Storing XML Documents in Databases

Albrecht Schmidt, Stefan Manegold, and Martin Kersten

Aalborg University, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst
al@cs.aau.dk
CWI, P. O. Box 94079, NL-1090 GB Amsterdam
First.Last@cwi.nl

Abstract

The authors introduce concepts for loading large amounts of XML documents into databases where the documents are stored and maintained. The goal is to make XML databases as unobtrusive in multi-tier systems as possible and at the same time provide as many services defined by the XML standards as possible. The ubiquity of XML has sparked great interest in deploying concepts known from Relational Database Management Systems such as declarative query languages, transactions, indexes and integrity constraints. This chapter presents now bulkloading is done in Monet XML, a main memory XML database system, and evaluates the cost of bulkloading and bulk deletion with respect to strategies which base on insertion and deletion of individual nodes. Additionally, we survey the applicability of the techniques to a wider class of XML storage schemas.

1 Introduction

Ever since the Extensible Markup Language (XML) [W3C, 1998b] began to be used to exchange data between diverse sources, interest has grown in deploying data management technology to store and query XML documents. A number of approaches propose to adapt relational database technology to store and maintain XML documents [Deutsch et al., 1999, Florescu and Kossmann, 1999, Klettke and Meyer, 2000, Shanmugasundaram et al., 1999, Tatarinov et al., 2002, O'Neil et al., 2004]. The advantage is that the XML repository inherits all the power of mature relational technology like indexes and transaction management. For XML-enabled querying, a declarative query language [Chamberlin et al., 2001] is available.

Traditionally, database technology has been offering support for processing large amounts of data. Recent research has provided valuable insights into the nature of semistructured and XML data and has attempted to integrate them into existing paradigms. However, there are still challenges that have to be met to scale XML databases up to production levels as achieved by relational engines and, thus, to gain acceptance amongst practitioners. Naturally, XML warehouses inherit the power of relational warehouses [Roussopoulos, 1997] but they also face the same challenges; in particular, update and consistency problems of materialised, replicated, and aggregated views over source data need to solved.

This chapter discusses techniques related to loading XML documents into a document warehouse. All techniques build on well-understood relational database technology and enable efficient management of large XML repositories. To get the most of relational database systems, we propose to do away with the pointer-chasing tree traversing operations, which many applications generate in the form of *edit scripts* and replace them with set-oriented operations. Edit scripts [Chawathe et al., 1996, Chawathe and Garcia-Molina, 1997] have been long known in text databases and are similar in behaviour to Document Object Model (DOM) [W3C, 1998a] traversals, which are standard in the XML world; they tend to put relational technology at a disadvantage due to their excessive use of pointer-chasing algorithms. We investigate the use of these scripts and propose alternative strategies for cases when they perform poorly.

We implemented our ideas in the XML extension of the Monet Database System [Schmidt et al., 2001, Schmidt et al., 2000]. A more detailed description of our experiments is found in [Schmidt and Kersten, 2002]. As we benchmarked the system's performance, it turns out that the use of edit-scripts is only sensible if they only update a rather small fraction of the database; once a certain threshold is exceeded, the replacement of a complete database segment is preferable. We discuss this threshold and try to quantify the trade-off for our example document database.

The application scenario which motivates our research consists of a set of XML data sources which are feature detectors that monitor multimedia data sources and analyse their

content. The detectors feed protocols of analyses into a central data warehouse. The warehouse now provides the following services: (1) insertion of a documents (a data source transmits a single protocol of an analysis to the warehouse), (2) insertion of versioned sets of documents (a set of check-out points transmits the result of a bulk analysis transcript to the warehouse), (3) deletion of documents and sets of documents (a document is deleted from the warehouse because it has become invalid or stale; duplicate analyses and erroneous insertion also happen frequently and need to be corrected), and, (4) execution of edit scripts that are transmitted from the sources and systematically correct errors in already inserted documents; for example, *a posteriori* normalisation of feature values is required frequently.

While we regard (1) as a special case of (2) and, hence, do not treat it separately, there is an obvious trade-off between a combination of (2) and (3) and the use of edit-scripts (4). More precisely, the question is: When is it cheaper to delete invalid data and re-insert a new consistent version than to use an edit script to 'patch' the warehouse? This and other questions will be dealt with in detail later.

