Application-Tailored XML Storage

Maxim Grinev, Ivan Shcheklein

Institute for System Programming of the
Russian Academy of Sciences

maxim@grinev.net, shcheklein@ispras.ru

Abstract

Several native approaches to storing large XML data sets exist. In all of these approaches the internal data representation is designed to support any ad-hoc XQuery query. In this paper we argue that XQuery and its data model are too universal and any one-size-fits-all XML representation leads to significant overheads in terms of representation size and complexity.

Based on the consideration that in many applications queries/updates workload is known in advance and does not change often, we propose an application-tailored XML storage. Elimination of the superfluous XQuery data model features and utilization of the various physical data representations improve performance on the specified workload, while ad-hoc queries support can be limited.

1 Introduction

XML/XQuery [1] provides great flexibility and extensibility. It is a universal model to represent data ranging from relational-like to content-oriented (including mixed content). XML applications are quite extensible as XML/XQuery can handle irregularity in data. However, this flexibility/extensibility leads to inefficiency. There are a number of general approaches [2, 3, 4, 5] that tackle the issue. However each approach has its own obvious advantages and disadvantages that make it applicable only for particular type of applications (see comparison with the approach in related work below).

Moreover, in any of the approaches there are a lot of features that are redundant for any particular applications. For example, suppose we have relational-like data stored in XML. When we query such data we don't usually use features like sibling/parent axes or document order which are supported in all general approaches. Another example is a set of queries to content-oriented XML (such as an encyclopedia article [7] or Microsoft Word XML format [6]). Such queries do not usually address rendering elements (such as para, bold, emphasize, etc which constitute the majority in content-oriented XML) but address meaningful semantic elements such as: author, date, bibliography, etc., so rendering elements can be stored in a compressed unqueryable form to improve the efficiency of updates and serialization.

We believe that efficient storage and processing of XML data cannot be achieved using any general approach. The only approach to achieve great efficiency for such a universal and flexible model as XML/XQuery is to choose appropriate data structures and processing techniques for a given application (i.e. given XML schema and set of queries and updates). We need to go beyond compiling query execution plans and compile an XML storage tailored for a given workload of queries/updates. This approach allows us to support flexible XQuery model at logical level but eliminate the XQuery data model overhead at physical level. For example, querying/updating relational-like data (even more: nested-table data) can have efficiency comparable with that provided by relational databases. Content-oriented data can be processed with efficiency that is comparable with pure text-oriented systems.

In this paper we summarize the preliminary results of in-progress research on building application-tailored XML storage.

1.2 Paper Outline

The rest of the paper is organized as follows. Within the next section we consider an example of the application which motivates this work. In Section 3 we give brief overview of the approach proposed. In Section 4 we propose physical data representation and illustrate it on example. Within the Section 5 we survey related work and consider existing approaches. And finally, Section 6 concludes and points out directions of our future work.

2 Motivating Example

To illustrate advantages and various aspects of the application-tailored XML storage we use simplified version of the Great Russian Encyclopedia (GRE) application [7]. Figures 1-2 show fragment of the encyclopedia XML and its descriptive schema (by definition, every path of the document has exactly one

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2008

```
<volume>
  <article id="2">
    <title>Cyclotron resonance</title>
    <authors>
      <author>Century S.Edelman.</author>
      <author>I. Kaganov</author>
    </authors>
    <body>
      <p>
        <b>Cyclotron resonance</b> Selective absorption of electromagnetic...
        <link idref="1">Effective weight</link>
        <p> ...  <i> ... </i> ... <b> ... </b> ... </p>
        <link idref="6">Lorentz force</link>...
      </p>
    </body>
  </article>
  ...
  <article id="3">
    <title>Dorfman Jacob Grigorevich</title>
    <authors>
      <author>I. Ivanov</author>
    </authors>
    <body>
      <p>
        <b>Dorfman Jacob Grigorevich</b> the Soviet physicist, the doctor...
        <link idref="2">Cyclotron resonance</link>
        ... <i> ... </i>...
      </p>
    </body>
  </article>
  ...
  <article id="1">
    <title>Effective weight</title>
    <authors>
      <author>I. Kaganov</author>
    </authors>
    <body> ... </body>
  </article>
   </volume>
```

Figure 1: Great Russian Encyclopedia Fragment.

