Purifying XML Structures
XML 構造の純化

by

Taro L. Saito
斉藤 太郎

A Doctoral Dissertation

Submitted to
the Graduate School of the University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

December 15th, 2006

Thesis Supervisor: Haruo Hosoya　細谷 晴夫
Assistant Professor of Computer Science

Abstract

XML has a tree structure, which allows various data expressions. In this work, we show the well-known notion of the functional dependency has an essential role in XML data modeling. The existing proposals of functional dependencies for XML are based on fixed paths, imposing severe structural constraints on XML data. We address this problem by introducing node-based functional dependencies so as to allow various XML structures to describe complex data. The drawback of this approach might be that querying all possible XML structures could result in costly computation. To overcome this, we introduce a novel query method, called *amoeba join*, and its optimization technique that selectively retrieves XML structures satisfying functional dependencies. Our approach brings a new perspective into XML query processing; no explicit path is required within query statements, since XML structures of interest are automatically determined by the functional dependencies. We call this conceptual change as the *purification* of XML structures, since it hides complexities of the data structure from the user. In addition, by using keys derived from functional dependencies, aggregation of differently structured XML sources becomes quite smooth. These techniques facilitate management of multiple hierarchies of XML data, and significantly enhances the expressive power of XML databases. To make the query processing feasible, we also show an efficient XML index that can be implemented on top of the B+-tree. Our experimental results confirm the utility and availability of the proposed methods. We also discuss the effective usage of our method to model scientific data, and the impact of this research on the future of XML data management.

論文要旨

木構造を持つ XML では様々なデータ表現が可能である。本論文では、データベース理論でよく知られている「関数従属性（Functional Dependency）」という概念が XML のデータモデリングにおいて重要な役割を担うことを示す。これまで XML 向けに提案されてきた関数従属性は固定されたパスに対して定義されていたため、XML の構造に必要以上に強い制約を課すものであった。この問題に対して、我々は、パスではなくノードを基準に関数従属性を導入し、XML データの表現に自由度与えることを試みた。この方式の問題点は、様々な構造で表現された XML データをすべて問い合わせするコストが高くつかもしれない、ということである。そのため、我々は amoeba join（アメーバ結合）という新しい問い合わせ手法を考案し、関数従属性を満たす XML の構造に狙いを定め、それらを効率よく取り出す技術を開発した。これらの手法は、XML の問い合わせにパスのような XML の構造を指定しなくても、目的となる木構造を関数従属性から自動的に定めることができるという点で、XML の問い合わせに新しい観点をもたらすものである。複雑な XML の構造をユーザーの目から隠し、関数従属性から構造を決めてしまうという、この発想の転換を「XML 構造の純化」と呼ぶ。さらに、関数従属性から派生して定義されるキーを用いることで、異なった構造を持った XML データを容易に集約することができるようになる。これらの技術は、複数の階層構造を持った XML データを管理するための障害を取り除き、XML データベースの表現力を高めるのにも役立つ。 また、問い合わせのために必要な、B+木上で実装可能な XML の索引構造についても紹介する。実験では、提案された手法の有用性と、それが実現可能であることを示す。そして、本研究の成果が科学データの表現にどのように活用でき、今後、XML データの管理にどのようなインパクトをもたらすのかについて議論する。

Acknowledgements

I would like to note here that I cannot complete this work without tremendous helps from my supervisors, colleagues, and my family.

I have been researching on XML databases since I was a undergraduate student, but I was not satisfied with the results I could achieve, because I always had a feeling that even if I pursue the development of the XML databases, a gap between the user and complexity of the system would remain; XML, which is a merely a text, provides a capability of describing data with ease. However, managing them, e.g. updating or querying XML, is far from easy, rather, frustrating for the user.

I had considered this gap can be filled with supporting systems, such as wizard or gorgeous user interfaces. As a student, I did not have enough human resources to develop both of the DBMS and such a highly sophisticated interfaces simultaneously; I got discouraged, but there was no need to do that. What gave me a new insight to this research is the experiences of developing SCMD (Saccharomyces Cerevisiae Morphological Database). I thank the members of SCMD project. I could learn a lot about managing databases in practice and its difficulties, and finally I could reach the idea of this thesis; querying XML without using *structures*. This concept simplifies the problems that had been bothering me. In addition, its implementation requires no more human resources other than myself.

Again, I thank my wife, Naoko, for supporting me, and our three-year-old son, Yui. I want to compensate for the time I couldn't play with you, struggling to write this thesis.

Taro L. Saito,
December 15th, 2006

Contents

List of Figures

Chapter 1

Introduction

1.1 XML

XML *(Extensible Markup Language)* [9] is now a global standard for describing structured data. Although there should be many reasons why XML has been so widely accepted, XML certainly has many interesting properties for achieving the current status. First, XML has a tree-structured data model; it can be used to describe various types of data without difficulties. Second, XML is merely a text, so there are few obstacles to read and write XML data for both of human and computers. That gives incentives to use XML data for many people; in fact, many organizations, including industry, academia, and also governments of several nations, have adopted XML to manage their own information.

In its infancy of XML (that is before 1997), it had been considered that XML is a format to describe relatively small size of data, and thus its usage was largely to send and receive text messages between applications or through the Internet. As XML has become more popular, it has been used not only as a small text format, but also as data stored in database management systems (DBMS). Until 2006, many vendors, including the Big-Three suppliers of relational databases (IBM, Microsoft, and Oracle), have launched brand-new XML database engines. This trend will certainly result in increased XML capability, and the potential handling size and capacity of XML data is quite huge.

1.2 Shortcomings of XML Query

A scenario, *'XML as a database'*, has evolved toward reality. Nevertheless, inconveniences of XML data have already materialized; to locate information of interest from such a large XML database is quite difficult without knowing its detailed structure. XML is a self-describing data; that means the structure of XML data is defined within the XML data itself. However, to explore an XML database, path-based query languages such as XPath [15] or XQuery [8] have been frequently used. XPath utilizes path-expression patterns to specify the target elements of XML. XQuery is an extension of the XPath, which adds some facilities to XPath such as loop operators, variables, etc. In these methods, the user has to explicitly specify path structures of interest within the query statements. In reality, however, it is usually the case that the user do not have such detailed knowledge about the actual instances of data structures within the database.

A summary of path structures, e.g. DataGuides [18] or 1-index [35], aggregates all kinds of paths occurred in the XML database, however, it is not sufficient to comprehend the actual data structure in the target context of the XML data; a path occurred in the some context may not appear in a different context. A schema of XML, e.g. DTD [9], XML Schema [53], and RelaxNG [41], to some extent resolves this uncertainty of path occurrences, but not entirely, since these types of schemas allow optional appearance of elements; instances of path structures still vary depending on the context within the XML data.

Hence, even if we have a schema or summary of path structures, the XML database is still like a black box for the user; writing correct path queries for specific contexts remains difficult. To remedy this problem, considerable effort and intensive research has been put into XML structure indexes, allowing the processing of descendant axis queries that require less knowledge of path structures. For example, with an XPath query //book, we can locate the book nodes in arbitrary depth from an XML document. However, this trend of XML indexing might not be addressing the real goal; People are enthusiastic about querying *data structures*, when they should be focusing on *structured data*; If we pursue writing paths in order to perform queries, we must first somehow acquire knowledge of path structures. To learn the path structure for some specific context, we must issue queries to a 'black box' database. This is a chicken-or-egg situation — which comes first, a path structure or query?

1.3 Structural Fluctuation

To address the above problem, relaxation of structural constraints in XPath queries is proposed when it is difficult to specify precise paths [4]. For example, an XPath query A/B can be relaxed to A//B by replacing the parent-child axis with the ancestor-descendant axis. This process reduces the burden of writing exact matches of path queries. However, the following example illustrates a problem, which is not normally identified in the context of query relaxation:

```
<order id="1">                             <book isbn="xx1">
  <book isbn="xx1"/>                         <order id="1">
  <customer id="c001"/>         ⇔              <customer id="c001"/>
</order>                                     </order>
                                           </book>
```

Figure 1.1: An example of structural fluctuation

The two XML fragments shown above (**Figure 1.1**) represent data with the same meaning, but with different structures: the hierarchies of order and book tag is reversed. We call this *structural fluctuation* in XML. It is a structural variation in XML fragments that have the same elements (e.g., order, book, and customer) and different structures.

XPath can track structural fluctuations, using disjunction in path patterns. For example, finding element pairs of order and book in **Figure 1.1** requires the concatenation of at least two types of XPath query; /order/book and /book/order. In general, however, query statements become more complex because there could be many more elements to query and structural fluctuations in the document. For example, the number of query trees required to cover all structural fluctuations consisting of order, book, and customer elements is $3^{3-1} = 9$ (**Figure 1.2**) because it is identical to the enumeration of all labeled trees with n nodes, when the differences in axis (// or /) are ignored. Its enumeration size is known to be n^{n-1}. Thus, the number of XPath expressions required to cover all possible path structures can easily balloon. Moreover, concatenating all n^{n-1} query trees into a single path expression can be a daunting task.

Although the existence of structural fluctuations is a serious obstacle to manage XML

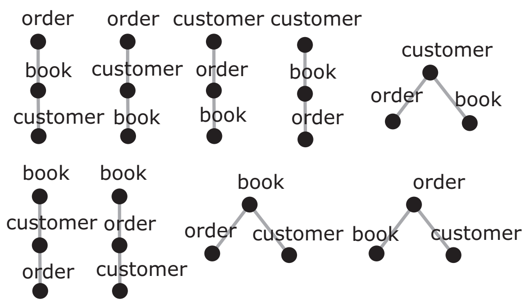


Figure 1.2: Structural fluctuations of order, book and customer nodes, the number of which is $n^{n-1} = 3^2 = 9$.

databases, this problem is caused by the flexibility of XML, which is one of the most beneficial properties of XML that allows various structures to describe data. We consider this property of XML should be utilized to enhance the expressive power of XML data. Unfortunately, however, there has been no feasible method to manage structural fluctuations. In this thesis, we introduce the notion of *amoeba*, which represents equivalent classes of XML structures, to handle structural fluctuations. For example, an amoeba of order, book and customer nodes, denoted ⟪order, book, customer⟫, can cover all structures in **Figure 1.2** with this single expression.

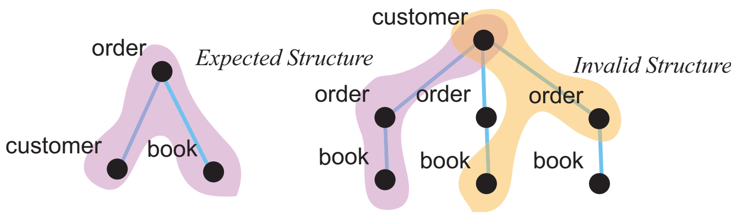


Figure 1.3: Structural fluctuations may contain invalid structures.

1.4 Purifying XML Structures with Functional Dependencies

A challenging problem is to query fluctuated XML data. We call this operation *amoeba join*. However, its computation is potentially costly because their instances could be numerous. For example, **Figure 1.3** shows several combinations of customer, order and book nodes. In this example, each order is expected to have a single book node. However, the computation of all possible structural fluctuations may contain unexpected structures, which connect irrelevant book and order nodes (the right-most one in **Figure 1.3** is one of the example), and also the number of such inappropriate structures might be enormous since we have to consider all of the permutations. To efficiently retrieve XML structures of interest, we have to consider not only structural fluctuations, but also *semantics* of XML data, such as each order node has a unique book node. For this purpose, we introduce *functional dependencies* for XML.

Functional dependency (FD) is a foundation of the database systems, which defines associations of data [2, 39]. For example, consider the following constraint: two persons with the same id (pid) value must have the same name. This rule can be described with the functional dependency; pid → name. In relational databases, a functional dependency $X \to Y$, where each X, Y is a set of attribute names, means if two tuples p, q agree on attribute X, denoted $p.X = q.X$, then $p.Y = q.Y$ must hold. Such semantic restrictions on data have important roles to sophisticate the database design, especially avoiding redundancies of data, which usually cause update anomalies. When Y is a full set of column names in a table, X is called a *key* or *primary key*, which is a special case of functional dependency, and indispensable to uniquely locate data tuples of interest.

The notion of functional dependency itself can be used independently from types of database systems, such as relational, or XML databases, etc. Therefore, in order to introduce functional dependencies, we have to define their mappings to the targeted database system. Relational databases use a flat data model comprising of tables and columns, thus applying functional dependencies $X \to Y$ is straightforward; each X, Y is a column name in a table. However, the same scenario cannot be applied to XML, since the structure of XML is a tree, and the structure itself adds some meanings to the data.

Functional dependency for XML has been studied from various perspectives. Buneman et al. discussed FDs in the form of keys for XML [10, 11]. The work of Lee et al. attempted to apply FDs to XML [32], and a more general definition of FDs for XML is provided by Arenas and Libkin [5]. Their approaches are based on paths; given sets X, Y of paths, an FD for XML is defined as $X \to Y$. However, these path-based definitions of FDs cannot handle XML documents containing structural fluctuations.

With path-based functional dependencies, for example, /order → /order/book, only the left-hand side structure in **Figure 1.1** is allowed. In general, however, the number of all possible structural fluctuations with n nodes of XML is n^{n-1}. It is highly restrictive to allow only one structure among n^{n-1} candidates.

In reality, however, a feasible treatment of this problem is to disallow structural fluctuations using a schema of XML. Although this method significantly limits the flexibility of XML data modeling, the users of XML have no other sensible solution, since without rigorous knowledge of data structures, it is quite difficult to correctly parse and interpret tree-structured XML data. For example, the standard of XML processing, e.g. SAX, DOM, XPath [15], etc. provides no simple way to handle structural fluctuations. This fact indicates that a change of data structure obliges programmers to modify the XML reader programs. Therefore, structures of XML are often limited by programming easiness.

Our initial motivation of this research comes from this inconvenience of XML data management. We observed that there can be various tree structures to represent semantically identical data. In the example of **Figure 1.1**, suppose every order node has a unique book information; this constraint is expressed by order → book. However, it does not matter whether book is a child of an order or vice versa as long as they are *structurally related* in the XML data, so specifying a fixed path, such as order/book, becomes less important. To handle various path structures as a whole, we can utilize the notion of amoeba. For example, structural fluctuations of 《order, book, customer》 that satisfies the FD order → book are described with a boolean combination of amoeba structures, that is 《order, book》 and 《order, book, customer》. This combination of structural constraints can eliminate the invalid XML structures which do not directly connect book and order nodes. For example, the upper-right structure in **Figure 1.3** is not allowed with this constraints. We generalized this notion of structural constraints as *amoeba domain*, which has a fundamental role to make correspondence between FDs and XML structures.

Previous work of FDs for XML [5, 10, 11, 32] infers functional dependencies from a path structure of an XML document. In contrast, our approach assumes FDs are defined outside of the XML data, and each FD is specified by using node names, e.g. tag or attribute names. Within the XML document, these nodes must satisfy structural constraints implied by FDs. This node-based definition of functional dependencies allows various data expressions compared to the path-based definitions, and consequently makes easier to design XML databases significantly.

Furthermore, this conceptual change from a path to node brings a new insight into XML query processing. For a long time since the specification of XML has appeared, it has been believed that path structure is required to query structured data. XPath [15] and XQuery [8], which are the de facto standard of XML query processing languages, are examples that require explicit path structures to perform queries. However, if functional dependencies define structures of XML data, there is no need to specify structures within query statements; what all we need to query XML data is to describe target nodes of interest with tag names, predicates, keywords etc. XML structures consisting of these target nodes can be determined automatically from a set of FDs. In this sense, the functional dependency can be a strong tool to *purify* XML structures

This replacement of XML structures with functional dependencies is essensial to overcome a dillemma when using XML data, that is, we have to employ hierarchical orders of XML nodes in order just to represent *relationships* of data; a form of the tree structure in an XML data is not always meaningful to the database user. For example, in **Figure 1.3**, the hierarchical order of order and customer nodes has no significant meaning as long as they are connected in the XML data, so both of the representations order/customer and customer/order can be used. However, a relational table representation using a schema (order, customer) cannot be used to substitute this XML data, since if we have to describe a situation that a customer has many orders, tree structures of XML are still required; we need an XML representation in which a customer node has many order nodes. To describe this type of a data model in a relational database, we have to separately manage customer and order nodes, then connect them with an additional table, e.g. (customer_id, order_id), which represents a relation of customer and order nodes. Managing three tables demands facilities of a DBMS, while an XML format can be described as a simple text data.

Even if we know a functional dependency, e.g. order → customer, is satisfied in a given data model, we have to choose one of the tree structures consisting of order and customer nodes in order to utilize the benefit of XML data, which can be described with a text. In other words, a tree structure is required to represent relationships of data but the hierarchical order is not. Thus, it is often the case hierarchical orders in an XML data are irrelevant to the semantics of the data. However, the current technologies for querying XML data, such as XPath or XQuery, demand explicit hierarchical structures within path-expressions, e.g. /order/customer, /customer/order, etc. By utilizing functional dependencies for querying XML data, we can simplify query statements by removing path structures irrelevant to the data model. And also, with the help of functional dependencies, we can reduce the burden of fixing a structure of XML data for the ease of traditional XML query processing methods, such as XPath or XQuery, etc. Hence, we can achieve the full benefits of XML data as a data representation format, while purification of XML structures with functional dependencies makes easier for both of the users and database managers to concentrate on the data models, rather than bothering to choose one of the tree structures from many candidates representing the same data model.

As another benefit of using functaional dependencies, we can improve the amoeba join processing so that it can efficiently retrieve XML structures implied by functional dependencies. In this thesis, we present an optimization technique, called *amoeba-join decomposition*, which produces an equivalent query schedule consisting of a set of nested amoeba joins. With this decomposition technique, we can avoid computing irrelevant data structures, and makes order of magnitude faster the amoeba join processing.

1.5 A Motivating Example from SCMD

Managing structural fluctuations is a key to enhance the expressive power of XML databases. Here, we explain its reason, illustrating our experiences of developing SCMD (*Saccharomyces Cerevisiae* Morphological Database) [47], which is a Bioinformatics database that collects information of yeast cells to identify gene functions. The process of detecting gene function was mainly conducted by some kinds of clustering analysis and statistical methods [36, 48], thus we had to arrange the cell data to report the results of these analysis. **Figure 1.4** shows its simplified example; clustering of cells according to their sizes:

```
<cell_list>                                        <clustering_result>
  <group>                                            <group>
    <cell id="1"><size>large</size></cell>             <size>large</size>
    <cell id="2"><size>small</size></cell>             <cell id="1"/>
    <cell id="3"><size>large</size></cell>             <cell id="3"/>
  </group>                                           </group>
</cell_list>                                       </clustering_result>
```

Figure 1.4: Clustering of cells according to size

In the left-hand side of the XML data, each size element is contained in a cell element, while in the right-hand XML data, the single size element belongs to the parent group element to avoid duplicates of size data. To retrieve information of cell sizes, we need two different XPath queries; //group/cell/size and //group[cell]/size, one of which demands a twig-query processing. This example indicates that we have to rewrite queries when we reorganize data structures of XML. In the meantime of the SCMD project, however, we had to perform a tremendous number of data organizations. It is tedious since these structural change has no significant meaning other than the readability or space efficiency. That is why we use the notion of functional dependency as a method to represent semantics of XML structures. The title of this work, *'Purifying XML Structures'* means to extract the essence of semantics implied by XML structures in order to query XML data using a more instinctive method than path structures.