2 Background

```
<image key="134" source="/cdrom/img1/293.jpeg">
  <date> 999010530 </date>
  <colors>
    <histogram> 0.399 0.277 0.344 </histogram>
    <saturation> 0.390 </saturation>
    <version> 0.8 </version>
  </colors>
</image>
```

Figure 1: Example document

XML documents are commonly represented as syntax trees. This section recalls some of the usual terminology we need to work with XML documents. In the sequel, **string** and **int** denote sets of character strings, respectively integers; **oid** denotes a set of unique object identifiers. Figure 1 shows an XML fragment, which is taken from the area of content-based multimedia retrieval [Schmidt et al., 1999]. Figure 2 displays the corresponding schema tree (dotted arrows indicate XML attribute relationships, straight lines XML element relationships).

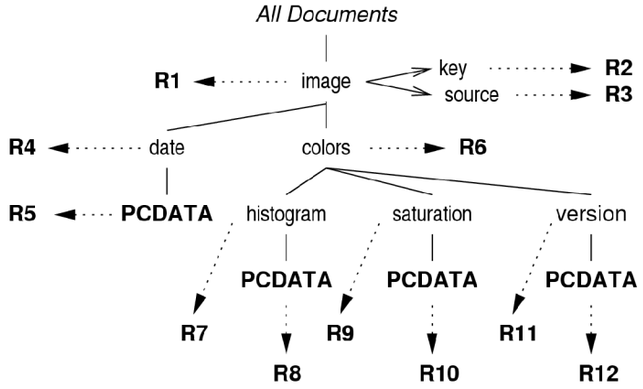

Figure 2: Schema tree of example document

Before we discuss techniques how to store a tree as a database instance, we introduce the notion of *associations*. They are used to cluster semantically related information in a single relation and constitute the basis for the Monet XML Model; the aim of the clustering process is to enable efficient scans over semantically related data, *i.e.*, data with the same element ancestry, which are the physical backbone of declarative associative query language like SQL. Different types of associations play different roles: associations of type ***oid*×*oid*** represent parent-child relationships. Both kinds of leaves, attribute values and character data, are modeled by associations of type ***oid*×*string***, while associations of type ***oid*×*int*** are used to keep track of the original topology of a document.

Paths describe the context of the element in the graph relative to the root node; we identify with *path*(*o*) the *type* of the association (·,*o*). The set of all paths in a document is called its *Path Summary*; it plays an important role in our query engine. The main rational for the path-centric storage of documents is to evaluate the ubiquitous XML path expressions efficiently; the high degree of semantic clustering distinguishes our approach from other mappings (see [Florescu and Kossmann, 1999] for a discussion). Our approach is to store all associations of the same 'type' in one *binary relation*. A relation that contains the tuple (·,*o*) is named *R*(*path*(*o*)). In Figure 2, the types or paths are the . Clustering XML elements by their type implies that we do not have to cope with many of the irregularities induced by the semi-structured nature of XML, which are typically taken care of with NULLs or overflow tables [Deutsch et al., 1999]. In the sequel, we describe the machinery we need to convert documents to Monet format and bulkload them efficiently. Also note that we are able to reconstruct the original document given this path-centric representation. A detailed discussion of the reconstruction can be found in [Schmidt et al., 2000]. We remark that we can also access the documents in an object-oriented manner, *i.e.*, object as node in the syntax tree, which is often more intuitive to the user and is adopted by standards like the DOM [W3C, 1998a]. However, we do not optimize for this as we see later.

