FluXQuery: An Optimizing XQuery Processor for Streaming XML Data

Christoph Koch* Stefanie Scherzinger† Nicole Schweikardt♮ Bernhard Stegmaier♯

*: Technische Universität Wien, Vienna, Austria, Email: koch@dbai.tuwien.ac.at
†: Technische Universität Wien, Vienna, Austria, Email: scherzinger@wit.tuwien.ac.at
♮: Humboldt Universität zu Berlin, Berlin, Germany, Email: schweikardt@informatik.hu-berlin.de
♯: Technische Universität München, Munich, Germany, Email: bernhard.stegmaier@in.tum.de

1 Introduction and Motivation

XML has established itself as the ubiquitous format for data exchange on the Internet. An imminent development is that of streams of XML data being exchanged and queried. Data management scenarios where XQuery [11] is evaluated on XML streams are becoming increasingly important and realistic, e.g. in e-commerce settings.

Naturally, query engines employed for stream processing are main-memory-based, yet contemporary XQuery engines consume main memory in large multiples of the actual size of the input documents (cf. [10, 8]). This excessive need for buffers has proven to be a serious scalability issue and significant research challenge [10, 9, 5, 3].

So far, the efficient evaluation of XPath on streams has been closely investigated to the point where state-of-the-art techniques use very little main memory [1, 4, 6, 7]. However, corresponding approaches to the effective and economical processing of XQuery on streams are still at a preliminary stage. XQuery, as a *data-transformation* query language, is of an entirely different nature than *node-selecting* XPath. This constitutes the need to develop sophisticated techniques for coping with and reducing main memory buffers during XQuery evaluation.

What is required is a well-principled machinery for processing XQuery which is parsimonious with resources in that it minimizes the amount of buffering necessary. Any such solution should allow for both extensibility and the leverage of a large body of the database community's related earlier work to take effect. Under these considerations, such machinery needs to employ an algebraic view of queries and optimizations.

So far, no principled work exists on algebraic query optimization for *structured data streams* (such as XML, but unlike flat tuple streams, e.g. [2]) which takes into account the special features of stream processing. In particular, we lack an algebra for querying structured data which truly captures the spirit of stream processing and which prepares the ground for optimizing query evaluation using schema information.

In this demonstration, we present the FluXQuery engine as the first optimizing XQuery engine for streams. Optimization in FluXQuery is based on a new internal query language called *FluX* [8] which slightly extends the main structures of XQuery by a construct for event-based query processing. By allowing for the conscious use of main memory buffers, it supports reasoning over the employment of buffers during query evaluation.

2 The FluX Query Language

We consider the following XQuery Q in a bibliography domain, as found among the XML Query Use Cases [12] (XMP Q3):

```
<results>
{ for $b in $ROOT/bib/book return
   <result> { $b/title } { $b/author } </result> }
</results>
```

This query lists the title(s) and authors of each book in the bibliography and groups them inside a "result" element. Note that the XQuery language requires that, within each book, titles are output before all authors. Now the DTD

```
<!ELEMENT bib  (book)*>
<!ELEMENT book (title|author)*>
```

* Work support by project Z29-N04 of the Austrian Science Fund (FWF).

† This research has been partly funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

allows each book node to have several title and several author children, while imposing no order among these items.

In the course of evaluating this query, we may output the title children of a book node as soon as they arrive on the stream, while the output of the author children must be delayed (using a memory buffer) until we reach the closing tag of the book node. Only then we may be sure that no further title nodes will be encountered and we may write the contents of the buffer containing author nodes to the output and then empty it. Later on, we may refill it with the author nodes from the next book.

Consequently, we only need to buffer the author children of one book node at a time, but not the titles. Current main memory query engines do not exploit this fact. Rather, they buffer either the entire book nodes or, as an optimization [10], *all* title and *all* author nodes of each book. Previous frameworks for evaluating or optimizing XQuery do not provide any means of making this seeming subtlety explicit and reasoning about it.

