Mapping XML to Key-Value Database

Pavel Strnad
Czech Technical University, FEE
Prague, Czech Republic
Email: pavel.strnad@fel.cvut.cz

Ondrej Macek
Czech Technical University, FEE
Prague, Czech Republic
Email: ondrej.macek@fel.cvut.cz

Pavel Jira
Czech Technical University, FIT
Prague, Czech Republic
Email: jirap1@fit.cvut.cz

***Abstract*—XML is a popular data format used in many software applications. Native XML databases can be used for persistence of XML documents or the document can be stored in a relational database.**

In this paper we propose an alternative storage for XML documents based on key-value databases. We propose three general algorithms for XML to key-value database mapping. The algorithms are optimized using specific features of key-value database Redis. Finally a performance comparison of implemented algorithms and common native XML databases is provided.

***Keywords*-XML; key-value database; Redis; RedXML; XML mapping**

I. Introduction

XML format is a popular data format used in many software applications, where the XML documents can be stored as files or in native XML databases, but often they are stored in Relational Database Management Systems (RDBMS).

In this paper we propose an alternative to mentioned storages of XML documents based on key-value databases, which are popular in last days. The next reason why to consider key-value databases as possible storage of XML documents is the speed of queries in key-value databases, which can improve the speed of XML querying. Therefore we have proposed algorithms for XML to key-value database mapping and implemented them using key-value database Redis [1]. Next a performance testing was performed on two basic scenarios of loading and saving an XML document. Results of these benchmarks and their comparison to already existing XML storages show if the key-value databases are a next alternative for storing XML data.

The paper is organized as follows: first related work in the area is discussed in Section II, then the algorithms for XML to key-value database mapping are introduced in Section III and their implementation and optimization is described in Section IV. Next, there is the performance comparison in Section V and finally the future work is discussed in SectionVI.

II. Related work

XML to Key-Value Database mapping: Vaidya and Gopal [2] presented an algorithm for mapping an XML document to a two dimensional array of hashes. In contrast with this work we introduce more sophisticated mapping algorithms and we provide a performance comparison with native XML databases. At the moment we are not aware of any other extensive research in this field, therefore we focus on mapping XML to RDBMS.

XML enabled databases: The concept of XML-enabled databases allows to manipulate data stored in relational database as an XML document. The approach is based on a special column type or on a mapping of an XML document to a relational schema [3] [4].

XML to RDBMS mapping: The problem of mapping XML document to RDBMS is well known issue. Many algorithms of mapping an XML structure to a relational database schema were proposed. Some of them are based on a structure of XML document [5] [6], others use an additional information about document querying to create more effective mapping [7] [8]. The issue of retrieving XML documents from RDBMS is addressed in [9] where several mapping algorithms are introduced and theirs efficiency and scalability is evaluated.

The mapping algorithms proposed in this paper allows key-value databases to be used as XML-enabled ones. All proposed algorithms are based only on structure of an XML document. The optimization of proposed algorithms is based on features specific for Redis database.

III. Algorithms

In this section we describe three basic principles of mapping XML documents (elements, attributes, etc.) into a key-value database. As an example key-value database platform we have chosen Redis. Each of proposed solutions differs in the way of mapping and some of them are dependent on specific features of Redis database. All mappings mentioned in this section are demonstrated on document books2.xml shown in Figure 1.

A. Redis Database Structure

Before we propose transformations for XML to key-value database mapping we introduce the structure of Redis database as the target of all mapping algorithms. The Redis database consists of environments containing collections,

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-247-9

which can be nested, so it is possible to create trees of collections.

The environment is the main logical unit in the database which contains collections, documents and sequences for generating identifiers. Each environment defines a context for various database settings. The environment information are stored in keys:

- **info** – contains only the field <*iterator*> representing the value of sequence for generating identifier for a new environment.
- **environment** – contains mapping from environment name to identifier.
- **IDenvironment<info** – is similar to the **info** key as it contains the <*iterator*> for collections and documents stored in the environment.
- **IDenvironment<collections** – for collections in the concrete environment it contains the mapping from the collection name to its identifier.

