Knowledge and Information Systems (2002) 4: 168–201

Knowledge and Information Systems

XSD: A Hierarchical Access Method for Indexing XML Schemata

Evangelos Kotsakis

Joint Research Center, Space Applications Institute, Ispra, Italy

Abstract. Search operations and browsing facilities over an XML document database require special support at the physical level. Typical search operations involve path queries. This paper proposes a hierarchical access method to support such operations and to facilitate browsing. It advocates the idea of searching large XML collections by administering efficiently XML schemata. The proposed approach may be used for indexing XML documents according to their structural proximity. This is obtained by organizing the schemata of a large XML document collection in a hierarchical way by merging structurally close schemata. The proposed structure, which is called *XML Schema Directory* (XSD), is a balanced tree and it may serve two purposes: (1) to accelerate XML query processing and (2) to facilitate browsing.

Keywords: Directory/indexing structures; Hierarchical access methods; Semi-structured data; Web data processing; XML schema directory; XML sources/documents

1. Introduction

In the last few years, there has been an increasing interest in managing semi-structured data (Abiteboul et al., 1997; Buneman, 1997; Buneman et al., 1999; Papakonstantinou and Velikhov, 1999). Techniques for extracting structured information from semi-structured data have been for long the main theme of research endeavors (Atzeni et al., 1997; Adelberg, 1998; Nestorov et al., 1998; Bertino et al., 1999). The recent emergence of the *eXtensible Markup Language* (XML) (Bray et al., 1998) as a standard for data representation and exchange on the World Wide Web has attracted the interest of many researchers who observed a resemblance between semi-structured models and XML. Literally, XML data sources (documents) may be viewed as entities whose structure is not fixed and

Received 15 Mar 2001
Revised 12 Apr 2001
Accepted 11 May 2001

regular and both the data described and the structure itself are blurred. It is expected that much of the data encoded in XML will be semi-structured and it will be irregular or incomplete and its structure will change rapidly and unpredictably.

With an increasing interest in XML data processing, the database community has recently devoted considerable attention to XML data management (Widom, 1999; Ceri et al., 2000). The main motivation originated in the area of electronic data exchange and electronic commerce (Meltzer and Glushko, 1998; Glushko et al., 1999; Blair and Boyer, 1999; Mylopoulos, 2000). Migrating web information to XML is a significant step in turning the web into a database. XML databases are deemed as repositories containing XML documents with or without explicit structure (schema).

The motivation for schema-based retrieval is to facilitate the manipulation of XML data sources by administering the schemata, which the XML data sources have been derived from. The challenge in developing XML database systems lies in providing a special-purpose data structure that accelerates query processing and facilitates browsing. Important issues in this context include the handling of XML data representation as well as the choice of an efficient access method.

This article introduces a hierarchical access method to support search operations in XML document databases as well as to facilitate browsing. In our approach, XML documents are indexed according to their structural proximity. Each document added to the XML database system is classified according to its structure (schema) and it is stored next to the most relevant documents. Consequently, structurally close documents are stored near to each other. The *XML Schema Directory* (XSD) structure does not require any transformation of XML documents into an internal form and it uses a document schema to distinguish query paths relevant to documents. The main objective of XSD is to provide an efficient way for identifying those XML documents, in an XML corpus, that include the paths of a given XML query. Consequently, it is proposed as a document access method aiming at retrieving all the XML documents which are relevant to a path query. Moreover, XSD may be used to support content-based search on large collections of XML documents. The idea of aggregating similar XML schemata into a merger schema allows faster query processing. The merger schema represents a general class that contains all those XML documents of the original schemata. As long as a schema becomes part of this merger, all XML documents derived from it are considered instances of the merger schema.

The access granules in XSD are XML documents rather than XML elements found in such documents. Further processing may be needed in case one is interested in retrieving parts of relevant XML documents. XML documents are retrieved according to whether the paths that constitute the XML query are embedded into them. Structural proximity between documents is taken into account to group structurally close documents in a hierarchical way, which tends to be advantageous in searching and browsing large collections of XML documents. In summary, this paper makes the following contributions:

- It proposes a novel access method, which is called XML Schema Directory (XSD) for XML data sources.
- In order to support XSD features, a novel merging operation between XML schemata is introduced.
- It provides all the necessary support operations and algorithms for searching and maintaining the XSD structure and it explains how XSD can be used in a filter/refine model for XML query processing.

The rest of this paper is organized as follows. Related work is presented in Section 2. Section 3 discusses XML schemata and XML query evaluation by introducing the concept of Document Compound Structure (DCS) to portray the structural composition of XML documents. Some general requirements for an efficient access method are also discussed. Section 4 introduces the basic operations between DCSs. Similarity measures between DCSs are also discussed and defined in terms of editing operations. Section 5 presents the XSD tree structure and discusses how an XML query may be evaluated by using the XSD tree. XSD maintenance operations and all the necessary supporting algorithms to perform these operations are also presented in this section and their performance is discussed. Conclusions are discussed in Section 6.

2. Related Work

A collection of XML documents can be seen as a collection of objects. Finding relevant objects in a given set has been for many years the main research effort of the information retrieval community (Grossman and Frieder, 1998; Baeza-Yates and Ribeiro-Neto, 1999). In a traditional text information retrieval system, each document is segmented into significant terms (words) and indexing structures are generated to indicate what term occurred in what document as well as term frequency, term weight and possibly position data. A user query, in such a system, consists of a set of terms and it may be literally viewed as a document. The information retrieval system retrieves those documents that are considered close to the query. Certain similarity or dissimilarity (distance) measures have been invented to estimate proximity between a query and a document (Wang et al., 1999).

Although numerous techniques exist to identify relevant documents, their effectiveness is not adequate and consequently they are not appropriate for use in an XML repository. XML documents cannot be sufficiently represented as multidimensional patterns (feature vectors), since their elements cannot construct a common feature space with constant dimensionality. The existence of such a feature space is difficult because XML documents are irregular; new tags can appear, which are not included in the existing feature space. Even if we extend the feature space by adding new dimensions in order to accommodate new features there are still three basic problems that complicate the situation: (a) it is likely the dimensionality becomes very large and the complexity will increase; (b) computations of distances between patterns may be non-continuous, since some features may not be comparable; (c) XML tags may be infinitely nested and consequently it would be difficult to develop features that capture containment relationships in an arbitrary depth. For instance, in traditional information retrieval systems, queries are limited to simple keyword-based expressions and documents are seen as streams of words. In an XML repository, both queries and documents are arbitrarily structured and consequently there is a need to develop new techniques that take into account the structural nature of the documents.

Collections of semi-structured sources have been proposed as the basis for improving query handling and indexing in Bertino et al. (1999). However, this approach is directed to classifying semi-structured sources by using generic classes (*structural expressions*). That is, a set of classes should be supplied for classifying the semi-structured objects. In practice, it might be difficult to specify such generic classes that capture abstract types whose realization could be found in

semi-structured sources. Constructing collections of semi-structured objects is also presented in Nestorov et al. (1998) through *approximate* typing. Approximate typing assumes that each single object is of a unique type and then elimination of types is accomplished by checking for equivalence among the initial types. In Bertino et al. (1999) and Nestorov et al. (1998) the resulting organizations are *approximate* types that are aimed at describing semi-structured objects.

Previous approaches in designing XML database systems use specialized models for storing XML data sources. The XML documents are transformed into internal structures according to the model, and future data handling is accomplished by accessing the internal structures. For example, the Lore system (McHugh et al., 1997), which is one of the pioneer database systems used to manage semi-structured XML and in general weakly defined data, is based on the OEM model (Papakonstantinou et al., 1995). Lore accomplishes the uploading of new documents by adding the elements of the documents in a tree-like structure and updating several indexes (value index, text index, link index and path index McHugh et al., 1998) during the insertion phase. From that point on, any data access is performed by considering the whole database as a huge tree that contains XML elements. After uploading a document, its contents are added to the tree and any path query returns those XML elements for which there is a match in the tree. The Lore approach seems to view an XML document as a database ('load file') and a set of documents as a single large database where all documents are blended together into a tree-like structure. In our approach a set of XML documents is handled as is (set of documents) and XSD is used to retrieve the whole content of relevant documents. In fact, the Lore approach is more database oriented, while XSD is document based. The retrieved objects in XSD are XML documents.

The XSD is a schema organization, which organizes several XML schemata, with the form of DCSs, into a tree structure. It is different from that in Adelberg (1998), Bertino et al. (1999), Goldman and Widom (1997), and Nestorov et al. (1998) at the level of organizing information; it is a construction that classifies XML schemata. However, the methods in Adelberg (1998), Bertino et al. (1999), Goldman and Widom (1997), and Nestorov et al. (1998) may be used to extract primitive schemata and then XSD may be applied to organize these schemata hierarchically. Therefore, under this perspective, XSD may act as a complement to the proposed techniques aiming to optimize queries that are targeted on large XML collections.

3. Organization of XML Data

3.1. XML Document Collection

Data on the web may be traditionally stored either in HTML files or in a traditional DBMS. In the former case, searching is feasible by employing a keyword-based search technique, while in the latter case the only way to extract useful information is to use parameterized queries. Both approaches are limited because the full text search is not accurate and the query parameterization is strongly associated with the underlying database schema. An immediate benefit from the use of XML would be the lack of necessity for developing rigid form-based interfaces. There are many reasons to use XML as an enabling data format and many potential applications have been suggested in Bosak (1997). Such applications include those that require the web client to mediate among

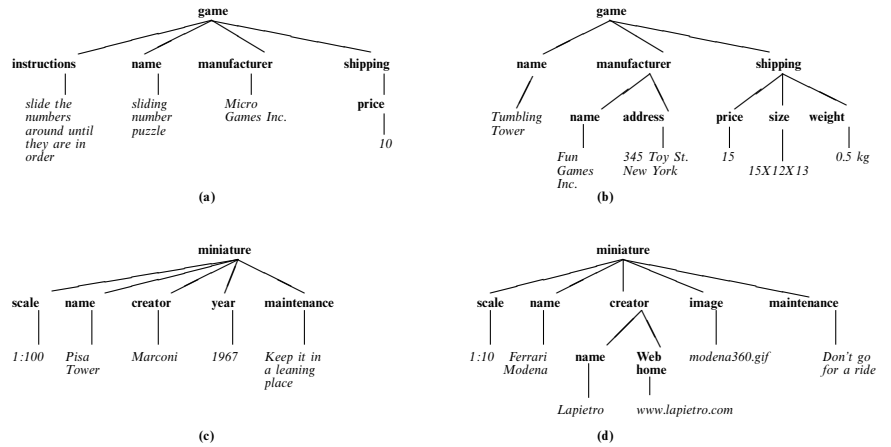

Fig. 1. Four XML documents: (a) and (b) are structurally close to each other, as are (c) and (d). It is desirable in this case to form two groups of documents: one containing (a) and (b) and the other containing (c) and (d).

heterogeneous databases and provide different views of the same data: those that use web clients to distribute significant proportions of the processing load as well as those that utilize intelligent web agents for information discovery. In all of these applications, XML advocates the philosophy that data belongs to its creators and data is exchanged without defining any binding between them and an authoring tool or scripting language. Therefore, XML is considered as a machine-readable format, which allows formatted data to be combined with other data and transformed into any other format for computation or display. Initially the data may be stored in some private repository by the creator. Such a repository may utilize existing database management systems. A web client may collect XML-formatted data from different repositories and store them as unique XML resources. In this case the client collection is generally viewed as a collection of XML documents. There are already a substantial number of information sources, including web sites, that publish information as XML documents and this number is growing rapidly.

For example, a toy manufacturer may maintain a catalogue of its entire collection of toys and provide an XML description for each toy. An on-line store may collect the XML-formatted descriptions of toys from many different manufacturers and maintain its own collection of XML sources regarding the toys. The question is, how should the on-line store maintain this collection of XML sources in a way that accelerates browsing and searching? The problem becomes more difficult if we consider that different toy manufacturers possibly use different description types for the same toy. Is there any possible way to group such XML documents so that similar toys are placed in the same group? This would be very advantageous in the case of someone wanting to browse the on-line store. The notion of similarity introduced here is based on the examination of the structural composition of the XML documents with a view to discovering the resemblance or differences.

