B) \ is \ true\}$$

where t_i is a tuple identified by r_i, $i = 1, 2$ and f is a boolean function that defines the join predicate against attribute A and B. A join index is created for any join operation of interest. Typically, since we may need fast access to JI tuples via either r values or s values depending on whether there are selections on relations R or S, a JI should be clustered on (r, s). One possible solution is to maintain two copies of the JI, one clustered on r and the other clustered on s.

In the XML context, as we pointed out before, a join Index over two sets E -a set of element positions - and F - a set of either attributes or element positions - can be defined as the set:

$$XJI = \{(r_1, r_2) | f(r_1, r_2) \ is \ true\}$$

where f is a predicate requiring some tag name for r_1 and r_2 and some structural relationship between them. Thus, in principle, an XJI has to be created for each triple $(tag_1, tag_2, relationship)$ of interest. In our approach, we consider two relationships (parent-child and element-attribute) and we create an index for each relationship, maintaining information concerning all the possible combinations of tag names.

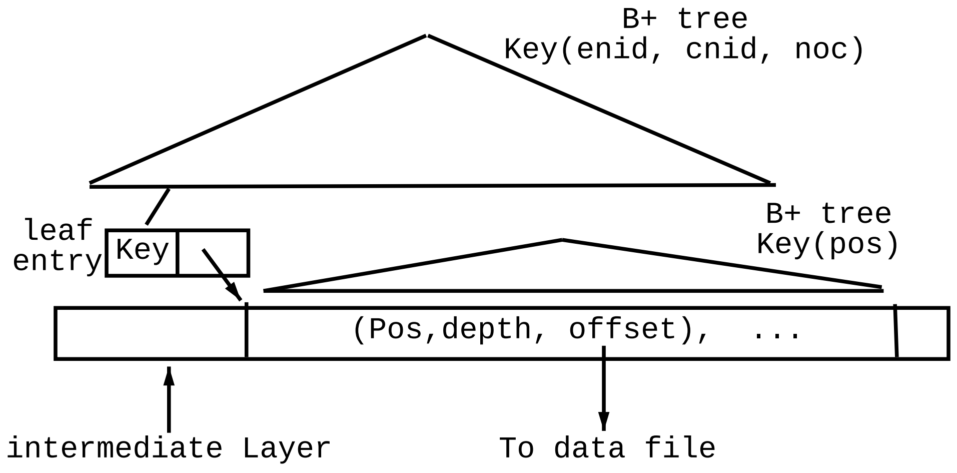


Figure 4.2: XJoin Index: Structure

The structure of the XJoin Index is composed of the following indexes (see Figure 4.2 , and in the following, E denotes the set of element positions, EN the set of element names, EV the set of value identifiers, A the set of attribute positions, AN the set of attribute names, and AV the set of attribute value identifiers.):

Name Index. Given a set of XML documents, all distinct name strings, corresponding to element and attribute names, are collected in the name index. Each distinct name string is identified by a unique name identifier (*nid*) returned by the name index. Using the name index greatly reduces the storage space of the whole index structure and also reduces the computational cost of string comparison in query processing.

Value Index. Since all value entities in XML data are considered as variable-length character strings, attribute and text values appearing in XML documents are collected into the value index and each string value is assigned an identifier(vid).

Element Index (EI). The element index allows one to quickly retrieve all elements with a given identifier; thus it directly supports the execution of simple path expressions over elements. It is implemented as a B^+-tree whose keys correspond to EN values. Each key value $enid$ in the leaves points to a list of tuples $(epos, evid, cnus)$, where $epos \in E$ is the position of an element named $enid$ with value $evid$. $cnum$ is the total number of children of the element in position $epos$. The lists are ordered with respect to the element positions and a B^+-tree index can be created over each list.

Attribute Index (AI). The attribute index allows one to quickly retrieve all attributes with a given identifier; thus it directly supports the execution of simple path expressions over attributes. It is implemented as a B^+-tree index whose keys correspond to AN values. Each key value $anid$ in the leaves points to a list of tuples $(apos, avid)$, where $apos \in A$ is the position of an attribute named $anid$ with value $avid$. The list is ordered with respect to attribute positions and a B^+-tree can be created for each list to make the access faster.

Counting XJI(CXJI). It is an XJI over the parent-child relationship, supporting counting expressions. It is organized as a B^+-tree with keys $(enid, cid, noc), enid \in EN, cnid \in EN, noc \in N \setminus \{0\}$. Each key value $(enid, cnid, noc)$ in the leaves points to a list of element positions corresponding to elements having $enid$ as tag name and at least *noc* children with *cid* as tag name. Note that only information concerning elements with at least one child are inserted in the index. The lists are ordered with respect to the element positions and a B^+-tree index can be created over each list to make the access faster.

Attribute XJI (AXJI). It is an XJI over the element-attribute relationship, in order to support attribute expressions. It is organized as a B^+-tree with keys $(enid, anid), enid \in EN, anid \in AN$. Each key value $(enid, anid)$ in the laves points to a list of tuples $(epos, avid), epos \in E, avid \in AV$, such that element with position *epos* has an attribute named *anid* with value *avid*. The lists are ordered with respect to the element positions and a B^+-tree can be created over each list to make the access faster.

It is important to point out that lists pointed by the various index leaf levels (also called *intermediate layers* in the following) are not disjoint. In particular, element positions are replicated in EI, CXJI, and AXJI. However, it is easy to prove that the overall space is still linear in the maximum between the total number of elements and the total number of attributes of the XML dataset.

Proposition 4.1 *The space occupancy of the XJoin Index is linear in $O(max(||E||, ||A||))$, where $||E||$ and $||A||$ denote the number of elements and attributes, respectively.*

Proof. It is immediate to prove that EI space occupancy is linear in $||E||$ and AI space occupancy is linear in $||A||$. The size of CXJI and AXJI is dominated by the size of their corresponding intermediate layer. The intermediate layer of AXJI is bounded by the total number of attributes whereas the intermediate layer of CXJI is bounded by the total number of elements. From these considerations, it follows that the space occupance of XJoin Index is linear in $O(max(||E||, ||A||))$. $\square$

4.4 XJoin Index: operations

In the following, we discuss search and update operations for the XJoin Index.

4.4.1 Search

In the following we discuss how simple, attribute and counting selections, as well as navigational expressions, can be executed in the XJoin Index. Complex selections are not discussed since their evaluation corresponds to executing a set of attribute or counting selections and then intersecting the obtained results.

Simple Selection. Simple path expressions are solved directly by using EI or AI, depending on the specific query. In case the selection is of type $@a = v$ or $e = v$, the list associated with a in AI or with e in EI has to be scanned to find values equal to v. In all the other cases, the index access returns the exact result.

Attribute Selections. Two different expressions have to be considered:

- $e[@a]$: we look for (e, a) in AXJI and we get the exact result.
- $e[@a = v]$: we look for (e, a) in AXJI. Then, the list associated with (e, a) has to be scanned to find values equal to v.

Counting Selections. Let $e_1[e_2(\theta n)]$ be the considered selection. We look for (e_1, e_2, n) in CXJI. Then, the exact result is retrieved according to the usual B^+-tree properties.

Navigational Selections. Let e_1/e_2 or $e_1//e_2$ be the navigational expression. The XJoin Index can also solve direct navigational expressions as a semi-join by using CXJI. Both direct and indirect navigational expressions can also be executed as a join. In this case, CXJI can be used to reduce the size of the set of e_1 elements to be used in the structural join. More precisely, in this case we look for $(e_1, e_2, 1)$ in CXJI to find elements e_1 with at least one e_2 child. Finally, we look for (e_2) in EI to find all e_2 elements. Then, we apply a structural join algorithm to the obtained results.

ollowing proposition gives the cost for solving the above simple sections.

Proposition 4.2 *Base expressions are supported by the XJoin Index with the following costs:*

Simple selections on elements: $O(log_F||EID|| + R/B)$

Simple selections on attributes: $O(log_F||AID|| + ||a||/B)$

Attribute selections: $O(log_F(||EID_A * \tilde{A}_{par}||) + N^a_{(e,a)}/B)$.

Counting selections: $O(log_F(||EID_C * \tilde{C}_{par}||) + N^c_{(e_1,e_2)}/B)$.

Navigational selections: $O(log_F(||EID_C*\tilde{C}_{par}||)+N^b_{(e_1,e_2)}/B+log_F||EID||+||e_2||/B+ C(Sj))$.

In the previous formulas:

- *N is the number of elements indexed; F is the fanout of the used index; R is the output size; B is the average number of element entries in each leaf page of the used index;*

- *$C(Sj)$ is the cost of applying a structural join algorithm to the retrieved lists;*
- *$||EID||$ and $||AID||$ are the number of distinct element and attribute identifiers, respectively;*
- *$||EID_C||$ and $||EID_A||$ are the number of distinct identifiers for which there exists at least one element with a child or with an attribute, respectively;*
- *$\tilde{A}_{par}$ and $\tilde{C}_{par}$ represent the average number of attributes or child element names, associated with elements having a certain name.*
- *$N^a_{(e,a)}$ is the number of elements e having an attribute called a.*
- *$N^c_{(e,a)}$ is the number of elements e having a child called a.*

4.4.2 Update

We consider four main update operations for the proposed index structure (updates can always be implemented as a deletion followed by an insertion):

- $Insert_A(enid, epos, anid, avid)$: an attribute named *anid*, with value *avid*, is associated with the element named *enid* in position *epos*.
- $Delete_A(enid, epos, anid)$: the attribute named *anid* associated with the element named *enid* in position *epos* is deleted.
- $Insert_E(enid, epos, evid, pnid, ppos)$: an element name *enid* is inserted in position *epos*, with value $evid, ppos$ is the position of the parent element and *pnid* the corresponding identifer.
- $Delete_E(enid, epos, pnid, ppos)$: the element named *enid* at position *epos* is deleted. *ppos* is the position of the parent element and *pnid* the corresponding identifier. The deleted element must not have nested elements.

Note that the update operations we are going to present should not be confused with updates of XML source documents, for which the tree is built. Such a problem is still an open issue.

Algorithms to implement the previous operations are rather straightforward since they correspond to insertion (deletion) in (from) B^+-trees. Figures 4.3 and 4.4 present the code for element insertion and deletion. In both cases, both EI and CXJI have to be updated, in order to insert(delete) the new element and update counting information accordingly.

```
Insert_E(enid,epos,evid,pnid,ppos):
Modify EI:
   insert  (enid), if it does not exist yet
   insert  (epos,evid,0) in the list
        associated with enid
   look for ppos in the list
      associated with epnid in EI
   let (ppos,pvid,p) be the retrieved tuple
   let p := p+1
Modify CXJI:
   look for (pnid,enid,1) in CXJI
   look for (ppos) in the intermediate layer
      and get the noc value
   remove ppos from the current position
   IF (noc > 1)
        insert ppos in the list
        associated with (pnid,enid,noc-1) in CXJI
```

Figure 4.3: Insertion of an element

```
Delete_E(enid,epos,pnid,ppos):
Modify EI:
   look for (enid)
   delete  (epos,evid,0) in the list
        associated with enid
   look for ppos in the list
      associated with pnid in EI
   let (ppos,pvid,p) be the retrieved tuple
   let p := p-1
Modify CXJI:
   look for (pnid,enid,1) in CXJI
   look for (ppos) in the intermediate layer
      and get the noc value
   remove ppos from the current position
   IF (noc > 1)
        insert ppos in the list
        associated with (pnid,enid,noc-1) in CXJI
```

Figure 4.4: Deletion of an element

4.5 Query processing strategies based on the XJoin Index

Given a branching path expression, the XJoin Index can be used to reduce the number of structural joins to be executed, shrinking the tree corresponding to the query to be executed. Indeed, as we discussed before, any index access corresponds to directly solving one relationship represented as an edge inside the tree. We call this *shrinking operation.*

Four main types of shrinking operations can be applied to a twig by using the XJoin Index. Shrinking operations can be represented in the twig by assigning annotations to nodes. The possible shrinking operations are the following: (in what follows, $a, a_1, ..., a_n$ represent attribute names, $v, v_1, ..., v_n$ represent values, $e, e_1, ..., e_n$ represent element names):

1. *Attribute-value shrink.* An edge of type attribute-value (a, v) is removed. It corresponds to a simple selection that can be solved by an AI access. We assign annotation $p(a, v)$ to node a.

2. *Element-value shrink.* An edge of type element-value (e, v) is removed. It corresponds to a simple selection that can be solved by an EI access. We assign annotation $p(e, v)$ to node e.

3. *Element-attribute shrink.* An edge of type element-attribute (e, a_i) is removed, as well as the edge (a_i, v_i) from the attribute to its value, if any. This corresponds to an attribute selection, that can be solved by an AXJI access. We assign annotation $p(e, a_i)$ or $p(e, a_i, v_i)$ to node e.

4. *Element-sub-elements shrink.* An edge of type element-child element (e, e_i) is removed, as well as the edge (e_i, v_i) from the child element to its value, if any. This corresponds to a counting selection, that can be solved by a CXJI access. We assign annotation $p(e, e_i)$ or $p(e, e_i, v_i)$ to node e.

In case one node is associated with no annotation, it means that we have just to retrieve elements or attribute with that tag name, through either an EI or AI access. The annotation in this case is either $p(e)$ or $p(a)$.

The previous shrinking operations can be combined in different ways. We identify three main strategies:

1. *Weak Shrinking Strategy.* By using this approach, we apply only attribute-value and element-value shrinks to the tree. This strategy corresponds to the one proposed in [66] and it can be implemented by using only EI, AI, and a structural join algorithm.

2. *Strong Shrinking Strategy.* By using this approach, all possible shrinking operations are applied to the tree. Note that for nodes associated with more than one annotation, a(non structural) equi-join has to be applied to the intermediate results. In order to implement this strategy, we need the whole XJoin Index and a structural join algorithm.

3. *Medium Shrinking Strategy.* By using this approach, we apply all possible shrinking operations providing that at most one annotation is associated with each node. In this case, no equi-join has to be applied. Also in this case, the strategy can be implemented by using the whole XJoin Index and a structural join algorithm.

Figure 4.5 presents the tree obtained by shrinking the tree presented in Figure 1.3 and inserting the corresponding annotations, according to the three proposed strategies. The obtained trees lead to different costs in computing the result of the original branching path expression, since a different number of structural joins is required in each case. It is a task of the query optimizer to decide which strategy will get better results, according to existing statistics.

4.6 Experimental Results

4.6.1 Experimental setup

We implemented the XJoin Index in C, on an ultra450 machine running Solaris 8. It has the processor of 500MHz and 3 Gigabytes of memory. For the experiments, we used synthetic datasets, created by the IBM XML generator [6], benchmark datasets, generated by XMark [15], and real dataset such like DBLP [1]. The usage of synthetic datasets is motivated by the need of controlling the structure and

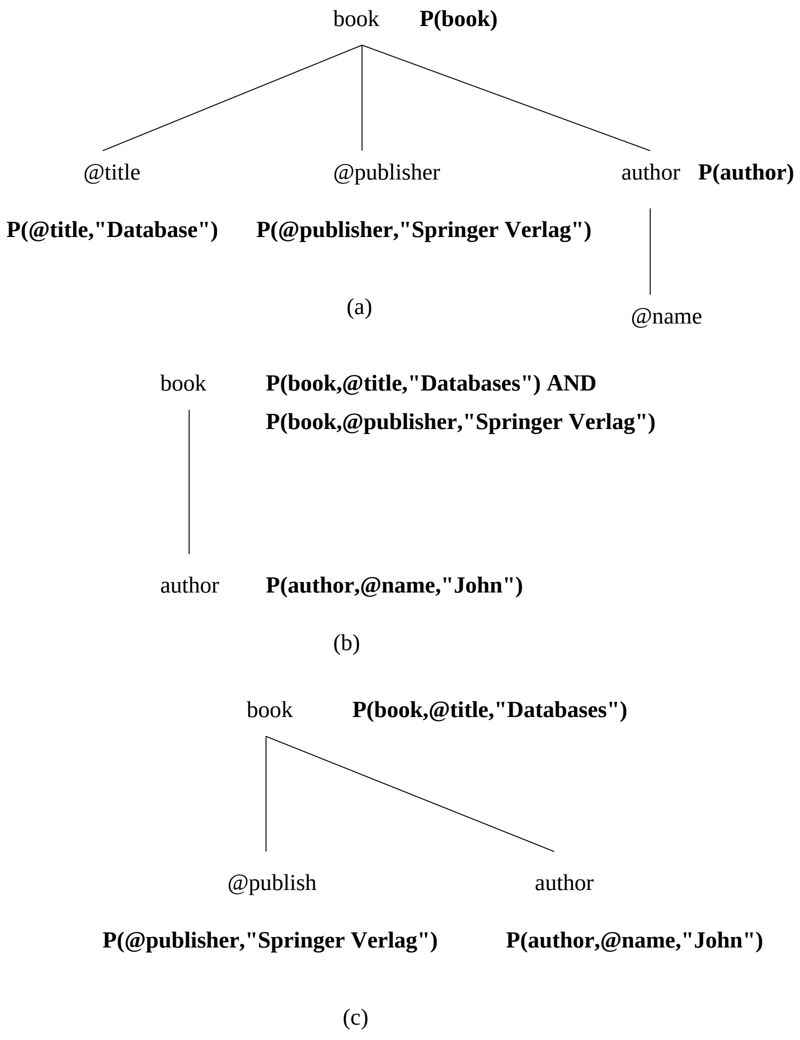

book P(book)
@title
P(@title,"Database")
@publisher
P(@publisher,"Springer Verlag")
author P(author)
@name
(a)
book P(book,@title,"Databases") AND
P(book,@publisher,"Springer Verlag")
author P(author,@name,"John")
(b)
book P(book,@title,"Databases")
@publish
P(@publisher,"Springer Verlag")
author
P(author,@name,"John")
(c)

Figure 4.5: Query plans corresponding to different shrinking strategies: a) weak shrinking; b) strong shrinking; c) medium shrinking

the join characteristics of XML documents. Document size vary from 120 to 280 Megabytes. Experiments have been conducted concerning space occupancy as well as query and update time. The results of the performed experiments are described below.