1.6 XML Data Modeling

With the capability of managing structural fluctuations, we can provide an elegant solution to XML data modeling; the use of multiple hierarchies to describe XML data. **Figure 1.5** illustrates a book store data using a tree model of XML. This book store manages a list of books, customers and their book orders. In relational databases, each data of books, customers and orders must be contained in a table, so their relationships, for example a customer has several orders, are described using foreign keys [39], which define connections between table data. In XML, these relationships can be described with structure, for example, a customer node has several order nodes as its children (center of the **Figure 1.5**). However, there is a case we want to manage order data separately from customers; for example, to create a list of orders (right-hand side of **Figure 1.5**). In this situation, it is convenient to arrange data structures so that each order node will be a parent node of a customer. XML processors, such as SAX, DOM, XPath etc., which trace XML data according to the tree structure, require as many programs as the number of the structural variants. In this paper, we show our method needs only a single query expression to retrieve various structures. This removes obstacles to use multiple hierarchies to represent the same meaning data.

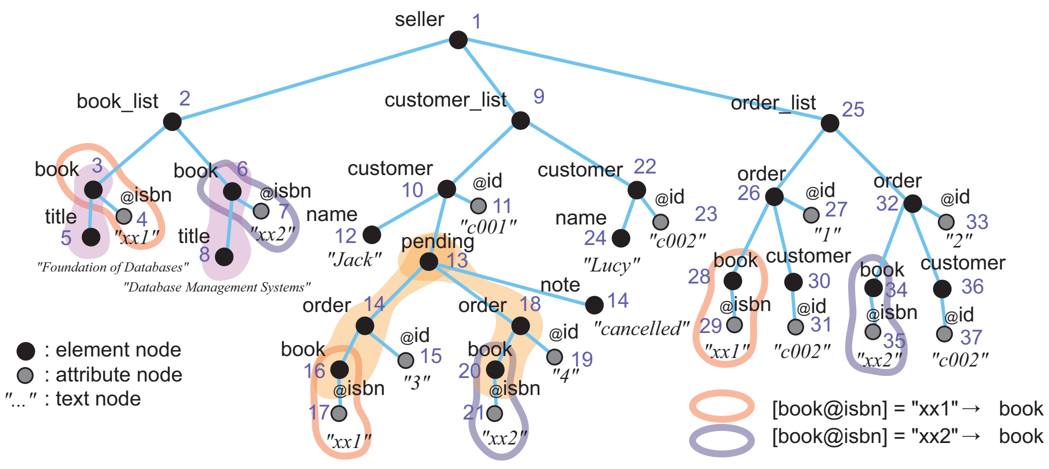


Figure 1.5: Managing various data structures within an XML document. Equivalent classes of XML nodes are shown with the colored circles

```
(Q1):   for $x in /seller/book_list/book
            $y in /seller/customer_list/customer/pending/order/book
        where $x/@isbn = $y/@isbn
        return $x/title

(A1):   AJ(book, [pending, order, title])
```

Figure 1.6: Queries for book titles, the order of which is pending

When managing multiple hierarchies of XML data, we also have to address the problem of data redundancies. For example, in **Figure 1.5** each title of a book should be presented once in the database to avoid redundancies. However, each order node requires a reference to a book and maybe its title. If we duplicate the information of a book title to each order node, it may cause update anomalies, so such database design should be avoided. However, this means we have to know which book node contains the title information. Requirement on such knowledge of XML structure is an serious obstacle to the management of large XML databases without any schema.

For example, consider to query book titles whose orders are pending in **Figure 1.5**. A query (Q1) in **Figure 1.6** illustrates this operation written in XQuery [8], which traverses book_list and customer_list elements separately, and performs value-based join on book/@isbn. The deficit of this query is that the user has to know the entire structure of XML document; the title of a book appears only under the book_list element, and the pending node is under the customer_list.

To overcome this problem, we utilize FDs to define foreign keys of XML nodes, called *ubiquitous keys*. For example, a ubiquitous key [book@isbn] $\rightarrow$ book, where $[n]$ denotes the text value of a node n, defines equivalent classes of book nodes whose isbn values are identical. In **Figure 1.5**, for example, the set of book nodes, $\{3, 16, 28\}$, forms one class: [book@isbn] = ”xx1”, and the set $\{6, 20, 34\}$ constitutes another: [book@isbn] = ”xx2”.

A query (A1) in **Figure 1.6** is an example that uses amoeba join and ubiquitous keys. This query first computes structural fluctuations consisting of book and title; (3, 5) and (6, 8), and another structure consisting of book, pending, order nodes; (16, 13, 14) and (20, 13, 18), then looks at book nodes of 3 and 6 that are equivalent to 16 and 20 respectively according to the FD [book@isbn] $\rightarrow$ book, and obtains the respective book titles in nodes 5 and 8. This operation enables the uses to perform queries without detailed knowledge of structures of XML. In addition, query expressions become considerably simpler compared to the path-based query methods.

1.7 XML Indexing

While the amoeba join with functional dependencies has a capability of querying XML data without using structures, we still need an efficient method to retrieve XML nodes from a database by using a variety of structural properties of XML, which are basically derived from *tree structures* of XML such as document order of nodes, subtree, sibling, ancestor, descendant nodes, etc. The other properties are *path structures* of XML, that consist of sequences of tag and attribute names, e.g //book/titile. These various aspects of XML make its query processing difficult, and index structures for this purpose, which are generally called the *structure indexes* [29], have attracted research attention, especially to accelerate the evaluation of XPath [15] queries, which is the *de facto* standard for navigating XML. XPath contains a mixture of tree and path traversal with several axis steps, e.g. the child-axis (/), the descendant-axis (//), ancestor-axis, sibling-axis, etc.

As XML gradually has established its position as a data representation format, tremendous number of structure indexes have been proposed, which are optimized for specific query patterns, including structural joins [33, 14], twig queries [27], suffix paths [57], ancestor queries [26], etc. This enthusiasm caused serious obstacles; compared to the number of proposed indexes, there has been quite a few number of comparative studies of these approaches. Furthermore, these comparisons are not sufficient since they cannot cover up all types of indexing methods in a single paper, so we still cannot evaluate which index is genuinely a good one. Every approach uses different implementation or specialized index structures, and hence it becomes quite a difficult task to compare performances of XML indexing methods. For example, some indexes pursue the space efficiency of indexes, such as compression techniques of node labels [37, 30], while the others implement special purpose data structures on disks.

Most of the proposed XML indexes are proved to be fast for their targeted queries, however, these are usually inefficient for their untargeted queries. For example, XR-tree [26], which is optimized for retrieving ancestor nodes that have specific tag names, cannot incorporate other efficient path labels such as *p-labels* [57], which is the fastest for suffix-path queries. That means in exchange for the performance of suffix path queries, XR-tree achieves fast ancestor query performance.

A natural question that follows is whether it is possible to achieve good performance for various types of query patterns using a standard B+-tree, which is a well-established disk-based data structure. Our answer to this question is affirmative. We show XPath queries can be performed with combinations of only two types of indexes; tree-structure and path-structure indexes. A challenging problem is that these index scans must be performed in a combined way, for example, we have to query ancestor nodes that belong to some suffix path.

Our approach to this problem is to integrate tree-structure and path-structure indexes into one multidimensional index. It accelerates query processing for complex combinations of structural properties. And also, it is possible to incorporate various types of XML indexes for tree-structures and path-structures.

As an integration approach, constructing multiple secondary B+-tree indexes does not help multidimensional query processing, since they work for only a single dimension; not the combinations of multiple dimensions. Moreover, the existence of multiple secondary indexes not only enlarges the database size, but also deteriorates the update performance.

Therefore, by using space-filling curve technique [31, 38, 42], we implemented the multidimensional index on top of a single B+-tree, which is beneficial in both of the query performance and database size.

1.8 Our Contributions

The outline and contributions of this thesis are as follows:

- We introduce the notion of *amoeba domain* to capture *structural fluctuations* of XML data, and it has an essential role to define *relation* in XML. (Chapter 2)
- We define functional dependencies for XML, and make correspondence between XML structures and FDs. (Chapter 3)
- We present the *amoeba-join decomposition* to efficiently retrieve XML structures satisfying functional dependencies. (Chapter 3)
- We introduce the notion of *ubiquitous keys* to denote equivalent classes of XML nodes, which is useful to aggregate information of a data object which is distributed throughout the XML document. (Chapter 3)
- We present three essential *amoeba join* processing algorithms. (Chapter 4)
- We introduce an efficient multidimensional index structure that can be implemented on top of the B+-tree. While some XML indexes in literature facilitate a few set of query patterns, our index is adaptive for various types of queries. Nevertheless, its index size remains compact. (Chapter 5)
- Experimental evaluations of the proposed methods. (Chapter 6)

In Chapter 7, we mention the related work and several open problems. We also present a more detailed discussion of managing scientific data and examples of SCMD data, then conclude this work.

Based on the above techniques, we have implemented an XML database system called Xerial , which will be publicly available at http://www.xerial.org/.

Several parts of this work has been published in journals and the proceedings of a workshop and conferences; Amoeba join was presented in [45]. Our XML indexing method is to be published in [46], and its related work including both of indexing and transaction management of XML were presented in [43, 44]. SCMD, which motivates this research, and also is one of the practical applications of this work, was mainly evaluated from the area of Biology [47, 48]. By using the data of SCMD, several unknown gene functions of yeast mutants were identified [36].

Chapter 2

Amoeba

The specification of XML itself imposes no rules for organizing XML data. Consequently, semantically identical XML documents may have different structures; we call this *structural fluctuation* in XML. Finding all the structural fluctuations in an XML document requires verbose path expression queries. To overcome this problem, in this chapter, we introduce the notion of *amoeba*, which gives a formal definition of structural fluctuations and can be used to represent an equivalent class of XML structures. We also present the *amoeba join*, which is an operation to retrieve a set of amoebas from an XML document.

2.1 Semantics of XML Data

Here, we demonstrate that XML structure provides surprisingly few semantics, clarifying the need to handle structural fluctuations. In XML, encapsulation of data with a tag is normally used to group data elements or text data. However, it inevitably leads to a structural hierarchy among the data elements, which may or may not express high and low ranks. The following XML example (**Figure 2.1**) represents organization data with both superficial and semantic hierarchy order between the managers David and Michael.

It is also possible to reverse the hierarchical order. In the following example (**Figure 2.2**), the belongs_to tag is used to switch the hierarchical positions of the managers David and Michael without losing the semantic relationship.

Furthermore, when a tag is used to group elements, there are generally no semantic

```
<org department="head office">
  <manager> David </manager>
  <org department="R&D">
    <manager> Michael </manager>
  </org>
</org>
```

Figure 2.1: Nested organization data

```
<org department="R&D">
   <manager> Michael </manager>
   <belongs_to>
     <org department="head office">
        <manager> David </manager>
     </org>
   </belongs_to>
</org>
```

Figure 2.2: A variation of the XML structure in **Figure 2.1**

ranks among the elements. For example, an manager element is allowed to have an org information as a child element, and vice versa. In both cases, we can write the same meaning data using different XML structures.

Therefore, hierarchical order does not directly represent the semantic relationship between data elements; semantics of data become clear only when they are explicitly given. We provide semantics of XML data as functional dependencies for XML, which is explained in Chapter 3. First, before introducing such semantics, in this chapter, we present a method to handle structural fluctuations when there is no given explicit semantics nor any schema.

2.2 XML Data Model

We use a tree model to represent XML data. XML documents comprise of three essential types of nodes: element nodes (E), attribute nodes (A), and text nodes (T). Element and attribute nodes have a name, and to distinguish element and attribute nodes, attribute names are prefixed with "@". Attribute nodes must be a child node of an element node. Attribute and text nodes must have a carry text. Element nodes may have child nodes, but attribute and text nodes are terminal. **Figure 2.4** illustrates the tree model of XML, in which edges to text nodes are omitted, and each node except text nodes is assigned an ID.

2.3 Amoeba

In the rest of the thesis, we have to handle variously structured fragments of an XML document. To represent these structural fluctuations, we introduce a useful notion of the *amoeba*, which is a relaxed definition of trees in the graph theory:

Definition 2.3.1 (Amoeba) *Given a set $f = \{d_1, \ldots, d_k\}$ of XML nodes, where $d_i \in E \cup A \cup T$, we say f is an amoeba if one of $d_1, \ldots, d_k$ is a common ancestor of the others, denoted by $\langle\langle d_1, \ldots, d_k \rangle\rangle$.*

The notion of amoeba gives a formal definition of structrual fluctuations:

Definition 2.3.2 (Structrural Fluctuation) *Structrual fluctuations of nodes $d_1, \ldots, d_k$, is a set of all possible amoebas, satisfying a predicate $\langle\langle d_1, \ldots, d_k \rangle\rangle$.*

What is important in the definition of amoeba is, whatever the structure of a node set is in the XML document, the node set can be considered as an amoeba as long as it has a root node within itself. For example, every structure in **Figure 2.3** is an amoeba. The root node of an amoeba is usually an element node because attribute and text nodes are terminal, but a node set which is a singleton, e.g. $f = \{d_1\}$, also can construct an amoeba. A benefit of this notion of the amoeba is that every structural fluctuation in the **Figure 2.3** can be described with a single expression; ⟪order, book, customer⟫.

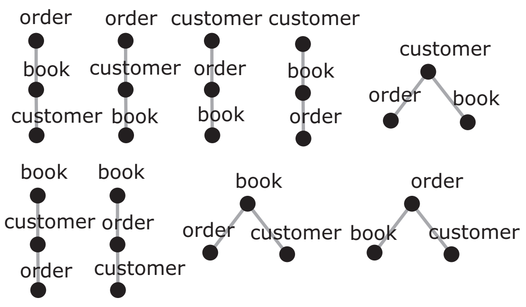


Figure 2.3: The amoeba of order, book and customer nodes covers all of the above structural fluctuations.

2.4 Amoeba Domain

An amoeba is an instance in an XML document. To describe a set of amoebas in the document, we need a pattern expression such as XPath [15]. In literature, an XPath expression is typically modeled as a pattern tree [24]. However, in the presence of structural fluctuations, it is too restrictive to demand that data structures must obey a single tree-pattern. Furthermore, concatenating all possible path structures into a single XPath expression can be tedious. Therefore, we use a short-hand notation of amoeba, e.g. $\langle\!\langle d_1, d_2, d_3 \rangle\!\rangle$, to describe XML data whose structure is unknown except it constructs an amoeba consisting of node d_1, d_2, and d_3.

To represent both of rigorous and fluctuated tree structures easily, we introduce the notion of an *amoeba domain*, which can express various patterns of path structures, including twigs and amoebas:

Definition 2.4.1 (Amoeba Domain) *An amoeba domain D is a pair $D = (V, P)$, where V is a set of XML nodes belonging to one of the node types in $\{E, A, T\}$, thus if $v \in V$ is an element or attribute node, v has a name. P is a boolean combination of predicates applied to nodes in V. P must contain at least a condition that V constructs an amoeba. Other allowable predicates in P are as follows:*

- *For two nodes u, v in V, u is an ancestor (or parent) of v.*
- *Value predicates $=, \neq, >$, etc. for a text node $t \in V$; e.g. $t = 100$.*
- *A condition to specify a subset of V, say $\{a, b, c\}$, constructs an amoeba, denoted $\langle\!\langle a, b, c \rangle\!\rangle$.*

Although there should be a discussion about the choice of predicates, we limit the number of predicates in the amoeba domain for illustrative purpose.

In the definition of the amoeba domain, a node set V must construct an amoeba, but it does not specify which node will be the common ancestor in order to allow flexibility of the data structure. On the other hand, an amoeba domain can represent a fixed tree structure, which is usual for XPath expressions. For example, by using an XPath expression, we simply denote an amoeba domain as D_1 = //book/title to specify a tree structure which consists of book and title nodes, where title nodes must be a child of a book node in the XML document.

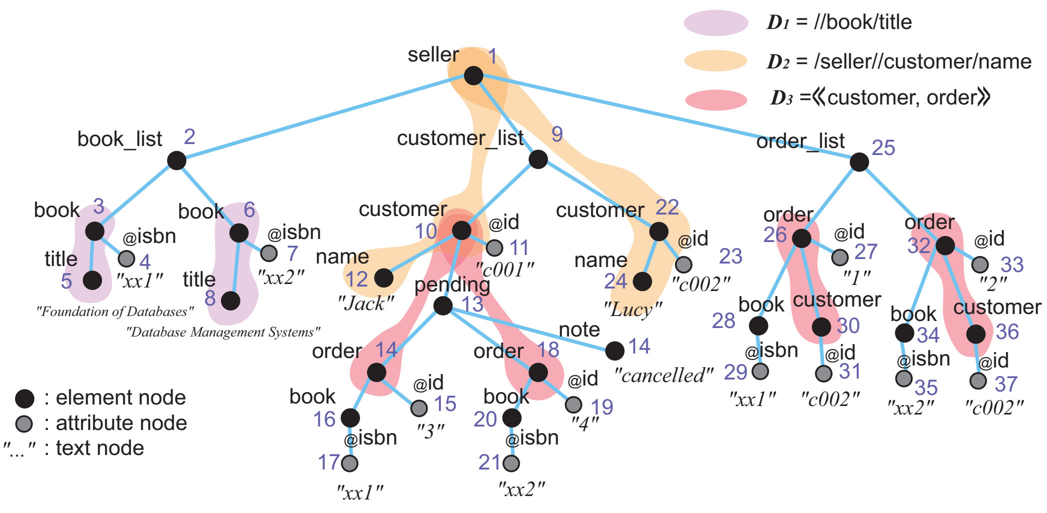


Figure 2.4: Amoeba domains

Given an amoeba domain D, its *instances*, denoted $[\![D]\!]$, is a set of amoebas in an XML document matching a structure pattern specified by D. **Figure 2.4** shows some examples of amoeba domains and their instances. For an amoeba domain D_1 = //book/title, we denote its instances as $[\![D_1]\!]$ or $[\![$//book/title$]\!]$. The amoeba domain of D_2 = /seller//customer/name involves three nodes; seller, customer and name, in which each name node must be a child of a customer node, and customer and name nodes must be an descendant of the seller node, which is the root of the XML document. To select the amoeba of customer and order nodes, we use a notation D_3 =$[\![$《customer, order》$]\!]$, which means a set of amoebas that one of the customer and order node is a common ancestor of the other in an XML document. Another extreme example is an XML document, which can be represented as $[\![$//*$]\!]$, which contains an amoeba consisting of every node in the document.

2.4.1 Node Labels

It is essential to have a capability to specify some nodes in an amoeba domain. In relational databases, a table has columns and each column has a name. Hence, the relational database users can perform algebraic operations by specifying data columns by name. As its counterpart in XML, we can use node names in the amoeba domain. For example, in an amoeba domain D_3 =《customer, order》, we can use node names customer and order to specify nodes. To avoid ambiguity of node names between several amoeba domains, we use a dot notation: for example, when D_4 = //order/@id, D_5 = //customer/@id, we can distinguish these two @id nodes as D_4.@id and D_5.@id, or we simply denote these node labels as order@id and customer@id. We denote a text node corresponding a node label n as $[n]$. For example, [order@id], [title] specify a text node of order@id, title, respectively.

In this thesis, we consider that the inputs and outputs of an XML query are *amoeba domains* $D_1, \ldots, D_k$, and a query is evaluated using *instances* of each input amoeba domain, $[\![D_1]\!]$, ..., $[\![D_k]\!]$. Then, the query produces instances of another amoeba domain. XML queries often involve intermediate results, which are themselves instances of amoeba domains. Constructing each intermediate amoeba domains and their node names can be a daunting task. Therefore, for readability, we assume node names are *inherited* to the intermediate amoeba domains.

For example, if we perform a filtering operation on an amoeba domain D = //book/@isbn and generate the another amoeba domain D' = //book[@isbn="xx1"], we can use the node name book and book@isbn in both of the amoeba domains D and D'.

