A study on BaseX and Benchmarking *

Wouter Lauret
wlauret@cs.uu.nl

Antonis Thomas
athomas@cs.uu.nl

Abstract

In this paper we analyze the architectural design of the native XML database system BaseX. In addition, we discuss how to perform beneficial benchmarking on such databases. We run experiments and compare the performance of BaseX to two other XML databases, namely MonetDB/XQuery and eXist. The experiments, their settings and the comparison are given in full detail in order to gain deep intuition on the performance of these systems. We show that for the larger set of our chosen test queries, BaseX performs best and is only outperformed by MonetDB/XQuery or eXist in very specific situations. We conclude that BaseX is a prototype, promising to become a worthy and efficient database management system.

*Assignment report for the Database Architecture Seminar, Department of Information and Computing Sciences, Utrecht University

1. Introduction

XML has been developed into the standard for universal exchange of textual data. Many XML documents are still small, but because of this development, they grow larger and larger in size and usage, exceeding main memory constraints. With the size growth, there comes an increasing request for XML as storage format for large data volumes. The tree structure of an XML document provides for plenty of well-known and efficient techniques to handle data. A popular format for storing XML data is through the use of relational encodings. It has been proven to be efficient in both terms of speed and memory usage.

This paper describes and evaluates three systems for storaging and querying XML formatted data. The main focus, as the title suggests, is given to BaseX. We run a range of experimentations from the widely referenced XMark benchmark project against three XML database management systems. A set of six queries was chosen for evaluation as to gain insight into the advantages and the limitations of BaseX, in comparison to MonetDB/XQuery and eXist.

BaseX is a database prototype of a table based tree encoding of XML nodes. The main focus of BaseX on achieving high performance is set on a memory saving representation of the original document. In this paper, we will put the results of the two other databases against the results we get for the BaseX database. The paper will analyze the architecture of the BaseX database management system and tries to justify the difference in performance based on a general view of the architectural differences.

The rest of this paper is outlined as follows. In Section 2, we review the architectural design of BaseX. In Section 3, we explain how to beneficially benchmark XML databases and give our choices of benchmarking experiments. In Section 4, we give details on the experimental settings and outline the architecture of both MonetDB/XQuery and eXist. Then, we provide with the actual results and some comments. In Section 5 and 6, we conclude our paper with an evaluation of our results and this work . Finally, Appendices A and B contain graphs plotted on our results and the code and description of the queries we run, respectively.

2. Inside BaseX

BaseX is an open source native XML database. It is a native XML database in the sense that it defines a logical model for an XML document. The model inside BaseX follows a table based tree encoding of XML nodes [6]. It stores and retrieves documents in the database, according to that model. In this section we shall have a look at how this model is composed and how BaseX takes advantage of it.

Different structures have been reviewed in order to implement an efficient model, which can map XML documents into such a structure. Since an XML document is composed in a fairly accessible structure, the tree, it is a natural decision not to evade from this model. The most popular idea therefore was to take the hierarchical structure of the tree and rebuild it in main memory [5]. This approach has been standardized to what now

is known as the Document Object Model (DOM) [9]. A straightforward implementation would implement all tree nodes as specific data structures, including references to the dependent nodes. This implementation has some downsides, however. For instance, if every node has its own data structure there is more information stored than needed and memory usage gets inefficient. This is why BaseX has the document storage in a relational encoding [4].

BaseX contains a relational XML document encoding as an integrated XPath processor. This way, encoded XML documents are represented in relational tables, consistent with the RDBMS architecture. XPath queries can be mapped to "regular" SQL queries through these tables. This also ensures that we can efficiently index documents according to index structures native to RDBMS's, such as B-trees. The resulting XPath processor supports all 13 XPath axes as defined by the W3C organization [3]. It applies query algorithms which have the *Staircase Join operator* as their main architecture [8]. The Staircase Join operator is tuned to exploit the logical model of XML encoding, stored in relational tables. In order to build the tree map visualization and allow user interaction, the XPath processor uses very similar algorithms. Standard B-Trees support the evaluation of staircase joins and the query optimizer may treat the staircase join as it would treat other join operators on relational data.

