Pushing XML Main Memory Databases to their Limits

Christian Grün
Database & Information Systems Group
University of Konstanz, Germany
christian.gruen@uni-konstanz.de

The wide distribution of XML documents and the standardization of the Query languages XPath and XQuery have led to a wide variation of XML database implementations. Yet the efficient processing of really large XML documents is still supported by just a few products such as e.g. MonetDB/XQuery as open-source solution [1] or X-Hive as commercial product [2]. Following the main memory and relational encoding approach of MonetDB/XQuery, we analyse how far we can push main memory approaches to yield usable real-time query results for XPath requests. Moreover, we apply memory efficient value indexes to reach an order-of-magnitude performance improvement for queries with predicate tests. The results of our implementation prototype are quite promising, surpassing the response times of any other implementation we tried so far.

1 Introduction

The XML Standard opens way to a wide range of text-processing facilities such as a unified exchange of text documents, the support of hierarchically structured data or a stream-based distribution and parsing of textual information. Many XML documents are still small, compared to the size of datasets which are stored in relational databases. However, XML is increasingly chosen as a storage format for extensive and complex text data. Popular examples are the DBLP database for scientific publications [3], the protein database SwissProt [4] or the Wikipedia [5].

Parsers like e.g. Galax [6] or Saxon [7] use a sequential access to XML documents and build a temporary main memory tree structure to answer queries. This approach works well for small documents, but the size of processable documents is limited; files with 100 or more MB get very slow or cause problems, even on systems with well-equipped main memory. In the scope of our research, we deal with the efficient storage and query of especially large XML documents. Files up to 13 GB have been successfully processed and queried by BaseX, our prototype implementation. Despite all we still share one aspect with parsers like Galax – we only work in main memory.

2 XML Mapping

Different data structures have been proposed to map XML documents into a more accessible structure. The most popular approach is to adopt the hierarchical structure of a document and rebuild it as a main memory tree. This approach has been standardized as Document Object Model (DOM). In most implementations, the tree nodes are implemented as specific data structures, referencing their dependent nodes and providing further information on their type. The tree-like structure is the most natural way of mapping XML documents as it can be easily accessed and modified, but it imposes quite a number of restrictions:

- XPath axes such as descendant, ancestor, preceding and following need to be parsed recursively, causing a bad runtime complexity
- the memory usage gets inacceptable for larger documents as each document node has its own data structure and offers a lot of information that can be derived from other document nodes

- tree structures are usually accessed by the root node. Additional data structures are needed to randomly access any nodes of the tree, causing additional memory overhead

An approach that has proven to be successful is the document storage in a relational encoding [8]. The MonetDB/XQuery implementation is based on the XPath Accelerator [9]. The current encoding is based on a `Pre/Size/Level` encoding, derived from the `Pre` and `Post` values. These values can be created during a sequential parsing of the XML instance. For each opening tag and content node, a new `Pre` value is assigned, and the `Post` value is assigned when a tag or node is closed (see Fig. 1).