To organize the content of the database the collections can be used. Each collection can contain a document or other collection. The collections are represented by following keys:

- **IDenvironment:IDcollection<info** – contains information about the collection - its name (*name*) and a parent identifier (*parent_id*). The parent identifier is used for effective navigation inside the collection trees and the environment.
- **IDenvironment:IDcollection:documents** – contains mapping from documents' names to theirs identifiers.
- **IDenvironment:IDcollection:collections** – contains mapping from sub-collections' names to theirs identifiers.

B. Naive Mapping

The first designed mapping of an XML document to key-value database is Naive mapping. This mapping is a straightforward solution and we introduce it as a reference solution which we compare to other proposed solutions.

The basic idea of Naive mapping is to store most of the information inside a key name which refers to a specific part of an XML document. Hence, the knowledge of the key provides important information without a need of querying a database. This finding is important for optimal implementation of querying languages such as XQuery [10]. The example of this key is following:

various::ebooks::books2.xml::
catalog::book>1::genre>2

For better understanding of the key above an equivalent XPath query is:

doc("books2.xml")/catalog/book[1]/genre[2]

Informally we can describe this key as: This key points to an element "genre" that is the second descendant node of an element "book" that is the first descendant node of a root element "catalog" and the root element "catalog" is located in "books2.xml" in "ebooks" collection and in a database named "various".

```
<?xml version="1.0" standalone="yes"
      encoding="UTF-8" ?>
<catalog>
   <book id="bk101" ISBN="123456"
         count="10">
      Author:
      <author>Gambardella, Matt</author>
      <title>XML Guide</title>
      <genre>Computer</genre>
      <genre>Technical</genre>
      <genre>Various</genre>
      <price>44.95</price>
      <publish_date>2000-10-01
      </publish_date>
      This book is nice.
   </book>
   <book id="bk102" ISBN="123457"
         count="9">
      Author:
      <author>Ralls, Kim</author>
      Title:
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <genre>Various</genre>
      <price>5.95</price>
      <publish_date>2000-12-16
      </publish_date>
   </book>
</catalog>
```

Figure 1. Example XML docuement books2.xml

As we can see from the example above every key stores a lot of information inside its name. This can be very useful in XPath access optimization. The separator of two colons "::" must not be appear in th names of elements, collections or files.

The previous example shows how keys used in the mapping looks like. The following paragraph describes utilized structures by Naive Mapping that can be referenced by a key.

Element mapping: The key pointing to an element contains a list of children keys. This is useful for recursive descent along children elements. Information about parent is stored directly in the child's key. The prefix various::ebooks::books2.xml:: is omitted for better readability.

Key:

various::ebooks::books2.xml::catalog::book>1

Copyright (c) IARIA, 2013.

A corresponding XPath Query:

doc("books2.xml")/catalog/book[1]

Content:

*{catalog::book>1::name>1,
catalog::book>1::genre>1}*

Attribute mapping: The key refers to a hash containing attributes of an element. For Attribute mapping we do not provide a corresponding XPath Query because XPath queries can not generate only attributes into the result.
Key:

various::ebooks::books2.xml::catalog::book>1<attributes

Content:

{'ISBN'=>'123456', 'count'=>'10'}

Text mapping: The key refers to a string representing mapped text. The same way of mapping is used for comments or CDATA sections. We only change the keyword "text" to "comment" or "cdata" respectively.
Key:

various::ebooks::books2.xml::catalog::book>1>text>2

Content:

"This book is nice."