The current BaseX version is disk-oriented, but in order to efficiently process changing documents BaseX can also be fully run in main memory. This means that it was of high importance to minimize the main memory consumption of all data structures. The main focus of the model to obtain good performance is on encoding the original representation of the XML tree as much as possible in order to preserve memory. At the core of BaseX, is the relationally encoded node table in which all the XML data of the document is stored and referenced. The model only encodes the essential node attributes which are needed for a complete and yet quick traversal of the XML tree. The main demands state that, with the stored information, it should be possible to reconstruct the original document and that query efficiency is preserved. All information that not contributes to the two main demands is skipped. Another characteristic of the BaseX database is that there is no DTD or XML schema required to encode an XML document.

BaseX only stores a reference to the parent ID, the node kind and a string reference. With these three attributes the XPath processor is able to restore XML instances and to process all 13 XPath axes. The ID of the node can be implicitly retrieved from its position in the table. More specifically, not the node ID itself is stored but the distance from the root to the node, since this is very beneficial for updates. The string is numerically encoded, which points to a kind specific index. This representation can be compressed even further by merging attributes [6]. As a result, BaseX is able to store a tuple in only eight bytes. XML attributes are stored in a format of name and value combination. As long as the difference of names and the distance to the element node is not too big, the attributes can be stored together in only eight bytes. The fact that nodes are stored in such a compact size results in very fast data access in both main memory and disk.

BaseX also supports the ability of index construction for attribute values and text nodes. These indexes are useful in accelerating value and text based queries. In the

internal functions of BaseX, queries are rewritten in order to first access the indexes before XPath backwards traversal. This is often very beneficial in content based queries as it can lead to a significant speed up. BaseX is a database prototype, representing XML documents in a table based fashion. The main target of BaseX is delivering fast results via compression. The fixed node size in BaseX makes it usable for main as well as secondary storage. BaseX is not a complete database management system yet and future work, planned for the project, might boost up performances even further.

3. Benchmarking XML Databases

A benchmark for XML databases needs to address specific points. In [11] a set of challenges is given, focused on the actual performance of such a database, which we will first discuss and then use to our own interest. We will give the most important of the challenges from [11] which aim at providing a comprehensive and conclusive analysis covering all performance critical aspects of processing XML databases.

- *Bulk loading.* In XML databases bulk loading assumes a very important role as most of the existing systems support "insert" only on a document level. Different models imply different levels of granularity when shredding the document and this seemingly simple operation may entail severe costs to setup and maintain indices or constraints.

- *Reconstruction.* Reconstructing the original document is the counterpart of bulk loading. It reveals the price of achieving lossless storage of the document, it is not a common operation in most scenarios but still necessary.

- *Path traversals.* One of the most basic and natural operations on a structured document is specifying paths. Efficient path traversal is a fundamental need of an XML database system, not only as a standalone operation but also as a part of more complex ones.

- *Ordered access.* Order is an omni-present feature when querying XML and it affects all aspects of data management. Sophisticated and flexible treatment of the document order should ensure that it is well integrated into the optimization proves up to a degree that it is ignored when not needed.

- *References.* An important modeling primitive is references. Chasing references requires efficient access methods supporting random access across the document rather than navigational access.

- *Joins.* The difficulties posed by joins are well-known from relational database systems. Data-centric applications require the combination of data based values. Modern XML database systems should be able to tackle efficiently with such operations.

- *Containment, full-text search.* These are elementary operations when querying XML. The problem and its intrinsic difficulties are well-known from several application domains.

In the experiments we describe in the next section, we picked and used the XMark benchmark project. It provides an open source document generator that outputs XML document files, which we will use to create the the benchmark databases. The document produced is modeled after a database as deployed by an Internet auction site. A full description of the generated file structure can be found at [12]. XMark is built with the above challenges in mind.

The queries we decided to use for our experiments cover the full spectrum of the above points. They form an indicative subset of the ones proposed in [12]; we had to turn some of them down mainly due to time limitations. On the other hand, the queries we run and present in the next section are enough to give a feeling about the performances of the XML database systems we will deal with. They include simple and complex index lookups, path expressions that need to prune irrelevant parts of the tree, expensive join operations, reconstructing parts of the document and full-text containment search. Extensive description of the queries we used and the corresponding XQuery code can be found in Appendix B.

4. Experiments

The benchmarking experiments were run on an average home computer system. Unfortunately, this results in not enough resources to stress the systems under consideration to their limits. Then again, one might argue that such a system provides more intuitive results and is closer to the average case of a small to medium business or home users. The exact specifications of our system can be found in Table 1.