4.6.2 Storage Requirement

In order to understand the overhead of the XJoin Index with respect to the space occupancy, we have compared the space occupancy of traditional element and attribute indexes (the only needed to implement the weak shrinking strategy) with that of using also the XJoin Index. Figure 4.6 reports results comparing EI and EI + CXJI sizes by varying the number of elements with at least one child. The total number of elements is nearly two million and the maximum number of elements with at least one child is set to be approximately 35% of the total number of elements. We believe this setting is reasonable as theoretically speaking, the percentage of these elements is approximately $\frac{1}{\alpha}$, where α is the branching factor and is usually a large number in XML tree, which means that $\frac{1}{\alpha}$ is likely to be small. We can also see this from real datasets, e.g., this percentage is about 9.4% (200948 versus 2137264) for DBLP XML record. We can see from the figure the linear increment of the size of CXJI when the number of elements with at least one child increases.

Figure 4.7 reports results comparing AI and AXJI sizes by varying the number of elements with at least one attribute, which we call attribute elements. The total number of elements is around two million and every attribute element has 2 attributes on average, which we believe is a reasonable assumption based on the statistics of DBLP XML record (526780 attributes vs 275760 attribute elements). When the number of attribute elements increases, the sizes of both AI and AXJI increase. Note that the size of AXJI is decided by the number of elements with attribute, while the size of AI is decided by the number of attributes. In case that most elements with attribute have large number of attributes associated with them, it is possible that the size of AI becomes larger than the size of AXJI though we store less information for an attribute than for an element.

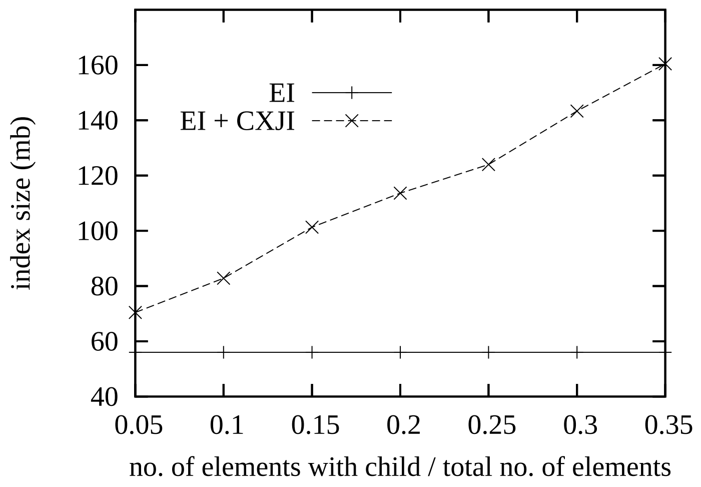


Figure 4.6: Space occupancy for artificial databases, by varying the number of branching elements

4.6.3 Search Efficiency

We performed two groups of experiments. The first group concerns attribute, counting, and direct navigational expressions. For them, we have compared weak and strong shrinking strategies. Note that strong shrinking for attribute and counting selections always correspond to simple index lookups. Then, we have compared strong and weak shrinking strategies in evaluating branching path expressions against XMark benchmark and DBLP datasets. We did not consider medium shrinking in the experiments since it will always get performance results in between those obtained by applying strong and weak shrink strategies. It is known that holistic techniques perform better than standard merge-join method to process twig queries. But since the aim of the experiments is to consider the impact of using difference indexes, for the sake of simplicity we implemented the merge-join method [104]. We compare the performance of both approaches by comparing the elapsed time of processing the queries.

Given a base expression $a[b(n)]$, in the following we denote with $S_(a, b)$ the

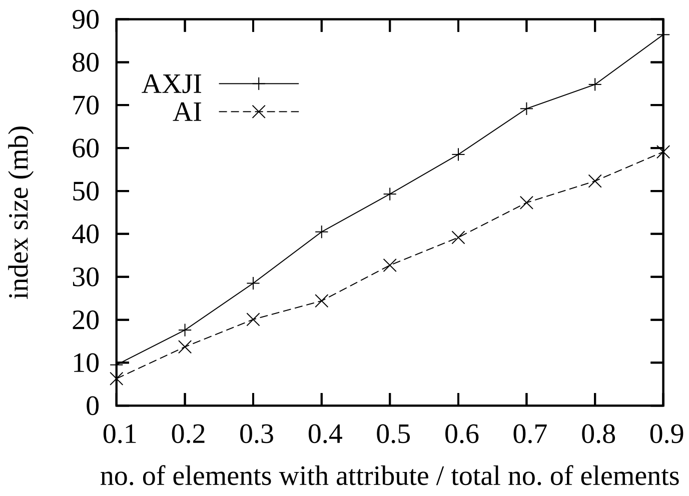


Figure 4.7: Space occupancy for artificial databases, by varying the number of element attributes

semi-join selectivity of element a with respect to b, i.e., the total number of a elements having at least one child element b. We also denote with $S(b, a)$ the semi-join selectivity of element/attribute b with respect to a, i.e., the total number of b elements/attributes having parent element a. Semi-join selectivity of attribute selections and navigational expressions is defined accordingly.

Base path expressions

Attribute Selections. For this group of experiments, we considered expressions like $a[cond]$ and we varied the number of predicates in *cond*, by fixing semi-join selectivities of each attribute selection. In order to change semi-join selectivities, we performed the experiments on artificial datasets. Figure 4.8 reports elapsed time for executing $a[@b_1\ AND\ @b_2...,\ AND\ @b_m],\ S(a, b_i) = S(b_i, a) = 25\%,\ i = 1, ..., m$ (we got the similar results for different selectivity values). We can see that strong shrinking outperforms weak shrinking for any number of attribute predicates in the condition. This is due to the fact that each additional predicate in the condition corresponds to an additional equi-join in strong shrinking but to a structural join in weak shrinking. Moreover, since structural join is more expensive than equi-join, weak shrinking line grows faster than the strong shrinking one. From Figure 4.9, we can also see that different selectivity values do not significantly vary weak shrinking performance. Indeed, in this case, all attribute selections are solved by applying a structural join over the same set of elements. Even if the number of returned elements changes when changing the selectivity, the number of steps performed in the structural algorithm is almost the same. This is not true by using a strong shrinking strategy. In this case, since we use a join index, results are directly influenced by selectivities. We got the same results from different numbers of predicates in the condition.

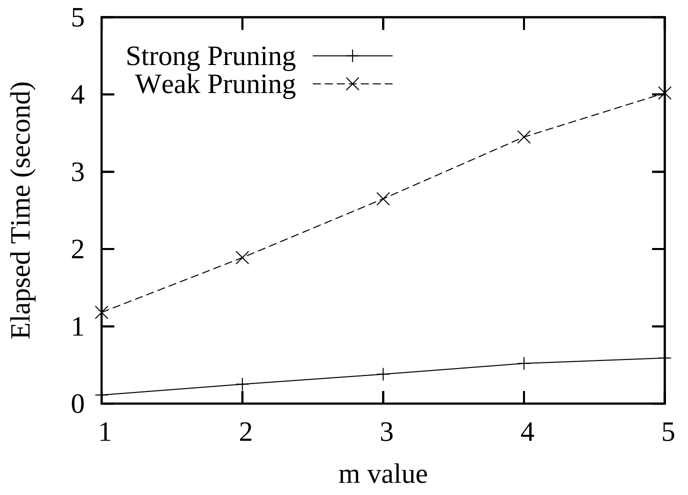


Figure 4.8: Elapsed time for attribute selections $a[@b_1\ AND\ @b_2...,\ AND\ @b_m]$ with respect to $m,\ S_(a, b_i) = S_(b_i, a) = 25\%,\ i = 1, ..., m$

Counting Selections. For this group of experiments, we considered expressions like $a[cond]$ and we varied the number of counting predicates in $cond$, by fixing semi-join selectivity of each counting selection. In order to change semi-join selectivity, we perform the experiments on artificial datasets. Similarly to attribute selections, by using the weak shrinking strategy, indexes are used just to retrieve sets of el-

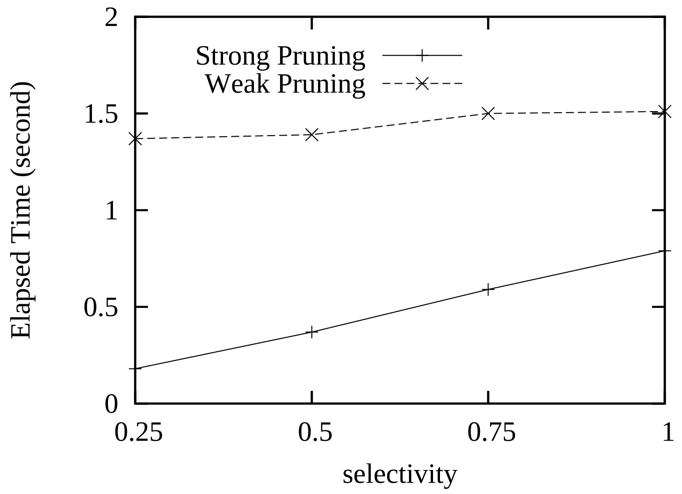


Figure 4.9: Elapsed time for attribute selection $a[@b_1\ AND\ @b_2]$ with respect to $S_(a, b_i) = S_(b_i, a),\ i = 1, 2$

ements with a certain tag. These sets are then joined by using a structural join algorithm to solve the counting selections. On the other hand, by using the strong shrinking strategy, each counting predicate can be solved by a single index lookup. Figure 4.10 (a) reports elapsed time for executing $a[b_1(n)\ AND\ ...\ AND\ b_m(n)]$, $S_{(a,b_i)} = S_{(b_i,a)} = 25\%,\ i = 1, ..., n$, considering both $n = 2$ and $n = 8$. We can see that strong shrinking outperforms weak shrinking for any number of counting predicates in the condition. We got the similar results from Figure 4.10 (b), where selectivity is set to be 75%. Similarly to attribute selections, since each additional predicate in the condition corresponds to an additional equi-join in strong shrinking but to a structural join in weak shrinking, weak shrinking line grows faster as structural join is more expensive than equi-join. Moreover, note that the strong shrinking strategy is independent from n, since different values of n does not require different processing. Therefore, in the Figure 4.10, the lines representing strong shrinking when n = 2 and n = 8 are almost coincident. On the other hand, weak shrinking strategy increases when n increases since additional processing has to be applied in this case. For similar reason in previous discussion on attribute selections, we can see that, from Figure 4.11, weak shrinking performances are almost independent from selectivity while for strong shrinking strategy, elapsed time increases as selectivity increases.

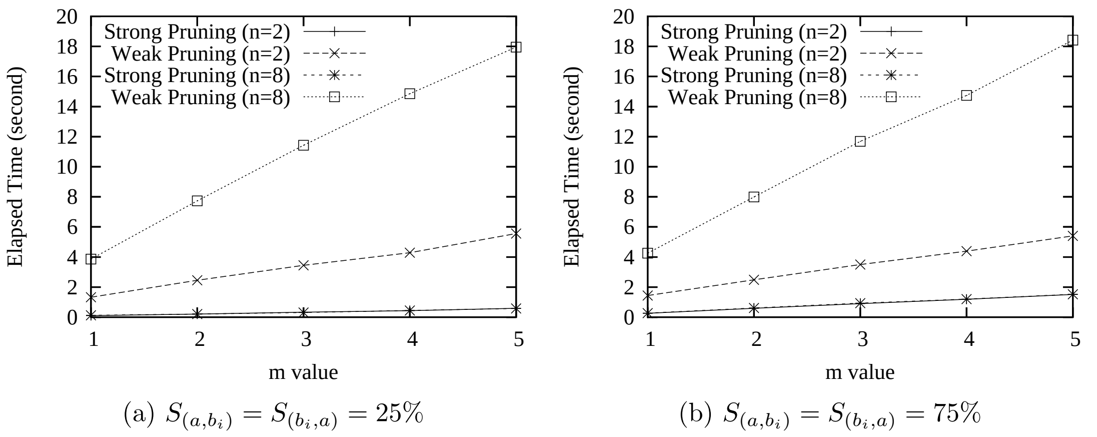


Figure 4.10: Elapsed time for counting selections $a[b_1(\geq n)\ AND\ ...\ AND\ b_m(\geq n)]$ with respect to m

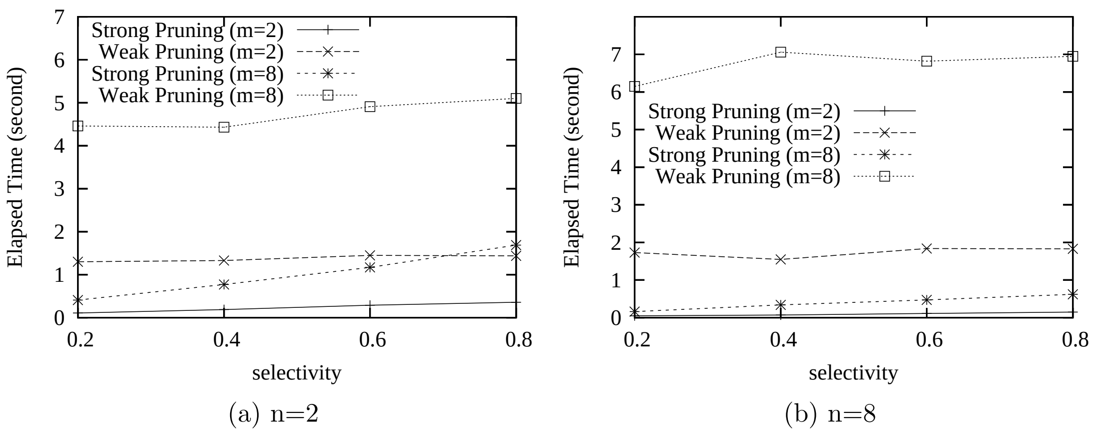


Figure 4.11: Elapsed time for counting selections $a[b_1(\geq n)\ AND\ ...\ AND\ b_m(\geq n)]$ with respect to $S_{(a,b_i)} = S_{(b_i,a)}$, $i = 1, ..., m$

Direct navigational selections. For this group of experiments, we considered expressions like e_1/e_2 and we varied element selectivity. Figure 4.12 reports the results we obtained. We can see that the elapsed time for weak shrinking is almost constant while the elapsed time for strong shrinking increases as selectivity increasing. As we described before, with the strong shrinking technique, we solve a/b by first retrieving those a elements with b children from CXJI, then join this partial set of a element with b elements. The number of a elements to be retrieved from CXJI, which is the main factor affecting elapsed time, increases by the increasing of selectivity. But for weak shrinking, no matter how many a elements have b children, we always retrieve them all. hence the elapsed time remain almost constant as long as the total number of a and b elements remains constant.

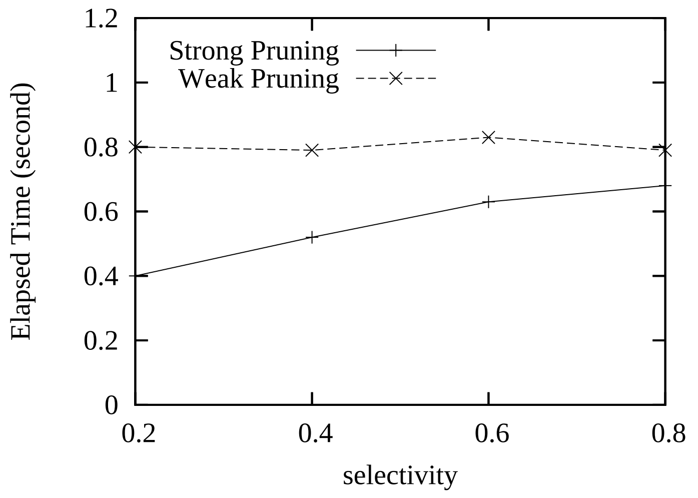


Figure 4.12: Elapsed time for direct navigational expressions e_1/e_2 with respect to $S_(e_1, e_2)$

Experiment on benchmark/real datasets

For this group of experiments, we considered various queries from the XMark dataset and DBLP dataset. We did not choose queries presented in the benchmark of XMark project, instead we designed some queries from scratch since we wanted to consider also counting queries which are not included in the XMark workload. Details of these queries are presented in Table 4.1. For all the queries, we applied strong and weak shrinking strategies and the results are presented in

Query	**Expression**	**Result cardinality**
Q1	profile[@income]	12823
Q2	person/phone	12679
Q3	profile[interest($\geq$10)]	511
Q4	person/watches[watch($\geq$10)]	1146
Q5	person[/profile[interest($\geq$3)]]/watches[watch($\geq$3)]]	1135
Q6	person[/profile[interest($\geq$20)]]/watches[watch($\geq$30)]]	2
Q7	item[@featured]/mailbox[mail($\geq$ 3)]]	101
Q8	item[@featured]/mailbox[mail($\geq$ 10)]	1
Q9	open_auctions/open_auction[bidder($\geq$ 20)]	258

Table 4.1: XMark Queries

Figure 4.13. We can see that strong shrinking outperforms weak shrinking in every query, especially for those queries involving counting selections, which proves the effectiveness of the XJoin index in reducing unnecessary I/Os. The results are consistent with those presented in the synthesis datasets.

