Updating the Pre/Post Plane in MonetDB/XQuery

Peter Boncz Stefan Manegold
CWI
Kruislaan 413
Amsterdam, The Netherlands
{boncz,manegold}@cwi.nl

Jan Rittinger*
University of Konstanz
P.O. Box D 188
Konstanz, Germany
rittinge@inf.uni-konstanz.de

ABSTRACT

We outline an efficient ACID-compliant mechanism for structural inserts and deletes in relational XML document storage that uses a region based *pre/size/level* encoding (equivalent to the *pre/post* encoding). Updates to such node-numbering schemes are considered prohibitive (i.e. physical cost linear to document size), because structural updates cause shifts in all *pre*-numbers after the update point, and require updates of the *size* of all ancestors, such that the root of the tree becomes a locking bottleneck. We show how such locking can be avoided by updating the *size* of ancestors using delta-increments, which are transaction-commutative operations. We also reduce the physical cost to the minimum (i.e. linear to update volume) by carefully exploiting the *virtual column* feature of MonetDB to store *pre* numbers (virtual columns are never materialized, and thus need not be updated). In our evaluation, we show the overhead of the update-feature in MonetDB/XQuery in terms of added XMark evaluation cost to stay within an acceptable limit (<30% on average).

1. INTRODUCTION

The MonetDB/XQuery system[1], allows to store schema-free XML documents in MonetDB and query them with XQuery. Figure 1 shows the system to consist of the *Pathfinder* XQuery-to-Relational Algebra compiler [5], and a small set of relational algebra extensions. Here, Pathfinder generates physical plans in MIL, the physical algebra on the binary relational model implemented by MonetDB [1]. Pathfinder currently supports almost the full XQuery standard, closely following the W3C Formal Semantics. MonetDB/XQuery is the first relational XQuery system we are aware of that fully supports both document and sequence order, XML schemas, and even recursive user-defined functions. MonetDB/XQuery also provides unsurpassed performance and scalability: the combination of efficient nested XPath axis evaluation with *loop-lifted staircase join*, algebraic order optimizations, and join expression translation and optimization into relational join plans, allows it to e.g. obtain interactive response times on all XMark queries even for the 1 GB document size on a standard PC [2].

When comparing XML database solutions with file-based management of XML documents, however, fast query performance is only one side of the story. The main other reason why XML databases are to be preferred over file-based XML document management, is update support. File-based XML storage causes a full re-write of the entire document on each change, and does not provide any concurrency. Therefore, *efficient* and *ACID-compliant* XML document update functionality is an important selling point of XML *database* technology.

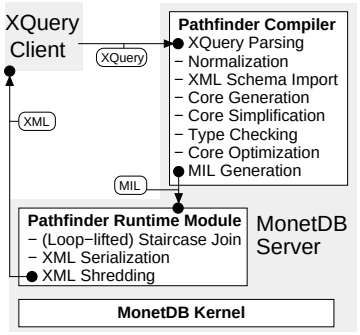


Figure 1: MonetDB/XQuery Architecture

Sticking to our approach of using relational technology to provide XML database functionality, the challenge of this paper is to map XML update requests, as formulated in a language like XUpdate [10], into update actions on the underlying relational XML document encoding. The particular encoding used in MonetDB/XQuery, maps all document nodes onto *pre/post* tuples (explained in Section 2.2). This XML document representation is exploited to full advantage when evaluating XPath steps efficiently, enabling cheap node order tests as well as significant positional node skipping in *staircase join* [6]. However, the naive implementation of this encoding poses two update performance challenges: *(i)* high physical update cost, and *(ii)* low granularity of transaction locking. Because of this, the *pre/post* plane is currently perceived to be a read-only representation of XML documents.

The main contribution of this paper is to show that both challenges can actually be met, converting the *pre/post* plane into a suitable representation to implement dynamic updates that are both efficient and allow concurrency. In the evaluation, we show that the overhead of our update scheme on large instances of the XMark benchmark (up to 1.1 GB) is limited to about 30%, such that even under updates, MonetDB/XQuery is still highly efficient and scalable.