CPU	Intel Core2 Duo @2.4Ghz, 4MB Cache
RAM	3GB DDR2
OS	Linux 2.6.31 x86-64

Table 1: Specifications of our running system

To enhance the actual meaning of the results we compared BaseX to two other XML database systems: MonetDB/XQuery and eXist. Note that all the systems used are open source. The experiments were run on out-of-the-box (i.e. not optimized) configuration. The output for the queries was sent to `/dev/null` device, that discards all the input it receives, to eliminate the time needed to print the results on screen. In addition, MonetDB/XQuery had to be compiled on our running system whereas for BaseX and eXist we used the java binaries. A general view on the architectures of the two systems is following:

MonetDB/XQuery is based on the MonetDB database kernel and uses Pathfinder as an Xquery compiler. The Pathfinder system compiles Xquery to relational algebra and feeds the underlying database system, which exploits MonetDB4's vertical fragmentation techniques. This results to a column-store database that is built from XML documents and can run XQuery queries by translating them to relation algebra. Further details on Pathfinder can be found at [7] and details on MonetDB4 and its interpreter language can be found at [2]. Note that MonetDB5 is already released but is built on different architecture that does not support Pathfinder and thus MonetDB4 is still used for the MonetDB/XQuery system. We used the stable version FEB2010SP2, since the actual last version JUN2010 was retracted due to serious bugs.

eXist provides its own XQuery implementation, which is backed by an efficient indexing scheme at the database core to support quick identification of structural node relationships. Since 2006, the indexing scheme the eXist authors chose to implement is the *dynamic level numbering* as proposed in [1]. This numbering scheme is based on variable-length ids and thus avoids to put a conceptual limit on the size of the document to be indexed. eXist system is claimed to be web-service oriented. Details on how the system works can be found at [10]. We used the stable version 1.4 (Codename: Eindhoven).

4.1. Results and comparison

The experiments were run on databases created by XML documents that were produced by the data generation tool of the XMark project. The instances of the documents we generated and used can be found in Table 2.

Name	Scaling Factor	Document Size
Small	0.1	11MB
Medium	1	111MB
Large	5	555MB

Table 2: The instances of XMark we used for our experiments

In addition, note that we also generated a 1.1GB instance. Our running system did not have enough resources for either MonetDB/XQuery or eXist to open this document. On the other hand, BaseX was capable of opening it and successfully running the same queries on the larger instance. The following table aggregates all the running times of BaseX, including the extra large instance. Times are given in milliseconds.

For our experiments, all the database systems where run in a client/server with no GUI mode. This was to minimize external effects on the database systems and much interaction with the operating system or the window manager. The queries were run several times and our results indicate the average. Before executing each query we were running some random ones to ensure that the main memory does not already contain

Doc	q1	q2	q3	q4	q5	q6
11MB	4	22	5	132	10	32
111MB	9	179	7	1517	114	261
555MB	13	794	10	8845	505	1274
1111MB	15	1564	16	18494	953	2543

Table 3: The time each query took in milliseconds when run on BaseX

the results we asked for. This is mainly due to the java-based systems, BaseX and eXist, where the garbage collector has to be evoked to empty the memory. An increase in performance could be observed if the same query was run many times in a row. This effect was less significant with MonetDB/XQuery.

Furthermore, we plotted the above running times in a bar graph that can be seen in Figure 1. Note that the vertical axis is on a logarithmic scale. The running times for each query are compared with respect to the document size. They can be seen to grow logarithmically as the document grows exponentially. It should be noticed that between the 111MB and 1111MB documents the 555MB document intervenes and that is why the growth seems unbalanced at that point.

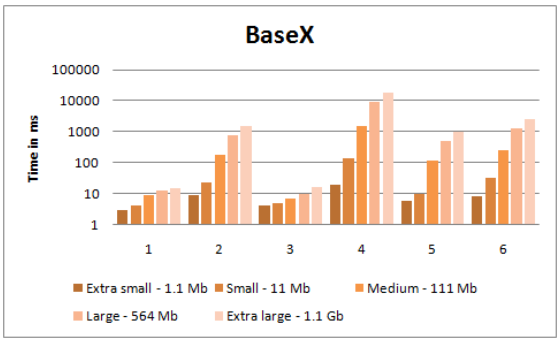

Figure 1: The running times of BaseX for all the queries and all the document sizes.

The focus of our benchmark is comparison of BaseX with the two alternative XML database systems. To present our results we plotted graphs that directly correlate the performance of MonetDB/XQuery and eXist with BaseX. The graphs can be found in Appendix A. The base of the graphs is the performance of BaseX and the bars indicate how well the alternative systems perform relatively to that. Note that the vertical axis is again on a logarithmic scale. Therefore, a bar at height Y in the graphs means that BaseX is Y times faster than the alternative system (thus, if $Y < 1$, BaseX is slower).

