View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Combined Static and Dynamic Analysis for Effective Buffer Minimization in Streaming XQuery Evaluation

Michael Schmidt Stefanie Scherzinger Christoph Koch

Saarland University Database Group, Germany
{schmidt,scherzinger,koch}@infosys.uni-sb.de

Abstract

Effective buffer management is crucial for efficient in-memory and streaming XQuery processing. We propose a buffer management scheme which combines static and dynamic analysis to keep main memory consumption low. Our approach relies on a technique that we call active garbage collection and which actively purges buffers at runtime based on the current status of query evaluation. We have built a prototype system for a practical fragment of XQuery which employs our buffer management scheme. The experimental results demonstrate the significant impact of combined static and dynamic analysis on reducing main memory consumption and running time.

1 Introduction

Over the past years, XQuery has evolved into a powerful and widely accepted query language for XML processing. Various in-memory XQuery engines have been developed [3, 6, 11, 13, 19] and it has been repeatedly observed that main memory consumption remains a crucial bottleneck in XQuery evaluation. In particular when XQuery is evaluated on streams, the input cannot be completely buffered prior to query evaluation. Here, good buffer management becomes *the* key prerequisite to performance.

Ideally, the buffer manager of a streaming XQuery engine will (1) only put data that is relevant for query evaluation into the buffer, (2) not keep data buffered longer than necessary, and (3) not keep multiple copies of the data in buffers. These goals are conflicting, for instance, a system optimal for (1) would have to be able to check satisfiability of XQuery expressions, an undecidable problem (this is implicit e.g. in [2]).

We claim that in order to come closer to meeting these three desiderata, a combination of static analysis and dynamic buffer minimization techniques is needed. In virtually all current systems, the decisions regarding what to buffer and when to delete from buffers are made at compile-time only, based on purely *static* query analysis [3,6,11,12,16]. Let us review the buffer management strategies of some existing XQuery engines.

Among the early work on XQuery buffer management is the static *projection* technique implemented in Galax [13], and refined in [3, 4], where only the parts of the input relevant to query evaluation are loaded into memory. Yet as the projected document is computed before query evaluation can start, buffer management during query evaluation is not an issue.

While Galax is an in-memory XQuery engine, other systems have been specifically designed to operate on XML streams [11, 12]. These works evaluate parts of the query on-the-fly with no or only little buffering, using static analysis of data dependencies and schema information [11], if available. However, for many practical queries involving blocking operators or descendant axes and wildcards, little can be evaluated on-the-fly [1, 5, 11, 12].

In the above systems, the decision when buffers are purged is made at compile-time. In the case of the FluX-Query engine [11] and similarly in [12], the lifetime of a buffer is associated with the scope of an XQuery variable. While buffers can be conveniently deleted once the scope of the associated variable ends, it becomes difficult to avoid that data is buffered twice. Such situations can arise if an XML node is bound by different variables, e.g. as is required for checking a condition and for producing output. In particular for queries with descendant axes and wildcards, it may become difficult to avoid duplicate buffering.

We argue that in order to come closer to satisfying desiderata (1) through (3), *both static and dynamic* analysis are required: Based on *static query analysis*, we can incrementally compute a *projection* of the input document, thus buffering only data that is relevant to query evaluation [13]. In addition, we can statically infer the moments during query evaluation when buffered nodes have *become* "irrelevant" for the remaining query evaluation, namely each time that a query subexpression has been evaluated. Yet to delete nodes from the buffer early on during query evaluation, *dynamic analysis* is required which takes into account the current buffer contents, the state of query evaluation, and the progress made in reading the input. Obviously, we may expect the impact of *combined static and dynamic analysis* on main memory consumption to be greater than what can be

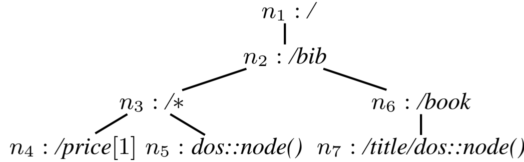


Figure 1. Projection tree.

achieved by static analysis alone.

Garbage Collection in XQuery Engines. In this paper, we propose *active garbage collection*, a novel buffer management technique for XQuery engines in which both static and dynamic analysis are exploited.

Garbage collection [20] is a well-understood technique for automatic memory management in programming languages. The basic principle of any garbage collector is to determine which data objects in a program will not be accessed in the future, and consequently, to reclaim the storage used by these objects. A simple yet effective garbage collection strategy is *reference counting* where every object counts the number of references to it. When a reference is created to an object, its reference count is incremented. Likewise, the reference count is decremented when a reference is removed. Once the count reaches zero, the object is deleted and its memory is reclaimed. A major advantage of this approach is that the memory overhead is small.

Our approach is strongly related to reference counting insofar as each node in the buffer keeps track whether it is still relevant to the remaining XQuery evaluation. Instead of counting references, we employ the concept of *roles* which are assigned to nodes. Intuitively, a role serves as a metaphor for the future relevance of a given node. While a traditional garbage collector is *passive* in the sense that it is invoked whenever there is no more space to allocate new objects, our approach differs in that it is *active*. That is, we purge buffers from irrelevant nodes early on. In fact, garbage collection is invoked whenever the scope of a variable ends. Thus, both the high watermark and the average main memory consumption remain low.

The basic idea behind *active garbage collection* is clean and simple: From the path expressions in the XQuery we statically derive a set of roles. While reading the input stream, the tokens are matched against the set of possible roles. A node can be assigned several roles when it is used in the query in several different contexts. Moreover, a role can be assigned to a node several times; this can happen if queries involve XPath expressions with descendant-axes.

At compile-time, we determine the moments during query evaluation when nodes lose roles. At runtime, the buffer manager is then notified that all nodes reachable via a path w.r.t. the current variable binding lose a certain role. Once a node has lost all of its roles, it can be safely deleted if none of its descendants is assigned any roles.

step	input stream	buffer contents	output stream
1			⟨*r*⟩
2	⟨*bib*⟩	bib$^{\{r_2\}}$	
3	⟨*book*⟩	bib$^{\{r_2\}}$ book$^{\{r_3,r_5,r_6\}}$	
4	⟨*title/*⟩	bib$^{\{r_2\}}$ book$^{\{r_3,r_5,r_6\}}$ title$^{\{r_5,r_7\}}$	
5	⟨*author/*⟩	bib$^{\{r_2\}}$ book$^{\{r_3,r_5,r_6\}}$ title$^{\{r_5,r_7\}}$ author$^{\{r_5\}}$	
6	⟨*/book*⟩	bib$^{\{r_2\}}$ book$^{\{\not r_3,\not r_5,r_6\}}$ title$^{\{\not r_5,r_7\}}$ ~~author~~$^{\{\not r_5\}}$	⟨*book*⟩ ⟨*title/*⟩ ⟨*author/*⟩ ⟨*/book*⟩
7		bib$^{\{r_2\}}$ book$^{\{r_6\}}$ title$^{\{r_7\}}$	

Figure 2. Active garbage collection.