We also compared strong and weak pruning strategies with TwigStack algorithm presented in [19] using a real XML data instance originated from DBLP XML record. The path expression of the query we used is:

$$inproceedings[author(\geq n)\ AND\ cite(\geq n)]$$

The result cardinalities with varying n are presented in Table 4.2 and the result of this group of experiment is represented in Figure 4.14. As expected, we can see from the results that weak pruning performs much worse than the other two methods. TwigStack has proven to be an quite efficient algorithm and it performs better than strong pruning when n is less than three. However, as the number of n increases, the number of *inproceedings* elements that satisfy the counting predicates gets fewer and fewer. Therefore, the index lookup on CXJI requires less time and

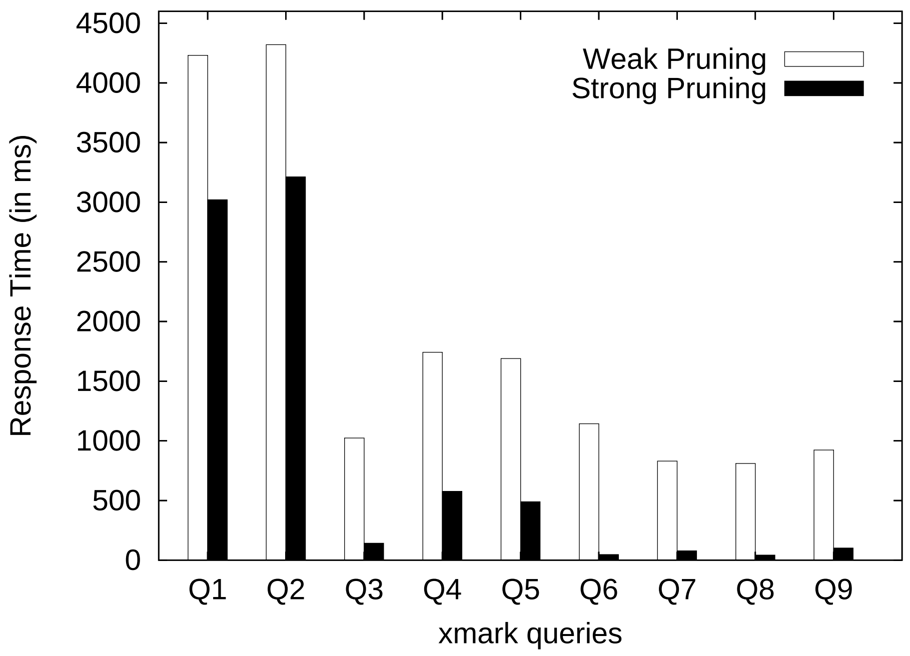


Figure 4.13: Results for XMark dataset queries

n	1	2	3	4	5	6	7	8	9	10
No. Results	689	455	231	98	46	21	8	4	3	1

Table 4.2: DBLP Result

strong pruning has therefore the best performance.

4.6.4 Update

Figure 4.15 shows the average elapsed time for both inserting/deleting a single element to/from the proposed index structure, which is build on the XMark benchmark dataset, for one hundred times under two different situations. Both insertion and deletion requires a number of index lookup and update. Therefore, the elapsed time is determined by the number of index lookups and updates. In case the if conditions in insertion and deletion algorithms are true, additional index operations are required, hence the total elapsed time increases. But in either case, the

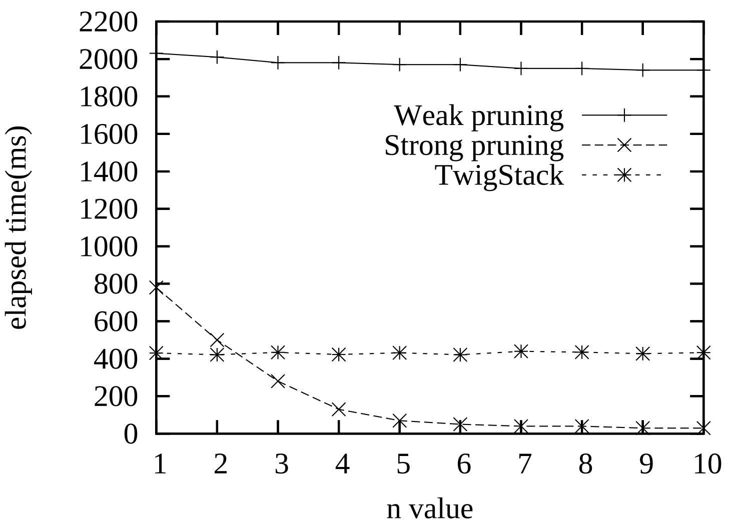


Figure 4.14: Results for Queries on DBLP Dataset

update operation is efficient enough.

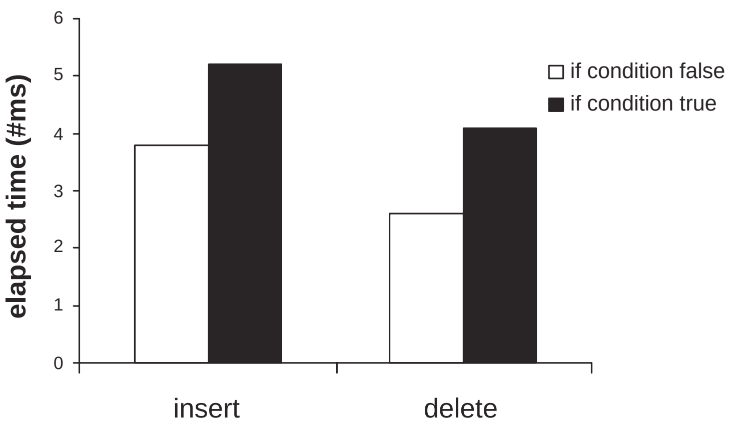


Figure 4.15: Element insertion and deletion

4.7 Concluding remarks

In this chapter, we have proposed the XJoin Index, a join index for executing branching path queries in an XML database. The XJoin index has been defined extending the well known join index defined for relational databases to the XML context. The main characteristics of the proposed index are simplicity, since it is based on a set of B^+-trees, and flexibility, since it supports the choice of different execution plans for the same branching query, and efficiently support attribute selections, element selections, and parent-child relationships. Among element selections, we consider also counting queries, determining whether an element has at least (or at most) a certain number of children with a specific name. We have subsequently presented algorithms and costs for search and update operations in the XJoin Index and we have shown how the XJoin Index can be used to prune the query twigs before applying structural join algorithms, which presents different approaches. The experimental results confirm that the XJoin Index effectively reduces the number of structural joins to be executed, thus improves the overall query response time.

Chapter 5

The Lazy Update Scheme

5.1 Introduction

Most research works in the area of XML document processing focus on how to efficiently query XML documents. However, we must also provide an efficient way to handle XML updates in order to fully evolve XML into a universal data representation and exchange standard. Basically, there are two classes of update operations. One is *value update*, where the value (*PCDATA*) of an element/attribute is changed. The other is *structural update*, where a new element/attribute is inserted into (removed from) the original XML document, thus causes structural change. We will focus our discussion in this chapter to the structural change of XML elements for two reasons, (i) value update is relatively easier to handle compared with structural change, (ii) attributes can be equivalently considered as subelements.

As mentioned in previous chapters, an XML document is usually modeled as an ordered tree, and every element/attribute is assigned an unique label(identifier) based on its position in the XML document. And this label is later used as the key or part of the key in the element index for efficient structural join processing. So the main problem of structural update is that, in order to maintain the correctness of structural joins, we may need to update the labels of possibly a large number of elements when the original XML document has been updated, which makes the update operation very inefficient. Previous attempts to solve this problem mostly rely on various dynamic labeling schemes. But they either cannot accommodate arbitrary dynamic updates , or involve heavy computation which affects the efficiency of structural join and update operation.

We therefore propose a different approach to handle XML update in this chapter. This approach relies on the fact that in real world scenarios, XML updates tend to be done in batch manner, i.e., multiple XML elements are inserted (or removed) together. We model the whole XML database as a single *super* document and consider update operations as inserting (or removing) XML *segments*, which corresponds to a set of elements that must be inserted (deleted) together, into (or from) the *super* document. In this model, every element has two positions. One is its *local* position with respect to the segment it belongs to, the other is its *global* position with respect to the super document. The local position of an element will not change once it is assigned to an element. This gives us the inspiration of using local position of an element as its identifier so that this identifier does not need to be modified when update occurs. But the local position cannot be directly used as the identifier of an element as it is not unique, thus is not suitable for structural join algorithms. The key point to solve this problem is that the number of inserted (or removed) segments is likely to be significantly less than the number of elements these segments contain. So we can build an in-memory *update log* to record the information of every segment. With the information of segments and the local positions of elements, we can therefore uniquely identify an element in the *super* document, thus we can perform structural joins without knowing the global positions of the elements. We are aware that, after many insertions, the size of update log could grow large. However, as shown in the experiments, the size of update log is small enough not to pose a problem to modern machines. Moreover, the database administrator can rebuild the index for the whole XML database during maintenance, and therefore clear the update log for future insertions.

We also note that in real world scenarios, XML documents are normally stored as just plain text files and update is likely to be done by simple text editing, i.e., only the start location in the super document and the length of the inserted (removed) segment are available to us. The update log shall be able to compute the structural information of the segment given only these two values.

We organize this chapter as follows. Section 5.2.2 presents the structure of the proposed *update log* and update algorithms for it. In Section 5.3, we present a structural join algorithms that works with our *lazy* approach. Results from experimental study on the update and structural join operations are given in Section 5.4. The results show that out approach is more efficient than existing dynamic labeling approaches for update and, additionally, it improves query processing performance. We conclude this chapter in Section 5.5.

5.2 The Data Structure

In this section, we introduce the in-memory update log and corresponding update operations, together with the element index.

5.2.1 Preliminaries

As we have mentioned, by adding a dummy root, the whole XML database, whether it have been organized with a tree or many sub-trees, can be considered as one *super document* and XML update operations are therefore modeled as inserting (or removing) XML segments into (or from) the *super document*. It is obvious that every segment inserted (or removed) must be a valid XML document itself so that the validity of the whole XML database is preserved, which also implies that every segment, except the dummy root, is included in at least one other segment. As part of the *super document*, every segment s has a starting position in the *super document.* which we refer to as its *global position*, denoted by $s.gp$, with respect to the *super document.* Each segment has a length, which is denoted by $s.l$. Based on global positions and length, we can then define a segment *containment* relationship.

Definition 5.1 *Let s_1, s_2 be two segments. The global position of s_i, denoted by $s_i.gp$, is defined as the offset of the starting tag of s_i inside the super document. The length of s_i, denoted by $s_i.l$, is the number of characters in s_i. s_1 contains s_2 if and only if $s_1.gp < s_2.gp$ and $s_1.gp + s_1.l > s_2.gp + s_2.l$. s_1 is the ancestor and s_2 is the descendant segment. If there exists no other segment s_3 such that s_1 contains s_3 and s_3 contains s_2, s_1 directly contains s_2. In this case, s_1 is the parent and s_2 is a child segment of s_1.* □

In Figure 5.1, rectangles represent XML segments and dashed lines the direct containment relationships among different segments. We can see that segment 3 is directly contained in segment 2, which is its parent. Segment 1 is its ancestor. Segment 4 and 5 are child segments of segment 3 and segment 6 is a descendant of segment 3.

Besides a *global position*, we also assign a *local position* to every segment (except the root), denoted by $s.lp$. The local position of a segment s_2 with respect to its parent s_1 is simply the number of characters in s_1 preceding s_2 and not contained in any left sibling of s_2, at the time when s_2 is being inserted.

Definition 5.2 *Let segment s_1 be the parent of segment s_2. The local position of s_2, denoted by $s_2.lp$, is defined as $s_2.lp = s_2.gp - s_1.gp - \sum_{(s \text{ is a left sibling of } s_2)} s.l$.*
□

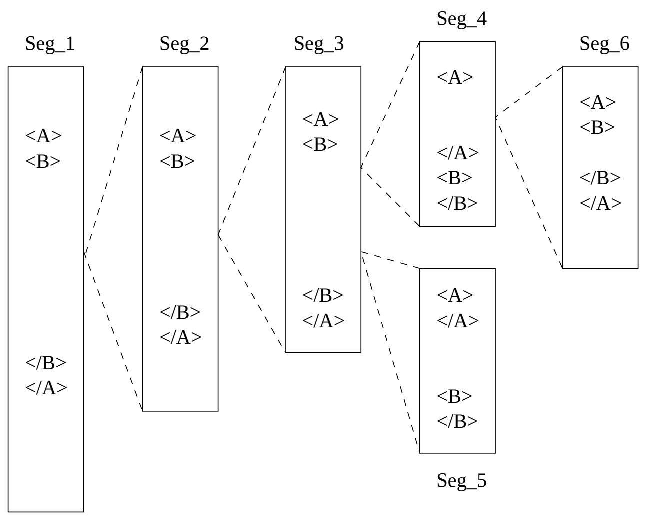


Figure 5.1: Segment containment relationship

Local positions, once assigned to a segment, never change. Insertions and deletions of left siblings of s may vary the local position of a segment s. However, such insertions (deletions) increase (decrease) the global position of s; thus, according to definition 5.2, $s.lp$ does not change.

Figure 5.2 represents the super document corresponding to Figure 5.1, pointing out segment length, global and local positions, assuming, for the sake of simplicity, each element is a dummy element which contains no character content and each tag requires n characters for its storage.

5.2.2 Structure of Update Log

The update log consists of two main data structures. The first consists of a B$^+$-tee to easily access segment information. The second is the tag-list, which is a simple inverted list that maps element tags to segments in the super document.

Figure 5.3 illustrates the structure of the considered B$^+$-tree. Keys are segment identifiers (*sid*), which are unique identifiers generated by the system when a new segment is inserted. The B$^+$-tree, call segment B$^+$-tree (SB-tree), associates *sid*'s

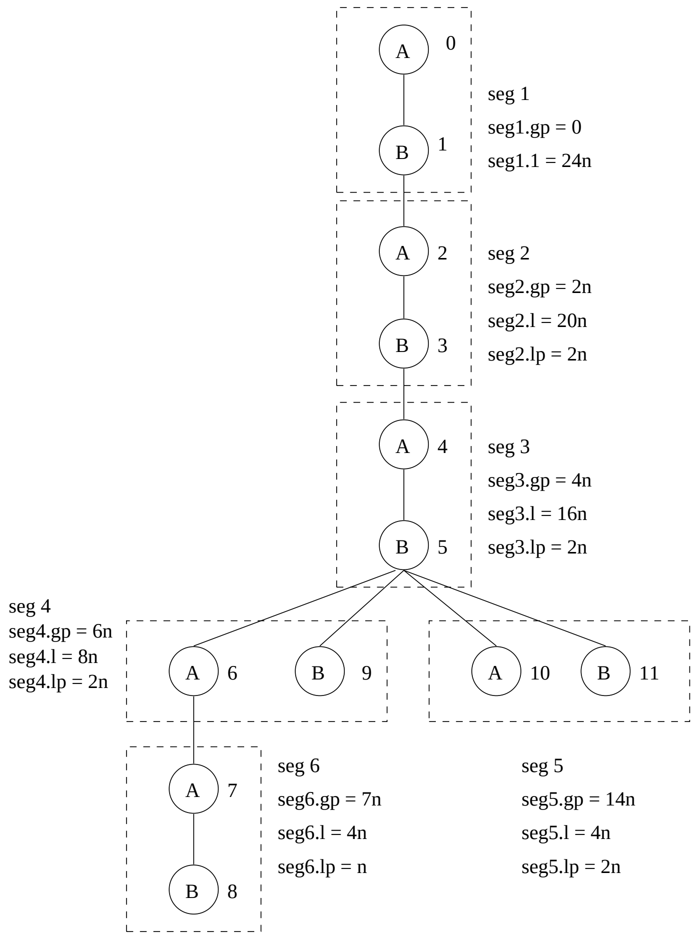


Figure 5.2: Super document corresponding to figure

with the following information: the segment global position *gp*, the segment length *l*, the segment local position *lp* with respect to its parent, a pointer to its parent, pointers to its children, sorted by global positions in ascending order. Every leaf node in the B$^+$-tree corresponds to a segment in the super document. Since the leaf level is organized as a tree-like data structure, we refer to it as the ER-tree (sEgment-Relationship tree) in the remainder of this chapter. The ER-tree is a segment-based representation of a document, according to the sequence of insertions/deletions that have been executed. The root node represents the dummy root of the super document. As we will see in Sections 5.2.3 and 5.3, the ER-tree simplifies update operations whereas the B$^+$-tree is needed for query execution.

Figure 5.4 illustrates the structure of the ER-tree for the segments shown in Figure 5.1.

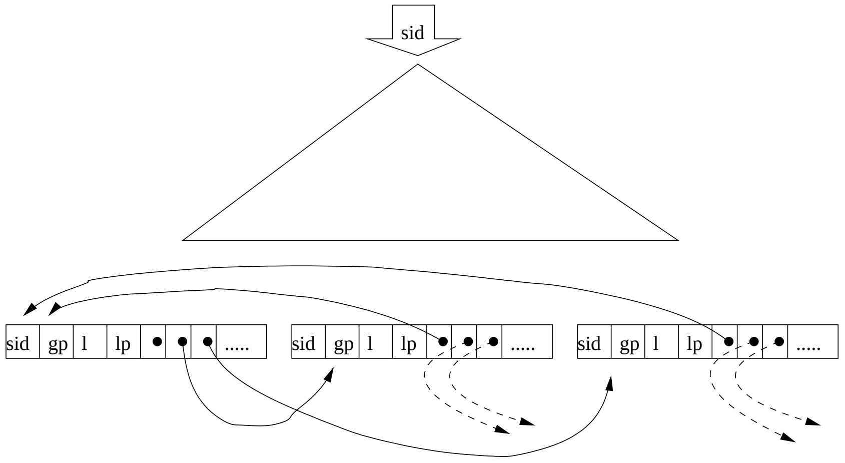


Figure 5.3: SB-tree (Segment B$^+$-tree)

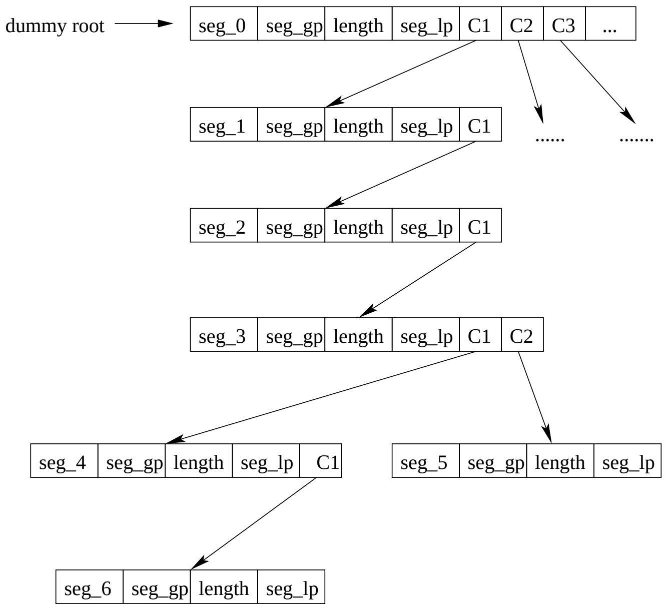


Figure 5.4: ER-Tree (sEgment Relationship tree)

Obviously, the size of SB-tree is $O(N)$, where N is the number of segments contained in the super document. We claim that N will be quite small compared with the size of the document, thus it is reasonable to assume that the SB-tree resides in main memory.