2.4.2 Projection

To retrieve a specific set of XML nodes from instances of an amoeba domain, we define the *projection* of an amoeba domain, denoted $\pi_{NL}(D)$, where NL is a list of node labels and D is an amoeba domain. Since the projected nodes may not comprise an amoeba when $|NL| > 1$, we add the root node of the XML document to each projection result to ensure that the output is also an amoeba domain. For example, a projection $\pi_{\text{order,book}}(D)$ returns amoebas consisting of three nodes; order, book, and the root node; even if order and book have no common ancestor, the root node is on their behalf. Note that, however, in the following descriptions, we omit the root node from instances of a projection for brevity. For example, given a list NL of node labels $L_1, \ldots, L_m$ and an amoeba domain D, each instance t of a projection $\pi_{NL}(D)$ is denoted $t = (t_1, \ldots, t_m)$, where each t_i corresponds the node label L_i, and t does not contain the root node unless it is contained in NL.

2.5 Amoeba Join

The requirement of path structures within query statements is a serious obstacle to using XML DBMSs. *Amoeba join* is a method for overcoming this problem by allowing structural fluctuations in the underlying XML database, and retrieving data matching the query of interest. The amoeba join requires no explicit path structures; node names and keywords are sufficient to perform searches.

Even when using a schema or DataGuides [18], which is a summary of path structures of XML, learning the entire structure is more difficult than creating a list of all types of tag and attribute names. We investigated a benchmark XML document provided by XMark [49] (scalability=1.0, 114 MB). The document contained 83 tags and attribute names and 548 distinct paths. As a consequence, database users should have much more information on tags and attributes than that of path structures, which may differ depending on context. This is why query processing without explicit path structures, which is achieved by amoeba join, is promising.

Given amoeba domains, for example, D_1 = //order, D_2 = //book and D_3 = //customer, an amoeba join constructs their amoebas, allowing structural fluctuations. A similar operation is a *structural join* [3], which concatenates two nodes p, q if p is an ancestor of q. The structural join is generally used to process descendant-axis (//) queries. However, to handle structural fluctuations, we also have to consider the following cases; p is an ancestor *or* descendant of q. In addition, there are indirect structural relationships involving more than two nodes, for example, node p and q are connected through another node r.

To define the amoeba join, we introduce some notations. Let NL be a list of node labels, and I be a list of input amoeba domains $D_1, \ldots, D_k$. We say *I covers NL* if for each amoeba domain D_i contains at least one node label in NL, and also every node label in NL is contained in some amoeba domain in I. The definition of the amoeba join is as follows:

Definition 2.5.1 (Amoeba Join) *Given a list NL of node labels $L_1 \ldots L_m$, and a list I of input amoeba domains $D_1, \ldots, D_k$ that covers NL, where each $D_i = (V_i, P_i)$, an amoeba join, denoted $AJ_{NL}(I)$, produces another amoeba domain $D' = (V', P')$ such that V' is a union of $V_1, \ldots, V_k$, and P' must hold all of the predicates $P_1, \ldots, P_k$, and also P' must*

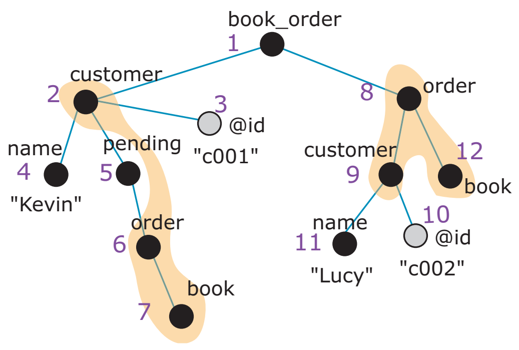


Figure 2.5: Amoeba join of book, customer, order nodes.

have a constraint that a node set specified by $L_1, \ldots, L_m$ *constructs an amoeba, that is*

$$V' = \bigcup_{1 \le i \le k} V_i, \qquad P' = \langle\!\langle L_1, \ldots, L_m \rangle\!\rangle \wedge \bigwedge_{1 \le i \le k} P_i$$

.

For example, $AJ_{\text{order,book,customer}}(D_1, D_2, D_3)$ contains an additional predicate ⟪order, book, customer⟫, and generates all instances of amoebas in the XML document, matching some structures in **Figure 2.3**. Amoeba join processing algorithms are described in Chapter 4.

The definition of $D' = AJ_{NL}(D_1, \ldots, D_k)$ says that the *partial* structure specified by NL is an amoeba. Nevertheless, every instance of D' becomes an amoeba. Let p, q be input amoebas of $AJ_{NL}(D_1, D_2)$, where $p \in [\![D_1]\!]$, $q \in [\![D_2]\!]$. When $p \cup q \in [\![AJ_{NL}(D_1, D_2)]\!]$, two amoebas p, q must have an overlap, since their partial structure constructs an amoeba. Since XML is not a DAG, the root node of this overlapped part (amoeba) cannot have two distinct parent nodes, eventually one of the amoeba roots of p, q must be an ancestor of the other amoeba, and thus $p \cup q$ is also an amoeba. When more than two amoebas are involved, we can apply the above discussion by seeing two of the input amoebas; the one containing the root node of the partial amoeba, and the other. Therefore, the result of $AJ_{NL}(D_1, \ldots, D_k)$ is also an amoeba domain.

When a node tuple $t = (t_1, \ldots, t_m)$ is an instance of $\pi_{NL}(I)$, where I is a list of input node domains, and NL is a list of node labels to join, we call t is an amoeba (or an amoeba tuple) if t has a common ancestor of the other nodes in t, and we call its common ancestor

$t_r (1 \leq r \leq m)$ the *amoeba root* of t. **Figure 2.5** is an example of amoeba join. An amoeba join $AJ_{\text{order, book, customer}}$(//order, customer, book) gives a set of amoebas containing node tuples $\{(6, \mathbf{2}, 7), (\mathbf{8}, 9, 12)\}$ (bold numbers represent amoeba roots). The amoeba join operation is adaptive to various XML structures; It can capture the order node (6) even if it is under the pending tag (5), and also it detects hierarchical change of order and customer nodes.

To locate such complicated XML structures, the least common ancestors (*lca*) are frequently used [34, 56]. Note that, however, amoeba join is not a process of computing *lca* of a given node set. The *lca* nodes of org, manager and location nodes in **Figure 2.5** include the root node, book_order (id = 1). Every node in XML can reach the others through the root. For example, node 2 and 12, which are apparently irrelevant, can be connected via the root. Therefore, the lca method is not appropriate for finding relationships between nodes. Amoeba join is similar to the *lca* method in that it finds a common ancestor; however, it limits common ancestor nodes to those belonging to one of the nodes specified in NL. By using this rule, the relationships among nodes are bound to a common ancestor, i.e., the amoeba root. The amoebas in **Figure 2.5** are bound by 2 or 8. If there is no such bound, as in the *lca* method, the relationships among the connected nodes are very weak.

When the root node of XML happens to be contained in one of the domains, any node tuple becomes an amoeba. In general, such a query is no use. If the root node is required in order to specify the context of queries, amoeba join with a context domain D_c, denoted $AJC_{NL}(D_c, D_1, \ldots, D_k)$, is preferable. It restricts the search region of the query under the specified context nodes.

2.5.1 Amoeba Join Syntax

Here, we introduce syntax of amoeba join expressions:

S := "`AJ(`" E (, E)* "`)`"

E := F | `$variable` := F | S

F := *XPath-expr* (P <`value`>)? | <`value`>

P := $\Rightarrow$ | = | $\neq$ | < | > | $\leq$ | $\geq$

To make expressive queries for extracting valuable information, we extended amoeba join to incorporate XPath expressions. For example, an amoeba join expression AJ(order, customer) collects all pairs (amoebas) of order and customer elements that have ancestor-descendant relationships. As along as each input domain of an amoeba join expression is given by an XPath expression, there is no need to specify a list NL of node labels to join, because we can use the return node of an XPath expression, e.g. title node in an XPath expression //book/titile, as a join target. In the next chapter, we explain situations when NL becomes different from the return nodes of the input amoeba domains, but such NL can be determined automatically. In order to specify NL explicitly, we can use variable references. For example, AJ($x = customer, $x/name, order) joins customer nodes, its child name nodes and order nodes. This expression is identical to an amoeba join operation $AJ_{\text{customer, name, order}}$(//customer/name, order), in which the amoeba domain D =//customer/name substitutes the variable references.

Amoeba join can be used with explicit path queries. AJ(/book_order, "Kevin") computes amoebas that have book_order nodes directly under the root node, and text nodes with a value of Kevin. To express a node x such that a text y occurs in the subtree rooted at x, we provide the statement $x \Rightarrow y$. For example, the expression customer $\Rightarrow$ "Kevin" designates the customer tag containing the text node "Kevin", whether it is a child or descendant node of the customer node.

Chapter 3

Purifying XML Structures

The notion of functional dependency has an essential role in data modeling. The existing proposals of functional dependencies for XML are based on fixed paths, imposing severe structural constraints on XML data. Our definition of FDs for XML are based on the notion of amoeba so as to allow various XML structures. In this section, we explain functional dependency can be a strong tool to represent XML structures, while allowing flexibility of data modeling.

3.1 Functional Dependency

In relational databases, a functional dependency (FD) can be defined on attributes (relational terminology for column names) in a relation. In tree-structured XML, however, there is no flat relation such as a table. Therefore, we first have to define the *relation* in XML. In the previous work of FDs [5, 10, 11, 32], it is considered that XML has no counterpart of relations in relational databases, so several new definitions of FDs tailored to XML, which are based on paths, have been developed. In contrast to these approaches, our method utilizes the notion of the amoeba to define a relation in XML, which usually has a zig-zag shape. A relational database has a flat schema to specify relations, while in XML, the amoeba structure of XML data forms relations.

We describe the functional dependency for XML with node labels in amoeba domains. Let X be a list of node labels, and Y be a single node label, a functional dependency for

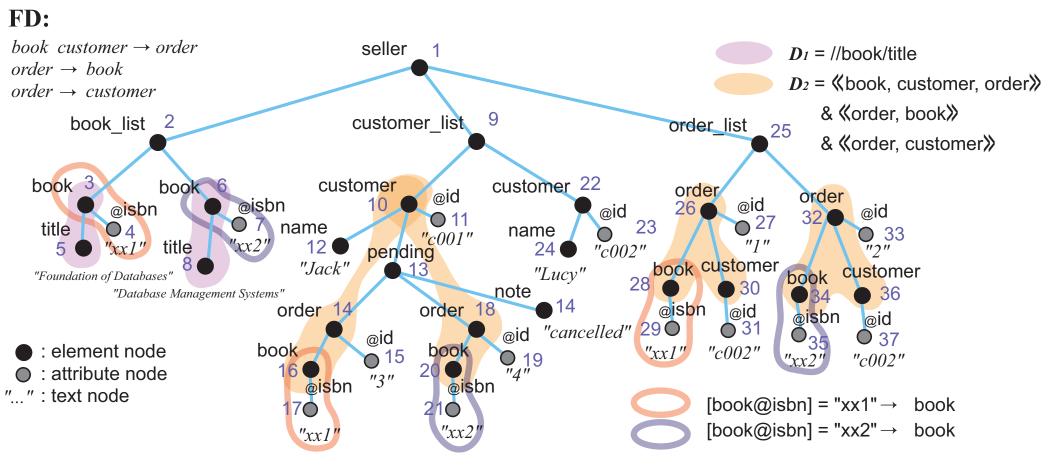


Figure 3.1: Functional dependencies define XML structures.

XML is expressed as $X \rightarrow Y$. Before defining FDs for XML, we show some notations: let F be a set of FDs, then $NL(F)$ is the set of node labels appearing in F. Given a list I of amoeba domains $D_1, \ldots, D_k$, if each D_i contains at least one node label in $NL(F)$, and all node labels in $NL(F)$ are contained in I, we say *I covers $NL(F)$*. Now, we introduce *relation* in XML, which is a special case of an amoeba domain:

Definition 3.1.1 (Relation in XML) *Let F be a set of FDs, and $D_1, \ldots, D_k$, where each $D_i = (V_i, P_i)$, be a list of amoeba domains that covers $NL(F)$. We say an amoeba domain $R(F) = (V, P)$ is a relation in XML if $V = \bigcup_{1 \leq i \leq k} V_i$, and $P = (\bigwedge_{1 \leq i \leq k} P_i) \wedge \langle\langle X_1, Y_1 \rangle\rangle \wedge \cdots \wedge \langle\langle X_m, Y_m \rangle\rangle$, where each $\langle\langle X_j, Y_j \rangle\rangle$ corresponds to each FD $X_j \rightarrow Y_j$ in F; nodes in X_j, Y_j must constitute an amoeba.*

When F contains an FD, $AB \rightarrow C$, an instance of a relation $R(F)$ must satisfy a structural constraint $\langle\langle A, B, C \rangle\rangle$; node A, B and C construct an amoeba. For example, when F = {book, customer → order, order → book, order → customer}, then a relation $R(F) = (V, P)$ has nodes book, customer, and order in the node set V, and its predicate P demands that 《book, customer, order》, 《order, book》 and 《order, customer》 must be amoebas. In **Figure 3.1**, an amoeba domain D_2 satisfies these predicates, thus we say D_2 is a relation with respect F.

We are now ready to define FD for XML:

Definition 3.1.2 (FD for XML) *We say a relation $R(F)$ satisfies an FD $X \to Y \in F$ if for each pair of amoebas $p, q \in [\![R(F)]\!]$, $p.X = q.X$ implies $p.Y = q.Y$, where $p.X$ is a list of nodes in p corresponding each node label in X.*

The equality of two nodes n_1, n_2 is defined as

$$n_1 = n_2 \text{ iff } id(n_1) = id(n_2),$$

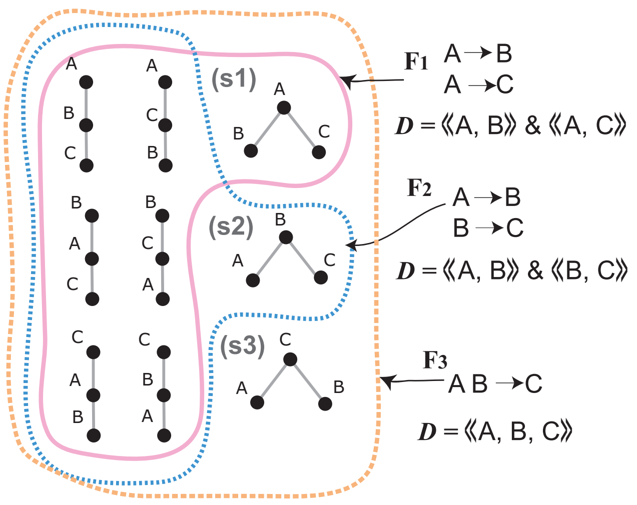
where the function $id(n)$ returns a node ID assigned to node n when $n \in E \cup A$, or a text value of n when $n \in T$.

Intuitively, an FD $X \to Y$ specifies that a node set belonging to X uniquely determines a node belonging to Y. For example, in **Figure 3.1**, an FD order $\to$ book is defined, that is for each order node, a single book node must be uniquely located; book nodes 16, 20, 28, 34 belong to order nodes 14, 18, 26, 32, respectively.

A relational database has a schema, so FDs can be defined using instance values of attributes in a relation. In XML, however, in addition to the value-based FDs, we also need *node-based* FDs to define a relation in XML. For example, with two FDs order $\to$ book and order $\to$ customer, we can identify both of book and customer nodes from an order node. It works as if we have an order table, the schema of which is (order, book, customer).

Although we use both of text values and node IDs to define FD, its fundamental property, that is, FD is defined using a relation R and its instances, is never changed, since our definition also fixes a relation $R(F)$ and uses its instances in $[\![R(F)]\!]$. The extension of the equality to the node IDs and text values has no side effect to the definition of FDs. Consequently, with the notion of the relation in XML, which is described with an amoeba domain, we can apply important results in the dependency theories even in XML, such as inference rules of Armstrong's axioms [2, 39, 52].

By using Armstrong's axioms, we can deduce a closure F^+ of FDs implied from F. Let F and G be sets of FDs, F and G are equivalent if $F^+ = G^+$ [52]. It has been well understood that this notion of equivalence is useful in defining a minimal set of FDs, especially in that we can use a set of FDs such that each FD contains only a single node in the right side.



```
<student id="1">
 <semester year="2006">
   <class id="CS301"/>
   <grade>A</grade>
 </semester>
</student>

<class id="CS301"/>
  <report>
   <student id="1"/>
   <grade>A</grade>
  </report>
</class>

<grade>
  <student id="1"/>
  <class id="CS301"/>
  <note>excellent!</note>
</grade>
```

Figure 3.2: Allowable structural fluctuations vary according to a set F of FDs, and useful examples of XML structures for F_3, when A = student, B = class, and C = grade.

3.2 Examples of Functional Dependencies

To make correspondence between FDs and XML structures, we consider XML nodes conforming to an amoeba are semantically related. If there are partial dependencies within an amoeba, XML structures must represent these relationships, so allowable structural fluctuations vary according to a set F of FDs. **Figure 3.2** illustrates variations of XML structures for several sets of FDs. An FD for two nodes is easy to represent, since it corresponds to ancestor-descendant relationships in XML (F_1 and F_2). An FD involving more than two nodes, e.g. A B → C, uses somewhat tricky structures 《A, B, C》, but these can be used effectively. For example, XML data in **Figure 3.2** illustrate the benefits of using various data structures for grouping and enhancing the information (see how semester, report and note elements are used).

Here, we show some examples of FDs using the XML data in **Figure 3.1**:

- book → book@isbn : Each book node must have an @isbn attribute node.
- book@isbn → book : The reverse of the above FD. In XML, every attribute must belongs to a single element, so this type of FD always holds for all attribute nodes.
- book, customer → order : Any order node is identified by a pair of book and customer

node. In other words, either of the book or customer node is not sufficient to locate an order node.

- [order@id] $\rightarrow$ order : Given an order@id value (text node), we can uniquely determine the order node. In this case, an order@id value is a key (global ID) of an order node, and no duplicate value of order@id is allowed in the XML document.

- book_list, [book@isbn] $\rightarrow$ book : With the information of a book_list node and @isbn value, a book node can be determined. For example, the book_list node 2 and a value "xx1" locate the book node 3. With this FD, no book node is allowed to have the same book@isbn value in the context of the book_list node. This example can be considered as a *relative key* [11], which localizes the key definition under the specified path, since uniqueness of [book@isbn] values is also localized in the context of the book_list node, but various data structures are allowed compared to the relative keys proposed in [11].

3.3 Resolving FDs

To represent that the result of an amoeba join satisfies structural constraints specified by F, we introduce another form of the amoeba join: $AJ_F(I)$, where I is a list of input amoeba domains:

Definition 3.3.1 (Amoeba Join Resolving FDs) *Given a set F of FDs, and a list I of input amoeba domains that covers $NL(F)$, where $NL(F)$ is a list of node labels appearing in F, we say an amoeba join of the form $AJ_F(I)$ resolves F, if its output amoeba domain is $R(F)$, a relation that conforms to F.*

The result of an amoeba join $AJ_F(I)$ must satisfy predicates in $R(F)$. In **Figure 3.1**, for example, the amoeba domain D_2 is a result of the AJ_F(order, book, customer).

3.4 Amoeba Join Decomposition

The computation of an amoeba join $AJ_F(I)$ usually involves many XML structures violating the given functional dependencies. For example, amoebas of 《book, customer, order》 in **Figure 3.1** contain some irrelevant structures such as $(16, 10, 18)$, which does not satisfy an FD order $\rightarrow$ book. If a customer node has many more order or book nodes, the number of irrelevant structures to be produced is likely to increase. Therefore, its computational cost could be quite huge.