XML declaration and file information mapping: The content of this key provides basic information about the document. This information contains the root element name, XML declaration and so on. We define a new delimiter "<" that is used to recognize special keys known as properties. In the next example we present a special key "info".
Key:

various::ebooks::books2.xml<info

Content:

{"root" => "catalog", "version" => "1.0", "encoding" => "UTF-8", "standalone" => "yes"}

The important advantage of Naive mapping is an easy implementation and high information density of keys' names. Main disadvantages of this approach are:

- The length of key names is enormous for deep nested elements.
- Rename operation of a document, element or collection is very inefficient, because every key containing old name has to be renamed.
- Delete operation is also inefficient.

The following approaches were developed to remove (or minimize) these disadvantages of Naive mapping.

C. Abbreviated Key Mapping

Abbreviated Key Mapping (AKM) is a mapping approach that eliminates the enormous length of keys of deep nodes and simplifies rename and delete operations. The basic idea of AKM is based on mapping of all elements, collections and files to unique identifier ID. In AKM the rename operation executes in a constant time ($O(1)$). We only change the name which is mapped to an ID. ID remains unchanged. IDs are used instead of names in keys.

AKM needs a new hash map which maps names to IDs. The last change is abbreviation of long identificators: *'attributes' => 'a', 'text' => 't', 'cdata' => 'd', 'comment' => 'c'*.

We have to define a new structure in the AKM's implementation. This structure is a hash map which maps original names to unique IDs. The example of the key is:

1:2:3:1:2>2:3>3

And an unbreviated version using Naive Mapping is:

*various::ebooks::books2.xml::
catalog::book>2::genre>3*

As you can see all names are replaced by identifiers. This key is much shorter than unabbreviated one. The AKM helper structure which maps names to IDs is stored for each document in the database independently. The name of this key is "emapping". The example of the mapping is:
Key:

1:2:3<emapping

Content:

*{"catalog" => "1", "book" => "2", "genre" => "3"
"<iterator>" => "3"}.*

Field " $< iterator >$ " is used for counting element IDs. If a new element is added " $< iterator >$ " is incremented and its value is used as ID of the new element. Hence, each document has its own iterator. This design decision was made with regards to the length of keys. Shorter keys are better because they are better aligned in a memory. AKM compared to naive mapping is much better performing in rename operation (in a constant time $O(1)$). On the other hand, performance of delete operation is still poor. Hence, we tried to find a better solution.

D. Centralized Hash Mapping

Centralized Hash Mapping (CHM) approach is based on a hash structure. The basic idea is to store the whole document in one hash structure to allow easy removing of documents from a database.

Redis database allows storing of string values into a hash structure. This lead to a change of a representation of documents in the database. The key which identifies this hash structure is:

1:2:3<content

The hash fields represent keys the same way as in previous mappings. The difference is that CHM does not use a document prefix to uniquely identify an element. The example of this key is:

1:2>1

This key contains following array of children:

"1:2>1:3>1 | 1:2>1:3>2"

As we mentioned above Redis allows only string values. Hence, we have to store children elements in a string delimited by "|".

Attributes are stored in a similar way:
Key:

1:2>1<a

Content:

1"value1"2"value2"3"value3

As we can see attribute names are also replaced by IDs. In this case we used quote symbol (”) as a delimiter. This delimiter is conflict–free because quote must not be included in an attribute's value according to XML specification [11]. Full EBNF grammar of the proposed mapping is presented on page 69 in Appendix E in Jíra's Master's Thesis [12].

CHM provides many advantages in contrast to previous approaches. Rename, delete and move operations are executed in a constant time $O(1)$.

IV. IMPLEMENTATION

The implementation of algorithms mentioned in previous section is covered in project named RedXML. RedXML is written in Ruby language and is available as an open-source project at github [13]. As a storage layer RedXML uses Redis database.

This section provides information about the performance optimization of the database structure.

A. Key Cutting

Key Cutting is a mapping approach that utilizes a memory optimization techniques [14] implemented in Redis. Key Cutting is built on a fact that storing hashes in Redis can be much more memory efficient than storing key/string pairs. This approach optimizes memory consumption of the database.