Example. The following XQuery expression first outputs all children of the bib node for which no price exists. Next, book titles contained in the document are output.

```
<r> {
  for $bib in /bib return
    ((for $x in $bib/* return
       if (not(exists $x/price)) then $x else ()),
     for $b in $bib/book return $b/title)
} </r>
```

We refer to the for-loop introducing variable $\$bib$ by $for_{\$bib}$, and likewise use $for_{\$x}$ and $for_{\$b}$.

In static analysis, we derive the *projection tree* with nodes $n_1, \ldots, n_7$ shown in Figure 1. In the following we use the abbreviation "*dos*" for *descendant-or-self*. The projection tree defines the parts of the input that are copied into the buffer. For instance node n_4 (which refers to the if-condition in the query) defines that only the first price node – without descendants – needs to be buffered. However, due to n_5, we are forced to buffer all children of the bib node with their complete subtrees.

Each projection tree node n_i is assigned the role r_i. While parsing the input stream, the nodes of the document will be incrementally projected into the buffer and marked with roles on-the-fly. Further, *signOff*-statements are statically inserted into the query. At runtime, these statements notify the buffer manager that certain nodes lose their roles.

```
<r> {
  for $bib in /bib return
    ((for $x in $bib/* return
       (if (not(exists $x/price)) then $x else (),
        signOff($x,r3), signOff($x/price[1],r4),
        signOff($x/dos::node(),r5))),
     (for $b in $bib/book return
       ($b/title,
        signOff($b,r6),
        signOff($b/title/dos::node(),r7))),
     signOff($bib,r2))
} </r>
```

The query is sequentially evaluated on the buffer until input is required that has not been buffered (yet). In this case, the query evaluator *blocks* and requests further input, upon which the input stream is read until a token is found that is matched by the projection tree (or the stream is exhausted). As soon as a matching token t is found, we assign for each matched projection tree node n_i the role r_i to token t. Next, the token is loaded into the buffer and query evaluation is resumed. In contrast to projection as implemented in Galax [13], where the whole document is projected into the buffer before starting evaluation, in our *pull-based* approach the buffer is filled incrementally during evaluation, as needed by the evaluator. Whenever the query evaluator encounters a *signOff*-statement, it notifies the buffer that certain nodes lose their roles. The buffer then performs the role updates and invokes active garbage collection.

Let us consider the evaluation of the query on the input stream $\langle bib \rangle \langle book \rangle \langle title/ \rangle \langle author/ \rangle \langle /book \rangle \ldots$. Figure 2 shows for several steps what has been read from the input stream, the current buffer contents, and the output produced so far. In step 1, the opening tag $\langle r \rangle$ is output. Next, the query evaluator tries to evaluate $for_{\$bib}$, but has to block as the required input is not yet available in the buffer. In step 2, $\langle bib \rangle$ is read. As it is matched by projection tree node n_2, this document node is copied into the buffer and assigned role r_2. The query evaluator evaluates $for_{\$bib}$ and binds variable \$*bib* to the buffered node. Next, it tries to evaluate $for_{\$x}$, but has to block as relevant data is missing.

In step 3, $\langle book \rangle$ is matched by several projection tree nodes, and hence is buffered and assigned the roles r_3, r_5, and r_6. Now variable $\$x$ is bound to the book node, but the evaluation of the next query subexpression "if not(exists($x/price)) then $x else ()" has to wait for input. In step 4, the bachelor tag $\langle title/ \rangle$ is read. As matched by the projection tree, they will be buffered and annotated with roles r_5 and r_7. The evaluation of the if-expression blocks again, also after reading the author node.

In step 6, $\langle /book \rangle$ is read. The if-expression can be evaluated and the node to which $\$x$ is bound is output. Next, the sequence of *signOff*-statements is evaluated. For instance, execution of "signOff($x, r_3)" causes the buffered book node to lose role r_3. The author node loses its single role r_5 in the course of evaluating the *signOff*-statements and, as it has no descendants, can be purged from the buffer. Each of the remaining nodes carries a role which marks it as relevant for the future evaluation of $for_{\$b}$. Now query evaluation again returns to evaluating $for_{\$x}$ and blocks until the next token has been loaded into the buffer.

Contributions

- This paper proposes the first buffer manager for streaming XQuery engines which employs *static and dynamic* analysis to reduce main memory consumption.
- We introduce the notion of assigning *roles* to buffered nodes. Roles serve as a metaphor for the relevance of a node for query evaluation. We show how roles are assigned to nodes, how nodes lose roles during query evaluation, and when nodes can be deleted from buffers.
- We extend the well-established technique of static document projection [3, 13] so that roles are assigned to document nodes on-the-fly during projection.
- We propose *active garbage collection* as a novel buffer management technique for streaming XQuery engines. We explore our technique for the practical fragment of composition-free XQuery [10].
- Our prototype implementation shows the significant impact of active garbage collection on main memory consumption and query evaluation time. As confirmed by our experiments with XMark data and queries, combined static and dynamic analysis outperforms systems which rely on static analysis alone [11].

Structure. We provide the preliminaries in Section 2 before introducing our XQuery fragment in Section 3. The static analysis presented in Section 4 forms the groundwork for active garbage collection at runtime, which is presented in Section 5. Implementation and optimizations in our prototype system are introduced in Section 6. We conclude with the discussion of experimental results in Section 7.

2 Preliminaries

Let *Tag* be a set of node labels (or "tags") and let *Char* be a set of characters. We consider XML without attributes as our data model. This poses no substantial restriction as attributes can be handled in the same way as children of a node. Each XML document has a root node, which we refer to by *root*. We will repeatedly switch between the dual views of XML documents as unranked, ordered, and node labeled trees over the two-sorted domain of nodes (with tagnames from *Tag*) and values (strings over alphabet *Char*), and streams of opening and closing tags, and character sequences. The depth-first left-to-right traversal of the tree in document order yields the corresponding XML stream, while the stream encodes an unranked labeled tree. For a document tree T, let *dom* be the set of nodes. When comparing node-sets, i.e. sets over domain *dom*, we compare node-identifiers only. $|T|$ denotes the size of T.

Definition 1 Let T be a document tree and let *dom* be the set of nodes in T. Let $S \subseteq$ *dom* be a node-set with *root* $\in S$. The *projection of* T *w.r.t.* S, denoted $\Pi_S(T)$, is the document tree consisting of the node-set S, and with the ancestor-descendant and following relationships as in T. $\Box$

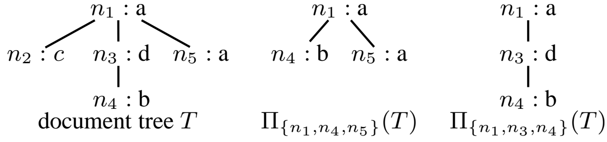
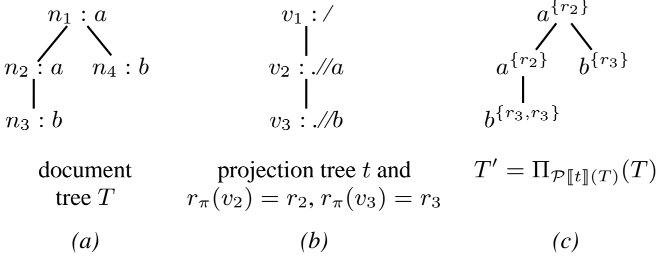


Figure 3. Document projection.



v_1 : /

v_2 : .//a v_3 : .//b

$a^{\{r_2\}}$

$a^{\{r_2\}}$ $b^{\{r_3\}}$

$b^{\{r_3\}}$

projection tree t' and $r_\pi(v_2) = r_2, r_\pi(v_3) = r_3$ $T'' = \Pi_{\mathcal{P}[\![t']\!](T)}(T)$

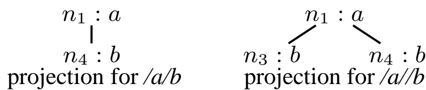
(d) *(e)*

Figure 4. Projection and role assignment.

