

# Adaptive XML Storage or The Importance of Being Lazy

Cristian Duda  
ETH Zurich

Institute for Information Systems  
8092 Zurich, Switzerland  
cristian.duda@inf.ethz.ch

Donald Kossmann  
ETH Zurich

Institute for Information Systems  
8092 Zurich, Switzerland  
donald.kossmann@inf.ethz.ch

## ABSTRACT

Building an XML store means finding solutions to the problems of representing, accessing, querying and updating XML data. The irregularity of both the structure and usage of XML, is, however, a big obstacle in achieving good performance. Relational Database Systems rely on a fixed-schema of records to represent and manage data, but XML data, irregular in structure and content, does not seem to allow this approach. This paper describes how the notion of database record has been extended and applied to XML storage and how the resulted store abstracts the structure of the XML data from the actual storage format. Furthermore, we argue that an adaptive (lazy) XML store and partial indexing are the key points in achieving good performance facing the range of different XML usage patterns. This allows for automatic, application-specific tuning, facing the range of challenges imposed by the current XML applications.

## Keywords

XML, XML Storage, XML Indexing, Partial Indexing, Lazy Indexing, Adaptivity, Laziness, Granularity.

## 1. INTRODUCTION

XML has a prominent role in current industry and research, either as a generic way to represent data or as a format for data integration and application interoperability. XML applications require, from this point of view, the capability to store, retrieve and query XML data. In order to face this range of challenges, XML research has focused on multiple angles of XML storage. The necessity to assign identifiers to nodes in an XML document, brought a variety of identifier and indexing schemes for XML [16],[9],[17] Related to this approach are works which try to map the existing relational database systems to the challenges of XML[8][19] Recently, new challenges have been posed by XQuery[2], the standard language for querying XML. XQuery is defined over the very generic XQuery Data Model and raises issues such as efficient query evaluation, efficient access and retrieval of XML data and maintainment of document order. Query evaluation and optimization[16][1] have their share of research work, such as [15],[9] to list ones relevant to our work. Concurrency and locking protocols for XML also raise specific

problems[[12][13]. The focus switches, therefore, from optimizing queries, reads and access (index structures and identifier schemes), to optimizing XML updates - issues which are rarely be addressed together. A common solution is the simplification of the requirements (such as reducing the XQuery language to substets) in order to allow allows better results when different techniques are combined. These choices can be restrictive for the application. As an example, good identifier schemes that are well supported by index schemes in a relational database [9], help evaluating XPath[5] expressions (restricted part of XQuery) based on containment, but show poor performance for updates.

What existing approaches lack is a uniform way of representing and thinking of XML; furthermore, they focus on one aspect of XML storage, and assume that the application will adapt to a particular usage pattern. Even the more flexible approaches which abstract away the underlying data model and choices made while developing the store, offer little place to real adaptation to the application (e.g., defining a threshold parameter of this notable work [15], requires a lot of knowledge of the underlying storage pattern). Another issue which affects existing approaches is that they function as all-or-nothing as what regards indexing: the previous example of the identifier scheme is a typical one, where advantages gained by knowing all node information is lost in poor performance updates. The conclusion is, one cannot achieve everything at once, but should focus on what it can achieve at a given moment, in a given usage context. Some existing approaches took the first partial approaches to doing that: either restricted at a special scenario such as query evaluation [4][19] or even more generic, closer to our work: [11][12]

In our opinion, the desiderata for an XML store has at its center the possibility to abstract away the actual XML model of the application, and to give enough room for determining the best way to store depending on the application usage pattern. The keywords are: adaptivity, laziness and partial. Baring this in mind, our contribution is the use of a flat representation - an XML instance is a sequence of *Ranges*: logical units similar to tuples in relational databases, whose size and existence is defined by the application usage pattern (inserts/deletes, etc.). This unit is supported by the choice of our XML representation, and opens the way to a lazy approach to storing, accessing and indexing XML data. Ranges replace a tree model with a flat one, and offer enough flexibility to have application-dependent indexing units, while still allowing for more granular indexing, if needed. Additional issues such as identifier schemes are, therefore, orthogonal to this model.