*Work was supported by the DFG Research Training Group GK-1042 *Explorative Analysis and Visualization of large Information Spaces.*

[1]MonetDB/XQuery and the Pathfinder compiler are available in open-source: `http://monetdb.cwi.nl` & `http://pathfinder-xquery.org`.

Informal Proceedings of the *Second International Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, June 16-17, 2005, Baltimore, MD, USA..

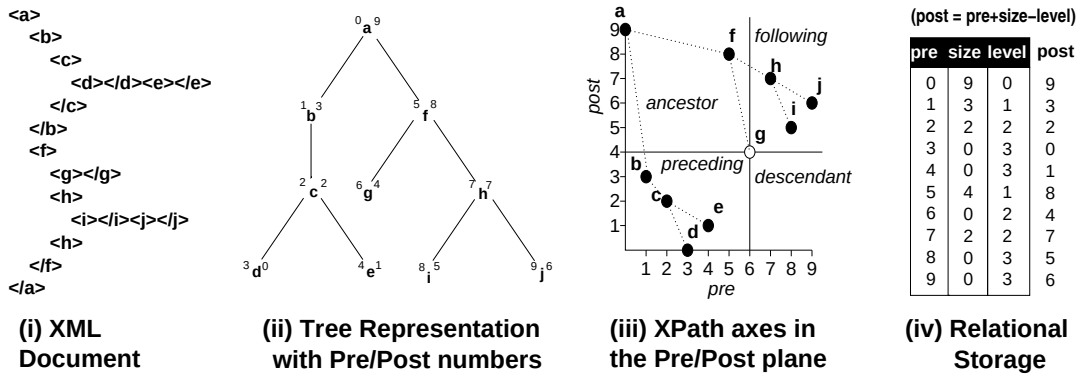


Figure 2: Relational Storage With *pre/size/level* To Support Efficient XPath Axis Traversal

1.1 Outline

In Section 2, we make our problem statement, by defining the update queries and identifying structural updates in the *pre/post* plane as the main challenge, both in terms of absolute update performance as well as concurrency. Then in Section 3, we describe how a *logical page* administration can help to keep update cost down. This scheme can be realized efficiently in MonetDB thanks to its concept of virtual columns, and its use of virtual memory to hide the physical page order (that after updates may differ from document node order) from query execution primitives such as staircase join. We also show how locking can be minimized by using transaction-commutative delta operations. We evaluate the overhead imposed on XQuery by our update scheme in Section 4.1. Finally, in Section 4.2 we provide an overview of related work before concluding in Section 5.

2. PROBLEM STATEMENT

We first define the XML updates that we need to support, then shortly introduce and summarize the *pre/size/level* variant of *pre/post* encoding for XML documents and its use for XPath evaluation, before discussing the impact of XML updates on this encoding.

2.1 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. In MonetDB/XQuery, value updates map quite trivially to updates in the underlying relational tables. Therefore, we focus this paper on *structural updates*.

Until the W3C formulates a standard for XML updates, the most often used update language is XUpdate [10]. We shortly summarize the syntax and semantics of its structural update commands:

`<xupdate:remove select="`*expr*`"/>` removes all nodes (and entire subtrees) to which the XPath *expr* evaluates.

`<xupdate:insert-before select="`*expr*`">`*element*`</xupdate:insert-before>` inserts an *element* node as a directly preceding sibling to all nodes in the result set of the XPath *expr*.

`<xupdate:insert-after select="`*expr*`">`*element*`</xupdate:insert-after>` behaves identically, except that the new node becomes the direct successor of the selected context nodes.

`<xupdate:element name="`*name*`">`*XML*`</xupdate:element>` specifies the top-level *element* to be inserted. It may contain nested *XML*, such that entire subtrees can be inserted.

