The GCX System: Dynamic Buffer Minimization in Streaming XQuery Evaluation

Christoph Koch Stefanie Scherzinger Michael Schmidt
Saarland University Database Group, Saarbrücken, Germany
{koch, scherzinger, schmidt}@infosys.uni-sb.de

ABSTRACT

In this demonstration, we present the main-memory based streaming XQuery engine GCX which implements novel buffer management strategies that combine static and dynamic analysis to keep main memory consumption low. Depending on the progress made in query evaluation, memory buffers are dynamically purged and minimized. In this demo, we show the various stages in evaluating a practical fragment of XQuery with GCX. We present the major steps in static analysis and demonstrate the mechanisms of dynamic buffer minimization. We apply our system to XML streams and demonstrate the significant impact of our approach on reducing main memory consumption and running time.

1. MOTIVATION

It has been repeatedly observed that memory consumption remains a crucial bottleneck when XQuery is evaluated by main-memory based systems [2–4,10,12,18]. In particular when processing large XML streams, it is often impossible to buffer the complete input prior to query evaluation, and an effective buffer management becomes *the* key prerequisite to performance. 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 [4, 10–12, 14]. Among the proposals discussed in previous works are the static projection of the input [2, 3, 12] and the streaming evaluation of parts of the query with no or only little buffering [5, 10, 11, 18]. However, for many practical queries involving blocking operators or descendant axes and wildcards, little can be evaluated on the fly [1, 10].

In [17], we show that a *combination* of static analysis and dynamic buffer minimization techniques further reduces main memory consumption during XQuery evaluation. The concept of *active garbage collection* forms the groundwork for the buffer management in our streaming XQuery engine GCX. GCX exploits static and dynamic analysis to actively purge main memory buffers based on the progress in query evaluation. GCX is an open source project and available online [8].

2. ACTIVE GARBAGE COLLECTION

Garbage collection [19] 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.

Active garbage collection for XQuery engines [17] is strongly related to reference counting, as each single 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 node. Roles are statically derived from the query. While reading the input, the XML nodes are matched against the set of 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 multiple times when queries involve the XPath descendant axis.

As an example, let us consider the XQuery below, which first outputs all children of the bib node for which no price exists. It then outputs all titles of books.

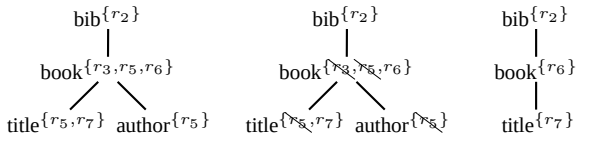
```
<r> {
  for $bib in /bib return
    (for $x in $bib/* return
       if (not(exists $x/price)) then $x else (),
     for $b in $bib/book return $b/title)
} </r>
```

Below we show the roles for this query and the paths addressing the nodes to which they will be assigned. For instance, role r_3 is assigned to all nodes in the document selected by XPath expression */bib/*$*$. In order to verify the existence of a price node, we are only interested in the first witness (hence the predicate $[1]$ in identifying r_4). These paths closely correspond to the notion of *projection paths* [2, 3, 12], and are evaluated accordingly: While the input stream is read, only the XML nodes that are matched by a projection path are considered query-relevant and consequently put into the buffer. While the input stream is being projected incrementally, roles are assigned to the buffered nodes on-the-fly.

r_1:	*/*	r_4:	*/bib/* $*$ */price[1]*
r_2:	*/bib*	r_5:	*/bib/* $*$ */descendant-or-self::node()*
r_3:	*/bib/*$*$	r_6:	*/bib/book*
		r_7:	*/bib/book/title/descendant-or-self::node()*

Figure 1*(a)* shows the buffer contents for the input stream prefix "$\langle bib\rangle\langle book\rangle\langle title/\rangle\langle author/\rangle\langle /book\rangle\ldots$". Each node is assigned roles, for instance, the *bib* node is assigned role r_2. Node *book* is matched by three projection paths, and is thus assigned three roles. Only nodes that carry at least one role are copied into the buffer.