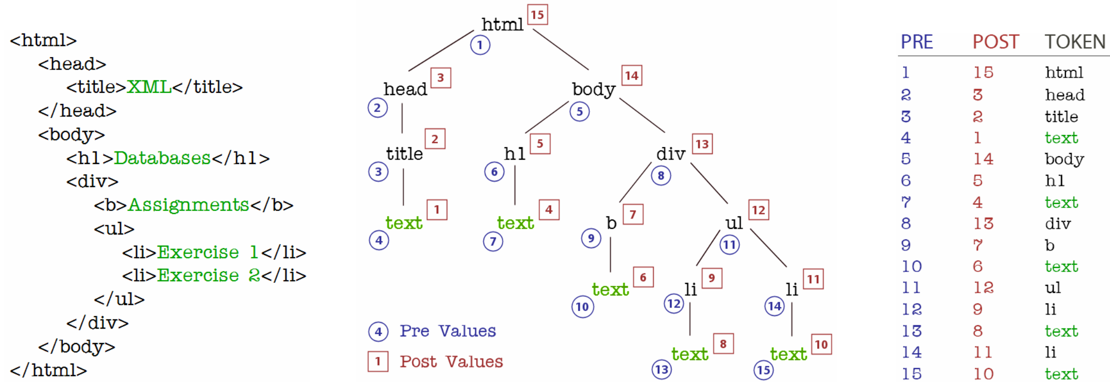


Figure 1: XML document, Tree Representation, Pre/Post Table

The relational encoding of XML documents facilitates efficient access to the major XPath axes `descendant`, `ancestor`, `following` and `preceding`. To comply with two principal affordances of the XPath specification, namely the orderedness of context sets and the absence of duplicate nodes in the result set, the Staircase Join algorithms were introduced [9], allowing a linear parsing of the major axes.

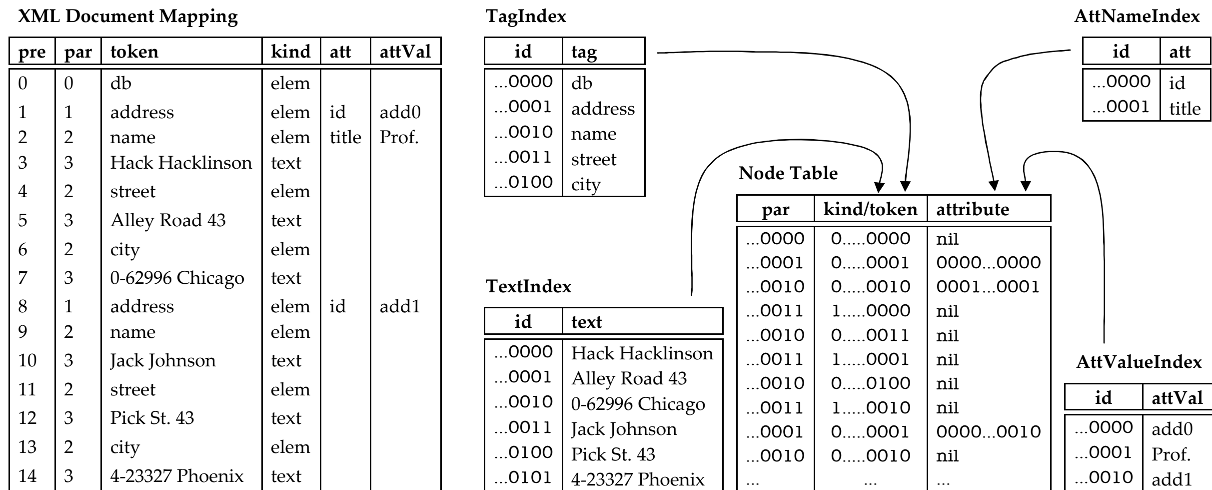
In our prototype we are working with a simple edge-based mapping, referencing the `Pre` and `Parent` node to further accelerate ascending traversals. The Staircase Join algorithms have been slightly modified to work with the chosen encoding.

3 Data Structures

BaseX is completely designed as a main memory application, so minimizing the memory consumption of all data structures was highly relevant. The core data of BaseX is the relationally encoded node table in which all XML data is stored and referenced. The node table for the XML document in Figure 2 is shown Figure 3. The table assembles the `Pre` and `Parent` value, the tag name or text content, the node kind and optional attributes. Tags & text contents, attribute names and attribute values (so-called Tokens) are uniformly indexed in separate hash structures; the index entries are referenced by integer values. The internal table representation, shown on the right in the figure, stores all information as integer values and merges some attributes to save memory. The `Parent` value is stored in an integer array, and the `Pre` value is implicitly given by the array position. The table further stores the tag name or text content by its integer reference and shares its uppermost bit with the node kind (0 for element, 1 for token). As a tag can have several attributes, the attributes are referenced by two-dimensional arrays. Though, the attribute name and value use again only one integer, sharing 10 and 22 bit; a `nil` reference is used when no attributes are given. The original values can be efficiently processed and accessed via CPU-supported bit-shifting algorithms.

```
<db>
  <address id='add0'>
    <name title='Prof.'>Hack Hacklinson</name>
    <street>Alley Road 43</street>
    <city>0-62996 Chicago</city>
  </address>
  <address id='add1'>
    <name>Jack Johnson</name>
    <street>Pick St. 43</street>
    <city>4-23327 Phoenix</city>
  </address>
</db>
```