`<xupdate:append select="`*expr*`" child="`*integer*`">`*element*`</xupdate:append>` appends the new node as a child of the context-nodes. The optional *integer* child expression indicates the position of the new child node (by default, it is appended as last child).

2.2 The pre/post Plane

Figure 2, consisting of four parts, depicts how MonetDB/XQuery encodes schema-free XML documents in relational tables. Part *(i)* shows the example document. In part *(ii)*, nodes of the XML tree are assigned *pre* and *post* ranks, which count how many tags have been opened and closed, respectively, as seen when parsing the document sequentially. In part *(iii)*, which plots all document nodes in a *pre/post* plane, we clearly recognize the tilted XML tree. It also shows that for each node (in this case the context node is `g`), the quadrants of the *pre/post* plane correspond to the major XPath axes: `descendant`, `following`, `ancestor` and `preceding`. As such, this representation allows to express all XPath axes as simple comparisons on the *pre* and *post* columns, which can be evaluated efficiently in an SQL-speaking RDBMS [3]. Finally, part *(iv)* of Figure 2 shows the actual relational XML representation used in MonetDB/XQuery, which instead of the *post* column stores two columns holding a tree *level* and subtree *size*. This *pre/size/level* encoding is equivalent to *pre/post* since *post = pre + size - level*.

While XPath axes can be executed perfectly well using SQL queries on the *pre*, *level* and *size* columns, extra efficiency can be won because these *pre, level* and *size* columns do not hold random integer values, but a tree representation. The *staircase join* [6] exploits this property, which allows it to conclude that certain regions of *pre* values *cannot* contain any result nodes for a XPath step. In such situations, the staircase join avoids any data access or computation and *skips* over these tuples. This makes its performance superior to relational nested index-lookup join.

The actual reason why MonetDB/XQuery uses *size/level* instead of *post*, is related to this node skipping. By "separating" the information, more skipping can be performed. For example, finding all children of a node pre_x works by checking the first child $pre_y = pre_x + 1$ and skipping to its siblings $pre_y = pre_y + size[pre_y] + 1$ until the last node in pre_x is reached ($pre_x + size[pre_x]$).

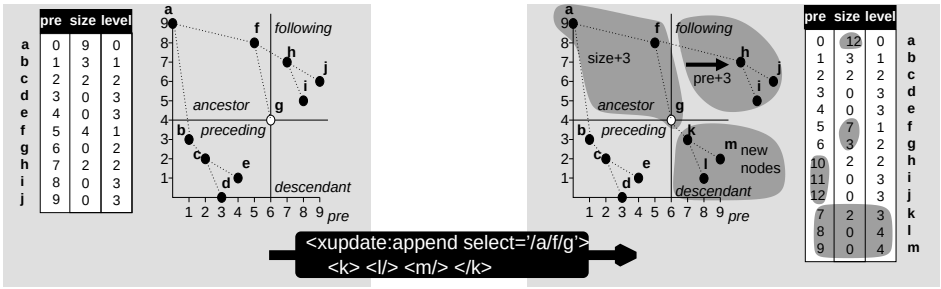


Figure 3: The impact of Structural Updates on *pre/size/level* XML Storage

Node skipping is particularly well-supported in MonetDB, as it stores the *pre* column using the `void` type. A `void` column holds a densely ascending node sequence (0,1,2,...). Such `void` columns are actually *not* stored at all: they take zero space. More importantly, careful design and implementation of relational tables in MonetDB allows to support lookup of `void` values using *positional* algorithms, such as *positional* select and *positional* join, that use highly efficient array-lookups. Thus, skipping to a particular node in staircase join comes down to array-lookup with the proper index, at the cost of a single CPU instruction.

