Bulkloading and Maintaining XML Documents

Albrecht Schmidt and Martin Kersten
CWI
P. O. Box 94079
NL-1090 GB Amsterdam
First.Last@cwi.nl

ABSTRACT

The popularity of XML as a exchange and storage format brings about massive amounts of documents to be stored, maintained and analyzed – a challenge that traditionally has been tackled with Database Management Systems (DBMS). To open up the content of XML documents to analysis with declarative query languages, efficient bulk loading techniques are necessary.

Database technology has traditionally been offering support for these tasks but yet falls short of providing efficient automation techniques for the challenges that large collections of XML data raise. As storage back-end, many applications rely on relational databases, which are designed towards large data volumes. This paper studies the bulk load and update algorithms for XML data stored in relational format and outlines opportunities and problems. We investigate both (1) bulk insertion and deletion as well as (2) updates in the form of edit scripts which heavily use pointer-chasing techniques which often are considered orthogonal to the algebraic operations relational databases are optimized for. To get the most out of relational database systems, we show that one should make careful use of edit scripts and replace them with bulk operations if more than a very small portion of the database is updated.

We implemented our ideas on top of the Monet Database System and benchmarked their performance.

Keywords. XML, Document Databases, Document Warehouses, Maintenance, Relational Databases

1. INTRODUCTION

The Extensible Markup Language (XML) [19] is extensively used as a data exchange and storage format. However, due to the lack of query engines that go beyond search engine functionality the massive amounts of XML data produced by today's applications often escape attempts to disclose them for analysis and maintenance. While it is certainly possible to convert XML data to other formats for which solutions exists, from a software engineering point of view it would be preferable to go for 'all XML' solutions. A viable approach to achieving this goal is to adapt relational database technology to store and maintain XML documents such as proposed in, *e.g.*, [7, 8, 10, 17]. The advantage of this approach is that the XML repository inherits all the power of mature relational technology like indexes, transaction management *etc.* As a first step towards this goal several declarative query languages [4] and data models have been proposed.

Traditionally, database technology has been offering support for processing large amounts of data. Whereas there has been considerable research into query languages and logical data models for XML data [1, 3], there have only been few proposals to tackle the problem of extending current technology to cope with the needs of applications that rely on intensive usage of XML resources. Recent research has provided valuable insights into the nature of semistructured and XML data and has positioned them in the database field. 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 [13] but they also face the same challenges; in particular, update and consistency problems of materialized replicated and aggregated views over source data need to solved.

As a step towards making XML the language of all web databases, we propose a framework that builds on well-understood relational database technology and enables 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 and replace them with set-oriented operations: many applications generate updates in the form of edit scripts. Edit scripts [5, 6] have been long known in text databases and are similiar in behavior to Document Object Model (DOM) [18] traversals, which are standard in the XML world; they clearly disadvantage relational technology 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 [14, 15] and benchmarked their 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.

SAC 2002, Madrid, Spain

The application scenario which motivates our research consists of a set of XML data sources, feature detectors that monitor multimedia data sources and analyze their content; they feed their protocols of the analysis 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* normalization of feature values is required frequently.

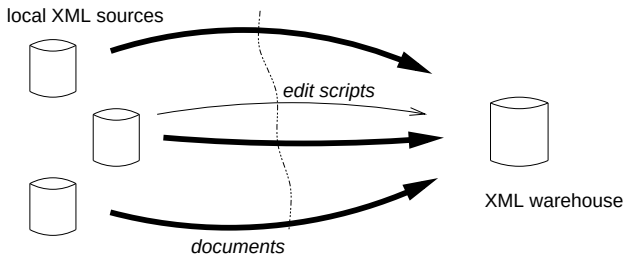

Figure 1: Application Scenario

While (1) is just a special case of (2) and hence is not treated separately in this paper, 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.

The rest of this paper is organized as follows: Section 2 introduces the experimental and theoretical framework; Section 3 describes the bulk loading techniques used to populate a database. We then discuss how edit scripts and bulk deletion are applied to document databases and assess their performance quantitatively. The last section summarizes the results and outlines future work.