For example $value1932 => ”value”$ consumes more memory than this hash representation: $value19 => \{”32” => ”value”\}$.

Main advantages of Key Cutting approach are:

- a decreasing of memory consumption by using hashes instead of strings,
- fast load operation of the whole document.

This approach has also following disadvantages:

- if there exist many keys in a database, the delete operation is slow ($O(n)$),
- decreasing of memory consumption is hardly predictable, since it differs from document to document.

B. Algorithms Optimization

The mapping algorithms were introduced in Section III, as we try to improve performance of the implementation we decided to optimize these mappings. The main idea of the optimization is to reduce the number of database queries, therefore we introduce following new keys:

1) **IDenvironment:IDcollection:IDdocument<namespaces** – is a special key for document namespaces. Its motivation is easier manipulation of a whole document (where the namespaces are not important) and faster querying (the namespace can be found by its key when needed).
2) **IDenvironment:IDcollection:IDdocument<info** – represents the content of the document. The elements and theirs content can be accessed by:
    - **IDroot:IDelement>order** – contains information about a single element - its attributes, text content and ordered set of its children keys.
    - **IDroot:IDelement>order>c>order** – contains the comment as a text.
    - **IDroot:IDelement>order>d>order** – contains the content of CDATA section.

This mapping allows us to query a document effectively as an XPath query can be rewritten directly into keys. Nevertheless some XPath queries (such as "//books") still need to go through the whole document structure.

V. EXPERIMENTS

This section introduces results of several performance experiments. Theirs aim was to show the capabilities of proposed mappings and their limits and to compare RedXML with native XML databases. We have chosen to compare RedXML with native XML databases, because they are very close to RedXML mapping techniques and granularity of stored information is similar. The RedXML was compared to the eXist [15], Berkeley DB XML [16] and BaseX [17]. These databases were chosen as they represent the state of the art in the field of XML databases. According to the measurement results we can choose the best way for future optimizations.

Performance tests were made on two scenarios – document loading and document saving. We used XML generator XMLGen [18] to generate documents used for the performance testing. XMLGen tool provides a switch *-f* to set a size factor of generated document. Documents sizes are shown in Table I.

Copyright (c) IARIA, 2013.

Table I
THE SIZE OF GENERATED DOCUMENTS USED IN THE EXPERIMENTS. THE FACTOR COLUMN REPRESENTS VALUE OF THE XMLGEN'S SWITCH *-f*.

Document name	Size	Factor
0-0-5.xml	1.4MB	0.01
0-1.xml	14.5MB	0.1
0-1-5.xml	21.7MB	0.15
0-2.xml	29.3MB	0.2

The main aim of the tests was to measure performance capabilities and memory usage of Redis database according to proposed algorithms. Both, an optimized and non-optimized, versions of algorithms were used so the optimization contribution can be verified. The measurement was done in two steps. First we did evaluation of the proposed mapping techniques. Second we compared these mappings to common used native XML database systems.

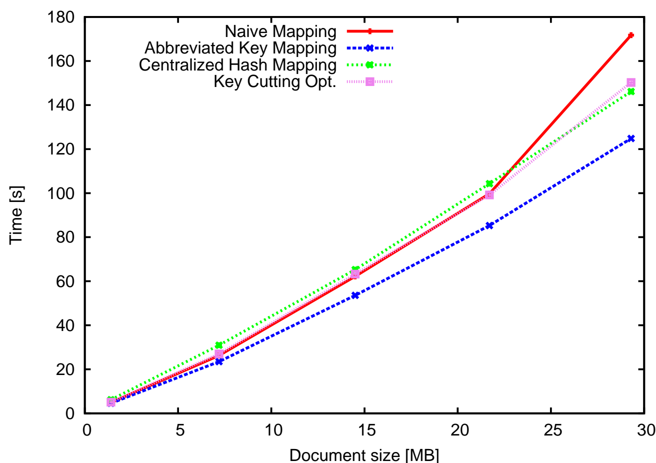

Figure 2. Saving documents of different sizes.