When *pre* numbers are stored in another RDBMS than MonetDB, the *pre* values are materialized in the tuples, and lookup is usually accelerated using a B-tree index, which is still fast ($O(log(N))$), but not constant in cost such as in MonetDB. We think that the representation of node numbers as simple *pre* integers that can be located positionally is the prime reason for the performance advantage of MonetDB/XQuery over other XQuery systems. It may seem that this choice is a trade-off too much skewed towards read-only efficiency only, since maintaining the densely populated physical representation of *pre* in MonetDB, poses a serious challenge to update efficiency, as described in the following.

2.2.1 Structural Update Problems

Figure 3 illustrates how the *pre/size/level* document encoding is affected by a subtree insert (cf. delete): 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 (because there are only $O(log(N))$ ancestors) but rather one of locking: the document root is an `ancestor` of all nodes and thus must be locked by every update. We treat transaction processing in Section 3.2, and concentrate on the problem of the shifts in *pre* here.

An advantage of using *size/level* instead of *post*, is that neither *level* nor *size* values are affected by structural updates, while *post* is. Still, updating the *pre* column, is impossible if it is designated to be a primary key. Even if not, the impact of maintaining (B-tree) indices when each XML update leads to half of the tuples changing value, will be prohibitive. In the case of MonetDB, where *pre* is stored as a `void` column, the result depicted in Figure 3 is even technically impossible, because `void` columns may never be modified! Thus, we conclude that the storage scheme used until now in MonetDB/XQuery and which produced the XMark results in [2] is a read-only solution.

3. UPDATEABLE XML ENCODING

Figure 4 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.

Figure 4 shows the example document being stored in two logical pages. The logical size is measured in an amount 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 amount of directly following consecutive unused tuples. This allows the staircase-join to skip over unused tuples quickly.

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. Also, inserts of subtrees whose size does not exceed the amount 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.

A logical page does *not* necessarily correspond to a single physical disk page. MonetDB uses virtual memory to load persistent data into memory, so we refer to virtual memory pages as physical pages here. We set the logical page size to the virtual memory-mapping granularity (usually 65536), such that logical page boundaries always coincide with virtual memory page boundaries. We also 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 4, three new nodes `k`, `l` and `m` are inserted as children of context node `g`. This insert of

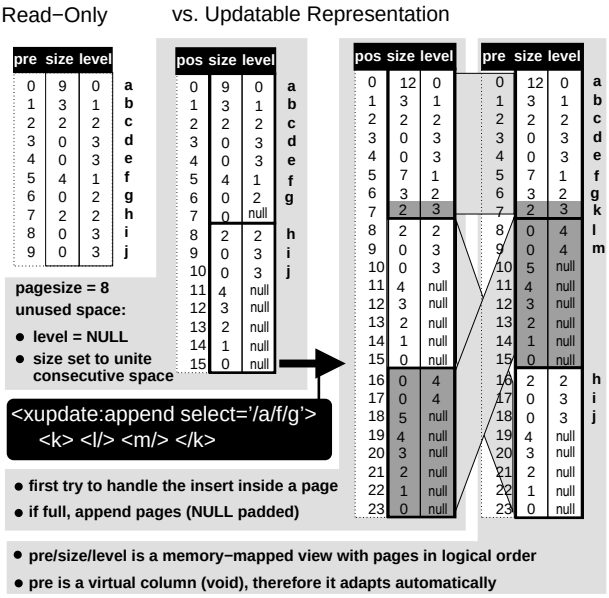


Figure 4: Updates With Logical Pages

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!

3.1 Storage Schema

Figure 5 shows the original relational schema of MonetDB/XQuery. The main tables are:

- *pre/size/level*, with one tuple for each document node.
- *attr*, with one tuple for each attribute.
- *prop*, holding all unique attribute values (as strings).
- *qn*, with one tuple for each qualified name (element or attribute).