Figure 3 shows an XML document tree T with node-set $\{n_1, \dots, n_5\}$ and tag names $\{a, b, c, d\}$, and the projected trees $\Pi_{\{n_1,n_4,n_5\}}(T)$ and $\Pi_{\{n_1,n_3,n_4\}}(T)$.

The goal in document projection is to preserve only those parts of the input document that are relevant for query evaluation, while discarding the rest. Previous work on projecting XML includes [3,4,13] and is based on projection paths to specify the nodes relevant for query evaluation. In [3,13], all document nodes which are matched by some prefix of a projection path *and* their ancestors are preserved, while in our approach ancestors of matched nodes need not always be kept. For instance, when projecting for XPath expression *//b* on tree T from Figure 3, we only preserve node n_4, rather than the projected document $\Pi_{\{n_1,n_3,n_4\}}(T)$.

Our approach is more effective in reducing the size of the projected documents when descendant axes are involved, For instance, consider the document tree from Figure 4(a). The projected documents for XPath expressions */a/b* and */a//b* are shown below.



Yet if we simultaneously project for *both* XPath expressions, e.g. as both occur in an XQuery expression, then we need to preserve the complete input tree in this example. Discarding node n_2 would promote node n_3 to a child of n_1, and the evaluation of XPath expression */a/b* on the projected document would produce an incorrect result.

A set of projection paths can be summarized in a projection tree. For instance, the projection tree in Figure 5(a) contains the XPath expressions */a/b* and */a//b*. Here, non-leaf nodes reflect the XPath (sub)expressions, while the leaf nodes are labeled with "*dos*" (*descendant-or-self*) and denote that, in projecting the document, descendant nodes of */a/b* and */a//b* must not be discarded.

Formally, a *projection tree* is an unranked, unordered tree where the root is labeled "/" (which denotes that all paths are absolute) and the inner nodes are labeled with location steps *axis::x*[p] where *axis* is an XPath axis *child*, *descendant*, or *descendant-or-self*, and x is either the symbol "$*$", a tagname, or the wildcard *node()*. Predicate [p] is either "[*true*]", in which case it can be omitted, or [*position*() = 1]. We will employ the position information for existence checks in XQuery expressions, where we are only interested in the first witness of a node. We use common XPath abbreviations, e.g. *//bib* for */descendant::bib*, and shorten *descendant-or-self* to *dos*.

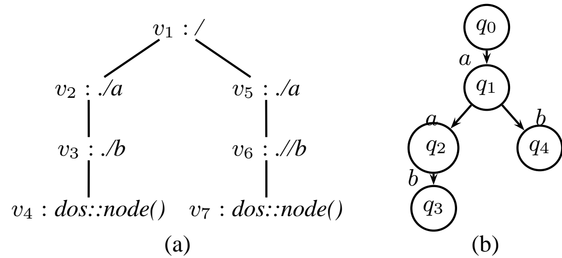


Figure 5. Projection with a lazy DFA.

Similar to processing XPath on streams [9], we realize stream preprojection with a lazily constructed deterministic finite automaton (DFA). Due to our restriction to a fragment of forward XPath [15], the decision whether to discard a document node can be already made when reading its opening tag from the input stream.

For instance, while projecting the input document from Figure 4(a), we compute the DFA in Figure 5(b). There is a straightforward mapping from DFA states (which reflect paths in the input document) to multisets of projection tree nodes. In detail, a DFA state q maps to a projection tree node v if the XPath representation of v (the path from root "/" to v) matches the input document path described by q. The multiplicity of the projection tree node in the multiset is defined as the number of possible path step assignments that lead to matches, e.g. XPath expression *//a//b* matches path */a/a/b* in two variations, either with path step *//a* bound to the first or to the second a in the path.

Example 1 For the projection tree in Figure 5(a) and the DFA in 5(b), q_0 maps to the singleton set $\{v_1\}$, q_1 maps to $\{v_2, v_5\}$, state q_2 maps to the empty set, state q_3 maps to $\{v_6\}$, and state q_4 maps to $\{v_3, v_6\}$.

Consider the same DFA and the projection tree in Figure 4(b). Here, state q_3 maps to the multiset $\{v_3, v_3\}$, as the XPath representation *//a//b* of v_3 matches path */a/a/b* of q_3 with multiplicity 2. $\square$

The mapping is exploited at runtime. Assume we are currently in a DFA state q and read an opening tag $\langle t\rangle$. We identify two cases where XML nodes read in the input document must be preserved. (1) There is a transition defined from state q into a state p under label t, where the successor state p maps to a node in the projection tree. (2) State q maps to nodes v and w in the projection tree (which need not be distinct), where v has a child labeled *child::a* and w has a child labeled *descendant::a* for the same tagname a. Intuitively, in case (1) the current node must be preserved as it matches a projection path, whereas case (2) avoids erroneous promotion of descendant nodes.

Example 2 For the document in Figure 4(a), the projection tree in 5(a), and the DFA in 5(b), a crucial point is reached when we are in DFA state q_1 and read node n_2 with tagname a from the document. The first condition for node preservation does not hold, as the successor state q_2 does not map to any node in the projection tree. Yet the second condition is satisfied, as state q_1 maps to v_2 and v_5 in the projection tree, which have children labeled *./b* and *.//b*. □

In general, given a projection tree t and an input document T, we denote the set of nodes which are in the projected document tree as $\mathcal{P}[t](T)$. For further examples, consider the projection tree t' in Figure 4 and the document trees T and T' in Figure 3. Here, $T' = \Pi_{\mathcal{P}[\![t']\!](T)}(T)$ is a projection of T w.r.t. t'.

Finally, we introduce the concept of roles, which forms the basis of active garbage collection at runtime. Let *roles* be a finite set of elements. A role-set is a multiset over *roles*, defined as a function where m : *roles* $\rightarrow \mathbb{N}$ maps roles to their multiplicity in the role-set. Naturally, multiplicity zero means a role is not contained in a role-set. A role set is empty if all roles have multiplicity zero. For syntactic convenience, we denote the empty role-set by $\emptyset$. We annotate nodes in document trees with role-sets, and introduce the *role-assignment function* ρ : *dom* $\rightarrow m$ which yields the multiset m of roles assigned to a given node. We further introduce functions for adding and removing roles, i.e. for a node n and a role r, let $\rho(n) = m$ and $m(r) = i$. Then after executing $add_\rho(r, n)$, $(\rho(n))(r) = i + 1$. Likewise, after executing $rem_\rho(r, n)$, if $i > 0$ then $(\rho(n))(r) = i - 1$, and if $i = 0$, the removal of roles is undefined.