At compile-time, we determine the moments during query evaluation when nodes lose roles. At these *preemption points*, the buffer manager is notified that all nodes reachable from the current variable via a path lose a role. Once a node has lost all of its roles, it



(a) Input document with role assignment **(b) Executing signOff commands** **(c) Garbage collection**

Figure 1: Active garbage collection.

can be discarded by the garbage collector, provided that none of its descendants is assigned a role. We need a mechanism which signals the buffer manager that nodes are about to lose roles. To this end, *signOff*-statements are inserted into queries at compile-time. We assume a sequential semantics to XQuery evaluation. Then the query from our running example is rewritten as follows.

```
<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/descendant-or-self::node(),r5))),
  (for $b in $bib/book return
   ($b/title,
    signOff($b,r6),
    signOff($b/title/descendant-or-self::node(),r7)
   )),
  signOff($bib,r2)) }
</r>
```

If the rewritten query is evaluated on the input tree from Figure 1*(a)*, we first output the opening tag $\langle r \rangle$. Subsequently, variable $\$x$ is bound to the *book* node, which then is written to the output together with its subtree. Next, the *signOff*-statements are executed. For instance, "signOff($x, r_3)" causes the buffered *book* node to lose role r_3, because this is the node to which variable $\$x$ is currently bound to. This node further loses role r_5. Likewise, the *title* and the *author* nodes each lose one instance of role r_5. Figure 1*(b)* shows the effect of role removal on the buffer.

The *signOff*-statements also trigger the garbage collection. Figure 1*(c)* shows the buffer contents after purging all nodes that are now irrelevant for the remaining query evaluation. The remaining nodes are required for the evaluation of the for-loop binding variable $\$b$, once the evaluation of the current for-loop has finished.

3. THE GCX XQUERY ENGINE

We have implemented active garbage collection for a prototype XQuery engine, called GCX. GCX supports the practical fragment of composition-free XQuery [9] with single-step nested for-loops[1], conditions, and joins, but does not yet cover aggregation. The system is implemented in *C++* which, in contrast to garbage collected languages, gives direct control over memory allocation and deallocation. This is crucial, as we want deletions put into effect immediately, to keep the main memory consumption of our query processor low *throughout* query evaluation.

Static analysis. Given an XQuery expression, a set of projection paths is derived (see [17]). Each projection path defines a role. When the input stream is read, each node that is matched by a projection path is copied into the buffer and assigned the corresponding roles. This can be done on-the-fly, with a lookahead of just one token. To mark the moments in time when buffered nodes are deleted during query evaluation, the preemption points in query evaluation are defined and *signOff*-statements are inserted into the query. The key to efficiency is to issue *signOff*-statements as early as possible so that the size of the main memory buffer remains small. At the same time, these commands must not be issued too early, as this could corrupt the query result.

In our demo, we will visualize the mapping between query expressions, paths, and roles, as shown in Figure 3(a). When a role in the role browser is selected, the corresponding line in the input query and the preemption points in the execution plan are highlighted. This allows the audience to interactively explore the connections between roles, paths, and *signOff*-statements.

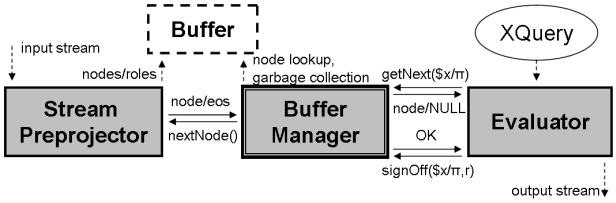


Figure 2: The GCX Runtime Architecture.

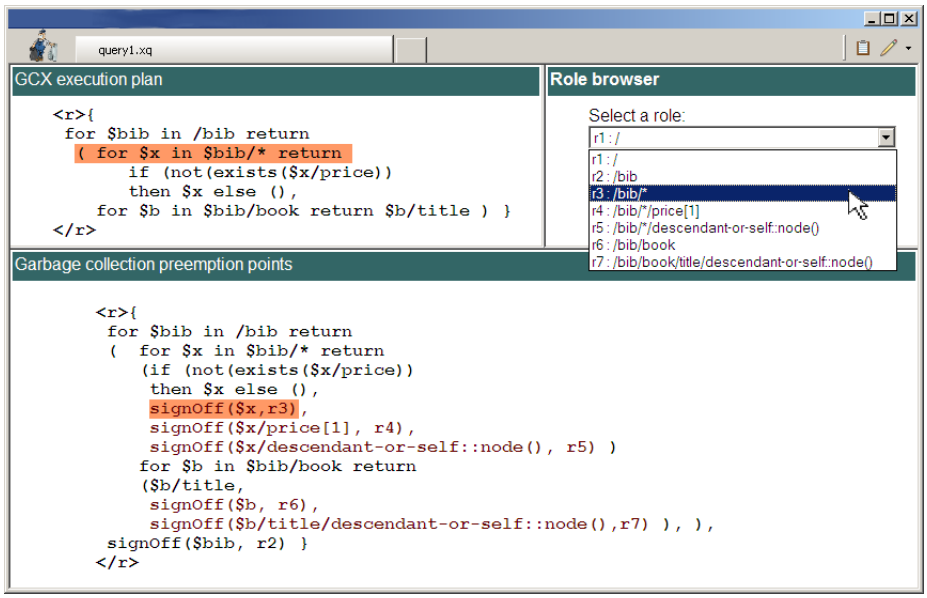
The runtime engine. The architecture of GCX comprises the *query evaluator*, the *stream preprojector*, and the *buffer manager*, as sketched in Figure 2. The interaction between the components is *pull-based*. The query evaluator sequentially evaluates the query expressions until it has to *block* either 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 consequence, a request is issued to the buffer manager, and query evaluation remains blocked until the buffer manager has responded. The buffer manager answers to the query evaluator. If data is required that is not resident in the buffer, the buffer manager 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 garbage collection. The stream preprojector reads the input until a token is matched by a projection path. The token is copied directly into the buffer, and roles are assigned. Via this chain of commands, GCX evaluates the query on-the-fly while processing the input stream.