Figure 5.5 illustrates the tag-list structure of the segments shown in Figure 5.1, where we suppose the tag ids for tag A and B are tid_1 and tid_2 respectively. Each list stores not only the segment id but also a *path* for every segment in the ER-tree. The *path* of a segment is actually the concatenation of the segment ids of all its ancestor segments plus its own segment id. Tag ids are sorted by ascending order, and within the path lists attached to the tags, the paths are sorted by global positions of the corresponding segments. The reason for storing the path is that it allows us to more efficiently perform operations needed by the structural join algorithm, as we will see in Section 5.3. The path is computed when the segment is inserted into the super document and the length of the path is at most $O(N)$, which occurs in the most highly nested case where every segment has at most one child segment, i.e., the ER-tree is reduced into a single linked list.We also associate the numbers of element occurrences for each tag id in each segment together with the paths, which helps us determine if a path should be removed from the path list when we remove segments from the super document. We will describe this in detail in Section 5.2.3. The size of the tag-list is $O(TN^2)$ where T is the number of different tag ids and N is the number of segments. This is the worst case estimation when every segment contains all tags and when the segments are most highly nested as we described above. We are aware that there could be more compact ways to represent the tag-list, but this approach is easier to maintain and since all the ids are simply numbers, the total size of that tag-list is still small enough to fit into the memory of modern machines.

Proposition 5.1 ***SPACE COMPLEXITY***. *Let N be the number of segments and T the number of element tags. The space complexity of the SB-tree and tag-list is O(N) and O(TN^2), respectively.* □

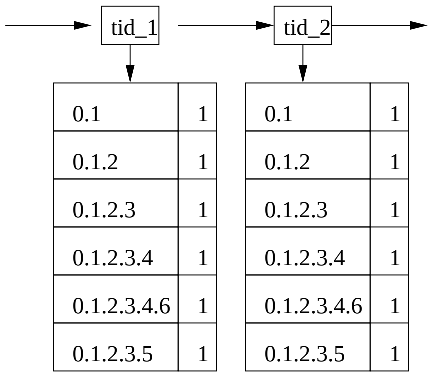


Figure 5.5: Tag-List

5.2.3 Updating the Update Log

The assumptions under which we define update operations are: (i) for each insertion/deletion of a segment, we assume to know only its global position and its length; local positions are transparent to the application and are detected during update execution; (ii) only segment information can be modified;elements are only inserted or deleted but never modified; (iii) inserting a segment into the super document results in adding a new node into the SB-tree, but removing a segment from the super document does not necessarily mean deleting a node from the SB-tree.

In case of insertion of a segment into the super document, the system will automatically generate a segment id for it. Then both the SB-tree and the tag-list need to be updated. More precisely, in case of insertion, we need to: (1) update the global positions of relevant nodes in the ER-tree; (2) compute the needed information for the segment from ER-tree; (3) add a node corresponding to the segment into the SB-tree; (4) update the tag-list accordingly. Figure 5.6 gives the algorithm for steps (1), (2), and (3). Function `AddNewSegment_Start` first increases global positions greater than that of the new segment by the length of the new segment. Then, it calls function `AddNewSegment`, that recursively traverses the ER-tree and adds the new node into the proper location in the child list of its

```
AddNewSegment_Start(new)
1.   For each node m in ER-tree, m.gp > new.gp
2.      m.gp = m.gp + new.l
3.   AddNewSegment(ER_root,new,empty_path)
4.   insert new.Node into the SB-tree

AddNewSegment(root,new,path)
1.   path := path + root.sid
2.   root.length := root.lengh + new.l
3.   If a child node k of root is an ancestor of new
4.   Then
5.         AddNewSegment(k,new,path)
6.   Else
7.      path := path + new.sid
8.      lengthSum := 0
9.      for each left sibling n of new
10.        lengthSum := lengthSum + n.l
11.     new.lp := new.gp - root.gp - lengthSum
12.     insert new.sid into the child list of root
13.  Endif
```

Figure 5.6: Adding a segment into the SB-tree

parent node. The path variable is initially empty. The root parameter is initially set to the root node of the ER-tree. We first append the segment id of the current root node into the path variable and increase the length of the current root node by the length of the segment represented by the new node. Then, according to definition 5.1, we check if any of the children of the current root fully contains the inserted node. If there exists such a child node, then the current root is not the parent node of the inserted node and we recursively call the `AddNewSegment` function, setting this child node as the new root. if there is no such child node, which means that the current root is the parent of the inserted node, then we simply insert the node into the child list of the current root. Of course, the new node must be inserted into a proper location in the child list such that the order in global positions of the child nodes is still preserved. New node's local position is then computed according to definition 5.2. Although child list could be long after many insertions, we can search or update a child list of size k in $O(log(k))$ time because the SB-tree resides in memory and efficient algorithms like binary search [30] can be used. It is obvious that the overall cost of adding a node into the SB-tree is bounded by $O(N)$, where N is the number of segments. This worst case also occurs in the case that segments are most highly nested, which we have described earlier. Concerning global position changes, again, although the cost of this propagation process is $O(N)$, it is still efficient in most cases because the SB-tree is memory resident. Thus, the overall time to update the SB-tree is in $O(N)$.

Once we have obtained the path of an inserted segment, we can update the tag-list accordingly. If the inserted segment contains the tag with tag id T, then its path (computed by the `AddNewSegment` function) is inserted into the path list associated with tag id T according to global ordering. The cost of locating a tag id is $O(log(T))$, where T is the number of different tags. The cost of inserting a path into the path list associated with a tag id is bounded by $O(log(N))$. Therefore the total cost of updating the tag list is $O(p(log(T)+log(N))))$, where p is the number of tags contained in the inserted segment.

Compared with inserting a new segment, removing a segment from a super document is a bit more complicated. Indeed, the removed segment may not be any of the existing segments recorded in the SB-tree. To clarify the problem, let *seg* be the segment to be removed. We formalize the relationship between the removed segment and segments in the ER-tree (except the dummy root) into three main cases:

1. *seg* is *contained* in a segment k, i.e., $k.gp < seg.gp$ and $k.gp + k.l > seg.gp + seg.l$. The length of k is reduced by $seg.l$.

2. *seg contains* a segment k, i.e., $seg.gp < k.gp$ and $seg.gp + seg.l > k.gp + k.l$. k and all its descendants are deleted from the ER-tree in this case.

3. seg intersects a segment k. If $k.gp < seg.gp < k.gp + k.l < seg.gp + seg.l$, it is a left intersection; if $seg.gp < k.gp < seg.gp + seg.l < k.gp + k.l$, it is a right intersection. The length of k is reduced by $k.gp + k.l - seg.gp$ (for left intersection) or $seg.gp + seg.l - k.gp$ (for right intersection).

Besides the previous cases, the global position of segments starting after seg ending position is reduced by $seg.l$.

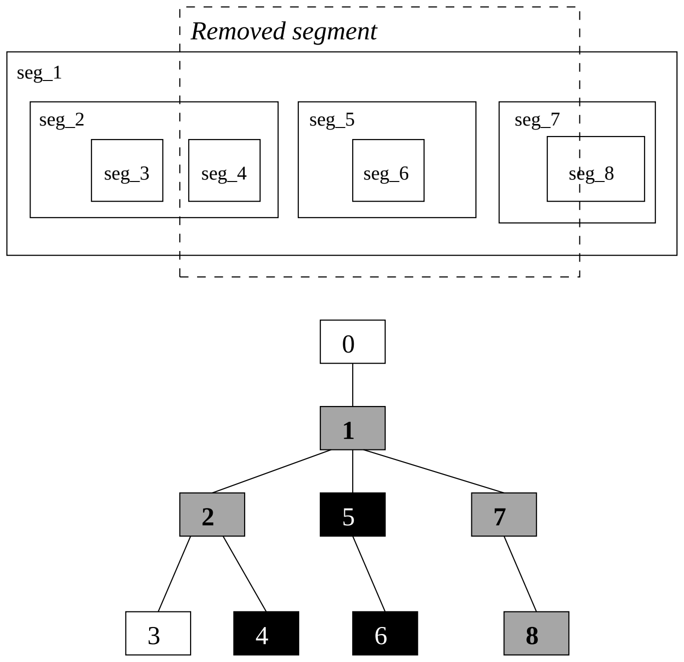


Figure 5.7: Example of Removing a Segment

The relationship between the removed segment and the super document is a combination of the three cases listed above. In Figure 5.7, the removed segment (dashed box) is contained in segment 1, contains segments 4,5 and 6, left intersects segment 2 and right intersects segments 7 and 8. In the corresponding ER-tree, black nodes refer to those segments that are to be completely deleted from the tree, gray nodes refer to those segments that are affected by the removed segment in terms of segment length and global position, white nodes refer to those segments that are not affected when deletion occurs. Node 0 is the dummy root, whose information is not supposed to change in all cases.

```
RemoveSegment_Start(seg_org)
1.  For each node m in ER-tree, s.t. m.gp > seg.gp+seg.l
2.     m.gp = m.gp - seg.l
3.  RemoveSegment(ER_root,seg_org)

RemoveSegment(root,seg)
1.  root.l := root.l - seg.l
2.  For every child node k of root
3.     If seg is contained in k
4.     Then
5.        RemoveSegment(k,seg)
6.     Else if k is contained in seg
7.     Then
8.        remove k from the child list of root
9.        remove k and its descendant nodes from SB-tree
10.    Else if seg left intersects k
11.    Then
12.       $seg_{aux}$.gp = seg.gp;
13.       $seg_{aux}$.l := k.gp + k.l - $seg_{aux}$.gp
14.       RemoveSegment(k,$seg_{aux}$)
15.    Else if seg right intersects k
16.    Then
17.       $seg_{aux}$.gp = k.gp;
18.       $seg_{aux}$.l := seg.gp + seg.l - k.gp
19.       RemoveSegment(k,$seg_{aux}$)
20.       k.gp := seg_org.gp
21.    Endif
22. Endfor
```

Figure 5.8: Segment removal algorithm

Figure 5.8 gives the algorithm for updating the SB-tree when a segment is removed from the super document. Function `RemoveSegment_Start(seg)` takes the current segment to be removed as parameter. We first reduce the global position of segments starting after *seg* ending position by the length of the current removed segment. Then, we call the recursive function `RemoveSegment` with the root of the ER-tree and the segment to be removed as parameters. In calling such function, the root segment will always contain the segment to be removed. `RemoveSegment` first updates the root length, then it checks the relationship between the current removed segment and the child segments of the current root. If the removed segment is contained in a child node, we just call function `RemoveSegment` recursively. If a child segment is contained in the current removed segment, we remove the node that corresponds to that child segment and all its descendants from the SB-tree. If the removed segment left intersects a child node, we update the length of the removed segment by using an auxiliary segment, as indicated in lines 12-13, and call function `RemoveSegment` recursively, setting the child node as the new root. If the removed segment right intersects a child node, we update the length and the global position of the removed segment by using an auxiliary segment, as indicated in lines 17-18, we call function `RemoveSegment` recursively, and we update the global position of the child node. The usage of an auxiliary segment allows the algorithm to maintain the correct information associated with the segment to be removed when checking the other child nodes. Moreover, in order to maintain the correctness of structural join results after a series of segment insertion/deletion, we need to maintain the information of each removed segment as well, which also helps decide which elements are to be removed from the element index. The cost of recursively updating the ER-tree is bounded by $O(N)$. The worst case happens when the segments are most highly nested and the removed segment intersects all of them. Since the cost of deleting a node from the SB-tree is $O(log(N))$, the total cost for updating the SB-tree is bounded by $O(Nlog(N))$.

The tag-list is updated after updating the element index. To update the tag-list when a segment is removed, we need to know the tag name and the number of elements actually removed from the super document, since a path has to be deleted only if no more elements with that tag are contained in the segment after deletion. The information concerning the type and the number of elements removed is computed when we actually perform the delete operation in the element index. The cost of updating the tag-list for one tag id is bounded by $O(log(T)+mlog(N))$, where m is the number of segment paths to be removed from the path list, T is the number of different tags in the super document, and N is the number of segments. $log(T) + log(N)$ is the worst case cost of locating a single path in the tag-list for a given tag id. The worst case occurs when the tag id is contained in all segments and a path is to be removed from the path list attached with this tag id. Therefore, the total cost of updating the update log when a segment is removed is $O(Nlog(N) + p(log(T) + mlog(N)))$, where p is the number of distinct tag names the removed segment contains.

Following proposition gives a summary of update cost of update log.

Proposition 5.2 ***Update Complexity***. *Let N be the number of segments, T the number of distinct element tag ids, p the average number of distinct element tag names in a segment, and m the average number of paths to be removed from a path list in the tag-list. The segment insertion cost is O(N + p(log(T) + log(N))) and the segment deletion cost is O(Nlog(N) + p(log(T) + mlog(N))).* □

5.2.4 Element Index

The element index is simply a B$^+$-tree. Every record in the index represents an element and is represented by the tuple $(tid, sid, start, end, LevelNum)$, where tid is the tag id of the element, sid is the segment id the element belongs to, $start$ is the starting position of the element in the segment identified by *sid*, i.e., the *local position* of the element, *end* is the local ending position of the element, $LevelNum$ is the depth at which the element appears in the document. According to the proposed labeling scheme, each element is univocally identified by the tuple $(tid, sid, start)$, which is in fact the key of the element index.

Search and insert operations for the element index are the same as those for standard B$^+$-trees. But when element records are removed from the element index, we need to record the number of removed elements with the same tid and sid, which is necessary when we decide if a segment path should be removed from a path list as we mentioned when we discussed the impact of deletion on the tag-list in Subsection 5.2.3.

5.3 Query Evaluation

Under the lazy XML update approach, existing structural join algorithms can still be used to compute pairs of ancestor/descendant or parent/child elements. As we discussed in Subsection 5.2.4, elements in the element index are identified by the ids of the segments in which they appear and their local positions. To use traditional structural join algorithms, we need to compute the global positions of elements. To do that, we first retrieve information of the segments that contain the elements to be participated into the structural join, then we compute the global positions of the element according to the local positions of the elements.

On the other hand, information concerning segments can be used to reduce the number of elements to be checked in structural join, thus improving the overall performance. In this section, we first present some results concerning containment relationship between elements and segments; then we show how to perform structural join by using the element index and the update log.

5.3.1 Preliminaries

The containment relationship between XML segments is closely related to the containment relationship between XML elements, which is the foundation of structural join. In general, we distinguish between *cross-segment* join, i.e., join between elements contained in distinct segments, and *in-segment* join, i.e., join between elements contained in the same segment.

In the following, we present two properties of cross-segment joins that will be useful in defining our structural join algorithm, Their proof follows from definition 5.1. The first property specifies that, in order to be related by an ancestor-descendant relationship, elements must be contained in pairs of ancestor-descendant segments, thus providing a necessary condition for pairs of segments to generate cross-segment joins; the second provides a sufficient and necessary condition for an element to generate cross-segment joins. In presenting such properties, given two segments S and T such that S contains T, P_T^S denote the local position of a segment L containing T and directly contained in S. For example, if S is identified by path 0.1.2 and T by 0.1.2.3.4.6, P_T^S is the local position of segment 3 with respect to S. If S directly contains T, we just set P_T^S to be the local position of T with respect to S. Moreover,we call $X - element$ an element with X as tag name.

Proposition 5.3 *Let S and T be two distinct segments. Let a be an A-element in S and b a B-element in T. The following results hold:*

1. *If a is an ancestor (parent, descendant, child) of b then S contains T (S directly contains T, T contains S, T directly contains S).*

2. *a is an ancestor of b if and only if a contains T and $a.start < P_T^S$ and*

$a.end > P_T^S$. □

Proof. Suppose that a contains b. We get the following inequalities: (i) $a.gp < b.gp$, $a.gep > b.gep$ (since $a//b$); (ii) $b.gp > N.gp$, $b.gep < N.gp + N.length$ (since b is contained in T). Now suppose that a does not contain T. We have the following cases: (i) a is in front of T. In this case, $a.gep < T.gp$. But from inequality (i), $T.gp < b.gp$, thus $a.gep < b.gp < b.gep$, which contradicts inequality (i); (ii) a is after T. In this case, $a.gp > T.gp + T.length$. But, from inequality (ii), $T.gp + T.length > b.gep > b.gp$, thus $a.gp > b.gp$, which contradicts inequality (i). This means that a must contain T.

Now assume that b is contained in T and a contains T. We have to prove that a contains b. We get the following inequalities: (i) $a.gp < T.gp$ and $a.gep > T.gp + T.length$ (since a contains T); (ii) $b.gp > T.gp$ and $b.gep < T.gp + T.length$ (since b is contained in T). From them, it follows that $a.gp < T.gp < b.gp$ and $a.gep > T.gp + T.length > b.gep$, thus a contains b. The other part of the proposition can be proved in a similar way.

If a contains T, necessarily contains the segment at position P_T^S by definition. Thus, Proposition 5.3(2) is completed proved.