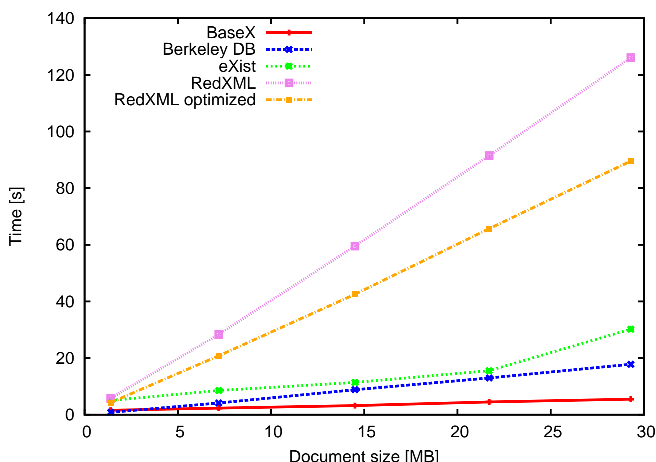

Figure 3. Saving documents of different sizes.

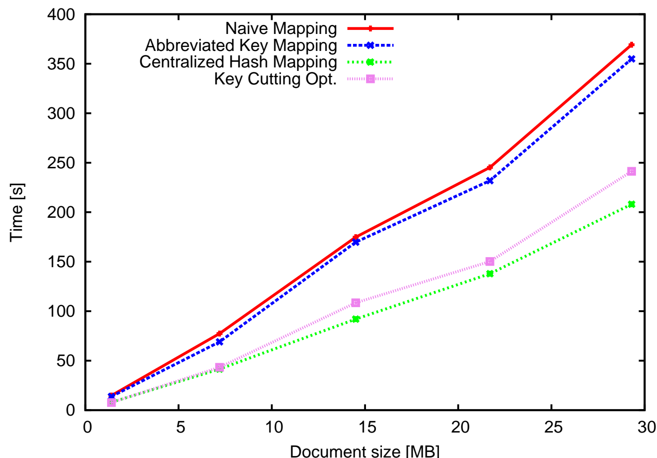

Figure 4. Loading documents of different sizes.

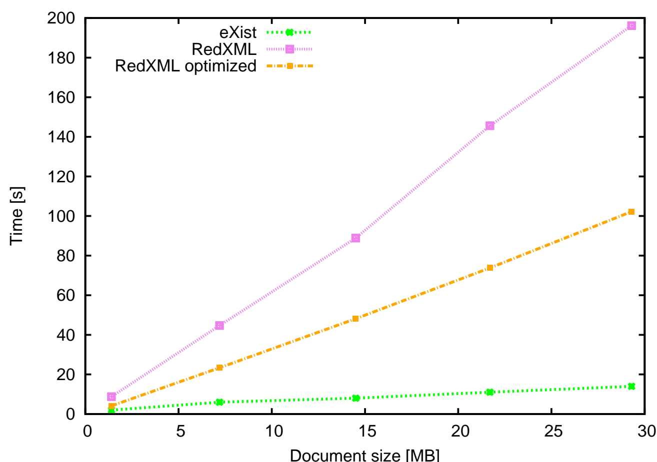

Figure 5. Loading documents of different sizes.

Hardware Configuration: In our experiments we used the following hardware and software configuration:

- Processor: Intel(R) Core(TM) 2 Duo T5500 1.66 GHz
- Memory: 1.5 GB
- Operating System: Debian 6.0.4, Kernel 2.6.32-5-amd64
- Redis version: 2.4.8

A. Databases Performance Comparison Results

The results of the performance tests for the document saving scenario of proposed approaches are shown in Figure 2. It is obvious that Abbreviated Key Mapping is most effective approach when saving large documents. All other mappings perform almost the same way.