Figure 1 shows four XML documents that might be used to describe toys in an on-line store. In order to facilitate browsing and searching, it is desirable to store similar documents together. As we can see, the document structures of

(a) and (b) are close to each other. Thus it would be useful to put (a) and (b) in the same class of documents. If someone is looking for game manufacturers, (a) and (b) documents will be candidates for further searching, while (c) and (d) will not. Therefore, the purpose of a method that supports efficient browsing and searching should be to efficiently identify the group of XML documents that are relevant to a given query. The problem of identifying relevant XML documents is slightly different from the one found in full text document information retrieval techniques. Relevance, in the context of information retrieval, inevitably involves the human factor in order to determine whether the result set of documents is according to the user expectations. This is because relevance, in this context, is based on subjective criteria that have to do with how the data in the documents are interpreted and how the desired information is derived from the documents. That is, subjective criteria defined by the user are mainly used to interpret the semantics of the terms found in the documents for a given context. In XML documents, relevance could be defined according to path inclusion. A path query is relevant to a document if its paths are included in the document. The rationale beside this definition is that tags on data elements identify the meaning of the data. Therefore, no additional criteria are needed for semantic interpretation, since each element comes together with its meaning in a given context.

3.2. XML Data and Document Compound Structure (DCS)

An XML data source is a composite structure that may consist of a single atomic element or several hundreds of such elements arbitrarily nested in several layers. An XML document may be interpreted *literally* or *semantically* (Goldman et al., 1999). In the semantic mode, the XML document is represented as a graph that includes semantic relationships between XML elements. This is supported by the current XML version (Bray et al., 1998) by assigning special meaning to some attributes. Attributes like ID and IDREF can be used to define relationships between XML elements. In the literal mode, an XML document is represented as a tree. There are no attributes with semantic interpretations. In this paper, XML documents are viewed literally and all kinds of attributes are visible as textual strings. Well-formed XML is also assumed, which places no restrictions on tags, attribute names or nesting patterns.

XML structural constraints may be expressed using a *Document Type Definition* (DTD). A DTD may define a class of XML documents using a context-free grammar with several restrictions. A DTD actually specifies what elements may occur and how the elements may nest in an XML document that conforms to the DTD. It serves two purposes: (1) it describes the characteristics of XML elements and (2) it declares constraints on the use of mark-up. Figure 2(a) shows a DTD specification to constraint XML documents such as those in Fig. 3(a).

While DTDs are adequate for some applications they may be insufficient for applications that impose constraints on the type of referenced elements and require the value of some elements to be within a certain range. The limitations of DTDs are summarized as follows: (1) they do not allow atomic types (except for #PCDATA, which means string) and (2) there is no way to define range specifications and type constraints. The XML schema proposal (Fallside, 2000; Thompson et al, 2000; Biron and Malhotra, 2000) defines facilities that address such needs. Figure 2(b) shows an XML schema whose instances are documents such as those in Fig. 3(a).

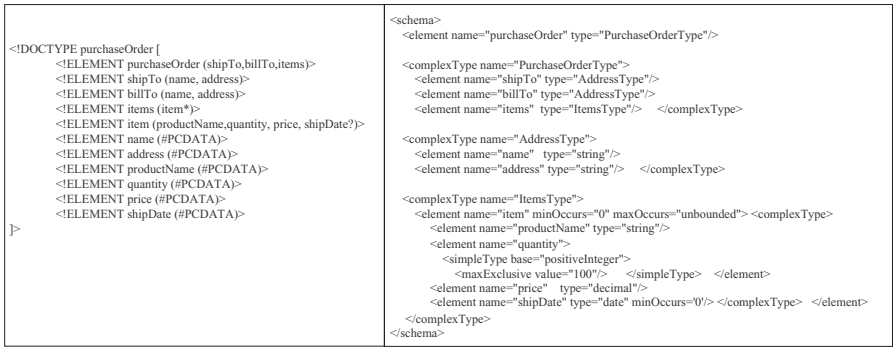

Fig. 2. (a) A Document Type Definition (DTD). (b) An equivalent XML schema.

When neither a DTD is supplied nor an XML schema, the XML documents are self-describing and they may be viewed as semi-structured sources. That is, the XML document is combined from data whose structure is not regular and its schema is contained within the data. In this case, a dynamic schema such as a DataGuide (Goldman and Widom, 1997) may be used to describe the XML document structure. DataGuides are dynamic schemata generated from semi-structured data sources describing every unique label path of the source once. Figure 3(b) shows the DataGuide that may be constructed from the document in Fig. 3(a).

Observing the structural composition of DTDs, XML schemata and DataGuides, we notice a remarkable similarity between these constructs. For instance, apart from the ability to specify type constraints, XML schemata are structurally composed in the same way as DTDs. There is also a close relationship between DataGuides and DTDs. The only difference between a DTD and a DataGuide is that a DTD is a structural composite that restricts allowable XML data, while a DataGuide infers the document structure directly from the XML documents.

For the sake of clarity, we introduce a rooted graph-based structural description, called *Document Compound Structure* (DCS), to portray the structural composition of XML documents. A DCS is simply a rooted directed graph that specifies nested relationships between XML mark-ups.

Definition 3.1. A *Document Compound Structure* (DCS) is a directed graph such that it contains a node designated as the root, from which there is a path to every other node. Each node in the graph carries its own label, which is a literal of an XML element mark-up. A DCS node is also known as an element. A child–parent pair of nodes in the DCS preserves the child–parent relationship between two mark-ups in an XML document. Paths are allowed to appear at most once in the graph.

Although an XML document may be generally represented as a tree, its structure may be a graph. A DCS explicitly captures the relationship between child–parent pairs of XML mark-ups and it may be deemed as a diagrammatic representation or a skeleton of an XML document. Any of the above-mentioned structural descriptions might be seen as a DCS specialized specification. A DTD may be

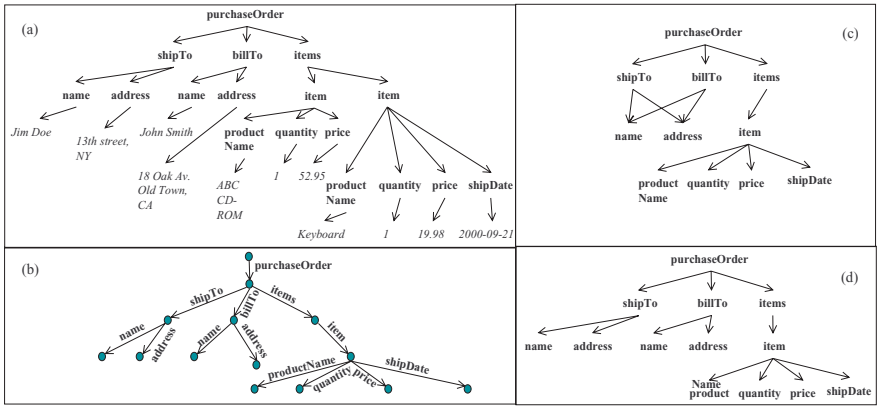

Fig. 3. (a) An XML document. (b) A DataGuide for this document. (c) A Document Compound Structure (DCS) for this XML document and (d) the unfolded DCS for this document. Notice that the DCSs in (c) and (d) can be easily derived from the DTD in Fig. 2(a) or the XML schema in Fig. 2(b).

seen as a DCS, which additionally may constrain the usability of some XML mark-ups. An XML schema may be seen as a DCS that has been enriched with type constraints and a DataGuide may be seen as a DCS supporting target sets (sets of objects, which are reachable via the DCS paths). In general, a DCS may be viewed as the common ground (or the conceptual intersection) between DTDs, XML schemata and DataGuides. The nodes of a DCS are the labels (mark-up elements) found in XML documents and an edge from label a to b represents a parent/child relationship between a and b. A label path starts from the root and terminates at the label. Each label path appears at most once. Figure 3(c) shows a DCS, which may be obtained from the DTD in Fig. 2(a), the XML schema in Fig. 2(b), or the DataGuide in Fig. 3(b).

We distinguish between two types of DCS: *primitive* DCS and *merger* DCS. A primitive DCS is derived directly from a DTD, XML schema or a DataGuide. A merger DCS is a structure obtained by merging one or more DCSs (primitive or merger). DCS merging is obtained by using a similarity criterion based on editing operations and it is discussed in detail in a subsequent section. In general, merger DCSs may be viewed as generic composite structures containing simpler DCSs. Figure 4(a) shows a merger DCS, which is obtained by merging the DCSs in Fig. 4(b) and (c). As we can observe in Fig. 4, the merger DCS contains the union of the elements in the simpler DCSs. In a sense, a merger DCS may be viewed as a bounding structure that unifies simpler DCSs.

Definition 3.2. An XML document is said to be an *instance* of a DCS if its structure is embedded in the DCS root-directed graph. This means that the XML schema (DTD or DataGuide), from which the XML document has been derived, will be completely enclosed in the DCS. In case the XML schema is a tree, then this tree is said to be embedded in the DCS if it is completely embedded in the unfolded root-directed graph of DCS. The unfolded root-directed graph is a tree which has the same root as the initial graph. It may be obtained from the initial

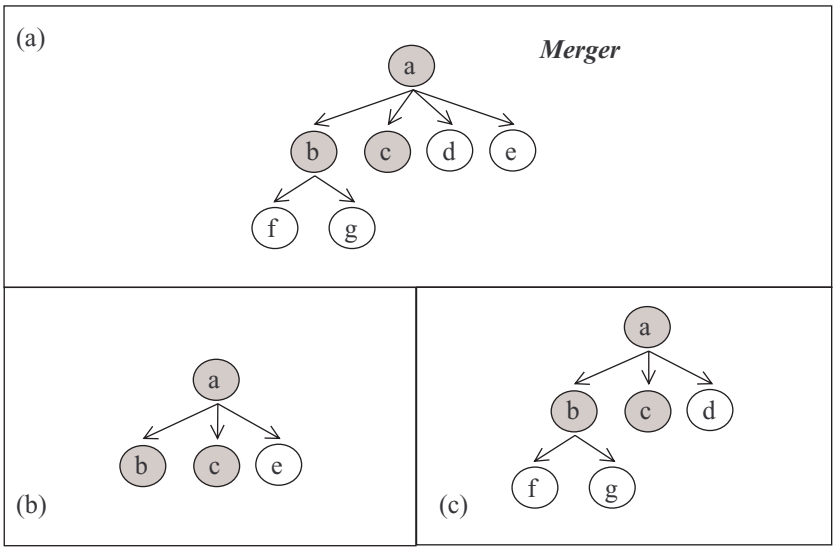

Fig. 4. DCS merging.

graph by unfolding recursively all the subgraphs rooted at the children of the initial graph root. Figure 3(d) shows the unfolded DCS, which may be obtained from the DCS in Fig. 3(c).

In order to facilitate the discussion and without loss of generality, we represent the DCSs as trees. In a large XML corpus, there may be many primitive DCSs, each one depicting a different schema. Figure 5 shows such primitive DCSs as structure composites of XML document groups. Each such DCS represents a class of documents that have been derived from it. One way to organize XML documents according to their structural similarity is to elaborate methods that administer DCSs. An XSD access method is introduced to address such needs. It focuses on organizing efficiently DCSs in a hierarchical manner by examining the structural similarity between DCSs.

As information is stored in XML, the ability to intelligently query XML data sources becomes increasingly important. Toward this objective, several XML query languages have been proposed (Deutsch et al., 1999; Bonifati and Ceri, 2000). The realization of a query system is mainly accomplished through a query engine, which accepts XML queries, executes them on a predefined set of XML sources and finally returns the result set.

The main role of an XML query language is to allow the formulation of queries and determine the result set of the XML elements that should be returned. Even though all of the proposed query languages address the problem of query formulation, they assume that the input to such a query is a set of known documents or nodes within multiple documents. In other words, to execute an XML query, the query engine should be supplied with (1) the query string and (2) the URL data sources on which the query will perform.