Role assignment is closely coupled to projection trees and lazy DFAs. As we will show later, each projection tree node v defines a role r. Assume we last recently processed document node n while entering DFA state q, with q mapping to a non-empty set of projection tree nodes V. Then the document node will be buffered, and, for each $v_i \in V$, we will assign the corresponding role r_i to n.

Example 3 Consider the mapping of the DFA from Figure 5 to the projection tree in Figure 4(b) (see Example 1). Figure 4(c) shows the projected document with role assignment for the document tree in 4(a). The multi-role r_3 for the first b node (in document order) has been assigned due to the mapping from q_3 to $\{v_3, v_3\}$. □

$$
\begin{array}{rl}
Q ::= & \langle a\rangle q\langle/a\rangle \\
q ::= & () \mid \langle a\rangle q\langle/a\rangle \mid var \mid var/axis :: \nu \mid (q,\ \ldots, q) \\
\mid & (\text{if } cond \text{ then } \langle a\rangle \text{ else } (), q, \text{if } cond \text{ then } \langle/a\rangle \text{ else } ()) \\
\mid & \text{for } var \text{ in } var/axis :: \nu \text{ return } q \\
\mid & \text{if } cond \text{ then } q \text{ else } q \\
cond ::= & \text{true()} \mid \text{exists } var/axis :: \nu \mid var/axis :: \nu\ RelOp\ string \\
\mid & var/axis :: \nu\ RelOp\ var/axis :: \nu \\
\mid & cond \text{ and } cond \mid cond \text{ or } cond \mid \text{not } cond \\
axis ::= & \text{child} \mid \text{descendant} \\
\nu ::= & a \mid * \mid \text{text()} \\
RelOp ::= & \leq \mid < \mid = \mid \geq \mid >
\end{array}
$$

Figure 6. XQuery fragment XQ.

3 Query language

In this section we define our XQuery fragment XQ, which comprises arbitrarily nested for-expressions, conditions, and joins. As argued in [10], this XQuery fragment covers most queries without aggregates that arise in practice. The *abstract syntax* of an XQ query Q is shown in Figure 6 where $a \in$ *Tag*, *string* denotes a string value, and *var* is a set of XQuery variables $\$x, \$y, \ldots$ with the distinguished root variable \$*root*, the unique free variable in any query. We restrict our discussion to XQuery expressions in which both *cond* expressions generated by line three are syntactically equal, in order to assure well-formed XML output. Our query fragment currently only supports atomic equality [10] and no aggregations. However, we point out that many syntactically richer fragments of XQuery can be rewritten into our fragment, as in many practical queries, let-expressions can be removed [10] and queries can be normalized [11,13], thus rewriting where-conditions to if-then-else expressions and replacing for-loops with multi-steps by nested single-step for-loops (where possible).

Semantics of XQ. The semantics of XQ is the standard XQuery semantics. However, as we are operating on XML streams, we will interpret XQ expressions *strictly sequentially*. In particular, our role update mechanism via *signOff*-statements relies on this evaluation order. We define the evaluation of an XQ expression α with k free variables using a function $[\![\alpha]\!]_k$ that takes a k-tuple of trees as input (i.e., an *environment* for k variables). The symbol $\uplus$ denotes list concatenation, l_i the i-th element of list l, $[\ldots]$ is the list constructor, and $[\,]$ denotes the empty list. A for-loop is sequentially evaluated to a list of XML tokens as follows,

$$
[\![\text{for } \$x_{k+1} \text{ in } \$y/axis::\nu \text{ return } \beta]\!]_k(\vec{e}) :=
\biguplus_{1\leq i\leq |l|} [\![\beta]\!]_{k+1}(\vec{e}, l_i) \text{ where } l = [\![\$y/axis::\nu]\!]_k(\vec{e})
$$

i.e. variable $\$x_{k+1}$ is bound successively to each node in the list of nodes obtained from evaluating location step expression \$*y/axis::*ν, and the body of the for-loop is evaluated immediately for each new variable binding. For a listing of the complete evaluation strategy of XQ, we refer to [10].

$$\frac{\textit{if } X \textit{ then } \alpha \textit{ else } \beta}{(\textit{if } X \textit{ then } \alpha \textit{ else } (),\ \textit{if } (\textit{not } X) \textit{ then } \beta \textit{ else } ())}\ \text{DECOMP}$$

$$\frac{\textit{if } X \textit{ then } (\alpha_1, ..., \alpha_n) \textit{ else } ()}{(\textit{if } X \textit{ then } \alpha_1 \textit{ else } (), \ldots, \textit{if } X \textit{ then } \alpha_n \textit{ else } ())}\ \text{SEQ}$$

$$\frac{\textit{if } X \textit{ then } \langle a\rangle\ \alpha\ \langle /a\rangle \textit{ else } ()}{(\textit{if } X \textit{ then } \langle a\rangle \textit{ else } (),\ \textit{if } X \textit{ then } \alpha \textit{ else } (),\ \textit{if } X \textit{ then } \langle /a\rangle \textit{ else } ())}\ \text{NC}$$

$$\frac{\textit{if } X \textit{ then for } \$x \textit{ in } \$y/\textit{axis::nt return } \alpha \textit{ else } ()}{\textit{for } \$x \textit{ in } \$y/\textit{axis::nt return if } X \textit{ then } \alpha \textit{ else } ()}\ \text{FOR}$$

Figure 7. Pushing down *if*-expressions

Pushing *if*-Statements. Our approach relies on the assumption that for each buffered node the number of initially assigned roles and the number of *signOff*-commands received during query evaluation coincide. Role assignment takes place while projecting the input stream. At this time, conditions in *if*-expressions in general can not yet be decided. As a consequence, for the execution of *signOff*-statements inside the *then* or *else* parts of *if*-expressions no guarantees can be made. We show in Section 4 that *signOff*-statements always will be inserted at the end of *for*-loops. By pushing all *if*-expressions down into *for*-loops, we guarantee that no *signOff*-command will be created inside an *if*-expression.

The rewriting rules for pushing *if*-statements are shown in Figure 7. In a first step, we apply rule DECOMP to each *if-then-else*-expression in the query. The resulting query, which contains only empty *if*-expressions with empty *else* parts, is then rewritten by applying rules SEQ, NC and FOR in arbitrary order, until a fixpoint is reached. Rule SEQ pushes an *if*-expression inside the constituents of a sequence expression, while rule NC, which basically decomposes a node construct expressions, pushes *if*-expressions inside node constructs. Rule FOR completes the set of rules, and pushes *if*-expressions inside *for*-expressions. In practice, we might decide to process only those *if*-expressions with a *for*-loop as a subexpression.

