



UNIVERSITÄT
DES
SAARLANDES

DIPLOMARBEIT

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

ausgeführt an der

Universität des Saarlandes

– Lehrstuhl für Informationssysteme –

unter Anleitung von

Professor Christoph Koch

durch

Michael Schmidt

Scheibenweg 21
66557 Illingen

Beginn der Diplomarbeit: 01.02.2006

Abgabe der Diplomarbeit: 27.07.2006

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die von mir eingereichte Diplomarbeit bzw. die von mir namentlich gekennzeichneten Teile selbständig verfasst und ausschließlich die angegebenen Hilfsmittel benutzt habe.

Saarbrücken, 27. Juli 2006

(Unterschrift)

Acknowledgements

My special thanks go to professor *Christoph Koch* and *Stefanie Scherzinger*. In particular, I want to thank professor *Christoph Koch* for having chosed the exciting topic, for excellent supervision, many discussions and all the time he spent in supporting my diploma thesis. *Stefanie Scherzinger* contributed to this thesis with many valuable ideas, background information, critics and support wherever possible. All these contributions significantly improved the quality of this thesis.

My thanks also go to my family, in particular to my father *Gerhard Schmidt* and to my mother *Martina Schmidt*. They enabled my studies and supported me throughout all those years.

Furthermore I want to thank all people that supported me throughout my studies and made contributions to this thesis, especially *Eric Haßdenteufel*, *Marie Leinen*, *Christoph Pinkel*, *Stefan Schmidt* and *Markus Thiele* for proof-reading and all the members of the Saarland University Database Group for discussions and suggestions.

Last but not least my thanks go to professor *Gerhard Weikum* who agreed to revise this thesis.

Danksagungen

Mein besonderer Dank gilt Herrn Professor *Christoph Koch* und Frau *Stefanie Scherzinger*. Ich danke Professor *Christoph Koch* für die Wahl des überaus interessanten Themas, für exzellente Betreuung, die zahlreichen Diskussionen und all die Zeit, die er in meine Diplomarbeit investiert hat. *Stefanie Scherzinger* hat mit vielen wertvollen Ideen, Hintergrundinformationen, Kritik und sonstiger Unterstützung wesentlich zum Gelingen der Arbeit beigetragen. All diese Beiträge haben die Qualität der Arbeit deutlich verbessert.

Mein Dank gilt ebenfalls meiner Familie, insbesondere meinem Vater *Gerhard Schmidt* und meiner Mutter *Martina Schmidt*. Durch Ihre Unterstützung und ihr Vertrauen haben sie mein Studium ermöglicht und mir den nötigen Rückhalt während all der Jahre gegeben.

Weiterhin danke ich allen Personen, die mir in der Zeit meines Studiums und der Diplomarbeit unterstützend zur Seite gestanden haben, insbesondere *Eric Haßdenteufel*, *Marie Leinen*, *Christoph Pinkel*, *Stefan Schmidt* und *Markus Thiele* für das Korrekturlesen der Diplomarbeit und allen Mitgliedern der Saarland University Database Group für aufschlussreiche Diskussionen und Anregungen.

Nicht zuletzt gilt mein Dank Herrn Professor *Gerhard Weikum* für die Bereitschaft, die Zweitkorrektur meiner Diplomarbeit zu übernehmen.

Abstract

Coming along with the proliferation of the internet and increasing bandwidths, the relevance of data stream processing has rapidly increased in recent years. Along the way, XML has evolved into a de facto standard format for data interchange. In the meantime, many applications have been developed that follow both trends as they have to deal with XML data streams. The efficient extraction of data from XML streams has become a non-trivial challenge, as streaming data may arrive at very high rates.

In consequence for the growing importance, different languages have been proposed to query data from XML documents, among them XQuery and XPath. Efficient evaluation of XQuery, which is more powerful than XPath, has been subject to extensive studies throughout the years. Native XML database management systems, which rely on a physical database, are ill-suited for data stream processing: Storing streamed data on physical volumes before query evaluation is infeasible as the amount of incoming data is very high and, moreover, hard disk access is slow compared to in-memory data manipulation. Systems that evaluate data in main memory, in practice have to deal with a strictly limited main memory size. For large data streams, effective buffer management becomes *the* key prerequisite to performance.

All existing approaches to reducing the size of in-memory buffers rely on static query analysis. We propose a buffer management scheme which combines static and dynamic analysis to keep main memory consumption low. Our system aims at both minimizing the high watermark of memory consumption and keeping the average memory consumption as low as possible. The approach relies on a technique that we call active garbage collection and which actively purges buffers at runtime based on the progress made in query evaluation. In order to implement active garbage collection we assign roles to buffered nodes. Roles express the future relevance of buffered nodes for query evaluation. During query evaluation, nodes incrementally lose their roles as they fulfill them respectively. A technique very similar to garbage collection by reference counting ensures that tokens that have lost all their roles are purged from the buffer early on.

In a prototype system, which was designed for a practical fragment of XQuery, we employ 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.

Abstract

Einhergehend mit der Verbreitung des Internets und sukzessiv ansteigenden Datenübertragungsraten hat die Verarbeitung von Datenströmen einen hohen Stellenwert eingenommen. Parallel dazu hat sich XML zu einem Standardformat für den Datenaustausch etabliert. In der Zwischenzeit wurden diverse praxisbezogene Anwendungen entwickelt, die auf XML-Strömen arbeiten, und somit beiden Trends folgen. Die effiziente Ausführung von Anfragen auf XML-Datenströmen ist, nicht zuletzt aufgrund schnell steigender Übertragungsraten, zu einem wichtigen Gebiet der Forschung geworden.

Aufgrund des verstärkten Einsatzes von XML wurden diverse Anfragesprachen zur Datenextraktion in XML-Szenarien entwickelt, wobei XQuery und XPath mittlerweile weitgehend verbreitet und anerkannt sind. Während ein Großteil der Forschung auf Techniken zur effizienten Ausführung dieser Anfragesprachen im Kontext nativer XML Datenbank Management Systeme fokussiert war, in denen die Daten vor der Verarbeitung physikalisch in einer Datenbank abgespeichert werden, bestehen für die Auswertung auf Datenströmen grundlegend andere Anforderungen. Hohe Übertragungsraten erfordern oftmals die Verarbeitung riesiger Datenmengen. Das Zwischenspeichern der Daten auf physikalischen Medien ist in solchen Szenarien nahezu unmöglich, zumal Datenträgerzugriffe um ein Vielfaches langsamer sind als die direkte Verarbeitung im Hauptspeicher. Systeme, die Daten ohne physikalisches Zwischenspeichern verarbeiten, müssen mit einem – in der Praxis stark limitierten – Hauptspeicher auskommen. Die Effizienz dieser Systeme ist eng an eine effektive Speicherverwaltung gekoppelt.

Alle existierenden Ansätze zur Speicherverwaltung in solchen Szenarien basieren auf rein statischer Anfragen-Analyse. Unser Ansatz hingegen verbindet statische und dynamische Analyse. Basierend auf dem derzeitigen Fortschritt der Anfrageausführung werden nicht mehr benötigte XML-Knoten frühzeitig aus dem Speicher entfernt. Um dieses Ziel zu erreichen werden gepufferten XML-Knoten sogenannte Rollen zugeordnet. Dabei stellt jede zugeordnete Rolle eine Funktion des Knotens für den noch ausstehenden Teil der Auswertung dar. Während der Ausführung, die wechselseitig zum Einlesen neuer Knoten vonstatten geht, verlieren Knoten ihre Rollen. Wird die letzte Rolle eines gepufferten Knotens entfernt, so kann er, vergleichbar mit Garbage Collection by Reference Counting, unmittelbar aus dem Puffer entfernt werden.

Wir haben unser Speicherminimierungsmodell in einem Prototypen für ein praktisches XQuery-Fragment realisiert. Der experimentelle Vergleich mit anderen Systemen belegt die signifikante Verbesserung, die durch die Kombination von statischer und dynamischer Analyse erzielt werden kann.

Contents

1	Introduction	1
1.1	Buffer Design in XQuery Engines	1
1.2	Garbage Collection in XQuery Engines	2
1.3	Contributions	8
1.4	Chapter Overview	8
1.5	Related Work	9
1.5.1	XQuery Engine Categorization	9
1.5.2	Streaming XQuery Evaluation	9
1.5.3	Stream Preprojection	11
1.5.4	Garbage Collection	13
2	Preliminaries	15
2.1	Document Projection	16
2.1.1	Projection Paths	16
2.1.2	Projection Trees	18
2.1.3	Minimal Projection	19
2.2	Role Association	20
3	Query language	21
3.1	Semantics of XQ	22
3.2	Introducing <i>signOff</i> -Statements to XQ	23
3.3	Roles and Dependency Paths	25
4	Static Analysis	28
4.1	Deriving Projection Trees	28
4.2	Assigning Roles to Buffered Nodes	30
4.3	XQ Rewriting	31
4.3.1	Query Normalization	31
4.3.2	Inserting <i>signOff</i> -Statements	34
5	Active Garbage Collection	37
5.1	Irrelevant Nodes	37
5.2	Buffer Cleanup Algorithm	37
6	System Implementation	39
6.1	System Architecture	39
6.2	System Components and Runtime Interaction	40
6.3	Buffer Representation	41
6.4	Projection Tree Construction	42

6.4.1	Projection Trees and Projection Paths	42
6.4.2	Base Projection Path Extraction	43
6.4.3	Base Projection Path Conversion	47
6.4.4	Projection Tree Setup Algorithm	50
6.5	Projection Tree Determinization	51
6.5.1	Projection Tree Fusion	54
6.5.2	Role Aggregation	55
6.5.3	Lazy DFA Construction	56
6.6	Syntax and Expression Implementation	59
6.7	Optimizations	60
6.7.1	Elimination of Redundant Roles	60
6.7.2	Tag Name Hashing	61
6.7.3	Pushing Down <i>signOff</i> -Statements	61
7	Experimental Results	62
7.1	Correctness Tests	62
7.2	Stream Preprojection	62
7.3	Evaluation Time and Memory Consumption	67
8	Future Work	70
8.1	Standard Database Techniques	70
8.2	Exploiting Schema Knowledge	70
8.2.1	Reducing Memory Consumption	71
8.2.2	Accelerating Query Evaluation	72
8.3	Incorporating Aggregation	73
8.4	Role Representation	74
9	Conclusion	75
A	XQ Concrete Syntax	76
B	UML I - Conditional Expressions	78
C	UML II - Non-Conditional Expressions	79
D	Queries Used for Correctness Tests	80
D.1	W3C “XMP” XQuery Usecases	80
D.2	DBMS Milestone Test Queries	82
D.3	User Defined Queries	85
E	XMark Test Queries	88

List of Figures

1	Projection tree.	3
2	Active garbage collection.	6
3	Document projection.	16
4	Projection path evaluation.	18
5	XQuery fragment XQ.	22
6	Projection and role assignment.	25
7	Projection tree of Example 9.	29
8	Semantics of role assignment.	30
9	Decomposition of if-statements.	31
10	Pushing down if-statements.	32
11	Static query rewriting.	35
12	Inserting <i>signOff</i> -statements.	36
13	Localized active garbage collection.	38
14	System architecture.	39
15	System interaction.	40
16	Inference rules for non-conditional expressions.	45
17	Inference rules for conditional expressions.	46
18	Projection tree setup algorithm.	51
19	Projection tree setup example.	52
20	Deterministic projection tree setup.	53
21	Deterministic projection tree.	58
22	Stream preprojection test queries.	64
23	XML stream used for preprojection tests.	65
24	DTD <i>D</i>	71

List of Tables

1	Stream preprojection results.	66
2	Benchmark results.	68

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 [5, 11, 23, 27, 42] 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.

1.1 Buffer Design in XQuery Engines

We claim that there are three desiderata for good buffer design: 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.

Strictly speaking, these goals are unattainable. For instance, a system optimal for desiderata one would have to be able to check satisfiability of XQuery expressions, an undecidable problem (this is implicit e.g. in [3]).

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 [5, 11, 23, 24, 34]. 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 [27], and refined in [5, 6], where only the query-relevant parts of the input 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 [23, 24]. 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 [23], if available. However, for many practical queries involving blocking operators or descendant axes and wildcards, little can be evaluated on the fly [2, 10, 23, 24].

In the above systems, the decision when buffers are purged is made at compile time. In the case of the FluXQuery engine [23] and similarly in [24], 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 loading only the “query-relevant” data into buffers [27]. In addition, we can statically infer the moments during query evaluation when buffered nodes may 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, a *dynamic analysis* is required which takes into account the current contents of buffers, 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 achieved by static analysis alone.

1.2 Garbage Collection in XQuery Engines

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

Garbage collection [44] 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 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 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 average main memory consumption remain low.

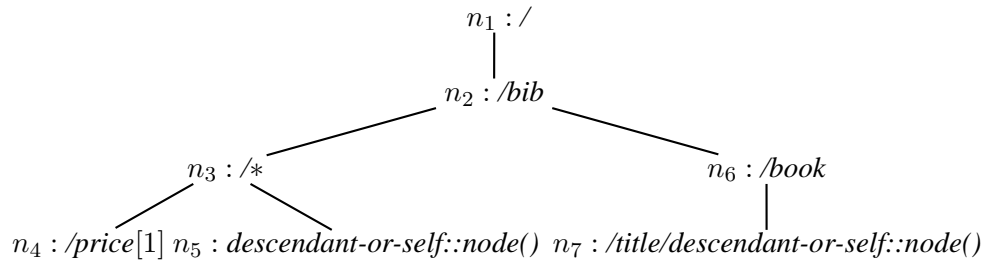


Figure 1: Projection tree.

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 then is 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 are assigned any roles.

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.

```

<result> {
  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
  )
} </result>

```

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, \dots, n_7 shown in Figure 1. The projection tree defines the parts of the input that are copied into the buffer. Consider for example node n_4 , which refers to the if-condition in the query. This projection tree node defines that, for each child of the bib node, only the first price tag – without

descendants – needs to be buffered. Buffering the first occurrence is enough, as the price node is used for an existence check, which evaluates to *true* if one or more occurrences can be found. However, due to projection tree node n_5 , we are forced to buffer all children of the bib node as well as their subtrees. Intuitively spoken, node n_5 subsumes node n_3 . Moreover, we emphasize that the price tag carries two different roles during query evaluation: it is used in an output expression as well as in an existence check expression.

Each projection tree node n_i is assigned the unique role r_i . When the input stream is parsed at runtime, the nodes of the document will be incrementally projected into the buffer and marked with roles on-the-fly. As mentioned before, nodes describe the future relevance of a node. A matching price tag for example will be marked with two roles, namely role r_3 capturing its relevance for the existence check and role r_5 for the corresponding output expression.

Before evaluation, *signOff*-statements are statically inserted into the query. At runtime, these statements notify the buffer manager that certain nodes lose their roles. Each *signOff*-statement refers to one or more nodes and exactly one role. The rewritten query is sketched below.

```
<result> {
  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)
} </result>
```

The query is sequentially evaluated on the buffer until input is required that has not (yet) been buffered. In the latter 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 [27], 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 streamed input document

$$\langle bib \rangle \langle book \rangle \langle title \rangle \langle author \rangle \langle /book \rangle \langle /bib \rangle$$

, which contains a single book with empty title and author tags. Figure 2 shows for several steps what has been read from the input stream, the current buffer contents, and the output produced so far.

Step 1. In the first step the opening tag $\langle result \rangle$ 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.

Step 2. Next $\langle bib \rangle$ 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.

Step 3. $\langle book \rangle$ is matched by the projection tree nodes n_3 , n_5 and n_6 , and hence is buffered and assigned the roles r_3 , r_5 , and r_6 . Now variable $\$x$ can be 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 further input.

Step 4. The opening- and closing tags of the title node are read. The tokens are matched by the projection tree, and consequently buffered and annotated with roles r_5 and r_7 . The evaluation of the if-expression blocks again, also after reading the author node.

Step 5. The opening- and closing tags of the author node are read. As they are matched by projection tree node n_5 , they are annotated with role r_5 and written to the buffer.

step	input stream	buffer contents	output stream
1			$\langle result \rangle$
2	$\langle bib \rangle$	$bib\{r_2\}$	
3	$\langle book \rangle$	$bib\{r_2\}$ $ $ $book\{r_3, r_5, r_6\}$	
4	$\langle title \rangle$	$bib\{r_2\}$ $ $ $book\{r_3, r_5, r_6\}$ $ $ $title\{r_5, r_7\}$	
5	$\langle author \rangle$	$bib\{r_2\}$ $ $ $book\{r_3, r_5, r_6\}$ $/ \quad \backslash$ $title\{r_5, r_7\} \quad author\{r_5\}$	
6	$\langle /book \rangle$	$bib\{r_2\}$ $ $ $book\{r_3, r_5, r_6\}$ $/ \quad \backslash$ $title\{r_5, r_7\} \quad author\{r_5\}$	$\langle book \rangle$ $\langle title \rangle$ $\langle author \rangle$ $\langle /book \rangle$
7		$bib\{r_2\}$ $ $ $book\{r_6\}$ $ $ $title\{r_7\}$	
8	$\langle /bib \rangle$	$bib\{r_2\}$ $ $ $book\{r_6\}$ $ $ $title\{r_7\}$	$\langle title \rangle$
9		$bib\{r_2\}$ $ $ $book\{r_6\}$ $ $ $title\{r_7\}$	
10		$bib\{r_2\}$	$\langle /result \rangle$

Figure 2: Active garbage collection.

Step 6. Tag $\langle/book\rangle$ 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.

- “*signOff*($\$x, r_3$)” causes the buffered book node to lose role r_3 .
- “*signOff*($\$x/price[1], r_4$)” takes no effect as there is no match for pathstep “price[1]” relative to the current binding of $\$x$.
- “*signOff*($\$x/descendant-or-self::node(), r_5$)” removes r_5 from all matching nodes, namely the book, the title and the author.

As the author node has lost its single role r_5 in the course of evaluating the *signOff*-statements and, as it has no descendants, it can be purged from the buffer.

Step 7. 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.

Step 8. The token $\langle/bib\rangle$ completes the buffer. As there is no second binding for variable $\$x$, evaluation proceeds with $for_{\$b}$. Variable $\$b$ is bound to the book which still is in buffer and the title is output.

Step 9. First, command “*signOff*($\$b, r_6$)” is executed and causes the book to lose its single role r_6 . For the time being, the book resides in buffer as there still exists the marked title descendant. After execution of “*signOff*($\$b/title/descendant-or-self::node(), r_7$)” the title itself loses its single role. Now both the book and the title can be removed from the buffer.

Step 10. Execution of “*signOff*($\$bib, r_2$)” removes role r_2 from the buffered bib tag. The buffer becomes empty before, finally, $\langle/result\rangle$ is written out.