3 XML Warehouses

3.1 Populating the XML Warehouse

There are two basic notions of interest that we are going to discuss in this section as indicated in the Introduction: Populating a database from scratch, *i.e.*, bulk load, and incremental insertion of new data into an already existing database. However, similar technology underlies both cases. Let us consider an example first. There are two standard ways of accessing XML documents: (1) A low-level event-based, called SAX [Megginson, 2001], scans an XML document for token like start tag, end tag, character data *etc*., and invokes user-supplied functions for each token that is encountered in the input. The advantage of the SAX parsers is they only require minimal resources to work. (2) The more high-level DOM interface [W3C, 1998a] provides a standard interface to parse complete documents to syntax trees. In terms of resources, the memory consumption of DOM trees is much higher, linear in the size of the document; thus, it frequently happens that large documents exceed the size of available memory. We propose a bulk load method that has only slightly higher memory requirements than SAX – *O*(*heightofdocument*) – but still keeps track of all the contextual information it needs. Thus, the memory requirements of the bulkload algorithm we use are very low as it does not materialize the complete syntax tree.

```
<image key="134" source="/cdrom/img1/293.jpeg">
<image><date>
<image><date>" 999010530 "
<image></date>
<image><colors>
<image><colors><histogram>
<image><colors><histogram>" 0.399 0.277 0.344 "
<image><colors></histogram>
<image><colors><saturation>
<image><colors><saturation>" 0.390 "
<image><colors></saturation>
<image><colors><version>
<image><colors><version>" 0.8 "
<image><colors></version>
<image></colors>
</image>
```

Figure 3: Path sequences in the example document

Since Monet XML stores complete paths, the bulk load routine has to track those paths. We do this by organizing the path summary as a schema tree which we use to efficiently map paths to relations. Each node in the schema tree represents a database relation and contains a tag name and reference to the relation. Figure 3 shows the path sequences generated by combining the SAX events of the parser and a stack in the following way. We attach OIDs to every tag when we put it on the stack. This way, we are able to record all path instances in the documents without having to maintain a syntax tree in (main) memory – an advantage that lets us process very large documents in relatively little memory. The function that performs the actual insertion is *insert*(*R*,*t*) where *R* is a reference to a relation and *t* is a tuple of the appropriate type. A first naive approach would thus result in the

following sequence of insert statements (disregarding the order in the document and whitespace due to lack of space):

1. insert(sys,(,image))
2. insert(R(image[key]),(,"134"))
3. insert(R(image[source]),(,"/cdrom/img1/293.jpeg"))
4. insert(R(image/date),(,))
5. insert(R(image/date/pcdata),(," 999010530 "))
6. insert(R(image/colors),(,))
7. insert(R(image/colors/histogram),(,))
8. insert(R(image/colors/histogram/pcdata), (," 0.399 0.277 0.344 "))
9. insert(R(image/colors/saturation),(,))
10. insert(R(image/colors/saturation/pcdata),(," 0.390 "))
11. insert(R(image/colors/version),(,))
12. insert(R(image/colors/version/pcdata),(," 0.8 "))

3.2 Database Maintenance

Once data reside in a database, maintenance of these data becomes an important issue. We distinguish between two different maintenance tasks: First, the update of existing data via *edit*-scripts for propagating changes of source data to the warehouse, and, second, the deletion and insertion of complete versions of documents which may have become stale or need to be added to the warehouse.

The concept of edit scripts to update hierarchically structured data is both intuitive and easy to implement on modern database systems [Chawathe et al., 1996, Chawathe and Garcia-Molina, 1997]. The scripts comprise three basic operations for transforming the syntax tree: *insertion* of a node, *deletion* of node, and *moving* a node. (We do not mention other operators that traverse the syntax tree, see [Chawathe and Garcia-Molina, 1997, Buneman et al., 1996].) We also view these operations as representatives for traversals that are defined in the DOM standard [W3C, 1998a]. Continuing our example, an edit script could insert additional subtrees that describe textures in the images or delete items that appear twice in the database. Typically, an edit script first pins the location of nodes to be changed; this process is often done by navigating through the syntax tree as object identifiers in the database are often not accessible to other applications. Once the location is found, the scripts then apply update, delete, and insert operations. Conceptually, an edit script may do two kinds of changes: *systematic* and *local* changes. Systematic changes may become necessary if a faulty application produced data with errors that are spread over parts of the XML document. In this case, the script traverses large parts of the syntax graph and applies similar changes at various places. In the relational context of our work, this may be an expensive restructuring process. On the other hand, if changes are only local, the script just visits a small number of nodes and patches them. This should be no resource-intensive problem, not in relational, object, or native systems.