The well-known results to the problems of storing, accessing, querying, and updating data in relational database systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Second International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, June 16–17, 2005, Baltimore, Maryland, USA.

(RDBMS) seem to have no direct applicability when facing the irregularity of XML Data. The intrinsic tree model of XML is not compatible to the flat model of tuples complying with a well-defined schema, stored in the pages of the RDBMS. Successful stories found a niche in defining appropriate index structures and identifier schemes that have a straightforward mapping to a relational data model [16][9][19]. Querying XML data using the previously defined index structures maps better to the set-oriented relational data model[19][1] In order to address the actual storage issues, native[14] XML store implementations [15] find an extension of the relational data model by arguing that tree structures and tree-based keys are a natural extension of the relational model to the XML world.

This paper is structured as follows: Section 2 formally defines our requirements for an XML store and Section 3 motivates our choice for XML Representation. The partial, lazy approach we take to indexing is discussed in Sections 4 and 5. Identifier schemes are briefly taken into consideration in Section 6, and shown to be orthogonal to the main storage model. Section 7 presents experimental results. After a discussion of related work in Section 8, conclusions and future work constitute the core of Section 9.

## 2. XML STORE DESIDERATA

In the context of the previous section, we derived the following list of requirements:

1. Store and access any instances of the XQuery DataModel
2. Support for XUpdate
3. Allow optimization of reads and/or updates.
4. Indexes
5. Support different Node Identifier schemes. In particular, support for stable and comparable identifiers should be offered.
6. Low storage overhead.
7. Support PSVI.

The XQuery Data Model[2] supports a wide range of XML applications (either read-oriented, or heavy-update scenarios), and support for XQuery itself is a must for a standard-compliant XML store. PSVI should be supported in order to avoid repeated evaluation of XML schema. Low storage overhead incurs by minimizing the quantity of data actually stored. In our approach we do not store all node identifiers, but store enough information to allow us regenerate them. See Section 6 for more details on Identifiers. The index structures are discussed in Sections 4 and 5.

The Store should support read operations (entire data source, but also a single node), and update operations (XUpdate) as described in Table 1. XUpdate operations specify a node and allow insertions of the data relative to this node (as previous siblings, next sibling, first child or last child of the node).

Table 1: Interface of the store.

insertBefore(id, ...)	read()
insertAfter(id, ...)	read(id):...
insertIntoFirst(id, ...)	deleteNode(id)
insetIntoLast(id, ...)	replaceNode(id, ...)
	replaceContent(id, ...)

Node identifiers are assigned, according to the XQuery DataModel to each node in the data instance. In particular, executing an XUpdate operation involves more steps: locating the target ID, identifying the insert position (e.g., as previous sibling, as next sibling, as first child, as last child), and performing the actual update.

## 2.1 Optimizing Reads vs. Optimizing Updates

Typical storage systems are faced with challenges of optimizing read operations or update operations, as required by the application. A store that achieves both optimally is a utopia since the structures required to support the first type of operation (fast indexes) negatively influence the performance of the other. In this work we take a middle approach, and try to optimize one or the other depending on the application load. **Adaptivity, flexibility and laziness** are another main desiderata expressed by our requirements.

The following sections argument how these requirements have been fulfilled in the particular case of our XML store. The important choices are: the XML representation, the definition of an arbitrarily granular unit Range, and the flexible index structures based on the existence of this unit.

## 3. XML REPRESENTATION

### 3.1 Choosing an XML Representation

Current research on XML takes one of few alternatives to represent and store XML data. XML data is either shredded on a relational database [8][19][10], special index structures, or a combination of the two [15]. There is usually a strong relationship between storing and representing XML on one side, and indexing and querying it on the other side and current approaches do not conceptually separate them. Usual approaches provide neither the data independence, nor the flexible granularity that make up our adaptivity requirement.