We introduce the FluX query language together with its process-stream construct which allows us to express precisely the mode of query execution just described. Given the DTD from above, XQuery Q may be phrased as a FluX query as follows:

```
<results>
{ process-stream $ROOT: on bib as $bib return
  { process-stream $bib: on book as $book return
     <result>
     { process-stream $book:
        on title as $t return {$t};
        on-first past(title,author) return
         { for $a in $book/author return {$a} } }
     </result> } }
</results>
```

A process-stream $x expression consists of a number of *handlers* which process the children of the XML tree node bound by variable $x from left to right. An "on a" handler fires on each child labeled "a" visited during such a traversal, executing the associated query expression. In the process-stream $book expression above, the on-first past(title,author) handler fires exactly once, namely as soon as the DTD implies for the first time that no further author or title node can be encountered among the children of $book. (As observed above, in the given, very weak DTD, this is the case only as soon as the last child of $book has been seen.) In the query associated with the on-first past(title,author) handler, we may safely use paths of the form $book/author or $book/title, because such paths cannot be encountered anymore. Consequently, we may assume that the query engine has buffered all matches of that path for us. It is a feasible task for the query engine to buffer only those paths that the query actually employs (see also [10]).

```
<!ELEMENT bib  (book)*>
<!ELEMENT book (title,(author+|editor+),
                                publisher,price)>
```

Figure 1: A DTD.

We call a FluX query *safe* for a given DTD if, informally, it is guaranteed that XQuery subexpressions (such as the for-loop in the query above) do not refer to paths that may still be encountered in the stream. The above FluX query is safe: The for-expression employs the $book/author path, but is part of an on-first handler that cannot fire before all author nodes relative to $book have been seen.

If the path $book/author in the previous FluX query was replaced by, say, $book/price and the DTD production for book were

```
<!ELEMENT book ((title|author)*,price)>
```

then the FluX query such modified would not be safe: On the firing of on-first past(title,author), the buffer for $book/price items would still be empty and the query result would be incorrect.

Let us now return to XQuery Q. This query can be processed more efficiently with the DTD shown in Figure 1: Here, no buffering is required to execute query Q because the DTD asserts that for each book, the title occurs strictly before the authors (we call this an *order constraint*).

Thus, we may phrase our query in FluX so as to directly copy titles and authors to the output as they arrive on the input stream:

```
<results>
{ process-stream $ROOT: on bib as $bib return
   { process-stream $bib: on book as $book return
      <result>
      { process-stream $book:
         on title  as $t return {$t};
         on author as $a return {$a} }
      </result> } }
</results>
```

3 FluXQuery System Architecture

FluXQuery is, to our knowledge, the first XQuery engine that optimizes query evaluation using schema constraints derived from DTDs[1]. Query optimization is carried out on an algebraic, query-language level (rather than, say, on some form of derived automata). Thus, a main strength of FluXQuery is its extensibility and the ability to benefit from a large body of previous database research on algebraic query optimization.

The main focus of our efforts was to develop a system for automatically rewriting XQueries into FluX queries and thereby optimizing (reducing) the use of main memory buffers. We have developed an algebra for optimizing XQuery on streams using a DTD

[1]Note that the static information required for optimization could just as well be derived from XML Schema.