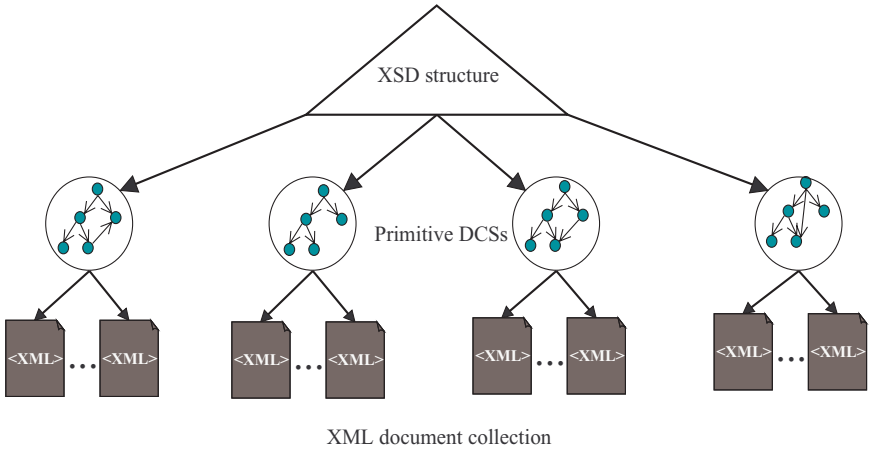

Fig. 5. Document Compound Structures (DCSs) in an XML document collection.

3.3. Querying XML Data and Access Method Requirements

Executing an XML query over a large corpus, which may consist of tens of thousands of XML documents with diverse schemata, may slow down the query-answering process. It is then obvious that, if the input documents are not known in advance, an access structure is required to alleviate the answering process by restricting the search space to a few documents which actually contain the desired data. Then these documents may be used as the input to the XML query.

Limiting the search space of a query means finding the XML documents which are relevant to the query. This implies dividing the initial search space of the query into two disjoint sets: one containing those documents which are relevant to the query and the other one containing those documents which are not. Therefore, the objective of an XML access structure must be twofold: (1) to partition the initial set of documents into relevant and not relevant and (2) to supply the XML query engine with only those XML documents that are relevant to the query.

Figure 6 shows a typical way of processing XML path queries by utilizing the XSD access method. An XML query may consist of path expressions and conditions. The query execution is divided into two steps: the *filter* step and the *refinement* step. The XSD structure is used in the filter step to produce a set of relevant XML documents, which is usually a fraction of the whole collection. For each document obtained during the filter step we check the query conditions. The query results include those XML elements in the relevant set of XML documents that satisfy the query conditions. In the refinement step we retrieve the exact elements in the relevant set of documents that satisfy the path conditions described in the query. If the predicates for these paths evaluate to true, the corresponding elements are added to the query result.

The question is which structure an access method must support in order to filter out the relevant documents to a given XML path query. The complexity and nesting capability of XML data pose certain problems in designing a satisfactory access method. Some of these problems are caused by inherent shortcomings of

the XML data. First, there is no straightforward way to store a collection of XML documents in a relational database with a fixed schema and tuple size. This is because of the irregularity or incompleteness of the data and the potential of the data structure to be unpredictably changed. The dynamic nature of XML data requires such data structures that support this dynamic behavior without deteriorating over time. Second, there is no standard algebra defined on XML data. This means that there is no standardized set of basic operators. The set of operators heavily depends on the given application that processes the XML data. An algebra for handling nested queries and general path expressions could be used to support XML query evaluation and can furnish a concise representation of query execution plans and optimization. If such a set of standard operators is specified, then more complex operators can be expressed by using the standard ones, and a basic operator optimization would result in an overall performance improvement.

Although some basic operators like *selection*, *regular-expression mapping* and *join* have been introduced (Beeri and Tzaban, 1999; Christophides et al., 2000), they are, in general, more expensive than the standard well-known relational operators. A retrieval query on an XML database requires fast execution of the search operations involving paths in several XML documents. To support such search operations, one needs a special access method. The main problem in the design of such a method is that it is difficult, if not impossible, to define total order among XML documents. Such an order could be used to preserve proximity. That is, there is no way to define a total order in a set of XML documents since we cannot define a mapping from the set of documents to a sorted sequence such that any two XML documents with similar schemata are close to each other in the sorted sequence. This makes the design of an access method in an XML database more difficult than in traditional databases. Although XML documents are ordered data sets of elements, a collection of XML documents of diverse schemata is not ordered. A data model for semi-structured data, named OEM, has been proposed and widely accepted (Papakonstantinou et al., 1995; Abiteboul et al., 1997) and recently adapted for modeling XML data (Goldman et al., 1999). OEM views such a collection of XML documents as an edge-labeled directed graph with a distinguished single root. As such, one may imply a total order in this model. However, this order is randomly defined; it is not based on schema similarity and it cannot be used to preserve document proximity.

Therefore, the requirements of desired XML access methods can be summarized as follows:

1. *Dynamics*: XML documents are inserted and deleted from the corpus and an access method should continuously keep track of the changes.
2. *Delete* and *Insert* operations should not be very expensive (do not require reorganization of the support structure of the access method).
3. *Scalability*: Access method should adapt to database growth.
4. *Efficiency*: Search should be fast and the method should require as little space as possible.
5. The support structure should preserve proximity based on schema similarity.

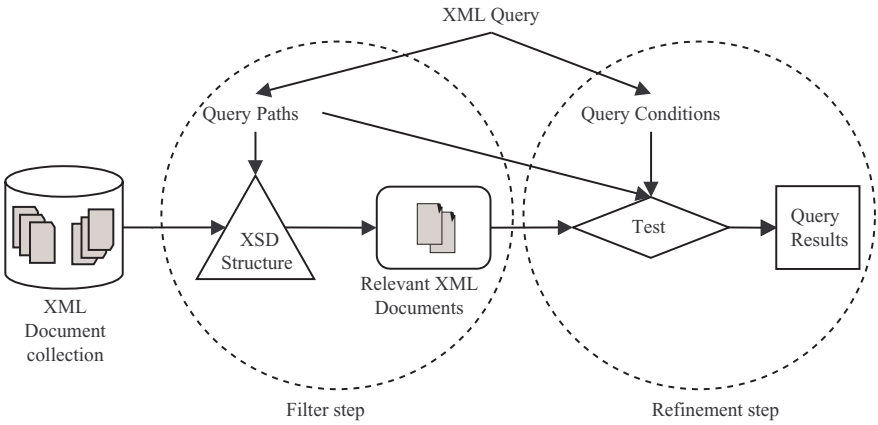

Fig. 6. Retrieving relevant XML documents in a filter/refinement model for path query processing.

3.4. XML Query Matching and Path Expressions

As mentioned previously, there exists neither an XML (or semi-structured) algebra, nor a standard XML query language. Some form of SQL-like syntax with regular expression enhancements often expresses XML queries. XML queries are based on pattern matching on the syntax of the XML data format and they express the ability to reach to arbitrary depth in the XML data tree. XML queries may consist of one or more path expressions because there is often a need to express multiple search conditions. All of these path expressions may be matched against the XML documents of the repository. Wildcards are often used because the schema of XML sources is not known in advance or it may change often. A simple path expression is a sequence of labels separated by a delimiter. In this paper, we use the slash character ('/') as the delimiter for separating labels. In order to facilitate the subsequent discussion about XSD structure, we introduce some definitions.

Definition 3.3. A *label* is a literal that describes a mark-up. A *label instance* is an instance of this mark-up in a DCS. A label instance is associated with the location of the label in a DCS. For example, in Fig. 3(d), 'name' is a label with two instances in the DCS (one being the child of the 'shipTo' label and the another one being a child of the 'billTo' label). If a DCS is a graph, the number of instances of a label is equal to the number of paths from the root to the label.

Definition 3.4. Let L be the set of all labels, then an expression '$l_1/l_2/,\ldots,/l_n$' is a *simple path expression* of length n with $l_i \in L$ and $1 \leqslant i \leqslant n$.

Definition 3.5. A *parameterized path expression* is a path expression which is specified by a regular expression.

There are two levels at which one can apply regular expressions on a path: (1) at label level and (2) at the alphabet level. Wildcard patterns may be used to represent arbitrary repetition of regular expressions. A parameterized path expression is actually a pattern, which may be matched against simple path expressions.

Definition 3.6. A DCS matches a parameterized path expression if there is at least one simple path in the DCS which matches the parameterized path expression. Algorithm 3.1 shows how to find whether a DCS tree matches a parameterized path expression (*ppe*). If a matching path is returned by Algorithm 3.1, then the parameterized path expression is matched against the DCS.

Definition 3.7. Let D be the set of all primitive DCSs of a document collection. D matches a parameterized path expression if there exists at least one primitive DCS in D that matches the parameterized path expression.

D may be used to check whether a given path exists in any of the XML documents in the collection. DCSs are mainly used to efficiently find simple paths that match a given parameterized path expression.

Definition 3.8. An XML query is *relevant* to a DCS if all of the path expressions of the XML query are matched against the DCS. In other words, this means that if a subtree of DCS matches the XML query pattern, then the query is relevant to this DCS. Algorithm 3.2 returns TRUE if an XML query is relevant to a given DCS.

Definition 3.9. An XML query is *relevant* to an XML document if the query is relevant to the primitive DCS from which the XML document has been derived.

When an XML query is relevant to a primitive DCS, all the XML documents that have been derived from this DCS are relevant to the query.

Algorithm 3.1. *FindPath(T, ppe)*

Input: A DCS tree T and a string of characters *ppe* describing the parameterized path expression.

Output: A set of paths M in T that match *ppe*.

Method: In general, *ppe* is a regular expression. In order to perform the checking, an automaton A, which is equivalent to the regular expression, is constructed; any non-deterministic automaton suffices (Hopcroft and Ullman, 1979). The method computes a set C (closure) as described in steps F1 and F2 and then when the set C converges a path (if exists) is constructed from those elements of C that contain the terminal state of A.

F1 *Initialization*: Let S be the set of states of the automaton A (i.e., $S = \{s_0, s_1, \ldots\}$) and N be the set of the nodes in DCS (i.e. $N = \{n_0, n_1, \ldots\}$). C is initially empty. Let $l(n)$ be the label of the node $n \in N$. Let s_0 be the initial state of A and n_0 be the root of T. If there is a transition $s_0 \xrightarrow{l(n_0)} s_i$ in A with $s_i \in S$, then add (n_0, s_i) to C. Repeat step F2 until C does not change any more.

F2 Choose a pair $(n, s) \in C$ and consider those children of n (say n') for which there exists a state $s' \in S$ such that the transition $s \xrightarrow{l(n')} s'$ is in A and then add (n', s') to C. Stop when C reaches a fix-point (C does not change any more).

F3 Let n be a node in T for which there exists a pair $(n, s) \in C$ with s being the final state of the automaton A. The path from node n toward the root of T is a path that matches *ppe*. For all such nodes n, construct the path by ascending T from n toward the root n_0 and add the path to M.

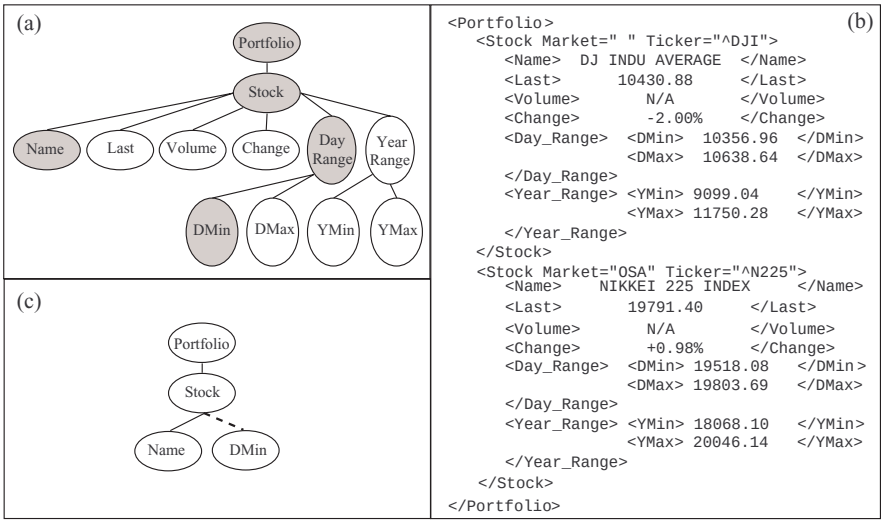

Fig. 7. Example of querying XML data. (a) Document Compound Structure (DCS). (b) XML document derived from this DCS. (c) A relevant XML query.