Query evaluation finishes with Step 10. As all nodes lost their roles during evaluation, the buffer finally has become empty.

1.3 Contributions

- We propose 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 [5, 27] so that roles are assigned to document nodes on-the-fly during projection. Moreover, we present optimizations of the stream preprojection techniques presented in the latter works.
- 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 [22].
- 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 [23].

1.4 Chapter Overview

We provide the preliminaries in Section 2. Beyond a formal definition of XML we cover the basic principles of stream preprojection, roles and role assignment. Our XQuery fragment XQ is introduced in Section 3. We present an abstract syntax for this fragment and discuss semantics, properties and expressiveness of XQ, before we formally extend our language by *signOff*-statements. Static analysis is presented in Section 4 and forms the groundwork for active garbage collection at runtime. After taking up the task of document projection from the preliminaries section, we address the topics of role extraction and query rewriting, i.e. inserting *signOff*-statements at the right positions in the query expression. Active garbage collection is discussed in Section 5. We describe the system runtime behaviour and identify rules for purging buffers at runtime via garbage collection. Section 6 focuses on the efficient implementation of our buffer management scheme. While the previous chapters describe the ideas from an abstract and mathematical point of view, the emphasis of this chapter lies on optimizations, algorithms and implementation techniques. The experimental results are discussed in Section 7. Beyond correctness tests used for the experimental validation of our prototype, we finally present benchmark results for both stream preprojection and query evaluation. The thesis concludes with some ideas on future work in Section 8 and a short conclusion (Section 9) that summarizes the ideas of our new buffer management schemes.

1.5 Related Work

In recent years, efficient XQuery [49] evaluation has been subject to extensive studies. Works cover both XQuery evaluation on physical databases and its evaluation on streaming data. In the following we discuss related work with focus on streaming scenarios.

1.5.1 XQuery Engine Categorization

XQuery evaluation engines can be categorized into three classes. The first one is the class of *native XML database management engines* [18, 19, 28, 43, 52]. These systems are characterized by physically stored XML data and, in general, query evaluation with constant memory usage. One focus of these systems lies on the efficient storage of XML data. Hard disk access is very expensive, thus appropriate indices and data structures for XML ensure that relevant data can be accessed efficiently. Query evaluation in many cases requires techniques very similar to those used in relational database management systems, e.g. advanced join techniques, join reordering, indexing, query evaluation plans and pipelined evaluation [38]. The challenge of a native XMLDBMS is the application of these techniques on structured, ordered and hierarchical data rather than flat relational data as given in a relational DBMS.

The Natix system [18] has been developed with emphasis on the efficient management of tree-structured data on the level of page allocation and physical placement. The system compiles XQuery expressions rather than interpreting them. Another main focus of the Natix system lies on order preserving join reordering and the appropriate choice of well-suited join algorithms.

The TIMBER system [19] basically addresses the same issues. While reusing standard techniques from relational database management systems like B-tree and value-based hash indexing [38], a carefully designed tree algebra is introduced that forms a mathematical foundation for the optimization of XQuery expressions.

Beyond native XML database management systems, there also exist *in-memory XML engines* [14, 37, 41]. While XML data is stored physically on hard disk, during query evaluation relevant parts of the data are loaded and processed in main memory. These engines share an important property with the third class of XQuery engines, namely *streaming XQuery engines*: All data are processed in-memory and, as main memory resources are limited in practice, a strong focus for both types of engines lies on buffer minimization.

1.5.2 Streaming XQuery Evaluation

Previous work on evaluation of XQuery and XPath [48] over streaming data includes [7, 12, 13, 17, 24–26, 30–35]. Due to increasing bandwidths and an increasing number of world wide connections, data streams in such scenarios may arrive at a very high rate [25].

In these high-traffic scenarios, storing data physically on hard disk is unfeasible, because streams are too large to be stored on the whole and, moreover, hard disk access is very slow compared to main memory data manipulation. As main memory resources are strongly limited in practice, it is desirable to decrease the amount of buffering. Once resources are exhausted, systems start swapping and, in most cases, query evaluation time becomes unacceptable.

One buffer minimizing technique is static stream preprojection [5, 6, 27, 36]. In projecting away non-relevant parts from the input stream before query evaluation, the amount of buffered data can be significantly reduced. We refer to the following Subsection 1.5.3 for a discussion of work related on stream preprojection.

Beyond stream preprojection, the single-pass query evaluation over XML documents has been the subject of various works. While this desideratum plays a minor role in native XML database systems, the topic becomes even more important for streaming scenarios, where the input document is delivered token by token. A pure single-pass evaluation in stream processing means that nothing has to be buffered at all; in principle everything can be evaluated on-the-fly. Unfortunately, in many practical cases a certain amount of buffering is inevitable [2], e.g. when computing joins over subtrees. We now discuss the strategies of existing systems designed for streaming XQuery evaluation.

The FluXQuery system [13] exploits situations where data can be evaluated on-the-fly. A static analysis of query and DTD identifies which parts of the input stream can be evaluated without any buffering. The query language is extended and the query is rewritten to exploit these situations. While the system behaves very well for many practical queries, it still has some drawbacks. In particular, the FluXQuery system does not support queries with descendant axes and, in some cases, buffers parts of the input stream multiple times, i.e. when dealing with wildcard and descendant axes.

With XML stream machines [25] on the other hand, buffer management becomes very explicit, even at the low level of reading tokens from and writing tokens to the buffer. In a first step, query subexpressions are translated into XML stream machines (XSMs). The resulting stream machines are composed to an XSM network which can further be optimized using DTD information. The XSM network finally is compiled into a *C* program that evaluates the specific query. Even here, parts of the input stream might be buffered multiple times, moreover the system only supports acyclic DTDs and the XQuery fragment is restricted to descendant axes.

A streaming system that offers support for aggregations and user-defined functions is presented in [24]. The system is similar to the FluXQuery engine insofar as it aims at on-the-fly evaluation wherever possible. First a so-called “stream data flow graph” is constructed which reflects the relations between the variables used in the underlying XQuery expression. The decision whether the query can be evaluated on the fly can be made by analyzing the stream data flow graph. The optimized query is finally compiled and executed on the input stream. Unfortunately, the details of buffer management are not

discussed in this work.

In [10], pipelining is proposed as a technique that can help to reduce the amount of cached data. The system particularly was designed to evaluate simple queries that produce an output far smaller than the input stream size. The key idea is to compile XQuery subexpressions into so-called iterators, which are connected through pipelines. The stream finally is pushed through the iterator network. While simple queries can be evaluated very efficiently, the evaluation of complex queries using XQuery joins and backward axes is less efficient as it requires stream duplication in many cases.

The BEA/XQRL Streaming XQuery Processor [12] is a commercial product which is fully compliant to the XQuery standard. In analogy to many native XML database systems, in a first step the query is optimized based on heuristics and cost estimations. The optimization phase includes standard database techniques like join optimizations, but also techniques that primarily apply to streaming scenarios, for example rewriting of XQuery subexpressions that require input stream reverse traversals. The query finally is compiled and executed on the stream, while all intermediate-results are stored in-memory.

None of the systems described above covers all three desiderata for XQuery buffer management postulated in the introduction. Although most systems aim at on-the-fly evaluation for query evaluation, there all use purely statical techniques like stream preprojection, query rewriting and query compilation. Dynamic buffer management at runtime, which takes under consideration the current state of query processing, is not a topic.

1.5.3 Stream Preprojection

In stream preprojection [5, 6, 27, 36], non-relevant parts of the input document are projected away before query evaluation. The query then is evaluated on the projected document. Projection can be realized by deriving a set of paths, called *projection paths*, from an XQuery expression. Every document path that does not match at least one of these paths can be discarded, as it will never be accessed during query evaluation and, for this reason, does not change the evaluation result. Stream preprojection is crucial for XQuery evaluation against streaming data as it can significantly reduce the amount of buffered data and, coming along with this optimization, decreases main memory usage.

Stream preprojection for XQuery has been proposed in [27] for the first time. The ideas of the latter works have also been implemented in the Galax system [14]. It is shown that this technique can be used to significantly reduce the size of the input document. Applying the technique to the XMark use case queries [53] for example, in most cases resulted in a document size reduction of more than 95%.

An extension of the stream preprojection technique presented above is discussed in [36]. While the Galax approach fails to analyze navigations on constructed elements, i.e. let expressions are not covered by their analysis, this work extends the rules accordingly to cover these constructs as well.

Another approach to XML document projection, called type-based projection, is presented in [5]. Projection is extended to support backward XPath axes and, moreover, optimizations for the fast processing of descendant axes are proposed. The drawback of the technique is that it requires external schema knowledge (e.g. a DTD or XML Schema).

Approaches [27, 36] focus on disk-resident XML data. A loading algorithm decides whether tokens can be projected away or have to be buffered, but not until the closing tag of a token has been read. This approach is insufficient for streaming scenarios, where we aim at discarding tokens early on, i.e. as soon as the opening tag has been read. In [36], no loading algorithm is presented at all. The question whether the technique is applicable to streaming scenarios remains unanswered. Although the preprojection technique from [5] consists of a “single bufferless one-pass traversal of the input document”, implementation details are not presented, so the efficient integration of this approach is questionable.

In contrast, our approach was designed for streaming scenarios. A one-pass traversal algorithm is presented that ensures that tokens can be projected on-the-fly. The decision whether a token can be discarded is made early on when reading the opening tag. Furthermore, compared to [27], we present an optimization that exploits semantic properties of the XQuery expression, i.e. for existence check paths we only buffer the first observation of the token required for the condition check. Moreover, we address the topic of descendant axes and propose an improvement that projects away more tokens than the Galax approach. We refer to Section 7 for an experimental comparison of the stream preprojection techniques. The algorithms for stream preprojection will be provided in Section 2 and 4.

Implementation techniques The task of stream preprojection is very similar to the evaluation of XPath expressions against streaming data. The challenge is, given a set of XPath expressions, to extract all matching document nodes. Valuable groundwork for filtering XML streams via (a set of) XPath expressions has been presented in [1] and was refined in [8]. Both systems use deterministic finite automata (DFAs) to process the input stream. In the SPEX system [29, 33], XPath backward axes are rewritten to forward axes to enable a single-pass filtering of the document. As our XQuery fragment only covers forward XPath axes, namely the child and descendant axes, such optimizations are not required.

In our approach we lazily construct a deterministic finite automaton. A similar approach has been proposed in [16]. In the latter work it is shown that, although there is a computational overhead for small documents, lazy DFAs behave very well in streaming scenarios where, in general, we have to deal with large XML documents. Moreover it is shown that the resulting deterministic automata are acceptably small in practice. As shown in our experiments, this also holds for our approach. The construction of a lazy DFA for stream preprojection is covered in Section 6.

1.5.4 Garbage Collection

Garbage collection [44] is a well-known technique used for automatic memory management in programming languages. Several techniques have been proposed and also implemented so far. The main task of a garbage collector is to identify and remove “dead” objects, i.e. objects we know for sure that they can never be accessed by the program in the future evaluation. In the following we shortly sketch common garbage collection techniques.

- Reference Counting Garbage Collection

In reference counting garbage collection, each object counts the number of references to it. At runtime, new references can be created while old references might expire. As soon as the number of references to an object reaches zero, the object has become inaccessible and can be removed.

- Mark-And-Sweep Garbage Collection

In a first pass over the buffer, every object that still is reachable is marked with a flag. Unmarked objects can safely be removed from the buffer in a second pass, as they are known to be inaccessible.

- Copying Garbage Collection

In copying garbage collection, the heap (which stores all objects) is divided into two areas. Objects initially are buffered in the first area. As soon as the first area is filled, garbage collection is started and all active objects are copied in the second part of the buffer, thereby ignoring dead objects. In the following, the program is executed on the second buffer area until it expires and so on.

Beyond these basic techniques, there exist various combinations and other garbage collection techniques. Our system relies on a technique strongly related to garbage collection by reference counting. Each buffered XML node maintains a set of roles, where each role indicates the future relevance of a node. The number of roles assigned to a node can be seen as the reference count of an object and, as soon as a node loses its last role, it can be removed from the buffer.

The memory overhead for reference counting by garbage collection is small in general. In copying garbage collection for example, only half of the buffer is used for storing objects whereas in our reference counting garbage collection approach, for each node, only a small amount of additional information needs to be stored, namely the roles assigned to the node¹. Moreover, the reference counting approach allows for in-time garbage collection. Once the counter of an object has become zero, it can immediately be

¹In principle, to save space, we could simply store the number of roles instead. This optimization is discussed in Section 8.

removed from the buffer. Consequently, main memory usage can be minimized incrementally and early on. This is crucial for efficient buffer minimization in XQuery evaluation where we aim at decreasing both the high-watermark and average memory consumption throughout the evaluation process. Both mark-and-sweep and copying garbage collection in contrast, only offer support for aggregated collection, which usually is started whenever memory resources expire. Instead of removing single objects early on, many dead objects are collected at the same time.

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

1. XML documents as unranked, ordered, and node labeled trees over the two-sorted domain of nodes (with tag names from Tag) and values (strings over alphabet $Char$), and
2. 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.

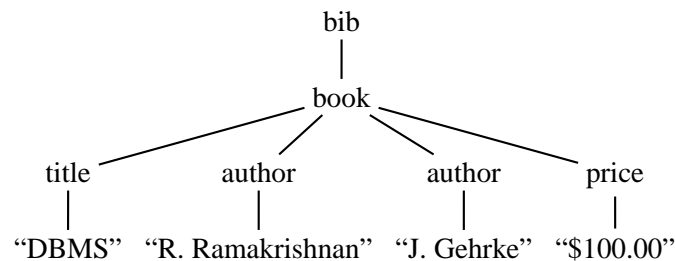
Example 1 (Dual views of XML documents) The XML document

```

<bib>
  <book>
    <title>“DBMS”</title>
    <author>“R. Ramakrishnan”</author>
    <author>“J. Gehrke”</author>
    <price>“$100.00”</price>
  </book>
</bib>

```

can be described by the tree



, and vice versa. □

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 , i.e. the number of nodes in T .

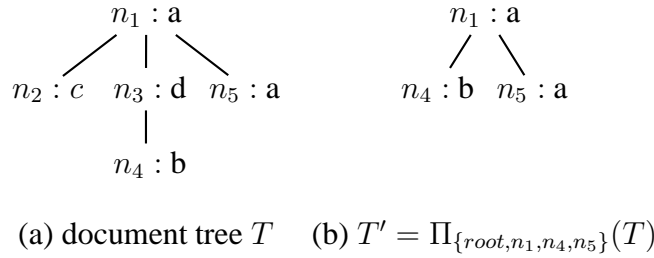


Figure 3: Document projection.

2.1 Document Projection

In the following, we introduce our concept of document projection via *projection trees*. The goal in document projection is to keep only those parts of the input document that are relevant for query evaluation. Previous work on projecting XML includes [5, 6, 27]. We discuss a refined, restructured case of these techniques, which particularly aims at projecting XML streams. A comparison between the projection techniques can be found in the experimental results.

Definition 1 (Document projection) 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 . \square

Example 2 (Document projection) 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 tree $T' = \Pi_{\{root, n_1, n_4, n_5\}}(T)$. For convenience we omit the root node when printing document trees, as it is part of every document tree. \square

We emphasize that, in contrast to the stream preprojection techniques presented in [5, 27], Definition 1 allows for interleaving nodes in paths when projecting a document, i.e. we are not forced to keep paths in complete, which can be useful when dealing with descendant axes.

2.1.1 Projection Paths

Path expressions are the primitives for navigating along the tree structure of XML documents. We now formally introduce paths as used internally for stream preprojection and role association. Paths expressions as used in the query are slightly different and will be

covered in Section 3. We refer to the abstract syntax definition of our XQuery fragment which is presented in Figure 5.

Definition 2 (Path) A *path* is a sequence $/s_1/\dots/s_n$ (for an absolute path, i.e. starting at the root node) or $s_1/\dots/s_n$ (for a relative path). ϵ denotes the empty path (for $n = 0$).

The *location step* expressions (also denoted as *path step* expressions) \mathfrak{s} are of the form *axis* $::$ *node-test* $[p]$, where *axis* is either the *child*, *descendant* or *descendant-or-self* axis.

A *node test* can either be a name in *Tag*, the wildcard symbol $*$ matching every tag node, or the wildcard *node()* matching any node type. Predicate p is either “*true*” or “*position()*= I ”². \square

We use the accustomed XPath abbreviations, i.e. we

1. omit predicate *true*, e.g. write `//bib` instead of `//bib[true]`,
2. write `//bib/*` for `/descendant::bib/child::*` and
3. shorten `//bib/book[position() = 1]` to `//bib/book[1]`.

By $\mathcal{P}[\pi](x)$ we denote the evaluation of path expression π on context node x according to the XPath semantics [15]. We define the evaluation of a path expression as a *projection path* using a function $\mathcal{PP}[\pi](x)$ which is characterized by the following equation.

$$\mathcal{P}[\pi](T) = \mathcal{P}[\pi](\Pi_{\mathcal{PP}[\pi](T)}(T)) \quad (I)$$

The intuition of the formula is the following. Let T' denote the subscript expression $\Pi_{\mathcal{PP}[\pi](T)}(T)$ from the formula above. T' describes the document tree obtained from projecting on the nodes that are selected by evaluating π as a projection path on T . Then π interpreted as an XPath query computes the same node-set when evaluated on T and T' . Intuitively spoken, in projecting XML documents w.r.t. a projection path, we retain all nodes that are relevant for this XPath expression.

As we do not require T' to be minimal, a naive evaluation of projection paths is the identity on T . Yet in practice, we will aim at discarding nodes that are not relevant to evaluating XPath expression π . In Figure 4 we define an evaluation strategy for projection paths which satisfies the semantics. While it does not yield the minimally projected tree, it can be performed very efficiently on streams.

²The motivation for considering the predicate “*position()* = 1” is the streaming evaluation of existence checks in XQuery conditions, where we are only interested in the first witness of a node.

<p>Let $axis :: \nu[p]$ be a location step expression.</p> $\mathcal{PP}[/\pi](x) = \{root\} \cup \mathcal{PP}[\pi](root)$ $\mathcal{PP}[\pi_1/\pi_2](x) = \{x\} \cup \mathcal{PP}[\pi_1](x) \cup \bigcup_{y \in \mathcal{P}[\pi_1](x)} \mathcal{PP}[\pi_2](y)$ $\mathcal{PP}[axis :: \nu[p]](x) = \{x\} \cup \mathcal{P}[axis :: \nu[p]](x)$
--

Figure 4: Projection path evaluation.