We do not have the space to discuss edit scripts in depth here and refer the reader to the above citations. However, we demonstrate their use with an example similar to that used in the performance discussion. Consider again Figure 1. A systematic change would, for example, require us to change all `dates` from Unix system time, *i.e.*, seconds since January 1 1970, to a more human readable format. The way we go about creating the appropriate edit script is the following: We look up all associations which assign a value to an attribute *unit*. Then, for all these nodes, we calculate the new date and replace the old one. Techniques for constructing automata that do the traversal can be found, *e.g.*, in [Neven and Schwentick, 1999]. Once such an automaton finds a node *n* that needs to be updated, it executes an *update*(*n*,*date*,*newdateformat*) statement. On the physical data model of Section 2, this is translated into a command that replaces the value of the respective association.

The point that is important for us is that edit scripts traverse parts of the XML syntax graph and manipulate individual nodes. This is in stark contrast to the second method mentioned above, bulk deletion and re-insertion where we delete a complete segment of the database and re-insert a corrected version. In the example scenario, this means that an individual detector re-sends the corrected version of a previously submitted document instead of a patching edit script. Generally, the underlying assumption is that the aforementioned data sources provide the capability of sending both, the edit-script and a complete updated document; however, this assumption holds for many practical applications as well as for our example: a detector may either send an edit script or re-transmit a corrected version of the complete document.

It is straight-forward to design an algebraic algorithm that does the deletion and visits every node at most once like a linear scan [Schmidt and Kersten, 2002]. Still, we need to

discuss when to use bulk deletion combined with re-insertion and when to use edit scripts. The next section looks quantitatively at when to go for what.

3.3 Performance Impressions

This section presents performance impressions of a data warehouse [Kersten and Windhouwer, 2000] containing actual features which are more elaborate but similar to the ones used in the running example. The data warehouse uses Monet XML as the physical storage model. A 500 MHz Pentium-class PC running Redhat Linux was our experimentation platform.

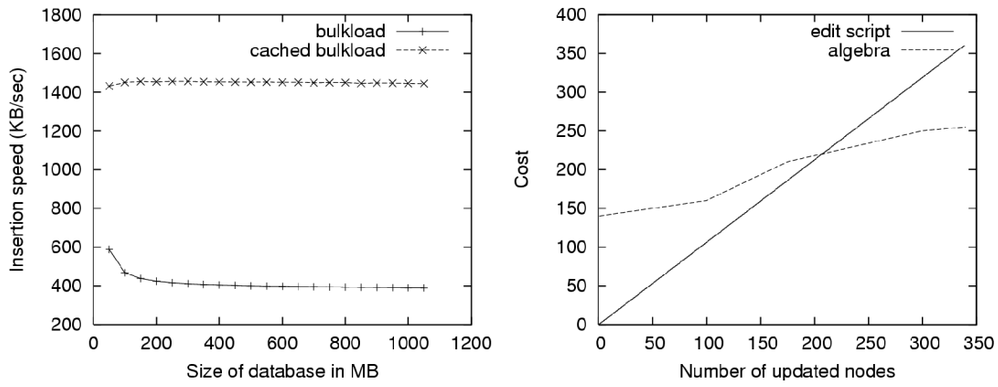

Figure 4: Performance of loading and updating data

Figure 4 shows the relationship between database size and insertion speed. The figure displays the speedup of an optimized approach with caching over a naive implementation. As one might expect the insertion into an empty database is faster than into an already densely populated one if no intelligent caching is used. As the database gets larger, insertion speed converges to a ratio of about 390 KB/sec. If schema trees are used, bulk load speed more than triples showing the potential of this technique, which has been explained in Section 3.1. Note that neither bulk load method blocks the database; both operate interactively and do not interfere with the transaction system. Note that the insertion performance in Figure 4 includes converting the textual representation of a document to executable database statements and, thus, random memory accesses (which can be alleviated with path caching), whereas deletion can be done as sequential scans.