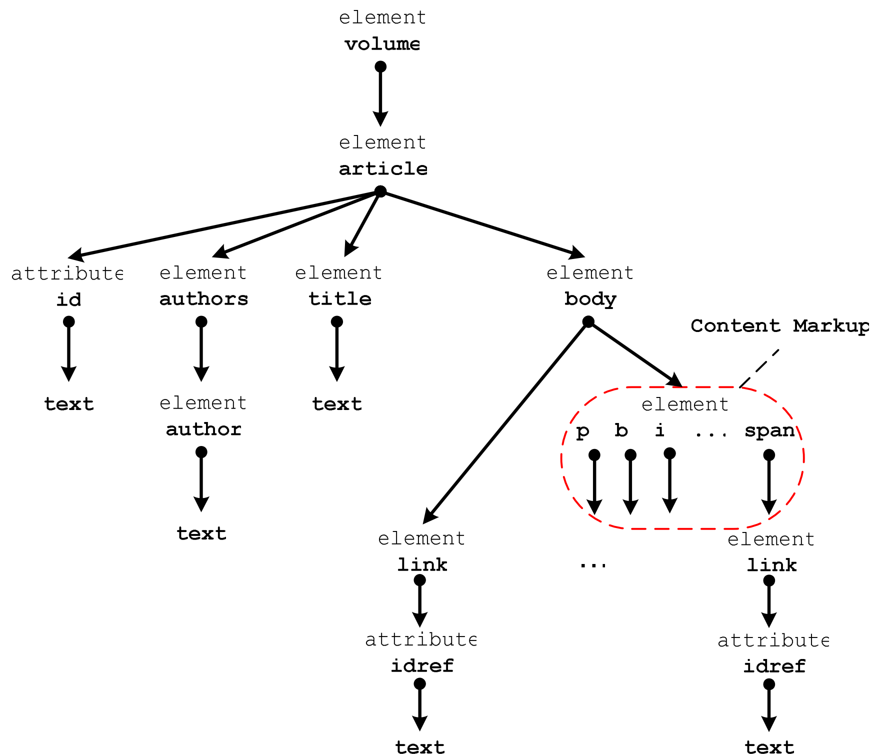

Figure 2: Great Russian Encyclopedia Descriptive Schema

path in the descriptive schema, and every path of the descriptive schema is a path of the document). In the example there is one volume that contains at least three articles. Each article in turn consists of the title, list of authors and body element contained content of the article.

For data processing encyclopedia application uses a number of predefined XQuery queries which are not likely to change often in the production system. Let us consider some of them (Q1-Q5):

(**Q1**): *List all articles' titles.*

```
declare ordering unordered;
volume/article/title
```

(**Q2**): *Get article by id.*

```
declare ordering unordered;
volume/article[@id eq "..."]
```

(**Q3**): *Get article by title.*

```
declare ordering unordered;
volume/article[title eq "..."]
```

(**Q4**): *List articles referenced from the article "1".*

```
declare ordering unordered;
for $i in volume/article
          [@id eq "1"]//link
return volume/article
       [@id eq $i/@idref]/title
```

(**Q5**): *List articles which have references to the article 'atom'.*

```
declare ordering unordered;
let $j := volume/article
          [title eq "atom"]/@id
for $i in volume/article
where $i//link[@idref eq $j]
return $i/title
```

Considering this simplified example we can point out some interesting observations concerned workload and XML data which we believe are more or less common to every XML processing application:

1. *Rendering Markup Content* - content part of the XML data usually contains a lot of rendering elements (e.g. HTML in Figures 1-2) which only aim are to be used in front-end applications (like browsers or Word processors) to display proper image on the screen and they are not used directly in queries. Often rendering markup language uses XML syntax and produces additional stress on XML database since it can't distinguish rendering elements and elements which reflect application-level entities (articles, persons, etc) and extensively used in application defined queries.

2. *Relational Like Content* - besides rendering elements we can single out attributes and elements with simple content (e.g. `id`, `idref` attributes and `title` element in the example) which are intended to be used in queries just as properties of the application-level entities. For example, `id` of the article is not interested for us itself; however we are interested in article with some `id`.