Example 3 (Projection path evaluation) Consider the evaluation of projection path $\pi = /descendant::a/descendant::b$ on the root node $root$ of document tree T from Figure 3. Following the evaluation strategy of Figure 4, we have

$$\begin{aligned}
& \mathcal{PP}[/descendant::a/descendant::b](root) \\
&= \{root\} \cup \mathcal{PP}[/a/b](root) \\
&= \{root\} \cup \mathcal{PP}[/a](root) \cup \bigcup_{y \in \mathcal{P}[/a](root)} \mathcal{PP}[/b](y) \\
&= \{root\} \cup \{n_1, n_5\} \cup \bigcup_{y \in \{n_1, n_5\}} \mathcal{PP}[/b](y) \\
&= \{root, n_1, n_5\} \cup (\mathcal{PP}[/b](n_1) \cup \mathcal{PP}[/b](n_5)) \\
&= \{root, n_1, n_5\} \cup (\{n_4\} \cup \emptyset) \\
&= \{root, n_1, n_4, n_5\}
\end{aligned}$$

The projected document tree T' is shown in Figure 3(b). Obviously, the characteristic equation for projection path evaluation (I) holds, i.e. we have

$$\mathcal{P}[p](\Pi_{\{root, n_1, n_4, n_5\}}(root)) = \mathcal{P}[p](root)$$

with $p = /descendant::a/descendant::b$: Both expressions evaluate to $\{n_4\}$. \square

2.1.2 Projection Trees

We next extend the definition of projection paths to that of *projection trees*. A projection tree is an unranked, unordered tree where the root node is labeled “/” and the remaining inner nodes are labeled with relative path expressions. A projection tree can be seen as a collection of projection paths: Each path from the root node to a leaf defines a projection path.

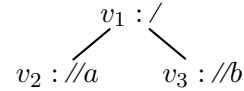
Thus, intuitively spoken, each fork in a projection tree defines projection paths relative to a context node. We now extend function $\mathcal{PP}[[t]](x)$ for evaluating a projection tree t in the natural manner. Let $p(C)$ refer to a node labeled p in the projection tree with set of children C . Then,

$$\mathcal{PP}[[p(C)]](x) = \mathcal{PP}[[p]](x) \cup \bigcup_{c \in C, y \in \mathcal{P}[[p]](x)} \mathcal{PP}[[c]](y).$$

When unifying the results of the evaluation of all projection paths described by the path tree against the same document, we get exactly the same result. Consequently, another evaluation strategy would be the following.

1. Identify each path π_i described by the projection tree.
2. For each path π_i , compute $N_i = \mathcal{PP}[[\pi_i]](\text{root})$, where *root* is the root node of the document to be projected.
3. The result is the union of all N_i .

Example 4 (Projection tree evaluation) Consider the document trees T and T' in Figure 3. The $T' = \Pi_{\mathcal{PP}[[t']]}(T)$ is a projection of T w.r.t. to projection tree t'



□

2.1.3 Minimal Projection

In principle we are interested in a minimal projection that fulfills equation (I). Obviously for each pair of projection tree and document, there exists at least one minimal projection. Given both the tree and the document its computation is always possible³, but in streaming scenarios we face the problem that the stream is delivered token by token. Consequently the decision whether to project away or keep the current token has to be made based on a (projected) prefix of the input stream.

Our projection technique aims at a trade-off between *effectiveness* (i.e. keeping the projected document as small as possible) and *efficiency* (i.e. with as little buffering overhead when parsing the input document as possible). With these considerations in mind, we implement the projection of XML streams as follows.

³We can simply check all power sets of document nodes to be a valid projection and choose the smallest one.

Similar to processing XPath on streams [8, 16], the projection tree is compiled into an automaton which matches the tokens from the XML stream against the nodes in the projection path. Due to our restriction to a fragment of forward XPath [33] and our evaluation semantics from Figure 4, the decision whether to discard or keep a document node can be already made when reading its opening tag from the input stream. While the actual streaming algorithm is straightforward and hence omitted at this point, we discuss possible optimizations to this technique in the implementation section.

2.2 Role Association

Let *roles* be a finite set of elements. A role-set is a multiset over *roles*, defined as a function where $m : \text{roles} \rightarrow \mathbb{N}$ 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* $\rho : \text{dom} \rightarrow m$ 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 $\text{add}_\rho(r, n)$, $(\rho(n))(r) = i + 1$. Likewise, after executing $\text{rem}_\rho(r, n)$, if $i > 0$ then $(\rho(n))(r) = i - 1$, and if $i = 0$, the removal of roles is undefined.

3 Query language

In this section, we define our XQuery fragment XQ, which comprises arbitrarily nested for-expressions, conditions, and joins. As argued in [20,22], this XQuery fragment covers most queries without aggregates that arise in practice.

The *abstract syntax* of an XQ query Q is shown in Figure 5 where $a \in \text{Tag}$, *string* denotes a string value, and *var* is a set of XQuery variables $\$x, \y, \dots with the distinguished root variable $\$root$, the unique free variable in any query.

Many syntactically richer fragments can be rewritten into our fragment and for a large set of practical queries expressed in full XQuery [49]

- let-expressions can be rewritten [20, 21],
- where-expressions can be replaced by if-then-else constructs [27], and
- multi step paths can be transformed to nested for-loops [23].

Example 5 (Path step rewriting) The following two queries are equivalent.

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

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

In contrast, the following two queries are not equivalent.

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

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

When evaluated on the stream $\langle a \rangle \langle a \rangle \langle b \rangle \langle a \rangle \langle a \rangle$, the first query extracts the XML document $\langle result \rangle \langle x \rangle \langle result \rangle$ whereas the second one returns $\langle result \rangle \langle x \rangle \langle x \rangle \langle result \rangle$. \square

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

Figure 5: XQuery fragment XQ.

3.1 Semantics of XQ

The semantics of XQ are the standard XQuery semantics [51]. 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 $\llbracket \alpha \rrbracket_k$ 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 , $[\dots]$ is the list constructor, and $[\]$ denotes the empty list. A for-loop is sequentially evaluated to a list of XML tokens as follows,

$$\llbracket \text{for } \$x_{k+1} \text{ in } \$y/axis::\nu \text{ return } \beta \rrbracket_k(\vec{e}) := \biguplus_{1 \leq i \leq |l|} \llbracket \beta \rrbracket_{k+1}(\vec{e}, l_i) \text{ where } l = \llbracket \$y/axis::\nu \rrbracket_k(\vec{e})$$

, i.e. variable $\$x_{k+1}$ is bound successively to each node in the list of nodes obtained from evaluating location step expression $\$y/axis::\nu$, 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 [22].

3.2 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 when certain nodes lose their roles. To this end, *signOff*-statements are inserted into queries at compile-time.

Definition 3 (Syntax of signOff-statements) A *signOff*-statement is an expression of the form $\text{signOff}(\$x/\pi, r)$ where $\$x$ is a variable, π is a relative path expression, and r is a role. \square

The next definition describes the semantics of *signOff*-statements. For each node matching the path step expression, the specified role is removed exactly once.

Definition 4 (Semantics of signOff-statements) 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

$$\llbracket \text{signOff}(\$x/\pi, r) \rrbracket_k(\vec{e})$$

is the following.

1. We define a node-set S . Let x be the node to which variable $\$x$ is currently bound, so $x = \llbracket \$x \rrbracket_k(\vec{e})$. If $\pi = \epsilon$, then $S = \{x\}$, otherwise S is the set of nodes reachable from node x via XPath expression π , i.e. $S := \mathcal{P}[\pi](x)$.
2. We remove role r from all nodes n in S by executing $\text{rem}_\rho(n, r)$. \square

In the following we state two requirements for the *safe* evaluation of an XQuery with *signOff*-statements.

Definition 5 (Safety of XQuery evaluation with *signOff*-statements) The evaluation of an XQuery expression with *signOff*-statements against an incoming XML stream is called *safe* exactly if the following conditions hold.

1. All node removals at runtime are defined.
2. After the query has been evaluated, all roles have been removed. □

Definition 5 enforces that exactly as many instances of roles are assigned to document nodes as are removed during query evaluation. This way, a strong relation between role assignment and query rewriting is established. In principle, the safety requirements stated above are not absolutely necessary for correct query evaluation. The only thing we need to guarantee is that no node loses its last role before it loses relevance for future query evaluation. Nevertheless, the safety statement above contributes to a simple and clear system. In principle, the definition above requires that each role is assigned to a node exactly as often as the node will be used in this role during evaluation. Vice versa, when strictly following this strategy, the safety conditions stated above will hold automatically.

Example 6 (Safety of XQuery evaluation) Consider the XQuery expression Q

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

For each a -labeled descendant of the root, the query outputs a node labeled a and nests all b -labeled descendants of the current a -node under it. The rewriting Q with *signOff*-statements for roles r_a and r_b is sketched below.

```
<result> {
  for $a in //a return
    (
      <a> {
        for $b in $a//b return
          ( <b/>, signOff($b, r_b) )
        } </a>,
      signOff($a, r_a)
    )
} </result>
```

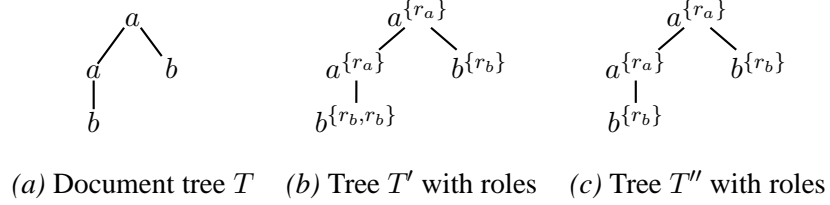


Figure 6: Projection and role assignment.

The evaluation of this query on document tree T' from Figure 6(a) is safe. During evaluation, variable $\$a$ is bound to each a -labeled node exactly once. Consequently the *signOff*-statement is executed once for each appearance of a and all a -labeled tag nodes in the buffer lose their role. The left side b -labeled tag node is matched twice, once for every a -labeled ancestor, therefore receives two updates and loses both roles. The second b -labeled node in contrast is matched once and loses its single role η .

In contrast, the evaluation of this query on document tree T'' from Figure 6(b) is not safe. The reason is that the first b -labeled node receives two updates for role η but carries role r_b only once. The first condition from Definition 5 is violated. \square

3.3 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 $Vars_Q$ denote the set of variables occurring in query Q . For two variables $\$x, \$y \in Vars_Q$, we say $\$y$ is the parent variable of $\$x$, denoted $parVar_Q(\$x) = \y if there exists a for-loop expression “for $\$x$ in $\$y/axis :: \nu$ return α ” in Q . We say $\$y$ is an ancestor variable of $\$x$, denoted $\$x <_Q \y , if either (1) $\$y = parVar_Q(\$x)$ 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 and ancestor relationships between variables. Variable trees are unranked and unordered, and are defined over the nodes $Vars_Q$ and with the edge relation $parVar_Q$.

Example 7 (Subexpressions, parent variables and variable trees) Consider the query expression Q

```
<result> {
  for $bib in /bib return
    (for $book in $bib/book return $book,
     for $article in $bib//article return $article)
} </result>
```

We introduce the abbreviation $for_{\$x}$ denoting the for-loop introducing variable $\$x$. Then $for_{\$book} \prec for_{\$bib} \prec Q$ and likewise $for_{\$article} \prec for_{\$bib} \prec Q$. Moreover, we have $\$bib <_Q \$book$ and $\$bib <_Q \$article$. The variable tree is sketched below.



Given two variables $\$x <_Q \y , the variable path between $\$y$ and $\$x$ is defined as follows. For $\$x = \y , $varpath_Q(\$y, \$x) = \epsilon$. Otherwise, $varpath_Q(\$y, \$x) = axis::\nu/varpath_Q(\$z, \$x)$ where $\$z$ is a variable such that $\$x \leq_Q \$z <_Q \$y$ with the query expression “for $\$z$ in $\$y/axis::\nu$ return α ” in Q . For Query Q from Example 7, e.g. we compute $varpath_Q(\$book, \$bib) = child::book$, and $varpath_Q(\$article, \$bib) = descendant::article$.

Let $r_Q : XQ \rightarrow roles$ be an injective function assigning a role to each XQ expression. We define dependencies $dep(\$x)$ 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 6 (Dependencies) Let Q be a query in XQ and $\$x \in Vars_Q$. The set of dependencies of variable $\$x$, denoted $dep(\$x)$, is defined as follows. Let $\beta \preceq Q$ with $r_Q(\beta) = r$, then

1. $\langle axis::\nu[1], r \rangle \in dep(\$x)$ if $\beta = \text{“exists}(\$x/axis::\nu\text{”}$,
2. $\langle axis::\nu/descendant-or-self::node(), r \rangle \in dep(\$x)$ if β is either an output expression of the form “ $\$x/axis::\nu$ ” or a node-value/node-node test expression of the form “ $\$x/axis::\nu \text{ RelOp } \chi$ ” or “ $\chi \text{ RelOp } \$x/axis::\nu$ ”, and
3. $\langle descendant-or-self::node(), r \rangle \in dep(\$x)$ where $\beta = \text{“}\$x\text{”}$ is an output expression.

□

Example 8 (Dependency paths) Consider the XQuery expression Q

```
for $book in //book return
  if (exists $book/price)
  then $book/author
  else $book
```

that outputs for each book found in the document either the author (in case a price tag exists) or the whole book. The set of dependency paths for variable $\$book$ is

$$\begin{aligned} dep(\$book) = & \\ & \{ \langle child::price[1], r_1 \rangle, \\ & \quad \langle child::author/descendant-or-self::node(), r_2 \rangle, \\ & \quad \langle descendant-or-self::node(), r_3 \rangle \} \end{aligned}$$

where r_1 is the role associated to the existence-check expression, r_2 is the role associated to subexpression “ $\$book/author$ ” and r_3 is the role associated to the output expression “ $\$book$ ”. \square

4 Static Analysis

The static analysis of the input query in our system tracks several goals:

1. The projection tree is computed from the query, so that at runtime, a projected version of the XML input stream can be computed where query-irrelevant parts of the input are not copied into the buffer in the first place.
2. A set of roles is derived from the query. While processing the input, roles are assigned to buffered nodes on-the-fly, the prerequisite for active garbage collection.
3. Finally, by statical query normalization and the insertion of *signOff*-statements, buffered nodes can be deleted at runtime once they have become irrelevant to query evaluation.

In particular, we can make the following guarantees for our approach.

Theorem 1 (Correctness) Let Q be an XQ query and let T be the input document tree. Let Q' be the rewritten query Q with *signOff*-statements and let T' be the projected document tree with roles assigned to its nodes. Then $\llbracket Q \rrbracket_1(T) = \llbracket Q' \rrbracket_1(T')$. \square

4.1 Deriving Projection Trees

Given an XQuery Q in our fragment, we statically derive a projection tree t such that all documents projected w.r.t. t are *sound* [5], so for a document tree T , $\llbracket Q \rrbracket_1(T) = \llbracket Q \rrbracket_1(\Pi_{\mathcal{PP}[t]}(T))$.

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 are bound are relevant to query evaluation, yet their subtrees are irrelevant for the variable bindings per se. These considerations are captured by the dependencies (see Definition 6).

Given query Q , we derive the projection tree t and a mapping r_π from nodes in t to roles in three steps.

1. Construct the variable tree of Q .
2. Extend the variable tree by nodes labeled with path expressions: For each variable $\$x$ and for each $\langle \$x/\pi, r \rangle \in \text{dep}(\$x)$, we add a node n with label “ π ”, an edge from $\$x$ to n , and we define $r_\pi(n) := r$.

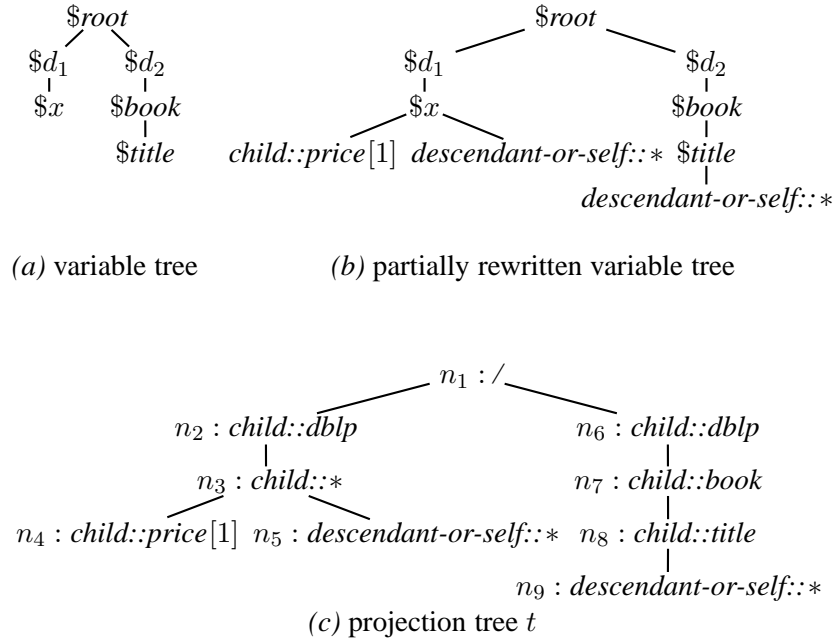


Figure 7: Projection tree of Example 9.

3. The root node is relabeled “/” and for each variable node n labeled $\$x$ with the corresponding for-loop $\beta = \text{“for } \$x \text{ in } \$y/axis::\nu \text{ return } \alpha\text{”}$, we relabel n with “ $axis::\nu$ ” and define $r_\pi(n) = r_Q(\beta)$.

In the implementation section, we discuss simplifications to projection trees, e.g. by merging subsuming branches.

Example 9 (Projection tree construction) Consider the XQuery expression

```
<result> {
  (for $d1 in /dblp return
    for $x in $d1/* return
      if (exists $x/price)
        then $x else (),
    for $d2 in /dblp return
      for $b in $d2/book return $b/title
  )
} </result>
```

<p>Execution of $\mathcal{R}^{r_\pi} \llbracket p(C) \rrbracket (x)$:</p> <pre> begin role $r := r_\pi(p)$; for each document node $n \in \mathcal{P} \llbracket p \rrbracket (x)$ do execute $add(r, n)$; for each child $c \in C$ of p do for each document node $y \in \mathcal{P} \llbracket p \rrbracket (x)$ do execute $\mathcal{R}^{r_\pi} \llbracket c \rrbracket (y)$; end </pre>

Figure 8: Semantics of role assignment.