From the graphs we gain valuable insights on the performance of the systems under consideration. First of all, concerning eXist we observe that its performance is only acceptable in the small instance. It's because of its web-service oriented character that eXist performs better on even smaller files and is optimized to work with such examples.

Concerning BaseX we can see that its performance is the best in the comparison in most cases. The general architectural design of BaseX and MonetDB/XQuery is similar, however, BaseX performs better because of its controlled usage of memory. MonetDB/XQuery is stronger for the expensive join query (4) because of its loop lifting techniques. BaseX uses index stuctures to speed up such queries which is somewhat slower but ensures constant memory usage. On the other, MonetDB/XQuery seems to have trouble with the "contains" query (6), where BaseX and even eXist perform better in some cases. As a final remark on the comparison, BaseX seems to perform best overall, although it is not as a complete system and should rather be seen as a database prototype.

A similar benchmark, from the perspective of MonetDB/XQuery can be found in [13]. This dates back to 2006 and all of the systems have changed or evolved since then. In addition, the experiments in [13] were run on a powerful server system. Context and hardware differ between both situations and thus comparison of the two benchmarks should be avoided. Our benchmark can be seen as addition, including BaseX and run on an average PC. In that sense, the two benchmarks complement each other.

5. Conclusion

BaseX is a database prototype, working on a compact table based representation of XML documents. The fixed node size is equally suitable for use in main and secondary storage. In the scope of our experiment, we tried to push the BaseX database management system to its limits by exposing it to a range of queries, each one taken from a different scope of information retrieval. In order to evaluate its performance, we put it up to two other database management systems, namely MonetDB/XQuery and eXist. We have executed a set of six queries on the three database systems, using four documents of different size. We compared runtimes of these queries in order to see how BaseX performs on different queries and different database sizes.

We can conclude that, even though it is still in a prototype phase, BaseX promises to be an excellent performing database management system or be used as the core for a production ready commercial system. It manages to outperform both MonetDB/XQuery and eXist on most types of queries and all databases of significant size. Only on the join query, Monet still proves to be on the upper hand. eXist only proves to be of any value in case of very small databases. In all other cases, the compact node table structure of BaseX makes it an optimal structure for data retrieval and manipulation.

6. Main contributions

BaseX is still in a prototype phase, and many improvements are yet to be made. This paper shows that BaseX is good on its way of becoming a complete, worthy and efficient XML database. Possible improvements are to be obtained in the field of join queries. However, it should be noted that, since it's still a prototype, BaseX does not have all planned functionalities implemented yet. When these are to be implemented, it may

result in changes to the data structure which may in turn result in loss of optimality. Therefore, our work can be seen as a guide in future development of BaseX.

References

[1] T. Böhme and E. Rahm. Supporting efficient streaming and insertion of XML data in RDBMS. In Z. Bellahsene and P. McBrien, editors, *DIWeb*, pages 70–81, 2004.

[2] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999.

[3] J. Clark and S. DeRose. XML path language (XPath). `http://www.w3.org/TR/xpath/`, 11 1999.

[4] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[5] C. Grün. Pushing XML main memory databases to their limits. In S. Brass and A. Hinneburg, editors, *Grundlagen von Datenbanken*, pages 60–64. Institute of Computer Science, Martin-Luther-University, 2006.

[6] C. Grün, A. Holupirek, M. Kramis, M. H. Scholl, and M. Waldvogel. Pushing xpath accelerator to its limits. In P. Bonnet and I. Manolescu, editors, *ExpDB*. ACM, 2006.

[7] T. Grust, J. Rittinger, and J. Teubner. Pathfinder: Xquery off the relational shelf. *IEEE Data Eng. Bull.*, 31(4):7–14, 2008.

[8] T. Grust and M. van Keulen. Tree awareness for relational DBMS Kernels: Staircase Join. In H. M. Blanken, T. Grabs, H.-J. Schek, R. Schenkel, and G. Weikum, editors, *Intelligent Search on XML Data*, volume 2818 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2003.

[9] P. L. Hégaret, R. Whitmer, and L. Wood. Document object model (dom). `http://www.w3.org/DOM/`, 1 2009.

[10] W. Meier. Index-driven XQuery processing in the eXist XML database. `http://exist-db.org/xmlprague06.html`, 6 2006.