Algorithm 3.2. *IsRelevant(T, Q)*

Input: A DCS tree T and an XML query Q. The query Q is a set of parameterized path expressions.

Output: It returns TRUE if Q is relevant to T, otherwise it returns FALSE.

Method: For each parameterized path expression p of the query, it checks whether p is matched against T. If T matches all of the path expressions of the query Q then it returns TRUE, otherwise FALSE.

IR1 Let $Q = \{p_1, p_2, \ldots, p_n\}$, where p_i $(1 \leqslant i \leqslant n)$ is a parameterized path expression. Let M_i be the set of paths in T such that $M_i = FindPath(T, p_i)$.

IR2 For each $p_i \in Q$ with $1 \leqslant i \leqslant n$ **do** $M_i = FindPath(T, p_i)$.
If $\forall M_i$ $(1 \leqslant i \leqslant n)$, $M_i \neq \emptyset$ **then** return TRUE, otherwise FALSE.

Example 3.1. To illustrate the above terms, let us consider the XML document and the DCS as shown in Fig. 7. The XML document in Fig. 7(b) could have been actually derived from a DTD or an XML schema whose skeleton may be depicted by the DCS in Fig. 7(a). Let us also assume the following simple XML query expressed in XQL (Robie et al., 1998), which requests the stock names of those stocks with a day minimum greater than 10000.

```
/Portfolio/Stock/Name[/Portfolio/Stock//DMin > 10000]
```

This query consists of two path expressions: one is a simple path expression '`/Portfolio/Stock/Name`' and the other one is a parameterized path expression '`/Portfolio/Stock//DMin`'. There is also a condition associated with the parameterized path expression. The double slash ('//') means that the simple paths that match this expression are those which contain a label `DMin` one or more levels deep below the label `Stock` (arbitrary descendants). The XQL query above

may be graphically depicted as shown in Fig. 7(c). In Fig. 7(c), each single slash delimiter is depicted as a solid edge connecting the labels, while a double slash delimiter is depicted as a dashed edge. The query in Fig. 7(c) is relevant to the DCS in Fig. 7(a) if there exists a subtree of the DCS which is an instance of the query pattern. The shaded nodes in Fig. 7(a) show the existence of such a tree in the DCS; therefore we may conclude that the query is relevant to all documents that are derived from this DCS. Executing the above query on the XML document in Fig. 7(b) with an XQL-compliant query engine yields the following result:

```
<Name> DJ INDU AVERAGE </Name>
<Name> NIKKEI 225 INDEX </Name>
```

By checking the occurrence of an XML query pattern in a DCS, we may find out which DCSs are relevant to the query and consequently consider all the XML documents derived from these DCSs as candidates for the refinement phase. If a query pattern does not occur in a DCS, then it is considered irrelevant to any document derived from this DCS. Executing an irrelevant query over some documents yields no results; i.e., the result set is empty.

A straightforward way to find relevant XML documents to a given query in a large XML repository which encompasses numerous schemata is to check exhaustively the query against every single DCS in the repository. However, doing so may slow down the process of finding the candidate set of XML documents for the refinement phase. The following section introduces the concept of merging DCSs into more generic structures in order to solve the problem of exhaustive checking.

4. DCS Basic Operations

4.1. DCS Merging

A merger DCS depicts a generic XML schema which combines two or more simpler DCSs. The introduction of the concept of merger DCS aims at limiting the initial search space by merging primitive DCSs into more general ones, which may then be used as matching targets against XML queries. The merging process is accomplished as follows.

Let X be a DCS. Let us denote as $root(X)$ the root label instance of X. Let us denote as L_X the multiset containing all the label instances of X. L_X is a multiset rather than a set since a label may occur more than once in a DCS at different locations. Let us denote as $C_X(l)$ the set containing all the children of the label instance l in X (child set). Let us denote as $T(l)$ the tree whose root is the label instance l and as $T_X(l)$ the subtree of X whose root is the label instance $l \in L_X$.

To ease the process of DCS separation, which is discussed in a subsequent section, a reference number $n_X(l)$ is introduced for each label instance l of the DCS X, which indicates the number of underlying primitive DCSs in which the label instance is present. In a primitive DCS each label instance can be referenced only once; consequently for all labels in a primitive DCS X $n_X(l) = 1$. A merger DCS is created by the unification of many DCSs. If a merger DCS is formed by joining m primitive DCSs and a label instance l appears in n DCSs, then the reference number of this label instance in the merger will be equal to n.

Let A and B be two DCSs and M be the resulting merger DCS. The merger

DCS M contains finally the union of elements of A and B. The merger DCS is built recursively by using Algorithm 4.1.

Algorithm 4.1. *Merge(A, B)*

Input: DCS trees A and B, which are to be merged.

Output: A merger DCS tree M that contains A and B.

Method: It merges two DCSs into a merger DCS. The merger DCS contains all the paths found in any of the original DCSs.

M1 *Initialization*: The method creates a temporary root label tl and sets $T_M(root(M)) = T(tl)$. It makes the tree A subtree of M by assigning the $root(A)$ to be a child of $root(M)$. It calls the recursive function *Join* in step $M2$ by $Join(T_M(root(M)), T_B(root(B)))$.

M2 Function $Join(T_M(m), T_B(b))$.

M2.1 if $b \notin C_M(m)$ **then** add b in $C_M(m)$
$n_M(b) = n_M(b) + n_B(b)$

M2.2 For each $x \in C_B(b)$ and $b \in C_M(m)$ **do** $Join(T_M(b), T_B(x))$.

M3 if $root(A) = root(B)$ **then** $root(M) = root(A)$.

Step M1 is the initialization step used to call the recursive function *Join*.

Step M2 merges two trees: one is a subtree of M and it is rooted by the label m and the other is a subtree of B and is rooted by the label b.

Step M2.1 checks to make sure that the label b is a child of label m. If it is not ($b \notin C_M(m)$), then it adds the label to the child list of m. This means that a new instance of the label is created and it is assigned to be a child of m, so that eventually $b \in C_M(m)$. The reference number of the label instance b in M, $n_M(b)$, is updated by adding to it the reference number of the label b in B, $n_B(b)$. If B is always a primitive DCS, $n_B(b)$ is equal to 1.

Step M2.2 performs the *Join* function for all the children of b. The recursion stops when there are no $x \in C_B(b)$ any more because $C_B(b) = \emptyset$, which means that b has no children (it is a leaf label).

Step M3 checks if the DCSs A and B have the same root label. tl is just an assisting label to realize the merging. The resulting merger DCS is the child of tl if both A and B have the same root label. In case A and B have different root labels, the merger DCS is rooted at tl, which is the parent label of both $root(A)$ and $root(B)$ labels.

In the discussion that follows, we assume merging of DCSs which have common root labels. The following example shows the merging of two primitive DCSs.

Example 4.1. *Constructing a merger DCS*. Let us consider the primitive DCS in Fig. 8(a), which is similar to that in Fig. 7(a), although it has a different structure. The merger DCS will be the one shown in Fig. 8(b). The merger DCS keeps the structure of both primitive DCSs. It could be seen as the union of the two primitive DCSs in Fig. 8(a), and Fig. 7(a). The reference number of each element of the merger schema is also shown on the nodes of the tree in Fig. 8(b). Those nodes that have reference number equal to 1 occur only in one of the corresponding primitive DCSs, while those nodes with reference number equal to 2 occur in both DCSs. The advantage of the merger DCS is that it is sensitive to queries that are relevant to both primitive DCSs. Considering again

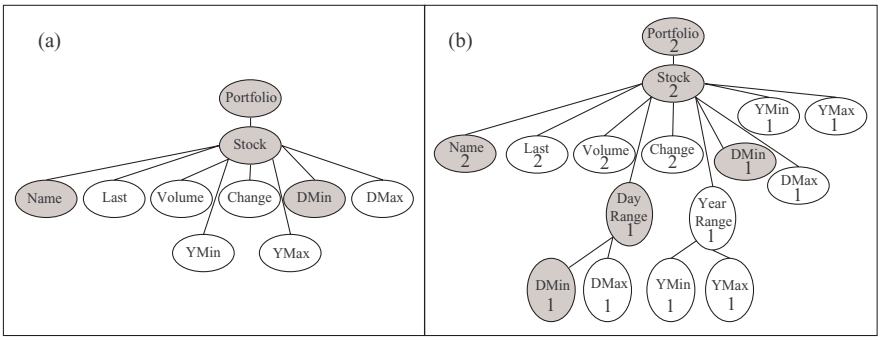

Fig. 8. (a) A primitive DCS, which is similar to that in Fig. 7(a). (b) The merger DCS obtained by merging the primitive DCSs in (a) and that in Fig. 7(a).

the query in Example 3.1: we see that the query is matched against both DCSs and consequently it is relevant to all the documents derived from these DCSs. The shaded nodes in Fig. 8 show the subtree, which matches the pattern tree in Fig. 7(c) of the XML query.

A merger DCS captures the relevance between primitive DCSs and allows us to avoid exhaustive checking to find out the relevant XML document to a given query. What is actually needed in this case is to check if the query is matched against the merger DCS. In that way, we may substantially decrease the search effort for relevant XML documents. Although using mergers to join DCSs is promising, it may not yield the expected gain if we do not consider carefully what to merge. The primitive DCSs to be merged should be as *similar* as possible. Merging similar DCSs offers an advantage since the resulting mergers act as meta-classes that organize the underlying XML documents in separate categories, which may be used to efficiently evaluate XML queries over large collections of XML sources. A subsequent section defines a measure for similarity between XSDs and discusses how it can be estimated.

4.2. Merge Performance

The merging operation, $Merge(A, B)$, is realized by performing a pre-order parsing in the tree B. Each label in B is visited once. Therefore, the cost for accomplishing the merging operation linearly depends on the size of the tree B. The processing of each label in B includes the following:

- Add the label to A if it is not present.
- Update the reference number of the label.

Both the above operations have a constant cost. Therefore, the *Merge* operation works in $O(|B|)$ time, where $|B|$ is the number of nodes in B.

4.3. DCS Separation

The Merge operation is used to join two relevant DCSs. The DCS separation procedure does exactly the opposite. It splits a merger DCS into two DCSs.

Separation is the complement of the Merging operation. That is, if A and B are two primitive DCSs, the sequence $C = Merge(A, B)$, $Separate(C, B)$ has no effect at all and it results in two primitive DCSs A and B. Let M be a merger DCS and B be a DCS that is contained in M and we want to extract B from M. The operation $Separate(M, B)$ removes an instance of the B subtree from M. Let us consider the example in Fig. 9. The merger DCS M unifies two other merger DCSs A and B. As we mentioned previously, the reference number in each label depicts the number of primitive DCSs in which the label instance is present. For example, label instance c in B has a reference number equal to 1 because one primitive DCS contains this label instance. The label instance c in the merger DCS M has a reference number 3, which is the sum of those in B and A (this summation is performed in step M2.1 in the *Merge* algorithm). Separating B from M will yield A. This means that the reference number of a label instance contained in B will be decreased to the extent dictated by B. In the case of the label instance c, the reference number will decrease by one, because one reference of this label instance exists in B. Doing so in label e, the resulting reference number is zero; in that case the label e is totally removed from M. The $Separate(M, B)$ operation is described in Algorithm 4.2.

Algorithm 4.2. *Separate(M, B)*

Input: DCS trees M and B. The tree B is a subtree of M.

Output: The DCS tree M from which B has been removed.

Method: It subtracts B from M. Label instance reference numbers are updated and if a reference number becomes zero the label instance is removed from the tree M.

S1 *Initialization*: Find where B is embedded in M.

S2 Let the tree $T_M(root(B))$ be the instance of B in M. We parse the tree $T_M(root(B))$, in any order (pre-order or post-order) and we decrease the reference number of each label instance by the amount dictated by the corresponding label instance in B (i.e. $n_M(b) = n_M(b) - n_B(b)$, where $n_M(b)$ and $n_B(b)$ are the reference numbers of label instance b in M and B respectively). If the result is zero for a label instance reference number ($n_M(b) = 0$), this label instance is removed from $T_M(root(B))$.