Our solution to this problem is to *decompose* an amoeba join operation into a nested form of several amoeba joins so that temporary results could be minimized by the use of functional dependencies. This method enables us to selectively retrieve XML structures that satisfy each functional dependency, and avoids extracting unwanted XML structures:

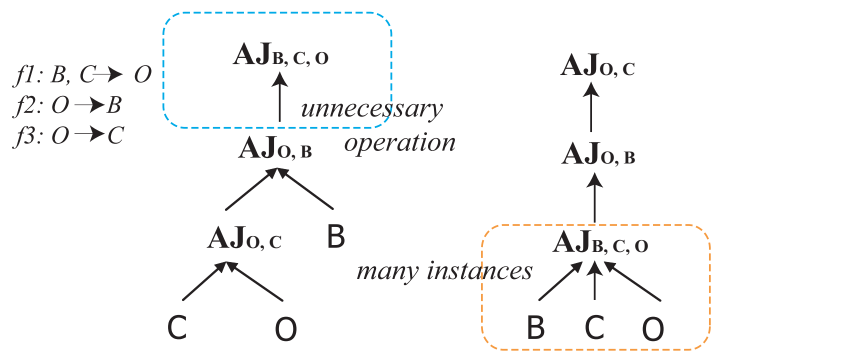
Theorem 3.4.1 (Amoeba Join Decomposition) *Let F be a set of FDs, and I be a list of input amoeba domains which covers $NL(F)$. Let $a : X \rightarrow Y$ be an FD in F, we can decompose an amoeba join operation as follows:*

$$AJ_F(I) = AJ_{X,Y}(AJ_{F-\{a\}}(I - I'), I')$$

where I' is a subset of I, and each amoeba domain in I' has no corresponding node labels in $NL(F - \{a\})$, so I' might be an empty list. When F or I is empty, $AJ_F(I)$ is an empty amoeba domain.

proof by induction. Let amoeba join expressions P, Q be $AJ_F(I)$, $AJ_{X,Y}(AJ_{F-\{a\}}(I - I'), I')$, respectively. When $|F| = 2$, an instance $p \in P$ is an amoeba, satisfying structural constraints implied by the FDs in $F - \{a\}$ and $a : X \rightarrow Y \in F$, consequently $P \subset Q$. Let q be an instance of Q. $F - \{a\}$ consists of an FD, say $A \rightarrow B$. Thus, the result of Q satisfies $\langle\langle X, Y\rangle\rangle$ and $\langle\langle A, B\rangle\rangle$, that is $q \in R(F)$; $Q \subset P$.

When $|F| > 2$, because of the hypothesis of the induction, $AJ_{F-\{a\}}(I - I')$ can be totally decomposed into the form $AJ_{X_1,Y_1}(AJ_{X_2,Y_2}(\ldots(AJ_{X_n,Y_n}(\ldots))))$, where $X_i \rightarrow Y_i \in F - \{a\}$. Thus, every instance of Q satisfies predicates $\langle\langle X_1, Y_1\rangle\rangle$, $\ldots\langle\langle X_n, Y_n\rangle\rangle$, in addition, $\langle\langle X, Y\rangle\rangle$ holds because of the out-most amoeba join operation. Therefore, the number of predicates is equal between P and Q, consequently $P = Q$. $\square$



$AJ_F(B, C, O)$
$= AJ_{B,C,O}(AJ_{F1}(B, C, O))$
$= AJ_{B,C,O}(AJ_{O,B}(AJ_{F2}(C, O), B))$
$= AJ_{B,C,O}(AJ_{O,B}(AJ_{O,C}(C, O), B))$

$AJ_F(B, C, O)$
$= AJ_{O,C}(AJ_{F1}(B, C, O))$
$= AJ_{O,C}(AJ_{O,B}(AJ_{F2}(B, C, O)))$
$= AJ_{O,C}(AJ_{O,B}(AJ_{B,C,O}(B, C, O)))$

Figure 3.3: Decomposition order affects the generated amoeba join schedules.

The decomposition in Theorem 3.4.1 can be used repeatedly to derive a series of amoeba joins to calculate the original amoeba join. **Figure 3.3** illustrates query schedules generated by amoeba join decompositions. For simplicity, we use B, C and O to denote both of node labels and amoeba domains of book, customer and order, respectively. When we choose the FD $B, C \rightarrow O$ as a first decomposition target, the query schedule becomes as left-hand schedule in **Figure 3.3**. This schedule reduces the search space of possible amoebas effectively by evaluating functional dependencies f3 and f2 in earlier steps, making it unnecessary to consider f1. On the other hand, the right-hand side schedule uses the other FD first, which ends up enumerating all possible structural fluctuations, and afterwards makes selections using FDs.

Functional dependencies are frequently observed between parents and their children, e.g., order@id → order; the order node must be the parent of the order@id node. In this case, we can explicitly decompose the amoeba join using parent-child join:

Corollary 3.4.2 *For an FD $a : P \rightarrow C$ or $C \rightarrow P$ in F, where P is the parent node of C, the amoeba join decomposition can be expressed as follows:*

$$AJ_F(I) = PC_{P,C}(AJ_{F-\{a\}}(I - I'), I'),$$

where $PC_{P,C}$ denotes the parent-child join, which is a specialized version of an amoeba *join that connects parent nodes P and child nodes C.*

3.5 FD Based XML Query Processing

When FDs are defined for an XML document, even if there are no explicit path structures in the query statements, a relation of XML specifies XML structures of interest. Unlike relational databases that have fixed tables, a relation in XML varies according to query context. In **Figure 3.1**, for example, if we want to query a pair of book and customer nodes, order nodes are also required to detect book and customer nodes connected through order nodes (illustrated as D_2). On the other hand, just to retrieve only book and title nodes (D_1), we cannot use FDs related to customer or order, since book nodes $3, 6$, which are not contained in D_2, should be included in the result. Let NL be a list of node labels contained in a query. We have to select FDs that are related to nodes in NL. In other words, NL defines a set of FDs that conforms to a relation.

All relevant nodes to query nodes in NL can be determined by calculating a closure NL^+ of node labels [52, 39]. The computation of NL^+ can be performed as follows: if there is an FD such that all node labels in its left side are contained in NL, add a node label in the right side of the FD to NL, then repeat this process until there is no change in NL. We call the set of FDs used in this process as *context FD*, denoted F_{NL}.

For example, suppose $F = \{\text{book customer} \rightarrow \text{order}, \text{order} \rightarrow \text{book}, \text{order} \rightarrow \text{customer}\}$. When $NL = \{\text{book, customer}\}$, $NL^+ = \{\text{book, customer, order}\}$, and F_{NL} is identical to F because all FDs are used. When $NL = \{\text{book, title}\}$, $NL^+ = NL$ and F_{NL} is empty as no FDs are involved. This means no structural constraint is found between book and title. If we add title $\rightarrow$ book to F, then F_{NL} becomes $\{\text{title} \rightarrow \text{book}\}$.

A relation $R(F_{NL})$ is a result amoeba domain of a query for nodes in NL. In other words, XML structures for querying nodes in NL are defined by $R(F_{NL})$. When no FD is defined for some node label $n \in NL$, we just perform amoeba joins $AJ_{NL}(AJ_{F_{NL}}(I), n)$ so as to retrieve amoebas containing n; other nodes in $NL - n$ satisfies F_{NL}. For example, $AJ_F(\text{book, order, title})$ is evaluated with a nested amoeba join:

$$AJ_{\text{book, order, title}}(AJ_F(\text{book, order}), \text{title}),$$

since there is no FD for title nodes. While an amoeba join AJ_F(book, customer) requires order nodes, but order is not specified in the query, so we have to eliminate order nodes using a projection after the amoeba join processing:

$$AJ_F(\text{book, customer}) = \pi_{\text{book, customer}}(AJ_F(\text{book, customer, order})).$$

3.6 Ubiquitous Keys

Tree structures are not sufficient to represent data models of the real world, which tend to have a graph structure [25]. To accommodate XML's single-tree structure to the graph-structured data model, we have to manage multiple hierarchies of XML data within a single XML document. **Figure 3.1** is an example of multiple hierarchies; book_list, customer_list and order_list.

In addition, to make connections between these hierarchies, we need keys for XML [11] to relate nodes in an XML document. A key is a special case of an FD, and it can be used to uniquely locate XML nodes. However, to manage mixtures of variously structured data, rather than using one-to-one connection of nodes with keys, which oblige the users to specify both sides of connected nodes explicitly, it is more convenient to make equivalent classes of XML nodes to automatically connect related nodes. In this section, we introduce the *ubiquitous keys*, which define equivalent classes of XML nodes.

We denote a *ubiquitous key* φ for a node label Y as $X \rightarrow Y$, which has the same syntax with FDs, but is different in that it imposes no structural constraint on XML data, and if two amoebas p, q agree on X, that is $p.X = q.X$, then $p \equiv_\varphi q$, that is p and q belong to an equivalent class specified by φ. For example, φ_1 = [book@isbn] $\rightarrow$ book is a ubiquitous key specifying that book nodes with the same @isbn value are equivalent.

A ubiquitous key φ_2: book, customer $\rightarrow$ order is another example. However, the right-hand side of φ_2 uses node IDs of book and customer nodes, which are usually managed in the internals of a DBMS, so the user cannot specify the equivalent class of order nodes within a query statement. In this case, we need ubiquitous keys for both of book and customer nodes. For example, ubiquitous keys [book@isbn] $\rightarrow$ book and [customer@id] $\rightarrow$ customer are required to provide text values for each of book and order nodes in φ_2.

We must define ubiquitous keys so that we can resolve all non-text nodes in the left-hand side with text nodes by recursively traversing a set of ubiquitous keys from the right-hand side to the left-hand side. We use $\rho(\varphi)$ to denote the set of all node labels found in the above traversals from φ, and call $\rho(\varphi)$ *key domains*. For example, given ubiquitous keys:

$$\begin{aligned}\varphi_1 &: \text{[book@isbn]} \rightarrow \text{book}\\ \varphi_2 &: \text{book, customer} \rightarrow \text{order}\\ \varphi_3 &: \text{[customer@id]} \rightarrow \text{customer},\end{aligned}$$

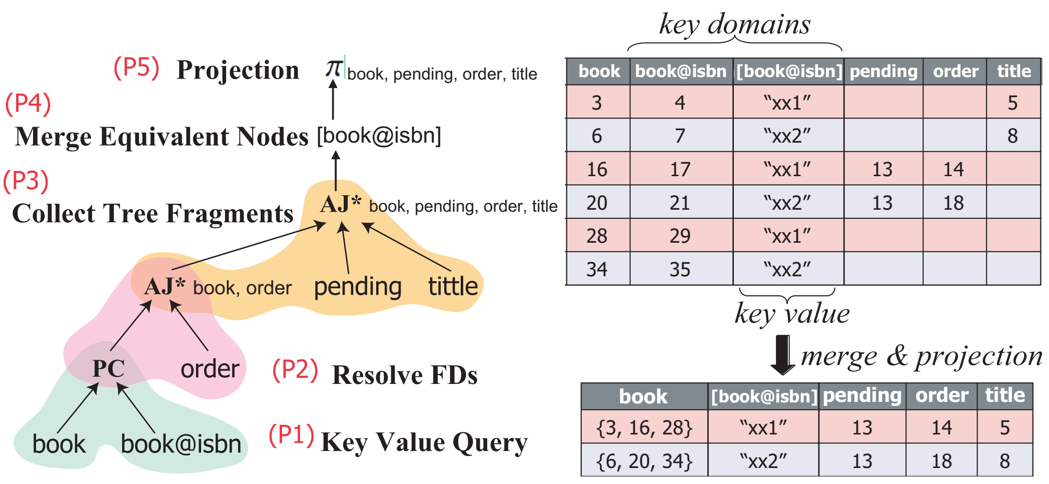
their key domains are as follows: $\rho(\varphi_1) = \{$[book@isbn]$\}$, $\rho(\varphi_2) = \{$book, customer, [book@isbn], [customer@id]$\}$, and $\rho(\varphi_3) = \{$[customer@id]$\}$. Equivalence of XML nodes u, v specified by a ubiquitous key φ, denoted $u \equiv_\varphi v$, can be determined by using text nodes contained in $\rho(\varphi)$:

Definition 3.6.1 (Equivalent Class of XML Nodes) *Let φ be a ubiquitous key, and $\rho(\varphi)$ be its key domains. Let F be a set of FDs, and R be a relation $R(F_{\rho(\varphi)})$, where $F_{\rho(\varphi)}$ is a set of context FDs for node labels in $\rho(\varphi)$. Given two nodes u, v contained in amoebas p, q in $[\![R]\!]$, respectively, $u \equiv_\varphi v$ iff for each text node label l in $\rho(\varphi)$, $p.l = q.l$.*

Let $NL = \rho(\varphi)$, equivalent XML nodes can be retrieved with a query $AJ_{F_{NL}}(I)$. For example, instances of $R(F_{\rho(\varphi_1)})$ are illustrated with red and blue circles in **Figure 3.1**, and book nodes enclosed with the same colored one are equivalent. On the other hand, there is no equivalent order nodes for φ_2, instances of which are in D_2, since there is no instances $p, q \in [\![D_2]\!]$ such that a pair of [book@isbn] and [customer@id] values are the same.

As for the value equality of text nodes, we simply compare text values as strings, and it is sufficient for the purpose of this paper. For various options of value equality of XML nodes, see the discussion in [11].

An amoeba join operation is capable of querying structural fluctuations. For example, however, there is no matching amoeba ⟪book, pending, order, title⟫ in the XML data of **Figure 3.1**, failing to retrieve meaningful information. Our approach to this problem is to collect fragments of amoebas allowing *null* values, for example (book, *null*, *null*, title) and (book, pending, order, *null*). Then, merge these tree-frame fragments based on the equivalence of book elements specified by ubiquitous keys. We use AJ^* to denote such an amoeba join operation that allows *null* values:



book	book@isbn	[book@isbn]	pending	order	title
3	4	"xx1"			5
6	7	"xx2"			8
16	17	"xx1"	13	14	
20	21	"xx2"	13	18	
28	29	"xx1"			
34	35	"xx2"			

book	[book@isbn]	pending	order	title
{3, 16, 28}	"xx1"	13	14	5
{6, 20, 34}	"xx2"	13	18	8

Figure 3.4: Amoeba Join Schedule of AJ^*_F(book, [pending, order, title]). AJ^* computes fragments of amoebas, then these fragments are merged to fill blank columns.

Definition 3.6.2 (AJ^*) *Let NL be a list of node labels, and I be a list of input amoeba domains covering NL, an amoeba join $AJ^*_{NL}(I)$ outputs the same amoeba domain with $AJ_{NL}(I)$, except that each result amoeba is allowed to have null nodes other than the node corresponding a first node label in NL.*

To facilitate ubiquitous keys, instead of AJ, we use AJ^* to decompose an amoeba join operation. **Figure 3.4** illustrates a schedule of an amoeba join query $Q = AJ^*_F$(book, [pending, order, title]), which computes the equivalent classes of book nodes while resolving FDs. First, to detect equivalence of book nodes, the schedule of Q collects book and book@isbn nodes, then perform PC-join of them since book@isbn $\rightarrow$ book (P1). Among the inputs of Q, book and order have a structural constraint given by the FD order $\rightarrow$ book, so we have to connect them by using a AJ^* operation (P2), then compute structural fluctuations of book, pending, order and title nodes (P3). **Figure 3.4** shows the intermediate result in the phase (P3). Although every node id of book@isbn in **Figure 3.4** is different, by comparing their key values, that is [book@isbn], we can find equivalent rows (amoebas). In (P4), equivalent rows are merged to fill the blank column, and incomplete rows that still has *null* values are eliminated. Finally, by using projection π, the query reports only requested nodes by users in (P5), and the result is the lower table in **Figure 3.4**.

Chapter 4

Amoeba Join Processing

Amoeba join can be used as a query method which does not require explicit path structures in query statements; tag names or keywords are sufficient to perform searches. This chapter introduces several amoeba join processing algorithms. The performance evaluation of these algorithms will be described in Chapter 6.

4.1 Preliminaries

Given a list I of input amoeba domains $D_1, \ldots, D_k$ and a list NL of node labels, an amoeba join $AJ_{NL}(I)$ receives instance tuples $d = (d_1, \ldots, d_k)$, where $d \in [\![D_1]\!] \times \cdots \times [\![D_k]\!]$. Then, the amoeba join $AJ_{NL}(I)$ locates a set of instance tuples, such that the projection of each instance tuple d on NL, denoted by $\pi_{NL}(d)$, composes *amoeba*.

Note that, amoeba join processing is independent of node retrieval from databases; we can separate database scans for instances of amoeba domains from the amoeba join processing. This independence is important because it enables amoeba join to be incorporated into other existing query-processing techniques.

In the algorithm descriptions that follow, we assume that every XML node is labeled with an interval $(start, end)$ [33]. This type of the XML index is illustrated in **Figure 5.1** in Chapter 5. A pair of two arbitrary intervals is disjoint; one subsumes the other as a subrange. By encoding XML tree structure hierarchy in the form of an interval tree, detection of ancestor-descendant relationships between two nodes becomes a containment test of two intervals, i.e. a node p is an ancestor of another node q iff $p._{start} < q._{start} \wedge q._{end} < p._{end}$.

4.2 Brute-force Amoeba Join

First, we describe a process to determine whether a given node tuple is an amoeba. The function isAmoeba(t) receives a node tuple $t = \pi_{NL}(d)$, which is a projection of an instance tuple d on NL, and returns *true* if it finds a node interval in t with the smallest start value that completely contains the other intervals. Such an interval is the common ancestor of the others; i.e., this node tuple constructs an amoeba.

With the decision function isAmoeba(t), we can write a simple brute-force amoeba join processing algorithm (**Algorithm 1**). This brute-force version computes all permutations of the input sets, but is apparently inefficient.

Algorithm 1 Brute-force Amoeba Join Algorithm

```
Input: Instances of amoeba domains $D_1, \ldots, D_k$, and a list $NL$ of node labels
Output: A set $R$ of amoebas
1: $R \Leftarrow nil$
2: for all instance tuple $d$ in the permutation $[\![D_1]\!] \times \cdots \times [\![D_k]\!]$ do
3:    if isAmoeba($\pi_{NL}(d)$) then
4:       push $d$ into $R$
5: return $R$
```

Two more efficient amoeba join algorithms are detailed below. The sweep algorithm improves the brute-force algorithm by sequentially sweeping the input node sets. The quicker algorithm reduces disk I/Os by localizing search regions.

4.3 Sweep Algorithm of Amoeba Join

By sorting the input amoeba domains in advance in the order of start values of nodes specified in NL, it becomes more efficient to find amoebas because the amoeba root of a node tuple $t = (t_1, \ldots, t_m)$ always has the smallest start value in $t_1, \ldots, t_m$. The sweep algorithm (**Algorithm 2**) searches amoeba root nodes by sweeping the sorted input node sets.