### 3.2 A Flexible Representation: Sequence of Tokens

In order to achieve our goal, we have chosen a representation which is able to express anything between very granular and very coarse instances of the Xquery Data Model. We use a representation derived from a pull-based XQuery parser and engine, already described in literature[7]. We will accordingly use the notion of *Token* to denote each part of the XQuery Data Model, as defined in this representation.

Tokens can be defined as a materialization of enriched SAX events. The model is richer than usual SAX events (or event-based parsing models), as it defines units that do not exist on the SAX model (attributes separated from their element, and given corresponding begin and end tokens) [7].

<ticket>	[BEGIN_ELEMENT [ID: 1] [ticket]
<hour>	[BEGIN_ELEMENT [ID: 2] [hour]
15	[TEXT_TOKEN ID: 3 15]
</hour>	[END_ELEMENT]
<name>	[BEGIN_ELEMENT [ID:4] [name]
Paul	[TEXT_TOKEN ID:5 Paul]
</name>	[END_ELEMENT]
</ticket>	[END_ELEMENT]

Figure 1: Sample XML document and corresponding Tokens

Figure 1 presents the tokens corresponding to a sample XML document.. In particular, Nodes in the XQuery Data Model, who must have an associated identifier, are also represented by a sequence of tokens. Here, in particular, the first one holds the identifier. A nested node is represented by a sequence of tokens starting with a *Begin token*, containing the Id, and an *End token* [7].

This particular representation of the XQuery Data Model offers us the following properties, which make it suitable for our goals:

1. Complete representation of the XQuery Data model
2. Independence of the API used in the actual application (flat model, as opposed to tree-based or event-based representation)
3. Allows flexible data granularity: the token is the most granular unit
4. PSVI : Post schema validation infocet [7]

Our representation is based therefore on the following fact: a token is the most granular unit (even more granular than an XML element); tokens can be grouped in more specific units (nodes/elements are a group of tokens). Other representations, in particular tree-based ones [15], do not provide this level of flexibility and are not extensible to the detail of the complete XQuery Data Model. PSVI fulfills one of our requirements since a token sequence (and, in particular, each XML Token) can also be associated to the XML schema type derived after a schema validation.

The aforementioned level of granularity allows a store implementation to use serialized tokens as the representation of the XML data. An XML data instance is represented by the full sequence of stored tokens.

### 3.3 The Storage Model

The storage model of the XML data in this store consists of token sequences serialized in sequential blocks/pages, in document order, as described in Figure 2. Each time data is inserted in the store, the corresponding tokens are generated and store in the corresponding positions in the storage: blocks are allocated accordingly. Tokens offer, therefore, a flat representation of the XML data, independent of the actual data model of the application that uses the store (such as DOM, SAX, etc.). Node Ids, requested in the XQuery Data Model, are also generated at insert time. We adopt the approach of stable identifiers, generated to all begin tokens of nodes, at insert time.



Figure 2: An XML Data instance is represented by a sequence of tokens

## 4. IMPLEMENTING DATABASE RECORDS FOR XML

### 4.1 Discarding the Option of a Full Index

As described in previous sections, the interface to the store defines read, insert and update operations. Since all operations are defined relative to a target node Identifier, it is nodes corresponding to these identifiers that need to be quickly located during the update.

Section 1 described existing approaches to XML data storage [1][16] that take the options of full data indexing - the purpose is to optimize queries and accelerate access to specific parts of XML data. We argue that this is not a valid choice: updates are too expensive in this way.

The advantages of a full index are the ability to quickly locates nodes. However, a full index has two main disadvantages: (a) inserts are expensive, and (b) storage requirements are very high. Besides the fact that the index grows in size for data-intensive applications, the vast majority of the entries will not even be used. This approach is in the way of fast updates.

A typical usage pattern will access the data based on semantic constraints, such as: insert a <purchase-order> element as the last child of the root (a <purchase-orders> element). Performing this operation means getting access to the root node and the position corresponding to its last child. The vast majority of other index entries, if this operation is repeated, will not be used. We also prove experimentally (see Section 8) a full index induces a big overhead: updates are more expensive. The lazy indexed introduces are the answer to this problem.