Figure 2: Sample XML document



XML Document Mapping

pre	par	token	kind	att	attVal
0	0	db	elem		
1	1	address	elem	id	add0
2	2	name	elem	title	Prof.
3	3	Hack Hacklinson	text		
4	2	street	elem		
5	3	Alley Road 43	text		
6	2	city	elem		
7	3	0-62996 Chicago	text		
8	1	address	elem	id	add1
9	2	name	elem		
10	3	Jack Johnson	text		
11	2	street	elem		
12	3	Pick St. 43	text		
13	2	city	elem		
14	3	4-23327 Phoenix	text		

TagIndex

id	tag
...0000	db
...0001	address
...0010	name
...0011	street
...0100	city

TextIndex

id	text
...0000	Hack Hacklinson
...0001	Alley Road 43
...0010	0-62996 Chicago
...0011	Jack Johnson
...0100	Pick St. 43
...0101	4-23327 Phoenix

Node Table

par	kind/token	attribute
...0000	0.....0000	nil
...0001	0.....0001	0000...0000
...0010	0.....0010	0001...0001
...0011	1.....0000	nil
...0010	0.....0011	nil
...0011	1.....0001	nil
...0010	0.....0100	nil
...0011	1.....0010	nil
...0001	0.....0001	0000...0010
...0010	0.....0010	nil
...	...	...

AttNameIndex

id	att
...0000	id
...0001	title

AttValueIndex

id	attVal
...0000	add0
...0001	Prof.
...0010	add1

Figure 3: Relational XML mapping (left), internal table representation in BaseX (right)

A second data structure, the hash index, is basically an implementation of a linked list hash structure. Similar to the node table, we flattened the index to work on simple integer arrays. The hash arrays are all sized by a power of two, and the bitwise `AND` operator (&) cuts down the calculated hash values to the array size; this approach works faster than conventional `modulo` (%) hash operations. Next, the array size does not rely on the calculation of prime numbers, and the index can be quickly resized and rehashed during the index creation.

Three arrays suffice to store all hash information (see Figure 4). The `TOKENS` array references the indexed tokens. The `ENTRIES` array references the positions of the first tokens, and the `BUCKET` array maps the linked list to an offset lookup. After a hash value has been calculated for the input token and trimmed to the array size, the `ENTRIES` array returns the pointer to the `TOKENS` array. If the pointer is nil, no token is stored; otherwise the token is compared to the input. If the comparison fails, the `BUCKET` array points to the next `TOKENS` offset or yields nil if no more tokens with the same hash value exist.

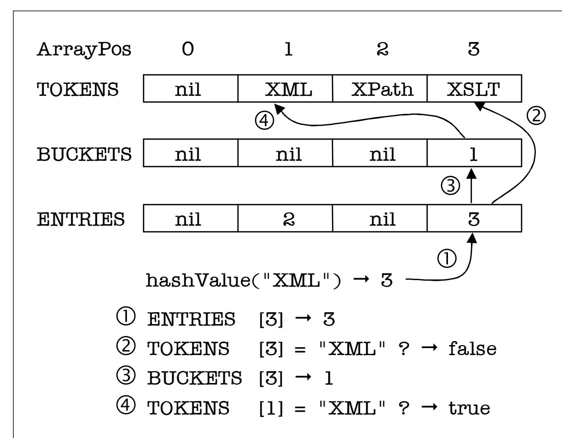


Figure 4: Sample XML document

Thanks to the optimizations, the main memory representation of an XML files occupies only 0.8 to 2.3 the size of the original document on disk. As indexes are used for all tokens anyway, the value index (which will be discussed in the next section) just represents 12-18% of the main memory data structure.