Introducing *signOff*-Statements to XQ. In implementing garbage collection, we assign roles to buffered nodes. Nodes lose roles when they have become irrelevant for the remaining query evaluation. Hence, we need a mechanism for signalling the buffer manager at runtime that certain nodes lose their roles. To this end, *signOff*-statements are inserted into queries at compile-time.

A *signOff*-statement is an expression of the form signOff($\$x/\pi, r$) where $\$x$ is a variable, π is a relative path expression, and r is a role. Let T be a document tree and let ρ be the role-assignment function. Let $\vec{e}$ be an environment of k variables, let $\$x$ be a variable in $\vec{e}$, and let r be a role, then the semantics of $[\![\text{signOff}(\$x/\pi,\ r)]\!]_k(\vec{e})$ is the following: First, we define a node-set S. Let x be the node to which variable $\$x$ is currently bound, so $x = [\![\$x]\!]_k(\vec{e})$. If $\pi = \epsilon$, then $S = \{x\}$, otherwise S is the set of nodes reachable from node x via XPath expression π, i.e. $S := \mathcal{P}[\![\pi]\!](x)$. Next, we remove role r from all nodes n in S.

We state two requirements for the *safe* evaluation of an XQuery with *signOff*-statements: (1) All node removals at runtime are defined, and (2) after the query has been evaluated, all roles have been removed. These conditions enforce that exactly as many instances of roles are assigned to document nodes as are removed during query evaluation.

Example 4 Consider the following query and a version extended with *signOff*-statements for the roles r_1 and r_2.

```
<q> {for $a in //a
      return
      <a>
       {for $b in $a//b
         return <b/>}
      </a>}
</q>
```

```
<q> {for $a in //a
      return
      ((<a>
        {for $b in $a//b
          return (<b/>,
          signOff($b,r2))}
        </a>),
       signOff($a,r1))}
</q>
```

The evaluation of this query on document tree T' from Figure 4 is safe, but not on T''. □

Roles and Dependency Paths. For two query expressions α and β, we write $\alpha \preceq \beta$ (resp., $\alpha \prec \beta$) to denote that α is a subexpression (resp., proper subexpression) of β.

Let $\textit{Vars}_Q$ denote the set of variables occurring in query Q. For two variables $\$x, \$y \in \textit{Vars}_Q$, we say $\$y$ is the parent variable of $\$x$, denoted $\textit{parVar}_Q(\$x) = \$y$ if there exists a for-loop expression "for $\$x$ in $\$y/\textit{axis} :: \nu$ return α" in Q. We say $\$y$ is an ancestor variable of $\$x$, denoted $\$x <_Q \$y$, if either (1) $\$y = \textit{parVar}_Q(\$x)$ or (2) there exists a variable $\$z$ such that $\$x <_Q \$z$ and $\$z <_Q \$y$. We write $\$x \leq_Q \$y$ if either $\$x = \$y$ or $\$x <_Q \$y$.

The variable tree of a query summarizes the parent-child relationships between variables. Variable trees are unranked and unordered, and are defined over the nodes $\textit{Vars}_Q$ and with the edge relation $\textit{parVar}_Q$. The variable tree for the query from the introduction is shown in Example 5.

Given two variables $\$x <_Q \$y$, the variable path between $\$y$ and $\$x$ is defined recursively as follows. For $\$x = \$y$, $\textit{varpath}_Q(\$y, \$x) = \epsilon$. Otherwise, $\textit{varpath}_Q(\$y, \$x) = \textit{axis} :: \nu/\textit{varpath}_Q(\$z, \$x)$ where $\$z$ is a variable such that $\$x \leq_Q \$z <_Q \$y$ with the query expression "for $\$z$ in $\$y/\textit{axis} :: \nu$ return α" in Q.

Let r_Q : XQ $\rightarrow$ *roles* be an injective function assigning a role to each XQ expression. We define dependencies $\textit{dep}(\$x)$ as sets of tuples $\langle \$x/\pi, r\rangle$ where $\$x$ is a variable, π is a path expression, and r is a role. Informally, dependencies contain paths relative to the binding of variable $\$x$. In particular, in evaluating existence checks on XML streams, we are only interested in the first witness, while in output and comparison expressions, we are interested in the relevant nodes together with their subtrees.

Definition 2 Let Q be a query in XQ and $\$x \in \mathit{Vars}_Q$. The set of *dependencies of variable* $\$x$, denoted $\mathit{dep}(\$x)$, is defined as follows. Let $\beta \preceq Q$ with $r_Q(\beta) = r$, then

- $\langle \mathit{axis}::\nu[1], r\rangle \in \mathit{dep}(\$x)$ if $\beta =$"exists($\$x/\mathit{axis}::\nu$)",
- $\langle \mathit{axis}::\nu/\mathit{dos}::\mathit{node}(), r\rangle \in \mathit{dep}(\$x)$ if β is either an output expression of the form "$\$x/\mathit{axis}::\nu$" or a condition expression of the form "$\$x/\mathit{axis}::\nu\ \mathit{RelOp}\ \chi$" or "$\chi\ \mathit{RelOp}\ \$x/\mathit{axis}::\nu$ ", and
- $\langle \mathit{dos}::\mathit{node}(), r\rangle \in \mathit{dep}(\$x)$ if $\beta =$"$\$x$". □

4 Static Analysis

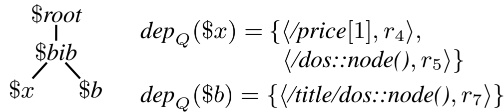
In the static analysis phase the projection tree is computed from the query, so that at runtime, a projected version of the XML input stream can be computed. Each projection tree node defines a role. As described in Section 2, these roles are assigned to buffered nodes while preprojecting the document. By statically inserting *signOff*-statements into the query, buffered nodes finally can be deleted at runtime, once they have become irrelevant to query evaluation. We can make the following guarantees.

Theorem 1 (Correctness) *Let Q be an XQ query, T be the input document tree, Q' be the rewritten query Q with signOff-statements, and let T' be the projected document tree with assigned roles. Then $[\![Q]\!]_1(T) = [\![Q']\!]_1(T')$.*

Deriving Projection Trees. Given an XQuery Q in our fragment, we now show how to derive the projection tree that will be used to compute the projected document. The key ideas of our approach are the following. For existence-checks in conditions, it suffices to keep the first witness for a path, as any further witnesses are irrelevant for query evaluation. Whenever a node is output or compared in conditions, the node and all of its descendants need to be contained in the projected document tree. Finally, when for-loops iterate over node-sets, the nodes to which the variables bind are relevant to query evaluation, yet their subtrees are irrelevant for the variable bindings per se. These considerations are captured by the dependencies (see Def. 2).

Given query Q, we derive the projection tree t and a mapping r_π from nodes in t to roles in three steps: First, we construct the variable tree of Q. Next, the variable tree is extended by nodes labeled with path expressions: For each variable $\$x$ and for each $\langle \$x/\pi, r\rangle \in \mathit{dep}(\$x)$, we add a node n with label "π", an edge from $\$x$ to n, and we define $r_\pi(n) := r$. As a final step, the root node is relabeled "/" and for each variable node n labeled $\$x$ with the corresponding for-loop $\beta =$"for $\$x$ in $\$y/\mathit{axis}::\nu$ return α", we relabel n with "$\mathit{axis}::\nu$" and define $r_\pi(n) = r_Q(\beta)$.