Figure 3 shows the performance of RedXML compared to common native XML databases when saving documents. The RedXML performance in optimized version is 25% more effective than unoptimized version. On the other hand

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-247-9

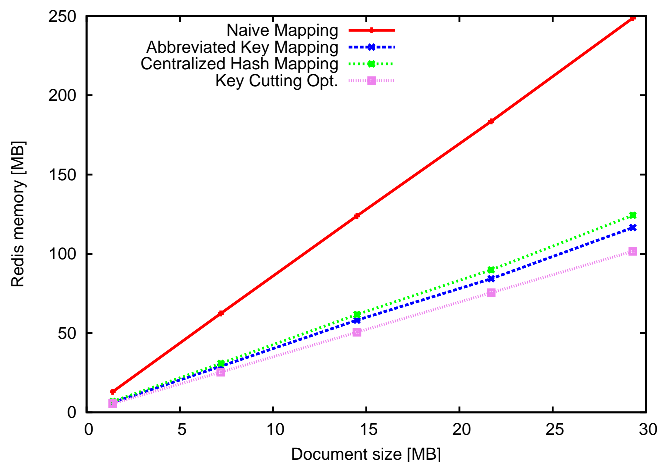

Figure 6. Memory consumption of Redis according to different mapping approaches.

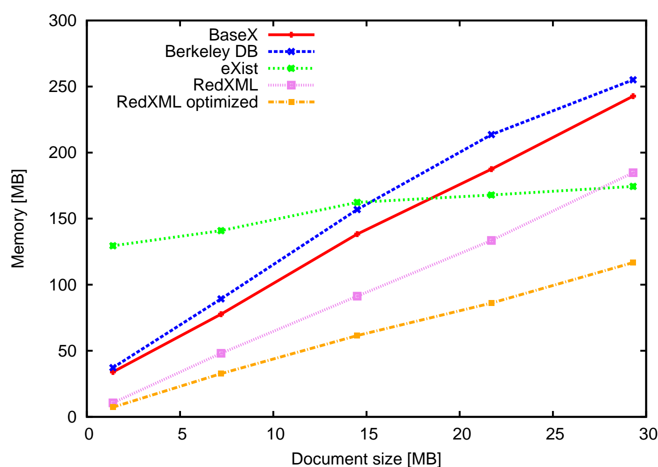

Figure 7. Memory consumption of different DB systems according to document size.

other databases are much more effective when saving a document. This is caused by an early stage of the RedXML project and by the fact that RedXML creates a lot of indices during saving, because of planned support of XQuery queries.

The results for the document loading scenario of proposed mappings are in Figure 4. Centralized Hash Mapping and Key Cutting Optimized algorithms are performing much faster than Naive Mapping and Abbreviated Key Mapping. It shows that the design of CHM and KC was successfuly implemented and it performs as intended.

The comparison with other databases is shown in Figure 5. BaseX and Berkeley DB XML databases were not measured because their document loading time was too short, it can be caused by lazy loading of documents or because mentioned databases saves a whole document so they are able to return it very fast when needed. The eXist database is again faster than RedXML or optimized RedXML, this can be partially caused by the performance difference between Java (eXist) and Ruby (RedXML). To speed up the document loading in the RedXML database the lazy loading or saving a copy of a whole document can be implemented.

The results of memory usage tests for the document saving are in Figure 6; the most effective algorithm is Key Cutting. It is 2.5x faster than Naive Mapping. Centralized Hash Mapping and Abbreviated Key Mapping have also good performance.

Figure 7 shows that database eXist is very effective in memory representation of XML documents in contrast with other databases for large documents. The optimized version of RedXML is most effective from memory usage perspective, even for large documents up to 30 MB it can be more effective than Bekeley DB XML, BaseX and eXist, but we would like to prove this hypothesis for larger documents in the near future. This optimization was focused on memory consumption and this benchmark verifies its implementation.