3. *Document Ordering Avoidance* - quite often document order of the result doesn't make sense for the application. Therefore in a lot of cases parent-child relationships are the only relationship we actually need between certain entities. In queries Q1 - Q5 we use standard XQuery prolog declaration to turn off results ordering.

4. *Known Workload* - we can derive a number of quite simple path expressions which play a role of basis for all queries in application. In our example we have `volume/article`, `volume/article/link`, `volume/article/title` and some modifications with predicates.

5. *Fixed Workload* - finally, just as in the GRE application we suppose that basic queries which are used in production systems are very rare subjected to be changed. We do not have ad-hoc queries but a number of well defined, possibly parameterized expressions.

As the paper progresses we will illustrate how these observations can be used to adjust XML storage for the specified queries.

3 Approach Overview

The approach we employ can be split into two distinct phases: storage compilation for the initial workload and recompilation phase in which storage is reorganized to restore good performance after the workload changed. Below is a brief description of each phase from the logical point of view. A sketch of the physical data representation is observed in next section.

3.1 Compiling Application-Tailored XML Storage

Given a specific query workload (that can include update queries) we compile optimized query plans for the workload and also produce a proper storage plan. In this plan features of XQuery data model that are not required to execute the workload are eliminated.

To build such a storage plan the following main techniques can be used:

1. *Combining structural and textual data representations* (and using appropriate techniques to process each type of representation). As we mentioned above, majority of elements (such as rendering and grouping elements) in content-oriented XML are not addressed by queries/updates at all or addressed in such a way that they can be parsed and processed efficiently on-the-fly (using XML streaming processing techniques). We have designed a method in which queries are analyzed to find nodes of XML data that should be stored in a structural way (preserving parent, child and/or other relationships between nodes like in [9] for example) to make evaluation of the specified queries and updates as much efficient as possible. The rest of XML data (e.g. rendering elements) are stored as compressed text in the same way as text content of the nodes. This method is quite flexible and is not restricted

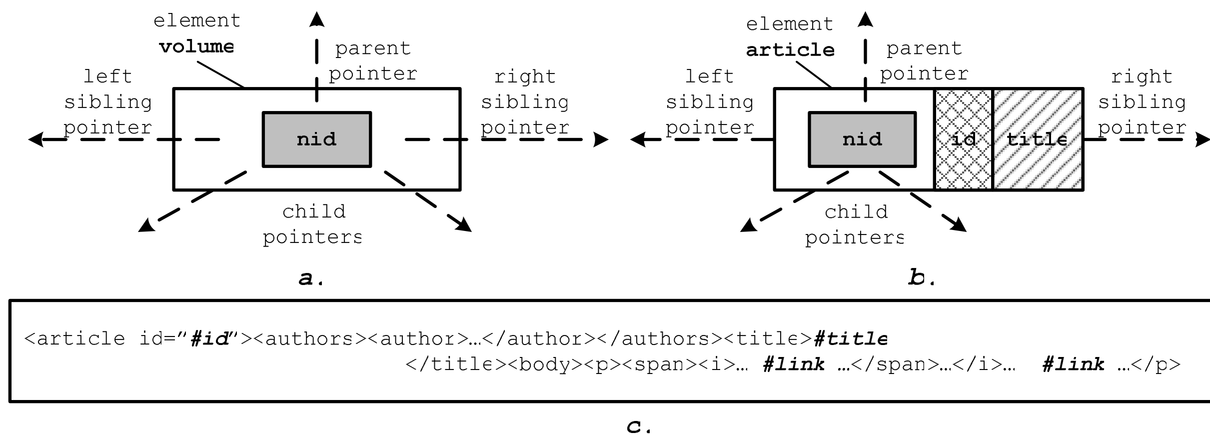

Figure 3: Three Ways of Storing XML Nodes.

by storing the whole XML sub-trees as text: structured elements may have textual children which in turn may have structured elements inside (see Section 4 for the method illustration). Compressing nodes in text representation does not eliminate at all an ability to query them. They still can be effectively processed using XML streaming techniques like in [18].

2. Using *various schemes of clustering* nodes in blocks. Sedna structured representation [9] can be combined with the Natix/DB2 [2, 3] approach on the basis of queries/updates analysis providing significant performance improvements in comparison with both approaches.