4 Querying

BaseX can both be run as a command-line query tool and as a client interface, sending requests for loading and querying documents to a server instance. The implemented XPath parser processes basic, non-recursive queries, including all location steps, kind tests and predicates. Different algorithms are chosen for queries with or without positional predicates, and some optimization steps are performed to simplify and reformulate the XPath query.

The first rewriting step combines location steps that would otherwise be parsed separately. As an example, the double slash (//) offers itself for being optimized. A query like `//step` is the abbreviated writing for the query `descendant-or-self::node()/child::step`. Before the `child` step is evaluated, a

No	Source	Query	Hits
01	DBLP	/dblp/www[./editor]/url	6
02	DBLP	//www[./editor]/url	6
03	DBLP	//inproceedings[./author/text()="Jim Gray"][./year/text()="1990"]/@key	5
04	DBLP	//book/author[text() = "C. J. Date"]	13
05	Swissprot	//inproceedings[./title/text() = "Semantic Analysis Patterns."]/author	2
06	Swissprot	//Entry/Keyword[text() = "Rhizomelic chondrodysplasia punctata"]	3
07	Swissprot	//Entry/PFAM[@prim_id = "PF00304"][..//DISULFID/Descr]	6
08	Swissprot	//Entry[./Org/text() = "Piroplasmida"]//Author	57
09	Wikipedia	/mediawiki/page[title/text() = 'XML']	1
10	Wikipedia	//page[title/text() = 'XML']	1
11	Wikipedia	//page[revision/contributor/username/text() = 'Chuck Smith']	14
12	Wikipedia	/mediawiki/page/title/text()[. >= 'Q' AND . < 'R']	6993
13	Wikipedia	/mediawiki/page/title/text()	2504732

Table 1: Main DBLP, Swissprot and Wikipedia Queries

large, intermediate result set is created, including all context nodes of a document. By merging this step into a descendant step, the step traversal can be distinctly accelerated. A similar optimization can be performed for self axes which can be pruned away in many cases. The implemented optimizations guarantee equivalent and correct result sets, but they cannot be applied to all queries, excluding e.g. location steps with positional predicates.

The most powerful rewriting process includes the value index into the query. As indicated before, we integrated a general value index for all text nodes and attribute values to speedup predicate selections. The index structure was enhanced with an inverted list, pointing to the token's Pre values. When predicates with string matches are encountered, the input XPath query is rewritten to call the value index. Descendant steps are converted to ancestor steps and vice versa. A simple query reformulation is shown for the following example query: doc("doc.xml")//address[@id = "add0"]/name. The string "add0" is first matched against the attribute value index. Next, a parent step is added for the address tag, and the root node test is moved into a predicate. Finally, the name child step is evaluated, yielding the final result set. The internal query reformulation can thus be represented as index::node()[attribute::id = "add0"]/parent::address[ancestor:root()]/child::name.

5 Performance

For our performance tests we used three XML instances: DBLP, Swissprot [10], and Wikipedia [5][1]. The XPath queries are shown in Table 1; they are similar to those evaluated in [11, 12]. All tests were performed on a dual 64-bit Opteron (using one processor) with 2.2 GHz and 16 GB RAM and the SUSE 10.1 operation system. BaseX was implemented in Java, using the 64-bit version of JDK 1.5.0_06.

We performed all queries on the current BaseX version and the 64-bit version of MonetDB/XQuery 4.8.20 which currently provides unrivalled performance for large XML documents [13]. Each query was evaluated ten times, and the best execution time was used as result. The execution time includes the time for compiling and processing a query and serializing the result.