In the Example 4.1, if M is the merger DCS in Fig. 8(b), and B is the DCS in Fig. 8(a), then $Separate(M, B)$ will result in the schema shown in Fig. 7(a).

4.4. Separate Performance

The operation of separating a DCS B from a merger M includes two steps. The first one is to find where B is located in M and the second one is to remove B from M The first step includes a search for finding where B is located in M. This is a tree-matching problem and algorithms already exist that provide a solution in $O(|M||B|)$ time (Kilpeläinen and Mannila, 1995), where $|M|$ and $|B|$ are the number of nodes in M and B respectively. B is parsed during the second step and its corresponding labels are removed from M by updating the label reference numbers. In this process, each label is visited once and the cost for updating the labels depends on the size $|B|$ of the tree B. Thus, this step is performed in $O(|B|)$ time.

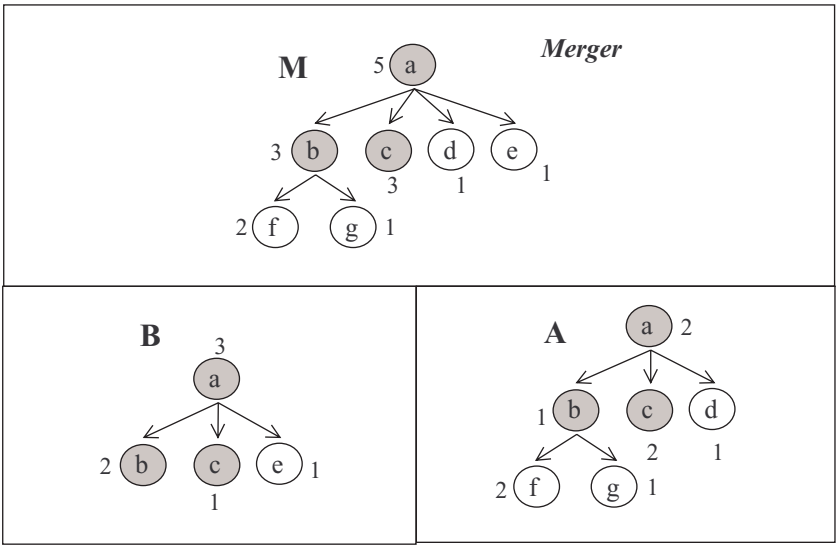

Fig. 9. DCS separation.

4.5. Similarity Measures Between DCSs

Similarity between DCSs is defined in terms of proximity, which is based on the distance between DCSs. The larger the distance between two DCSs, the more dissimilar they are. DCSs may be viewed as trees and therefore the distance between them should be based on tree differences. A mathematical model for obtaining distance measures between DCSs is therefore discussed. Similarity measures between trees are based on syntactical approaches and they are further discussed in Zhang (1995).

The distance between two DCSs is based on the edit operations that should be performed to one of them in order to obtain the other. An edit operation on a DCS may be an *insertion*, a *deletion* or a *substitution* of one node by another. Insertion is the complement of deletion.

Insertion of a node x into a DCS as a child of node y may be accomplished so that the resulting DCS contains x as a child of y with no children or takes as children some of the children of y. Let $y_1, \ldots, y_k$ be children of y, then for some $0 \leqslant i \leqslant j \leqslant k$, the children of y in the resulting tree (after the insertion of x) will be $y_1, \ldots, y_i, x, , y_j, \ldots, y_k$. If $j = i + 1$, x has no children otherwise x has children $y_{i+1}, \ldots, y_j$.

Deletion of a node x from a DCS is accomplished so that the father y of x takes all the children of x. Let $y_1, \ldots, y_k$ be the children of y and $x = y_i$, and let $x_1, \ldots, x_j$ be children of x, then the children of y in the resulting tree (after the deletion of x) will be $y_1, \ldots, y_{i-1}, x_1, \ldots, x_j, y_{i+1}, \ldots, y_k$.

Substitution of a node x by a node y is accomplished so that the children of y in the resulting DCS are the children of x in the original DCS.

Any of the above elementary edit operations may be represented as a general

substitution of the form $\alpha \rightarrow \beta$, which means α *is replaced by* β. In the case of the substitution operation above, α and β represent two distinct nodes. The deletion of a node α may be represented as $\alpha \rightarrow \emptyset$ (i.e. $\beta = \emptyset$), where $\emptyset$ is the null node. The insertion of the node β may be represented as $\emptyset \rightarrow \beta$ (i.e. $\alpha = \emptyset$).

An editing operation $\alpha \rightarrow \beta$ is associated with a cost $w(\alpha \rightarrow \beta)$. This cost can be different for different nodes. For example editing a node, which is closer to the root might have higher cost than editing a leaf node or vice versa. In the case that there is no distinction between nodes, a universal editing weight for all nodes may be used.

The distance between two DCSs T_1 and T_2 is measured in terms of the number of editing operations required to change T_1 into T_2 taking into account the cost of each editing operation. The cost w to be a distance metric should satisfy the following metric axioms (Kruskal, 1983; Zhang and Shasha, 1989).

1. $w(\alpha \rightarrow \beta) \geqslant 0$ and $w(\alpha \rightarrow \alpha) = 0$
2. $w(\alpha \rightarrow \beta) = w(\beta \rightarrow \alpha)$
3. $w(\alpha \rightarrow \gamma) \leqslant w(\alpha \rightarrow \beta) + w(\beta \rightarrow \gamma)$

Let S_i be a sequence of editing operations $s_{i1}, s_{i2}, \ldots, s_{ik_i}$ that changes the DCS T_1 into T_2. The cost of performing all the operations in the editing sequence is then given by

$$W(S_i) = \sum_{j=1}^{k_i} w(s_{ij}) \tag{1}$$

The distance between the DCSs T_1 and T_2 is then formally defined as

$$\delta(T_1, T_2) = \min\{W(S_i) | S_i \text{ is an editing operation sequence}\} \tag{2}$$

where S_i is an editing operation sequence that changes T_1 into T_2. An algorithm that estimates the above-defined distance between trees is discussed in Zhang and Shasha (1989), which also presents a dynamic algorithm that solves the minimum distance problem in sequential time.

4.6. Properties of Merger DCS

A merger DCS is a bounding structure that unifies simpler DCSs. The main use of a merger DCS is to facilitate the evaluation of XML queries. In order to achieve this, DCSs must have the following properties:

Subset property: If M is a merger of A and B, then A and B are subtrees of M, i.e., $A \subseteq M$ and $B \subseteq M$.

Distance property: Let M_X be the merger of $X_1, X_2, \ldots, X_n$ and M_Y be the merger of $Y_1, Y_2, \ldots, Y_m$. Then $\delta(X_i, X_j) \leqslant \delta(X_i, Y_k)$, for any $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$ and $1 \leqslant k \leqslant m$. This means that the distance between two DCSs belonging to the same merger is smaller than the distance of any pair of DCSs belonging to different mergers. This property keeps a merger tightly bound and cohesive.

5. The XSD Structure and Maintenance Algorithms

5.1. XSD Tree

XSD is a hierarchical structure whose aggregation technique to cluster XML schemata is based on DCS merging. The objects used for merging are primitive DCSs. The motivation for forming such a structure is that DCSs relevant to a path query tend to be more similar to each other than irrelevant DCSs and they may be merged together. XSD structure is mainly used to accelerate query processing by considering only a small number of relevant DCSs and consequently a limiting number of XML documents, rather than the entire corpus of XML documents. An alternative advantage of XSD structure is that it may also be suggested as a method for facilitating browsing. XSD is a data structure, which is designed to ease the process of finding relevant XML documents in a corpus and it is mainly used as an access means to XML repositories.

An XSD tree has two types of node entries; *internal* (non-leaf node) entries and *leaf node* entries (see Fig. 10). Each entry E in an XSD node has two fields, named S and P, which are denoted $E.S$ and $E.P$ respectively. The $E.S$ field accommodates a DCS and $E.P$ contains one or more pointers. If the entry E is in a non-leaf node, E has the form $(E.S, E.P)$, where $E.S$ represents a merger DCS and the $E.P$ field contains a single pointer to a child XSD node. The child node may contain many entries whose DCSs compose the merger schema $E.S$ of the parent. In other words, the merger schema $E.S$ is the union of the DCSs of the entries in the child node pointed to by $E.P$.

If the entry E is in a leaf node, the $E.S$ represents a primitive DCS and $E.P$ is a list of pointers, each one pointing to an XML document that is derived from this primitive DCS. Consequently, a leaf node entry E has the form $(E.S, E.P)$, with $E.P = E.X_1, E.X_2, \ldots, E.X_n$, where $E.X_1, E.X_2, \ldots, E.X_n$ are pointers to XML documents which are instances of the $E.S$ primitive DCS. Every leaf node may contain an arbitrary number of references (i.e., many XML documents may be derived from the same DCS).

The number of entries in an XSD node is n, with $m \leqslant n \leqslant M$, where M and m are the maximum and minimum numbers respectively of the DCSs that can be accommodated in the XSD node. The relationship between m and M could be defined as $m \leqslant \frac{M}{2}$. It is worth noting that while the leaf nodes may have an arbitrary number of pointers to XML documents, the non-leaf nodes have a limited number of pointers to child nodes. This is because the XSD structure is tuned to organize DCSs, which are actually XML schemata, rather than XML sources. Limiting the number of DCSs that can be accommodated in a node between m and M is done for performance purposes. For instance, small values of M will create a deep XSD structure, whereas large values of M may create a wide XSD structure. In general, m and M may vary and different values may be used in a way that increases the performance.

Figure 10 shows the structure of a typical XSD organization and illustrates the merger DCSs as entries of non-leaf nodes and primitive DCSs as entries of leaf nodes. In Fig. 10 the nodes N_1, N_2 and N_3 contain merger DCSs and the leaf nodes N_4, N_5 and N_6 contain primitive DCSs. A DCS entry of an intermediate node is the merger of all descendant DCSs. For example, the schema S_5 in the intermediate node N_3 in Fig. 10 is the merger of S_{10} and S_{11}. Referring to Example 4.1, S_5 may represent the DCS in Fig. 8(b) and the child DCSs S_{10} and S_{11} may be the primitive DCSs in Fig. 7(a) and Fig. 8(a) respectively.

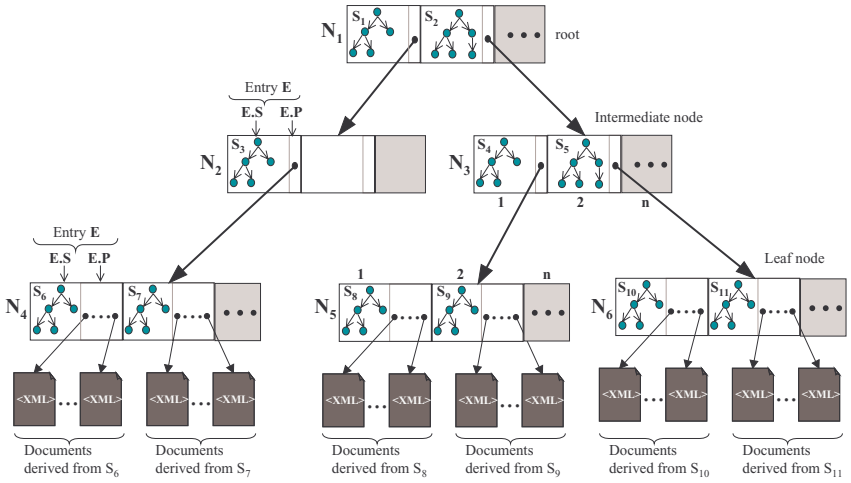

Fig. 10. XSD Tree. Each node may contain n entries. DCSs in intermediate node entries are mergers, while DCSs in leaf node entries are primitive. A primitive DCS points to all XML documents derived from it ($m \leqslant n \leqslant M$).

Two basic algorithms are proposed for maintaining the XSD structure: the first one for inserting new primitive DCSs and the other one for deleting existing primitive DCSs. There are also two other operations, which are used in the XSD structure but these are not for maintaining the XSD tree. They are mainly used to *associate* and *disassociate* XML documents to an existing primitive DCS. The association (disassociation) of an XML document to a primitive DCS is accomplished by adding (removing) a pointer to that leaf entry at which the DCS resides. These operations are trivial and they will not be discussed further.