### 4.2 The Notion of Range

We define the notion of Range as a sequence of tokens. In the idea that a full node index cannot accommodate the entire application usage patterns, we introduce the Range as a less granular unit.

Our store defines the range as an insert unit: by populating an XML DataSource with tokens corresponding to a node we create a first range (the initial content). A successive insert of a new child, define a new range, while the existing range is split in two. The choice of the representation (tokens), as presented in Section 3, has a crucial role in making the notion of Range possible: each sequence of tokens can constitute a range. The real choice is made by the application itself.

We can draw an analogy between the implementation of relational database systems, where a Record is as a sequence of variable-sized fields and our model, where a Range is a sequence of variable-sized tokens. What we add in the XML approach is that a Range is defined by the usage pattern of the application, and not by a fixed schema. In our model, we must also maintain an order between Ranges in order to preserve document order of tokens. Ranges, as also described in the previous section, are the way to obtain cheap updates.

### 4.3 The Range Index (Coarse-grained index)

The role of an index is to quickly locate a position in the store. The range index is to locate the range corresponding to an ID specified in an update operation. Ranges represent insert units – as opposed to a full index, which would have contained all Ids individually. The range index contains less entries, but it is also fuzzer (i.e., it refers to an interval of Identifiers instead of to a single one). The Storage level is able to retrieve a range, given its Id.

A consequence of this way to represent data is that Node Identifiers need not be stored together with the tokens they refer to. By knowing the start identifier of a Range and by successively reading successive the tokens of that range, identifiers can be generated and re-associated to the tokens they belong to. The advantage is better space utilization (low storage overhead), a valuable resource in case of variable-length IDs. Section

#### 4.4 The Storage Model (revised)

Based on the existence of Ranges as the logical storage unit, the Storage level comprises chained blocks, which, at their turn, contain ordered ranges. Document order is preserved through the chaining of blocks and through the ordering of ranges inside blocks. (see Figure 3).

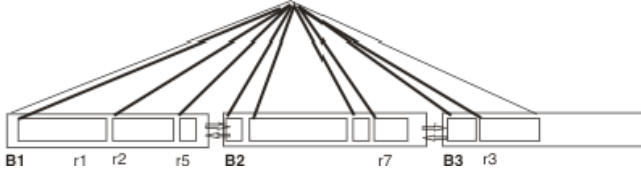


Figure 3: The XML Data Instance as a sequence of ranges, and the Range Index

#### 4.5 Functionality of the Range Index

This section presents a simple usage scenario of ranges in the store. We assume that we have on an initially empty Data Source.

The operations which are performed are:

1. Insert 2 sibling Nodes (contain 100 nodes in total)
2. Insert a child (40 nodes) as the last child of the node which is identified by 60: insertIntoLast(60, <<new data>>)

The effects on the Store are that tokens are created for the inserted data, and they are stored sequentially on the pages. Node Ids are created, but only the ranges are inserted in the Range Index.

The detailed description of the effects of each step on the Range Index are described below:

1. Allocate 100 identifiers for the inserted nodes, and create range 1 with Ids 1-100. Range 1 is stored in Block 1.
2. Insertion of the child :
  - a. Locate second node using the Range index (id 60 is in range 1)
  - b. Locate range and offset of the end token of the node with the Id 60.
  - c. Split range number 1 in two (create range 3)
  - d. Create a new range corresponding to the inserted data (2), and allocate 40 unique identifiers
  - e. Store the new range (Block 1) and insert the split range in the storage. (Blocks 2).

Table 2. Range Index (Coarse index) with an initial range.

RangeId	BlockId	StartId	EndId
1	1	1	100

Table 3. The Range Index (Coarse index) after an insert (nodes 101-140, range 2) and split of range 1.