Algorithm 2 Sweep Amoeba Join Algorithm

```
Input: Sorted instances of input amoeba domains $D_1, \ldots, D_k$, and a list $NL$ of node labels
Output: A set $R$ of amoebas
 1: $R \Leftarrow$ nil.
 2: while true do
 3:   if some of $[\![D_1]\!], \ldots, [\![D_k]\!]$ is empty then
 4:     return $R$ // no more amoeba tuples
 5:   create a node tuple $t = (t_1, \ldots, t_m) = \pi_{NL}(d)$, where $d$ is an instance tuple
      $d$ =($[\![D_1]\!]$.front, ... $[\![D_k]\!]$.front)
 6:   Let $s$ be the smallest start node index in $t$, then $t_s$ is the smallest node in $[\![D_1]\!], \ldots,$
      $[\![D_k]\!]$
 7:   if isAmoeba($t$) then
 8:     // $s$ is the amoeba root node index in $t$
 9:     By searching the range of $(t_s.start, t_s.end)$ in each $[\![D_j]\!]$ $(1 \leq j \leq k, j \neq s)$, collect
        instances of $[\![D_j]\!]$ that have descendant nodes of $t_s$, which are specified in $NL$,
        then construct a set $A_j$ of these instances.
10:     $A_s$ is a set containing only the instance of $[\![D_s]\!]$ which has the current amoeba
        root $t_s$.
11:     If every $A_j(1 \leq j \leq k)$ is not empty, all permutations of $(A_1, \ldots, A_k)$ construct
        amoeba tuples, so insert them into $R$.
12:   remove the instance containing $t_s$ from $[\![D_s]\!]$ // all amoebas rooted by $t_s$ are found
```

In Step 6 of **Algorithm 2**, a node t_s with the the smallest value in the input is assumed to be an amoeba root. Because no other element in the input sets has a smaller start value than t_s, scanning the range of $(t_s.start, t_s.end)$ in $D_j(1 \leq j \leq k, j \neq s)$ is sufficient to find all descendant nodes of t_s (Step 9). Then using these descendant nodes and t_s, we can enumerate all amoeba tuples rooted by t_s (Step 11). When the algorithm reaches Step 12, it is assured that all amoebas whose root's start value is smaller than or equal to the current amoeba root candidate t_s are found.

4.4 Heuristics for Search Space Reduction

Here, we introduce the *quicker algorithm*, a more elaborate version of amoeba join, which is integrated with index look-ups. While the sweep algorithm reads all nodes in the given query domains from the database, the quicker algorithm (**Algorithm 3**) tries to reduce this disk I/Os. For brevity of explanation, to specify each instance of an input amoeba domain, we mention only nodes, which belongs to NL, contained in the instance. For example, we denote a node v specified in NL in an instance of an amoeba domain D as $v \in [\![D]\!]$ ignoring the other nodes contained in the instance.

For a node tuple to be an amoeba, each node in the tuple must be a descendant of the amoeba root node. When a node v is considered a part of an amoeba, its amoeba root is either v or one of its ancestor nodes. **Figure 4.1** illustrates this idea of localizing database scans within the descendant area of an amoeba root node candidate. Given a pivot node, which is considered a component of an amoeba, the quicker algorithm in Step 5 finds its ancestor nodes, i.e., amoeba root candidates, then searches the descendant area for other components of amoeba tuples. The quicker algorithm chooses pivot nodes from the smallest domain, say D_i, because the smaller the cardinality $|D_i|$, the fewer amoeba root candidates and their descendant nodes (components of amoebas).

For this purpose, we use the frequency count (or its estimation) of nodes belonging to each of the query domains. Given domains of an amoeba join query $AJ_{NL}(D_1, \ldots, D_k)$, let $E = (e_1, \ldots, e_k)$ be frequency of $D_1, \ldots, D_k$. When the value of $|D_i|$ is available, $e_i = |D_i|$, if not, $e_i = \infty$. A function f sorts e_i so that $e_{f(1)} \leq \cdots \leq e_{f(k)}$. Quicker algorithm chooses pivot nodes from $D_{f(1)}$ (Step 4).

The quicker algorithm (**Algorithm 3**) utilizes three types of index scan; for retrieving nodes in $D_{f(1)}$, which is the smallest domain (Step 2); for retrieving ancestor nodes of a pivot node (Step 5); and for scanning descendant nodes of an amoeba root candidate (Step 11). A database index that supports these three types of index scans is required to perform the quicker algorithm. In Chapter 5, we show an efficient XML index that supports these scans.

This type of search space reduction (**Figure 4.1**) is not available in the *lca* method, because an *lca* node tends to be the root of XML; it does not reduce the search space at all. Another reason that makes this optimization possible is the design concept of amoeba join,

Algorithm 3 Quicker Amoeba Join Algorithm

Input: Input amoeba domains $D_1, \ldots, D_k$, a list NL of node labels, and sorting function f

Output: A set R of amoebas

```
 1: Initialize priority queues (sorted by start order) $Q_i \Leftarrow empty$ $(i = 1, \ldots, k)$
 2: fill the $Q_{f(1)}$ with nodes in $D_{f(1)}$ by fetching from the database (index scan)
 3: for $i = 1 \ldots |D_{f(1)}|$ do
 4:   $pivot = Q_{f(1)}$.top
 5:   query pivot's ancestor nodes (index scan), then push them into corresponding $Q_p(p \neq f(1))$.
 6:   repeat
 7:     $s$ = the smallest start node index in $Q_1$.top, ..., $Q_k$.top.
 8:     $t_s = Q_s$.top // an amoeba root candidate
 9:     pop all entries $q$ ahead of the $t_s$, i.e. $\forall q \in Q_i, q.start < t_s.start$
10:     for $j = f(1) \ldots f(k)$ do
11:       push unread descendant nodes of $t_s$ in $D_j$ into $Q_j$. (index scan)
12:       goto Step 17 if $Q_j$ is empty ($t_s$ cannot be an amoeba root)
13:     // all of the $Q_p(p \neq f(1))$ is not empty
14:     By searching the range of $(t_s.start, t_s.end)$ in each $Q_j(1 \leq j \leq k, j \neq s)$, collect descendant nodes of $t_s$, then construct a set of these nodes $A_j$.
15:     $A_s = \{t_s\}$ // contains only the current amoeba root candidate
16:     If every $A_j(1 \leq j \leq k)$ is not empty, all permutations of $(A_1, \ldots, A_k)$ construct amoeba tuples, so insert them into $R$.
17:     pop $Q_s$ // all amoebas rooted by $t_s$ are computed
18:   until $s = f(1)$ // exit when the pivot node is popped
19: return $R$
```

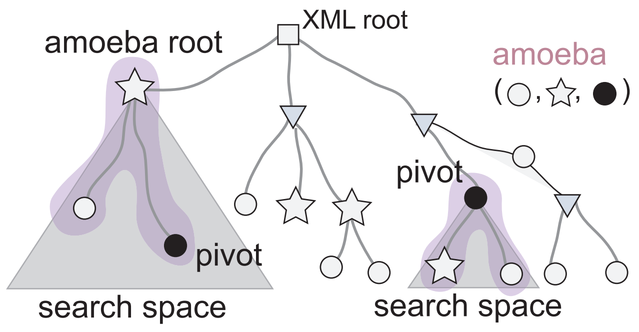


Figure 4.1: A small number of pivot nodes helps to reduce index scan ranges.

which tries to find common ancestor nodes from *specific domains*, while the approach of the *lca* or *slca* [34, 56] is to find common ancestors from the *entire nodes* in an XML document.

4.5 Disk I/O Performance

When the height of an XML tree is h, the number of nodes retrieved by one ancestor query is at most h. The quicker algorithm retrieves ancestor nodes for each node in $|D_{f(1)}|$, and thus it requires $h|D_{f(1)}|$ node retrievals from the database. Another factor that defines the disk I/O performance of the quicker algorithm is how many node retrievals the heuristic of **Figure 4.1** saves. Let D'_i be a sub-domain of D_i, which is retrieved by the quicker algorithm; then, the number of nodes scanned in the quicker algorithm is $h|D_{f(1)}| + |D'_1| + \cdots + |D'_k|$. However, the sweep algorithm consumes all nodes in the query domains; i.e., it searches $|D_1| + \cdots + |D_k|$ nodes. When $|D_{f(1)}|$ is sufficiently small, as in the example shown in **Figure 4.1**, $|D'_i|$ is typically considerably smaller than $|D_i|$. In addition, the height of the XML, h, is generally limited; only rarely is h larger than 100. Consequently, the quicker algorithm is often less costly in terms of disk I/O than the sweep algorithm.

This search space reduction is similar to *pushing selection*, a query optimization technique for relational databases [39]. XML typically contains many repeat paths, and therefore, reducing the size of query domains by attaching conditions, such as predicates on text values, to the path expression queries is a common method. Hence, the quicker algorithm utilizes a simple optimization to reduce disk I/Os.

Chapter 5

XML Indexing

Several indexing methods have been proposed to encode tree structures and path structures of XML, which are generally called *structure indexes*. These indexes enable us to efficiently query XML data according to several structural conditions. Although amoeba join operations can be performed without using structures, initial inputs of the amoeba join, e.g. //book, //customer/name, etc., should be retrieved using node names or some path contexts. In addition, the quicker algorithm described in the previous chapter requires retrievals of ancestor or descendant nodes of a specific context node.

From these considerations, we observed that an index for XML has to process tree structures and path structures *simultaneously*. In this chapter, to handle these various aspects of XML, we present the design of our novel multidimensional index. Previous work of XML indexing has often developed specialized data structures tailored to some query patterns to handle this multidimensionality, however, their availability to the other types of queries has been obscure. Our method can be implemented on top of the standard B+-tree, and is adaptive to various combinations of structural predicates, including suffix paths, ancestor, descendant, etc.

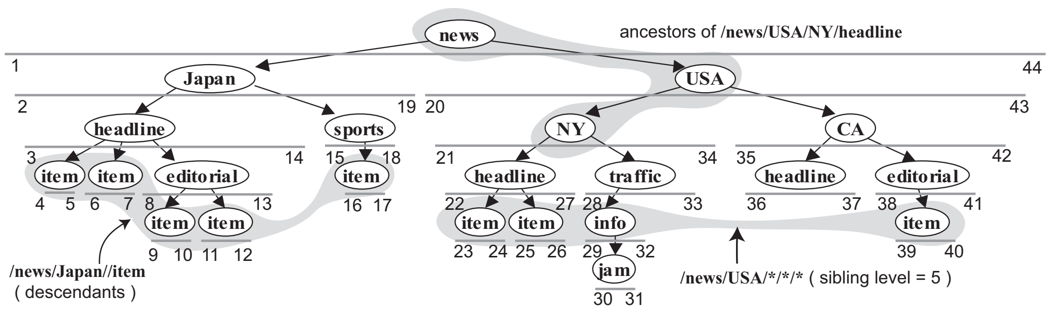


Figure 5.1: Interval labels of XML

5.1 Backgrounds

5.1.1 Tree-Structure Indexes

XML has a tree structure, however, ancestor and descendant axes cannot be efficiently evaluated with standard tree navigations such as depth-first or breadth-first traversals. To make faster the process of ancestor-descendant queries, two types of node labeling methods have been developed; the *interval label* [33] and the *prefix label* [37]. The interval label (see also **Figure 5.1**) utilizes containment of intervals to represent ancestor-descendant relationships of XML nodes. For example, if an interval label of a node *p* contains another interval of a node *q*, *p* is an ancestor of *q*. The prefix label assigns node id paths from the root node to each node so that if the label of a node *p* is a prefix of the label of a node *q*, *p* is an ancestor of *q*. Both of the node labeling strategies are fundamentally same in that they are designed to instantly detect ancestor-descendant relationships of two XML nodes. This operation is called *structural joins* [3]. **Figure 5.1** shows an example of the interval label. The interval assigned to each node subsumes intervals of its child and descendant nodes. These node labels are favorable in that they can be aligned in the document order of nodes by seeing start values of these intervals. Therefore, these labels can also be used to traverse the tree structures of XML in both of the depth-first and breadth-first manner. We call this types of indexes for tree navigation, *tree-structure* indexes.

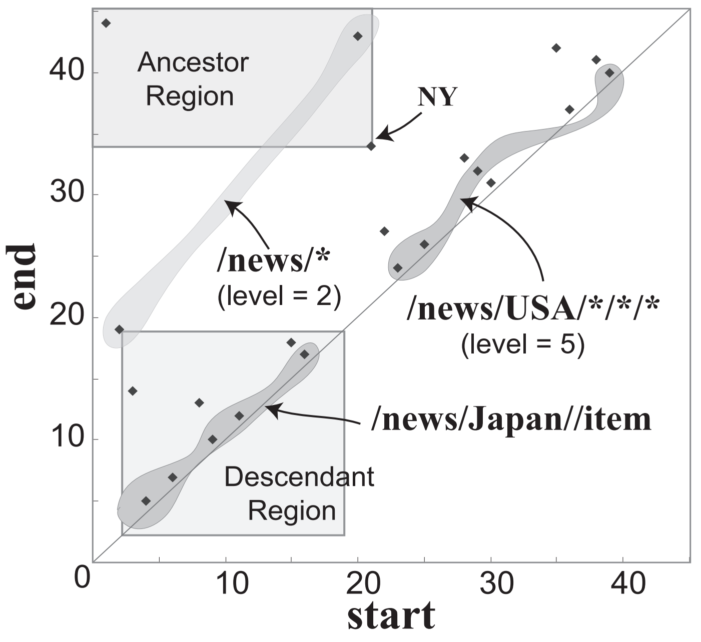


Figure 5.2: Projection of the interval labels (start, end) on a 2D-plain

5.1.2 Path-Structure Indexes

To efficiently evaluate path expression queries, in addition to the tree-structure indexes, we also need *path-structure* indexes, which reduce the overhead of tree navigation by clustering nodes that belong to the same tag or attribute name paths. There are many proposals how to encode path structures of XML, varying from simply assigning an integer id to each independent path in the XML document [29] to creating a summary graph of path structures [18, 35].

When a query contains the descendant-axis (//), we can localize the search spaces for the descendant nodes. For example, an XPath query //Japan//item can be decomposed into two paths; //Japan and //item, and the search space for //item will be localized according to the results of //Japan (**Figure 5.1** and **Figure 5.2**). Therefore, the tree-structure and path-structure indexes should be integrated to evaluate this types of queries.

In addition, we usually have to query not only with tag names of XML nodes but also with *suffix paths*. For example, an XPath //Japan//headline/item contains a suffix path

path ID	inverted path
1	news\
2	Japan\news\
3	headline\Japan\news\
4	headline\NY\USA\news\
5	headline\CA\USA\news\
6	item\headline\Japan\news\
7	item\headline\NY\USA\news\
8	item\editorial\Japan\news\
9	item\editorial\CA\USA\news\
10	item\sports\Japan\news\
11	editorial\Japan\news\
12	editorial\CA\USA\news\
13	sports\Japan\news\
14	USA\news\
15	NY\USA\news\
16	traffic\NY\USA\news\
17	info\traffic\NY\USA\news\
18	jam\info\traffic\NY\USA\news\
19	CA\USA\news\

Figure 5.3: Inverted Path Labels of **Figure 5.1**

//headline/item. Rather than querying headline and item nodes individually, it is far more efficient to scan nodes that have suffix paths //headline/item directly, since there are many item nodes whose parents are not the headline nodes. To improve accessibility to suffix paths, the p-label has been proposed [57]. The essence of the p-label is to invert the sequences of paths occurring in the XML document, which we call *inverted paths*, and align these inverted paths in the lexicographical order considering each tag or attribute name in the paths as a comparison unit. **Figure 5.3** shows an example of inverted paths, where each inverted path is labeled with an integer id. To evaluate an XPath query //item, we have to collect nodes whose inverted path ids are contained in the range $[6,11)$. When a more detailed path is specified, for example, //headline/item, the query range narrows to $[6,8)$. With the inverted path ids, we can perform suffix path queries with range searches.

5.2 Multidimensional Aspects of XML

In this section, we show why the integration of tree-structure and path-structure indexes is so important. **Figure 5.2** shows the mapping of the intervals (start, end) in **Figure 5.1** into a two-dimensional plane. The benefit of the interval labeling is that we can enclose all ancestor (descendant) nodes of some nodes within its upper left (lower right) rectangular region. For example, all ancestor nodes of the NY node $(21,34)$ is enclosed in its upper left rectangle. The process of a query, say /news/Japan//item, has to accurately extract item nodes within the subtree rooted by Japan (a shaded region in the figures). Since some item nodes exist out of this region, the index structure for XML demands the capability to capture nodes by the combination of start, end, and paths.

Although the 2D plane is useful to track ancestor and descendant nodes, we also need the information of the node depth (level) to process wild-cards in path expressions. For example, XPath expressions /news/* and /news/US/*/*/*, which are useful to investigate the structure of XML database, require the level information to efficiently collect the answer nodes, since without indexes for the level values, its process has to traverse the tree-structure in depth-first or breadth-first manner; it is inefficient when the depth is deep. Our experimental results confirm the inefficiency of these tree traversal methods for wild-card queries, i.e. the sibling-axis steps.

XPath [15] has 11 types of axis steps for tree navigations, that are *ancestor*, *descendant*, *parent*, *child*, *attribute*, *preceding-sibling*, *following-sibling*, *ancestor-or-self*, *descendant-or-self*, *preceding*, and *following*[1]. Among them, the six types of axis steps, ancestor(-or-self), descendant(-or-self), preceding and following, can be processed with the two-dimensional indexes for the interval labels (start, end). The parent, child, preceding-sibling and following-sibling axis-steps require all of start, end, level values, since start and end values are not sufficient to detect parent-child relationships of nodes. If attribute nodes of XML are modeled as child nodes of tags, the attribute-axis can be seen the same with the child-axis. Therefore, all of 11 axis-steps can be processed with the combination of (start, end, level) indexes, i.e. a tree-structure index.

In addition to the tree-structure indexes, if we have a path-structure index, we can efficiently answer XPath queries, even if these answers are contained in meandering regions as illustrated in the query region for /news/Japan//item in **Figure 5.2**. Therefore, to create a multi-dimensional index of tree-structures and path-structures of XML is a key to accelerate XML query processing.

[1]The other two axis steps defined in XPath [15] are the *namespace* and *self* axis, which do not require any tree traversal.

5.3 Multidimensional XML Index

In order to construct an XML database, we utilize a combination of tree-structure indexes, and path-structure indexes. Although, there are many proposals for labeling each type of index, we adopted labels that can be easily represented with integers.

We encode every XML node with a label:

$$(\mathsf{start}, \mathsf{end}, \mathsf{level}, \mathsf{path}, \mathsf{text}),$$

where start and end represent interval labels of XML, and level is the node depth. The path is the inverted path id described in Section 5.1.2. The text is a text content enclosed in the tag or attribute. Every attribute element in XML is assigned the same interval and level value with its belonging tag, so as to learn the subtree range of the tag from the index of the attribute node.

Although we utilized the interval labels for tree structures, other labeling schemes, such as prefix labels, can substitute them, and the XML label will be (prefix-label, level, path, text). Each prefix label contains all prefix labels of its ancestor nodes, so there is no need to have end values for ancestor queries. The path labels also can be replaced simply with tag IDs or other labels.

The above labeling scheme is used to create our multidimensional index. To index multidimensional data, it is general to use R-tree, which groups together nodes that are in close spatial proximity. However, implementations of R-tree are not yet as matured as B+-tree, which is broadly employed in the industrial strength DBMSs, while R-tree is not. Although B+-tree is a one-dimensional index structure, we can store the multidimensional data into the B+-tree by using a space-filling curve [42], such as Hilbert curve, Peano curve etc. The space-filling curve traces the entire multidimensional space with a single stroke, and it can be used to align multidimensional points in one dimensional space.

However, what kind of space-filling curve is suited for XML indexing? To answer this question, let us confirm our objective to construct multidimensional indexes; that is to make clusters of nodes that have same attribute values as possible, for example, nodes having same level values and same suffix paths, so that we can efficiently query nodes with combinations of these attribute values, i.e. start, end, level and path.

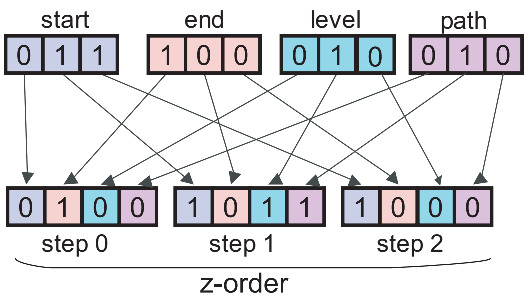


Figure 5.4: Interleave function generates a z-order from an index (start, end, level, path)

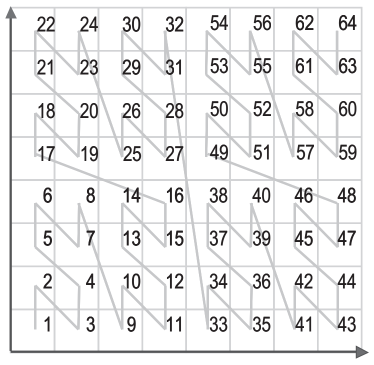


Figure 5.5: Z-order specifies a position on the z-curve.

