MonetDB/XQuery—Consistent and Efficient Updates on the Pre/Post Plane

Peter Boncz[1], Jan Flokstra[3], Torsten Grust[2], Maurice van Keulen[3], Stefan Manegold[1], Sjoerd Mullender[1], Jan Rittinger[2], and Jens Teubner[2]

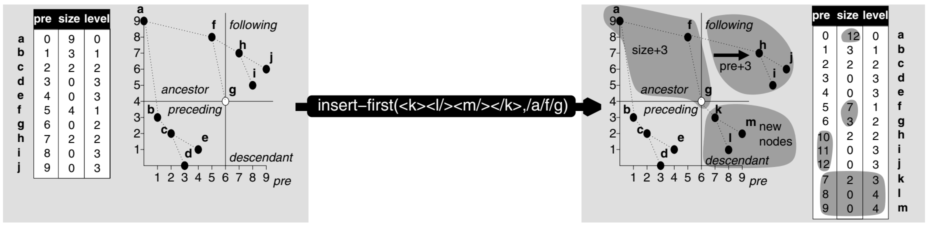
[1] CWI Amsterdam, The Netherlands
{boncz, manegold, sjoerd}@cwi.nl
[2] Technische Universität München, Germany
{grust, rittinge, teubnerj}@in.tum.de
[3] University of Twente, The Netherlands
{keulen, flokstra}@cs.utwente.nl

1 Introduction

Relational XQuery processors aim at leveraging mature relational DBMS query processing technology to provide scalability and efficiency. To achieve this goal, various storage schemes have been proposed to encode the tree structure of XML documents in flat relational tables. Basically, two classes can be identified: *(1)* encodings using *fixed-length surrogates*, like the *preorder ranks* in the *pre/post* encoding [5] or the equivalent *pre/size/level* encoding [8], and *(2)* encodings using *variable-length surrogates*, like, e.g., ORDPATH [9] or P-PBiTree [12]. Recent research [1] showed a clear advantage of the former for efficient evaluation of XPath location steps, exploiting techniques like cheap node order tests, positional lookup, and node skipping in *staircase join* [7]. However, once updates are involved, variable-length surrogates are often considered the better choice, mainly as a straightforward implementation of structural XML updates using fixed-length surrogates faces two performance bottlenecks: *(i)* high physical cost (the *pre*order ranks of all nodes following the update position must be modified—on average 50% of the document), and *(ii)* low transaction concurrency (updating the *size* of all ancestor nodes causes lock contention on the document root).

In [4], we presented techniques that allow an efficient and ACID-compliant implementation of XML updates also on the *pre/post* (respectively *pre/size/level* encoding) without sacrificing its superior XPath (i.e., read-only) performance. This demonstration describes in detail, how we successfully implemented these techniques in *MonetDB/XQuery*[1] [2, 1], an XML database system with full-fledged XQuery support. The system consists of the *Pathfinder* compiler that translates and optimizes XQuery into relational algebra [6], on top of the high-performance MonetDB relational database engine [3].

[1] MonetDB/XQuery and the Pathfinder compiler are available in open-source: `http://monetdb-xquery.org/` & `http://pathfinder-xquery.org/`; the second version including XML updates will be released well before EDBT 2006.


	pre	size	level
a	0	9	0
b	1	3	1
c	2	2	2
d	3	0	3
e	4	0	3
f	5	4	1
g	6	0	2
h	7	2	2
i	8	0	3
j	9	0	3

pre	size	level	
0	12	0	a
1	3	1	b
2	2	2	c
3	0	3	d
4	0	3	e
5	7	1	f
6	3	2	g
10	2	2	h
11	0	3	i
12	0	3	j
7	2	3	k
8	0	4	l
9	0	4	m

Fig. 1. The impact of Structural Updates on *pre*/*size*/*level* XML Storage

2 XML Updates

XML updates can be classified as: *(i) value updates*, which include node value changes (be it text, comment or processing instructions), and any change concerning attributes (attribute value changes, attribute deletion and insertion). Other modifications are *(ii) structural updates*, that insert or delete nodes in an XML document. With the *pre*/*size*/*level* encoding, value updates map quite trivially to updates in the underlying relational tables. Therefore, we focus on *structural updates* in the remainder.