Example 5 Consider the query from the introduction with its variable tree and dependencies as shown below.



The final projection tree is shown in Figure 1. □

Rewriting XQ Queries. At runtime, the goal is to issue *signOff*-statements as early as possible so that the size of the main memory buffer remains small. At the same time, update commands must never be issued too early, as this could corrupt the query result. The insertion of *signOff*-statements into queries must assure the latter.

Definition 3 Let Q be an XQ query and let $\$z \in \mathit{Vars}_Q$. Variable $\$z$ is *straight* if either $\$z = \$\mathit{root}$ or there is a query expression $\beta =$"for $\$z$ in $\$y/\mathit{axis}::\nu$ return α" such that (1) $\$y$ is straight and (2) there is no for-loop expression $\gamma =$"for $\$u$ in $\$v/\mathit{axis}'::\nu'$ return α'" where $\$u$ is no ancestor variable of $\$z$ and $\beta \prec \gamma \preceq Q$. □

Definition 4 Let Q be a query and let $\$x \in \mathit{Vars}_Q$. The *first straight ancestor variable* of $\$x$ is defined as

$$\mathit{fsa}_Q(\$x) \stackrel{def}{:=} \begin{cases} \$x & \text{if } \$x \text{ is straight} \\ \mathit{fsa}_Q(\mathit{parVar}_Q(\$x)) & \text{otherwise.} \end{cases}$$ □

Example 6 Variables $\$a$ and $\$b$ in the queries from Example 4 are straight, i.e. $\mathit{fsa}_{Q_1}(\$a) = \$a$ and $\mathit{fsa}_{Q_1}(\$b) = \$b$. In the queries from Figure 9, variable $\$b$ is not straight, in particular, $\mathit{fsa}_{Q_2}(\$b) = \$\mathit{root}$. □

We are now in the position to state the rules for inserting *signOff*-statements into queries. Informally, at the end of the scope of each variable $\$x$, all nodes that depend on $\$x$ and for which $\$x$ is the first straight ancestor variable lose their assigned roles. The *static XQ rewriting rules* shown below use algorithm su_Q (Figure 8). This algorithm computes all *signOff*-commands for a given variable, i.e. (1) for each variable $\$x$ different from $\$\mathit{root}$ the role update for all document nodes variable $\$x$ will be bound to is emited, and (2) for each dependency of the variable, a corresponding *signOff*-statement will be created.

$$\frac{\beta : \langle a\rangle\ \alpha\ \langle /a\rangle}{\langle a\rangle\ (\alpha, \mathit{su}_Q(\$\mathit{root}))\ \langle /a\rangle} \quad (\beta = Q)$$

$$\frac{\beta : \{\text{for } \$x \text{ in } \$y/\sigma \text{ return } \alpha\}}{\{\text{for } \$x \text{ in } \$y/\sigma \text{ return } (\alpha,\ \mathit{su}_Q(\$x))\}} \quad (\beta \prec Q)$$

The first rule applies to the query Q itself and inserts the corresponding *signOff*-statements for variable $\$\mathit{root}$. The second rule inserts *signOff* commands for the remaining variables, always at the end of their introducing for-loops.

Example 7 Let Q (Q') denote the query (rewritten query) from Example 4. Each buffered document node to which a variable $\$a$ or $\$b$ is bound loses its role once the scope of the respective variable ends. We denote the result of evaluation query Q against document tree T with variable $\$\mathit{root}$ bound to the document root by $[\![Q]\!]_1(T)$. For document trees T and T' from Figure 4, we can verify that $[\![Q]\!]_1(T) = [\![Q']\!]_1(T')$. □

Example 8 Let Q (Q') denote the query (rewritten query) from Figure 9. Figure 4 shows the projection tree t' for Q and the annotated projection of T. The role updates for the b nodes are issued with the end of the scope of variable $\$a$. Here, $[\![Q]\!]_1(T) = [\![Q']\!]_1(T'')$. □

Algorithm $su_Q(variable\ \$x)$:
begin
  if ($\$x \neq \$root$) **then**
  begin
    let $\$x$ be defined in β :"for $\$x$ in $\$y/axis::\nu$ return α";
    emit "signOff($\$x$, $r_Q(\beta)$)";
  end
  for each variable $\$z$ in $Vars_Q$ such that $fsa_Q(\$z) = \$x$
  begin
    let $\sigma = varPath_Q(\$x, \$z)$;
    for each $\langle \pi, r \rangle \in dep_Q(\$z)$ **emit** "signOff($\$x/\sigma/\pi, r$)";
end end

Figure 8. Static query rewriting.

```
<q>
  {for $a in //a
    return
      <a>
       {for $b in //b
         return <b/>}
      </a>
} </q>
```

```
<q> ({for $a in //a
       return
       ((<a>
          {for $b in //b
           return <b/>}
          </a>),
        signOff($a,r1))},
     signOff($root//b,r2))
</q>
```

Figure 9. Inserting *signOff*-statements.

5 Active Garbage Collection

Active garbage collection relies on the correct interplay of (1) the assignment of roles to buffered document nodes and (2) the timely removal of roles and ultimately, document nodes from the buffer. A buffered node is called *irrelevant* if neither the node itself nor any of its descendants carry a role. In the following discussion we assumes that the buffer contains the projected input document and that buffered nodes for which the closing tag has not yet been read are marked "unfinished". At runtime, streaming document projection and role assignment are coupled, so that document nodes are always copied into the buffer together with their corresponding roles.

Normally, traditional garbage collectors start searching for memory that can be freed whenever there is no more space to allocate new objects. Our approach differs in that garbage collection is *active*. That is, we purge buffers from irrelevant nodes every time a *signOff*-statement is issued by the query evaluator. As the garbage collector is invoked quite often, it is desirable to restrict the search space for irrelevant nodes within the buffer. Figure 10 shows how we handle *signOff*-statements and perform a *localized* garbage collection: After a node has lost a role due to a *signOff*-statement, the garbage collector checks whether it can be deleted. If this is possible, the garbage collection proceeds bottom-up in the tree. Thus, deletion of nodes from the buffer can propagate up to the document root node. The treatment of "unfinished" nodes in the buffer requires extra care. An unfinished node is not deleted to avoid buffer corruption. Instead, it is *marked* deleted and ultimately purged from the buffer once the corresponding closing tag is read from the input stream.

Algorithm *signOff*($\$x/\pi$, *role* r):
begin
  let x be the node to which $\$x$ is bound;
  let node-set S be defined as follows:
    if ($\pi = \epsilon$) **then** $S := \{x\}$ **else** $S := \mathcal{P}[\![\pi]\!](x)$;
  for each node n in S
  begin
    execute $rem_\rho(r, n)$; *// remove role from nodes in S*
    while ($n \neq root$ **and** n is irrelevant) *// local search*
    begin
      let p be the parent node of n;
      if (n is finished) **then** delete n;
      else mark n as deleted
        and *ultimately* delete n when its closing tag is read;
      $n := p$;
end end end

Figure 10. Localized garbage collection.