3. *Eliminating redundant structures* (such as redundant pointers, numbering labels, etc.) and flattening structure when possible (i.e. removing grouping elements etc.). Analyzing queries/updates we can identify which structures are redundant to support the queries/updates. Redundant pointers and grouping element can be eliminated and data can be represented using more compact data structures. For example, "relational-like" XML data can be stored in records similar to that used in relational storages. Such records are still not as rigorous as relational records to support possible irregularity in data but it is much more efficient then to use any of the general approaches. We can also flatten the structure of XML in many cases. For example, if the person element contains the address element which in turn contains street, house, city as sub-elements - such the address structure can be flatten.

Note also that the techniques eliminate only necessary XML-specific features of XML/XQuery data model. They don't lead to losing the data or don't lead to emerging of redundant data. However our approach can be naturally extended with such powerful techniques as data projection on the basis of static query analysis (e.g. identifying constant-based predicate) or materialized views (might lead to data redundancy).

And last but not least, in the result of building storage structures for a given application we will get simpler data structures than that used in general approaches: elements are less interconnected. It allows improving not only query/update performance but also opens the door for improving locking granularity and building a distributed system. For example, we can implement data parallelism on shared-nothing architecture. The main critique of the shared-nothing approach [8] is that it works well only for particular queries/update workload. But in our approach we optimize for particular application so shared-nothing fits our approach well.

3.2 Recompiling Application-Tailored XML Storage

When the application is modified we might need to recompile the queries and storage. We can employ a flexible policy of recompiling the database. First, the frequency at which we need to recompile the database depends on the level of optimization that we choose to customize the storage for the given application (it might a parameter of optimization as in programming language compilers: O1-O5). Second, it might be required not as often as it might seem. Indeed it is very unlikely that new queries which address a relational-like XML will start using sibling pointers that were removed at the phase of storage compilation.

But in general case we might really need to recompile the whole database into the new structures.

The solution is as follows. The whole database can be reconstructed using massive-parallel distributed processing. It is true for small and middle sized databases that it can be done quite fast even on commodity hardware. If the reconstructed database is distributed (see on the possibility to build a distributed system above) it can be done really fast. In case of simple scheme of partitioning the database (when it is not optimized for things like collocated joins [10]) the reconstruction can be done in parallel just transforming each document independently. In more complex case (e.g. data are partitioned to use collocated joins) we can employ techniques like map-reduce [11] to repartition the data using the keys for different joins.

There are two main options in reconstructing the database. Simple solution is to stop the database and reconstruct it. As it presumably does not take a long time for small or middle sized database the down time