To prove Proposition 5.3(1), we note that if a is an ancestor of b, according to Proposition 5.3(2), a contains T. Since segments are either nested or disjoint, it follows that S contains T. If a is a parent of b, no elements must be contained between a and b. Thus, T must be directly contained in S. □

Consider Figure 5.9, where S_n and E_n represent the starting and ending position of element n. We see that the A-element S_2 in segment 2 contains segment 3 and so is its ancestor A-element S_1. Therefore, according to proposition 5.3(2), we have two join results: $(2 : S_1, 3 : S_1)$ and $(2 : S_2, 3 : S_1)$. Segment 2 is contained in the A-element S_4 in segment 1, hence , the A-element S_4 in segment 1 contains segment

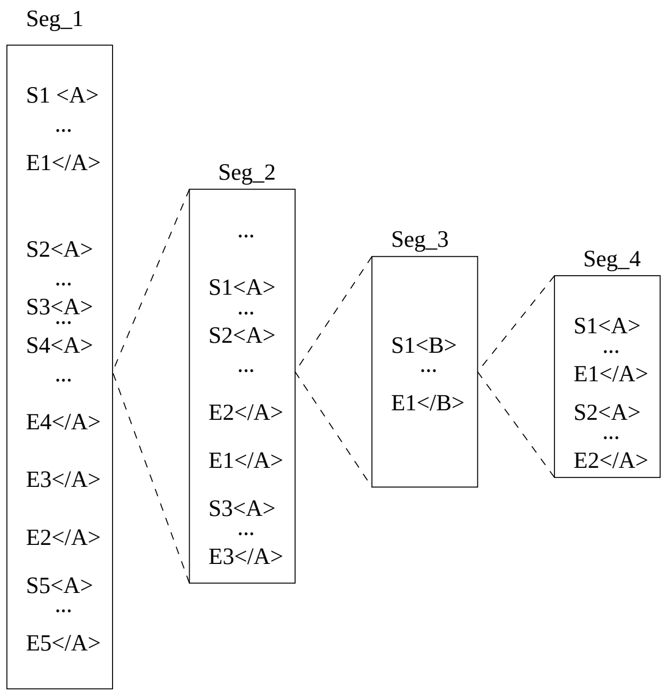


Figure 5.9: Cross-segment join between segments

3 as well. We get another three results from A-element S_4 in segment 1 and its ancestor A-elements, which are S_2 and S_3. The three pairs are $(1 : S_2, 3 : S_1)$, $(1 : S_3, 3 : S_1)$ and $(1 : S_4, 3 : S_1)$. We can finally see that A-element S_3 in segment 2 does not produce any result with B-element in segment 3 since it does not contain segment 3. The same happens with A-elements S_1 and S_5 in segment 1.

5.3.2 The Lazy-Join Algorithm

In the following, we present a structural join algorithm that uses segment information to improve the processing. It is a variation of the stack-based algorithm proposed in [12], called `Stack-Tree-Desc`. We consider this algorithm because it is quite efficient and, at the same time, it is easy to implement. The algorithm we propose, called `Lazy-Join` to highlight that it relies on a lazy XML update approach, returns pairs of ancestor/descendant elements first sorted by descendant

```
Algorithm Lazy-Join($SL_A$,$SL_D$)
1.  sa = $SL_A$.firstNode; sd = $SL_D$.firstNode; OutputList = NULL; stack = empty_stack();
2.  While (stack is empty) /* the stack is empty*/
3.    If (sa.gp < sd.gp)
4.      stack.push(sa);
5.      sa = $SL_A$.nextNode;
6.    Else if (sa.gp = sd.gp)
7.      Append the result of Stack-Tree-Desc(sa,sd) to OutputList;
8.      sd = $SL_D$.nextNode;
9.    Endif
10. Endwhile
11. While (($SL_A$ and $SL_D$ are not empty) /* both lists are not empty */
12.   If (sd.gp > stack.top.gp + stack.top.l) stack.pop();                /* Step 1 */
13.   Else if (sa.gp < sd.gp)                                            /* Step 2 */
14.     If (sa contains sd)
15.       remove from stack.top() elements e such that e.lep < $P_{sa}^{stack.top()}$;
16.       stack.push(sa);
17.     EndIf
18.     sa = $SL_A$.nextNode
19.   Else if (sa.gp $\geq$ sd.gp)                                        /* Step 3 */
20.     For every segment sa1 in stack, starting from stack bottom
21.       For every element a1 in sa1 such that a1.start < $P_{sd}^{sa1}$, starting from lowest lp
22.         If (a1.end > $P_{sd}^{sa1}$)
23.           For every element d1 in sd
24.             Append (sa.sid,a1.lp,sd.sid,d1.lp) to OutputList;
25.     If (sa.gp = sd.gp) Append the result of Stack-Tree-Desc(sa,sd) to OutputList;
26.     sd = $SL_D$.nextNode
27. While ($SL_D$ and the stack are not empty) /* $SL_A$ is empty /*
28.   If (sd.gp > stack.top.gp + stack.top.l) stack.pop();
29.   Elseif (stack.top contains sd)
30.     For (sa1 = stack.bottom; sa1 != NULL; sa1 = sa1.up)
31.       For (a1 = sa1.bottom; a1 != NULL; a1 = a1.up)
32.         If (a1.start < $P_{sd}^{sa1}$ && a1.end > $P_{sd}^{sa1}$)
33.           For (b1 = sd.bottom; b1 != NULL; b1 = b1.up)
34.             Append (sa.sid,a1.lp,sd.sid,b1.lp) to OutputList;
35.     sd = $SL_D$.nextNode;
36.   Endif
37. Endwhile
```

Figure 5.10: Algorithm Lazy-Join

positions.

`Lazy-join` differs from traditional structural join algorithms in two aspects: (i) it computes the result starting from two lists of segment identifiers, instead of two lists of element identifiers; (ii) it relies on proposition 5.3 to improve cross-segment join computation. Any traditional structural join algorithm can be used to generate in-segment joins.

Suppose the path expressions is $A//D$. The algorithm starts from two lists of segment identifiers SL_A and SL_D, the first containing A-elements, the second containing D-ones, sorted by global positions. These lists are extracted from the tag-list (see Section 5.2.2). The basic idea of the algorithm is to merge the two lists of segments (thus, each segment in the lists is accessed just once), according to their global position, using a stack. The stack at all times contains a sequence of segments from SL_A. Each segment in the stack is a descendant of the segment below it. For each segment s, we push: (i) its identifier, global position, and length (retrieved from the SB-tree); (ii) the local starting and ending positions of A-elements in s (retrieved from the element index).

At each step, the ancestor-descendant relationship between the current segment in SL_A - say s_a - and the current segment in SL_D - say s_d - is checked. Based on the result of this comparison, the stack is manipulated, pointers in the lists advanced, and results produced, according to proposition 5.3. More precisely, three distinct operations can be executed, until SL_A or SL_D becomes empty:

1. *Pop Segments*: $s_d.gp > stack.top.gp + stack.top.l$. This means that s_d is not a descendant of the top segment in the stack, thus no future segment from SL_D will be a descendant of the current top of the stack (since segments are sorted by their global position). Therefore, we can pop the stack. No result is generated.

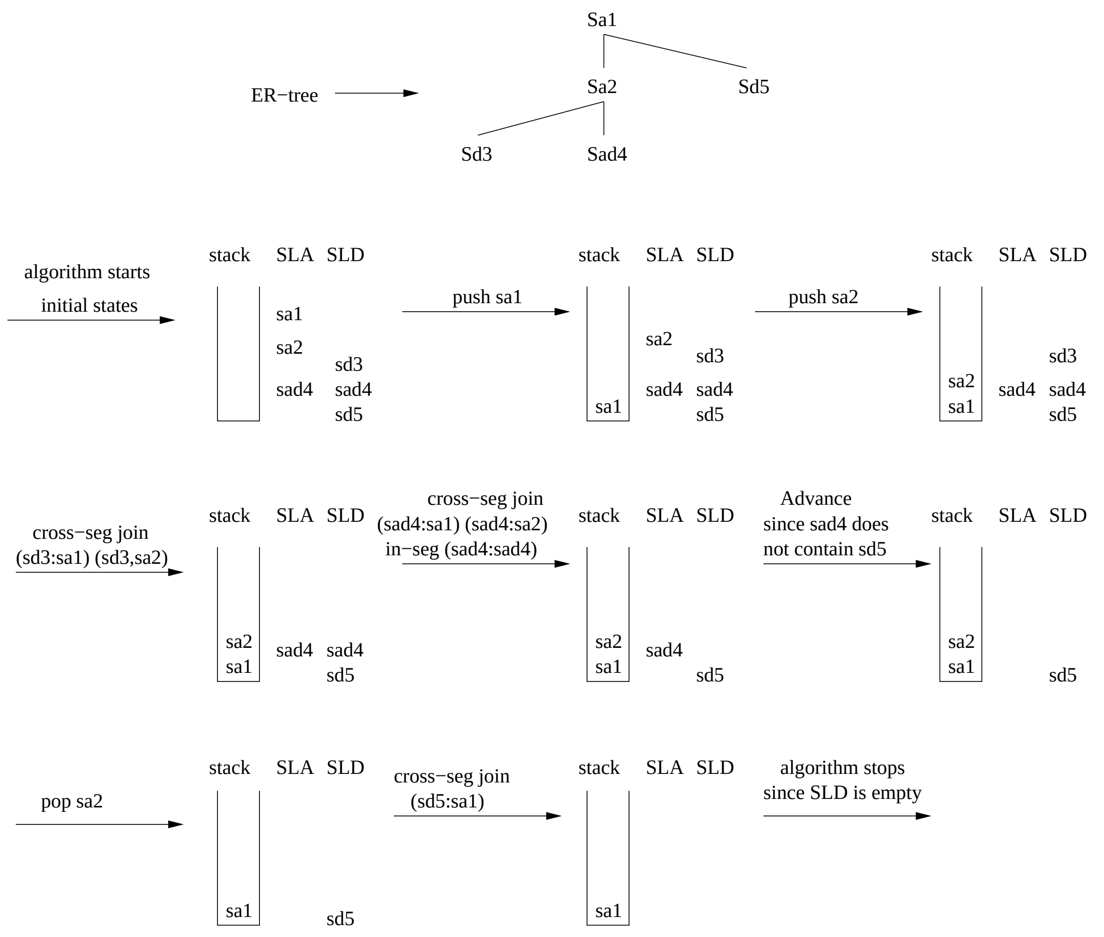


Figure 5.11: algorithm processing for query $A//D$: segments sa_i contain A-elements and not D-ones, segments sd_i contain D-elements and not A-ones, segments sad_i contain both D- and A-elements

2. *Push Segments*: $s_a.gp < s_d.gp$. Due to step 1, we know that s_d is a descendant of the top segment in the stack. Since $s_a.gp < s_d.gp$, s_a is a descendant of the top segment, too. If s_a contains s_d, s_ad is pushed into the stack since, due to 5.3, it may generate joins with s_d. Then, We advance SL_A. No result is generated.

3. Join Generation: $s_a.gp \geq s_d.gp$. Due to step 1 and 2, we know that s_d is a descendant of all the segments in the stack, thus, according to proposition 5.3, all segments in the stack may generate cross-segment joins with s_d. However, differently from [12], not all elements in the stack will generate joins with elements in s_d. Thus, for each segment in the stack, say s_a1, we generate cross-segment joins with D-element in s_d only if condition 2 of proposition 5.3 is satisfied. No condition has to be checked for elements in s_d.

If $s_a.gp = s_d.gp$, in-segment joins are then generated. Elements are joined based on their local positions by using any structural join algorithm (e.g., `Stack-Tree-Desc`. Note that if $s_a.gp > s_d.gp$, s_a and s_d cannot generate joins due to proposition 5.3(1), due to the ordering of SL_A, any future segment in SL_A cannot generate joins with s_d. Thus SL_D is advanced.

If SL_D becomes empty before SL_A, no more joins can be generated and the algorithm stops. One the other hand, if SL_A becomes empty before SL_D, we just process the remaining segments in SL_D according to steps (1) and (3), until SL_D or the stack become empty.

According to the algorithm above, elements in the stack do not represent a chain of ancestor-descendant relationships. On the other hand, this property is satisfied by the algorithm presented in [12]. In order to reduce the overhead, due to the fact that more elements than required are inserted in the stack, the `Lazy-Join` algorithm can be optimized by inserting in the stack only element that can *potentially* generate cross-segment joins. Since the stack contains a chain of ancestor-descendant segments, according to proposition 5.3(2), those element are such that they contain at least one child segment. We get this behavior by modifying step(2) as follows: (i) we push only elements containing at least one segment. The information can be checked by using child information associated with each leaf value in the SB-tree. Note that, since in-segment joins are computed before a segment from SL_A is pushed into the stack, no pairs are lost; (ii) before pushing a segment s_a into the stack, we remove from the top segment the elements ending before s_a starts, since they will not contain any future segment from SL_A. Figure 5.10 presents lazy-join algorithm and Figure 5.11 presents an simple example of its application. We finally observe that the `Lazy-Join` algorithm can be easily extended to compute parent-child relationships. In this case, according to proposition 5.3(1), segments must share a parent-child relationship. Thus , at step 3 of the algorithm, cross-segment joins can be generated only from the pair of segments $(stack.top, sd)$. For each pair of elements (a_1, d_1), the join is generated if $d_1.LevelNum = a_1.LevelNum + 1$.

5.3.3 Analysis of Lazy-Join Algorithm

Correctness of the proposed algorithm follows from proposition 5.3 and [12]. Concerning time complexity, the algorithm implements a merge of two lists. Let $p_A(p_D)$ be the average number of $A-(D-)$ elements in one segment. The operations performed by the algorithm are the following:

- *Push segments.* Each segment in SL_A is pushed at most once. Since for each pushed segment s_a we need to access the SB-tree to get necessary information about s_a, the cost of pushing segments is $(O(|SL_A|*(log(N)+log(NE)+p_A))$, where N is the total number of segments and NE is the total number of elements.

- *Pop Segments.* Each segment in the stack can be popped at most once. Thus, the cost is bounded by $|SL_A| * p_A$.

- *Join Generation.* In Step 3, for each segment in the stack, at most all its A-elements are checked. Each of them either generates cross-segment joins with all D-elements in sd or with none of them (lines 21-24). Thus, the cost of accessing D-element is amortized by the cost of returning join results in output. Moreover, according to the applied optimization, only in the top segment more than one element may generate no join with elements in sd(all elements ending before sd starts). Since SL_D is ordered, such elements will generate no join with any next segment from SL_D and therefore can be removed from the stack. By applying this additional optimization, each top segment is completely analyzed once during the overall join computation. Thus, the complexity of generating cross-segment joins is $O(|SL_D| * (log(NE) + log(N)) + |SL_A| * p_A + OutputList)$, where $log(NE)$ is due to the element index access needed to retrieve D-elements contained in sd, $log(N)$ is due to the computation of P_{sd}^{sa} for the top element in the stack (for the others, it can be computed after each push operation and stored in an auxiliary data structure), and $OutputList$ is the number of returned pairs. Concerning in-segment joins, since we use the algorithm proposed in [12], the cost is $O(S_AD(P_A+P_D+log(NE))+OutputList)$, where S_AD is the number of segments containing both A- and D-elements.

Proposition 5.4 ***TIME COMPLEXITY***: *The time complexity of algorithm* `Lazy-Join` *is* $O(|SL_A| * p_A + S_AD * p_D + seg\_overhead + OutputList)$ *where* $Seg\_overhead = (|SL_A| + |SL_D|) * (log(N) + log(NE))$ □

We note that cost $S_AD * p_D$ is the maximum overhead due to in-segment joins. Thus, by fixing the total number of joins as well as the number of segments, increasing the percentage of cross-segment joins improves the performance. With the respect to the `Stack-Tree-Desc` algorithm, whose complexity is linear in the number of A- and D- elements, i.e., in $|SL_A| * p_A + |SL_D| * p_D$, we note that, since $S_AD * p_D \leq |SL_D| * p_D$, `Lazy-Join` outperforms `Stack-Tree-Desc` depending on: (i) the percentage of in-segment joins; (ii) the segment overhead, which in turn depends on the number of segments.

5.4 Performance Study

5.4.1 Experiment Setup

We have implemented the update log and the related element index in C++, on an ultra 450 machine with processor of 500MHz and 3 Gigabytes of main memory. For the experiments, we used both synthetic datasets, created by the IBM XML generator [6] and benchmark datasets, generated by XMark [15]. We used synthetic datasets in order to more easily get the characteristics we need for analyzing the properties of the proposed algorithms. On the other hand, by using XMark datasets, we can analyze the proposed techniques in more realistic situations. When using synthetic datasets, to simulate the real world scenario, we chopped the datasets into many small segments and inserted these segments into an initially dummy XML document while maintaining the validity of the super document. Segment size varies from few kilo bytes to several hundred kilo bytes and the number of elements they contain varies from a few to several thousands.

Experiments have been conducted concerning update log space occupancy and building time, as well as time to execute structural joins and to update the structures. In the last two cases, we considered two different assumptions, resulting in different query/update times. Under the first assumption (that we call *lazy dynamic* (LD)), we assume to maintain the update log incrementally updated, thus at query time the update log is ready to be used; under the second assumption (that we call *lazy static* (LS)), we further reduce update time by assuming to maintain incrementally updated only the ER-tree and to keep the tag-list unsorted. Path lists are sorted and the B$^+$-tree generated from scratch just before querying the XML database.

5.4.2 Update Log Space and Building Time

The update log consists of both the SB-tree and the tag-list. Figure 5.12 reports the size of each component and the total size of the update log, in term of kilo bytes, when the number of inserted segments varies. Each segment contains all element tags appearing in the tag-list (the worst case for tag-list update). We can see that the size of the tag-list increases much faster than that of SB-tree and the tag-list size contributes a large percent of the total size of update log. This is because, as stated in Section 5.2.2, the size of the tag-list is $O(TN^2)$ while that of the SB-Tree is $O(N)$, where T is the number of distinct tag ids and N is the number of segments. We also note that in the nested case, tag-list size increases faster, as discussed in Section 5.2.2. As we have claimed before, the size of the update log is considerably small. Its size is only about 95 kilobytes for balanced ER-trees and 195 kilobytes for nested ER-trees after over 300 insertions, which cannot be a problem for modern machines. Figure 5.13 reports similar results for update log building time.