The step-wise extraction of the projection tree for this query is shown in Figure 7. Part (a) shows the variable tree for the query above. In part (b), the variable tree has been extended by the corresponding dependencies. Variable x has two dependencies, namely $\langle child::price[1], r_1 \rangle$ for the existence check expression with role r_1 associated and dependency $\langle descendant-or-self::node() \rangle$ for the output expression “ x ” with associated role r_2 . For variable d_2 we have the single dependency $\langle child::title/descendant-or-self::node(), r_3 \rangle$ where r_3 is the role of output expression “ $b/title$ ”. Part (c) shows the final projection tree which is obtained by relabeling all variable nodes. \square

4.2 Assigning Roles to Buffered Nodes

In active garbage collection, we assign roles to buffered nodes. Role assignment is tightly bound to document projection. Let t be a projection tree and r_π a function assigning a role to every node in the projection tree. For a given input document T , we implement role assignment via the function $\mathcal{R}^{r_\pi} \llbracket t \rrbracket (T)$.

Let $p(C)$ be a node in the projection tree labeled p and with set of children C , and let x be the current context node. Then $\mathcal{R}^{r_\pi} \llbracket p(C) \rrbracket (x)$ is evaluated as shown in Figure 8.

Example 10 For the query from Example 6 and the roles r_{root}, r_a, r_b , Figure 6(b) illustrates role assignment for a concrete document tree⁴. Note that the b -labeled node carries role r_b twice, as it is matched two times by the algorithm. \square

⁴The assignment of role r_{root} via $r_\pi(v_1) = r_{root}$ to the implicit root node of the XML document is not shown.

$$\frac{\text{if } \chi \text{ then } \epsilon \text{ else } \epsilon}{\epsilon} \text{ IFDECOMPOSITION I}$$

$$\frac{\text{if } \chi \text{ then } \alpha \text{ else } \epsilon \quad \alpha \neq \epsilon}{\text{if } \chi \text{ then } \alpha} \text{ IFDECOMPOSITION II}$$

$$\frac{\text{if } \chi \text{ then } \epsilon \text{ else } \beta \quad \beta \neq \epsilon}{\text{if } (\text{not } \chi) \text{ then } \beta} \text{ IFDECOMPOSITION III}$$

$$\frac{\text{if } \chi \text{ then } \alpha \text{ else } \beta \quad \alpha \neq \epsilon \quad \beta \neq \epsilon}{(\text{if } \chi \text{ then } \alpha, \text{ if } (\text{not } \chi) \text{ then } \beta)} \text{ IFDECOMPOSITION IV}$$

Figure 9: Decomposition of if-statements.

4.3 XQ Rewriting

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 never must be issued too early, as this could corrupt the query result. The insertion of *signOff*-statements into queries will assure the latter. We start by a discussion of query normalization rules before discussing the insertion of *signOff*-statements.

4.3.1 Query Normalization

In the query rewriting phase, we push if-statements into for-loops. First, if-expression are decomposed, i.e. we transform each if-expression into a sequence of two if statements, one for the then-part and one for the else-part, respectively⁵. The decomposition of if-statements can easily be realized by the inference rules presented in Figure 9.

The inference rules also cover special cases where the then- and/or the else-part is empty, as they are very common in practice. Given that both parts are empty, the if-construct produces no output and can be eliminated completely (IFDECOMPOSITION I).

⁵The formal extension of the XQ syntax by simple if-statements without else-parts is straightforward and will be ignored.

$$\frac{\text{if } \chi \text{ then } (\alpha, \beta)}{(\text{if } \chi \text{ then } \alpha, \text{if } \chi \text{ then } \beta)} \text{ PUSHSEQUENCE}$$

$$\frac{\text{if } \chi \text{ then } \langle a \rangle \alpha \langle /a \rangle}{(\text{if } \chi \text{ then } \langle a \rangle, \text{if } \chi \text{ then } \alpha, \text{if } \chi \text{ then } \langle /a \rangle)} \text{ PUSHNODECONSTR}$$

$$\frac{\text{if } \chi \text{ then } \{ \text{for } \$x \text{ in } \$y/\text{axis}::\text{nt return } \alpha \}}{\text{for } \$x \text{ in } \$y/\text{axis}::\text{nt return } \{ \text{if } \chi \text{ then } \alpha \}} \text{ PUSHFOR}$$

Figure 10: Pushing down if-statements.

Rules IFDECOMPOSITION II and IFDECOMPOSITION III cover both cases in which exactly one of these parts is empty, while rule IFDECOMPOSITION VI covers the general case, where nothing can be optimized.

The rules for pushing down if-statements are presented in Figure 10. Pushing down an if-statement into a sequence expression for example, covered by rule PUSHSEQUENCE, is realized by its application to both parts of the sequence. Rule PUSHNODECONSTR, which applies to node construct expressions, and rule PUSHFOR, which covers for-loops, are very similar. The set of inference rules, applied to the query with normalized if-statements until a fixpoint is reached, pushes all if-expressions as deep as possible into the expression syntax tree.

Query normalization is necessary to ensure the correctness of the active garbage collection system. As shown in the following subsection, *signOff*-statements will always be inserted at the end of for-loops. The execution of for-loops which are encapsulated into if-statements depends on the value of the if-condition, which is undecidable in the rewriting phase. Consequently we can never be sure whether *signOff*-statements will finally be executed. Obviously, safe query evaluation, as introduced in Definition 5, can not be guaranteed. Pushing if-statements into for-loops makes sure that no role updates at runtime will depend on conditions.

We now will demonstrate both sets of inference in an example. First if-statements will be decomposed, then the query will be rewritten using the inference rules for if-pushing.

Example 11 (Query normalization) Consider the XQ expression

```
<result> {
  for $bib in /bib return
    if (exists $bib/book)
    then <book-list> {
      for $book in $bib/book return
        <book> { $book/title } </book>
    } </book-list>
  else <no-books/>
} </result>
```

The application of the if-decomposition rules yields the following query.

```
<result> {
  for $bib in /bib return
    (
      if (exists $bib/book)
      then <book-list> {
        for $book in $bib/book return
          <book> { $book/title } </book>
        } </book-list>,
      if (not (exists $bib/book))
      then <no-books/>
    )
} </result>
```

Finally we sequentially apply rules PUSHNODECONSTR and PUSHFOR to the first if-statement. The resulting query is shown below.

```
<result> {
  for $bib in /bib return
    (
      (
        if (exists $bib/book) then <book-list>,
        for $book in $bib/book return
          if (exists $bib/book)
            then <book> { $book/title } </book>,
        if (exists $bib/book) then </book-list>
      ),
      if (not (exists $bib/book)) then <no-books/>
    )
} </result>
```

□

While the normalization rules are useful well-founded theory, they are rather inefficient in practice. As illustrated in Example 11, pushing down if-statements creates multiple instances of the same statement. They all have to be evaluated, thus the same condition has to be evaluated multiple times. Caching and reusing the result complicates the evaluation process, so we decided for a slightly different approach. Instead of query normalization, we changed the XQ if-statement semantics. As usual, we first evaluate the condition and, based on this result, either the then- or the else-part. The subsequent evaluation of all *signOff*-statements that are nested under the non-matched part ensures independence from undecidable conditions and, computes the same result.

4.3.2 Inserting *signOff*-Statements

Definition 7 (Straight variables) Let Q be an XQ query and let $\$z \in \text{Vars}_Q$. Variable $\$z$ is *straight* if either $\$z = \$root$ or there is a query expression $\beta = \text{“for } \$z \text{ in } \$y/\text{axis}::\nu \text{ return } \alpha\text{”}$ such that (1) $\$y$ is straight and (2) there is no for-loop expression $\gamma = \text{“for } \$u \text{ in } \$v/\text{axis}'::\nu' \text{ return } \alpha'\text{”}$ where $\$u$ is no ancestor variable of $\$z$ and $\beta \prec \gamma \preceq Q$. \square

Definition 8 (First straight ancestor) Let Q be a query and let $\$x \in \text{Vars}_Q$. The *first straight ancestor variable* of $\$x$ is defined as

$$fsa_Q(\$x) \stackrel{def}{:=} \begin{cases} \$x & \text{if } \$x \text{ is straight} \\ fsa_Q(parVar_Q(\$x)) & \text{otherwise.} \end{cases} \quad \square$$

Example 12 (Straight variables, first straight ancestors) Variables $\$a$ and $\$b$ in query Q_1 from Example 5 are straight, i.e. $fsa_{Q_1}(\$a) = \a and $fsa_{Q_1}(\$b) = \b . In query Q_2 from Figure 12, variable $\$b$ is not straight, in particular $fsa_{Q_2}(\$b) = \$root$. \square

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* are shown below, and algorithm $su_Q(var \$x)$ is provided in Figure 11. Note that both the rewriting rules and the algorithm apply to normalized queries.

$$(\beta = Q) \frac{\beta : \langle a \rangle \alpha \langle /a \rangle}{\langle a \rangle (\alpha, su_Q(\$root)) \langle /a \rangle}$$

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

```

Algorithm  $su_Q(\text{variable } \$x)$ :
begin
  if ( $\$x \neq \$root$ )
  then
    begin
      let  $\$x$  be defined in  $\beta$  : “for  $\$x$  in  $\$y/axis::\nu$  return  $\alpha$ ”;
      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 11: Static query rewriting.

The first rule, which applies to the query Q itself, inserts the corresponding *signOff*-statements for the $\$root$ variable. In the second rule, all for-loop subexpressions are rewritten such that their original body is evaluated before the *signOff*-statements are executed.

Example 13 (XQuery rewriting) Consider the query Q from Example 6 and its valid rewriting Q' . Each buffered document node to which a variable $\$a$ or $\$b$ is bound loses its role once the scope of the respective variable ends. For document trees T and T' from Figure 6, we can verify that $\llbracket Q \rrbracket_1(T) = \llbracket Q' \rrbracket_1(T')$. \square

In the following, we will ignore the *signOff*-command for the $\$root$ variable. The reason is that the elimination of both the role assigned to the *root* node of the document and the *signOff*-statement for the corresponding variable $\$root$ is always possible. We refer the reader to Subsection 6.7.1 for a discussion of redundant role elimination. As we do neither show the document *root* node in figures, nor the variable $\$root$ in the queries⁶, the corresponding statement will be ignored for consistency reasons.

⁶We always used the shortcut */bib* for $\$root/bib$ before and will do so in the following.

```

<result>
{
  for $a in //a return
    <a>
    {
      for $b in //b return
        <b/>
    }
  </a>
}
</result>

```

(a) Query Q

```

<result>
{
  for $a in //a return
    (
      <a>
      {
        for $b in //b return
          <b/>
      }
      </a>,
      signOff($a, r_a),
      signOff($root//b, r_b)
    )
}
</result>

```

(b) Rewritten query Q'

Figure 12: 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.

5.1 Irrelevant Nodes

We now specify which nodes are irrelevant to query evaluation and can thus be safely purged from the buffer without affecting the ongoing query evaluation. Unless a node is irrelevant, we conservatively assume that it is still relevant to query evaluation.

Definition 9 (Irrelevant document nodes) A (buffered) document node x is *irrelevant* if $\delta(x) = \emptyset$ and for all descendants y of x , $\delta(y) = \emptyset$. \square

The following discussion 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 whenever a document node x is matched by a set of nodes n_1, \dots, n_k in the projection tree, x is assigned the roles associated with n_1, \dots, n_k and copied into the buffer. Hence, all nodes that are copied into the buffer are relevant as they carry at least one role. Next, we discuss the loss of roles at runtime.

5.2 Buffer Cleanup Algorithm

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 13 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* for deletion 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 algorithm rem( $r, n$ ); // remove role from nodes in  $S$ 
      while ( $n \neq \text{root}$  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
  end

```

Figure 13: Localized active garbage collection.

6 System Implementation

We have implemented active garbage collection for a prototype XQuery engine, called the “Garbage Collected XQuery system” GCX . It 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 targeting at low memory consumption.

6.1 System Architecture

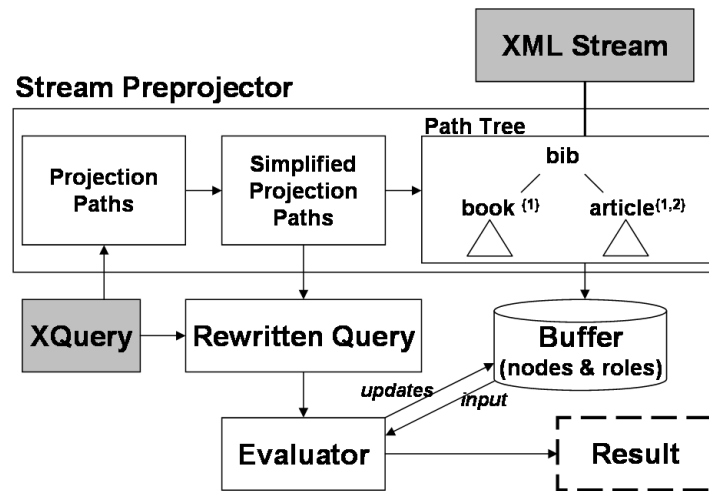


Figure 14: System architecture.

Figure 14 shows the system architecture from a rather abstract and functional point of view. The set of projection paths is computed by statical query analysis. Redundant paths are eliminated (see Subsection 6.7.1) and finally a path tree is constructed. The incoming stream is matched against this tree and, in doing so, we project away unneeded tokens and assign roles to tokens that pass preprojection. Before evaluation, the query is rewritten using the set of minimized projection paths. The main task of query rewriting is the insertion of *signOff*-commands for each minimized projection path. The query is evaluated on the buffer which contains the projected and marked stream. As part of the evaluation process, *signOff*-commands are emitted as requested by the rewritten query, causing buffered nodes to lose their roles.

6.2 System Components and Runtime Interaction

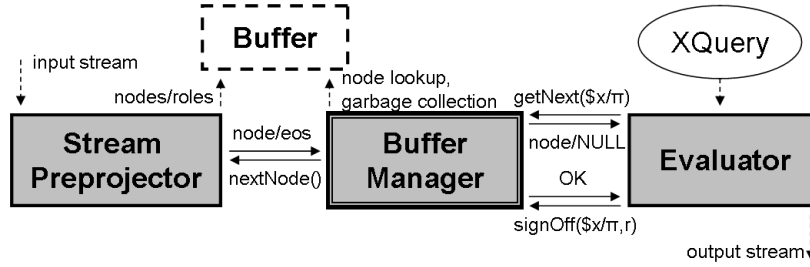


Figure 15: System interaction.

While we described system architecture from an abstract point of view in Subsection 6.1, we now switch to the implementation details and discuss the interaction between the system components during query evaluation. Our system, GCX, comprises three main components, the *query evaluator*, the *stream preprojector*, and the *buffer manager*. The *pull-based* interaction between these components is illustrated in Figure 15.

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 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 reads the input stream until a token is matched by the projection tree. In the latter case, the token is copied directly into the buffer, and roles are assigned on-the-fly. Via this chain of commands, the query evaluator incrementally reads the input stream and evaluates the query on-the-fly.

Note that in this pull-based approach, the query evaluator has control over the time when new tokens are loaded into the buffer. Only when evaluation is blocked due to an

incomplete buffer, new tokens are requested by the evaluator. As soon as evaluation can proceed, stream preprojection is interleaved and evaluation is resumed. This approach significantly contributes to small buffers and low main memory consumption throughout the evaluation process.

Moreover, all system components work in large parts independently from each other. Communication between the components works via simple and clear interfaces, e.g. the “*getNext(\$x/π)*” command requests the next binding for a variable from the buffer manager, the call “*nextNode()*” tells the stream preprojector to project one more node into the buffer. These interfaces underline the strict separation of the system components and their functionality. While the buffer manager is responsible for query evaluation, it is not involved into buffer management and stream preprojection. It only cares about the computation of next bindings for variables and the execution of the active garbage collection algorithm on the buffer, i.e. it is aware of stream preprojection and query evaluation. The only task of the stream preprojector is to project away unneeded tokens and to buffer needed tokens together with their assigned roles.

Also note that, at each time in query processing, exactly one of these components has control. This is crucial to avoid conflicts on shared data structures. Consider for example the stream preprojector and the buffer manager. As both components work on the buffer, running them in parallel (i.e. using a threaded architecture) would require special care: Appending tokens to and removing tokens from the buffer at the same time might result in inconsistent states. This problem does not exist in our approach, as at no time both the stream preprojector and the buffer manager can be active.

6.3 Buffer Representation

As our query fragment is composition-free [22], all XQuery variables will be bound to existing document nodes. Hence, we can use a *single* buffer which contains the projected document tree, where some nodes are marked “unfinished” as their closing tags have not been read yet. Buffer representation in general is a trade-off between memory consumption and evaluation time: Buffering an unstructured or even encoded string minimizes memory usage but disqualifies for efficient access to child and descendant nodes. In contrast, complex data structures as proposed in [45] require more space but offer support for efficient search, e.g. using indices. As our system primarily aims at low main memory consumption, we decided for a simple tree data structure, with parent-child and next-sibling pointers between nodes. Additionally, each buffered tag node maintains a flag that indicates whether the closing tag has been read so far and role information⁷.

⁷After application of the role aggregation optimization which will be presented in Subsection 6.5.2, only tag nodes carry roles.

6.4 Projection Tree Construction

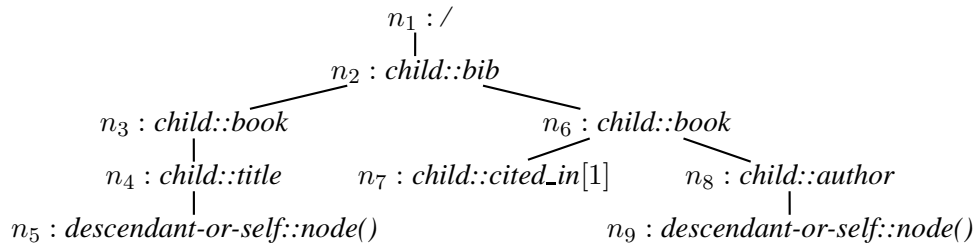
Projection trees can be interpreted as a collection of projection paths, i.e. each projection tree path from the root node to a leaf describes a projection path. In Subsection 4.1 we presented a formal approach to projection tree computation. We now reconsider the topic from an algorithmic point of view. Throughout the chapter, we will discuss issues and problems that arise in implementation and present a solution very similar to DFA determinization in order to realize efficient stream preprojection. We start with a discussion of the relation between projection trees and projection paths (Subsection 6.4.1), present inference rules for *base projection path extraction* (Subsection 6.4.2), describe how these base projection paths can be extended to projection paths and conclude with an efficient algorithm for projection tree setup basing on the set of projection paths (Subsection 6.4.4).