Dynamic buffer management. At runtime, *signOff*-statements cause buffered nodes to lose roles. Eventually, the buffered nodes can be purged from the buffer. In our demonstration, we visualize the dynamic buffer behavior as shown in Figures 3(b) and (c). Consider the query from the introduction. We show the buffer consumption for different inputs. Each input document contains a *bib* root node with ten children of the form

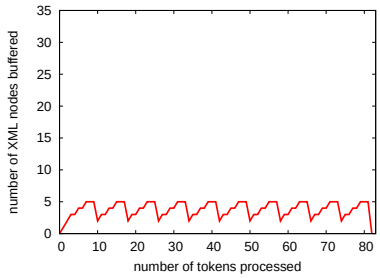
$$\langle t \rangle \langle author \rangle \langle /author \rangle \langle title \rangle \langle /title \rangle \langle price \rangle \langle /price \rangle \langle /t \rangle$$

where *t* is either tag *book* or *article*, a total of 82 tags forming 41 document nodes. For each new token read from the input (x-axis), we plot the number of buffered nodes after the token has been processed (y-axis). Figure 3(b) plots the buffer consumption for documents consisting of nine *article* nodes and one *book* node (in this order). Resuming the discussion from the introduction, article nodes and their descendants are assigned roles r_4 and r_5, and receive *signOff*-commands immediately after the article has been processed in the first *for*-loop. Thus, articles are processed one at

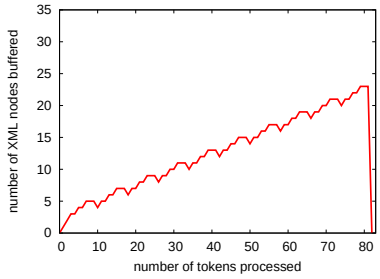
[1]A for-loop is *single-step* if it is of the form "for $\$x$ in $\$y$*/axis::*$\nu$ return α" where *axis* is an XPath axis and ν is an XPath node test.



(a) **Visualization of input query, the set of roles, and the preemption points for garbage collection computed by static query analysis**



(b) **9×*article* + 1×*book***

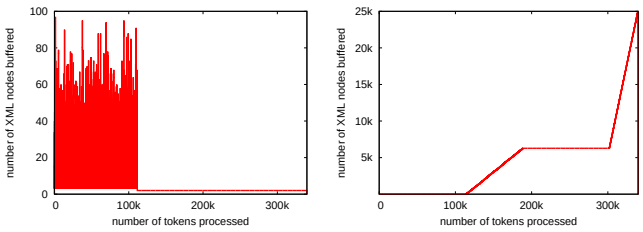


(c) **9×*book* + 1×*article***

Figure 3: Exploration of Static Analysis and Dynamic Buffer Management.

a time and memory consumption is bounded. The document for Figure 3(c) starts with 9 *book* nodes. For these nodes, the *title* child with role r_7 must be kept, while *price* and *author* children are removed when receiving the signOff-commands from the first *for*-loop. Although nodes that have become irrelevant are removed timely, buffer consumption increases. When the closing tag of the *bib*-node is read, 23 nodes are buffered in total.

Figure 4 plots the buffer for two XMark queries [20] on a 10MB document generated with the XMark data generator. As the fragment supported by GCX does not cover the full XQuery standard, queries were adapted accordingly. The rewritten queries can be found at the GCX download page [8]. Note that the y-axes in Figure 4 scale differently.



(a) Query Q6 **(b) Query Q8**

Figure 4: Buffer Plots for Queries on 10MB XMark Document.

The XMark DTD divides the document into six larger sections, namely *regions*, *categories*, *catgraph*, *people*, *open_auctions*, and *closed_auctions*. Query $Q6$ requires *item* tags beneath *regions*, which are located at the beginning of the document. GCX processes items one at a time and maintains a low main memory consumption of less than 100 buffered nodes. Once the *regions* section has been processed, the buffer becomes almost empty. In contrast, query $Q8$ performs a value-based join between persons (in the *people* section) and closed auctions. In the first three sections, no relevant tokens are encountered. In the *people* section, the first partition of join partners is loaded into the buffer (the first diagonal), followed by a phase where no relevant tokens are matched (the plane). The join partners are found in the *closed_auction* section.