Updates are considered in Figure 4 as well. Edit scripts as presented in Section 3.2 are run against the database created in the previous experiment; the updates they apply are systematic. The figure plots the performance of edit scripts against a simple algebraic bulk replacement strategy which consists of deleting those documents that need updating from the database and subsequently loading updated documents into the database. With respect to when to choose which technique, the two lines in Figure 4 show that once more than approximately 220 entries are changed by the edit script, one should consider reverting to bulk operations for performance reasons. The threshold of 220 entries is surprisingly low; however, one should keep in mind that relational databases are not optimized for pointer-chasing operations. We remark that this threshold also depends on the characteristics of the XML document, especially on the ratio between text and mark-up.

3.4 Other Storage Mappings

This section discusses some opportunities and limitations of the methods discussed in this chapter. It takes up ideas found in the literature and relates them to the techniques we developed for Monet XML.

3.4.1 Positional Mappings and Extent Mappings

Recently, the idea of using the OIDs located on a path as components of a positional number system like the Dewey system caught on [Tatarinov et al., 2002, O'Neil et al., 2004]. We can easily adjust the mapping algorithms which produce the Monet mapping to include the whole list of OIDs which are on the stack rather than just include the two lowest ones as the algorithm in Section 3.1 does. This just requires us to look at the complete stack before producing an insertion statement. This kind of positional mapping is especially suited for implementation of query primitives based on tree automata [Comon et al., 1997] since many of the query primitives like axis navigation can be translated transparently to state transitions [Ludäscher et al., 2002].

Another type of mapping is called *extent mapping*. In extent mappings, not only the element OID is recorded but also the range in which the descendant node OIDs can be

found; an example of such a mapping can be found in [Zhang et al., 2001]. The insertion algorithm can be adapted by delaying the generation of an insert statement until not the start tag of an element but its end tag is encountered. This way, enough information can be gathered about the position of the end tag.

3.4.2 Semantic Mappings

A *semantic mapping* in our context is a mapping which takes into account data semantics. For example, a mapping might decide to map an XML element to an instance of a particular datatype. Semantic mappings generally follow two mutually exclusive principles. First, they may analyse schema information such as given by DTDs [Shanmugasundaram et al., 1999] or XML Schema and generate a database schema with features that imitate those found in E/R mappings. This kind of mapping is fully automatic. Second, users may manually specify application-specific constraints that help map document fragments to tuples in relational databases. These mappings try to overcome the semantic gap between XML and relational databases by equipping users with a language to specify which parts of a set of documents correspond to which part of a database schema.

Although semantic mappings are more complex to implement in a streaming manner like the Monet mapping where only a stack is needed to keep track of all necessary information, we can overcome much of the complexity by falling back to conceptually simpler mappings like the Monet mapping or a positional mapping. This is done by implementing the mapping as a two-step process. In the first step, the Monet mapping of an incoming XML document is used to derive database relations. In the second step, these relations are combined in database queries to form the semantic entities produced by the semantic mapping. This abstraction can greatly facilitate the complexity of the mapping software.

4 Future Trends

In the future, we are likely to witness a tighter integration of the two paradigms which dominate XML processing: data management and information retrieval. This is likely to pose new challenges with respect to database maintenance and indexing; it is also likely to necessitate new data structures as well as novel query processing and update strategies and will thus require adaptations of the strategies presented in this chapter.

5 Conclusion

This chapter discussed performance considerations for typical problems in relational XML document data warehousing, especially the trade-off between algebraic and pointer-chasing algorithms for updates. For practical purposes, it turned out that it often is better to replace a complete database segment and re-insert the updated data than to patch an existing version with expensive edit-scripts. In particular, our experiments showed that once the patched data volume exceeds a small percentage of the database, one should resort to bulk replacement. For good insertion performance, the use of schema trees has been beneficial.

References

[Buneman et al., 1996] Buneman, P., Davidson, S. B., Hillebrand, G. G., and Suciu, D. (1996). A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 505–516, Montreal, Canada.

[Chamberlin et al., 2001] Chamberlin, D., Florescu, D., Robie, J., Siméon, J., and Stefanescu, M. (2001). XQuery: A Query Language for XML. available at http://www.w3.org/TR/xquery as of 22 March.