6.4.1 Projection Trees and Projection Paths

Example 14 (Projection trees as multisets of projection paths) Let Q be the following XQuery expression

```
for $bib in /bib return
(
  for $book1 in $bib/book return
    $book1/title,
  <cited_authors> {
    for $book2 in $bib/book return
      if (exists $book2/cited_in)
      then $book2/author
  } </cited_authors>
)
```

The query first returns a list of book titles and, subsequently, encapsulated in a *<cited authors>* tag, for each book that is cited the list of all authors. Following the rules for projection tree setup presented in Subsection 4.1, we compute the following projection tree.



This projection tree describes the following projection paths.

- The path from n_1 to n_5 , namely
`/child::bib/child::book/child::title/descendant-or-self::node()`
- The path from n_1 to n_7 , namely
`/child::bib/child::book/child::cited_in[1]`
- The path from n_1 to n_9 , namely
`/child::bib/child::book/child::author/descendant-or-self::node()`

In principle, a projection tree might define the same path multiple times. Assume for example we output titles instead of authors in the second pass. Then the projection paths from n_1 to n_5 and from n_1 to n_9 would be equal. Thus, a projection tree defines a *multiset* of projection paths rather than a set. Same-labeled paths can be distinguished by the underlying query subexpression they have been derived from. In all cases, this subexpression will be a variable or a varstep expression, which may be part of another expression. Path `/child::bib/child::book/child::cited_in[1]` for example has been derived from the varstep expression `"$book2/cited_in"`, which itself is a subexpression of an existence check expression. If the variable uniquely identifies the path, i.e. there exists no identical path that was bound to this variable, we usually refer to this path as the path which relies on the corresponding variable. \square

6.4.2 Base Projection Path Extraction

In the following we present a set of inference rules for base projection path extraction. The extracted set of paths slightly differs from the projection paths presented in Example 14 and, for this reason, is called the set of *base projection paths*. The latter name has been chosen because some of the base projection paths only capture a prefix of the original projection path. In the following we use the names base projection paths and base paths interchangeably. The extension to full projection paths will be discussed in Subsection 6.4.3. Before presenting the inference rules, we formally introduce the notion of path environments.

Definition 10 (Path environment) A *path environment* $PEnv$ is a function that maps variables to absolute paths. The empty path environment $PEnv_\epsilon$ maps each variable to the empty path ϵ . The function $getPath_{PEnv}(\$x)$ returns the current binding for variable $\$x$ in path environment $PEnv$. We write $PEnv \cup \{(\$x, p)\}$ to denote that path environment $PEnv$ is extended by the mapping $\$x \rightarrow p$, where p is an absolute path. \square

Example 15 (Path environment) Let $PEnv := PEnv_\epsilon \cup (\$x, /bib)$ and $PEnv' := PEnv \cup (\$y, /bib/book)$ be two path environments. Then

$$\begin{aligned} \text{getPath}_{P_{Env'}}(\$x) &= /bib, & \text{getPath}_{P_{Env}}(\$x) &= /bib, \\ \text{getPath}_{P_{Env'}}(\$y) &= /bib/book, \text{ and } & \text{getPath}_{P_{Env}}(\$y) &= \epsilon. \end{aligned}$$

□

We are now in the position to present the inference rules for base projection path extraction. The inference algorithm will be started on an XQ expression under path environment P_ϵ . It returns a tuple $(pp_{nec}, pp_{ec}, pp_v)$ where

- pp_{nec} is the base path partition multiset of non-existence-check base paths. It contains all output expression paths as well as all paths used for node-node and node-value comparison checks.
- pp_{ec} is the base path partition multiset of existence-check base paths, i.e. contains all paths used for existence checks.
- pp_v is the base path partition multiset of variable paths, i.e. contains all base paths variables are bound to during query evaluation.

Figure 16 shows the inference rules for non-conditional expression and Figure 16 shows the inference rules for conditional expression, respectively. In the following we shortly discuss the inference rules.

[EMPTY] The empty query ϵ contains no base projection paths.

[STRING] The output of a fixed string s implies no base projection paths.

[VARIABLE] Variable output expressions introduce a new *non-existence-check base path* defined by the path the variable is bound to in the current path environment.

[SEQUENCE] For a sequence expression of the form “ $\alpha \beta$ ”, we first extract the base paths for subexpression α , second the base paths for subexpression β . The result is the union of the corresponding base projection path partition multisets computed for α and β .

[NODECONSTRUCTION] The base projection paths defined by a node construction expression are computed by evaluating the encapsulated subexpression.

[VARSTEP] Variable step expressions introduce a new *non-existence-check base projection path*. The base projection path can be constructed by appending the step expression to the path the variable is bound to in the current path environment.

[FOR I and FOR II] The inference rules covering for-loops are the most complicated. In both rules, we introduce a new path environment by adding a mapping for the introduced variable $\$x$. The new environment is used to infer the base projection paths for the return-expression. Additionally, variable $\$x$ implies a base projection path by itself, namely a *variable projection path*. Consequently, the path to which $\$x$ is bound in the new path environment is added to the partition set of variable projection paths. While the first

$$\begin{array}{c}
\frac{}{PEnv \vdash \epsilon \Longrightarrow (\emptyset, \emptyset, \emptyset)} \text{EMPTY} \\
\\
\frac{}{PEnv \vdash s \Longrightarrow (\emptyset, \emptyset, \emptyset)} \text{STRING} \\
\\
\frac{p = \text{getPath}_{PEnv}(\$x)}{PEnv \vdash \$x \Longrightarrow (\{p\}, \emptyset, \emptyset)} \text{VARIABLE} \\
\\
\frac{PEnv \vdash \alpha \Longrightarrow (P_1, P_2, P_3) \quad PEnv \vdash \beta \Longrightarrow (P'_1, P'_2, P'_3)}{PEnv \vdash \alpha \beta \Longrightarrow (P_1 \cup P'_1, P_2 \cup P'_2, P_3 \cup P'_3)} \text{SEQUENCE} \\
\\
\frac{PEnv \vdash \alpha \Longrightarrow (P_1, P_2, P_3)}{PEnv \vdash \langle t \rangle \alpha \langle /t \rangle \Longrightarrow (P_1, P_2, P_3)} \text{NODECONSTRUCTION} \\
\\
\frac{p = \text{getPath}_{PEnv}(\$x)}{PEnv \vdash \$x/\text{axis} :: nt \Longrightarrow (\{p/\text{axis} :: nt\}, \emptyset, \emptyset)} \text{VARSTEP} \\
\\
\frac{p = \text{getPath}_{PEnv}(\$y) \quad PEnv' = PEnv \cup \{(\$x, p/\text{axis} :: nt)\} \quad PEnv' \vdash \alpha \Longrightarrow (P_1, P_2, P_3)}{PEnv \vdash \text{for } \$x \text{ in } \$y/\text{axis} :: nt \text{ return } \alpha \Longrightarrow (P_1, P_2, \{p/\text{axis} :: nt\} \cup P_3)} \text{FOR I} \\
\\
\frac{PEnv' = PEnv \cup \{(\$x, /\text{axis} :: nt)\} \quad PEnv' \vdash \alpha \Longrightarrow (P_1, P_2, P_3)}{PEnv \vdash \text{for } \$x \text{ in } /\text{axis} :: nt \text{ return } \alpha \Longrightarrow (P_1, P_2, \{/\text{axis} :: nt\} \cup P_3)} \text{FOR II} \\
\\
\frac{PEnv \vdash_c \chi \Longrightarrow (P_1, P_2, P_3) \quad PEnv \vdash \alpha \Longrightarrow (P'_1, P'_2, P'_3)}{PEnv \vdash \text{if } \chi \text{ then } \alpha \Longrightarrow (P_1 \cup P'_1, P_2 \cup P'_2, P_3 \cup P'_3)} \text{IF}
\end{array}$$

Figure 16: Inference rules for non-conditional expressions.

$$\begin{array}{c}
\frac{p = \text{getPath}_{PEnv}(\$x)}{PEnv \vdash_c \text{exists } \$x/\text{axis} :: nt \implies (\emptyset, \emptyset, \{p/\text{axis} :: nt\})} \text{EXISTENCE} \\
\\
\frac{p = \text{getPath}_{PEnv}(\$x)}{PEnv \vdash_c \$x/\text{axis} :: nt \text{ RelOp } s \implies (\{p/\text{axis} :: nt\}, \emptyset, \emptyset)} \text{NODEVALUE} \\
\text{where RelOp} \in \{ <, \leq, =, \geq, > \} \\
\\
\frac{p_x = \text{getPath}_{PEnv}(\$x) \quad p_y = \text{getPath}_{PEnv}(\$y)}{PEnv \vdash_c \$x/\text{axis}_x :: nt_x \text{ RelOp } \$y/\text{axis}_y :: nt_y \implies (\{p_x/\text{axis}_x :: nt_x, p_y/\text{axis}_y :: nt_y\}, \emptyset, \emptyset)} \text{NODENODE} \\
\text{where RelOp} \in \{ <, \leq, =, \geq, > \} \\
\\
\frac{PEnv \vdash_c \alpha \implies (P_1, P_2, P_3)}{PEnv \vdash_c \text{not } \alpha \implies (P_1, P_2, P_3)} \text{NEGATION} \\
\\
\frac{PEnv \vdash_c \alpha \implies (P_1, P_2, P_3) \quad PEnv \vdash_c \beta \implies (P'_1, P'_2, P'_3)}{PEnv \vdash_c \alpha \text{ Op } \beta \implies (P_1 \cup P'_1, P_2 \cup P'_2, P_3 \cup P'_3)} \text{AND/OR} \\
\text{where Op} \in \{\text{and, or}\} \\
\\
\frac{}{PEnv \vdash_c \text{true} \implies (\emptyset, \emptyset, \emptyset)} \text{TRUE}
\end{array}$$

Figure 17: Inference rules for conditional expressions.

inference rule captures for-loops which introduce a variable relative to another variable, the second rule covers for-loops which introduce a variable relative to document root /.

[IF] In a first step we evaluate the if-condition using the inference rules for conditions presented below⁸. Subsequently we compute the partition sets of the *then*-part. The result is obtained by merging the extracted base projection path partition sets. Note that we do not cover if-then-else expression, as they were replaced in the query normalization phase (Subsection 4.3.1).

[EXISTENCE] Existence-check expressions imply an *existence-check base projection path*. The base projection path is obtained by appending the corresponding path step expression to the current binding of the involved variable.

[NODEVALUE] Node-value comparisons introduce a *non-existence-check base path*.

[NODENODE] Node-node comparison expressions are similar to node-value comparison expressions. As they carry two paths, we introduce two *non-existence-check base paths*.

[NEGATION] The base projection paths defined by a negation expression of the form “*not* α ” are the paths defined by its inner expression α .

[AND/OR] Boolean expressions connected through an *and* (*or*) operator are evaluated independently from each other. The result is obtained by conjoining the corresponding partition sets.

[TRUE] *true*-expressions do not introduce a new base projection path.

6.4.3 Base Projection Path Conversion

We start the discussion on base projection path conversion, i.e. the relation between base projection paths and projection paths, by demonstrating the extraction algorithm from the previous subsection in a short example.

Example 16 (Base projection path extraction) Consider the query from Example 14. The application of the inference rule on this query returns the tuple $(pp_{nec}, pp_{ec}, pp_v)$ where

$$\begin{aligned} pp_{nec} &= \{ /child::bib/child::book/child::title, \\ &\quad /child::bib/child::book/child::author \}, \\ pp_{ec} &= \{ /child::bib/child::book/child::cited_in \}, \text{ and} \\ pp_v &= \{ /child::bib, /child::bib/child::book, /child::bib/child::book \} \end{aligned}$$

Note that path $/child::bib/child::book$ occurs twice in pp_v . It has been extracted for both variables $\$book1$ and $\$book2$. When comparing the set of base projection paths to the set of projection paths from Example 14, we can observe a strong relation between the paths in pp_{nec} and pp_{ec} and the set of projection paths. In detail, the following relations hold.

⁸For convenience, we use operator \vdash_c for conditional expression inference rules.

- The non-existence-check base path $/child::bib/child::book/child::title$ is a prefix path of $/child::bib/child::book/child::title/descendant-or-self::node()$
- Non-existence-check base path $/child::bib/child::book/child::author$ is a prefix path of $/child::bib/child::book/child::author/descendant-or-self::node()$
- Existence-check base path $/child::bib/child::book/child::cited_in$ is structural identical to $/child::bib/child::book/child::cited_in[1]$

In this example, the multiset of projection paths can be constructed by (1) appending the pathstep $/descendant-or-self::node()$ to every non-existence-check base path and (2) adding attribute “[1]” to every existence-check base path. Note that this extension is very similar to the projection tree setup algorithm discussed in Subsection 4.1. We next discuss the role of variable base paths which, under certain circumstances, may also imply projection paths by themselves. \square

While running the inference algorithm, the path environment – which initially is empty – is extended by new bindings for variables. Consider inference rule FOR I and the for-expression below.

“for $\$x$ in $\$y/child :: a$ return α ”

In applying the inference rule, the new path environment $PEnv' = PEnv \cup \{(\$x, p/child :: a)\}$ is computed, where p is the path to which variable $\$y$ is bound in the current path environment $PEnv$. The new path environment binds variable $\$x$ to path $p/child :: a$. Nested for-loops in our XQ fragment are always single-step loops, thus for any proper prefix path (except the empty path ϵ) of a base path, there exists a variable that is bound to this prefix path in path environment $PEnv'$. We denote the variable associated to a prefix path pp_{pre} of base projection path pp by $v_{pp_{pre}}^{pp}$.

Example 17 (Prefix path variables) Consider the base projection path

$$pa = /child::bib/child::book/child::author$$

from Example 16. Further let $pa_1 = /child::bib$ and $pa_2 = /child::bib/child::book$ be the proper prefix paths of pa . Then $v_{pa_1}^{pa} = \$bib$ and $v_{pa_2}^{pa} = \$book2$. For base projection path

$$pt = /child::bib/child::book/child::title$$

we also have two proper prefix paths, namely the paths $pt_1 = /child::bib$ and $pt_2 = /child::bib/child::book$. Here, $v_{pt_1}^{pt} = \$bib$ and $v_{pt_2}^{pt} = \$book1$.

Although pa_2 and pt_2 are identical, their prefix path variables are different. The reason is that pa_2 was introduced using variable $\$book2$, whereas pt_2 is based on variable $\$book1$ of the origin XQuery expression. \square

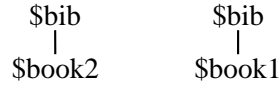
We next formally introduce base path variable trees.

Definition 11 (Base path variable tree) Let p be a base projection path of length n and let p_i be the prefix path of p with length i , where

$$\begin{cases} 1 \leq i \leq n & \text{if } p \text{ describes a variable or a variable output expression} \\ 1 \leq i < n & \text{otherwise} \end{cases}$$

The base path variable tree t is the tree that (1) contains all variables $v_{p_i}^p$ for valid i , and (2) variable $v_{p_m}^p$ is an ancestor of $v_{p_n}^p$ exactly if $m < n$. By $vars_t$ we denote the set of all variables $v_{p_i}^p$ in tree t . \square

Example 18 (Base path variable tree) Let pa and pt be the projection paths from Example 17. For the base path variable tree t_{pa} of projection path pa we have $vars_{t_{pa}} = \{\$bib, \$book2\}$ and $\$bib$ is an ancestor of $\$book2$. For the base path variable tree t_{pt} of base path pt we have $vars_{t_{pt}} = \{\$bib, \$book1\}$ and $\$bib$ ancestor variable if $\$book1$. The variable trees t_{pa} (left side) and t_{pt} (right side) are shown below. Note that each base path variable tree describes a single path and, for this reason, never contains any branches. \square



Definition 12 (Subsumption) Let t_1 and t_2 be two base path variable trees. t_1 is *subsumed* by t_2 if $vars_{t_1} \subseteq vars_{t_2}$. \square

The definition naturally extends to base paths. A base projection path p_1 is *subsumed* by base path p_2 if the base path variable tree t_{p_1} of p_1 is subsumed by the base path variable tree t_{p_2} of p_2 . In the definition above we do not require the parent-child relation in both variable trees to be identical. This requirement automatically holds, because we assume variable names in XQuery expressions to be unique⁹.

Our goal was to compute the set of projection paths starting with set of base projection paths and the query expression given. How to convert existence-check base paths and non-existence-check base paths has been illustrated in Example 16, but the role of variable base paths has not yet been discussed. A variable base path only generate a projection path if it is not subsumed by any other projection path. In this case, a projection path identical to the base projection is added to the set of projection paths.

⁹Unique variable names can be easily obtained by alpha-renaming.

Definition 13 (Projection path reconstruction) Let pp_{nec} , pp_{ec} and pp_v be the sets of base projection paths extracted from the inference rules presented in Subsection 6.4.2. The set pp of projection paths is constructed as follows.

1. For each path $p \in pp_{nec}$ add projection path $p/descendant-or-self::node()$ to pp .
2. For each path $p \in pp_{ec}$ add projection path $p[1]$ to pp .
3. For each path $p \in pp_v$ s.t. p is not subsumed by any path in $pp_{base} = pp_{nec} \cup pp_{ec} \cup pp_v - \{p\}$ add projection path p to pp . □

Following this definition, for Example 16 no variable base path contributes to the set of projection path as all variable paths are subsumed.

1. Base variable path $/child::bib$ is subsumed by both other variable paths, namely both paths labeled $/child::bib/child::book$.
2. The variable path $/child::bib/child::book$ which refers to variable $\$book1$ of the original query shown in Example 14) is subsumed by non-existence-check base path $/child::bib/child::book/child::title$.
3. The variable path $/child::bib/child::book$ in contrast refers to variable $\$book2$ and is subsumed by non-existence-check base path $/child::bib/child::book/child::author$ amongst others.

Example 19 (Projection path reconstruction) Consider the XQuery expression

```
for $bib in /bib return
  for $book in $bib//book return <book/>
```

Running the inference algorithm on this query returns the tuple $(pp_{nec}, pp_{ec}, pp_v)$ where $pp_{nec} = pp_{ec} = \emptyset$ and $pp_v = \{/child::bib, /child::bib/descendant::book\}$. While base projection path $/child::bib$ is subsumed by base path $/child::bib/descendant::book$, the latter base path is not subsumed by any other path. Following Definition 13, the set of projection consists of a single path, namely $pp = \{/child::bib/descendant::book\}$. □

6.4.4 Projection Tree Setup Algorithm

So far, our analysis covers projection paths extraction. We now sketch how the projection tree is constructed based on a set of projection paths. An algorithm for projection tree setup is sketched in Figure 18. After creating a root node in step (1), for each projection path a new branch is added to the root node in step (2). In the final step (3), the algorithm merges together branches that rely on the same variable of the origin XQuery expression.