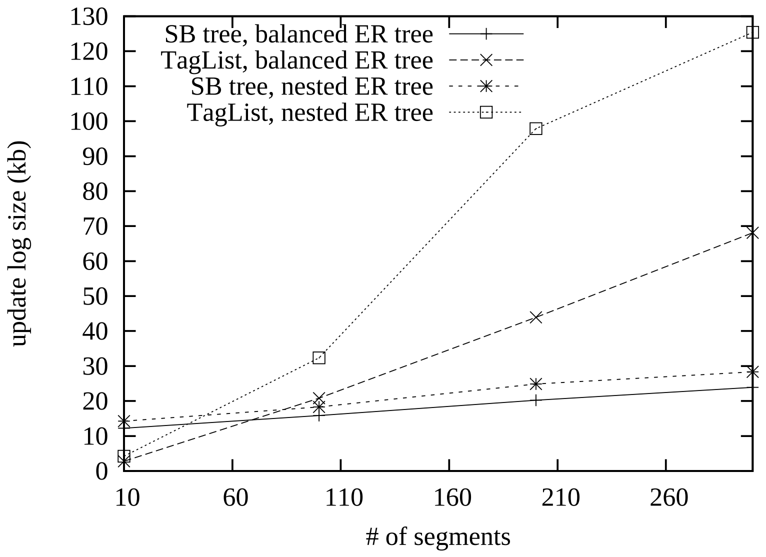


Figure 5.12: Update Log Size

5.4.3 Structural Join Processing

We compared the elapsed time of solving path expressions like $A//D$ by LS, LD, and the `Stack-Tree-Desc` algorithm proposed in [12] (denoted by STD in the following). Three main groups of experiments have been performed.

The aim of the first group is to analyze the impact of cross-segment joins in query performance. To this purpose, we fixed the number of segments and the number of A- and D- elements. Then, we varied the percentage of cross-segment joins. Since the structure of the ER-tree determines how many segments containing D-elements can be skipped, we considered two different ER-tree structures: a completely nested one (which corresponds to the worst case) and a balanced one, which corresponds to a more reasonable real situation. Figure 5.14 reports the results for the nested and balanced case, by considering 50 and 100 segments. We can see that when the number of cross-segment joins increases, the performance of LS and LD increases, since, as we saw in Section 5.3, the cost of performing cross-segment joins is lower than the cost of performing in-segment joins. On the other hand, the

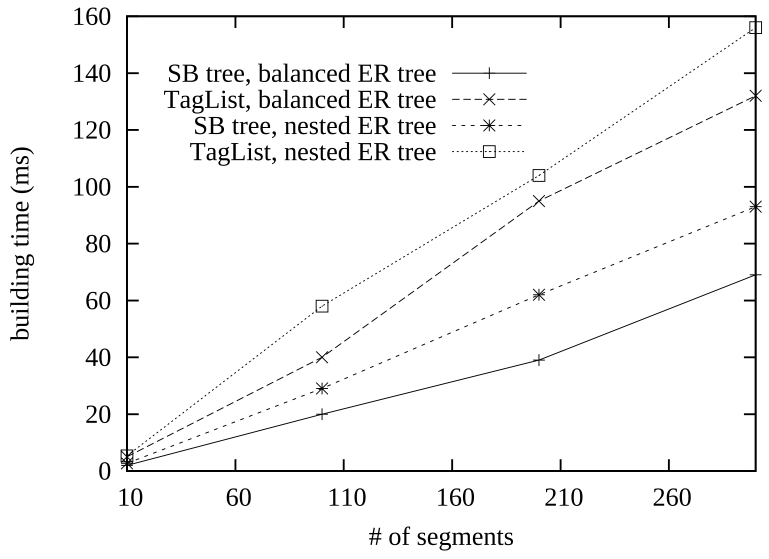
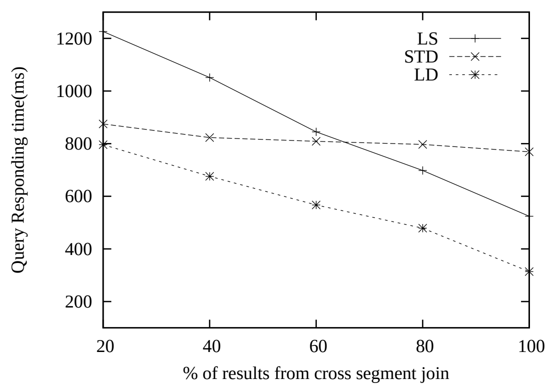


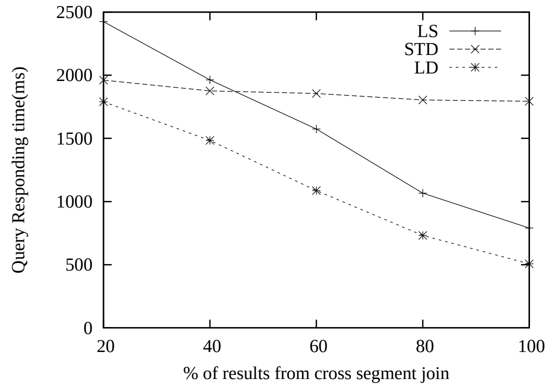
Figure 5.13: Elapsed time for building the update log

cost of STD is almost constant since, even if it can be influenced by the position of elements generating joins, it has to read all elements that may potentially generate joins even if some of them will not generate any result. From this experiment we also note that, as expected, LD is always more efficient than STD, due to the fact that by LD entire segments can be skipped and the segment processing overhead is very low. On the other hand, LS is more efficient than STD for high cross-segment join percentage (higher than 60% in this experiment).

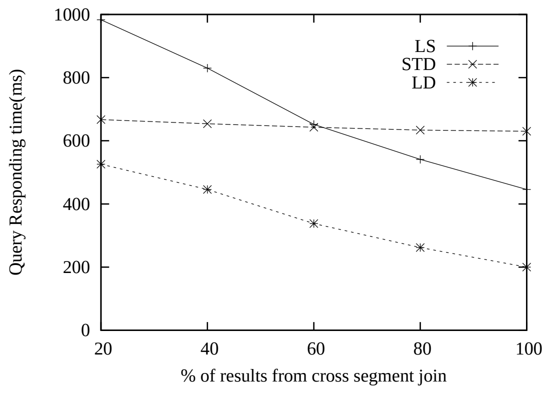
In the second group of experiments, we investigated the impact of number of segments in structural join performance. To this purpose, we fixed a document (which contains about 120k elements and whose size is approximatively 10 Megabytes), we changed the number of segments and the structure of the ER-tree, and we executed the same query over all situations. In all cases, the percentage of cross-segment joins is around 20%. Results from LD and STD are reported in Figure 5.15. We can see that the higher is the number of segments the higher is the processing time since the segment lists to be scanned are longer. We also note that, for more than



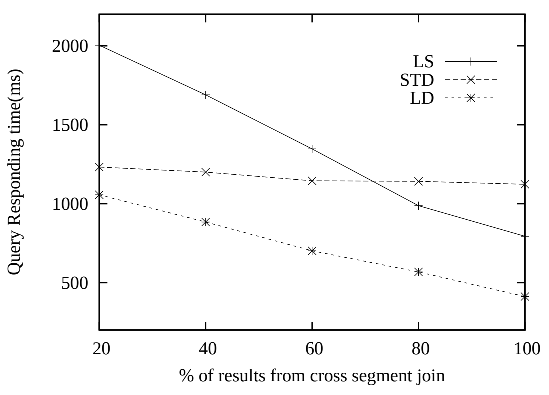
(a) 50 segments



(b) 100 segments



(c) 50 segments



(d) 100 segments

Figure 5.14: Elapsed time for structural join over: (a)-(b) nested ER-trees; (c)-(d) balanced ER-trees

180 segments and balanced ER-tree, LD performs worse than STD. This is because, in this case, the overhead in segment processing is higher than the improvement got from cross-segment join computation.

Finally, in the third group of experiments, we analyzed the performance of LD on an XMark dataset, slightly modified to increase the number of cross-segment joins. The size of the dataset is about 100 Megabytes and it contains about 3 millions elements. Table 5.1 and Figure 5.16 present the considered queries and the obtained results. For the experiment, we chopped it into 100 segments and we considered a balanced ER-tree. The percentage of cross-segment joins is about 20% to 30%. We can see that for all the considered queries, LD, differently from LS,

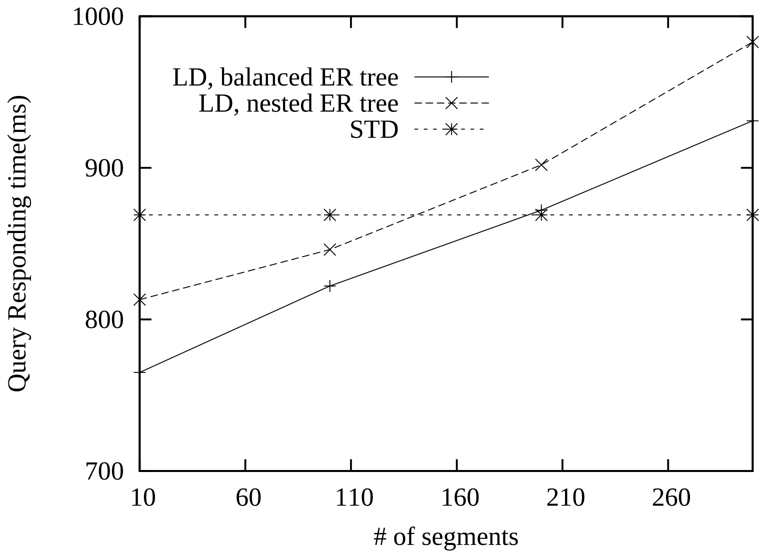


Figure 5.15: Elapsed time for structural join over the same document, with different ER-trees

Query	**XPath expression**	**Result cardinality**
Q1	person//phone	413170
Q2	profile//interest	494240
Q3	watches//watch	879891
Q4	person//watch	1455040
Q5	person//interest	1074792

Table 5.1: XMark Queries

outperforms STD. The results are coherent with those presented in Figure 5.15.

From the first two groups of experiments, it follows that, when the number of segments is very high or when the percentage of cross-segment joins decreases, and therefore for the special case when one segment coincides with one element, the performance of LS and LD may decrease. In those cases, nested segments can be collapsed together in order to reduce the overall number of segments, increase their sizes, and improve query performance. Alternatively, traditional structural join

Figure 5.16: Elapsed time for structural join over XMark datasets.

algorithms can still be used. At the same time, as we will see in Subsection 5.4.4, the usage of segments is still useful since it always improves update performance.

5.4.4 Update Processing

In order to analyze update time of the lazy approach, we considered two different experiments. In the first experiment, we compared LD with a traditional approach, labeling elements by their starting and ending positions. For that, we inserted a segment into XMark datasets of variable size and we reported the elapsed time of updating the element index (and the update log, for the lazy approach). We considered the average case in which the inserted segment causes half the elements to change their global positions. Figure 5.17 reports the obtained results in log scale. We can see that, as the size of the XML document increases, the insertion time of the traditional approach increases dramatically while that of LD remains low and almost constant. The reason is obvious. For the traditional approach, whenever a new segment is inserted, many of the indexed element records have to be updated. However, for LD, we need only to insert a new segment into the in-memory log, and to insert element records of that segment into the element index. No update of existing element records is required. We note that for the lazy approach, global relabeling of segments is required. However, since the number of segments is usually much less than that of elements, the overhead of this step does not greatly affect the insertion cost.

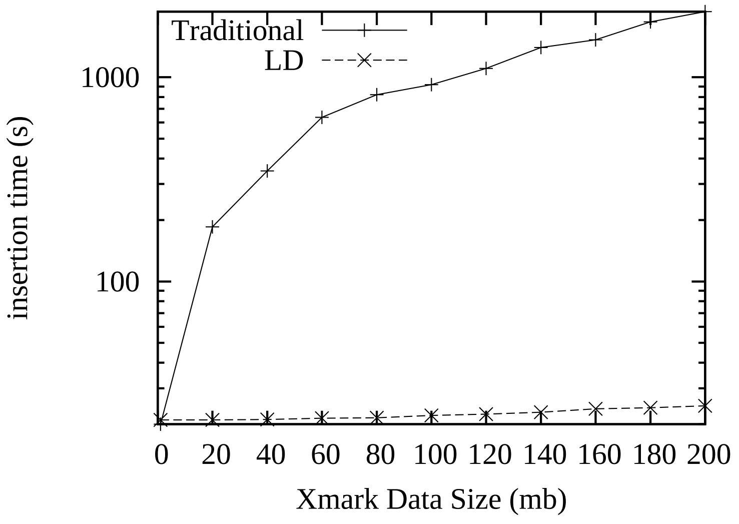


Figure 5.17: Elapsed time of inserting one segment.

The aim of the second experiment is to compare update performance of LD and LS with that of approaches based on immutable labeling schemes. To this purpose, we considered the prime numbering scheme recently proposed in [101] (denoted by PRIME in the following). Since the structure of the ER-tree influences the length of paths in the tag-list and the number of segments to be updated, we considered both the balanced and nested case. Figure 5.18 and 5.19 show the elapsed time of inserting one element in a document chopped into 100 segments, by changing the number of elements (maintaining fixed the number of distinct tag names) and the number of distinct tag names (maintaining fixed the number of elements) in the inserted segment, respectively. Since our approach is based on segments, to determine the update cost of each single element (needed in order

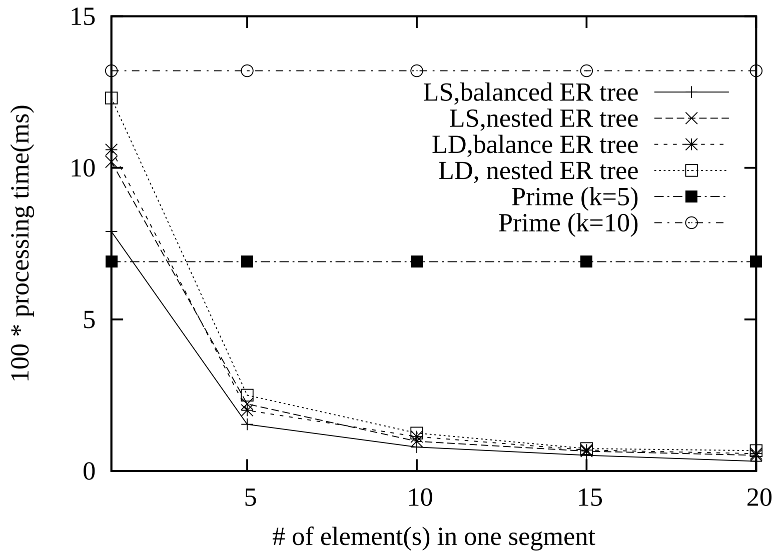


Figure 5.18: Elapsed time of inserting one element by varying the number of elements.

to compare LD and LS with PRIME), we divided the time required to insert the segment by the number of elements it contains. K value in the figure is the number of prime numbers that share the same simultaneous congruence in PRIME. We can see that LS and LD require much less time than PRIME. Indeed, PRIME requires recomputing at least one simultaneous congruence value in the table of simultaneous congruence values and this recomputation process contributes large part of the total processing time and it is also very costly according to the algorithm presented in [101]. One the other hand, under the lazy approach, no complicated computation is required. We see that by increasing the number of elements inside a segment, the insertion time decreases. This is due to the fact that, in order to obtain the element insertion time, we divide the segment insertion, which is constant in this experiment, by the number of elements contained in one segment. On the other hand, costs increase when the number of tag names increases since more path lists must be updated in tag-list. The structure of the ER-tree also influences costs. We can see that nested ER-trees require higher costs since path lengths

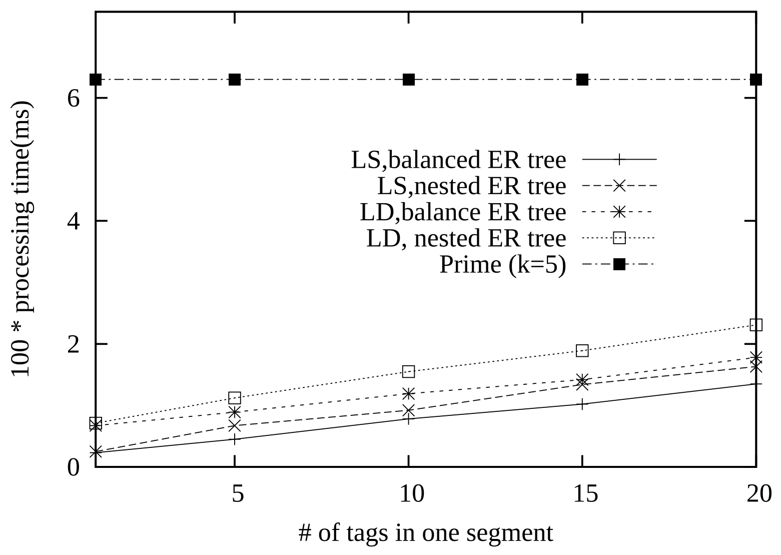


Figure 5.19: Elapsed time of inserting one element by varying the number of tag names.

increase and tag-list update cost increases as well. This is even more evident from Figure 5.20, showing how insertion costs for LD change when varying the number of segments. From the experiment results shown in Figure 5.18, 5.19 and 5.20, we can see that, as expected, insertion time varies almost linearly with respect to the number of segments. Moreover, LS is more efficient compared with LD, even if the gain in performance is very small. Since LD guarantees better query performance, it provides the best compromise between update and query processing time.