VI. Conclusion and Future Work

In this paper we have proposed four mappings of XML documents to a key-value database, next we have provided their implementation and optimization for key-value database Redis. Resulting XML-enabled database is called RedXML. RedXML can be easily deployed on multiple computers to achieve high scalability thanks to Redis platform. According to solutions provided in related-work we provide the solution that is dependent only on a few Redis commands which have specified time complexity. Hence, if we optimize these commands we can achieve better performance of RedXML.

The RedXML database was compared to several native XML databases from performance and memory consumption point of view. The results are promising especially according to memory consumption. In other benchmarks RedXML is not performing as well as other database systems.

The future work on RedXML will focus on the optimization of XML document querying as the main goal of the project is to provide effective XQuery implementation. Next research will focus on optimization of mapping an XML document into key-value database according to known queries over the mapped XML document.

References

[1] “Redis,” Available: http://redis.io 21.9.2012.

[2] A. Vaidya and A. Gopal, “XML Representation using Hash Key for Performance Improvement,” *International Journal of Computer Science and Application*, no. 2010, 2010.

[3] “Oracle XML-Enabled Technolgy,” available: http://docs.oracle.com/cd/B10464_05/web.904/b12099/adx01int.htm 27.9.2012.

[4] "IBM - DB2 database software," Available: http://www-01.ibm.com/software/data/db2/ 27.09.2012.

[5] A. Deutsch, M. Fernandez, and D. Suciu, "Storing semistructured data with STORED," in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '99. New York, NY, USA: ACM, 1999, pp. 431–442.

[6] D. Florescu and D. Kossmann, "A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database," INRIA, Rapport de recherche RR-3680, 1999.

[7] M. Klettke and H. Meyer, "Xml and object-relational database systems - enhancing structural mappings based on statistics," in *ACM SIGMOD Workshop on the Web and Databases*, 2000, pp. 63–68.

[8] P. Bohannon, J. Freire, P. Roy, and J. Simeon, "From XML Schema to Relations: A Cost-Based Approach to XML Storage," *ICDE*, vol. 00, p. 64, 2002.

[9] A. Chebotko, M. Atay, S. Lu, and F. Fotouhi, "Xml subtree reconstruction from relational storage of xml documents," *Data Knowl. Eng.*, vol. 62, no. 2, pp. 199–218, 2007.

[10] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, "XQuery 1.0: An XML Query Language," 2007.

[11] T. Bray, F. Yergeau, J. Cowan, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, "Extensible Markup Language ({XML}) 1.1," 2004.

[12] P. Jíra, "Transformation of XML into key-value database Redis," Master's thesis, Czech Technical University in Prague, Czech Republic, 2012. [Online]. Available: https://dip.felk.cvut.cz/browse/pdfcache/jirapave_2012dipl.pdf

[13] P. Jíra and P. Strnad, "RedXML Concept," Available: https://github.com/jirapave/RedisXmlConcept 21.9.2012.

[14] "Redis Memory Optimization," Available: http://redis.io/topics/memory-optimization 21.9.2012.

[15] "eXist Project Homepage," Available: http://www.exist-db.org 21.9.2012.

[16] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley db," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 43–43.

[17] C. Grün, S. Gath, A. Holupirek, and M. H. Scholl, "Xquery full text implementation in basex," in *Database and XML Technologies, 6th International XML Database Symposium, XSym 2009, Lyon, France, August 24, 2009. Proceedings*, ser. Lecture Notes in Computer Science, Z. Bellahsene, E. Hunt, M. Rys, and R. Unland, Eds., vol. 5679. Springer, 2009, pp. 114–128.

[18] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse, "The XML Benchmark Project," CWI, Amsterdam, The Netherlands, Tech. Rep. INS-R0103, 2001.

Copyright (c) IARIA, 2013. **ISBN: 978-1-61208-247-9**