2. PRELIMINARIES

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 and integers and **oid** a set of unique object identifiers. We can now define a XML document formally (a/b denotes that b is a child element or descendant of a, $a[b]$ means that b is an attribute of a, see [19] for details):

```
<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 2: Example document

DEFINITION 1. *An* XML document *is a rooted tree* $d = (V, E, r, label_E, label_A, rank)$ *with nodes* V *and edges* $E \subseteq V \times V$ *and a distinguished node* $r \in V$*, the root node. The function* $label_E : V \rightarrow$ **string** *assigns string labels to nodes, i.e., element denominations.* $label_A : V \rightarrow$ **string** $\rightarrow$ **string** *assigns pairs of strings, attributes and their values, to nodes. Character Data (CDATA) are modeled as a special attribute of cdata nodes,* $rank : V \rightarrow$ **int** *establishes a ranking to allow for the textual order among sibling nodes.*

Figure 2 shows an XML fragment, which is taken from the area of content-based multimedia retrieval [16]; Figure 3 displays the corresponding tree (arrows indicate XML attribute relationships, straight lines XML element relationships).

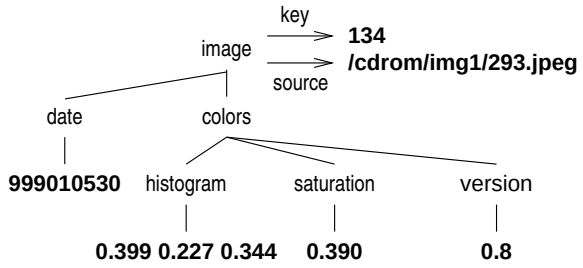

Figure 3: Syntax tree

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 relations 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.

DEFINITION 2. *A pair* $(o, \cdot) \in$ (**oid** $\times$ **oid** $\cup$ **oid** $\times$ **string** $\cup$ **oid** $\times$ **int**) *is called an* association.

The different types of associations play different roles: associations of type **oid** $\times$ **oid** represent parent-child relationships. Both kinds of leaves, attribute values and character data are modeled by associations of type **oid** $\times$ **string**, while associations of type **oid** $\times$ **int** are used to keep track of the original topology of a document.

DEFINITION 3. *For a node* o *in the syntax tree, we denote the sequence of labels along the path (vertex and edge labels) from the root to* o *as* $path(o)$.

Paths describe the position of the element in the graph relative to the root node and we also use $path(o)$ to denote the *type* of the association $(\cdot, o)$. The set of all paths in a document is called its *Path Summary*, which plays a central 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 achieved distinguishes our approach from other mappings (see [8] for a discussion). Our approach is to store all associations of the same 'type' in one *binary relation*. A relation that contains the tuple $(\cdot, o)$ is named $R(path(o))$. We can now define the mapping.

DEFINITION 4. *Given an XML document d, the* Monet transform *is a quadruple* $M_t(d) = (r, \mathbf{E}, \mathbf{A}, \mathbf{T})$ *where r remains the root of the document and*

$$\mathbf{E} = \bigcup_{(o_i, o_j, s) \in \tilde{E}} R(path(o_i)/s)\langle o_i, o_j \rangle,$$

$$\mathbf{A} = \bigcup_{(o_i, s_1, s_2) \in label_A} R(path(o_i)[s_1])\langle o_i, s_2 \rangle,$$

$$\mathbf{T} = \bigcup_{(o_i, i) \in rank} R(path(o_i)[rank])\langle o_i, i \rangle\big);$$

r is the root of the document, $\mathbf{E}$ *the relations that describe element relationships,* $\mathbf{A}$ *those for attributes, and* $\mathbf{T}$ *records the topology among elements.*

Encoding *path* to a component into the name of the relation achieves a significantly higher degree of semantic clustering than implied by plain data guides [9]. In other words, we use *path* to group semantically related associations. A direct consequence of the decomposition schema is that we do not need to cope with irregularities induced by the semi-structured nature of XML, which are typically taken care of with NULLs or overflow tables [7]. The rest of this paper will now deal with 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 its Monet transform:

PROPOSITION 1. *The above mapping is lossless, i.e., for an XML document d there exists an inverse mapping* M_t^{-1} *such that d and* $M_t^{-1}(M_t(d))$ *are isomorphic.*

A discussion of the inverse mapping can be found in [15]. The Monet transform also enables an object-oriented perspective, *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 [18].

DEFINITION 5. *An object o is a set of strong and weak associations* $\{A_1\langle o, o_1\rangle, A_2\langle o, o_2\rangle, \dots\}$. *A strong association is an association that is present in every object of the same type (i.e., path); a weak association is an association that is* not *present in every object of the same type.*

This perspective is used when we need to DOM-like traversals or run edit-scripts against the database.

3. 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,