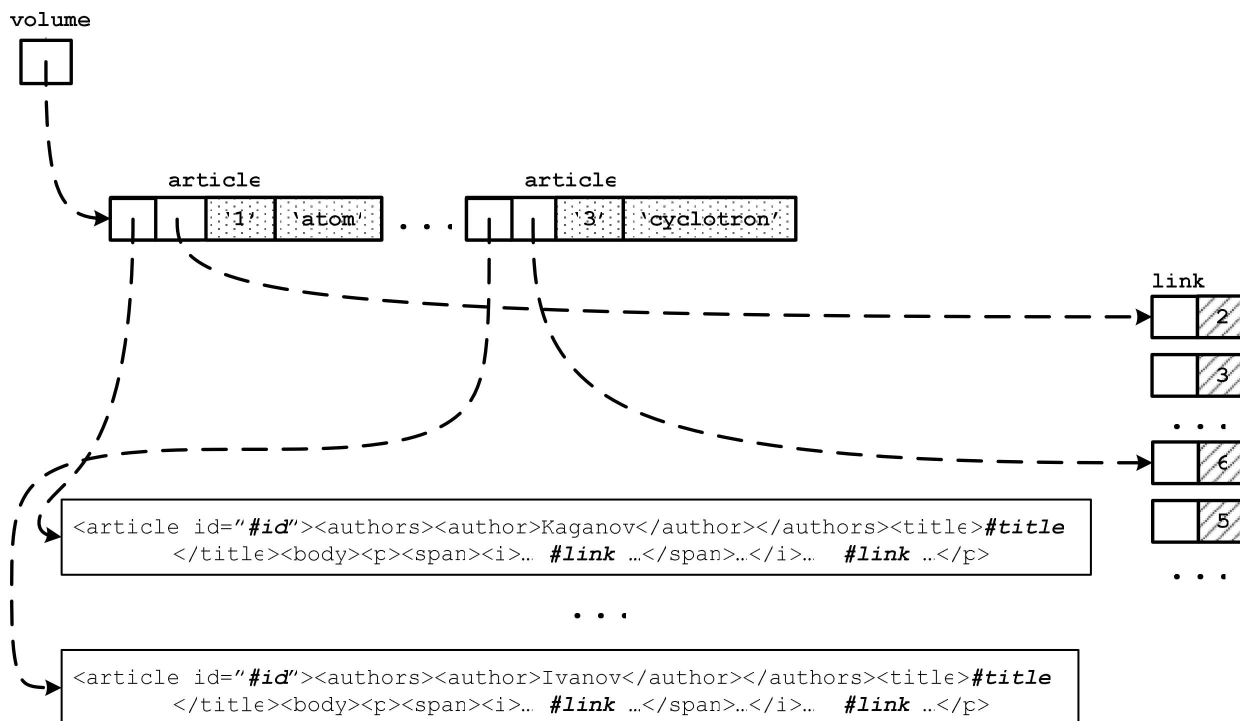

Figure 4: Two Articles Stored in the Application-Tailored Way.

should not be long. Advanced solution is to use snapshot isolation (shadow/versioning) transaction mechanism [12] to reconstruct the database without stopping it.

4 Data Representation

In this section we give a sketch of the physical data representations used in application-tailored XML storage to assure best performance for the given workload.

Let us return to the example described in Section 2 (Figures 1-2). Descriptive schema defines XML nodes decomposition according to pathways in the document. For each group the best storage method is determined in compliance with workload (Figure 3, *a-c*):

- *Node descriptor* - each node in group can be stored as 'node descriptor' structure, which through direct pointers reflects child/sibling/parent relationships between nodes. This way gives effective navigation and goes very well to evaluate structured path expressions [9]. Besides (or along with) direct pointers in this approach numbering scheme can be effectively employed [11, 9]. Every node descriptor can have a label 'nid'. The main goal of using numbering scheme is to quickly determine ancestor-descendant relationship between any pair of nodes in the hierarchy of XML data. It can also be used for determining document order relationship.

- *Value packed in node descriptor* – like in relational databases for some nodes we employ structures similar to the relational records [20]. Record is packed in node descriptor of the ancestor (like id and title values shown in Figure 3, *b*). Actually, in this option we cluster application-level entity (like article or person) with its "relational-like" flat properties (e.g. id, name etc). It gives several advantages. Firstly, presentation is as much compact as possible since we do not have to store irrelevant pointers and numbering scheme labels. Second, it speeds up a whole number of path expressions, particularly with predicates with condition on packed nodes (like //aricle[@id eq "1"]). Finally, it also speeds up serialization process.

- *Node packed in text* - nodes which are not expected to be queried (e.g. rendering elements) can be stored in the textual form. Obviously, it saves space (since we do not have to allocate data blocks for each type of nodes) and speeds up serialization process. As mentioned in Section 3 this method is quite flexible and is not restricted by storing the whole XML sub-trees as text. Textual node can have placeholders inside for the descendants stored using first two options.

Database executor uses descriptive schema to determine the way the node is stored and as an entry point to the data blocks in which node is located.

In Figure 4 storage plan for the example defined in Section 2 is shown. According to this plan:

- 'article' and 'link' nodes are stored in a structural way using node descriptors – since we directly query and serialize them in Q1-Q5

queries. Numbering schema labels and some pointers are eliminated because document order and sibling axes are not used in the workload;

- 'id', 'idref' attributes values are packed in their parents node descriptors – they have simple content and are used in predicates to filter out application-level entities like article or link;
- 'title' nodes is queried and serialized in path expressions in Q1-Q5. Though they are also flatten in their parents nodes descriptors since they have simple content;
- 'author' nodes along with content markup nodes encircled in Figure 2 are packed in textual representation which parent is article's node. It is used only to serialize article and has #id, #title and #link placeholders for the id, title and link values respectively.