5.2. Indexing Overhead

The primitive DCSs, which actually are the XML schemata to be indexed, are stored at the leaf nodes of the XSD tree. All intermediate nodes are used to facilitate the access to the primitive DCSs. The intermediate nodes outline the space overhead for maintaining the XSD structure. Let N be the number of primitive DCSs and n be the number of entries that can be accommodated in an XSD node. The number of intermediate nodes is given by

$$\frac{N-n}{n(n-1)} \tag{3}$$

Therefore the intermediate nodes constitute

$$\frac{N-n}{n(N-1)}100\% \tag{4}$$

of the total number of XSD nodes, if each leaf node contains n DCSs (given that $N = n^{(d+1)}$ and the XSD tree is complete). This is depicted in Fig. 11. As the number n of entries in an XSD node becomes smaller, we need more intermediate nodes for the same N and the space overhead increases.

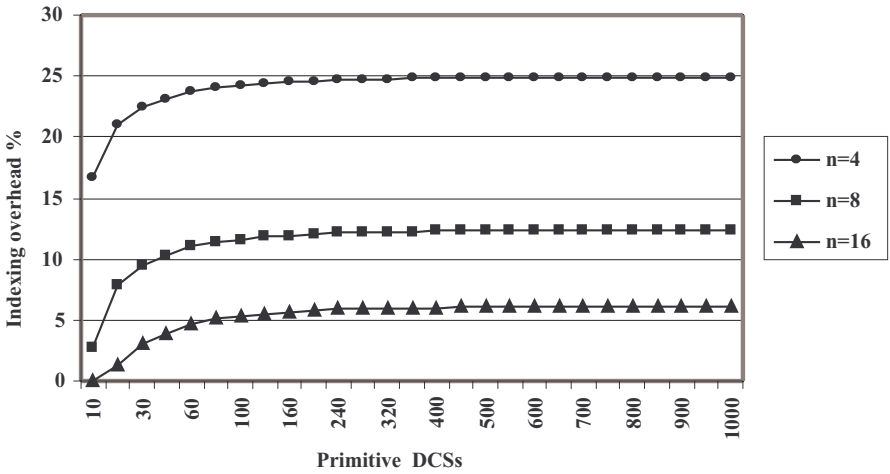

Fig. 11. Indexing overhead as a function indicating the percentage of nodes used as intermediate nodes in an XSD tree.

5.3. Query Evaluation Through XSD

An XML query is evaluated by searching for DCSs that match the query. The matching of a query against a DCS is accomplished by using Algorithm 3.2. Algorithm 5.1 is the *Search* algorithm for finding relevant primitive DCSs in the repository and eventually all the XML documents that are derived from these DCSs.

Algorithm 5.1. *Search(T, Q)*

Input: An XSD tree whose root is referenced by T and a path query Q.

Output: A set D of those XML documents, which are relevant to Q.

Method: Decomposes the search space and recursively search the XSD tree. More than one subtree under a visited node may need to be searched.

SR1 *Search non-leaf nodes*: If T is not a leaf, then for each entry E of T check whether the merger schema $E.S$ matches Q by calling $IsRelevant(E.S, Q)$. If so, invoke $Search(E.P, Q)$.

SR2 *Search leaf nodes*: If T is a leaf, for all entries E of T check whether $E.S$ is relevant to Q by calling $IsRelevant(E.S, Q)$ and, if so, mark those XML documents that are derived from those primitive $E.S$ that match the query Q and add them to the set D.

The *Search* algorithm starts from the root and moves towards the leaves by checking whether the query is matched against any intermediate merger DCSs. All the nodes from the root to the desired leaf primitive DCSs (those that match the query) are visited and need to be searched. In that way, exhaustive matching is avoided and eventually only relevant DCSs are searched. This limits the number of searching steps and, on the other hand, returns exactly those XML documents that are relevant to the query. As a consequence, no irrelevant document is returned and all relevant documents are considered in the result set. Moreover,

the searching space is reduced and eventually the relevant XML documents are efficiently retrieved avoiding exhaustive searching.

Example 5.1. *Query evaluation* To illustrate the *Search* algorithm, let us consider the query A and the XSD structure as shown in Fig. 12. The XSD tree in Fig. 12 consists of three nodes N_1, N_2 and N_3, from which N_2 and N_3 are leaf nodes containing primitive DCSs and N_1 is an intermediate type node (in this case the root of the tree) that contains merger DCSs. The objective is to identify those XML documents in the repository which are relevant to the query A. This means that the relevant XML documents should contain paths that match the query A. The query A (expressed in XQL as '$a/*/f$') consists of a single parameterized path expression, which is matched against any path that starts with the label 'a', ends with label 'f' and has any label between 'a' and 'f'. The *Search* algorithm starts from the root and it checks first whether the query is relevant to M_1 by calling $IsRelevant(M_1, A)$. Since an instance of the parameterized path ('$a/b/f$') is in M_1, M_1 is assumed to be relevant to the query. Doing the same with M_2, we see that M_2 is not relevant to the query A (no instance of the query is in M_2). So far, the search space has been divided into two subspaces: one under M_1 and another under M_2. Since only M_1 is relevant to the query A, we visit the child of the root node N_1, which is pointed to by M_1. Then we repeat the same checking with any DCS in the child node N_2, and we find eventually that only P_2 is relevant to the query A and therefore all the XML documents, which are derived from P_2 (i.e. D_{21}, D_{22}) are considered to be relevant to the query A. Query B is relevant to the XML documents D_{31}, D_{32}, D_{41} and D_{42}.

An interesting aspect of XSD is that it allows visiting DCSs that are close to the one that matches the query. For example, P_1 does not match the query A, although it is close enough to matching the query A. It contains the path 'a/b', which partially match the query. This feature may be used for ranking the output according to a matching degree. In this example, D_{21} and D_{22} may be in the class of 100% matching, while D_{11} and D_{12} are in the class of 66% (since 2/3 of the query is matched against P_1). Such metrics, which are based on the extent to which a query is matched against a DCS, may be used to estimate approximate matching. Another advantage of XSD is that it can facilitate browsing. One can start from the root of an XSD tree and selectively visit child nodes by checking the structures of the DCSs. Following the parent/child node links, one can identify those XML documents that fit some preference criteria.

5.4. Search Performance

Usually, searching for a primitive DCS requires order $\log N$, where N is the number of primitive DCSs in the leaf nodes. Let d be the depth of the XSD tree (root's depth is zero). Let n be the number of entries in each XSD node. If the XSD tree is complete, the number of leaf nodes is n^d, and if each leaf node contains n entries then

$$N = n^{(d+1)} \tag{5}$$

Each primitive DCS may be accessed by visiting $(d + 1)$ nodes including the root and the leaf node, which actually accommodates the primitive DCS. The worst case is the one where the query is compared to all n entries in each XSD node we visit as we descend the XSD tree. A comparison is realized by invoking

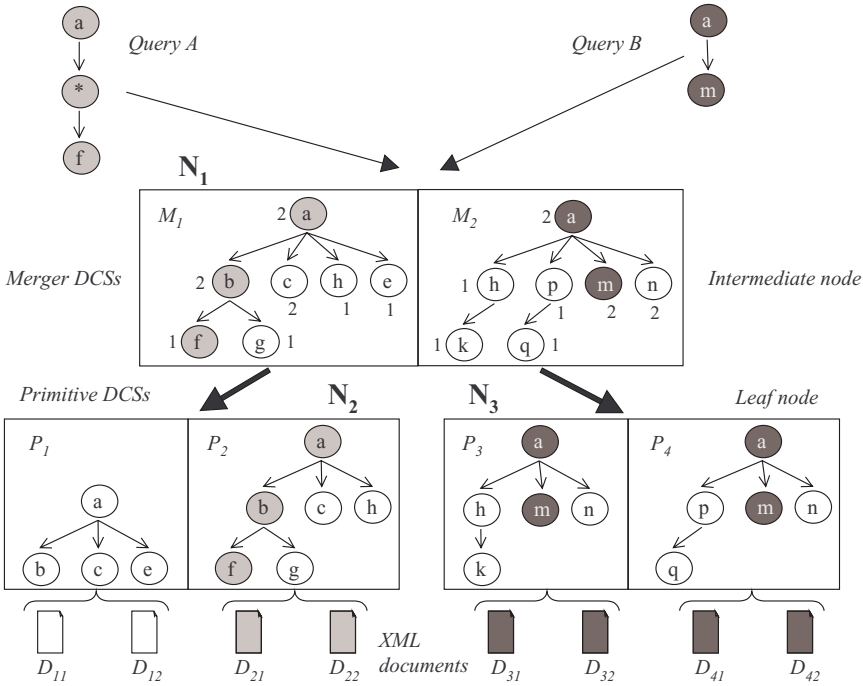

Fig. 12. XML query evaluation.

the procedure *IsRelevant*() discussed in Algorithm 3.2. The maximum number of comparisons is $n(d+1)$, which is also given by

$$\frac{n}{\log n}(\log N) \tag{6}$$

Figure 13 shows the number of comparisons as a function of the number N of the primitive DCS stored in an XSD tree. This figure shows how the number of comparisons increases in terms of N (ranging from 10 to 1000) for three different values of n. This clearly indicates that the XSD access method requires much fewer comparisons than the exhausting search in order to identify those primitive DCSs that are relevant to the query. For example, for $n = 8$ and $N = 700$, it requires just 25 comparisons.

5.5. Primitive DCS Insertion

Before we proceed further in discussing the *Insert* algorithm, we examine the metric which is used to measure the degree to which a merger DCS contains a primitive DCS. The containment degree $C_d(M, P)$ is estimated by assessing the amount to which the merger M will be extended if M and the primitive DCS P are merged and the resulting DCS is assigned to M (i.e., $M = Merge(M, P)$). Formally, $C_d(M, P)$ is defined as follows.

Definition 5.1. Let N be the number of nodes in primitive DCS P and L be the

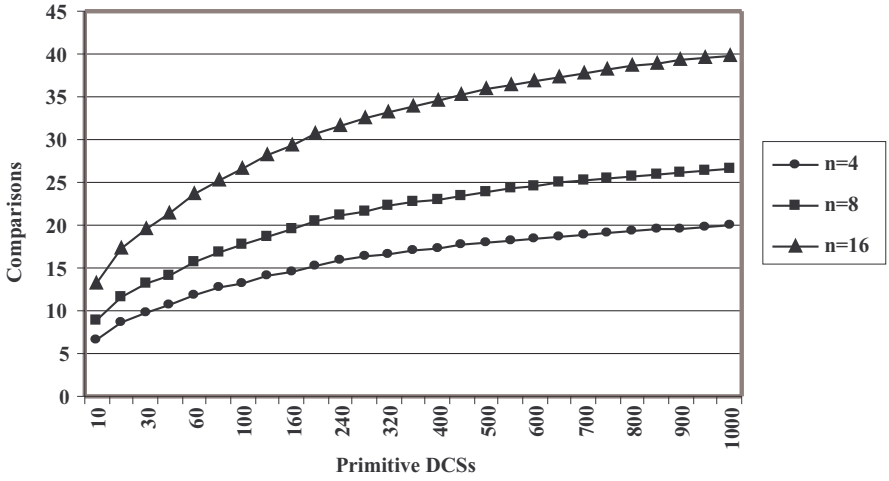

Fig. 13. Number of comparisons in terms of N (number of primitive DCSs).

number of nodes that will be added to M after merging it with the primitive P. The containment degree of M with respect to P is defined as

$$C_d(M,P) = \frac{N-L}{N} \tag{7}$$

If N nodes are added after merging (i.e., $L = N$), then $C_d(M,P) = 0$, which means that no part of P is contained in M. If no node is added (i.e., $L = 0$), then $C_d(M,P) = 1$, which means that P is totally contained in M. In general $C_d(M,P)$ is a real number between 0 and 1 inclusive that specifies the extent to which the primitive DCS P is contained in the merger M. Let us consider the DCSs in Fig. 14, where M_1 is merged with P_n and it is transformed into M_1^*, ($M_1^* = Merge(M_1, P_n)$) and M_2 is merged with P_n and it is transformed into M_2^*, ($M_2^* = Merge(M_2, P_n)$). The number of nodes in P_n is $N = 6$. In order to accommodate the new primitive P_n, M_1 is extended by adding just one node (node k), thus $L = 1$ and $C_d(M_1, P_n) = 5/6$. Doing the same with M_2, M_2 is extended by three nodes (b, f, e), thus $L = 3$ and $C_d(M_2, P_n) = 1/2$.