[Chawathe et al., 1996] Chawathe, A., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change Detection in Hierarchically Structured Information. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 493–504.

[Chawathe and Garcia-Molina, 1997] Chawathe, S. and Garcia-Molina, H. (1997). Meaningful Change Detection in Structured Data. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 26–37.

[Comon et al., 1997] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (1997). Tree automata techniques and applications. available at http://www.grappa.univ-lille3.fr/tata. (released 1 October 2002) as of 22 March.

[Deutsch et al., 1999] Deutsch, A., Fernandez, M. F., and Suciu, D. (1999). Storing Semistructured Data with STORED. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 431–442, Philadephia, PA, USA.

[Florescu and Kossmann, 1999] Florescu, D. and Kossmann, D. (1999). Storing and Querying XML Data Using an RDBMS. *Data Engineering Bulletin*, 22(3).

[Kersten and Windhouwer, 2000] Kersten, M. and Windhouwer, M. (2000). *DMW Homepage*. Available at http://www.cwi.nl/~acoi/DMW/ as of 22 March.

[Klettke and Meyer, 2000] Klettke, M. and Meyer, H. (2000). XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *International Workshop on the Web and Databases (WebDB)*, pages 63–68.

[Ludäscher et al., 2002] Ludäscher, B., Mukhopadhyay, P., and Papakonstantinou, Y. (2002). A Transducer-Based XML Query Processor. In *Proceedings of the International Conference on Very Large Data Bases*, pages 227–238.

[Megginson, 2001] Megginson, D. (2001). SAX 2.0: The Simple API for XML. http://www.megginson.com/SAX/ as of 22 March.

[Neven and Schwentick, 1999] Neven, F. and Schwentick, T. (1999). Query Automata. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205–214.

[O'Neil et al., 2004] O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N. (2004). ORDPATHs: Insert-Friendly XML Node Labels. available at http://www.cs.umb.edu/~poneil/ordpath.pdf as of 22 March.

[Roussopoulos, 1997] Roussopoulos, N. (1997). Materialized Views and Data Warehouses. In *Proceedings of the 4th KRDB Workshop*. available at http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-8/ as of 22 March.

[Schmidt and Kersten, 2002] Schmidt, A. and Kersten, M. (2002). Bulkloading and Maintaining XML Documents. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 407–412.

[Schmidt et al., 2001] Schmidt, A., Kersten, M., and Windhouwer, M. (2001). Querying XML Documents Made Easy: Nearest Concept Queries. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 321–329.

[Schmidt et al., 2000] Schmidt, A., Kersten, M., Windhouwer, M., and Waas, F. (2000). Efficient Relational Storage and Retrieval of XML Documents. In *International Workshop on the Web and Databases*, pages 47–52, Dallas, TX, USA.

[Schmidt et al., 1999] Schmidt, A., Windhouwer, M., and Kersten, M. L. (1999). Feature Grammars. In *Proc. of the Int'l. Conf. on Information Systems Analysis and Synthesis*, Orlando, Florida.

[Shanmugasundaram et al., 1999] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., and Naughton, J. (1999). Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 302–314, Edinburgh, UK.

[Tatarinov et al., 2002] Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E., and Zhang, C. (2002). Storing and querying ordered XML using a relational database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–215.

[W3C, 1998a] W3C (1998a). Document Object Model (DOM). available at http://www.w3.org/DOM/ as of 22 March.

[W3C, 1998b] W3C (1998b). Extensible Markup Language (XML) 1.0. available at http://www.w3.org/TR/1998/REC-xml-19980210 as of 22 March.

[Zhang et al., 2001] Zhang, C., Naughton, J., DeWitt, D., Luo, Q., and Lohman, G. (2001). On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

Terms

XML. The eXtensible Markup Language as defined at http://www.w3.org/XML/.

Document Databases. A collection of documents with a uniform user interface.

Document Warehouses. A document database consisting of documents gathered from various independent sources.

Database Maintenance. The task of updating a database and enforcing constraints.

Relational Databases. Widely used variety of databases based on the relation algebra by Codd.

Bulkload. Adding a (large) set of data to a database rather than individual tuples.

Edit script. A set of instructions that walk and update a document database node by node.