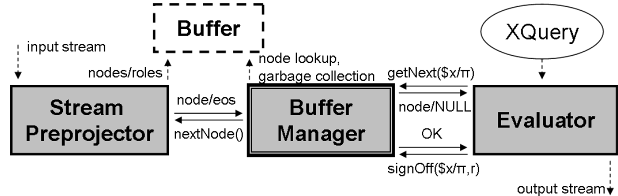


Figure 11. System architecture.

Our experiments confirm that the overhead imposed by the buffer cleanup algorithm is small in practice. A key prerequisite to this small overhead is that in algorithm *signOff* for each updated node, buffer updates start at the local position of the update and stop as soon as the first irrelevant node is detected. The concepts of *aggregate roles* and *elimination of redundant roles* presented in Section 6 will further reduce the computational overhead of buffer cleanup.

6 System Implementation

We have implemented active garbage collection for a prototype XQuery engine, the GCX system. GCX is implemented in C^{++} which, in contrast to garbage collected languages, gives direct control over memory allocation and deallocation, a crucial aspect when designing a query engine with low memory consumption.

System Architecture. The architecture of GCX comprises three components, the *query evaluator*, the *stream preprojector*, and the *buffer manager*, as sketched in Figure 11. The interaction between the components is *pull-based* as follows. (1) The query evaluator evaluates the rewritten XQ expression until it has to *block* either because a new node is required (e.g. when a variable is bound to the next node in its for-loop) or a *signOff*-statement is encountered. In both cases, a request is issued to the buffer manager, and query evaluation remains blocked until the buffer manager has responded. (2) The buffer manager answers to the requests of the query evaluator. If data is required that is not resident in the buffer, the buffer manager in turn

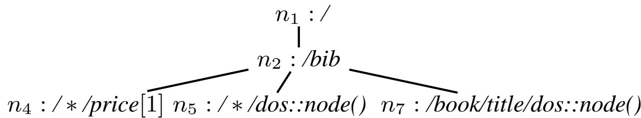


Figure 12. Projection tree.

sends *nextNode()*-requests to the stream preprojector until the data is available in the buffer or it has become evident that the data does not exist in the input (e.g. as the input has been exhausted). The reception of *signOff*-statements triggers the active garbage collection, as discussed in Section 5. (3) Once it has been activated by the buffer manager, the stream projector processes the input stream until a token relevant to query evaluation is detected. This token is then copied directly into the buffer, together with its associated roles. Via this chain of commands, the query evaluator incrementally reads the input stream and evaluates the query on-the-fly. In the following, we discuss the design decisions in GCX .

Buffer Representation. As our query fragment is composition-free [10], all XQuery variables bind to nodes in the document tree. Hence, there is a *single* buffer which contains the (currently relevant) projected document tree. Our buffer datastructure is simple, with parent-child and next-sibling pointers between nodes, thus keeping the memory overhead for the tree representation small. Moreover, we use a symbol table to replace tagnames by integers.

Early Updates. In the rewritten query from the introduction, "signOff($book/title/dos::node(), r_7)" is issued after the title node has been output. Yet if a book has more than one title, garbage collection is only invoked after *all* titles have been output. To avoid this suboptimal behaviour, we rewrite all output expressions "$\$x/\sigma$" to equivalent expressions "for $\$y$ in $\$x/\sigma$ return $\$y$" with new variable $\$y$. In the latter case, static query rewriting yields "for $y in $x/$\sigma$ return ($y, signOff($y, r))", where r is a new role. Now, titles lose role r_7 immediately after they have been output.

Aggregate Roles. Each output expression involving a variable induces a role that will be assigned to complete subtrees rather than a single node, i.e. every node that will be output as part of the expression is marked with the role. In our implementation, the root node of the subtree is assigned an *aggregated role* instead, which implicitly is "inherited" by its descendants. This optimization reduces the size of the role-set while it requires only minor changes to the projection tree and the garbage collection mechanism.

Elimination of Redundant Roles. For many queries, roles are introduced that actually are redundant. For instance, consider the query from the introduction. If the projection tree is changed as shown in Figure 12 and the *signOff*-statements "signOff($x,$r_3$)" and "signOff($b,r_6)" are removed from the rewritten query, then query evaluation and active garbage collection are still executed correctly. Redundant roles can be detected by inspecting projection trees. If they are not assigned during stream projection, and if the corresponding *signOff*-statements are removed from the queries, then both main memory consumption and runtime benefit from this optimization.

7 Experimental Results

Our implementation GCX was experimentally evaluated using a number of XMark [21] queries. Our prototype was implemented exactly as described in this paper. We emphasize that no other optimizations were applied than described in the implementation section.

As the XQ fragment introduced in Section 3 does not cover the full XQuery standard, queries were adapted accordingly. In detail, for all benchmarks, we converted XML attributes into subelements, replaced aggregations such as *count*($\$x$) by outputting the value of $\$x$ instead and rewrote multi step paths in for-loops to single step paths. $Q20$ is identical to $Q20$ from [7], with paths steps transformed to single path step expressions. All systems were benchmarked using the adapted streams and queries.

We considered XMark documents of sizes between 10MB and 200MB, generated with the XMark data generator. Benchmarks were carried out on a 3GHz CPU Intel Pentium IV with 2GB RAM, running SuSe Linux 10.0. All Java-based systems were executed using J2RE v1.4.2. As reference implementations we considered a broad spectrum of XQuery engines: The most appropriate systems are Saxon v8.7.1 [18], FluXQuery [7], and QizX/open v1.1 [17] (all three Java based), as they are capable of evaluating XQuery on large input documents. In particular, the FluXQuery engine has been designed for XML stream processing. In our experiments, we provided the XMark DTD to FluXQuery. Further, we considered the MonetDB system v4.12.0 combined with XQuery-module v0.12.0 [14] which relies on secondary storage. Finally, we used the in-memory XQuery engine Galax v0.6.8 [8], a reference implementation for the XQuery standard. While Galax has not been designed with XML stream processing in mind, it is often consulted in XQuery benchmarks and – for this reason – also included here. Note that the static projection of Galax [13] could not be made to work.

The focus of our experiments is primarily on main memory consumption, but we also considered query execution time. Main memory consumption was measured with the Linux *top* command. For each system and query we set a timeout of 1 hour. Figure 1 shows the results of our experiments. For each system and size of the input document, we measured the high watermark of non-swapped memory consumption, and the total query evaluation time. "n/a" indicates that the query could not be expressed in the language supported by the specific engine, while "-" denotes failure, e.g. caused by segmentation faults. With the Java-based engines, we could observe that due to effects caused by automatic memory management and the Java Virtual Machine, memory consumption often increased with the document size even though the buffer size remained constant (e.g. for FluXQuery).

Table 1. Benchmark results.