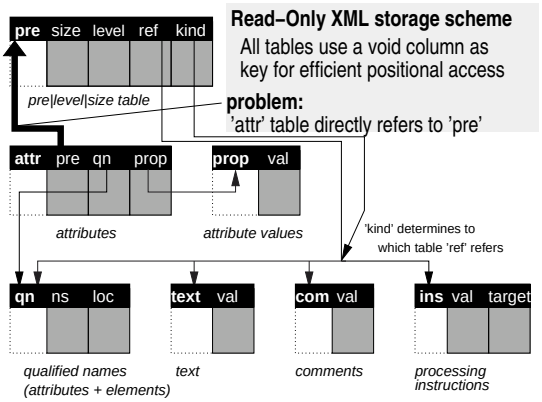


Figure 5: Original Read-Only Schema

- *text,com* and *ins*, that hold node values.

Each table uses a `void` column as a key, indicated by a white column with dotted lines. This allows MonetDB to use fast positional join when queries navigate through the schema over foreign keys.

In the new, updateable schema, all *pre* columns are replaced by *pos*. Even though the logical page mechanism limits node shifts to within a page, typically thousands of tuples are affected. This remains a problem for the attribute table, which refers back to *pre* (now *pos*) values. Thus, each structural update would incur significant maintenance in the attribute table. For this reason, we decided to give each node a unique *node* number that never changes through its lifetime. It is added as a column to the *pos/size/level* table. At shredding time, *node* numbers are identical to *pos* numbers (the latter ones may change later under updates).

The final shape of the XML storage scheme is summarized in Figure 6. A new *pageOffset* table is maintained to maintain a logical page order under updates. In MonetDB, this *pageOffset* table is used by the new adaptive memory mapping primitive to construct the *pre/size/level* view.

To translate unique *node* numbers into *pre* values, first a (positional) lookup into a new *node/pos* table is done. The resulting *pos* numbers can be "swizzled" into a *pre* values, by a lookup into the *pageOffset* table and some simple bitwise arithmetic: *pageOffset[pos >> 16] << 16 + pos* & 65535

Figure 7 depicts the update work needed for two scenarios: *(a)* when the insert can be handled within a logical page and *(b)* when a new logical page is needed. The tuples after the insert point are moved forward in the page first. Then, the new *pos* of the moved tuples has to be modified in the *node/pos* table. Finally, the newly inserted nodes are written into the *pos/size/level* table. They need a unique *node* number, which can generally be found inside the logical page in the *node/pos* table by scanning for NULL *pos* values (if not, tuples are appended to the *node/pos* table). If a new logical page had to be inserted (case *(b)*), then a new entry for it is appended to the *pageOffset* table, and the offset of *all* pages after the insert point is incremented.

While we omit the formal descriptions here, the our update mechanism can be captured in a straightforward fashion in rules that translate XUpdate statements in bulk relational (SQL) update queries on the *pos/size/level*, *pageOffset*, and *pos/node* tables; thus extending the relational XQuery mapping framework presented in [5].