RangeId	BlockId	StartId	EndId
1	1	1	70
2	1	101	140
3	2	71	100

Tables 2 and 3 describe the configuration of the Range Index during inserts.

#### 5. THE LAZY/PARTIAL INDEX

The notion of Range and that of Range Index allow us to optimize update operations: fewer entries are inserted to the range index - a big step forward in comparison to the full index approach.

The price to pay for having cheap updates is that reads become more expensive. Since the nodes cannot, in general, be accessed directly anymore (they are 'hidden' inside a certain range), additional lookups need to be performed. A full index would have been able to directly access each node. The full index has an advantage of quick lookups, but it is unacceptable because of two things: the high memory overhead and expensive update operations.

The solution to this problem is a Partial Index[18]: using the advantages of the full index, but only when needed. The result of lookup operations, performed during updates is inserted in the partial index: either the range of a token, the offset of a token inside its range, the location (range, offset) of the end token of the node (i.e. node token) inside the range and the position of the end token of the node inside. With the help of this additional structure, a repeated search for the same logical position will benefit from the existence of this value in the partial index ("jump" to the end of the given node).

The partial index stores, therefore, information on the individual nodes: their exact ranges and offset inside the range (for simplicity, the example in Table 3 only describes the existence of the range of a token). Because this very granular functionality is the one which we try to avoid from traditional approaches, the partial index is actually a combination between a real index (such as those defined by other identifier schemes) and a cache. The combination between the Range Index and the Partial Index (see Figure 4) achieves the goal of being, adaptive, flexible and lazy in the XML world. This can be done by not trying to index everything, but only if and when needed.

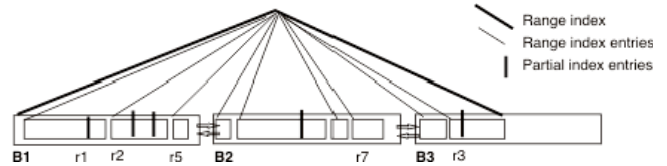


Figure 4: Partial Index entries enrich the coarse Range Index

Referring to the example from Section 4.5, the partial index (considered empty at the beginning), would be used as follows:

1. Inserting on an empty data source does not create entries in the Partial Index
2. Inserting a new node:
  - a. Locating node with Id 60 using the range index in range 1: a new entry is inserted in the partial index to indicate the range.
  - b. Locating the end token of the id 60. This means that after the insert, the location of the end token is the range 3: a new entry is inserted in the partial index to indicate the range (see Table 4).

**Table 4: The Partial Index (Granular) after the second insert: lookup positions have been memorized**

NodeID	Begin Token	End Token
60	(Range) 1	(Range) 3

## 6. ORTHOGONALITY OF ID SCHEMES

As mentioned in the introduction, proposals for indexing and identifier schemes for XML constitute a large part of the existing research on XML [16][19][6][19]. Indexing XML data relies heavily on the fact that nodes in an XML document are assigned an identifier. In particular, update operations are expressed based on these identifiers and indexes can be build on top of them. The XQuery Data Model, which we implement, requires document order in the XML representation and unique identity of Nodes inside this representation. We use Node Ids, generated at insert time, and we obtain document order by chaining blocks and maintaining order of Ranges inside blocks.

As opposed to previous approaches, our model provides a separation from the API of the application: a range can span over several nodes, or over parts of a node (represented as a sequence of tokens). Ids of nodes are, currently, orthogonal to our way of indexing.

### 6.1 Low Storage Overhead

The Range Index is a coarse-grained index (see Section 4). This already means lower storage overhead over a full index approach. Beside its role of locating a Range in the storage, the Range Index uses properties of Ids for locating the range of a node with the given id. This functionality can be described as follows:

$$rangeIndexLocate : \{ID\} \rightarrow \{R\}$$

where  $\{ID\}$  is the set of Node identifiers in the store, and  $\{R\}$  is the set of all ranges in the Range Index. Currently, we achieve this by maintaining information on each  $[startId, endId]$  interval of the ids inside a Range. Since a Range is defined as a sequence of tokens in document order, we can obtain further decrease storage overhead by only storing the Identifiers of the first node inside a range. The Id schemes with this property generate the Id of the next token ID using a simple factory function:

$$idFactory : \{ID\} \times \{token\} \rightarrow \{ID\}$$

Many existing identifier schemes are compatible to our current approach, because they have the previous property. [19][17].