To meet this demand, we chose a straightforward approach; *bit-interleaving* of coordinate values. It gives a position on the *z-curve* [38, 42], which is also a space-filling curve. The interleave function illustrated in **Figure 5.4** receives coordinate values of a point as input, and from their bit-string representations, it retrieves single bits from heads of coordinate values in a round-robin manner, then computes the *z-order*, which is an absolute position on the z-curve (**Figure 5.5**). This linear ordering of XML nodes enables us to implement the multidimensional index on top of the B+-tree. In addition, each step in the z-order in **Figure 5.4** has a role to split each dimension. The first step splits each dimension into two, and the second step splits each slice into 2, resulting in $2^2 = 4$ slices, and so on. If two nodes are close in the multidimensional space, their z-values are also likely to be close in the some steps. It means these nodes will be probably placed in the same leaf page in the B+-tree, and it is the nature of bit-interleaving.

5.3.1 Normalizing Index Resolution

The interleave function extracts bits beginning from the MSB (most significant bit). When value domains of the interleaved indexes are far different, for example, the domain of start values is $0 \leq start < 2^{10}$, and that of level values is $0 \leq level < 2^3$, the change of a value in a smaller domain has as equal significance to the z-order as that of the larger domains. In general, the depth of XML documents is not greater than 100, while the interval label for XML requires as large a value as the number of nodes, which can be more than 100,000. Thus, if we use the same bit-length number to represent each index value, the level values are less important in the z-order, and we fail to separate XML nodes level by level. It will deteriorate the sibling query performance.

To avoid this problem, we adjust the *resolution* of each index, which is the maximum bit length that is enough to represent all values in the index domain. We denote the resolution of an index as r. For example, when a domain of some index is a range $[0, v_{max})$, its resolution r is $\lceil \log_2 v_{max} \rceil$. The $normalize_m(v)$ function converts an integer value v, whose resolution is r, into an m-bit integer value. We define $normalize_m(v) = \lfloor v/2^{r-m} \rfloor$, ignoring the fraction. For example, when $m = 8$ and the resolution of each index of (start, end, level, path) is 10, 10, 3 and 4, respectively, an XML index (100, 105, 3, 2) is normalized to the 8-bit values (25, 26, 96, 32). By using normalized index values to compute z-orders, we can adjust the resolution so that level or path values, whose domains are usually small, affect much more to the z-order than start or end values. We simply denote this normalization process for some node p as $normalize(p)$.

5.3.2 Range Query Algorithm

The idea that utilizing z-curve for multidimensional indexes is first mentioned in the zkd-BTree [38] and is improved in the UB-tree [7]. Although both of them extended the standard B+-tree structure to make it efficient for multidimensional queries, we introduce a multidimensional range query processing algorithm without modifying existing B+-tree structures. In our algorithm, we need only two standard functions for the B+-tree; *find* and *next*. The $find(k)$ receives a key value k and finds the smallest entry whose key value is greater than or equal to k. The $next(e)$ returns the next entry of an entry e in the B+-tree.

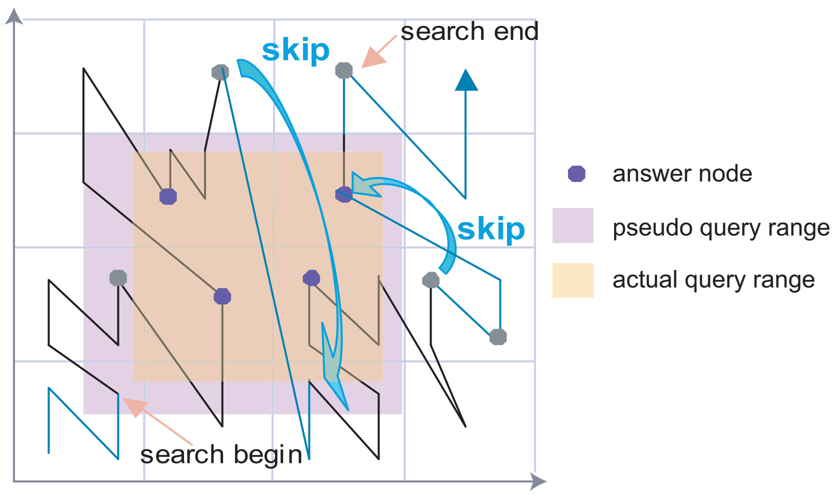


Figure 5.6: Range query algorithm

We denote the z-order of a node p = (start, end, level, path) as $zorder(p)$, and coordinates specified by the z-order z as $coord(z)$. Then, $coord(zorder(p)) = p$. Each entry in the B+-tree has the structure: $zorder(normalize(p)) \Rightarrow p$, and sorted in the z-order. To perform a multidimensional query for a hyper-rectangle region $Q(p_s, p_e)$, where p_s and p_e are the multidimensional points specifying the beginning and end points of the query range, we can utilize a property of z-orders; all points p in the query range Q satisfy $zorder(p_s) \leq zorder(p) \leq zorder(p_e)$ [7].

Algorithm 4 shows the range query algorithm, and **Figure 5.6** illustrates its behavior. Since all nodes are aligned in the z-order in the B+-tree, we have to scan the key range of z-order from $zorder(normalize(p_s))$ to $zorder(normalize_{ceil}(p_e))$, where $normalize_{ceil}(p)$ is a function to calculate ceiling values, $\lceil v/2^{r-m} \rceil$, of each normalized coordinate value $v/2^{r-m}$, where r is a resolution of a dimension, and m is a common bit length after normalizing each coordinate. That z-orders computed from normalized coordinate values may have round errors, so there is a case that $coord(normalize(p))$ is contained in the normalized query range $NQ(normalize(p_s), normalize_{ceil}(p_e))$, but p is not in Q, since if we de-normalize NQ, illustrated in **Figure 5.6** as pseudo-query range, it is always equal to or larger than Q. Even though, the containment test for NQ (Step 10) is useful to detect whether the current z-order is completely out of range of Q. In this case, we can compute the *nextZorder*

Algorithm 4 Range query algorithm

```
Input: $Q(p_s, p_e)$ : query range
Output: A node set within the query range
1: $NQ = (normalize(p_s), normalize_{ceil}(p_e))$ // normalized query range of $Q$
2: $z_s = zorder(p_s)$, $z_e = zorder(p_e)$
3: $z = z_s$ // set the initial z-order to the beginning of the query range
4: // find an entry $e$ in the B+-tree that has the smallest z-order larger than $z$.
5: $e = find(z)$
6: while $e$ is not $nil$ do
7:    $z = e.z$   // $e.z$ is the z-order (key value) of the entry $e$
8:    if $z > z_e$ then
9:       return // end of the query
10:   if $coord(z)$ is contained in $NQ$ then
11:      while $e$ is not $nil$ and $e.z = z$ do
12:         // retrieve nodes whose z-order is $z$ in the B+-tree
13:         if the entry $e$ is contained in $Q$ then
14:            output $e$
15:         $e = next(e)$   // move to the next entry of $e$ in the B+-tree
16:   else
17:      $nextZorder$ = the smallest z-order larger than $z$ and contained in $NQ$.
18:      $e = find(nextZorder)$
```

that re-enters into the query box NQ (Step 17). It skips some nodes in the outside of the query box and saves disk I/O costs. An efficient algorithm to compute next z-orders is described in [40]; in essence, this algorithm locates the most-significant bit-position, say j, in the z-order that can be safely set to one without jumping out of the query range, then adjusts other bit values which are lower than j so that the z-order becomes the smallest one contained in the the query range but larger than the original z-order.

Chapter 6

Experimental Results

We evaluated the performance of our XML indexing method in comparison with the standard XML index structures. By using these indexes, we also measured the performance of several amoeba join processing algorithms, and the performance gain achieved by the amoeba-join decomposition techniques.

6.1 Settings

Implementation. We implemented all of the database indexes and algorithms in C++. The index implementation uses B+-trees provided by the BerkeleyDB library [50]. Given an XML document, we labeled each XML node with (start, end, level, path ID, text). The pair (start, end) is an interval representation of XML nodes [33], described in Chapter 5. The level is the depth of a node in the XML tree, and required to detect parent-child relationships of XML nodes. The path ID represents an inverted path ID assigned to each independent path. The text is a text content encapsulated by tags or attributes.

Machine Environment. As a test vehicle, we used an Windows XP, Pentium M 2GHz notebook with 1GB main memory and 5,400 rpm HDD (100GB).

Database Settings. The page size of the B+-tree is set to 1K. We prepared a somewhat larger size of the page caches (128MB) so as to ignore the effect of inadvertent disk I/O delay, which is usually caused by random-disk accesses. This is because moving the disk head is the most time-consuming disk operation, in addition, the effect of which significantly varies according to OS status and the types of storages, e.g. IDE, SCSI, RAID5, flash memories, etc. Hence, we set up so that most of the queries can be performed on the main memories after the database contents are cached, but a query that inherently requires many page-fetch operations has to perform actual disk reads; This type of query processing becomes slower when we have smaller caches. In the following experiments, for each query, we measured the average performance of hot-runs when disk pages are fully or partially cached.

6.2 XML Index

First, we evaluated the query performance of the proposed index in Chapter 5 for several types of queries, e.g. ancestor, descendant, sibling, and path-suffix queries, which are the basic components to process more complicated queries such as structural joins [3], twig-queries [27], etc.

Our system, called Xerial, uses the index structure described in Chapter 5, which has the form: *z-order* ⇒ (start, end, level, path, text), where the values of *z-order* are keys of B+-tree entries; each XML nodes are sorted in ascending order of the z-order within the B+-tree. Every z-order is represented with 64-bit integer. And also, all of start, end, level and path values are described with 32-bit integers. These values are normalized to 16-bit integers when computing z-orders.

To clarify the benefit of our method, we prepared two competitors for Xerial; start index and path-start index. The start index simply sorts XML nodes in the order of start values. It has the data structure (start ⇒ end, level, path, text) in B+-tree. The path-start index, ((path, start) ⇒ end, level, text), sorts nodes first by path, then by start orders. These structures can localize search space of path queries within some subtree range, and similar structure is utilized in [14, 33]. However, the following experiments reveal that such simple integration of indexes has several weak points.

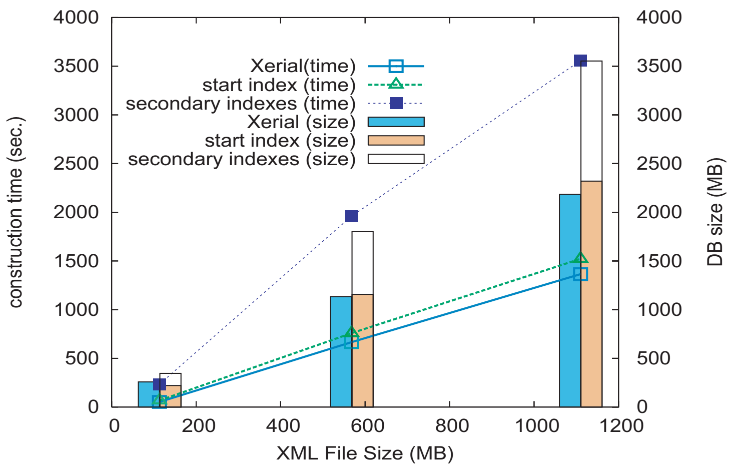


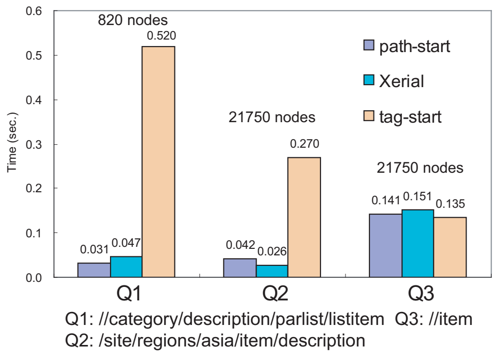
Figure 6.1: DB construction time and DB size.

6.2.1 Database Size

We compared database sizes of start index and Xerial. **Figure 6.1** shows their actual database sizes and construction times for various scaling factors (1 to 10) of the XMark's benchmark XML documents [49]. The secondary index in **Figure 6.1** shows the database size if we constructs three B+-tree indexes for end, level, path values to complement the functionality of the start index. Even though Xerial has additional z-orders, its database size is almost the same with the start index, and also it is much more compact than creating multiple secondary indexes. It is mainly because the B+-tree of Xerial has some duplicate entries having the same z-orders, and it makes smaller the number of key values stored in the internal pages of the B+-tree, while the start index stores each start value, which has no duplicates. Therefore, the use of z-order is beneficial to make lower the depth of the B+-tree.

6.2.2 Query Performance

The following experiments are conducted on an XMark document (113MB, scaling factor = 1.0), and we measured the average times for individual query operations, ignoring the output costs of reporting the query results.



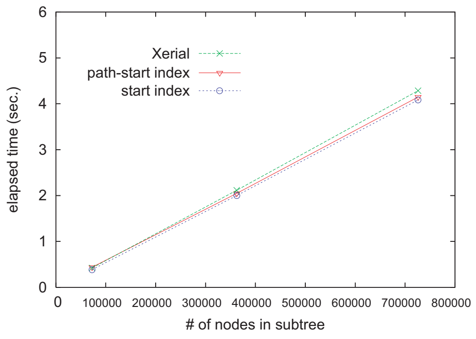


Figure 6.2: Suffix-path query performance (left)

Figure 6.3: Subtree query performance (right)

Suffix Path Query. First, we compared performance of suffix path queries. **Figure 6.2** shows how fast each index can collect nodes that have the same path suffixes. The path-start index, which has clusters of suffix paths is the fastest, and Xerial performs as fast as the path-start index, because the interleave function of Xerial also plays a role to group together nodes which have the same suffix path. The start index is weak in processing this type of queries since it has to scan the whole index because the information of paths is stored in the leaf pages of the B+-tree, so we omit its result.

In order to show that the importance of having flexibility for the choice of node labels, we also compared the performance of suffix path queries when inverted path cannot be used. The tag-start index uses tag IDs instead of inverted path IDs, so it must perform several nested structural joins [3] to achieve the answer, and shows poor performance other than the $Q3$, that is the tag-only query.

Subtree Retrieval. The start index is the most suitable data structure for subtree retrievals because nodes in a subtree are sequentially ordered. It shows the fastest result (**Figure 6.3**). Nevertheless, both of Xerial and path-start index show almost identical performance to the start index.

Ancestor Retrieval. Ancestor query is useful to retrieve parent or ancestor information from some node directly accessed from additional secondary index structures such as the one for traversing IDREF edges, or inverted indexes for text contents. This query needs to find nodes which satisfy $start < s \wedge e < end$, where (s, e) are start and end position

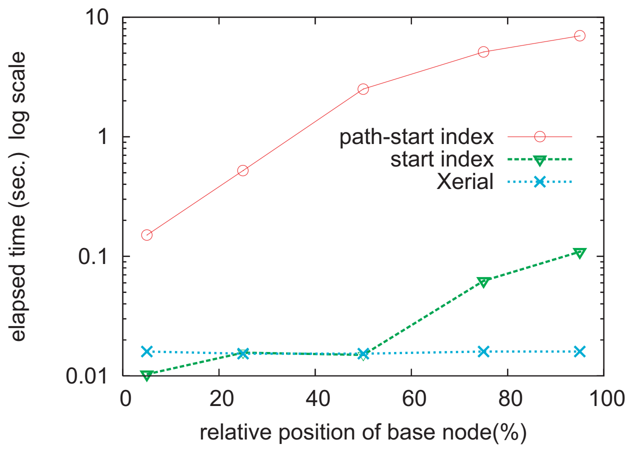


Figure 6.4: Ancestor query performance

of the base node of the query. **Figure 6.4** shows the performance of the ancestor queries for various positions of base nodes, whose level is 12. The start index processes this query from the root node, and it skips subtrees which are not the ancestor of the base node. The performance of Xerial is stable, because it can eliminate the search space by using a combination of start and end axes. On the other hand, the path-start index breaks the start order down into multiple clusters grouped by path IDs. Consequently, it cannot utilize the tree structure of XML. In addition, it cannot eliminate the search space by using the end values, therefore it is inefficient when the base node of the query has a lot of preceding nodes in the document order. The start index has the same deficit. This result indicates that the ancestor query performance of start and path-start indexes depends on the database size, and the position of the query context node.

Sibling Retrieval. Notable usage of sibling node retrievals is to find blank spaces for node insertions, to compute parent-child joins and wild-card(*) queries. Xerial remarkably outperforms the other indexes (**Figure 6.5**). This is because these indexes except Xerial have difficulty to find nodes in the target level. The start index must repeat searching the tree for a node in the target level with a depth-first traversal, while skipping unrelated descendant nodes occasionally. The path-start index performs this process in every cluster of paths. This descendant skip works well when the target depth of sibling is low; however, as the level becomes deeper, it cannot skip so many descendants and the cost of the B+-tree

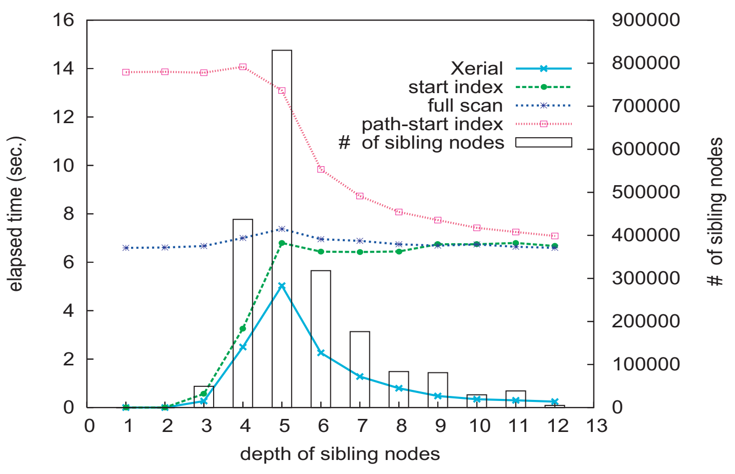


Figure 6.5: Sibling query performance

searches increases. To see this inefficiency, we also provided the result using sequential scan of the entire index, and it shows similar performance to the start index and path-start index for deep levels.

In summary, to efficiently process queries of suffix paths, siblings, subtrees and ancestors, the start-index and the path-start index require additional secondary indexes. For example, start index should have indexes for level and path, and path-start index needs at least three indexes for end, level, and suffix path. Our experiments show Xerial's all-around performance for various types of queries. In spite of this faculty, the index size remains compact.

Discussion. Here, we would like to mention some tips that finally lead us to this performance. At first, we used 32-bit integers to represent z-orders, but this implementation performs poorly for every types of queries in the experiments. This is because the 32-bit z-order splits each dimension only to 2^8 slices. It is too coarse and results in that too many nodes are assigned the same z-orders; there are many overflowed B+-tree pages, slowing down every search operations. On the other hand, if we use the finest resolution so that every point in the multidimensional space has a unique z-order, such large key values will soon fill the internal pages of the B+-tree and end up lowering the B+-tree's branching factors. This also deteriorates the search speed. The optimal resolution might be achieved when *each disk page* have a unique z-order. To accomplish this, the UB-tree [7] extended the implementation of the B+-tree.