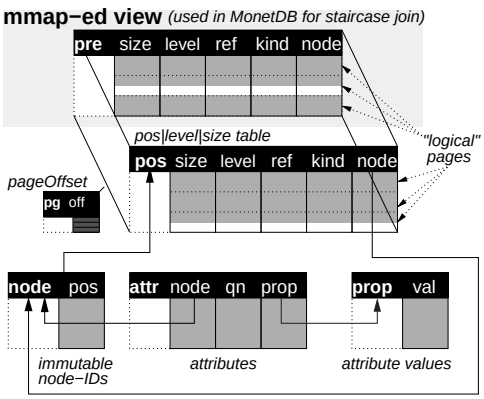


Figure 6: New Updateable Schema

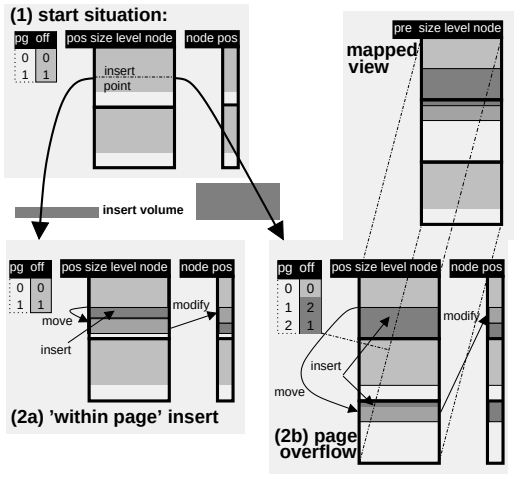


Figure 7: Structural Insert with Bulk Updates

```
write-transaction()
  incrementally:
  - read-lock pages during XPath execution
  - write-lock all pages that need to be updated

  for each table update:
  - create a copy-on-write memory-mapped view on the base table
  - perform updates in this view (so others don't see the changes yet)
  - keep a differential-list with all changes made

  except bulk-updates in pre/size/level and node/pos (see Figure 7):
  - only write into newly appended pages (thus not seen by others)
  - when done, flush those pages to disk
  - reference these pages in a private pageOffset table

  prepare a list of affected ancestors, and delta-sizes for them

  run XML document validation (if there is a schema)

  get global write-lock
    compute the size of all affected ancestors
    write into the WAL:
    - new size of all affected ancestors
    - shifts introduced in pageOffset table
    - the differential lists made for the copy-on-write table views

    /* commit succeeded */
    write size of all affected ancestors into /pre/size/level table
    make a new pageOffset table
    carry through the differential-lists into the base tables
  release all page-locks
  release global write-lock
```

Figure 8: Transaction Management Pseudo-Code

3.2 ACID properties

Read-only XQuery queries just acquire a global read-lock while they run. XML update requests use the algorithm described in Figure 8, which implements strict two-phase locking with write-ahead logging (WAL) using shadow paging. We summarize how the ACID properties are maintained:

isolation MonetDB uses shadow paging mechanism for isolation, by creating a temporary *copy-on-write* memory-mapped view on the base table [1]. In such a memory-map, all pages are initially shared with the base table, but the OS transparently replaces all pages that are written into by pages from the swap-file, such that the base table is never altered. MonetDB keeps delta-tables (differential lists) for all updates, to propagating all changes to the base tables when the transaction commits. Special treatment is given to the bulk-updated areas in the *pos/size/level* and *node/pos* tables (Figure 7). These areas are written only in newly appended logical pages, that are referenced (and thus seen) only by the transaction in its private copy of the *pageOffset* table.

atomicity Before getting the global write-lock, transactions have only written in isolation. All of these changes are carried through either by propagating the differential lists to the base tables, or in the case of the *pos/level/size* table, by creating a new *pageOffset* table that includes the modified logical pages. Writing the WAL is the crucial stage in transaction commit, it consists of a single I/O.

consistency As a last stage before trying to commit the transaction, the new XML document should be validated, using the mechanism described in [4]. If this (or any other action) fails, the transaction is aborted.

durability In case of a crash during commit, we may lose the new version of the *pageOffset* table, the new *size* values of all ancestors, and parts of the changes to the other tables carried through with differential-lists. All this information is present in the WAL, such that during recovery an up-to-date version of the database can be restored.

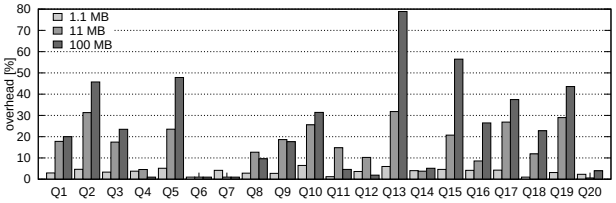
Notice that while a transaction runs, the *size* of some affected ancestors as well as the *pageOffset* tables may be changed by concurrent transactions (that insert/delete nodes located in different logical pages) that commit earlier. Our transaction protocol has been made immune to such changes by working with *delta* rather than new absolute values. For example, a transaction computes how much a *size* value must be incremented, rather than the absolute new value. As delta operations are commutative, it does not matter in which order they are executed. This allows MonetDB/XQuery to avoid locking all ancestors during the entire transaction.