[11] A. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse. Why and how to benchmark XML databases. *SIGMOD Record*, 30(3):27–32, 2001.

[12] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical report, Amsterdam, The Netherlands, 2001.

[13] M. Team. XMark 1MB-11GB comparison with eXist, X-Hive, Galax and Berkeley DB-XML. `http://monetdb.cwi.nl/XQuery/Benchmark/XMark/index.html`, 2 2006.

A. Appendix: Graphs

This appendix provides the query execution time results in milliseconds for all the systems we tested, for the six queries we used (can be found in Appendix B), for all sizes of the XMark document (from 1.1MB to 555MB). The graphs below in this page show the execution speed of the alternative systems relative to BaseX (on a logarithmic scale). Thus, a bar at height Y in the graphs means that BaseX is Y times faster than the alternative system (thus, if $Y < 1$, BaseX is slower).

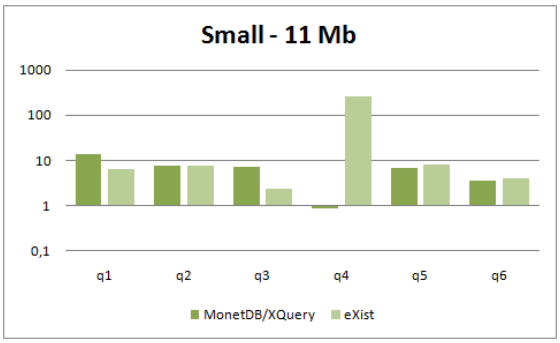

Figure 2: The relative running times for the small -11MB- document.

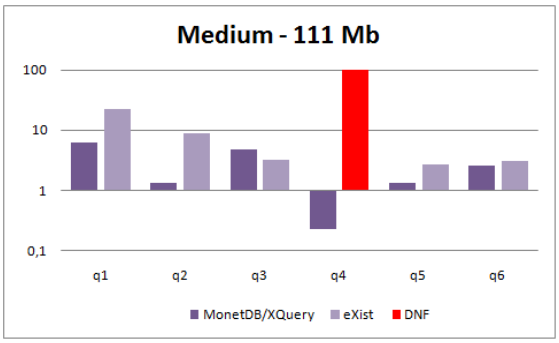

Figure 3: The relative running times for the medium -111MB- document. DNF stands for did not finish in one hour.

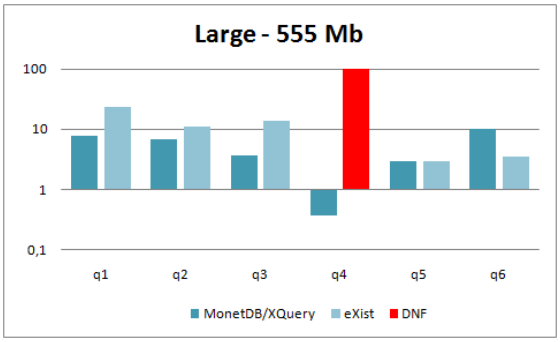

Figure 4: The relative running times for the large -555MB- document. DNF stands for did not finish in one hour.

B. Appendix: Query overview

In this appendix we will give full description and exact XQuery code for the six queries we run for our experiment. These queries are an indicative subset of the ones originally described in [12].

Query 1 returns the name of the person with ID 'person0'. It tests the database ability to handle simple string lookups with a fully specified path.

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

Query 2 returns the first and current increases of all open auctions whose current increase is at least twice as high as the initial increase. It is a complex application of index lookups.

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2 <=
                        $b/bidder[last()]/increase/text()
return
  <increase
   first="{$b/bidder[1]/increase/text()}"
   last="{$b/bidder[last()]/increase/text()}"/>
```

Query 3 returns the number of all items on all continents. It investigates how well the query processor can optimize path expressions and prune traversals of irrelevant parts of the tree.

```
let $auction := doc("auction.xml") return
for $b in $auction//site/regions return count($b//item)
```

Query 4 returns the names of persons and the number of items they bought. It joins person, closed_auction and is the most expensive query in our study.

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

Query 5 returns the names of items registered in Australia along with their descriptions. It tests for the ability of the database to reconstruct portions of the original XML document.

```
let $auction := doc("auction.xml") return
for $i in $auction/site/regions/australia/item
return <item name="{$i/name/text()}">{$i/description}</item>
```

Query 6 returns the names of all items whose description contains the word "gold". It conducts a full-text search in the form of keyword search.

```
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```