5 Related work

There are a few works on building customizable XML storage exists. In the OrientStore [13] system authors propose approach based on the combination of Natix [2] and Sedna [4] storage strategies but the choice of the strategy is data-driven (schema-driven) and the physical representation contains all the features to execute any ad-hoc query. There are a number of approaches like in the LegoDB [14, 15] or XCacheDB [21] which are very close to our work but these storages are based on the relational storage which brings its limitations and overhead. Ideas proposed in all these works might be useful but they do not provide any complete solution for building application-tailored XML storage.

What we propose should not be mixed up with component database [16, 17]. We think that component databases are too general approach which cannot be efficiently implemented in practice. We do not propose to generate database systems for various kinds of database applications such as OLTP, OLAP, etc and for various hardware and software platforms such as PDA, desktop, or server. We just extend the idea of query optimization from building query execution plan to choosing storage structures also. That is aimed at first place to reduce XML/XQuery-specific overhead caused by extra flexibility and extensibility of XML/XQuery.

6 Conclusion and Future Work

In this paper we propose a method of compiling query-driven XML storage designed to reduce the overhead caused by the universality of XQuery data model. According to our preliminary studies and experiments, proposed method allows us to reduce the size of internal representation from several times to orders of magnitude (consequently optimize buffer memory usage) and to store data in a way that minimize the number of blocks addressed by the queries/updates. The overall effect of such optimization should make XML database significantly effective.

This paper reports the preliminary results of in-progress research. The feature work includes prototyping the system and conducting performance experiments. Also we are going to design an XQuery optimizer to construct storage plan automatically (or semi-automatically using a small number of hits) and a method of reconstructing internal XML representation which does not require database shutdown.

References

[1] XQuery 1.0: An XML Query Language. W3C Recommendation 23 January 2007, w3.org/TR/2007/REC-xquery-20070123

[2] T. Fiebig, S. Helmer et al. Anatomy of a Native XML Base Management System, The VLDB Journal 11/ 4, 2002

[3] M. Nicola, B. van der Linden. Native XML support in DB2 universal database. In Proceedings of the VLDB, Trondheim, Norway, 2005

[4] M. Grinev, A. Fomichev, S. Kuznetsov, K. Antipin, A. Boldakov, D. Lizorkin, L. Novak, M. Rekouts, P. Pleshachkov. Sedna: A Native XML DBMS, www.modis.ispras.ru/sedna

[5] M. Haustein, T. Härder. An efficient infrastructure for native transactional XML processing. Data Knowledge Eng., June 2007

[6] E. Ehrli. Walkthrough: Word 2007 XML Format Microsoft Corporation, June 2006

[7] "Great Russian Encyclopedia" Publishing Company, http://www.greatbook.ru/ (in Russian)

[8] S. Chandrasekaran, R. Bamford. Shared Cache - The Future of Parallel Databases. In Proceedings of the ICDE, 2003.

[9] A. Fomichev, M. Grinev, S Kuznetsov. Descriptive Schema Driven XML Storage. Technical Report, MODIS, Institute for System Programming of the Russian Academy of Sciences, 2004

[10] Join methods in partitioned database environments, IBM DB2 Database Information Center, http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp

[11] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, December 2004

[12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. A Critique of ANSI SQL Isolation Levels. SIGMOD International Conference on Management of Data San Jose, May 1995

[13] X. Meng, D. Luo, M. Lee, J. An. OrientStore: A Schema Based Native XML Storage System. In Proceedings of the VLDB, 2003

[14] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, J. Siméon: Bridging the XML Relational Divide with LegoDB. In Proceedings of the ICDE, 2003.

[15] M. Ramanath, J. Freire, J. Haritsa, and P. Roy. Searching for Efficient XML to Relational Mappings. Technical Report, DSL/SERC, Indian Institute of Science, 2003

[16] M. Seltzer. Beyond Relational Databases: There is More to Data Access than SQL, ACM Queue 3/3, April 2005.

[17] S. Chaudhuri, G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. The VLDB Journal, 2000

[18] D. Florescu et al. The BEA Streaming XQuery Processor. The VLDB Journal 13/3, September 2004

[19] Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. Proceedings of the VLDB Conference, Roma, Italy, 2001

[20] H. Garcia-Molina, J. Ullman, J. Widom. Database Systems: The Complete Book. Prentice Hall, October 2001

[21] A. Balmin, Y. Papakonstantinou. Storing and Querying XML Data Using Denormalized Relational Databases. The VLDB Journal, 14(1):30-49, 2005.