- (1) Create a new tree t with root node labeled “/”.
- (2) For each projection path p , set up a branch as child of the root node.
The new branch represents the ordered sequence of path steps in p , i.e. for each path step in p a node is constructed and the parent-child relation of t is extended by each pair of successive path step expressions in p .
- (3) Each projection tree node n_i describes a prefix path of a projection path.
In a final step we merge together same-level projection tree nodes. Starting in level 1, we traverse all levels in order. Two projection tree nodes n_i, m_i describing the prefix paths p_{n_i} and p_{m_i} of projection paths p_n and p_m are merged together in t exactly if the following conditions hold.
 - (a) n_i and m_i share the same parent node
 - (b) n_i and m_i carry identical path step expressions and
 - (c) $v_{p_{n_i}}^{p_n}$ equals $v_{p_{m_i}}^{p_m}$.

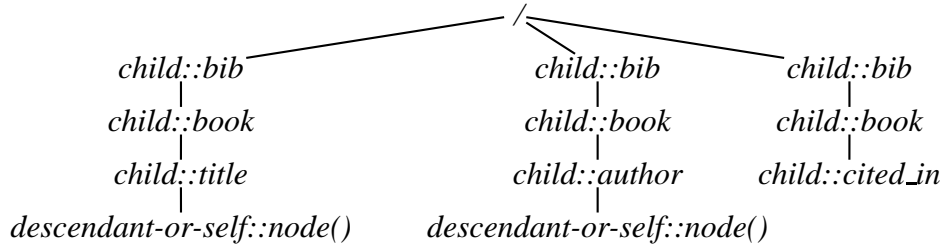
Figure 18: Projection tree setup algorithm.

Example 20 (Projection tree setup algorithm) Consider the set of projection paths extracted in Example 16. The construction of the projection tree is illustrated in Figure 6.4.4.

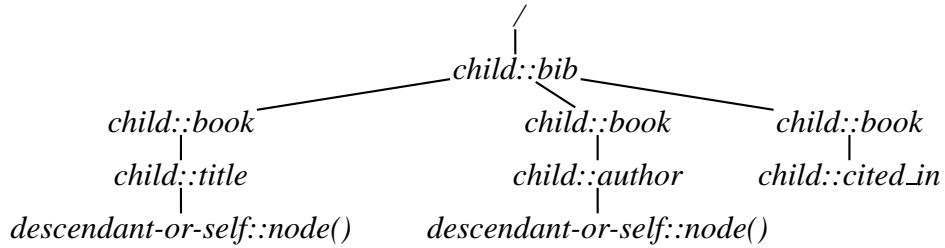
Part (a) illustrates the projection tree immediately after application of step (2) of the algorithm presented in Figure 18. As there exist three projection paths, the root node / contains three branches, one for each projection path. In part (b), the /bib nodes of all three branches have been merged together. All of them share the same parent (namely the root node /), carry the same label /bib and share the common base variable \$bib. The final projection tree has been shown in Example 14. It is obtained by merging the /book nodes of the second and third branch. Note that the /book node of the first branch is not merged: Its base path variable \$book1 differs from base variable \$book2 of the other /book nodes, thus condition (c) is violated. \square

6.5 Projection Tree Determinization

We now describe how the projection tree can be used for efficient stream preprojection and marking. As shown in Subsection 4.1, for each projection tree, there exists a function r_π assigning a unique role to each node. The key idea (which fails for projection trees with descendant axes, as we will see in the following example) is very simple and straightforward. The projection tree – interpreted as finite automaton – can be used to



(a) Projection tree after application of steps (1) and (2)



(b) Partially merged projection tree

Figure 19: Projection tree setup example.

track the incoming stream. We initially start in the root state “/”. When reading token t from the input stream, we distinguish three cases.

1. t is an opening tag
If there exists a transition from the current state to a node n_i such that the path step expression of n_i matches t , then t is buffered together with role $r_\pi(n_i)$ and the current state is changed to state n_i , correspondingly¹⁰.
2. t is a closing tag
In this case, the corresponding buffered tag is marked finished and the previous state is restored.
3. t is character data
If there exists a transition from the current state to node n_i such that the step ex-

¹⁰Note that in some cases the role will be assigned multiple times to a certain node. For simplicity, this topic is ignored here as it is an orthogonal issue and has already been discussed in Subsection 4.2.

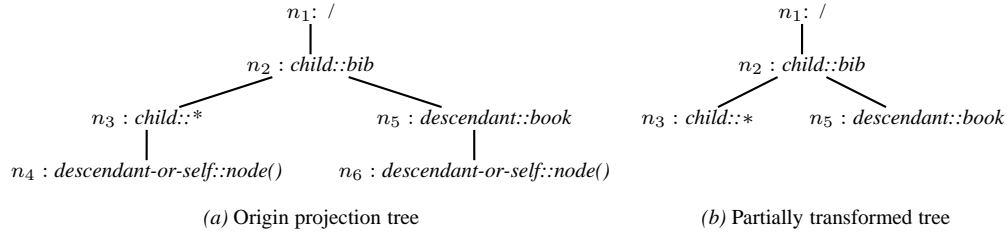


Figure 20: Deterministic projection tree setup.

pression of n_i matches t , then t is buffered together with node role $r_\pi(n_i)$. The state remains unchanged.

While the algorithm works fine for path trees without descendant axis labels and wildcard nodes, the technique fails if the projection tree contains nodes with descendant-axis labels. Moreover, due to sibling tag and wildcard nodes, in some cases we would be required to track different states in parallel. Both problems are discussed in Example 21.

Example 21 (Problems in stream preprojection) Consider the XQuery expression

```
for $bib in $root/bib return
(
  for $x in $bib/* return $x,
  for $b in $bib//book return $b
)
```

and the corresponding projection tree shown in Figure 20(a). Further let r_π be the function mapping each projection tree node n_i to role r_i . We discuss the preprojection of an input stream starting with sequence $\langle bib \rangle \langle book \rangle \dots$

Projection tree nodes are interpreted as states, thus in the following the notions of nodes and states are used interchangeably. We initially start in state n_1 of the projection tree. Tag $\langle bib \rangle$ is matched by state n_2 , consequently we buffer the tag together with role $r_\pi(n_2) = r_2$ and change the current state to n_2 . Next we read tag $\langle book \rangle$, which matches both child nodes of n_2 , namely n_3 and n_5 . This case is not captured by the rules presented above. Intuitively, we should track both states in parallel. In doing so we assign roles r_3 and r_5 and change the current state to the composed state (r_3, r_5) . The next problem arises with the descendant-axis node n_5 . Assume we encounter a nested book somewhere in the current book, this is we read another $\langle book \rangle$ tag before receiving tag $\langle /book \rangle$ for the current book. Obviously, this book tag should be matched by node n_6 , which describes the path $/child::bib/descendant::book$. The problem is that our current state (r_3, r_5) as well as subsequent states discard for matching the tag when strictly following the rules presented above. Descendant-axis nodes would require special handling. \square

Example 21 shows that the type of projection trees introduced so far is hard to use for stream preprojection for two reasons.

- Document paths might be matched by multiple projection tree nodes in parallel, thereby introducing composed states which complicate the implementation.
- Descendant-axis nodes require special care. Intuitively spoken, they should be handled as if they were present in different levels of the original document tree.

The problems above, in particular the special handling of descendant axis nodes, imply other problems, e.g. reconstruction of the previous state might become difficult and time-consuming. We argue that both problems rule out an efficient and clean implementation of stream preprojection. For this reason, we lazily transform the nondeterministic projection tree into a deterministic finite automaton during stream preprojection. Lazy DFA construction is a well-known technique from automata theory and has been successfully employed in [16]. In the following we sketch our technique for projection tree determinization.

6.5.1 Projection Tree Fusion

In the first step of projection tree determinization, we merge together all same-labeled sibling nodes in a top-down traversal of the projection tree. In the origin projection tree, each role was assigned to exactly one projection tree node. The fusion of nodes demands for an extended role assignment function, which maps nodes to a set of roles instead of a single role.

The extended role assignment function r_π^+ is constructed following two rules.

1. For each unmerged node n , the new function r_π^+ is defined as follows.

$$r_\pi^+(n) \stackrel{def}{=} \{r_\pi(n)\}$$

Informally spoken, r_π^+ returns the set containing the former role of node n .

2. When merging together nodes n_{i_1}, \dots, n_{i_j} to a single node $n_{i_1..j}$, the new role assignment function α^+ maps to the aggregated set of roles, i.e.

$$r_\pi^+(n_{i_1..j}) \stackrel{def}{=} \bigcup_{k \in \{1, \dots, j\}} \{r_\pi(n_{i_k})\}.$$

There is one special case that is worth mentioning. Existence-check expressions introduce projection tree nodes with additional attribute “*position()*=1”. As we aim at projection tree determinization, these nodes will be merged with nodes carrying attribute “*true*”.

We use attribute “true” for the merged node, thus drop attribute “ $position()=1$ ”. The information contained in attribute “ $position()=1$ ” instead is decoded in the role carried by the existence-check node¹¹.

6.5.2 Role Aggregation

As described in Definition 13, each non-existence-check base projection path introduces a *descendant-or-self::node()* child node in the corresponding projection path. Such nodes match document subtrees rather than single document nodes, thus the corresponding role is assigned to the document node matching the base projection path as well as to all its descendants.

In our implementation, given non-existence-check role r , we create an aggregated role r^+ and, instead of assigning role r to each matching node n and all its descendants, we simply assign aggregated role r^+ to the single node n . This role minimizing technique simplifies the update mechanism as only the single node which carries the aggregated role needs to be updated. The implementation of this optimization is presented in Definition 14.

Definition 14 (Implementation of role aggregation) We implement role aggregation as follows. For each *descendant-or-self::node()* projection tree leaf node¹² n with role r and parent node n_{parent} , we

1. discard node n , and
2. extend the role assignment function r_π^+ as follows.

$$r_\pi^+(n_{parent}) := r_\pi^+(n_{parent}) \cup \{r^+\}$$

□

Note that the introduction of aggregated roles induces some minor changes to the *signOff*- and *gc*-algorithms, as role updates will be sent to the aggregated, role carrying node only. These changes are straightforward and will not be further discussed here.

Example 22 (Role aggregation) Application of the role aggregation rules to the projection tree derived from the query in Example 21 (Figure 20(a)) yields the projection tree in Figure 20(b). The associated role assignment function r_π^+ contains the mappings $n_1 \mapsto \{r_1\}$, $n_2 \mapsto \{r_2\}$, $n_3 \mapsto \{r_3, r_4^+\}$ and $n_5 \mapsto \{r_5, r_6^+\}$. □

¹¹This can simply be realized via a function that returns *true* for all existence-check roles, false otherwise.

¹²Note that these nodes always are leaf nodes.

6.5.3 Lazy DFA Construction

The projection tree resulting from the previous subsection still is deterministic. Indeed same-labeled nodes have been merged, but there still remain three scenarios causing problems for a straightforward stream preprojection algorithm that interprets the tree in an automaton-like manner.

- Wildcard nodes may be matched in parallel to tag nodes.
- Descendant-axis nodes may be matched in parallel to non-descendant-axis nodes.
- Descendant-axis nodes may be matched in different levels.

In order to give an intuition on how to construct the deterministic projection tree, we describe the key ideas informally and by examples. The algorithm, which is rather technical and contains many special cases, does not contribute to the understanding of the techniques and is omitted.

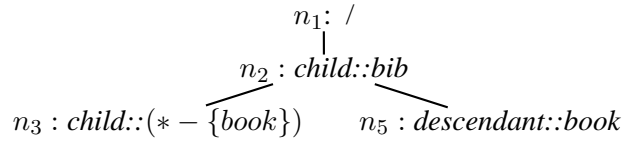
Our strategy for removing the nondeterminism caused by wildcard nodes is rather simple. The problem with wildcard nodes is that they are matched in parallel to same-level tag nodes in the projection tree. Assume for example there exists a wildcard node labeled *child::** with two sibling non-wildcard nodes, say *child::author* and *child::title*. The wildcard node then is relabeled to “*child::(* – {author, title})*”, expressing that the node now matches every tag besides author and title tags. Coming along with this rewriting, we have to make the following adjustments.

1. The roles of the wildcard node are added to the *child::title* and *child::author* node role set.
2. All child subtrees of the wildcard node are added as subtrees of both the *child::title* and *child::author* nodes¹³.

Nodes *child::title* and *child::author* now represent composed states. A node that matches *child::title* gets assigned the former roles of the title node as well as the former roles of the wildcard node, as they were added to the role set of the title node. Moreover, after matching the title node, we can match all children of the title node as all subtrees of the wildcard have been copied to the title node.

Example 23 (Wildcard node transformation) Consider the projection tree from Figure 20(b) with role assignment function r_{π}^{+} as defined in Example 22. The wildcard transformation rules transform the projection tree as follows.

¹³This step might introduce same-labeled children, thus merging is required again.



The new role assignment function is described by the mappings $n_1 \mapsto \{r_1\}$, $n_2 \mapsto \{r_2\}$, $n_3 \mapsto \{r_3, r_4^+\}$ and $n_5 \mapsto \{r_3, r_4^+, r_5, r_6^+\}$. Note that the wildcard node roles were added to the set of book node roles. \square

The elimination of the problems introduced by descendant-axis nodes is more sophisticated. The key idea is to throw away all axes from the projection tree. Coming along with this transformation, descendant-axis nodes are distributed within the subtree spanned by the former descendant node, so that they can be matched in every sub level. The role assignment function is adapted accordingly, i.e. roles are distributed within the subtree as well. The resulting deterministic projection tree becomes infinite and, for this reason, is computed lazily during stream preprojection. Nodes that are matched during stream preprojection are unfolded on demand, thus the size of the deterministic projection tree is limited by the structure of the document. Once a branch has been unfolded, it can be matched by identical input document paths without recomputation. Although there is a computational overhead for small documents, this technique behaves very well in streaming scenarios, where, in general, we have to deal with large input documents [16]. Moreover, as the structure of input documents is homogeneous in practice^{d4}, the size of the deterministic automaton remains acceptably small.

Example 24 (Deterministic projection tree) Figure 21 shows the deterministic projection tree derived from the projection tree introduced in Example 22 and shown in Figure 20(b). The left side of the projection tree has been unfolded up to level four, the right side up to level three (level four is structurally equal to the left side). Nodes labeled with “...” have not yet been computed. The role association function r_π^+ for the deterministic projection tree contains the mappings

$$\begin{array}{llll}
 n'_1 \mapsto \{r_1\}, & n'_4 \mapsto \emptyset, & n'_7 \mapsto \{r_5, r_6^+\}, & n'_{10} \mapsto \{r_3, r_4^+, r_5, r_6^+\}, \\
 n'_2 \mapsto \{r_2\}, & n'_5 \mapsto \emptyset, & n'_8 \mapsto \emptyset, & n'_{11} \mapsto \emptyset, \\
 n'_3 \mapsto \{r_3, r_4^+\}, & n'_6 \mapsto \{r_5, r_6^+\}, & n'_9 \mapsto \{r_5, r_6^+\}, & n'_{12} \mapsto \{r_5, r_6^+\}
 \end{array}$$

, where the roles r_i are the same roles as used in Example 22.

The deterministic projection tree does not carry any axis information. The former *//book* node has been distributed within all subtrees. Once we are in a level where the descendant is matchable, we can either match node “*book*” (i.e. when reading a book tag), or we match its sibling node “** - {book}*” (i.e. when reading a tag different from book), thereby entering the next deeper level where node “*book*” can be matched again. \square

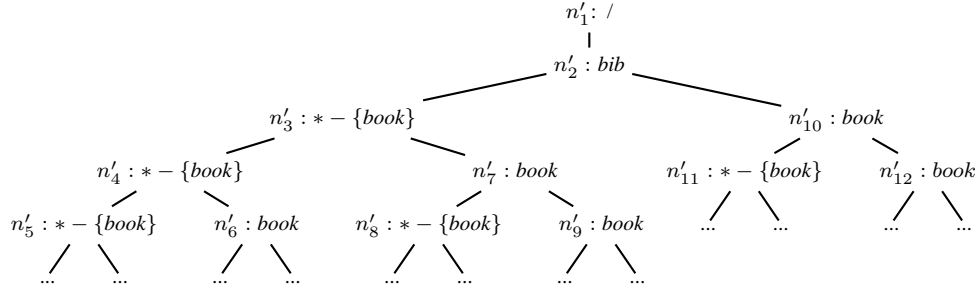


Figure 21: Deterministic projection tree.

Once more, instead of covering the exact rules of stream preprojection, we conclude the discussion with a detailed example that illustrates the ideas of stream preprojection and role assignment using a deterministic projection tree.

Example 25 (Stream preprojection and role assignment) We consider the deterministic projection tree from Figure 21 and the associated set of roles shown in Example 24. In the following we describe the process of preprojecting and marking following XML stream.

```

<bib>
  <book>
    <title>"Book Title"</title>
    <author>"Book Author"</author>
  </book>
  <article>
    <title>"Article Title"</title>
    <author>"Article Author"</author>
    <cite>
      <book>"Book Title"</book>
    </cite>
  </article>
</bib>

```

We initially start in state n'_1 of the projection tree. When reading tag $\langle bib \rangle$, we match child node n'_2 . The $\langle bib \rangle$ tag is buffered together with role set $r_\pi^+(n'_2) = \{r_2\}$ and the current state is changed to n_2 . Next tag $\langle book \rangle$ is matched by state n'_{10} and buffered together with roles $r_\pi^+(n'_{10}) = \{r_3, r_4^+, r_5, r_6^+\}$. Subsequently the $\langle title \rangle$ is read, which matches node n'_{11} . In principle, we can discard document nodes that match nodes with

¹⁴This is often implied by DTD or XML schema information [50] restrictions.

empty role-set¹⁵, but node n'_{10} carries an aggregated role. Recall that aggregate roles are “inherited” to the whole subtree, thus we have to buffer the $\langle title \rangle$ tag with the empty role set. The current state is changed to n'_{11} . The character data “Book Title” is buffered for the same reason. Note that processing character data does not change the current state. When reading closing tag $\langle /title \rangle$, the buffered title is marked finished and the current state is reset to n'_{10} . The author is processed analogously to the title: Everything is buffered with the empty role set and we end up in state n'_{10} again. We next read the closing tag $\langle /book \rangle$. The buffered book is marked finished and the current state is reset to parent state n'_2 . In the following we buffer tags $\langle article \rangle$ (matching state n'_3), $\langle title \rangle$ (matching state n'_4) and corresponding character data as well as tag $\langle author \rangle$ with character data. Having processed the closing tag $\langle /author \rangle$, we end up in state n'_3 . Next tag $\langle cite \rangle$ is read, matched by projection tree node n'_4 and buffered with empty role set. The following $\langle book \rangle$ tag is matched by node n'_6 and, consequently, buffered with role set $r_\pi^+(n'_6) = \{r_5, r_6^+\}$. Note that this role set is different from the role set assigned to the former book. When the whole stream has been processed, we end up in state n'_1 again. \square

In the previous example, nothing could be projected away. The reason is that in the original query (see Example 21), the for expression “for $\$x$ in $\$bib / * return \x ” forces us to keep all children of the bib node. In general, during stream preprojection we can discard any token that is (1) not matched by a projection tree node or (2) matched by a projection tree node with empty role set and, for both cases (1) and (2) it must be valid that there exists no ancestor node which carries an aggregated role.