5.5 Concluding Remarks

In this chapter, we have presented a lazy approach to handle XML updates. Differently from all the other existing approaches for XML updates, under the lazy approach multiple XML elements (called XML segments) are inserted (deleted) into (from) the whole XML database without modifying element identifiers. Thus, no update to existing records in the element index is required. To support the

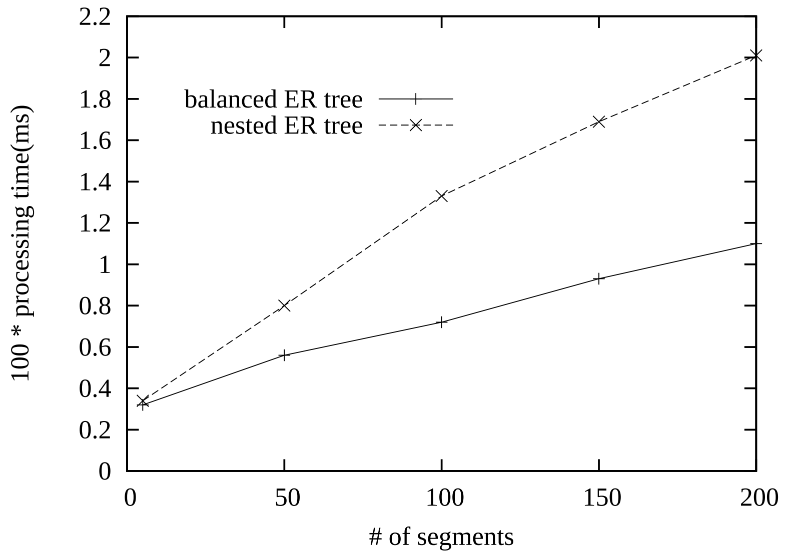


Figure 5.20: Elapsed time of inserting one element by varying the number of segments.

proposed approach, specific data structures have been designed and a structural join algorithm relying on the usage of segments has been proposed. Experimental results show that our approach outperforms other existing solutions in handling updates and in the mean time, may give better performance compared with traditional structural join algorithms, such as the one proposed in [12].

Chapter 6

Conclusion

6.1 Summary of Main Contributions

This thesis presents our solutions to solve three major issues in the area of XML document processing. The first one is the XStorM mapping scheme for mapping XML documents into relational tables. The second one is the XJoin Index, our indexing solution for efficient evaluation of branching path queries. The third one is our *lazy* approach for handling XML update.

Mapping XML data into relational tables remains the main trend of XML storage and various mapping schemes have been proposed. These schemes either generate too many small tables, which decreases query performance, or they require DTD or schema information. Our new mapping scheme, XStorM, overcomes these drawbacks by making a distinction between XML elements that represent entities in the real world, i.e., objects, and XML elements that represent properties of entities, i.e., attributes. XStorM avoids excessive fragmentation of XML data by mapping each object together with the majority of its attributes to a core relational table. We use a data-mining algorithm to identify the frequent patterns in the original XML data and we generate the core table schema from these patterns. The overflow that cannot be fit into the core relational table is stored in separate overflow tables. The names of attributes and the names of the relational tables contain structural information of the original XML file, so fast reconstruction is possible. Our performance study has demonstrated that XStorM gives good query performance, minimizes storage space and is scalable.

Structural join is generally considered a core operation for solving XML path queries, and quite a number of index structures were proposed to speed up structural join. These index structures mainly focus on how to evaluate a single structural join operation more efficiently and they do not help reduce the number of structural joins to be executed. Inspired by the join index proposed in the relational context, we have proposed a new indexing scheme, *XJoin Index*, to speed up the evaluation of branching path queries. The XJoin Index reduces the number of structural joins to be executed in branching path query processing by pre-computing some (semi-)join results, i.e., it trades space occupancy for query processing efficiency. The main features of the XJoin Index include: (i)It is simple to implement as it is entirely based on B$^+$-trees, constructed over specific tuples of values. (ii)It is flexible as it can be coupled with other structural join algorithms, and several join plans can be chosen based on the usage of the XJoin Index. We also present three possible query strategies to shrink twigs in branching path queries by applying the XJoin Index. We have conducted experiments concerning space occupancy as well as query and update time, and our experimental results show that the XJoin Index can efficiently reduce the number of structural joins to be executed, thus improving overall query performance.

XML elements are usually assigned with labels (identifiers) according to their positions in the document for efficient query processing, especially structural join. These labels are usually the keys or parts of the keys in the element indexes. Therefore, it is possible that a large number of such labels in the index need to be modified when new elements are inserted into (or removed from) the original document. To solve this problem, previous research focused on developing various dynamic labeling schemes, which are either not flexible enough or entail heavy computation. We propose a brand new *lazy* approach to efficiently handle XML documents. This lazy approach is based on the fact that XML updates tend to be done in batch manner. Multiple elements, which form what we call a *segment*, are inserted into (or removed from) the original document together. We show that an in-memory update log can be constructed to record every segment that is inserted because the number of segments is likely to be significantly less than the number of elements. With the help of this update log, we completely avoid re-labeling every element when update occurs. This segment-based update model not only improves update efficiency, but also generally improves structural join efficiency. This is because we can skip those elements that are definitely not included in the final results by the containment relationship between elements and segments. Experimental studies on both update and structural join operations show that this *lazy* approach significantly improves update efficiency, and it improves structural join efficiency most of times too.

6.2 Future Work

In this section, we discuss some limitations of our proposed work and some directions to be explored in future work.

Our work on the XStorM mapping scheme can be extended in several directions. First, the object identification algorithm described in Section 3.2.2 can only identify objects whose paths are of the same length. In some real world documents, there are objects which have paths of different lengths. Therefore, it is necessary to develop an improved method to handle this kind of data. Another possible extension is to develop a complete query re-writing mechanism that translates XML queries into SQL queries that work on top of our XStorM mapping scheme.

As described in Chapter 4, the proposed XJoin Index is able to help a query optimizer choose different query plans according to several twig shrinking strategies. If the twig pattern is complex, it is not trivial to decide which twig shrinking strategy should be applied to obtain optimal performance. One possible way that guides the selection of twig shrinking strategies is by estimating the intermediate results of structural joins before performing any selection or join operation. A number of estimation methods have been proposed in recent years [98, 78, 79, 65]. In the next phase of our work, we will construct a detailed cost model for use in query optimization algorithms. This model will allow the selection of the most efficient shrinking strategy according to existing statistics on data distribution. The estimation methods mentioned above should also be quite helpful in that regard.

The lazy update scheme proposed in Chapter 5 uses the *segment* update paradigm to avoid relabeling elements when an update occurs. The performances of both update and structural operations depend on the size and shape of the ER-tree, i.e., the total number of segments and the distribution of the segment insertion positions. As we have seen in Section 5.4, if the number of segments increases or the insertion positions of new segments are not evenly distributed, the performances of both the update and structural join processes decrease. This is simply because we need more time to modify the update log (in case of updates) or to retrieve information from the update log (in case of performing segment-based structural joins). To make the *lazy* scheme more efficient, we will explore the possibility of developing dynamic segment packing techniques, where several segments can be packed together as a whole without destroying the containment relationship between elements and segments. We believe by doing so, the size and complexity of the ER-tree could be decreased , thus making search and update operations on it more efficient.

Another possible direction is to explore is how to incorporate concurrency control into our lazy update scheme. Currently, when a new segment is inserted, we need to lock its ancestor elements, and other inserted segments with the same ancestors need to wait until the lock is released. However, update operations of the update log are not atomic operations. Therefore, it is not always necessary to lock all the ancestor segments and all information of these segments when updates take place. We intend to further investigate the relationships between segments and segment update operations, and possibly propose a more detailed model for the update operations of *update log*, and more efficient update algorithms that incorporate concurrency control.

Bibliography

[1] Dblp bibliographies. http://www.informatik.uni-trier.de/ ley/db/.

[2] Document object model (dom) specification, http://www.w3.org/dom/.

[3] eXcelon XML platform. http://www.exceloncorp.com/platform/extinfserver.shtml.

[4] Extensible Markup Language (XML). http://www.w3.org/XML/.

[5] Hypertext markup language. http://www.w3.org/MarkUp/.

[6] IBM XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[7] The standard generalized markup language. http://www.w3.org/MarkUp/SGML/.

[8] Xml document type declaration. http://www.w3.org/TR/REC-xml/.

[9] XML Path Language (XPath). http://www.w3.org/TR/xpath.

[10] XML Query (XQuery). http://www.w3.org/XML/Query/.

[11] XML Schema. http://www.w3.org/XML/Schema.

[12] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–, 2002.

[13] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. Qrs: A robust numbering scheme for xml documents. In *ICDE*, pages 705–707, 2003.

[14] J. Anderson and J.M.Bell. *Number Theory with Application.* Prentice-Hall, New Jersey, 1996.

[15] A.Schmidt, F Wass, M.Kersten, D.Florescu, L.Manolescu, M.Carey, and R.Busse. The xml benchmark project. In *Technical Report CWI*, 2001.

[16] Elisa Bertino, Barbara Catania, and Wen Qiang Wang. Xjoin index: Indexing xml data for efficient handling of branching path expressions. In *ICDE*, page 828, 2004.

[17] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, pages 64–, 2002.

[18] Philip Bohannon, Sumit Ganguly, Henry F. Korth, P. P. S. Narayan, and Pradeep Shenoy. Optimizing view queries in rolex to support navigable result trees. In *VLDB*, pages 119–130, 2002.

[19] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[20] Barbara Catania, Wen Qiang Wang, Beng Chin Ooi, and Xiaoling Wang. Lazy xml updates: Laziness as a virtue of update and structural join efficiency. In *SIGMOD Conference*, 2005.

[21] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD Conference*, 2005.

[22] Yi Chen, Susan B. Davidson, Carmem S. Hara, and Yifeng Zheng. Rrxf: Redundancy reducing xml storage in relations. In *VLDB*, pages 189–200, 2003.

[23] Yi Chen, Susan B. Davidson, and Yifeng Zheng. Constraints preserving schema mapping from xml to relations. In *WebDB*, pages 7–12, 2002.

[24] Yi Chen, Susan B. Davidson, and Yifeng Zheng. Blas: An efficient xpath processing system. In *SIGMOD Conference*, pages 47–58, 2004.

[25] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274, 2002.

[26] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. Apex: an adaptive path index for xml data. In *SIGMOD Conference*, pages 121–132, 2002.

[27] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic xml tree. In *Proceeding of the ACM International Symposium on Principles of Database Systems, pages 271-281*, 2002.

[28] Sara Cohen, Yaron Kanza, and Yehoshua Sagiv. Generating relations from xml documents. In *ICDT*, pages 285–299, 2003.

[29] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, 2001.

[30] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1998.

[31] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive xquery to sql translation using dynamic interval encoding. In *SIGMOD Conference*, pages 623–634, 2003.

[32] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with stored. In *SIGMOD Conference*, pages 431–442, 1999.

[33] Alin Deutsch and Val Tannen. Mars: A system for publishing xml from mixed and redundant storage. In *VLDB*, pages 201–212, 2003.

[34] Paul F. Dietz. Maintaining order in a linked list. In *Proceeding of the 14th Anunual ACM Symposium on Theory of Computing, pages 122-127*, 1982.

[35] Leonidas Fegaras and Ramez Elmasri. Query engines for web-accessible xml data. In *VLDB*, pages 251–260, 2001.

[36] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of xml middle-ware queries. In *SIGMOD Conference*, 2001.

[37] Daniela Florescu and Donald Kossman. Storing and querying xml data using an rdbms. In *Bulletin of IEEE Computer Society Technical Committee on Data Engineebring*, 1999.

[38] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: A system for extracting document type descriptors from xml documents. In *SIGMOD Conference*, pages 165–176, 2000.

[39] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[40] Torsten Grust. Accelerating xpath location steps. In *SIGMOD Conference*, pages 109–120, 2002.

[41] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[42] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis Viglas, Yuan Wang, Jeffrey F. Naughton, and David J. DeWitt. Mixed mode xml query processing. In *VLDB*, pages 225–236, 2003.

[43] H.Prüfer. Neuer beweis eines satzes über permutationen. In *Archiv für Mathematik und Physik*, pages 27:142–144, 1918.

[44] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: A native xml database. *VLDB J.*, 11(4):274–291, 2002.

[45] Sushant Jain, Ratul Mahajan, and Dan Suciu. Translating xslt programs to efficient sql queries. In *WWW*, pages 616–626, 2002.

[46] Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of twig queries with or-predicates. In *SIGMOD Conference*, pages 59–70, 2004.

[47] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *ICDE*, pages 253–263, 2003.

[48] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed xml documents. In *VLDB*, pages 273–284, 2003.

[49] C C Kanne and G Moerkotte. Relational storage and retrieval of xml documents. In *Techinical Report 8/99, University of Mannheim*, 1999.

[50] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of xml data. In *ICDE*, page 198, 2000.

[51] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *SIGMOD Conference*, pages 133–144, 2002.

[52] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Pradeep Shenoy. Updates for structure indexes. In *VLDB*, pages 239–250, 2002.

[53] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[54] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. An xml indexing structure with relative region coordinate. In *ICDE*, pages 313–320, 2001.

[55] Meike Klettke and Holger Meyer. Xml and object-relational database systems - enhancing structural mappings based on statistics. In *Informal Proceeding. WebDB Workshop, pages 151-170*, 2000.

[56] Donald Knuth. *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition.* Addison Wesley, Reading, MA, 1998.

[57] Rajasekar Krishnamurthy, Venkatesan T. Chakaravarthy, Raghav Kaushik, and Jeffrey F. Naughton. Recursive xml schemas, recursive xml queries, and relational storage: Xml-to-sql query translation. In *ICDE*, pages 42–53, 2004.

[58] Alexander Kuckelberg and Ralph Krieger. Efficient structure oriented storage of xml documents using ordbms. In *EEXTT*, pages 131–143, 2002.

[59] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone: Integrating structured and semistructured data. In *Proceeding of DBPL Conference, Kinloch Rannoch, Scotland*, 1999.

[60] Dongwon Lee and Wesley W. Chu. Constraints-preserving transformation from xml document type definition to relational schema. In *ER*, pages 323–338, 2000.

[61] Changqing Li and Tok Wang Ling. An improved prefix labeling scheme: A binary string approach for dynamic ordered xml. In *DASFAA*, pages 125–137, 2005.

[62] Changqing Li and Tok Wang Ling. Qed: a novel quaternary encoding to completely avoid re-labeling in xml updates. In *CIKM*, pages 501–508, 2005.

[63] Chengkai Li, Philip Bohannon, Henry F. Korth, and P. P. S. Narayan. Composing xsl transformations with xml publishing views. In *SIGMOD Conference*, pages 515–526, 2003.

[64] Hanyu Li, Mong-Li Lee, Wynne Hsu, and Chao Chen. An evaluation of xml indexes for structural join. *SIGMOD Record*, 33(3):28–33, 2004.

[65] Hanyu Li, Mong Li Lee, Wynne Hsu, and Gao Cong. An estimation system for xpath expressions. In *ICDE*, 2006.

[66] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[67] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *VLDB*, pages 193–204, 2005.

[68] Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, and Wei Ni. Efficient processing of ordered xml twig pattern. In *DEXA*, pages 300–309, 2005.

[69] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering xml queries on heterogeneous data sources. In *VLDB*, pages 241–250, 2001.

[70] Pedro José Marrón and Georg Lausen. On processing xml in ldap. In *VLDB*, pages 601–610, 2001.

[71] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[72] M.Ferná, D.Suciu, and W.Tan. Silkroute: Trading between relations and xml. In *Procceding of WWW Conference*, 2000.

[73] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[74] M.Mani and D.Lee. Xml to relational conversion using theory of regular tree grammars. In *VLDB Workshop on EEXTT*, 2002.

[75] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *ICDE*, pages 79–90, 1997.

[76] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD Conference*, pages 903–908, 2004.

[77] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260, 1995.

[78] Neoklis Polyzotis, Minos N. Garofalakis, and Yannis E. Ioannidis. Approximate xml query answers. In *SIGMOD Conference*, pages 263–274, 2004.

[79] Neoklis Polyzotis, Minos N. Garofalakis, and Yannis E. Ioannidis. Selectivity estimation for xml twigs. In *ICDE*, pages 264–275, 2004.

[80] Dallan Quass, Jennifer Widom, Roy Goldman, Kevin Haas, Qingshan Luo, Jason McHugh, Svetlozar Nestorov, Anand Rajaraman, Hugo Rivero, Serge Abiteboul, Jeffrey D. Ullman, and Janet L. Wiener. Lore: A lightweight object repository for semistructured data. In *SIGMOD Conference*, page 549, 1996.

[81] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD Conference*, pages 134–144, 2003.

[82] Raghu Ramakrishnan. *Database Management Systems*, chapter 5. Tom Casson, 1997.

[83] Praveen Rao and Bongki Moon. Prix: Indexing and querying xml using prüfer sequences. In *ICDE*, pages 288–300, 2004.

[84] Kanda Runapongsa and Jignesh Patel. Storing and querying xml data in ordbmss. In *EDBT XML-Based Data Manangement (XMLDB) Workshop*, 2002.

[85] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John E. Funderburk. Querying xml views of relational data. In *VLDB*, pages 261–270, 2001.

[86] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as xml documents. In *VLDB*, pages 65–76, 2000.

[87] Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A general techniques for querying xml documents using a relational database system. volume 30, pages 20–26, 2001.

[88] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[89] Takeyuki Shimura, Masatoshi Yoshikawa, and Shunsuke Uemura. Storage and retrieval of xml documents using object-relational databases. In *Proceeding of DEXA Conference. pages 206-217, Florence, Italy*, 1999.

[90] Adam Silberstein, Hao He, Ke Yi, and Jun Yang. Boxes: Efficient maintenance of order-based labeling for dynamic xml data. In *ICDE*, pages 285–296, 2005.

[91] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating xml. In *SIGMOD Conference*, 2001.

[92] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[93] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[94] Patrick Valduriez. Join indices. In *ACM Transactions on Database Systems, 12(2), pages 218-246*, 1987.

[95] Roelof van Zwol, Peter M.G. Apers, and Annita N. Wilschut. Modelling and querying semistructured data with moa. In *Workshop on Semi-Structured Data and NonStandard Data Formats*, 1999.

[96] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. Vist: A dynamic index method for querying xml data by tree structures. In *SIGMOD Conference*, pages 110–121, 2003.

[97] Ke Wang and Huiqing Liu. Discovering typical structures of documents: A road map approach. In *SIGIR*, pages 146–154, 1998.