In the buffer plots, both types of queries show a characteristic buffer footprint. Query Q6 can be evaluated in streaming fashion with low memory consumption, while the join query Q8 is inherently blocking, and has a main memory consumption that is linear in the size of the input.

Performance results. We present excerpts from our experiments [8, 17] with our C++ prototype, using queries and data from the XMark benchmark [20]. The queries were adapted as described at [8], to match the XQuery fragment supported by GCX. Our execution platform is a 3GHz CPU Intel Pentium IV with 2GB RAM, running with SuSe Linux 10.0. The Java-based systems were executed using J2RE v1.4.2.

GCX is an in-memory XQuery engine geared towards streaming query evaluation. We chose the following reference systems.

- The FluXQuery engine (in Java) [6,10] is the most natural choice for a reference implementation which was available to us. FluXQuery is also a main-memory XQuery engine geared towards XML stream processing, and it implements a similar XQuery fragment. FluXQuery can exploit schema information, and was provided the XMark DTD in our experiments.

- The in-memory query engines Galax [7] (OCaml), QizX/open v1.1 [15] (Java), and Saxon v8.7.1 [16] (Java) implement full XQuery. 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.

Unfortunately, there are only few implementations of streaming XQuery engines publicly available. This makes it difficult to set up extensive comparative experiments. Acting from this necessity, we further considered experiments with MonetDB [13] v4.12.0 with XQuery v0.12.0, a mature XML database system. As a secondary-storage implementation, MonetDB uses index structures to speed up query evaluation, which is not done by the GCX engine. On the

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 / 111MB	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

Figure 5: GCX Benchmark Results.

other side, MonetDB XQuery stores the entire data physically before query evaluation. To account for the fact that GCX and the other main memory engines read the complete input document for each query evaluation, we forced the MonetDB server to reload the complete document in each run.

Figure 5 shows the behavior of GCX and the other systems in comparison to the reference implementations for several XMark queries. We measure the high watermark of non-swapped memory consumption and the query evaluation time in seconds. With the Java-based engines, we observe that due to automatic memory management and the Java Virtual Machine, memory consumption often increases with the document size even though the amount of data buffered remained constant (e.g. for FluXQuery). Query $Q6$ contains descendant axis XPath expressions which is not supported by FluXQuery (denoted by "n/a"). Failures are denoted by "-".

It is remarkable that GCX evaluates all queries on inputs of up to 200MB, except for $Q8$, which computes an XQuery join and requires a certain amount of buffering, with only 1.2MB main memory consumption. 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, our prototype even outperforms query engines which exploit schema information [10].

4. THE GCX PROJECT ONLINE

GCX is a C++ project. The first version of GCX was released in January 2007 and is freely available as open source under the Berkeley Software Distribution license, to facilitate related research in this area. For current information on the GCX project, see `http://www.infosys.uni-sb.de/projects/streams/gcx/`.

5. 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] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyên. "Type-Based XML Projection". In *Proc. VLDB'06*, pages 271–282, 2006.

[3] S. Bressan, B. Catania, Z. Lacroix, Y. G. Li, and A. Maddalena. "Accelerating Queries by Pruning XML Documents". *TKDE*, 54(2):211–240, 2005.

[4] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. "Query Processing of Streamed XML Data". In *Proc. CIKM'02*, pages 126–133, 2002.

[5] M. Fernández, P. Michiels, J. Siméon, and M. Stark. "XQuery Streaming á la Carte". In *Proc. ICDE'07*, pages 256–265, 2007.

[6] "The FluXQuery Engine", 2004. http://www.infosys.uni-sb.de/projects/streams/fluxquery/.

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

[8] "GCX Download Page", 2007. http://www.infosys.uni-sb.de/projects/streams/gcx/.

[9] C. Koch. "On the complexity of nonrecursive XQuery and functional query languages on complex values". *TODS*, 31(4):1215–1256, 2006.

[10] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. "Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams". In *Proc. VLDB'04*, pages 228–239, 2004.

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

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

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

[14] D. Olteanu. "SPEX: Streamed and Progressive Evaluation of XPath". *TKDE*, 19(7):934–949, 2007.

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

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

[17] M. Schmidt, S. Scherzinger, and C. Koch. "Combined Static and Dynamic Analysis for Effective Buffer Minimization in Streaming XQuery Evaluation". In *Proc. ICDE'07*, pages 236–245, 2007.

[18] H. Su, E. A. Rundensteiner, and M. Mani. "Semantic Query Optimization for XQuery over XML Streams". In *Proc. VLDB'05*, pages 277–288, 2005.

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

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