The containment degree metric C_d is mainly used by the *Insert* algorithm to find where to place new primitive DCSs. The purpose of the *Insert* algorithm is to add a new primitive DCS to the XSD structure.

Algorithm 5.2. $Insert(T, P_n)$

Input: An XSD tree whose root is referenced by the pointer T and a new primitive DCS P_n.

Output: The XSD tree that results after the insertion of P_n.

Method: Finds recursively where to place the new entry containing the primitive DCS P_n. It starts from the root and descends until a leaf is reached. It adds the entry into the leaf node and adjusts the XSD tree accordingly by performing any necessary node splitting.

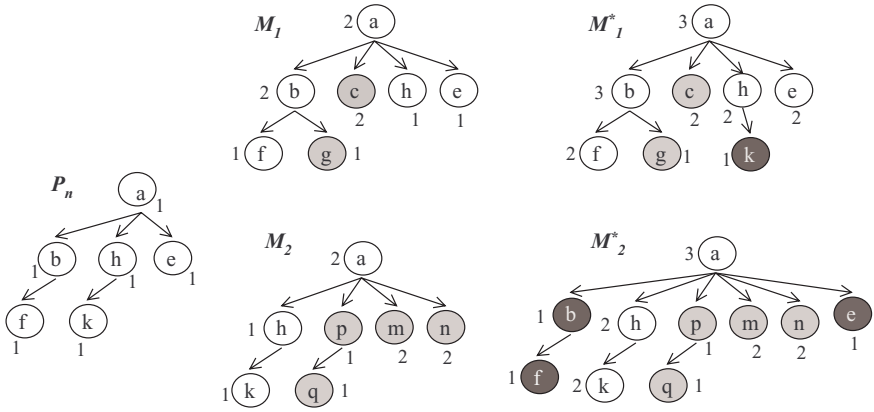

Fig. 14. Containment degree. M_1^* results from the merging of M_1 and P_n. M_2^* results from the merging of M_2 and P_n. The containment degree of M_1 and M_2 with respect to P_n is 5/6 and 1/2 respectively. This means that 5/6 of P_n is contained in M_1 and 1/2 of P_n is contained in M_2. Therefore M_1 contains a larger portion of P_n than M_2.

I1 *Insert into non-leaf node*. If T is not a leaf, then for all entries E in T, compute the containment degree $C_d(E.S, P_n)$. Let D be the entry in T for which the $C_d(D.S, P_n)$ is maximum, then invoke $Insert(D.P, P_n)$.

I2 *Insert into leaf node*. If T is a leaf, then check if there is a place where the new entry containing the P_n can be accommodated. A node can accommodate a new entry if the number of existing entries in the node is less than M (maximum threshold).

I2.1 If there is room and the new entry can be placed in T, then add the new entry in T and adjust the XSD tree by merging P_n with any ascendant merger DCS in the non-leaf nodes from T up to the root. That is, the new primitive DCS P_n is merged first with the DCS of the parent entry of the leaf T, then with the DCS of the grandfather entry of T and so on until the update reaches the root of XSD tree.

I2.2 Otherwise, if the number of existing entries is M and a new entry cannot be placed in T, then the node T splits into two nodes containing disjoint groups of entries by invoking $SplitNode(T)$. The new entry is placed in one of the resulting nodes and the XSD is adjusted starting from the new node towards the root by re-estimating all the ascending merger schemata in the non-leaf nodes all the way up to the root.

To illustrate how the *Insert* algorithm works, let us consider the XSD as shown in Fig. 15. Figure 15(a) shows the XSD before the insertion of the new primitive DCS P_n, while Fig. 15(b) shows the XSD after the insertion of P_n. Figure 15 shows the same P_n, M_1 and M_2 as shown in Fig. 14. The *Insert* algorithm first checks the containment degrees of the root entries $C_d(M_1, P_n)$ and $C_d(M_2, P_n)$ and chooses that entry which yields the largest C_d (M_1 in this case). This means that the new entry P_n will be inserted in the subtree rooted by the node N_2, which is the child pointed to by M_1. N_2 is a leaf node and in this case the *Insert* algorithm

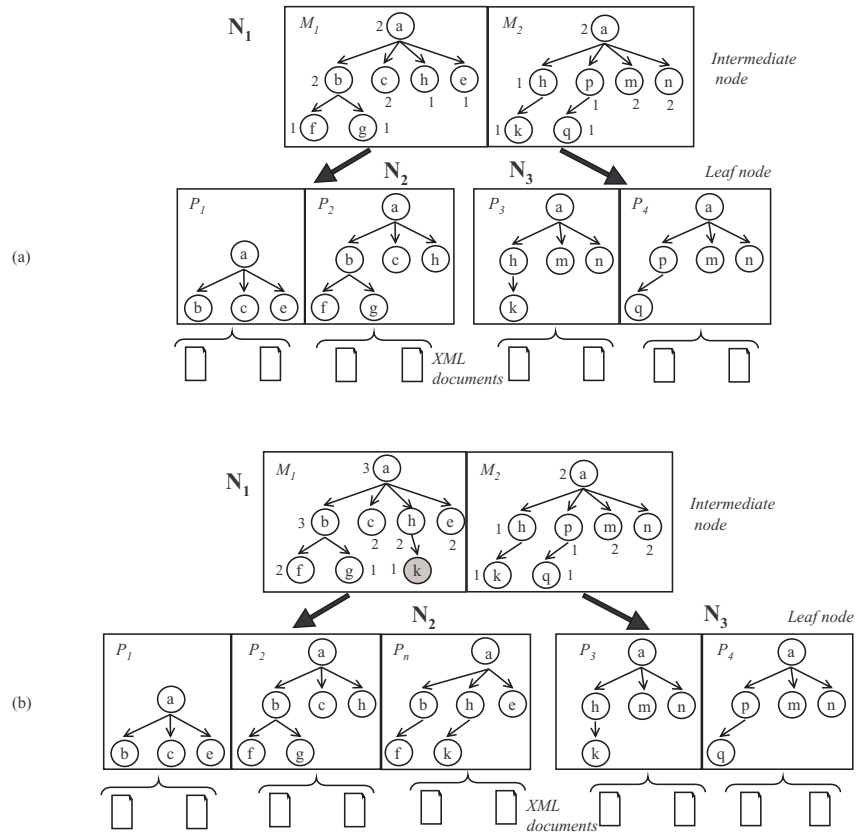

Fig. 15. Inserting a new primitive DCS. (a) XSD structure before the insertion of the primitive DCS P_n in Fig. 14. (b) XSD structure after the insertion of P_n.

checks for available space in N_2 to store the entry P_n. If there is no space a split operation is performed and the entries in N_2 are divided into two disjoint sets. In our example P_n is stored in N_2 and any ancestor of N_2 is merged with P_n all the way up to the root, so that eventually the XSD is updated to reflect the insertion of the new entry (i.e., M_1 is modified accordingly).

In case a node overflows (when the number of entries exceeds the maximum threshold M) it is necessary to split the node into two disjoint sets in such a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches. The proposed node-splitting technique is based on a clustering algorithm, which is used to create two tightly bound clusters of DCSs (one for each node). For this purpose, the similarity distance discussed in Section 4.5 is employed to define DCS proximity. In principle, any clustering technique which is based on distance metrics can be used for this purpose. The *SplitNode* procedure, described below, is based on an agglomerative clustering algorithm (Jain et al., 1999).

Algorithm 5.3. *SplitNode(L)*

Input: An XSD leaf node referenced by the pointer L.

Output: Two new nodes L_1 and L_2 containing two disjoint sets of entries.

Method: A leaf node split may cause consequent splitting to ascendant non-leaf nodes and propagate node split upwards as far as the splitting in lower levels makes nodes in the higher levels to overflow. A leaf node split can make the XSD tree grow taller if the splitting is propagated up to the root.

SN1 *Compute proximity*. The method computes the proximity matrix containing the distance between each pair of DCSs in the node. It treats each DCS as a distinct cluster.

SN2 *Reduce the total number of clusters*. It finds the most similar pairs of DCSs and merges them into a single DCS. It updates the proximity matrix to reflect this merging.

SN3 *Repeat*. It repeats step SN2 until the number of clusters is two.

SN4 It finally assigns the two clusters to L_1 and L_2.

5.6. Insertion Performance

The primitive DCS to be inserted is checked against existing merger DCSs in order to find where to place the new DCS. Each such comparison includes the evaluation of the containment degree C_d of the merger entry with respect to the new entry. The containment degree C_d determines to what extent the XSD merger entry contains the new entry. The worst case is the one where the new entry is compared to all of the entries in an XSD node. In this case, the number of such comparisons is $n * d$, where n is the number of entries in an XSD node and d is the depth of the XSD tree. If there is no room for the new entry, then the insertion cannot be completed and a node split must be performed in order to make room for the new entry. Splitting is done to keep the XSD tree balanced. However, if no split is performed, the cost for inserting a new primitive DCS is equal to $n * d$ comparisons, where each comparison corresponds to an estimate of the containment degree C_d. The *SplitNode* algorithm is based on a clustering approach that begins with each DCS as if it was a distinct (singleton) cluster and successively merges clusters together until the number of clusters becomes two. Splitting may be propagated to other intermediate nodes upwards to the root. It is difficult to estimate the number of splits, since node splitting is performed in a dynamic fashion and depends on the number of entries n in the XSD node as well as the degree of similarity between primitive DCSs. However, the cost of a node split can be estimated as a number of pairwise comparisons among the entries of an XSD node. Each such comparison requires the estimation of the distance between every pair of entries in an XSD intermediate node. If an XSD node has n entries, the total number of entries to be checked is n plus the new one (i.e., $n + 1$) and the number of comparisons to be performed is $\frac{n(n+1)}{2}$ in the first iteration. There are $n - 1$ iterations and the total number of comparisons for completing clustering is given by

$$\sum_{i=1}^{n-1} \binom{i+2}{2} \tag{8}$$

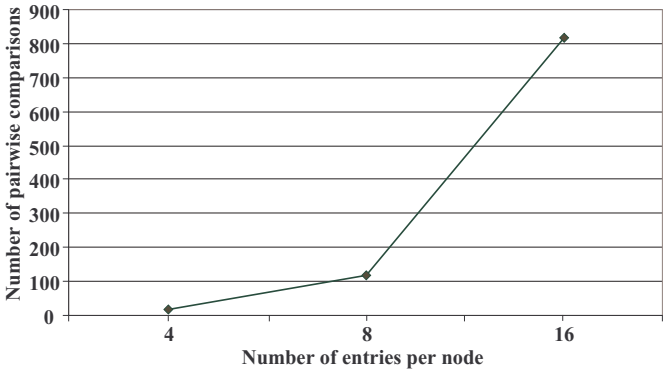

Fig. 16. Number of pairwise comparisons for performing the agglomerative clustering during a node split.

Figure 16 shows how the number of pairwise comparisons increases with regard to n (for $n = 4, 8, 16$) in the case of a node split. As we can see, the number of pairwise comparisons increases exponentially. When n is small, the node splitting is performed very efficiently but the XSD tree becomes deep and we visit more intermediate nodes performing more comparisons during searching. On the other hand, when the number of entries per node becomes large, the XSD tree becomes flat and the search becomes less costly, while the split operation requires more pairwise comparisons for accomplishing the agglomerative clustering. This is the ultimate cost for obtaining good search performance.

5.7. Primitive DCS Deletion

The Delete operation is used to remove an existing primitive DCS P_o from the XSD structure whose root is T.

Algorithm 5.4. $Delete(T, P_o)$

Input: An XSD tree whose root is referenced by the pointer T and a primitive DCS P_o to be removed from the XSD tree.

Output: The XSD tree that results after the deletion of P_o.

Method: Deleting P_o results in removing all XML documents which are derived from the primitive DCS P_o.