6.3 Amoeba Join Processing

We measured the performance of three amoeba join algorithms explained in Chapter 4; brute-force (BF), sweep (SW), and quicker algorithm(QK). The first two algorithms can incorporate various indexing techniques, so we compared them using sequential scans (S) of XML nodes, and more efficient index-based scans (I), using the XML index of Xerial. This led to five types of amoeba join algorithms: BF/S (brute-force with sequential scan), BF/I (brute-force with index scan), SW/S (sweep join processing with sequential scan), SW/I (sweep join processing with index scan), and QK (quicker algorithm), which is a mixture of index scanning and join processing. The goal of this section is to demonstrate how we can reduce the computational cost of amoeba joins by using combinations of XML indexes and amoeba join algorithms.

Implementation. The sequential scan method (S) reads XML nodes sequentially stored in the B+-tree in the order of their start values. As for the index-base scan methods (SW/I and BF/I) and the quicker algorithm (QK), we used Xerial's index which is efficient for finding descendant nodes that belong to specific paths. The parent node retrieval in the quicker algorithm (QK) utilizes ancestor query in the Xerial's index. In addition, to efficiently retrieve input amoeba domains containing text predicates, we constructed an inverted index for text values (text ⇒ start) that looks up the start value (ID) of a node from its text value. This type of the inverted index is used for QK, SW/I, and BF/I.

The reason we evaluated the sequential scan method is that it is somewhat analogous to node stream processing, such as in handling SAX events, since this scan reads the entire list of nodes to perform a query. Another reason to compare the index-based scan methods to the sequential scan methods is to see that the former method, using some index structures, is not too complex to invoke a lot of random disk access. Too much random

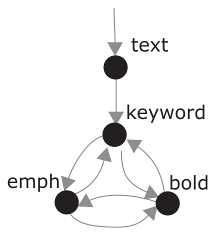
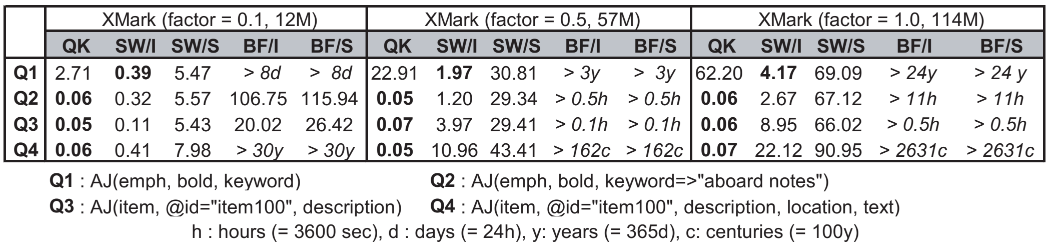


Figure 6.6: Structural fluctuation in XMark



	XMark (factor = 0.1, 12M)					XMark (factor = 0.5, 57M)					XMark (factor = 1.0, 114M)				
	QK	**SW/I**	**SW/S**	**BF/I**	**BF/S**	**QK**	**SW/I**	**SW/S**	**BF/I**	**BF/S**	**QK**	**SW/I**	**SW/S**	**BF/I**	**BF/S**
Q1	2.71	**0.39**	5.47	*> 8d*	*> 8d*	22.91	**1.97**	30.81	*> 3y*	*> 3y*	62.20	**4.17**	69.09	*> 24y*	*> 24 y*
Q2	**0.06**	0.32	5.57	106.75	115.94	**0.05**	1.20	29.34	*> 0.5h*	*> 0.5h*	**0.06**	2.67	67.12	*> 11h*	*> 11h*
Q3	**0.05**	0.11	5.43	20.02	26.42	**0.07**	3.97	29.41	*> 0.1h*	*> 0.1h*	**0.06**	8.95	66.02	*> 0.5h*	*> 0.5h*
Q4	**0.06**	0.41	7.98	*> 30y*	*> 30y*	**0.05**	10.96	43.41	*> 162c*	*> 162c*	**0.07**	22.12	90.95	*> 2631c*	*> 2631c*

Q1 : AJ(emph, bold, keyword) **Q2** : AJ(emph, bold, keyword=>"aboard notes")
Q3 : AJ(item, @id="item100", description) **Q4** : AJ(item, @id="item100", description, location, text)
h : hours (= 3600 sec), d : days (= 24h), y: years (= 365d), c: centuries (= 100y)

Figure 6.7: Amoeba join performance (sec.).

access may make query-processing algorithms slower than a sequential scan of all records.

The quicker algorithm (QK), used rough estimates of node frequencies; if D_i, an input amoeba domain has a text predicate, we assume $|D_i| = 1$ or otherwise $|D_i| = \infty$, because the response size of a keyword search is usually less than that of a path query. Although a more accurate estimation strategy could be accommodated, this is sufficient for locating one of the small domains.

Data Sets. In the following experiments, we also used XMark's benchmark XML documents[49], which contains a lot of structural fluctuations under its text tags. **Figure 6.6** shows a part of its DataGuide [18], a summary of path structure. The cycles in the DataGuide show that three tags keyword, emph, and bold occur in arbitrary order within the document. We prepared three types of XMark document, varying the scaling factors (f = 0.1, 0.5 and 1.0). Their structures were too complex to determine the path structures for a specific context, showing that amoeba join is also useful for querying such complicated XML data.

Amoeba Join Performance

Figure 6.7 shows the performance of the amoeba join queries (Q1 to Q4). In the brute-force algorithms BF/I and BF/S, some the computational complexity was too huge to compute the result; thus, we show their estimation time, which was calculated using the permutation size of a query and the elapsed time for processing its first 500,000 nodes.

In Q1, the quicker algorithm was slower than the sweep algorithm (SW/I) because the sizes of emph, bold, and keyword were fairly large. As a consequence, excessive ancestor node retrievals in the quicker algorithm deteriorated its performance. When a query contains predicates (Q2, Q3, and Q4), the quicker algorithm performs an order of magnitude faster than the others because the size of the domain constrained by a constant gets smaller. Therefore, a combination of QK and SW/I algorithms provides the fastest performance; when there is no low-frequency domain in a query, it uses the SW/I, and otherwise it uses the QK.

The performance of SW/S scaled according to the database size. Although the time required to scan the entire database was the same from Q1 to Q4, the processing of Q4 in SW/S was the slowest; because the tuple size k of a query affects the join performance. The same is true for SW/I. However, the performance of the quicker algorithm was stable regardless of tuple size.

6.4 Nested Amoeba Join

In this section, we present the differences of the amoeba join performance when using amoeba join decomposition techniques.

Data Set. The data set we used in the experiments is synthesized XML documents of the book store data, the data structure of which is the same with the one illustrated in **Figure 3.1** in Chapter 3, but the numbers of book, customer and order nodes are different. We prepared various sizes of this XML data; 1M, 5M, 25M and 50M, in which we defined FDs: F = { book, customer → order, order → book, order → customer}, and a ubiquitous key: [book@isbn] → book.

Figure 6.8 shows performances of various types of amoeba join queries, and **Figure 6.9**

	query statement	1M	#result	5M	#result	25M	#result	50M	#result
Q1	*AJ(book, order)*	0.24	9403	1.14	46146	5.60	223637	11.34	447174
Q2	*AJ(customer, order)*	0.14	9403	0.64	46146	3.24	223637	6.44	447174
Q3	*AJ(order, book, customer)*	3.00	583603	14.89	2857746	Out of Memory		Out of Memory	
Q4	*AJ F (order, book, customer)*	**0.51**	**9403**	**2.11**	**46146**	**10.41**	**223637**	**21.25**	**447174**
Q5	*AJ*(book, order)*	0.32	16094	1.51	79102	7.46	383420	15.17	766740
Q6	*AJ*(book, [order, pending, titile])*	1.17	66094	7.74	359102	112.08	1753420	486.66	3516740
Q7	*AJ* F (book, [order, pending, titile])*	**0.91**	**16594**	**4.30**	**81902**	**21.19**	**397120**	**88.51**	**792420**

Figure 6.8: Performance of amoeba joins (sec.) and their result sizes.

	query statement	schedule
Q1	*AJ(book, order)*	*AJ (book, order)*
Q2	*AJ(customer, order)*	*AJ (customer, order)*
Q3	*AJ(order, book, customer)*	*AJ (order, book, customer)*
Q4	*AJ F (order, book, customer)*	*AJ o,c (AJ o,b (order, book), customer)*
Q5	*AJ*(book, order)*	*AJ* (book, order)*
Q6	*AJ*(book, [order, pending, titile])*	*AJ* b,o,p,t (PC(book, book@isbn), order, pending, titile)*
Q7	*AJ* F (book, [order, pending, titile])*	*AJ* b,o,p,t (AJ* b,o (PC(book, book@isbn), order), pending, title)*

Figure 6.9: Queries used in the experiments, and their evaluation schedules.

shows their evaluation schedules using the nested form of amoeba joins. Q1, Q2 and Q3 are the amoeba join operations without using decomposition techniques, while Q4 takes FDs into consideration. Q3 suffers from many instances of structural fluctuations; for the data set of 25M and 50M, we could not complete the query Q3, because this computation caused out of memory errors. On the other hand, Q4 is evaluated efficiently.

Since every order node must always accompany unique book and customer nodes due to the FDs, the number of results should be the same through Q1 to Q4. However, the result size of Q3 is quite huge. Suppose that each customer under the customer_list (illustrated in **Figure 3.1**) has n order nodes, where each order node has a child book node. Since Q3 does not consider the connection from order to book, for each customer node, there are $n \times n$ instances of the amoeba ⟪order, book, customer⟫. This is the reason of the poor performance of Q3. It indicates that the right-hand schedule in **Figure 3.3** cannot be used to compute the relation for order nodes.

Q5, Q6 and Q7 use the amoeba join allowing *null* values. Q6 and Q7 retrieve book@isbn nodes to detect equivalence of book nodes. While Q6 does not consider the FD: order $\rightarrow$ book, Q7 joins order and book nodes with the $AJ*$ operation. The reason Q7 outperforms Q6 is that, in Q6, there are a tremendous number of amoebas that connects book nodes and order nodes through a pending node; these amoebas violate the FD order $\rightarrow$ book.

Considering that enclosing XML nodes with tags is an essential method to enhance the information of XML data, e.g. the example of the pending node or XML data in **Figure 3.2**, to properly decompose an amoeba join according to FDs is quite important to optimize its processing.

Chapter 7

Conclusions

To the best of our knowledge, we are the first to propose the notion of *amoeba*, which makes it possible to smoothly incorporate functional dependencies into XML. Our experimental results of amoeba join and its decomposition techniques confirm the availability of our methods. Our proposed XML index method is efficient in terms of the index size, query performance and its availability in that it can be implemented on top of the standard B+-tree. In this chapter, we explain related work, and discuss several open problems and future work, then finally we conclude this thesis.

7.1 Related Work

7.1.1 Amoeba Join

Querying an XML database without knowledge of path structure was first addressed in [34], and refined in [56]. Both studies used variations of the least common ancestor method (*lca*) to find the smallest tree containing all target nodes. Among the *lca* nodes that connect common node sets (tags or keywords), the one that forms the smallest subtree is defined as the smallest least common ancestor (*slca*) [56]. The precise definition of *slca* is as follows: given k node sets $D_1, D_2, \ldots, D_k$, for example, D_1 and D_2 are node sets matching XPath //book, //order, a node v belongs to the *slca* if $v \in lca(D_1, \ldots, D_k)$ and for all $u \in lca(D_1, \ldots, D_k)$, v is not an ancestor of u. In summary, a subtree rooted from an *slca* node does not contain other *lca* nodes.

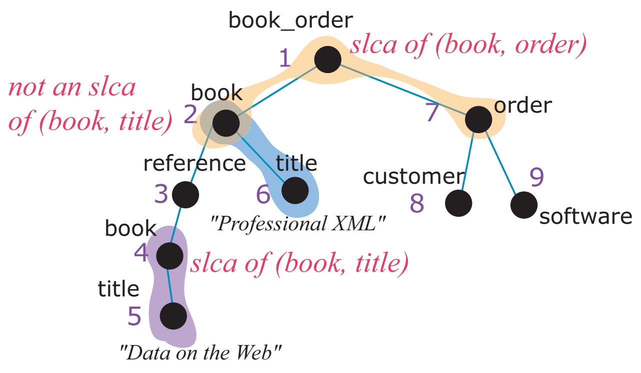


Figure 7.1: The *slca* node of book and order, and the *slca* node of book and title

One problem with this approach is that the *slca* might be the root node of an XML document. XML is a single rooted tree, so every node set can be connected using the root node. **Figure 7.1** shows illustrates this defect. An *slca* of book, order nodes is the root node, book_order. In addition, when the *slca* approach is applied to **Figure 7.1** to find pairs of book and title elements, it misses the node pair (2, 6) because the node 2 contains the node 4, which is an *lca* of book, title node 4, thus node 2 is not an *slca*. In general, XML data semantics are too complex to be detected automatically using simple rules. In addition, although the method of [56] is optimized to search for *slca* nodes, it focuses mainly on keyword versus database queries. It cannot detect element inclusion relationships. For example, it can find the keyword “Data on the Web”, but is not capable of assuring this word is is contained within the parent title tag; it might be connected to the book node 2 in **Figure 7.1**.

XRank [22] applies keyword-based search to XML. It locates XML elements that contains all given keywords. Unlike *slca*, XRank is aware of recursion of XML structure. However, it suffers from two drawbacks: (1) it does not distinguish tag name from textual content; (2) it cannot express complex query semantics [34].

Finding an exact match in XPath queries can be difficult, and thus studies have investigated ways to relax the condition of rigorous matching in regular path expressions [4]. The types of relaxation are explained in [4]. These include dropping or weakening predicates or query nodes, and adding an explicit disjunction, which is similar to querying all structural fluctuations. The proposed amoeba join method contains the essence of query relaxations, but is novel in that it is also able to handle situations in which the high and low nodes of a query tree are reversed.

DogmatiX [55] attempts to solve structural fluctuations using nearest neighbor heuristics that connect nodes within some metrics. However, the method cannot address all possible n^{n-1} structural fluctuations.

Approximate join [21] locates documents with similar structures and different forms. It is more general than amoeba join because it includes changes in tag names. Although approximate join can accommodate various similarity measures, it is optimized to tree edit distances, which must preserve ancestor node order in a query; that is, unlike amoeba join, it cannot reverse ancestor-descendant relationships.

Static typing of XML [12] is another way to handle structural fluctuation. It detects mismatches between paths described in query statements and schemata. Such discrepancies are reported as compile-time (static) errors of the query. This prevents writing invalid queries that do not match any document path. In other words, it is not necessary to cover all path possibilities because the query compiler presents the available paths. A major drawback of this approach, however, is that it requires a schema, which is not mandatory in XML.

7.1.2 Functional Dependencies

Functional dependencies and keys are well studied themes to reduce redundancy of data and to avoid update anomalies [2, 39, 52]. In recent years, these notions have been imported to XML; keys [10, 11] and FDs [5, 32] for XML have been proposed. The XML standard [9] also has a key specification using ID attributes in DTD and IDREFS to refer another nodes from attributes of XML elements. However, these keys cannot be used other than assigning global IDs.

Reducing redundancy and several update anomalies in XML databases are important themes for research. These topics have been studied mainly in RDBMS, and have recently been imported into XML by Arenas and Libkin [5]. They discussed a normalization of an XML document and proposed XNF (XML Normal Form), which is an XML counterpart of Boycodd-Normal Form (BCNF) [39] in relational databases. BCNF has an essential role to detect redundant data in tables and to sophisticate the relational databases design. All of these previous studies tailored to XML use fixed paths to specify FDs, which cannot handle structural fluctuations of XML data.

Jagadish et al. [25] exemplified the needs of multiple hierarchies to describe data with XML. Their proposal is to represent various aspects of the data model with multiple XML trees with different colors. However, the use of colors exceeds the descriptive power of XML. By contrast, equivalent classes of XML nodes presented in this paper can be used without extending the syntax of XML. These two approaches are conceptually the same in that they make connections between XML nodes.

7.1.3 XML Database

In literature, there are various types of XML labeling methods. The 1-index [35] and DataGuide [18] summarize structures of XML documents, by using *bi-similarity* relationship or its variant [1]. Efficient updates for the 1-index is studied in [28]. It solves the problem of cascading updates of the 1-index caused when IDREF edges are added. Although these indexes are useful to glance the entire structure of an XML document, they are not sufficient to detect ancestor-descendant relationships of given two nodes.

In this thesis, we use the interval representation [33] of tree structures of XML. As another approach to label XML nodes, we can use contiguous orders [17] or Dewey [13] orders etc. Note that, however, node insertions or deletions need heavy maintenance of these numbers. Tatarinov et al. [51] provides lazy-renumbering strategies of contiguous node labels, but it still needs periodical maintenance of node labels. The integer intervals are also weak for updates, since blank space for future node insertions will be exhausted.

There are some proposals to make these labels tolerant for node insertions, including VLEI codes [30], ORDPATH [37], P-PbiTree labeling [58] etc. Among them, to implement ORDPATH [37] is relatively easier since it is a variation of the Dewey order, and has no ambiguity in the node labeling strategy. However, to retrieve sibling nodes efficiently, the ORDPATH requires additional index for level values. Our proposed index has a capability to incorporate these labeling strategies as long as we can define the total order on node labels. For example, the combination of ORDPATH, level and inverted path values provides update tolerant and high-performance XML index. The weak point of the OR-

DPATH is that its label length is not scalable for large XML documents. For the detail discussion about the lower or upper bound of the length of XML node labels, see the work by Edith Cohen et al. [16].

Although we have presented our original implementation of XML indexes using the B+-tree, to implement XML databases with existing RDBMS might be possible. However, some points should be taken into consideration. First, almost all relational databases are not *tree-aware* [20]; i.e. query optimizations using properties of XPath axis steps or search space reduction using the combination of tree structures and suffix paths are not available. Second, all of XML node labels are to be stored in a single table; it might lower the concurrency of transactions since every transaction requires the table lock.

In Chapter 6, we observed excessive ancestor queries in (Q1) (**Figure 6.7**) are the bottle neck of the performance. As a solution to this, we can use the stair case join proposed by Grust et al. [20], which saves unnecessary disk scans by computing overlaps of query regions.

ViST [54] index structure utilizes path-prefix of XML and supports a top-down descendant traversal. The benefit of this top-down approach is that it can localize search regions of descendant tags within some subtrees. This combination of subtree regions and tag names is similar to the range query presented in this thesis.

Similar optimization by creating XML specific index structure is researched by Jiang et al. [26]. Neither of them, however, mentions the integration of tag names or path suffixes, which we have proved to be a key factor to improve ancestor or wild-card query performance.

The use of the UB-tree [7] to index XML documents is proposed in [6]. Its coordinates are combinations of text values, document IDs, and paths and their appearance orders generated from DTDs. Integration of text values to the index is an interesting problem that we do not have mentioned in this work. Note that, however, if we integrate text values to the index structure, every update to the text values invokes subsequent maintenance of the indexes. On the other hand, Kaushik et. al. proposed efficient algorithms to process queries containing predicates for text values [29]. Their approach assumes text value indexes are separated from structure indexes, so it is more promising in that it can leverage traditional IR technologies to index text contents of XML.

7.2 Discussion

7.2.1 Managing Scientific Data

In general, data models of the real world require not only tree-structures of XML but also graph structures to represent relationships between data objects. This highlights a problem in XML data modeling: *one hierarchy isn't enough* [25]. To complement tree-structured XML to accommodate graph-structured data model, we need duplicate materialization of the same objects (deep copies), or node references to connect related data (shallow copies).

Two XML documents in **Figure 7.2** and **Figure 7.3** illustrate their differences. These XML data represent cell data retrieved from our Bioinformatics database, named SCMD (*Saccharomyces Cerevisiae* Morphological Database) [47] (**Figure 7.4**), which collects cell images of budding yeast mutants for analyzing gene functions from the shape of cells. These SCMD's XML data contain information of cell shapes and clusters of cells grouped by the cell sizes. **Figure 7.2** uses deep copies to fill the information of each element, while **Figure 7.3** utilizes shallow copies to reduce duplicate elements.