6.6 Syntax and Expression Implementation

The syntax used for implementation, which is strongly related to the abstract syntax presented in Figure 5, is shown in Appendix A. Note that we allow brackets only where absolutely necessary, i.e. for sequence expressions and for conditions using *and*, *or* and *not*. It is also worth mentioning that both *and* operators and *or* operators bind identically strong. We further require variable names to be unique and introduce the variable $\$root$, the only free variable in a XQ expression, which is initially bound to the document root.

The UML class concept for expressions is illustrated in the Appendix. For readability reasons, it has been divided into two parts. While Appendix B illustrates the implementation of conditional expressions, non-conditional expressions are presented in Appendix C.

All expressions have an abstract base class `Expression` that defines a common set of virtual functions used within expressions and, for most of them, offers a default implementation. The most interesting functions are

¹⁵These nodes result from the descendant-axis rewriting roles and, intuitively spoken, their only purpose is to keep track of the current document tree level.

- *void eval(Environment* env, unsigned modus)*
evaluates the expression within a given environment. The modus is used to restrict evaluation to nested *signOff*-statements and addresses the implementation approach presented in Subsection 4.3.1, i.e. the topic of if-statements nested inside for-loops.
- *bool scopeCheck()*
checks for variable scopes being valid
- *void extractProjectionPath(ProjectionPaths* pp)*
extracts the projection paths from an expression and fills the passed class *pp*
- *void extractVarTree(VarTree* vt)*
extracts the variable tree set up by the structure of the expression
- *extractFSAMap(FSAMap* fsamap)*
extracts variable to first straight ancestor mappings

The `Expression` class directly inherits all non-conditional expressions. Conditional expressions themselves have an abstract base class `CondExpression`. This class introduces a virtual function *bool evalCond(Environment* env)* which is used to evaluate conditional expressions. The class `CondExpression` itself inherits from the `Expression` class. Every conditional expression, except node-node and node-value test expressions, inherit directly from `evalCond`. Node-node and node-value test expressions in contrast share the common base class `NodeTestCondExpression` (inherited from `evalCond`), which provides the function for relational operator evaluation.

6.7 Optimizations

We finally present some optimizations that contribute to both space and runtime improvements. All optimizations presented in the following are also implemented in our prototype. We refer to a discussion of unimplemented optimization techniques in Section 8.

6.7.1 Elimination of Redundant Roles

Immediately after base projection path computation, we can drop redundant roles from the corresponding role assignment function. The elimination of redundant *sendUpdate*-statements comes along with this step, consequently query evaluation speeds up. The elimination applies to roles implied by variable base paths. Using the notion of subsumption introduced in Definition 12, many roles generated by subsumed variable base paths can be dropped¹⁶. Note that the implementation requires some minor changes in the buffer update, stream preprojection and query rewriting algorithm that are not discussed here.

¹⁶We skip the details, which are rather technical.

Example 26 (Role elimination) Consider the set of variable base paths in Example 16. We can drop all roles implied by these base paths, as all three variable paths are subsumed by at least one base projection path. For the corresponding path tree shown in Example 14, roles r_2 , r_3 and r_6 for the nodes n_2 , n_3 and n_6 can be dropped. \square

6.7.2 Tag Name Hashing

During stream preprojection we construct a bidirectional hash table mapping tags to integers and vice versa. Instead of the data we simply store the identifying integer for each buffered tag node. Although there is small computational overhead in the stream preprojection phase, this common technique saves space and speeds up query evaluation as node-node comparisons can be implemented via fast integer equality tests.

6.7.3 Pushing Down *signOff*-Statements

Query preprocessing can improve the effectiveness of garbage collection. In the rewritten query from the introduction, the command

```
signOff($book/title/descendant-or-self::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. This can be easily avoided by rewriting all output expressions “ $\$x/\sigma$ ” contained in the original query to equivalent expressions “for $\$y$ in $\$x/\sigma$ return $\$y$ ” where $\$y$ is a new variable. The rewritten query is shown below.

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

The corresponding *signOff*-command now will be pushed into the new for-loop that introduces variable $\$b'$. Updates will be sent immediately after each title.

7 Experimental Results

We have experimentally evaluated our prototype implementation. In this section we present the queries used for correctness tests and discuss benchmarking results for both stream preprojection and query evaluation.

7.1 Correctness Tests

We experimentally verified correctness of our system using miscellaneous queries from different projects. All evaluation results were compared to the results delivered by the XQuery reference implementation Galax [14]. In detail, the following queries were used for correctness tests. Note that some queries were slightly adapted as our system does not support the full XQuery standard.

- **W3C “XMP” XQuery usecases**

The queries from the W3C “XMP” XQuery usecases [46] are provided in Appendix D.1. All queries were evaluated on the original usecase XML documents.

- **DBMS milestone test queries**

Furthermore we used queries from the Saarland University DBMS course held in summer term 2005 [39, 40]. The set of verified queries is presented in Appendix D.2.

- **XMark benchmarking project**

We verified correctness for the XMark queries used in our benchmarking tests (Subsection 7.3). The set of queries in XQ syntax is shown in Appendix E.

- **User defined queries**

Last but not least we used a set of user defined queries to check the correctness of crucial system components. The query collection focuses on queries with descendant axes and the assignment of multiple role instances. All user defined queries are listed in Appendix D.3.

Moreover, for all test queries we verified the safety requirement stated in Definition 5. In all cases, the buffer was empty after query execution and no undefined role updates were sent during query evaluation.

7.2 Stream Preprojection

We have experimentally evaluated our stream preprojector and have compared results to other stream preprojection techniques.

- We compared our system to the stream preprojection technique proposed in [27] and implemented in Galax [14]. Unfortunately, we could not get Galax stream preprojection to work, yet the computation of the preprojection paths succeeded. We manually applied the projection paths on the input document, strictly following the rules presented in [27].
- We also compared our results to the type-based stream preprojection system presented in [5], which is freely available for download [4].

Unfortunately, the type-based projection implementation only offers support for single XPath expressions rather than XQuery. Path extraction on XQuery expression requires manual extraction of all XPath expressions described by the query. For queries containing more than one XPath expression, we evaluated all XPath expressions in sequence and computed the resulting document by taking the union of all computed documents. Furthermore, type-based projection does not support position attributes, which are useful when dealing with existence check projection paths, where only the first witness of a node needs to be buffered. Consequently, all XPath position attributes were rewritten to attribute “*true*”. While XPath 1.0 [47] lacks of support for discarding subtrees, the prototype offers a flag “*-nosub*” which enables for projecting away unneeded subtrees. In our tests, the flag was set wherever possible.

Tests were carried out for a set of five user defined queries which cover different aspects of stream preprojection. The test queries in XQ syntax are shown in Figure 22. In all cases, the queries were evaluated against the XML stream shown in Figure 23. Table 1 shows the results of stream preprojection¹⁷.

*Q*₁. This query can be easily projected using standard stream preprojection techniques. Only a single path, i.e. */child::book/child::title* and all its descendants, needs to be extracted, everything else can be thrown away. All systems were able to compute the minimal projected document.

*Q*₂. The second query also contains a single projection path, */child::book/child::author*. As the author is not used for output, in contrast to *Q*₁, all subtrees can be thrown away. Consequently, type-based projection was executed using flag “*-nosub*”. Even here all systems compute the minimal projected document.

*Q*₃. The third query addresses descendant processing. Here, a smart implementation might throw away single path steps, e.g. the *<bib>* tag and its closing tag can be thrown away as not required for query evaluation. It becomes clear from Table 1 that our system

¹⁷Whitespaces, line breaks and tabs are ignored in our discussion but used in Table 1 and Figure 23 for readability reasons.

Q_1 :

```
<q1>
for $book in $root/book return
  for $title in $book/title
    return $title
</q1>
```

Q_2 :

```
<q2>
for $book in $root/book return
  for $author in $book/author
    return <author_match></author_match>
</q2>
```

Q_3 :

```
<q3>
for $section in $root//section return
  <section></section>
</q3>
```

Q_4 :

```
<q4>
for $book in $root/book return
  if (exists $book/author)
  then $book/title
  else ()
</q4>
```

Q_5 :

```
<q5>
if (exists $root//section)
then <yes></yes>
else <no></no>
</q5>
```

Figure 22: Stream preprojection test queries.

```

<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
  <section>
    <title>Introduction</title>
    <p>T1</p>
    <section>
      <title>Audience</title>
      <p>T1</p>
    </section>
    <section>
      <title>Web Data and the Two Cultures</title>
      <p>T2</p>
      <figure>
        <title>Traditional client/server architecture</title>
        <image/>
      </figure>
      <p>T2</p>
    </section>
  </section>
  <section>
    <title>A Syntax For Data</title>
    <p>T1</p>
    <figure>
      <title>Graph representations of structures</title>
      <image/>
    </figure>
    <p>T1</p>
  </section>
  <section>
    <title>Base Types</title>
    <p>T1</p>
  </section>
</book>

```

Figure 23: XML stream used for preprojection tests.

	GCX	Galax (Expected Result)	Type-Based Projection
Q1	<code><book></code> <code><title>Data on the Web</title></code> <code></book></code>	<code><book></code> <code><title>Data on the Web</title></code> <code></book></code>	<code><book></code> <code><title>Data on the Web</title></code> <code></book></code>
Q2	<code><book></code> <code><author></author></code> <code><author></author></code> <code><author></author></code> <code></book></code>	<code><book></code> <code><author></author></code> <code><author></author></code> <code><author></author></code> <code></book></code>	<code><book></code> <code><author></author></code> <code><author></author></code> <code><author></author></code> <code></book></code>
Q3	<code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code>	<code><book></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code></book></code>	<code><book></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code></book></code>
Q4	<code><book></code> <code><title>Data on the Web</title></code> <code><author></author></code> <code></book></code>	<code><book></code> <code><title>Data on the Web</title></code> <code><author></author></code> <code><author></author></code> <code><author></author></code> <code></book></code>	<code><book></code> <code><title>Data on the Web</title></code> <code><author></author></code> <code><author></author></code> <code><author></author></code> <code></book></code>
Q5	<code><section></section></code>	<code><book></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code></book></code>	<code><book></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code><section></code> <code><section></section></code> <code><section></section></code> <code></section></code> <code></book></code>

Table 1: Stream preprojection results.

is the only one that manages to apply this optimization¹⁸.

Q_4 . This query contains an existence check expression. For each book, only the first occurrence of the `<title>` has to be buffered. While our system applies this optimization, the other systems unnecessarily keep all author nodes.

Q_5 . The final query combines the descendant and the existence-check scenario. GCX supports the combination of both techniques and computes the minimal projected document.

¹⁸Although the document root node `bib` is projected away, the result still can be interpreted as a sequence of two XML streams which can be evaluated in order.

The examples demonstrate the power of our stream preprojection algorithm. While the computation of the minimal projected document in a single pass is undecidable in general¹⁹, our system managed to compute the minimal projected document for all example queries.

7.3 Evaluation Time and Memory Consumption

In order to assess the merits of active garbage collection, our implementation GCX was experimentally evaluated using a number of XMark [53] queries. Our prototype was implemented exactly as described in this paper. As the XQ fragment introduced in Section 3 does not cover the full XQuery standard, queries were adapted correspondingly. In detail, we converted XML attributes into subelements and replaced aggregations such as *count(\$x)* by outputting the value of *\$x* instead. *Q20* is identical to *Q20* from [13]. We considered XMark documents of sizes between 10MB and 200MB, generated with the XMark data generator. Tests 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** [41] (Java based),
- **FluXQuery** [13] (Java based), and
- **QizX/open v1.1** [37] (Java based).

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** [28], which relies on secondary storage.

Finally, we used the in-memory XQuery engine **Galax v0.6.5** [14], 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 [27] 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

¹⁹Undecidability immediately follows from the undecidability of conditional expression, i.e. we cannot decide whether projection paths encapsulated into if-statements will be accessed in the future.

Query		GCX	FluXQuery	Galax	MonetDB	Saxon	QizX
XMark Q1	10MB	0.18s / 1,2MB	1.64s / 91MB	2.59s / 52MB	0.87s / 22MB	1.55s / 33MB	1.20s / 39MB
	50MB	0.91s / 1,2MB	4.16s / 130MB	17.96s / 260MB	4.87s / 98MB	4.53s / 194MB	3.52s / 150MB
	100MB	1.83s / 1,2MB	7.22s / 130MB	42.44s / 529MB	7.25s / 225MB	7.51s / 372MB	6.49s / 263MB
	200MB	3.49s / 1,2MB	12.80s / 130MB	2:16.92 / 982MB	13.30s / 268MB	14.29s / 105MB	11.76s / 489MB
XMark Q6	10MB	0.15s / 1,2MB	n/a	2.57s / 57MB	0.71s / 29MB	1.36s / 61MB	1.14s / 30MB
	50MB	0.77s / 1,2MB	n/a	17.45s / 261MB	3.51s / 96MB	4.25s / 253MB	3.70s / 190MB
	100MB	1.54s / 1,2MB	n/a	43.44s / 512MB	7.13s / 222MB	7.63s / 572MB	6.54s / 233MB
	200MB	2.93s / 1,2MB	n/a	2:08.74 / 975MB	13.01s / 243MB	13.97s / 1,0GB	12.08s / 489MB
XMark Q8	10MB	9.52s / 11MB	16.93s / 150MB	1:15.42 / 108MB	1.00s / 22MB	5.68s / 168MB	1.48s / 41MB
	50MB	3:49.41 / 47MB	6:19.76 / 190MB	38:07.25 / 505MB	4.71s / 119MB	1:46.29 / 385MB	4.94s / 230MB
	100MB	15:07.93 / 94MB	25:39.48 / 240MB	timeout	9.40s / 292MB	7:03.58 / 640MB	08.95s / 406MB
	200MB	timeout	timeout	timeout	17.47s / 366MB	25:43.46 / 1,1GB	16.24s / 628MB
XMark Q13	10MB	0.18s / 1,2MB	1.55s / 68MB	2.95s / 55MB	0.81s / 31MB	1.54s / 60MB	1.62s / 40MB
	50MB	0.82s / 1,2MB	4.06s / 130MB	17.50s / 259MB	3.64s / 97MB	4.50s / 345MB	4.03s / 125MB
	100MB	1.64s / 1,2MB	7.05s / 130MB	42.53s / 523MB	7.38s / 225MB	8.07s / 503MB	6.94s / 335MB
	200MB	3.34s / 1,2MB	12.54s / 130MB	2:05.70 / 998MB	13.36s / 244MB	15.45s / 1,05GB	14.19s / 489MB
XMark Q20	10MB	0.27s / 1,2MB	1.61s / 54MB	3.33s / 65MB	0.85s / 29MB	1.65s / 82MB	1.42s / 45MB
	50MB	1.20s / 1,2MB	4.30s / 130MB	22.59s / 294MB	4.17s / 111MB	5.00s / 345MB	4.16s / 231MB
	100MB	2.43s / 1,2MB	7.55s / 130MB	53.71s / 585MB	8.49s / 243MB	9.03s / 572MB	8.68s / 403MB
	200MB	4.69s / 1,2MB	13.52s / 130MB	2:24:50 / 1,1GB	15.54 / 294MB	21.36s / 1,1GB	19.70s / 622MB

Table 2: Benchmark results.

Linux *top* command. For each system and query we set a timeout of 1 hour. Figure 2 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. 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).

The experimental results (Table 2) 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. The gap significantly increases with document size.

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 very well w.r.t. main memory consumption and execution time and also show that, for a large class of queries, we can even beat query engines which exploit schema information [23].

8 Future Work

8.1 Standard Database Techniques

Our system uses simple and straightforward operator implementation. Search is realized via simple buffer scans and joins are implemented as nested loop joins. We could significantly improve query runtime when using standard database optimization techniques. Instead of simple nested loop joins we should use *advanced join techniques* like hash joins or index based joins. Even other common database techniques like, e.g. join reordering and query rewriting based on cost estimation, might improve performance.

Another strategy would be to extend the stream preprojector to *set up indices on-the-fly*. Consider for example projection paths implied by variables introduced in for loops. Each node the variable will be bound to during evaluation is recognized by the preprojector, as it matches the associated projection path. The construction of indices therefore can be easily embedded in the current system.

Last but not least, *on-the-fly evaluation* should be embedded in our system. Although early buffer cleanup seems to compensate on-the-fly evaluation for many practical queries, outputting parts of the input stream on-the-fly, where possible, can still improve both evaluation performance and buffer minimization.

Note that our system primarily aims at low main memory consumption. Techniques like advanced join implementation and indexing to some degree contradict the dogma of streaming XQuery evaluation insofar as they require additional space to maintain data structures, e.g. hash tables or indices. The decision what to use and where to use indices is always a trade-off between runtime and space efficiency and should be handled with care, in order to maintain the benefits of low memory consumption.

8.2 Exploiting Schema Knowledge

Query optimization based on schema knowledge has been subject to extensive studies [5, 23, 25]. Document type definitions [9] (DTDs) or XML Schemata [50] can help to improve both memory consumption and runtime. So far, our system uses no schema information.

DTDs imply structural constraints on the document. A document is said to be *valid* w.r.t. a given DTD if it satisfies all constraints imposed by the DTD.

Example 27 (DTDs) Consider DTD D presented in Figure 24. Each book in a document that is valid w.r.t. D is restricted to consist of exactly one title, followed by a set of authors, followed by a set of references, followed by a single price tag. Moreover, these tags are required to occur in order, e.g. for each book authors will be listed before reference tags. \square

```

<!ELEM root          (bib) >
<!ELEM bib           (book|article) >
<!ELEM book          (title, (author)*, (reference)*, price) >
<!ELEM article       (title, (author)*, (reference)*, journal) >
<!ELEM title         #PCDATA >
<!ELEM author        #PCDATA >
<!ELEM reference     #PCDATA >
<!ELEM title         #PCDATA >
<!ELEM price         #PCDATA >
<!ELEM journal       #PCDATA >

```

Figure 24: DTD *D*.

8.2.1 Reducing Memory Consumption

The FluXQuery system for example exploits DTD order information to decrease the amount of buffering. We demonstrate the key idea in Example 28.

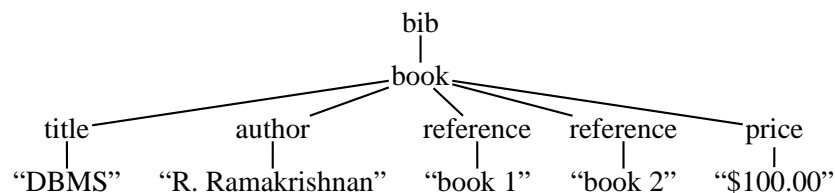
Example 28 (Exploiting DTD order constraints) Consider the following XQuery expression.