### 6.2 Stable and Comparable Identifiers

Stable identifiers are the way to build indexes on the store: such an index can be external and based on logical node identifiers. Currently, stable identifiers can be obtained by assigning unique integer number to nodes at insert times This simple approach allows us to define actual ranges of Ids (in the example of Table: 1-70, 101-1). The Ids inside ranges are comparable Ids inside ranges. We can obtain a semi-stable document order at read time (since tokens are stored in document order and read sequentially). The combination of order between ranges and order of ranges in the storage, can also be put in connection to partially-stable identifier schemes, such as those described in [19].

Ids which are both stable and fully comparable in the document order, can currently only be obtained by using a different

identifier scheme (in particular [17]). Further details are omitted for lack of space.

## 7. EXPERIMENTAL RESULTS

The following micro benchmarks reflect the effects of using a coarse index and a partial index, as opposed to a full index, in our representation. The identifier scheme associates unique integer values to each node, at insert time. However, only ranges become entries in the index. A memory-based partial index lazily adds information on the location of tokens inside ranges (begin and end token).

Experimental setup: our first implementation is built completely on top of a relational database system. We use Java and JDBC to define express the operations of the interface of the store. The test platform was a Pentium 4 2,8Ghz/512MB RAM, running SuSe Linux9.0, and using a MySQL as a database.

The parameters that influence the results of the benchmark are the size and number of ranges. A coarse-grained index means low update overhead but a larger overhead at read and lookup times. On the other side, an index containing many entries (even coarse-grained) also leads to performance decrease at insert time. The partial index improves reads especially in the case of more coarse-granular range sizes, as it builds entries lazily (cache-like).

The experiments involve the following micro benchmarks: Inserts, sequential reads, and random reads of small pieces of data in the presence of a full index, range index, and, respectively the combination of range index + partial index (see Table 5). The metric is kilobytes/second (read speed, relative to data size).

**Table 5: Experimental results: Lazy indexing in XML storage**

Indexing approach	Insert (kb/s)	Seq.scan (kb/s)	Random reads (kb/s)
Full Index (max granularity)	27,91	1298,59	672,22
Range Index (many, granular entries)	97,07	1333,47	136,98
Range Index (few, coarse, large entries)	91,03	1333,47	33,41
Range Index (few, coarse, large entries) + Partial Index (memory)	182,32	1333,47	994,36

The results reflect the expected behavior: the Range Index clearly brings advantages in what regards update speed: less entries are entered the index. As the number of entries increases, however, even with smaller data quantity, the advantages diminish (many, granular entries). The Partial Index helps to achieve cheaper reads and lookups (especially when the range index is coarse). We are considering more optimizations of the read/update/storage overhead.

## 8. RELATED WORK

Our work bears close resemblance to existing native XML storage research[12][11]. Even though the flexible granularity is taken into consideration by the authors, the tree model of the XML data is not abstracted away but is used to define, in a similar approach, partial indexes. Other similarities to this independent research involve sequentially storing the XML data, but the notion of

variable-size range and varying granularity are not entirely contained in the proposed approach: element node (i.e., fine-grained units) are still the storage unit in this approach. The hybrid approach taken by Natix[15] and its tree-based model also brings ideas in the field of granularity of records, but it relies on a tree model. Logic identifiers have not been extensively studied in this research. A special mention deserves a project[3] which, under the umbrella of enhancing existing identifier schemes based on containment join-based algorithms for evaluation of XPath expressions, defines the notion of ‘segment’ (similar to a range), as a group of elements which are treated as a single insert operation, instead of separate ones. The experiments study the effect of this coarse-granular unit, especially as the segments increase in number. Their performance is degraded because of the eager approach to indexing the content of the segment (group of elements) define lazily. Other approaches try to take adaptive and laziness in XML processing, but in the field on queries and path evaluation[4].