Deep copies of data objects in XML lead to an expansion of data size and update anomalies, while slimmed down XML data with shallow copies using IDREFs [9] or value-based references results in heavy use of value-based joins in queries.

The use of shallow copies is obviously efficient in terms of the storage space, however, when developing SCMD, we frequently used deep copies to report the analysis results of clustering, some statistical methods, etc. It is because writing queries to connect shallow copies was a tedious task, and updates of the existing data is rare, so we duplicated the data many times. However, with the capability of ubiquitous keys, the use of shallow copies is not a serious obstacle to the XML data modeling. We are now working on to develop an efficient query scheduler that takes ubiquitous keys into considerations.

In addition, we need hierarchical key structures, which is frequently used in scientific databases, to manage enormous amount of data. In SCMD (**Figure 7.4**), we maintain millions of cell data, extracted from 273,813 photos (micrographs). Each photo and cell belong to one of 4,782 types of yeast mutants. For such a large database, it seemed to be a hard task to assign a global ID to each cell, since according to the tuning parameters,

```
<scmd>

  <cell_list>
    <cell id="1">
      <size>498</size>
      <cluster id="c1">
        <property>large</property>
      </cluster>
    </cell>

    <cell id="2">
      <size>120</size>
      <cluster id="c2">
        <property>small</property>
      </cluster>
    </cell>

    <cell id="3">
      <size>521</size>
      <cluster id="c1">
        <property>large</property>
        <note>largest</note>
      </cluster>
    </cell>
  </cell_list>


  <cluster id="c1">
    <property>large</property>
    <cell id="1">
      <size>498</size>
    </cell>
    <cell id="3">
      <size>521</size>
      <note>largest</note>
    </cell>
  </cluster>

  <cluster id="c2">
    <property>small</property>

    <cell id="2">
      <size>120</size>
    </cell>
  </cluster>
</scmd>
```

Figure 7.2: Deep XML

```
<scmd>

  <cell_list>
    <cell id="1">
      <size>498</size>
    </cell>

    <cell id="2">
      <size>120</size>
    </cell>

    <cell id="3">
      <size>521</size>
    </cell>
  </cell_list>

  <cluster id="c1">
    <property>large</property>
    <cell id="1"/>
    <cell id="3">
      <note>largest</note>
    </cell>
  </cluster>

  <cluster id="c2">
    <property>small</property>
    <cell id="2"/>
  </cluster>
</scmd>
```

Figure 7.3: Shallow XML

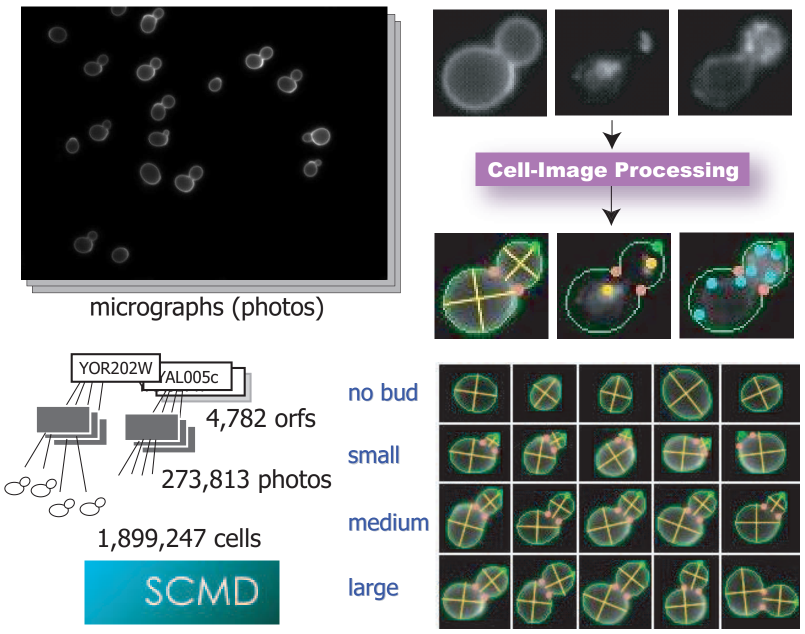


Figure 7.4: Hierarchical key structures of SCMD, and examples of clusters grouped by cell sizes

our image processing program fails to extract some cells or entire cells in a photo. Thus, we used triples of local IDs of cells, photo IDs, and ORFs (the unique identifiers of yeast mutants, e.g. YOR202w, YAL005c,... etc.), as primary keys of cells. **Figure 7.5** shows an example of XML data when integrating these hierarchical key structures.

There are $1 : m$ relationships between nodes, orf:photo and photo:cell; SCMD uses localized cell IDs for each photo, and local photo IDs for each orfs (**Figure 7.4**). Therefore, to identify each cell and its shallow copies in the database, we need the photo and orf information to which the cell belongs.

Functional dependencies and ubiquitous keys are indispensable to describe this types of semantic relationships. For example, we can define FDs and ubiquitous keys for SCMD as follows:

FDs:.

- cell $\rightarrow$ photo : Each cell belongs to a single photo.

```
<scmd>
  <cell_list>
    <orf id="YOR202w">
       <photo id="1">
         <cell id="1">
            <size>498</size>
         </cell>
       </photo>
       <photo id="2">
         <condition date="2006-Jul-01">
           <humidity>98</humidity>
           <cell id="1">
             <size>120</size>
           </cell>
         </condition>
       </photo>
    </orf>

    <orf id="YAR005c">
       <photo id="1">
          <cell id="1">
            <size>521</size>
          </cell>
       </photo>
    </orf>
  </cell_list>

  <orf id=''YOR202w''>
    <cluster id="c1">
      <property>large</property>
      <cell id="1">
         <photo id="1"/>
         <note>largest</note>
      </cell>
    </cluster>

    <cluster id="c2">
      <property>small</property>
      <cell id="1">
         <photo id="2"/>
      </cell>
    </cluster>
  </orf>
</scmd>
```

Figure 7.5: Clustering result of the cells of YOR202w according to their sizes.

- photo → orf : Each photo belongs to a single orf.

(Ubiquitous) Keys:.

- orf, photo, [cell@id] → cell : Given orf and photo nodes, and a cell id, we can identify a single cell.
- orf, [photo@id] → photo : A pair of orf node and photo@id value identifies a photo.
- [orf@id] → orf : An orf@id is a global ID of an orf node.

By allowing structural fluctuations to the XML database that has a hierarchical key structure, we can reduce the burden of data modeling. In the definition of hierarchical keys proposed by Buneman et. al [11], every cell element must obey a fixed structure, e.g. //orf/photo/cell. Our method enables XML structures reorganized as long as they satisfy the given set of FDs. For example, in **Figure 7.5**, we can insert the condition data when the photo was taken (humidity, etc.), and also we can replace the hierarchical order of orf, photo and cell nodes (see around the cluster nodes) to make clarify that these data represents clusters of cells.

7.2.2 Functional Dependencies

Our definition of FDs for XML is novel in that it utilizes the notion of amoeba domains, and thus several problems remain open. First, we need efficient storage for XML data utilizing FDs. In relational databases, keys has an essential role to perform loss-less decomposition of relations [2]. It is beneficial to reduce storage space; it enables internal pages of index structures, such as B-trees, to hold only key values, since other attribute data placed in leaf pages can be associated with keys. What is the counterpart of loss-less decomposition in XML?

An efficient XML index for the amoeba join is also required. Although a relation in XML is dynamically defined, patterns of relations using nodes in $NL(F)$ might not be so many. Thus, it might be possible to avoid evaluating the same patterns of amoeba joins by creating some indexes in advance.

The amoeba join decomposition has several options of its processing order. One of the optimization methods is to remove unnecessary predicates. For example, some set of structural constraints, e.g. $\langle\langle A, B\rangle\rangle$ and $\langle\langle B, C\rangle\rangle$, substitutes for the other predicate such as $\langle\langle A, B, C\rangle\rangle$. In addition, the order selection of amoeba join schedules should be addressed, which might require cost-based estimation of a query plan.

As we shown in the experimental results, some input amoeba domain that has few nodes, for example, [[order@id = "1"]], can be utilized to localize the search region, and makes efficient the amoeba join processing. This types of techniques, called pushing-down predicates, might also be useful.

As for data modeling, we are not sure that FDs can be used on behalf of a schema of XML, because to describe FDs for every nodes in an XML document might be troublesome, and to confirm the correctness of the semantics implied by the FDs might be difficult, too. We need some sophisticated way to specify FDs for XML, in addition, we also have to confirm that updates of XML data do not cause any violation of FDs.

Another problem we have to tackle is *ownership* of XML nodes. **Figure 7.6** illustrates this problem; the R&D (research and development) department has a Biology section; David is a manager of R&D department; Kevin is a manager of the Biology section. In this example, the amoeba domain $\langle\langle$dept, manager$\rangle\rangle$ contains two pairs, (dept, manager = "David"), and (dept, manager="Kevin"); the latter is incorrect result. A simple solution is to define the amoeba domain of manager nodes as D_1 =//dept/manager, D_2 =//section/manager, then distinguish these managers with node labels D_1.manager and D_2.manager. However, this approach fixes the path structure, and consequently we are not allowed to enclose the manager nodes with another tag, e.g. dept/detail/manager, etc. Therefore, we need some method to detect which node has the *ownership* of each manager node. One of the solution to this problem might be to create a scope that hides manager elements under the section node from the dept node when evaluating amoeba joins.

```
<dept name="R&D">
   <manager>David</manager>
   <section name="Biology">
       <manager>Kevin</manager>
   </section>
</dept>
```

Figure 7.6: An example in which ownership of the manager nodes must be considered.

7.2.3 XML Database

One of the reason we pursue the use of B+-tree to index XML data is that we aim to support transactions in our XML DBMS. Transaction management of DBMS involves several essential components of the DBMS [19]; page buffer manager, lock manager, database logging for recovery, and also access methods, such as B+-trees or R-trees. These modules seem to be able to implement independently, however, all of them have a lot of interdependencies. High-performance implementation of transactional indexes include intricate protocols for latching, locking, and logging. The B+-trees in serious DBMSs are riddled with calls to the concurrency and recovery code, and this logic is not generic to all access methods - it is very much customized to specific logic of the access method, and its particular implementation [23]. Therefore, care should be taken to adopt another index structure; it inevitably means throwing out the tons of detailed techniques we have developed to achieve the transaction performance with B+-trees. That is also a reason why the transaction management of R-tree, which is famous as a multidimensional index structure, is not seriously supported in most of the DBMS products, including both of commercial and open-source programs.

7.3 Conclusions

The existence of variously structured XML data, called structural fluctuation, has been a serious obstacle to manage XML databases. This problem arises because there are varieties of XML structures to represent the same meaning data. Nevertheless, by *utilizing* structural fluctuations, we have emphasized that we can enhance the expressive power of XML data. The contributions described in this paper include:

- The notion of amoeba domain to capture structural fluctuations. With this capability, functional dependencies and keys are smoothly incorporated into XML.
- A departure from path-expression queries by using dynamic relations; XML structures of interest are automatically determined from a set of FDs and input node sets.
- Amoeba join operations and its decomposition techniques for optimizing query execution.

- Ubiquitous keys to denote equivalent classes of XML node, which makes easier to manage multiple hierarchies of XML data.
- Several amoeba join processing algorithms. Among them, the quicker algorithm performed well, and it is scalable to the size of an XML document.
- An XML indexing method, which facilitate various types of XML queries, including ancestor, descendant, sibling, suffix path, etc. In addition, our index structure and multidimensional range query algorithm can be implemented on top of the standard B+-tree.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann, San Francisco, CA, 2000.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[3] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, and Jignesh M. Patel. Structural joins: A primitive for efficient XML query pattern matching. In *proc. of ICDE*, 2002.

[4] Sihem Amer-Yahia, Laks V.S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible structure and full-text querying for XML. In *proc. of SIGMOD*, 2004.

[5] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. In *proc. of ACM TODS*, 2004.

[6] Michael G. Bauer, Frank Ramsak, and Rudolf Bayer. Indexing XML as a multidimensional problem. Technical report, Technische Universitat Munchen, 2002. TUM-I0203.

[7] Rudolf Bayer and Volker Markl. The UB-tree: Performance of multidimensional range queries. Technical report, Technische Universitat Munchen, 1998.

[8] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Floresch, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML query language - W3C working draft, November 2003. available at http://www.w3.org/TR/xquery.

[9] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition), October 2000. available at http://www.w3.org/TR/REC-xml.

[10] Peter Buneman, Susan Davidson, Wenfei Fan, Carmen Hara, and Wan-Chiew Tan. Reasoning about keys for XML. In *University of Pennsylvania Technical Report*, 2000.

[11] Peter Buneman, Susan Davidson, Wenfei Fan, Carmen Hara, and Wan-Chiew Tan. Keys for XML. In *proc. of WWW*, 2001.

[12] Don Chamberlin, Denise Draper, Mary Fernandez, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, and Philip Wadler. *XQuery from the Experts*. Addison Wesley, 2004.

[13] Akmal B. Chaudhri, Awais Rashid, and Roberto Zicari. *XML Data Management - Native XML and XML Embeded Database Systems*. Addison Wesley Professional, 2003.

[14] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *proc. of VLDB*, 2002.

[15] James Clark and Steve DeRose. XML path language (XPath) version 1.0, November 1999. available at http://www.w3.org/TR/xpath.

[16] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic XML trees. In *proc. of PODS*, pages 271–281, 2002.

[17] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, INRIA, 1999.

[18] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *proc. of VLDB*, 1997.

[19] Jim Gray and Andreas Reuter. *Transaction Processing - Concepts and Techniques*. Morgan Kaufmann, 1993.

[20] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *VLDB*, 2003.

[21] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate XML joins. In *proc. of SIGMOD*, 2002.

[22] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *proc. of SIGMOD*, 2003.

[23] Joseph M. Hellerstein and Michael Stonebraker. *Readings in Database Systems. Forth Edition*. MIT Press, 2005.

[24] H. V. Jagadish, Laks V. S. Lakshman, Divesh Srivastava, and Keith Thompson. TAX: A tree algebra for XML. In *proc. of DBPL*, 2001.

[25] H. V. Jagadish, Laks V. S. Lakshmanan, Monica Scannapieco, Divesh Srivastava, and Nuwee Wiwatwattana. Colorful XML: One hierarchy isn't enough. In *proc. of SIGMOD*, 2004.

[26] Haifeng Jiang, Hongjun Lu, Wei Wang, , and Beng Chin Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *proc. of ICDE*, 2002.

[27] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *proc. of VLDB*, 2003.

[28] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Pradeep Shenoy. Updates for structure indexes. In *proc. of VLDB*, 2002.

[29] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *ICDE*, page 829, 2004.

[30] Kazuhito Kobayashi, Wenxin Wenxin, Dai Kobayashi, Akitsugu Watanabe, and Haruo Yokota. VLEI code: An efficient labeling method for handling XML documents in an RDB. *ICDE*, pages 386–387, 2005.

[31] Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the Hilbert space-filling curve. *SIGMOD Record*, 30(1), 2001.

[32] Mong Li Lee, Tok Wang Ling, and Wai Lup Low. Designing functional dependencies for XML. In *proc. of 8th International Conference of Extending Database Technology*, 2002.

[33] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *proc. of VLDB*, 2001.

[34] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free XQuery. In *proc. of VLDB*, 2004.

[35] Tova Milo and Dan Suciu. Index structures for path expressions. In *Database Theory - ICDT 99*, volume 1540 of *Lecture Notes in Computer Science*, 1999.

[36] Yoshikazu Ohya, Jun Sese(*), Masashi Yukawa(*), Fumi Sano, Yoichiro Nakatani, Taro L. Saito, Ayaka Saka, Tomoyuki Fukuda, Satoru Ishihara, Satomi Oka, Genjiro Suzuki, Machika Watanabe, Aiko Hirata, Miwaka Ohtani, Hiroshi Sawai, Nicolas Fraysse, Jean-Paul Latge, Jean M. Francois, Markus Aebi, Seiji Tanaka, Sachiko Muramatsu, Hiroyuki Araki, Kintake Sonoike, Satoru Nogami, , and Shinichi Morishita. High-dimensional and large-scale phenotyping of yeast mutants. *PNAS*, vol. 32, December 2005. (* equally contributed authors). Available from http://www.pnas.org/cgi/content/abstract/0509436102v1.

[37] Patrick O'Neil, Elizabeth O'Neil, Shankar pal, Istvan Cseri, and Cideon Schaller. ORDPATHs: Insert-friendly XML node labels. In *proc. of SIGMOD*, 2004.

[38] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *proc. of PODS*, 1984.

[39] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[40] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the UB-tree into a database system kernel. In *proc. of VLDB*, 2000.

[41] RelaxNG. http://relaxng.org.

[42] H. Sagan. *Space-Filling Curves*. Springer-Verlag New York, Inc, 1994.

[43] Taro L. Saito and Shinichi Morishita. Transaction management for XML (in japanese). In *proc. of Joint Symposium on Parallel Procesing (JSPP)*, pages 103–110, 2002.

[44] Taro L. Saito and Shinichi Morishita. Xerial: An update tolerant and high concurrent XML database. In *proc. of Fourth Data Mining Workshop of Japan Society for Software Science and Technologies (JSSST)*, pages 13–20, 2004. Available from http://www.xerial.org/pub/paper/2004.

[45] Taro L. Saito and Shinichi Morishita. Amoeba join: Overcoming structural fluctuations of XML data. In *proc. of Ninth International Workshop on the Web and Databases (WebDB)*, pages 45–50, 2006.

[46] Taro L. Saito and Shinichi Morishita. Efficient integration of structure indexes. In *proc. of 12th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2007. (to be appeared).

[47] Taro L. Saito, Miwaka Ohtani, Hiroshi Sawai, Fumi Sano, Ayaka Saka, Daisuke Watanabe, Masashi Yukawa, and Yoshikazu Ohya. SCMD: *Saccharomyces Cerevisiae* morphological database. *Nucleic Acids Research (NAR)*, vol. 32:D319–D322, 2004. http://scmd.gi.k.u-tokyo.ac.jp/.

[48] Taro L. Saito(*), Jun Sese(*), Yoichiro Nakatani, Fumi Sanoand Masashi Yukawa, Yoshikazu Ohya, and Shinichi Morishita. Data mining tools for the *saccharomyces cerevisiae* morphological database. *Nucleic Acids Research (NAR)*, vol. 33:W753–W757, 2005. (* equally contributed authors).

[49] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana manolesch, and Ralph Busse. XMark: A benchmark for XML data management. In *proc. of VLDB*, 2002.

[50] Sleepycat Software. BerkeleyDB. available at http://www.sleepycat.com/.

[51] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *proc. of SIGMOD*, 2002.

[52] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems. VOLUME I.* W H Freeman & Co (Sd), 1988.

[53] XML schema part 0, 1 and 2. available at http://www.w3.org/XML/Schema.

[54] H. Wang, S. Park, W. Fan, and P. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *proc. of SIGMOD*, 2003.

[55] Melanie Weis and Felix Naumann. DogmatiX tracks down duplicates in XML. In *proc. of SIGMOD*, 2005.

[56] Yu Xu and Yannis Papaconstantinou. Efficient keyword search for smallest LCAs in XML databases. In *proc. of SIGMOD*, 2005.

[57] Susan B. Davidson Yi Chen and Yifeng Zheng. BLAS: An efficient XPath processing system. In *proc. of SIGMOD*, 2004.

[58] Jeffrey Xu Yu, Daofeng Luo, Xiaofeng Meng, and Honghun Lu. Dynamically updating XML data: Numbering scheme revisited. In *proc. of WWW*, 2005.