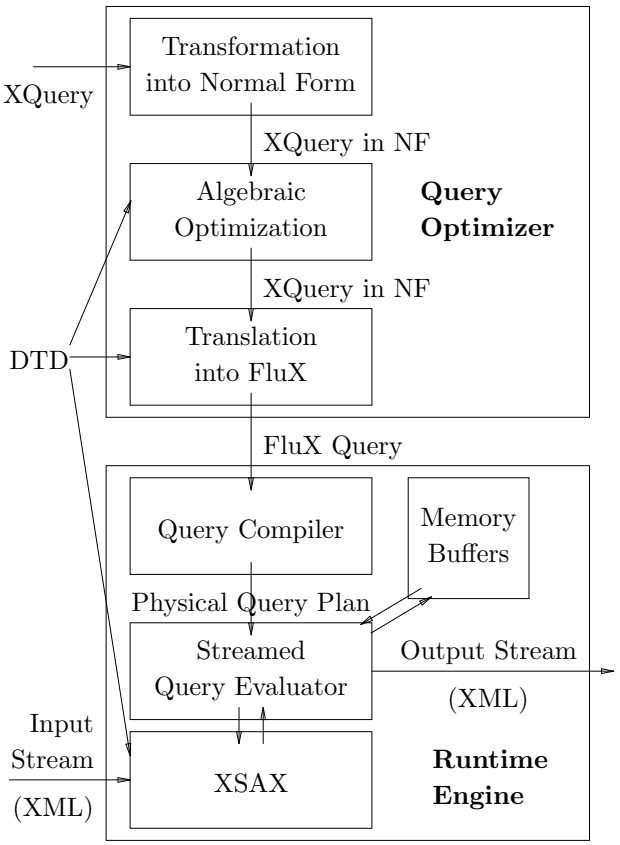

Figure 2: The FluXQuery system architecture.

and an efficient algorithm for DTD-aware scheduling of XQueries as FluX queries [8].

We next discuss the system architecture of FluXQuery, which consists of the *query optimizer* and the *runtime engine*, as depicted in Figure 2.

3.1 The Query Optimizer

Our query optimizer translates user queries written in XQuery into optimized FluX queries. The translation and optimization proceeds in three steps.

First, XQueries are rewritten into a *normal form* which allows us to use a simple set of equivalences as rewrite rules in the subsequent optimization steps.

Next, we statically optimize the normalized XQuery, exploiting schema information gained from the DTD. More precisely, we employ *algebraic optimizations* that are based on *cardinality constraints* and *language constraints* derived from the DTD. As a result, we may generate FluX queries which can be evaluated more efficiently on data streams conforming to the given schema.

For the intuition behind cardinality constraints, consider the following query expression with two subsequent for-loops

$$\{\ \texttt{for \$x in \$book/publisher return}\ \alpha\ \}$$
$$\{\ \texttt{for \$x in \$book/publisher return}\ \beta\ \}$$

where α and β are arbitrary subexpressions. In order to perform two iterations over the same set of `publisher` nodes, we are automatically forced to buffer all such nodes.

However, if the schema states that a `book` node has at most one `publisher` among its children, as does the DTD of Figure 1, then we denote this cardinality constraint by $\texttt{publisher} \in ||^{\leq 1}_{\texttt{book}}$.

Application of the algebraic optimization rule

$$\frac{\{\ \texttt{for \$}x\ \texttt{in \$}r/a\ \texttt{return}\ \alpha\ \}\quad \{\ \texttt{for \$}x\ \texttt{in \$}r/a\ \texttt{return}\ \beta\ \}}{\{\ \texttt{for \$}x\ \texttt{in \$}r/a\ \texttt{return}\ \alpha\ \beta\ \}}\quad \left(a \in ||^{\leq 1}_{\$r}\right)$$

merges both for-loops into a single and equivalent for-loop:

$$\{\ \texttt{for \$x in \$book/publisher return}\ \alpha\ \beta\ \}.$$

Clearly, the second query is preferable, as it requires only one loop over publishers instead of two subsequent iterations. Depending on the nature of subqueries α and β, we may even be able to evaluate the query expression completely on-the-fly.

Based on language constraints derived from the DTD, we can also eliminate unsatisfiable conditional subexpressions. Again, consider the DTD of Figure 1. Then we may eliminate an expression

```
if $book/author = "Goedel"
and $book/editor = "Goedel" then α
```

where α is an arbitrary subexpression, since the DTD does not permit `book` elements with both `author` and `editor` children.