Query		GCX	FluXQuery	Galax	MonetDB	Saxon	QizX
XMark Q1	10MB	**0.18s / 1.2MB**	1.59s / 50MB	5.45s / 186MB	0.86s / 30MB	1.48s / 80MB	1.20s / 38MB
	50MB	**0.92s / 1.2MB**	3.96s / 111MB	42.33s / 880MB	3.69s / 98MB	4.29s / 292MB	3.74s / 195MB
	100MB	**1.87s / 1.2MB**	6.94s / 111MB	02:07 / 1,8GB	7.19s / 225MB	7.96s / 547MB	6.56s / 285MB
	200MB	**3.53s / 1.2MB**	12.27s / 111MB	timeout	13.60s / 244MB	14.30s / 973MB	11.82s / 480MB
XMark Q6	10MB	**0.34s / 1.2MB**	n/a	7.66s / 240MB	0.98s / 29MB	1.73s / 82MB	1.56s / 33MB
	50MB	**1.68s / 1.2MB**	n/a	57.98s / 1.2GB	5.06s / 111MB	5.78s / 292MB	6.13s / 169MB
	100MB	**3.33s / 1.2MB**	n/a	5:08 / 2GB	9.94s / 253MB	10.85s / 622MB	11.74s / 484MB
	200MB	**6.42s / 1.2MB**	n/a	timeout	19.95s / 337MB	20.14s / 1.2GB	20.33s / 805MB
XMark Q8	10MB	**13.15s / 9.8MB**	18.04s / 128MB	01:04 / 377MB	02:56 / 407MB	6.61s / 145MB	9.89s / 148MB
	50MB	**05:13 / 43MB**	06:51 / 169MB	33:08 / 1.8GB	03:26 / 1.35GB	02:02 / 352MB	03:38 / 265MB
	100MB	**22:07 / 86MB**	27:01 / 216MB	timeout	-	08:39 / 650MB	14:27 / 397MB
	200MB	**timeout**	timeout	timeout	-	32:43 / 1.15GB	52:05 / 636MB
XMark Q13	10MB	**0.17s / 1.2MB**	1.60s / 52MB	5.92s / 182MB	0.80s / 31MB	1.53s / 48MB	1.26s / 28MB
	50MB	**0.85s / 1.2MB**	3.98s / 111MB	43.91s / 899MB	3.64s / 98MB	4.45s / 292MB	3.85s / 195MB
	100MB	**1.69s / 1.2MB**	7.00s / 111MB	02:04 / 1.8GB	7.34s / 224MB	8.35s / 547MB	6.81s / 285MB
	200MB	**3.24s / 1.2MB**	12.33s / 111MB	timeout	13.52s / 271MB	15.02s / 1.05GB	12.30s / 480MB
XMark Q20	10MB	**0.25s / 1.2MB**	1.65s / 48MB	6.95s / 215MB	0.85s / 34MB	1.65s / 62MB	1.43s / 39MB
	50MB	**1.24s / 1.2MB**	4.19s / 111MB	53.08s / 1,5GB	4.17s / 120MB	4.90s / 292MB	4.18s / 195MB
	100MB	**2.48s / 1.2MB**	7.37s / 111B	03:14 / 2GB	8.47s / 247MB	9.13s / 622MB	8.71s / 350MB
	200MB	**4.74s / 1.2MB**	13.14s / 111MB	timeout	16.40s / 296MB	16.58s / 1.15GB	15.80s / 628MB

The experimental results confirm our expectations, namely the significant impact of combined static and dynamic buffer minimization on XQuery evaluation. Regarding memory usage, even for small stream sizes, GCX outperforms most competitors by a factor of 10 or more. Notably, FluXQuery can evaluate queries Q1 and Q13 with very little buffering, yet GCX shows an overall good performance for small and large documents.

For queries $Q1$, $Q6$, $Q13$ and $Q20$, memory consumption of our prototype is independent of the input stream size. Little has to be buffered at a time and we observe that low main memory consumption coincides with low evaluation time, also for the FluXQuery system. Note that $Q6$, which contains descendant axis XPath expressions, is not supported by FluXQuery. $Q8$ involves an XQuery join and more nodes have to be buffered. However our system manages to evaluate this query with low main memory consumption. Similar to the FluXQuery system, joins are implemented as naive nested loop joins, so runtime deteriorates for larger input documents on $Q8$. While runtime is vital for practical systems, this is an orthogonal issue and can be easily improved with standard database techniques.

In summary, the experiments confirm that our buffer management approach via active garbage collection performs well both w.r.t. main memory consumption and execution time. For a large class of queries, we can even outperform query engines which exploit schema information [11].

References

[1] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. "On the Memory Requirements of XPath Evaluation over XML Streams". In *Proc. PODS'04*, pages 177–188, 2004.

[2] M. Benedikt, W. Fan, and F. Geerts. "XPath Satisfiability in the Presence of DTDs". In *Proc. PODS*, pages 25–36, 2005.

[3] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. "Type-Based XML Projection". In *Proc. VLDB'06*, 2006.

[4] S. Bressan et al. "Accelerating Queries by Pruning XML Documents". *TKDE*, 54(2):211–240, 2005.

[5] L. Fegaras, R. Dash, and Y. Wang. "A Fully Pipelined XQuery Processor.". In *XIME-P*, 2006.

[6] L. Fegaras et al. "Query Processing of Streamed XML Data". In *Proc. CIKM 2002*, pages 126–133, 2002.

[7] "The FluXQuery Engine", 2004. http://www-db.cs.uni-sb.de/~scherzin/FluXQuery.html.

[8] "Galax". http://www.galaxquery.org/.

[9] T. J. Green et al. "Processing XML Streams with Deterministic Automata". In *Proc. ICDT'03*, pages 173–189, 2003.

[10] C. Koch. "On the complexity of nonrecursive XQuery and functional query languages on complex values". *ACM Transactions on Database Systems*, **31**(4), 2006. *To appear.*

[11] C. Koch et al. "Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams". In *Proc. VLDB'04*, pages 228–239, 2004.

[12] X. Li and G. Agrawal. "Efficient evaluation of XQuery over streaming data". In *Proc. VLDB'05*, pages 265–276, 2005.

[13] A. Marian and J. Siméon. "Projecting XML Documents". In *Proc. VLDB'03*, pages 213–224, 2003.

[14] "MonetDB/XQuery". http://monetdb.cwi.nl/XQuery/.

[15] D. Olteanu et al. "XPath: Looking Forward". In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 109–127, 2002.

[16] D. Olteanu et al. "An Evaluation of Regular Path Expressions with Qualifiers against XML Streams". In *Proc. ICDE'03*, page 702, 2003.

[17] "Qizx/open". http://www.axyana.com/qizxopen/.

[18] "Saxon". http://saxon.sourceforge.net/.

[19] H. Su et al. "Semantic Query Optimization for XQuery over XML Streams". In *Proc. VLDB*, pages 277–288, 2005.

[20] P. R. Wilson. "Uniprocessor Garbage Collection Techniques". In *Proc. IWMM'92*, pages 1–42, 1992.

[21] "XMark". http://monetdb.cwi.nl/xml/.