W3C has not formulated a standard for XML updates, yet. However, we expect that a future standard will include the functionality of the UpdateX language as proposed in [11]. Given that there is no standard XML update language (and hence syntax), yet, we decided to keep the changes in our XQuery parser limited by not using the syntax proposed in [11], but rather implement the same update functionality by means of a series of new XQuery operators with side effects.

Consistent Bulk Processing. Semantically, *which* nodes are updated and with *what* values is determined solely using the pre-image (i.e. snapshot semantics). Still, updates need to be applied in the order mandated by XQuery evaluation, which conflicts with the bulk relational query execution employed in MonetDB/XQuery (where optimized query execution may use a different order). To overcome this problem, the update operators initially just produce a *tape* of *intended* updates. This tape is represented by an XQuery item sequence, and thus in the end is yielded in the correct order. Finally, after optimization (in which duplicate updates or updates on deleted nodes are pruned), these updates are applied and committed. In our opinion, this optimized bulk approach to updates is unique to MonetDB/XQuery. Note that the update tape, which separates query evaluation and update execution, bears some resemblance to the idea of monad-based I/O [10] in purely functional programming languages, *e.g.*, Haskell.

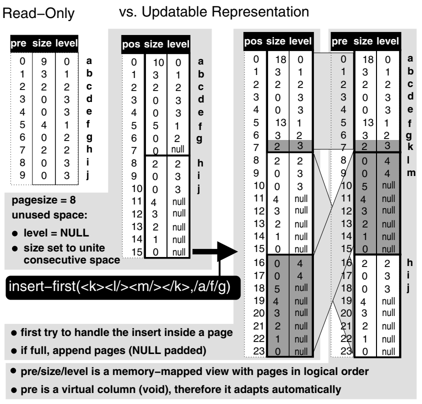
Structural Update Problems. Figure 1 illustrates how the *pre*/*size*/*level* document encoding is affected by a subtree insert (a delete raises similar issues): all *pre* values of the nodes `following` the insert point change, as well as the *size* of all `ancestor` nodes. The former issue imposes an update cost of $O(N)$, with

N the document size, because on average half of the document are `following` nodes. The latter issue is not so much a problem in terms of update volume (the number of ancestors is bound by the tree's height, remaining small even for large XML instances) but rather one of locking: the document root is an `ancestor` of all nodes and thus must be locked by every update. This problem, however, can be circumvented by maintaining for each transaction a list of nodes of which the *size* is changed, together with the *delta* rather than the absolute changed value. This allows transactions to release locks on *size* immediately, and commit anyway later (even if the *size* of a node has been changed meanwhile by another committed transaction, we can just apply the delta to set it to a consistent state).

With the problem of locking contention on *size* removed this way, in the sequel we concentrate on the problem of the shifts in *pre* here.

Page-Wise Remappable Pre-Numbers. Figure 2 shows the changes introduced in MonetDB/XQuery to handle structural updates in the *pre*/*size*/*level* table. The key observations are:

- the table is called *pos*/*size*/*level* now.
- it is divided into *logical pages*.
- each logical page may contain *unused tuples*.
- new logical pages are appended only (i.e., at the end).
- the *pre*/*size*/*level* table is a view on *pos*/*size*/*level* with all pages in logical order. In MonetDB, this is implemented by mapping the underlying table into a new virtual memory region.


Read–Only
vs. Updatable Representation
pagesize = 8
unused space:
● level = NULL
● size set to unite consecutive space
insert–first(<k><l/><m/></k>,/a/f/g)
● first try to handle the insert inside a page
● if full, append pages (NULL padded)
● pre/size/level is a memory–mapped view with pages in logical order
● pre is a virtual column (void), therefore it adapts automatically

pre	size	level	
0	9	0	a
1	3	1	b
2	2	2	c
3	0	3	d
4	0	3	e
5	4	1	f
6	0	2	g
7	2	2	h
8	0	3	i
9	0	3	j

pos	size	level	
0	10	0	a
1	3	1	b
2	2	2	c
3	0	3	d
4	0	3	e
5	5	1	f
6	0	2	g
7	0	null	
8	2	2	h
9	0	3	i
10	0	3	j
11	4	null	
12	3	null	
13	2	null	
14	1	null	
15	0	null	