4. EVALUATION

We have described a mechanism that allows to update XML documents encoded in the *pre/size/level* plane. The main idea is to use a logical paging scheme, to limit shifts in *pre*-numbers to stay within a logical page. The logical pages can be laid out in a logical order, which may be different from the physical tuple order in the *pos/size/level* table. In the case of MonetDB, we can use memory-mapping techniques to re-create the *pre/size/level* table as a view on the *pos/size/level* table, such that the staircase join need not even be aware of this. The only modification to staircase join was skipping over unused tuple areas (this can be done fast, by looking at the *size* of the first unused tuple seen).

We think that use of our mechanism is not restricted to MonetDB only. Using a *pos/size/level* table, where *pos* is e.g. a SQL 2003 *generated column*, will work fine in any RDBMS, and the computation of *pre* from *pos* using a *pageOffset* table is perfectly expressible in SQL. Just like original staircase join, a RDBMS will not use positional lookup[2], but can still be accelerated with B-tree indices. Instead of using the memory-mapped view on *pre*, as MonetDB does, the implementation of staircase join in an extensible RDBMS (as demonstrated in Postgres [7]) could be made aware of the difference between *pos* and *pre* and perform the required calculations using the *pageOffset* table explicitly.

[2]We think that RDBMSs could introduce positional-select and -join as physical algorithms for accessing SQL 2003 generated columns.



	1.1 MB		11 MB		110 MB		1.1 GB	
Q	ro	up	ro	up	ro	up	ro	up
1	0.034	0.035	0.045	0.053	0.170	0.204	1.334	1.939
2	0.043	0.045	0.067	0.088	0.317	0.462	2.483	4.136
3	0.120	0.124	0.241	0.283	1.458	1.800	12.656	16.427
4	0.053	0.055	0.066	0.069	0.459	0.459	3.927	4.190
5	0.039	0.041	0.051	0.063	0.163	0.241	1.211	2.254
6	0.020	0.020	0.023	0.023	0.060	0.060	0.368	0.408
7	0.024	0.025	0.029	0.029	0.083	0.083	0.544	0.607
8	0.071	0.073	0.118	0.133	0.730	0.800	10.198	11.268
9	0.109	0.112	0.161	0.191	0.873	1.027	12.439	14.575
10	0.279	0.297	0.657	0.825	5.088	6.686	51.843	67.198
11	0.083	0.084	0.162	0.186	3.426	3.584		
12	0.083	0.086	0.127	0.140	1.717	1.750		
13	0.050	0.053	0.066	0.087	0.208	0.372	1.436	3.341
14	0.050	0.052	0.213	0.221	1.789	1.881	17.918	18.371
15	0.065	0.068	0.082	0.099	0.255	0.399	1.855	3.736
16	0.072	0.075	0.093	0.101	0.253	0.320	2.043	2.879
17	0.047	0.049	0.067	0.085	0.307	0.422	2.652	4.137
18	0.032	0.032	0.042	0.047	0.136	0.167	1.091	1.577
19	0.064	0.066	0.107	0.138	0.583	0.837	5.152	7.940
20	0.130	0.133	0.173	0.174	0.578	0.601	4.988	5.507

Figure 9: read-only 'ro' vs. updateable 'up' schema (XMark performance in seconds)

4.1 Experiments

We implemented the new updateable *pos/size/level* schema with the additional *node/pos* table in an experimental version of MonetDB/XQuery. To quantify the overhead over the original read-only *pre/size/level* schema, we ran the XMark benchmark. For the updateable schema, we created a scenario where about 20% of the logical pages were kept unused. This scenario mimics the state of the database after a series of XUpdate operations (e.g., inserts and deletes). We ensured that the document sizes were equal with both schemas. Hence, the *pos/size/level* table of the updateable schema occupies about 25% more space than the *pre/size/level* table of the read-only mapping. Moreover, the updateable schema contains the extra *node* column in the *pos/size/level* table and has the additional *node/pos* table that is positionally joined each time an attribute is looked up after an XPath step (that yields *pre* numbers).