Finally, the pre-optimized XQuery is rewritten into FluX, with `process-stream` extensions (as briefly described in Section 2) enabling a streaming execution of the query. The key idea here is to exploit *order constraints* defined by the DTD. For instance, the DTD of Figure 1 ensures that all `title` elements precede all `author` elements. The rewriting process schedules the execution of query subexpressions with respect to order constraints and therewith generates FluX queries with reduced buffer consumption.

In contrast to existing techniques, our algebraic optimizations aim at minimizing the size of main memory buffers, rather than the execution speed.

A main strength of our approach is its extensibility, and even though our system is currently restricted to a (powerful) *fragment* of XQuery with nested loops and joins, our approach can be generalized to larger XQuery fragments.

3.2 The Runtime Engine

The second part of our FluXQuery system architecture is the runtime engine. It evaluates FluX queries as obtained from XQueries by the query optimizer. The runtime engine is organized as follows.

The *query compiler* transforms an optimized FluX query into a physical query plan. It first computes the *buffer description forest* data structure, BDF for short, which defines those paths of the input document which need to be buffered. Based on the BDF, it schedules query operators, such as the execution of `process-stream` expressions, the streamed execution of for-where-return-statements, and buffer population. Our approach improves on that of [10] in that it allows us to avoid the buffering of the data which can be processed on the fly.

The resulting query can either be compiled into an internal representation, which is interpreted during execution, or directly into executable JAVA code.

Finally, the physical query plan is executed by the *streamed query evaluator*. The latter uses our validating SAX parser, *XSAX*, which is an extension of a standard SAX parser that in addition produces on-first events in addition to customary SAX-events (such as on-begin-element).

Basically, the XSAX parser works as follows. We first register the DTD and all on-first event handlers of the input query with the XSAX parser. Based on this information, the XSAX parser builds a finite state automaton and lookup-tables for validating the input and generating on-first events. While reading the input XML stream, the state of this automaton is checked and the on-first events are properly inserted among the generated stream of conventional SAX events. The streamed query evaluator processes these events and delivers its output in turn as an XML stream.

4 Conclusions

FluXQuery currently supports a fairly powerful XQuery fragment with arbitrarily nested for-loops and joins, but does not yet cover aggregation.

The FluX query language, an algorithm for rewriting XQuery into FluX and thereby scheduling event processors using the DTD, as well as buffer management, are described in detail in [8]. There, the efficiency of our system is also benchmarked against two other XQuery engines. Our experiments show that FluXQuery consumes both far less memory and runtime than other XQuery systems. The difference is particularly clear for main memory consumption, which is of great importance in stream processing and vital to scalability.

Our algebraic optimization techniques will be described in detail in a forthcoming paper.

References

[1] M. Altinel and M. Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In *Proc. VLDB 2000*, pages 53–64, Cairo, Egypt, 2000.

[2] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. "STREAM: The Stanford Stream Data Manager". In *Proc. SIGMOD 2003*, page 665, 2003.

[3] P. Buneman, M. Grohe, and C. Koch. "Path Queries on Compressed XML". In *Proc. VLDB 2003*, pages 141–152, 2003.

[4] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. "Efficient Filtering of XML Documents with XPath Expressions". In *Proc. ICDE 2002*, San Jose, California, USA, February 26–March 1 2002.

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

[6] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. ICDT'03*, 2003.

[7] A. K. Gupta and D. Suciu. "Stream Processing of XPath Queries with Predicates". In *SIGMOD Conference*, pages 419–430, 2003.

[8] 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 2004*, 2004.

[9] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. "A Transducer-Based XML Query Processor". In *Proc. VLDB 2002*, pages 227–238, 2002.

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

[11] World Wide Web Consortium. "XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002), 2002. http://www.w3.org/TR/query-algebra/.

[12] "XML Query Use Cases. W3C Working Draft 02 May 2003", 2003. http://www.w3.org/TR/xmlquery-use-cases/.