The performance results are visualized in Figure 5. BaseX is shown twice here, including and excluding the value index. As can be seen in the results, the execution time is by order of magnitudes faster when the index is applied to the query; all index-based queries take less than a millisecond, even for the large Wikipedia data. The index can not be applied to all queries: Query 1, 2 and 13 have no predicates, and Query 10 consists of a range predicate. Query 10 and 11 take longest for MonetDB. The comparison between Query 1 & 2 and Query 9 & 10 indicates that the descendant-or-self steps seems to be responsible for the delay, yielding a large intermediate context set before the next step is processed. However, the response time for Query 4 is surprisingly fast.

[1] version from 2006/02/28 (file size: 4.3 GB)

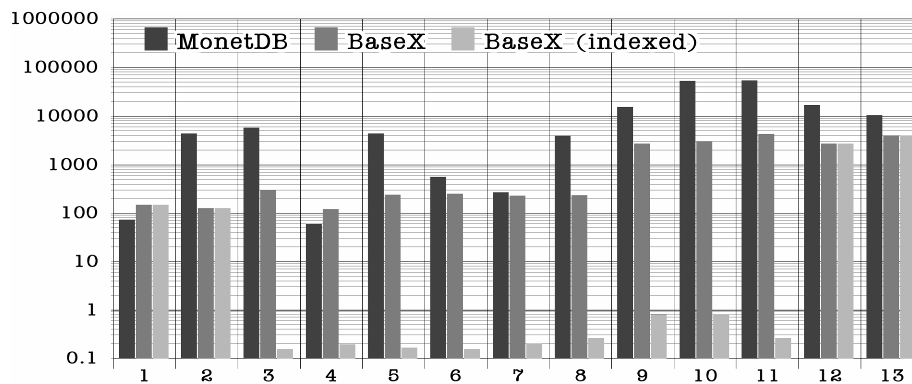


Figure 5: Query execution times (time in ms)

MonetDB/XQuery is a full-fledged query processor, supporting the complete XPath and XQuery functionality, whereas BaseX is still limited to XPath 1.0 queries. Though, the code framework was carefully designed to meet the complete requirements of XPath and XQuery, including the parallel processing of multiple XML instances, DTD and XMLSchema parsing or transient document creation.

6 Conclusion

In the scope of our reaearch, we try to push XML main memory processing and querying to its limits, both in terms of optimized memory usage and efficient querying. We minimize the memory consumption by relationally encoding XML information in a compact node table structure and applying indexes to uniformly store and reference tokens, and we speedup querying by removing unnecessary location steps out of XPath queries. Moreover, we take advantage of the built-in value index for string predicates, accelerating the execution time for many queries by order of magnitudes. We plan to enhance our indexes in our future to efficiently process a wider range of predicate types, including numeric operations.

Literature

[1] MonetDB/XQuery. http://monetdb.cwi.nl

[2] XHive. http://www.x-hive.com

[3] M. Ley. DBLP – Digital Bibliography & Library Project. http://www.informatik.uni-trier.de/~ley/db

[4] SwissProt – Protein Knowledgebase. http://www.expasy.org/sprot

[5] Wikipedia Database Dump. http://download.wikimedia.org

[6] M. Fernández, J. Siméon, et al. Implementing XQuery 1.0: The Galax experience. In VLDB, pp. 1007-1080, 2003

[7] M. Kay. SAXON – The XSLT and XQuery Processor. http://saxon.sourceforge.net

[8] D. Florescu & Kossmann. Storing and Querying XML Data using an RDMBS. In IEEE, Vol. 22, pp. 27-34, 1999

[9] T. Grust, M. v. Keulen & J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In VLDB, pp. 524-535, 2003

[10] University of Washington, XML Repository. http://www.cs.washington.edu/research/xmldatasets

[11] P. Rao & B. Moon. PRIX: Indexing and Querying XML using Prüfer Sequences. In ICDE, pp. 288-300, 2004

[12] A. Barta et al. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In VLDB, pp. 133-144, 2005

[13] P. Boncz, T. Grust, et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. To appear in SIGMOD/PODS, 2006