```
1  <image key="134" source="/cdrom/img1/293.jpeg">
2  <image><date>
3  <image><date>" 999010530 "
4  <image></date>
5  <image><colors>
6  <image><colors><histogram>
7  <image><colors><histogram>" 0.399 0.277 0.344 "
8  <image><colors></histogram>
9  <image><colors><saturation>
10 <image><colors><saturation>" 0.390 "
11 <image><colors></saturation>
12 <image><colors><version>
13 <image><colors><version>" 0.8 "
14 <image><colors></version>
15 <image></colors>
16 </image>
```

Figure 4: Path sequences in the example document

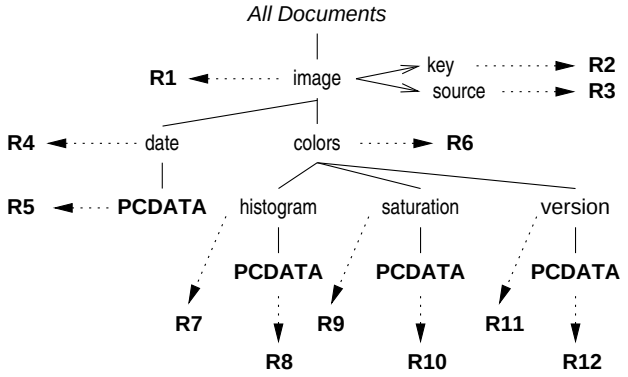

Figure 5: Schema tree of example document

and incremental insertion of new data into an already existing database. However, we use the same technique for 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 [11], which scan a XML document for token like start tag, end tag, character data *etc.*; user supplied functions are called on encountering for each type of token. The advantage of the SAX parsers is they only require minimal resources to work efficiently. There is also a high-level DOM interface [18] which provides a standard interface to parse trees of complete documents. In terms of resources, the memory consumption of DOM trees is much higher, linear in the size of the document; thus, it may happen that large documents exceed the size of available memory. In this chapter we propose a bulk load method that has only slightly higher memory requirements than SAX – $O(height\ of\ document)$ – but still keeps track of all the contextual information it needs and which would otherwise only available through a DOM interface. Thus, the memory requirements of the bulkload algorithm we use are very low as it does not materialize the complete syntax tree to generate insertion statements.

Since Monet XML stores complete paths, the bulk load

routine need to track those paths. We do this by organizing the path summary as a *schema tree* which we use to map efficiently paths to relations. Each node in the schema tree represents a database relation and contains a tag name and reference to the relation. Figure 4 shows the path sequences generated by combining the SAX events of the parser and a stack.

We can now 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 amounts of 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 due to lack of space):

1. $insert(sys, \langle o_1, image\rangle)$
2. $insert(R(image[key]), \langle o_1, \text{“}134''\rangle)$
3. $insert(R(image[source]), \langle o_1, \text{“}/cdrom/img1/293.jpeg''\rangle)$
4. $insert(R(image/date), \langle o_1, o_2\rangle)$
5. $insert(R(image/date/pcdata), \langle o_2, \text{“}999010530''\rangle)$
6. $insert(R(image/colors), \langle o_1, o_3\rangle)$
7. $insert(R(image/colors/histogram), \langle o_3, o_4\rangle)$
8. $insert(R(image/colors/histogram/histogram), \langle o_4, \text{“}0.3990.2770.344''\rangle)$
9. $insert(R(image/colors/histogram/pcdata), \langle o_4, \text{“}0.3990.2770.344''\rangle)$
10. $insert(R(image/colors/saturation), \langle o_3, o_5\rangle)$
11. $insert(R(image/colors/saturation/pcdata), \langle o_5, \text{“}0.390''\rangle)$
12. $insert(R(image/colors/version), \langle o_3, o_6\rangle)$
13. $insert(R(image/colors/version/pcdata), \langle o_6, \text{“}0.8''\rangle)$