## 9. CONCLUSIONS AND FUTURE WORK

This paper described a data representation and model of an XML store, inspired by the notion of records in relational databases. The immediate advantages are independence of our data format from the API used by the XML application, and the possibility to adapt to the application pattern. The store achieves this by lazily creating its storage and index structures and optimizes for reads or updates according to the how the application focuses on one or the other. The process is transparent to the application.

Our approach to XML Storage involves an exploration of a number of design options. We are currently evaluating experimentally the effects of variable-sized ranges as logical unit for XML data representation. The effect of functionality of the partial index is also to be taken into account. Structural properties of the actual elements of the XQuery DataModel, such as hierarchical or sibling relationships can also be maintained by the Partial Index.

Another aspect to explore, not addressed here, is concurrency. The flat model proposed in this paper allows the definition of these concepts on a three-layer architecture: blocks, ranges and tokens. Again, the principles of storage already defined in the context by relational database systems, have an immediate application here. The issue that differs from the relational world is the necessity to always maintain the order between ranges. This is ongoing work.

## 10. REFERENCES

- [1] Al-Khalifa, S.; Jagadish, H.V.; Koudas, N.; Patel, J.M.; Srivastava, D.; Yuqing Wu, Structural joins: a primitive for efficient XML query pattern matching ICDE 2002
- [2] Boag, S. D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0 An XML Query Language, W3C Working Draft, Nov 2003.
- [3] Catania B, Beng Chin Ooi, Wenqiang Wang, Xiaoling Wang Lazy XML Updates: Laziness as a Virtue of Update and Structural Join Efficiency, *ACM SIGMOD*, June 2005
- [4] Chin-Wan Chung J.-K. M. and K. Shim. APEX : An adaptive path index for XML data. *ACM SIGMOD*, June 2002.
- [5] Clark J., S. DeRose, XML Path Language (XPath), Version 1.0. W3C Recommendation (Nov. 2000)
- [6] Cohen, E., Haim Kaplan, Tova Milo: Labeling Dynamic XML Trees. *PODS 2002*: 271-28
- [7] Florescu D., C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. *In Proceedings of International Conference on Very Large Databases (VLDB)*, pages 997–1008, Berlin, Germany, Sept. 2003.
- [8] Florescu D., D. Kossmann, Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin* 22(3), 1999.
- [9] Grust T. Accelerated XPath location steps. *ACM SIGMOD* 15(5):795–825, June 2002.
- [10] Halverson A., V. Josifovski, G. Lohman, H. Pirahesh, M. Moerschel. ROX: Relational Over XML. In *VLDB 2004*
- [11] Haustein M., Härder T, Fine-Grained Management of Natively Stored XML Documents, Submitted
- [12] Haustein M., Theo Härder: Adjustable Transaction Isolation in XML Database Management Systems. *XSym 2004*: 173-188
- [13] Helmer S., C.C. Kanne, and G. Moerkotte, Lock-based protocols for cooperation on XML documents, Technical report, the University of Mannheim, 2003
- [14] Jagdish H. et al. Timber: A native XML database. *ACM SIGMOD*, page 672, June 2003.
- [15] Kanne C. C. and G. Moerkotte. Efficient storage of XML data. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2000.
- [16] Li Q. and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, September 2001.
- [17] O’Neil Patrick E., Elizabeth J. O’Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury: ORDPATHS: Insert-Friendly XML Node Labels. *SIGMOD Conference 2004*
- [18] Stonebraker. M. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989
- [19] Tatarinov I., et al. Storing and Querying Ordered XML using a Relational DBMS, *In VLDB*, 2002.
- [20] World Wide Web Consortium, Extensible Markup Language (XML). W3C Recommendation, February 1998.
- [21] World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation Sept. 2001.
- [22] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model, W3C Working Draft, April 2005