The queries were run on an Opteron 1.6 GHz Linux machine with 8 GB RAM. [3] Figure 9 shows the overhead, i.e., increase in evaluation time, of the updateable schema over the read-only schema. With small documents (1.1 MB and 11 MB) the overhead never exceeds 7% and 33%, respectively, with an average of about 15% for the 11 MB document. In the 110 MB XMark document size, 8 out of the 20 queries exhibit a running time increase of less than 10%, another 4 queries stay below 25%, 6 queries incur an overhead between 25% and 50%, and only 2 queries require 56% respectively 78% more evaluation time. For the 1.1 GB document size, the impact increases, but stays on average below 30%. This we deem an acceptable result, especially given the increases ($> 25\%$) in the storage size of the schema as described above.

[3] On that machine, MonetDB/XQuery also runs the 11 GB XMark efficiently; it needs RAM in order of the document size.

4.2 Related work

We introduced a mechanism, which allows updates on fixed size keys. The work in [9] describes a different approach to insert new nodes, which (like we) uses fixed size keys, relying on a marking scheme or a perfect hash. However, even the fastest variant proposed completely enumerates all keys, which is something we manage to avoid.

Alternatively, indexing schemes built on *variable length* keys realize inserts by extending the key of one of the adjacent siblings. The prominent approach is ORDPATH [8], which uses a bit-compressed Dewey Order. Another variant, P-PBiTree, claiming to minimize key length and renumbering costs is presented in [11]. While an insert operation in such indexing schemes may be cheaper than our page-based mechanism, we lack an XML update benchmark to properly assess this. On the other hand, the variable-length keys have a comparison cost that is higher than simple integer comparison, and skipping cannot use array-like positional access such as in MonetDB, but should use (slower) B-tree lookup. Also, the size of variable-length keys may degenerate on repeated inserts into the same subtree.

5. CONCLUSION

We have presented a mechanism that allows efficient updates in the *pre/size/level* variant of the *pre/post* plane. This is achieved by limiting shifts in *pre*-numbers to within so-called logical pages. Within a logical page, the nodes are stored in document order, but the pages no longer need to be consecutively ordered physically, thus allowing to insert new pages in-between at low cost. Locking contention due to shifts in properties of the root node, is avoided by carrying out such changes using commutative delta-operations.

This mechanism will be used in a future version of the MonetDB/XQuery system, which uses the Pathfinder XQuery-to-Algebra compiler. In MonetDB, the update mechanism uses memory-mapping techniques to conserve the efficient positional-skipping used in staircase join, causing the overhead of allowing updates to stay within acceptable limits.

6. REFERENCES

[1] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam, May 2002.

[2] P. A. 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, Amsterdam, NL, March 2005.

[3] T. Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD Conf.*, June 2002.

[4] T. Grust and S. Klinger. Schema Validation and Type Annotation for Encoded Trees. In *<XIME-P/>*, June 2004.

[5] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB Conf.*, August 2004.

[6] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. In *Proc. VLDB Conf.*, September 2003.

[7] S. Mayer, T. Grust, M. van Keulen, and J. Teubner. An Injection of Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proc. VLDB Conf.*, August 2004.

[8] 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.*, Paris, France, June 2004.

[9] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. SIGMOD Conf.*, 2001.

[10] XUpdate. `http://www.xmldb.org/xupdate/`.

[11] J. Xu Yu, D. Luo, X. Meng, and H. Lu. Dynamically Updating XML Data: Numbering Scheme Revisited. *World Wide Web Consortium*, 8(1), March 2005.