pos	size	level
0	18	0
1	3	1
2	2	2
3	0	3
4	0	3
5	13	1
6	3	2
7	2	3
8	2	2
9	0	3
10	0	3
11	4	null
12	3	null
13	2	null
14	1	null
15	0	null
16	0	4
17	0	4
18	5	null
19	4	null
20	3	null
21	2	null
22	1	null
23	0	null

pre	size	level	
0	18	0	a
1	3	1	b
2	2	2	c
3	0	3	d
4	0	3	e
5	13	1	f
6	3	2	g
7	2	3	k
8	0	4	l
9	0	4	m
10	5	null	
11	4	null	
12	3	null	
13	2	null	
14	1	null	
15	0	null	
16	2	2	h
17	0	3	i
18	0	3	j
19	4	null	
20	3	null	
21	2	null	
22	1	null	
23	0	null	

Fig. 2. Updates With Logical Pages

Figure 2 shows the example document being stored in two logical pages. The logical size is measured in a number of tuples (here: 8) instead of bytes. The document shredder already leaves a certain (configurable) percentage of tuples unused in each logical page. Initially, the unused tuples are located at the end of each page. Their *level* column is set to NULL, while the *size* column holds the number of directly following consecutive unused tuples. This allows the staircase-join to skip over unused tuples quickly. For the same reason, the *size* of existing nodes now also embraces the unused tuples within the respective subtrees.

The advantage of unused tuples is that structural deletes just leave the tuples of the deleted nodes in place (they become unused tuples) without causing any shifts in *pre* numbers. And since unused tuples are counted in the *size* of their ancestors, deletes do not require updates of the *size* of their ancestors. Also, inserts of subtrees whose sizes do not exceed the number of unused tuples on the logical page, do not cause shifts on other logical pages. Larger inserts, only use page-wise table appends. This is the main reason to replace *pre* by *pos*. The *pos* column is a densely increasing (0,1,2,...) integer column, which in MonetDB can be efficiently stored in a *virtual* (non-materialized) `void` column.

We introduced new functionality in MonetDB to map the underlying disk pages of a table in a different non-sequential order into virtual memory. Thus, by mapping in the virtual memory pages of the *pos/size/level* table in logical page order, overflow pages that were appended to it, become visible "halfway" in the *pre/size/level* view.

In the example of Figure 2, three new nodes `k`, `l` and `m` are inserted as children of context node `g`. This insert of three nodes does not fit the free space (the first page that holds `g` only has one unused tuple at *pos=7*). Therefore, a new logical page must be inserted in-between. Thus, we insert eight new tuples, of which only the first two represent real nodes (`l` and `m`), the latter six are unused. Thanks to the *virtual column* feature of MonetDB, in the resulting *pre/size/level* view, all *pre* numbers after the insert point automatically shift, at no update cost at all!

3 Conclusion

In our demonstration, we will show the performance and scalability of both read-only and update queries on potentially huge XML databases, provided by MonetDB/XQuery. The demonstration will graphically show how the key techniques described here influence the behavior of the system.

References

1. P. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time. Technical Report INS-E0503, CWI, 2005.
2. P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery—The Relational Way. In *Proc. VLDB Conf.*, 2005. (Demo).
3. P. Boncz and M.L. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2), 1999.
4. P. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proc. XIME-P*, 2005.
5. T. Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD Conf.*, 2002.
6. T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB Conf.*, 2004.
7. T. Grust, M. v. Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. VLDB Conf.*, 2003.
8. T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in Any RDBMS. *ACM Trans. on Database Systems*, 29(1), 2004.
9. P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATH: Insert-Friendly XML Node Labels. In *Proc. SIGMOD Conf.*, 2004.
10. S. Peyton-Jones and P. Wadler. Imperative Functional Programming. In *Proc. POPL Conf.*, 1993.
11. G. M. Sur, J. Hammer, and J. Simeon. UpdateX - An XQuery-Based Language for Processing Updates in XML. In *Proc. PLAN-X*, 2004.
12. J. Xu Yu, D. Luo, X. Meng, and H. Lu. Dynamically Updating XML Data: Numbering Scheme Revisited. *World Wide Web Consortium*, 8(1), 2005.