Note that this sequence of insert statements requires us to hash the *complete* path to a relation name. By exploiting the hierarchical structure of the schema tree we can do much better. So we now address the question how to map the paths efficiently to relations. We can do away with much of the hashing if keep track of the context, *i.e.*, the current node, in the schema tree: when we encounter a start tag, we look at the sons of the current context. There are now two cases: (1) we find a son that represents the tag, or, (2) there is no son that represents the tag. In the first case, we simply push the son on the stack, thus making it the current context, and store the OIDs in the relation that is associated with the son. If we don't find a child node that represents the tag, then the path does not yet exist in the database. In this case, we create a node and the respective relation and continue processing with the newly created node as in (1). If we encounter an end tag we 'pop' the stack twice, *i.e.*, pop both the start and corresponding end tag. The performance analysis at the end of this paper quantifies the improvement this simple trick brings about.

We note that we can easily extend the bulkload procedure to records *extents* of elements, *i.e.*, the textual position of a start tag and its corresponding end tag. In [20], the authors present such a schema to improve the performance of containment queries. We can also use the extent mechanism to implement a multi-attribute schema for documents which come along with a DTD by reserving slots for every $1:1$ parent-child relationship specified in the DTD and flushing tuples once the end of their extent is reached.

4. MAINTAINING THE DATABASE

Once data reside in a database, maintenance of these data becomes an important issue. In this paper, 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; it is defined in [5, 6]; the scripts comprise four basic operations (we do not mention other operators that traverse the syntax tree, see [6, 3]) for transforming the syntax tree:

1. $insert(n, f, k)$ add a leaf n as kth to node f,
2. $delete(n)$ remove a leaf n,
3. $update(n, a, v)$ change the attribute a of node n to v,
4. $move(n, m, k)$ a node n into the position of the kth son of m.

We also view these operations as representatives for traversals that are defined in the DOM standard [18]. Note that [5] do not assume the presence of object identifiers; in our case, these identifiers are provided by either the database or the source data (or both) so that we can make use of this feature at no cost. Following 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 edit script traverses large parts of the syntax graph and applies changes. 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 2. A systematic change would, for example, require us to change all `date`s 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 [12]. Once such an automaton finds a node n that needs to be updated, it executes an $update(n, date, new\ date\ format)$ 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 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 retransmit a corrected version of the complete document. Additionally, all data items items have a unique identifier which then can be used as an orientation to replace the automaton that guides the edit script by algebraic joins which were been shown to have a more efficient execution model [15].

The algebraic algorithm that deletes a complete database segment with root r looks as follows:

```
razor (relation roots)
  o_1, ..., o_n := offspring_relations(roots)
  foreach o ∈ {o_1, ..., o_n} do
    if (offspring(o) ≠ ∅) do
      razor(offspring(o))
    end
  end
  delete(o_1, ..., o_n)
end
```

Note that this algorithm is efficient because it visits every node once in a breadth-first search like manner, imitating a single scan over the relevant parts of the document (for simplicity, we left out the deletion of rank information). The complementary question, how to translate an edit script into algebraic insert question, is rather straight forward: the trick is to dump those parts of the database that are to be inserted or updated into relations and then add those relations to the database.

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.

5. QUANTITATIVE ANALYSIS

This section presents performance impressions of a data warehouse containing actual feature more elaborate but similar to the ones used in the example. The data warehouse uses the physical storage model of Section 2; thus, our results may need slight modifications if applied to systems that use other data models. However, we believe that relational database management systems should behave in a similar manner as our implementation on top of the relational Monet database kernel [2].

Figure 6 displays the relationship between database size and insertion speed. The figure displays the naive approach and the optimization with the schema tree. 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 that triples showing the potential of this technique, which has been explained in Section 3.

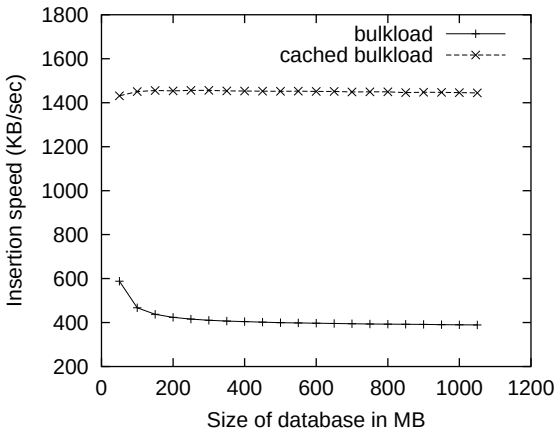

Figure 6: (incremental) Bulkload performance

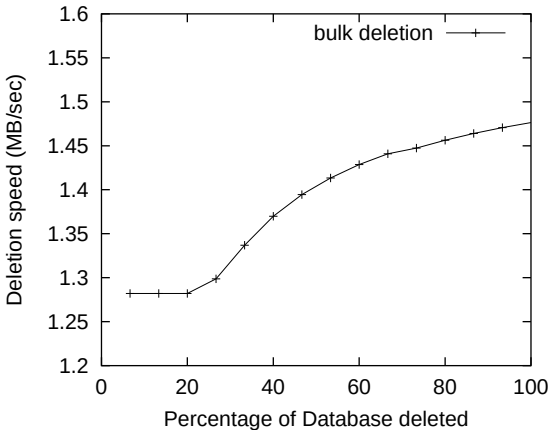

Figure 7: Bulk deletion

Note that neither bulk load method blocks not block the database; both operate interactively and do not interfere with the transaction system.

Bulk deletion is assessed in Figure 7. The algorithm presented in Section 4 is run against the database created in the previous experiment. Each run, segments of around 55 MB are deleted. Note that the insertion performance in Figure 6 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.

Eventually, with respect to when to choose which technique, the two lines in Figure 8 show that once more then 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 suprisingly low; however, one should keep in mind that relational databases are not optimized for pointer-chasing operations. We also remark that the threshold also depends on the characteristics of the XML document, especially on the ratio between text and mark-up. Nevertheless, it does not vary greatly for different types of documents.

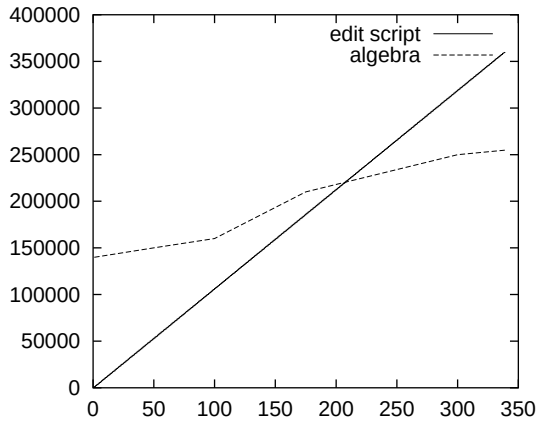

Figure 8: Edit scripts versus bulk deletion and insertion

6. CONCLUSION AND FUTURE WORK

This paper discussed performance considerations for typical problems in relational XML document data warehousing, especially the trade-off between algebra and pointer-chasing algorithms. 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.

Concerning future work, we concentrate on developing a cost model for choosing automatically when to use which update technique, edit scripts or their algebraic equivalents. We are also looking at how to provide efficient versioning and replication support.

7. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, 1999.

[3] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. 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, 1996.

[4] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery: A Query Language for XML, February 2001. available at `http://www.w3.org/TR/xquery`.

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

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

[7] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 431–442, Philadephia, PA, USA, 1999.

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

[9] R. Goldman and J. Widom. Approximate DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, 1999.

[10] M. Klettke and H. Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. pages 63–68, 2000.

[11] D. Megginson. SAX 2.0: The Simple API for XML. `http://www.megginson.com/SAX/`, 2001.

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

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

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

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

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

[17] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. 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, 1999.

[18] W3C. Document Object Model (DOM). available at `http://www.w3.org/DOM/`, 1998.

[19] W3C. Extensible Markup Language (XML) 1.0. available at `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

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