D1 Find the leaf node containing P_o and remove P_o from the node.

D2 Update all the ancestors X of P_o by invoking $Separate(X, P_o)$.

D3 If the deletion of P_o causes underflow, the under-full node may be merged with any sibling node. The decision on what sibling node to choose may be achieved by using the containment degree. The C_d measure dictates choosing that sibling node whose merger DCS will be extended less if we add to it all the DCS entries from the under-full node. In other words, the sibling node to be chosen is the one whose merger DCS M_i yields the largest $C_d(M_i, M_r)$, where M_r is the merger of the DCSs in the under-full node.

D4 If the root node has only one child, make this child the new root of XSD.

For all the ancestors DCSs X of the primitive leaf DCS P_o, the *Delete* algorithm invokes $Separate(X, P_o)$. That is, the contribution of P_o to merger DCSs all the way up to the root is removed. This operation may cause underflow on the parent node of the primitive DCS when the number of entries becomes less than m (the minimum number of entries that can be accommodated in a node). In that case a new *Merge* operation between parent node entries may be performed. In order to make sure that the number of children of any merger DCS is between m and M, merging due to deletion may propagate up to the root. If the root has only one child, then this child becomes the new root of the XSD tree. Step D3 may cause subsequent splitting due to merging. An easy implementation of this step is to perform reinsertion of the under-full node DCS entries by invoking $Insert(T_i, D_j)$, where T_i is a pointer to an ancestor node and D_j is a DCS in the under-full node.

5.8. Deletion Performance

Deleting an existing primitive DCS requires two steps: (1) find the leaf node containing the primitive DCS and (2) remove the DCS from this location. Finding the leaf node containing the DCS is done by first comparing the DCS to the entries of the root. The entry with containment degree equal to 1 is chosen (entry completely contains the DCS in question). If such an entry does not exist, the DCS is not accommodated in this XSD tree. Then, the node that descends from this entry is visited and the DCS is compared to the entries of this node. The entry with containment degree equal to 1 is again chosen and this goes on until we reach the leaf node where the DCS resides. In the worst case we need to check all entries of a node in order to find the one that fully contains the DCS. If n is the number of entries in an XSD node, $n*(d+1)$ comparisons are needed, where d is the depth of the XSD tree. Removing the DCS from the leaf node detaches the DCS from all mergers toward the root and this causes d invocations of the *Separate* procedure. Removing a DCS from a leaf node may cause underflow, which may subsequently make sibling nodes merge. In this case, the cost of merging two nodes is given as previously described by the *Merge* operation.

6. Conclusions

The XSD approach is proposed for organizing semi-structured schemata in a hierarchical way and it may be viewed as a meta-schema organization. The XSD approach is based on clustering XML schemata rather than on classifying semi-structured sources such as XML documents. It generates meta-classes, which are called DCSs, in a dynamic way from primitive DCSs. The XML documents derived from primitive DCSs become members of these meta-classes (merger DCSs). XSD serves two purposes: (1) it is an indexing structure that XML queries may use to find the most relevant XML documents in a large XML repository; (2) it is an organization for XML schemata which allows similar schemata to be stored at the same place. A merger DCS may be viewed as a generic schema that encompasses many primitive specialized DCSs.

The XSD access method efficiently maps a query path to a set of XML documents by taking into account the structural proximity of documents and stores structurally close documents at the same place. Document retrieval is accomplished by matching the query paths against the document structure. Those

documents that match the paths (i.e., the paths are embedded in the documents) are considered relevant to the query and they are included in the result set. XSD is not based on any model and there is no need to map XML-encoded information into a data model in order to retrieve relevant documents.

In a *filter-and-refine* XML query execution, the XSD access method restricts the search to a subset of XML documents, which is usually a subspace of the entire corpus. Once the subset has been defined, the XML queries may be executed on these documents. The rationale is to avoid excessive computation where the data is likely not to satisfy the query. The advantages of the XSD structure are summarized as follows: (1) there is no need to deal with documents that do not match the XML query. This results in executing XML queries faster since the search space is limited only to the relevant XML documents. (2) The accuracy in answering XML queries is high. (3) Maintaining the XSD structure is not difficult since new XML schemata may be added or old ones deleted without performing time-consuming operations that require reorganization of the whole XSD structure. (4) XSD may be used to facilitate browsing.

Acknowledgements. Part of this research was accomplished while the author was visiting the Federal Institute of Technology (ETH), Zurich, Switzerland and the VTT Information Technology, Helsinki, Finland within the framework of the fellowship program of the European Research Consortium for Informatics and Mathematics (ERCIM).

References

Abiteboul S, Quass D, McHugh J, Widom J, Wiener JL (1997) The Lorel query language for semistructured data. International Journal on Digital Libraries 1(1): 68–88

Adelberg B (1998) NoDoSE: A tool for semi-automatically extracting semistructured data from text documents. In Haas LM, Tiwary A (eds) Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD 1998). ACM Press, Seattle, WA, pp 283–294

Atzeni P, Mecca G, Merialdo P (1997) To weave the web. In Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proceedings of 23rd international conference on very large data bases (VLDB'97). Morgan Kaufmann, Athens, pp 206–215

Baeza-Yates R, Ribeiro-Neto B (1999) Modern information retrieval (ACM Press Series). Addison-Wesley, Reading, MA

Beeri C, Tzaban Y (1999) SAL: an algebra for semistructured data and XML. In Cluet S, Milo T (eds) Proceedings of the ACM SIGMOD workshop on the web and databases (WebDB'99), Philadelphia, PA, pp 37–42

Bertino E, Guerrini G, Merlo I, Mesiti M (1999) An approach to classify semistructured objects. In Guerraoui R (ed) Proceedings of the 13th European conference on object-oriented programming (ECCOP'99). Lecture Notes in Computer Science, vol 1628. Springer, Lisbon, pp 416–440

Biron PV, Malhotra A (2000) XML schema part 2: datatypes. W3C working draft. Available at http://www.w3.org/TR/2000/WD-xmlschema-2-20000407/

Blair B, Boyer J (1999) XFDL: creating electronic commerce transaction records using XML. Computer Networks 31(11–16): 1611–1622

Bonifati A, Ceri S (2000) Comparative analysis of five XML query languages. ACM SIGMOD Record 29(1): 68–79

Bosak J (1997) XML, Java, and the future of the web. World Wide Web Journal 2(4): 219–227

Bray T, Paoli J, Sperberg-McQueen C-M (1998) Extensible Markup Language (XML) 1.0. W3C Recommendation. Available at http://www.w3.org/TR/1998/REC-xml-19980210

Buneman P (1997) Semistructured data. In Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS97). ACM Press, Tucson, AZ, pp 117–121

Buneman P, Abiteboul S, Suciu D (1999) Data on the web: from relations to semistructured data and XML. Morgan Kaufmann, San Mateo, CA

Ceri S, Fraternali P, Paraboschi S (2000) XML: current developments and future challenges for the database community. In Zaniolo C, Lockemann PC, Scholl MH, Grust T (eds) Advances in database technology: Proceedings of the 6th international conference on extending database

technology (EDBT 2000). Lecture Notes in Computer Science, vol 1777, Springer, Konstanz, pp 3–17
Christophides V, Cluet S, Simon J (2000) On wrapping query languages and efficient XML integration. In Chen W, Naughton JF, Bernstein PA (eds) Proceedings of the 2000 ACM SIGMOD international conference on management of data. ACM Press, Dallas, TX, pp 141–152
Deutsch A, Fernandez MF, Florescu D, Levy AY, Maier D, Suciu D (1999) Querying XML data. IEEE Data Engineering Bulletin 22(3): 10–18
Fallside DC (2000) XML schema part 0: primer. W3C working draft. Available at http://www.w3.org/TR/2000/WD-xmlschema-0-20000407/
Glushko RJ, Tenenbaum JM, Meltzer B (1999) An XML framework for agent-based E-commerce. Communications of the ACM 42(3): 106–114
Goldman R, Widom J (1997) DataGuides: enabling query formulation and optimazation in semistructured databases. In Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proceedings of the 23rd international conference on very large data bases (VLDB'97). Morgan Kaufmann, Athens, pp 436–445
Goldman R, McHugh J, Widom J (1999) From semistructured data to XML: migrating the lore data model and query language. In Cluet S, Milo T (eds) Proceedings of the ACM SIGMOD workshop on the web and databases (WebDB'99), Philadelphia, PA, pp 25–30
Grossman DA, Frieder O (1998) Information retrieval: algorithms and heuristics. Kluwer Academic, Dordrecht
Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages, and computation. Addison-Wesley, Reading, MA
Jain AK, Murty MN, Flynn PJ (1999) Data clustering: a review. ACM Computing Surveys 31(3): 264–323
Kilpeläinen P, Mannila H (1995) Ordered and unordered tree inclusion. SIAM Journal on Computing 24(2): 340–356
Kruskal JB (1983) An overview of sequence comparison. In Sankoff D, Kruskal J-B (eds) Time warps, string edits and macromolecules: the theory and practice of sequence comparison. Addison-Wesley, Reading, MA, pp 1–44
McHugh J, Abiteboul S, Goldman R, Quass D, Widom J (1997) Lore: a database management system for semistructured data. ACM SIGMOD Record 26(3): 54–66
McHugh J, Widom J, Abiteboul S, Luo Q, Rajaraman A (1998) Indexing semistructured data. Technical Report, Computer Science Department, Stanford University
Meltzer B, Glushko RJ (1998) XML and electronic commerce: enabling the network economy. ACM SIGMOD Record 27(4): 21–24
Mylopoulos J (2000) End-to-end E-commerce application development based on XML tools. IEEE Data Engineering Bulletin 23(1): 29–36
Nestorov S, Abiteboul S, Motwani R (1998) Extracting schema from semistructured data, In Haas LM, Tiwary A (eds) Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD 1998). ACM Press, Seattle, WA, pp 295–306
Papakonstantinou Y, Velikhov P (1999) Enhancing semistructured data mediators with document type definitions. In Proceedings of the 15th international conference on data engineering (ICDE'99). IEEE Computer Society Press, Sydney, pp 136–145
Papakonstantinou Y, Garcia-Molina H, Widom J (1995) Object exchange across heterogeneous information sources. In Yu PS, Chen ALP (eds) Proceedings of the 11th international conference on data engineering(ICDE'95). IEEE Computer Society Press, Taipei, Taiwan, pp 251–260
Robie J, Lapp J, Schach D (1998) XML Query Language (XQL). In Proceedings of the W3C Query Languages workshop (QL'98), Boston, MA. Available at http://www.w3.org/TandS/QL/QL98/pp/xql.html
Thompson HS, Beech D, Maloney M, Mendelsohn N (2000) XML schema part 1: structures. W3C working draft. available at http://www.w3.org/TR/2000/WD-xmlschema-1-20000407/
Wang JT-L, Wang X, Lin K-I, Shasha D, Shapiro BA, Zhang K (1999) Evaluating a class of distance-mapping algorithms for data mining and clustering. In Proceedings of the fifth ACM SIGKDD international conference on knowledge discovery and data mining. ACM Press, San Diego, CA, pp 307–311
Widom J (1999) Data management for XML: research directions. IEEE Data Engineering Bulletin 22(3): 44–52
Zhang K (1995) Algorithms for the constrained editing distance between ordered labelled trees and related problems. Pattern Recognition 28(3): 463–474
Zhang K, Shasha D (1989) Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal on Computing 18(6):1245–1262

Author Biography

Evangelos Kotsakis received his B.Sc. degree in computer science from the University of Athens, Greece in 1993, his M.Sc. and Ph.D degrees in engineering from the University of Salford, England in 1994 and 1998 respectively. From 1998 to 1999, he worked on space applications in the Joint Research Center, Ispra, Italy. From 1999 to 2000, he held visiting posts in the Federal Institute of Technology (ETH), Zurich, Switzerland and VTT Information Technology, Helsinki, Finland. He is currently a research associate in the Joint Research Center at Ispra, Italy. His research interests include Web data management, data warehousing, data mining and mobile systems.

Correspondence and offprint requests to: Dr Evangelos Kotsakis, Joint Research Center (CCR), Via Fermi 1, TP261, I-21020 Ispra (VA), Italy. Email: kotsakis@acm.org