[98] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Pbitree coding and efficient processing of containment joins. In *ICDE*, pages 391–, 2003.

[99] Wen Qiang Wang, Mong-Li Lee, Beng Chin Ooi, and Kian-Lee Tan. Xstorm: A scalable storage mapping scheme for xml data. In *WWW Posters*, 2001.

[100] Wen Qiang Wang, Mong-Li Lee, Beng Chin Ooi, and Kian-Lee Tan. Xstorm: A scalable storage mapping scheme for xml data. *World Wide Web*, 4(1-2):101–119, 2001.

[101] Xiaodong Wu, Mong-Li Lee, and Wynne Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE*, pages 66–78, 2004.

[102] Wang Xiao-ling, Luan Jin-feng, and Dong Yi-sheng. An adaptable and adjustable mapping from xml data to tables in rdb. In *EEXTT*, pages 117–130, 2002.

[103] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. Xrel: a path-based approach to storage and retrieval of xml documents using relational databases. *ACM Trans. Internet Techn.*, 1(1):110–141, 2001.

[104] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.

Appendix

In this appendix, we first give the details of benchmark queries used in Chapter 3 and their corresponding SQL translations and file operations(for STORED) in different mapping schemes. Then we presents the response time of executing these SQL queries in detail. We have presented these results in bar charts in Chapter 3 already. But some values presented in the bar charts are not complete because we have to set the upper bounded of the y axis to an appropriate value.

In oracle database, a table name cannot exceed 30 characters. Therefore, we need to have a name index to map tag names to numbers so that the name of core and overflow relational tables (for STORED and XStorM scheme) will not exceed the limit. The mapping is shown in Table 6.1. According to the name index, the core relational table *core_sigmodRecord_issue_article* is mapped to "*c_0_1_2*" and overflow table *of_article_authors_author* is mapped to "*o_2_7_8*".

Tag	*SigmodRecord*	*issue*	*article*	*title*	*issueNumber*
Number	0	1	2	3	4
Tag	*initPage*	*endPage*	*authors*	*author*	*description*
Number	5	6	7	8	9

Table 6.1: Tag to number mapping

Query 1. Reconstruct Object with object id “4212”

Binary Scheme:

```
select DISTINCT title.source, to_number(issuenumber.value), title.value,
to_number(initPage.value), to_number(endPage.value), author.value,
description.value from title, issuenumber, initPage, endPage, authors, author,
description where title.source = 4212 AND title.source = issuenumber.source
AND title.source = initPage.source AND title.source = endPage.source
AND title.source = authors.source AND author.source = authors.nodeID
AND title.source = description.source
```

STORED Scheme:

```
select oid, issuenumber_0, title_0, to_number(initPage_0), to_number(endPage_0),
author_0, author_1, author_2, description_0 from c_0_1_2
where oid = 4212
```

Retrieve overflow graphs under oid 4212, from overflow graphs files.

XRel Scheme:

```
select e1.docID, e1.start, e1.end, t2.value, t3.value, t4.value, t5.value, e6.index, t6.value,
t7.value from Element e1, Element e6, Text t2, Text t3, Text t4, Text t5, Text t6, Text t7,
Path p1, Path p2, Path p3, Path p4, Path p5, Path p6, Path p7
where e1.start = 2134 AND e1.end = 2759 AND e1.docID = 0
AND e1.pathID = p1.pathID AND p1.pathexp LIKE ‘#/SigmodRecord#/issue#/article’
AND t2.pathID = p2.pathID AND p2.pathexp LIKE ‘#/SigmodRecord#/issue#/article#/title’
AND t2.start > e1.start AND t2.end < e1.end AND t2.docID = e1.docID
AND t3.pathID = p3.pathID and p3.pathexp LIKE ‘#/SigmodRecord#/issue#/article#/issueNumber’
AND t3.start > e1.start AND t3.end < e1.end AND t3.docID = e1.docID
AND t4.pathID = p4.pathID AND p4.pathexp LIKE ‘#/SigmodRecord#/issue#/article#/initPage’
AND t4.start > e1.start AND t4.end < e1.end AND t4.docID = e1.docID
AND t5.pathID = p5.pathID AND p5.pathexp LIKE ‘#/SigmodRecord#/issue#/article#/endPage’
AND t5.start > e1.start AND t5.end < e1.end AND t5.docID = e1.docID
AND e6.pathID = p6.pathID AND p6.pathexp LIKE ‘#/SigmodRecord#/issue#/article#/authors#/author’
AND e6.start > e1.start AND e6.end < e1.end AND e6.docID = e1.docID
```

AND t6.pathID = p6.pathID AND t6.start > e6.start AND t6.end < e6.end

AND t7.pathID = p7.pathID AND p7.pathexp LIKE '#/SigmodRecord#/issue#/article#/description'

AND t7.start > e1.start AND t7.end < e1.end AND t7.docID = e1.docID

XStorM Scheme:

select A.oid, A.issuenumber_0, A.title_0, A.initPage_0, A.endPage_0,

A.author_0, A.author_1, A.author_2, B.attrIndex, B.value, A.description_0

from c_0_1_2 A, o_2_7_8 B

where A.oid = 4212 AND A.oid = B.oid

Query 2. Find articles that have "initPage" between 500 and 600

Binary Scheme:

select source from initPage where to_number(value) > 500

AND to_number(value) < 600

STORED Scheme:

select oid from c_0_1_2

where to_number(initPage_0) > 500 and to_number(initPage_0) < 600

XRel Scheme:

select e1.docID, e1.start, e1.end from Element e1, Path p1, Text t1

where e1.pathID = p1.pathID AND t1.pathID = p1.pathID

AND p1.pathexp LIKE '#/SigmodRecord#/issue#/article#/initPage'

AND to_number(t1.value)> 500 AND to_number(t1.value) < 600

AND e1.docID = t1.docID AND t1.start > e1.start AND t1.end < e1.end

XStorM Scheme:

select oid from c_0_1_2

where to_number(initPage_0) > 500 and to_number(initPage_0) < 600

Query 3. Find the article that has the 10th author named "Pinar Koksal" and has issuenumber equal to 15

Binary Scheme:

```
select DISTINCT authors.source from authors, author, issuenumber
where author.source = authors.nodeID AND authors.source = issuenumber.source
AND to_number(issuenumber.value) = 15 AND author.ordinal = 9
AND author.value = ‘Pinar Koksal’
```

STORED Scheme:

```
select DISTINCT oid from c_0_1_2
where to_number(issuenumber_0) = 15
```

Retrieve “author” overflow graphs with index 9 and value “Pinar Koksal” from overflow graph files

XRel Scheme:

```
select e1.docID, e1.start, e1.end
from Element e1, Element e2, Text t2, Text t3, Path p1, Path p2, Path p3
where e1.pathID = p1.pathID AND p1.pathexp LIKE ‘#/SigmodRecord#/issue#/article’
AND e2.pathID = p2.pathID
AND p2.pathexp LIKE ‘#/SigmodRecord#/issue#/article#/authors#/author’
AND e2.start > e1.start AND e2.end < e1.end AND e2.docID = e1.docID
AND e2.index = 9 AND t2.pathID = p2.pathID AND t2.start > e2.start
AND t2.end < e2.end AND t3.pathID = p3.pathID
AND p3.pathexp LIKE ‘#/SigmodRecord#/issue#/article#/issueNumber’
AND t3.start > e1.start AND t3.end < e1.end
AND t3.docID = e1.docID AND to_number(t3.value) = 15
```

XStorM Scheme:

```
select DISTINCT core_0_1_2.oid from c_0_1_2, o_2_7_8
where c_0_1_2 = o_2_7_8.oid AND to_number(issuenumber_0) = 15
AND attrIndex = 9 AND value = ‘Pinar Koksal’
```

Query 4. Find articles that have first author ‘Dallan Quass’ and 7th author ‘Svetlozar Nestorov’ or just first author ’Kenneth A. Ross’ (no 7th author)

Binary Scheme:

select DISTINCT A1.source from authors A1, author A2
*where A2.source = A1.nodeID AND A2.ordinal = 0 and A2.value = 'Kenneth A. Ross' AND NOT EXISTS (select * from author A3 where A3.ordinal = 6 AND A3.source = A2.source)*
UNION
select DISTINCT A1.source from authors A1, author A2
where A2.source = A1.nodeID AND A2.ordinal = 6 AND A2.value = 'Svetlozar Nestorov' AND A1.source IN (select DISTINCT A3.source from authors A3, author A4 where A4.source = A3.nodeID AND A4.ordinal = 0 and A4.value = 'Dallan Quass')

STORED Scheme:

select DISTINCT oid from c_0_1_2 where author_0 = 'Kenneth A. Ross'
Find "author" overflow graph with value "Svetlozar Nestorov" and ordinal 6

select DISTINCT oid from c_0_1_2 where author_0 = 'Dallan Quass'
Check "author" overflow graphs with oids returned from above SQL query, remove those oids with ordinal 6.

XRel Scheme:

select e1.docID, e1.start, e1.end
from Element e1, Element e2, Element e3, Text t2, Text t3, Path p1, Path p2
where e1.pathID = p1.pathID AND p1.pathexp LIKE '#/SigmodRecord#/issue#/article'
AND e2.pathID = p2.pathID
AND p2.pathexp LIKE '#/SigmodRecord#/issue#/article#/authors#/author'
AND e2.start > e1.start AND e2.end < e1.end AND e2.docID = e1.docID
AND e2.index = 0 AND t2.pathID = p2.pathID AND t2.docID = e2.docID
AND t2.start > e2.start AND t2.end < e2.end AND t2.value = 'Dallan Quass'
AND e3.pathID = p2.pathID
AND e3.start > e1.start AND e3.end < e1.end AND e3.docID = e1.docID
AND e3.index = 6 AND t3.pathID = p2.pathID AND t3.docID = e3.docID

AND t3.start > e3.start AND t3.end < e3.end AND t3.value = 'Svetlozar Nestorov'

UNION

select e1.docID, e1.start, e1.end from Element e1, Element e2, Text t2, Path p1, Path p2

where e1.pathID = p1.pathID AND p1.pathexp LIKE '#/SigmodRecord#/issue#/article'

AND e2.pathID = p2.pathID

AND p2.pathexp LIKE '#/SigmodRecord#/issue#/article#/authors#/author'

AND e2.start > e1.start AND e2.end < e1.end AND e2.docID = e1.docID

AND e2.index = 0 AND t2.pathID = p2.pathID AND t2.docID = e2.docID

AND t2.start > e2.start AND t2.end < e2.end AND t2.value = 'Kenneth A. Ross'

*AND NOT EXISTS (select * from Element e3, Element e4*

where e3.docID = e1.docID AND e3.pathID = p1.pathID

AND e4.pathID = p2.pathID AND e4.start > e3.start

AND e4.end < e3.end AND e4.docID = e3.docID

AND e4.index = 6)

XStorM Scheme:

select DISTINCT C.oid from c_0_1_2 C where C.author_0 = 'Kenneth A.Ross'

*AND NOT EXISTS (select * from o_2_7_8 O where O.oid = C.oid and O.attrIndex*

= 6)

UNION

select DISTINCT oid from c_0_1_2 where author_0 = 'Dallan Quass' AND oid

IN (select DISTINCT oid from o_2_7_8 where attrIndex = 6 and value =

'Svetlozar Nestorov')

Query 5. Find articles that have initPage = 388 or endpage = 2 or 7th author 'Svetlozar Nestorov'

Binary Scheme:

select source from initPage where to_number(value) = 388

UNION

select source from endPage where to_number(value) = 2

UNION

select authors.source from author, authors where author.source = authors.nodeID

AND author.ordinal = 6 AND author.value = 'Svetlozar Nestorov'

STORED Scheme:

select oid from c_0_1_2 where to_number(initPage_0) = 388 OR

to_number(endPage_0) = 2

Find "author" overflow graph with value "Svetlozar Nestorov" and ordinal 6

XRel Scheme:

select e1.docID, e1.start, e1.end from Element e1, Text t2, Path p1, Path p2

where e1.pathID = p1.pathID AND p1.pathexp LIKE '#/SigmodRecord#/issue#/article'

AND t2.pathID = p2.pathID AND p2.pathexp LIKE '#/SigmodRecord#/issue#/article#/initPage'

AND e1.docID = t2.docID AND t2.start > e1.start AND t2.end < e1.end

AND to_number(t2.value) = 388

UNION

select e1.docID, e1.start, e1.end from Element e1, Text t2, Path p1, Path p2

where e1.pathID = p1.pathID AND p1.pathexp LIKE '#/SigmodRecord#/issue#/article'

AND t2.pathID = p2.pathID AND p2.pathexp LIKE '#/SigmodRecord#/issue#/article#/endPage'

AND e1.docID = t2.docID AND t2.start > e1.start AND t2.end < e1.end

AND to_number(t2.value) = 2

UNION

select e1.docID, e1.start, e1.end from Element e1, Element e2, Text t2, Path p1, Path p2

where e1.pathID = p1.pathID AND p1.pathexp LIKE '#/SigmodRecord#/issue#/article'

AND e2.pathID = p2.pathID

AND p2.pathexp LIKE '#/SigmodRecord#/issue#/article#/authors#/author'

AND e2.docID = e1.docID AND e2.start > e1.start AND e2.end < e1.end

AND e2.index = 6 AND t2.docID = e2.docID AND t2.pathID = p2.pathID

AND t2.start > e2.start AND t2.end < e2.end AND t2.value = 'Svetlozar Nestorov'

XStorM Scheme:

select DISTINCT oid from c_0_1_2

where to_number(initPage_0) = 388 OR to_number(endPage) = 2

UNION

select DISTINCT oid from o_2_7_8 where attrIndex = 6

AND value = 'Svetlozar Nestorov'

Query 6. Find articles that have attribute, issuenumber, title, initPage and 9 authors

Binary Scheme:

select source from issuenumber INTERSECT

select source from title INTERSECT

select source from initPage INTERSECT

select DISTINCT authors.source from author, authors

where author.source = authors.nodeID AND author.ordinal = 8

STORED Scheme:

select DISTINCT oid from c_0_1_2

where issuenumber_0 IS NOT NULL AND title_0 IS NOT NULL

AND initpage_0 IS NOT NULL AND author_0 IS NOT NULL

AND author_1 IS NOT NULL AND author_2 IS NOT NULL

Check "author" overflow graphs with oids returned from above SQL query,

Remove oids that do not have corresponding overflow graphs with ordinal 8.

XRel Scheme:

select e1.docID, e1.start, e1.end from Element e1, Element e2, Path p1, Path p2

where e1.pathID = p1.pathID AND p1.pathexp = '#/SigmodRecord#/issue#/article'

AND e2.pathID = p2.pathID

AND p2.pathexp = '#/SigmodRecord#/issue#/article#/issueNumber'

AND e2.docID = e1.docID AND e2.start > e1.start AND e2.end < e1.end

INTERSECT

select e1.docID, e1.start, e1.end from Element e1, Element e2, Path p1, Path p2

where e1.pathID = p1.pathID AND p1.pathexp = '#/SigmodRecord#/issue#/article'

AND e2.pathID = p2.pathID AND p2.pathexp = '#/SigmodRecord#/issue#/article#/title'

AND e2.docID = e1.docID AND e2.start > e1.start AND e2.end < e1.end

INTERSECT

select e1.docID, e1.start, e1.end from Element e1, Element e2, Path p1, Path p2

```
where e1.pathID = p1.pathID AND p1.pathexp = '#/SigmodRecord#/issue#/article'
AND e2.pathID = p2.pathID AND p2.pathexp = '#/SigmodRecord#/issue#/article#/initPage'
AND e2.docID = e1.docID AND e2.start > e1.start AND e2.end < e1.end
INTERSECT
select e1.docID, e1.start, e1.end from Element e1, Element e2, Path p1, Path p2
where e1.pathID = p1.pathID AND p1.pathexp = '#/SigmodRecord#/issue#/article'
AND e2.pathID = p2.pathID AND p2.pathexp = '#/SigmodRecord#/issue#/article#/authors#/author'
AND e2.docID = e1.docID AND e2.start > e1.start
AND e2.end < e1.end AND e2.index = 8
```

XStorM Scheme:

```
select DISTINCT c_0_1_2.oid from c_0_1_2, o_2_7_8
where c_0_1_2.oid = o_2_7_8.oid AND issuenumber_0 IS NOT NULL
AND title_0 IS NOT NULL AND initpage_0 IS NOT NULL
AND author_0 IS NOT NULL AND author_1 IS NOT NULL
AND author_2 IS NOT NULL AND attrIndex = 8
```

Query Response Time (in ms)

Query 1:

XML Size	*Binary*	*XRel*	*STORED*	*XStorM*
1MB	240	265	211	203
10MB	387	454	331	323
20MB	452	662	421	403
40MB	465	832	451	454
100MB	798	1143	532	503

Query 2:

XML Size	*Binary*	*XRel*	*STORED*	*XStorM*
1MB	231	251	261	241
10MB	243	254	273	252
20MB	250	262	272	264
40MB	246	271	283	276
100MB	253	432	402	398

Query 3:

XML Size	*Binary*	*XRel*	*STORED*	*XStorM*
1MB	221	245	231	202
10MB	265	304	273	232
20MB	276	342	421	265
40MB	331	411	489	307
100MB	20342	22431	2031	387

Query 4:

XML Size	*Binary*	*XRel*	*STORED*	*XStorM*
1MB	230	254	244	212
10MB	395	467	432	347
20MB	578	511	653	567
40MB	804	723	853	767
100MB	43564	46342	3112	1213

Query 5:

XML Size	*Binary*	*XRel*	*STORED*	*XStorM*
1MB	244	256	214	215
10MB	311	324	272	255
20MB	521	413	321	304
40MB	783	721	433	426
100MB	39821	41567	1768	1254

Query 6:

XML Size	*Binary*	*XRel*	*STORED*	*XStorM*
1MB	321	354	278	273
10MB	678	702	342	311
20MB	1143	1204	467	386
40MB	1764	1775	611	435
100MB	5342	5873	2343	1134