```

<result> {
for $bib in /bib return
  for $book in $bib/book return
    (
      for $author in $book/author return
        $author,
      for $reference in $book/reference return
        $reference
    )
} </result>

```

The query extracts every author followed by every reference for each book child of the bib tag. Consider its evaluation on the following streamed input document.



First assume there is no DTD available. Tags `bib` and `book` need to be buffered as required for navigation. The author can be discarded immediately after output. When next reading the first reference tag, we are not yet allowed to output it. This is because there still might exist other book authors, which had to be output before. The same reasoning holds for the second reference tag. Finally when reading $\langle/book\rangle$ we can be sure there is no more author child. We can output the buffered references and purge the buffer immediately after.

Now assume we evaluate the query on a document satisfying DTD D (Figure 24). D implies that, for each book, authors will be listed before references. Thus, when reading the first reference tag, it is guaranteed that no more author for the current book exists. Consequently when evaluating the query above, both authors and references can be output on-the-fly. The amount of buffered data decreases. \square

FluXQuery uses static DTD and query analysis to detect situations as described in the example above. The query language finally is extended by constructs for on-the-fly evaluation and, based on the results of static analysis, the query is rewritten accordingly before execution. The FluXQuery system offers no support for descendant axes and it is not clear whether FluXQuery can be extended accordingly. The question whether – and in which extension – the ideas of FluXQuery can be incorporated in our approach has to be investigated.

8.2.2 Accelerating Query Evaluation

In many situations, DTDs can also be used to speed up query evaluation. Optimizations capture both stream preprojection and query evaluation. In the following we sketch some of these optimization approaches.

DTD-Based Stream Preprojection. The stream preprojector matches a set of projection paths against the incoming stream, thereby filtering out unmatched document paths. Here, the knowledge of a DTD can be very useful to exclude unmatchable paths. Initially, each projection path is checked to be valid w.r.t. to the DTD. Invalid paths can be thrown away as they will never be matched by a valid document.

While the latter optimization is purely static, even during stream preprojection we might run into situations where projection paths can be excluded. If there exist projection paths with descendant-axis path steps, at some point in input stream processing we might detect that the descendant is unmatchable in the current subtree, thus the projection path can be excluded for this subtree. Reducing the number of projection paths early on reduces the search space and finally speeds up query evaluation.

Example 29 (DTD-based stream preprojection) Consider the DTD from Figure 24. Further assume a query with the following set of existence-check and variable projection paths.

$$pp = \{/bib, /bib/book, /bib/book/section, //price\}$$

By static DTD analysis we can exclude projection path */bib/book/section* a priori: This path will never be matched by any document valid w.r.t. to the given DTD as no book contains section tags. Assume the stream starts with $\langle bib \rangle \langle book \rangle$. Projection paths “*/bib*” and “*/bib/book*” are matched and, for the current subtree, the only projection path that still can be matched is the path “*//price*”. When next reading an article, we can – exploiting DTD information – also be sure that the “*//price*” projection path will not be matched in the current subtree, as only books carry price children. The article and all its descendants can be projected away without any more checks. \square

Accelerating Buffer Scans. Even for query evaluation, DTD knowledge can be very useful. The computation of new bindings for variables requires search on the buffer, i.e. we have to scan the buffer for nodes matching a path step relative to a given node. This problem is very similar to stream preprojection as described in the previous paragraph. In the preprojection scenario we have to match a (set of) path(s) against the document. Runtime search in principle is a special case, that requires to match single pathsteps instead²⁰.

Example 30 (DTD-based runtime search improvement) Consider the XQuery expression

```
for $x in //price return $price
```

Without schema information, the computation of the next binding for $\$x$ requires to scan the complete buffer. Each tag has to be checked to be a “price” tag. A high number of equality checks can significantly slow down query evaluation.

Using DTD information, we often can get rid of runtime checks. Consider the evaluation of the query against a document valid w.r.t. DTD D from Figure 24. Assume we read an $\langle article \rangle$ while scanning the buffer for the next binding for variable $\$x$, this is scanning the buffer for a price tag. D implies that articles carry no price descendants, thus we can skip all subtree nodes, resuming search with the next sibling of the article. \square

8.3 Incorporating Aggregation

Currently our system does not support XQuery aggregation operators. While the corresponding extension of the XQ fragment is straightforward, one challenge would be to embed aggregation in our buffer management scheme.

²⁰Our fragment only supports single step path expressions.

8.4 Role Representation

The safety requirement stated in Definition 5 enforces that exactly as many instances of roles are assigned to document nodes as are removed during query evaluation. In our implementation, we currently assign sets of roles to nodes. Similar to garbage collection by reference counting, we could instead use a single role counter for each node, which keeps track of the current number of roles assigned to the node. It is decreased by one each time a node receives a role update. Once the role counter reaches zero, the safety requirement ensures that the node has lost all its roles.

This technique accelerates the role update mechanism: Instead of set operations we only have to deal with simple counter increment and decrement operations. More important, the amount of memory required for saving the role information is decreased. Instead of maintaining a set of roles for each node²¹, we simply associate a single integer counter to each node. This optimization might significantly reduce the amount of main memory consumption when a large number of nodes needs to be buffered at a time.

²¹Roles are internally stored as integers.

9 Conclusion

As shown in the experimental results, our combined static and dynamic buffer management scheme behaves very well for many practical XQuery expressions. We conclude that the combination of static and dynamic buffer management is crucial for the good performance regarding both space and time efficiency.

We think that our system, which relies on ideas very similar to garbage collection by reference counting, follows an intuitive paradigm. In static analysis, we project away irrelevant parts of the input stream. Tokens that pass the projection phase get assigned a set of roles before they are loaded into the buffer, this way preparing garbage collection at runtime. The well-established technique of lazy DFA construction is used for stream preprojection and, once more, proves to be very efficient for practical queries as given in the XMark benchmark project. Runtime overhead in the preprojection phase is very low whereas the benefit of stream preprojection is high.

A single buffer tree ensures that nothing is buffered twice during evaluation. Moreover, the buffer tree structure is minimal, with only parent-child and next sibling pointers. Even without any index structures, for most queries this buffer representation proves to be very efficient. The lack of index structures in most cases is compensated by fast buffer scans, as the buffer solely contains evaluation-relevant data.

Active garbage collection at runtime, an approach very similar to garbage collection by reference counting, ensures that ‘tokens that have become irrelevant for future evaluation, are purged from the buffer early on. This technique significantly contributes to the goal of low memory consumption throughout the whole query evaluation process.

A XQ Concrete Syntax

Start	::=	Exp
Exp	::=	EmptyExp ConstExp SequenceExp ForExp VarStepExp VarExp NodeConstructExp IfExp
EmptyExp	::=	()
ConstExp	::=	s
SequenceExp	::=	(ExpList)
ExpList	::=	Exp Exp , ExpList
VarExp	::=	\$var
ForExp	::=	<i>for</i> VarExp <i>in</i> VarStepExp <i>return</i> Exp
NodeConstructExp	::=	<t/> <t></t> <t> Exp </t>
IfExp	::=	<i>if</i> CondExp <i>then</i> Exp <i>else</i> Exp
VarStepExp	::=	VarExp PathStepExp
PathStepExp	::=	axis :: t axis :: *
axis	::=	<i>child</i> <i>descendant</i>

```

CondExp      ::=  CondNegExp
               |  CondNegExp or CondExp
               |  CondNegExp and CondExp

CondNegExp   ::=  CondElemExp
               |  not CondElemExp

CondElemExp  ::=  ( CondExp )
               |  true
               |  exists VarStepExp
               |  VarExp relop "s"
               |  VarStepExp relop "s"
               |  VarStepExp relop VarStepExp
               |  VarStepExp relop VarExp
               |  VarExp relop VarStepExp
               |  VarExp relop VarExp

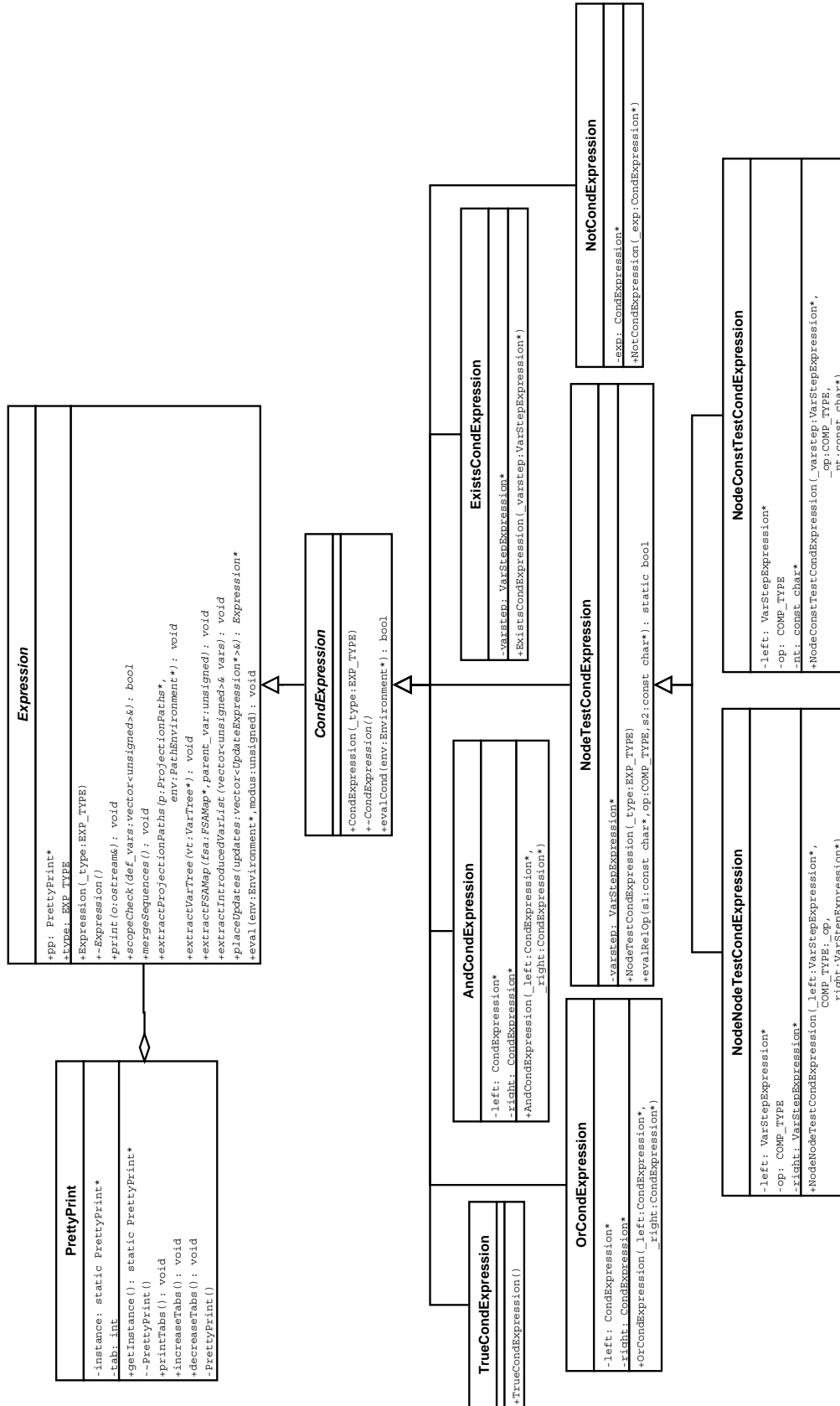
relop        ::=  <
               |  <=
               |  =
               |  >=
               |  >

```

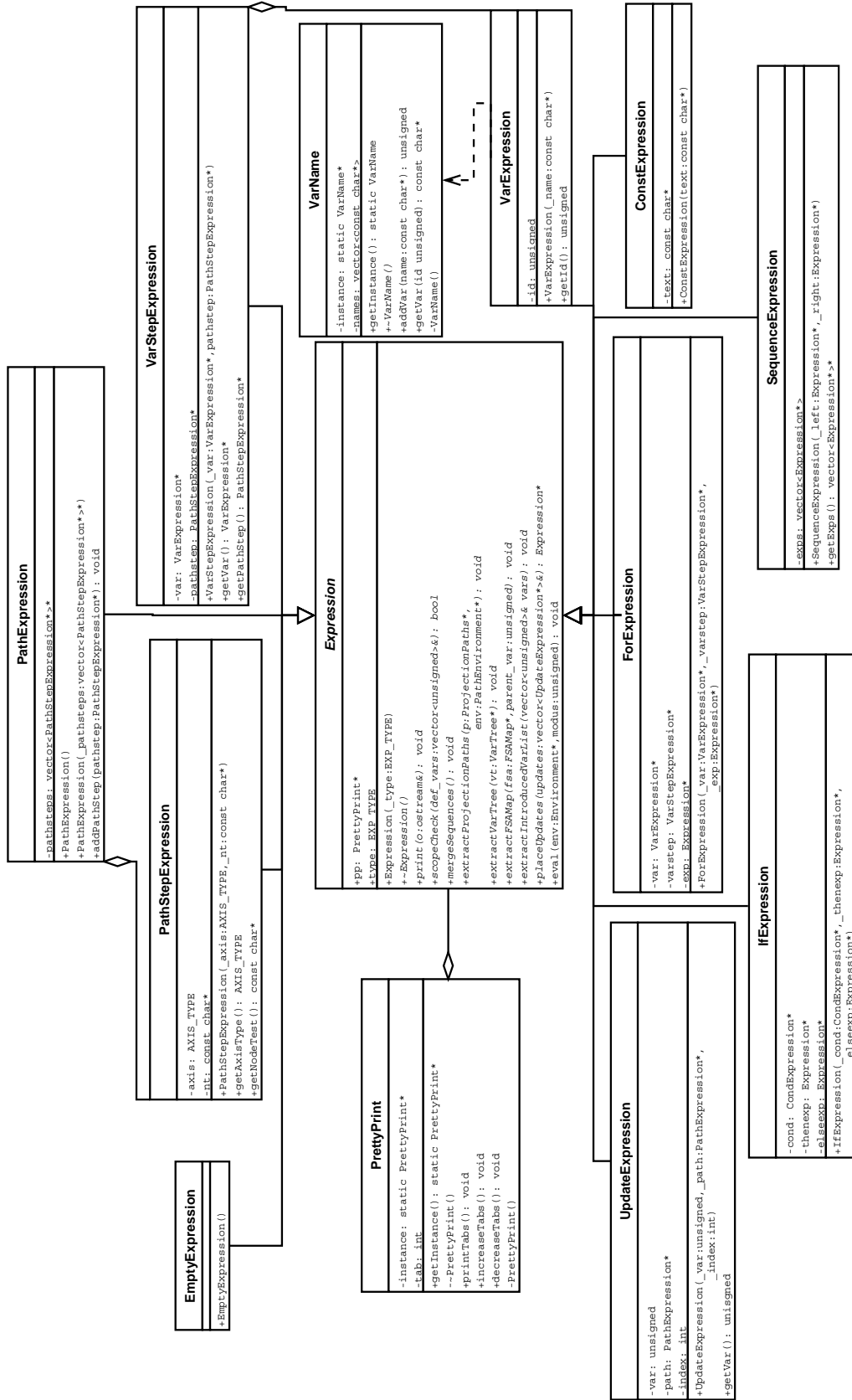
where

- *s* is a fixed string
- *t* is a tag identifier
- *\$root* is the variable that initially is bound to the document root node

B UML I - Conditional Expressions



C UML II - Non-Conditional Expressions



D Queries Used for Correctness Tests

D.1 W3C “XMP” XQuery Usecases

```
<result>
  <bib>
    for $bib in $root/bib return
      for $book in $bib/book return
        if ($book/publisher="Addison-Wesley" and
            $book/year>"1991")
          then
            <book>
              ( $book/year, $book/title )
            </book>
          else ()
  </bib>
</result>
```

```
<results>
  for $bib in $root/bib return
    for $book in $bib/book return
      for $title in $book/title return
        for $author in $book/author return
          <result>
            ($title,$author)
          </result>
</results>
```

```
<results>
  for $bib in $root/bib return
    for $book in $bib/book return
      <result>
        ( $book/title, $book/author )
      </result>
</results>
```

```
<result>
  <bib>
    (
      for $book in $root//book return
        if (exists $book/author)
          then
            <book>
              ( $book/title, $book/author )
            </book>
          else (),
      for $book2 in $root//book return
        if (exists $book2/editor)
          then
            <reference>
              (
                $book2/title,
                for $editor in $book2/editor return
                  $editor/affiliation
              )
            </reference>
          else ()
    )
  </bib>
</result>
```

D.2 DBMS Milestone Test Queries

```
<result>
  for $a in $root//section return
    for $b in $a//p return
      if ($b="T2")
      then $a
      else ()
</result>
```

```
<result>
  for $x1 in $root//p return
    if ($x1="T2")
    then
      for $x2 in $root//section return
        if ($x2//p=$x1)
        then $x2
        else ()
    else ()
</result>
```

```
<title-list>
  for $section in $root//section return
    if (exists $section//image)
    then
      for $title in $section/title return
        $title
    else ()
</title-list>
```

```
<title-list>
  for $section in $root//section return
    if ($section//image="")
    then
      for $title in $section/title return $title
    else ()
</title-list>
```

```
<title-list>
  for $section in $root//section return
    for $title in $section/title return
      <new-title>
        for $child in $title/* return
          $child
      </new-title>
</title-list>
```

```
<title-list>
  for $section in $root//section return
    <section>
      if (exists $section//section)
      then <subsection></subsection>
      else
        (
          for $title in $section/title return
            <subsection>
              for $child in $title/* return
                $child
            </subsection>,
          for $element in $root//* return
            <element></element>
        )
    </section>
</title-list>
```

```
<title-list>
  for $section in $root//section return
    if (exists $section//section)
    then ()
    else
      for $title in $section/title return
        $title
</title-list>
```

```
<result>
  for $a in $root//section return
    if ($a/title="Introduction")
      then $a/title
    else ()
</result>
```

```
<doc>
  $root
</doc>
```

```
<a>
  (
    (
      <a></a>,
      <b>
        (
          ( $root//section, $root//title ),
          ( <c>bla1</c>, bla2 )
        )
      </b>
    ),
    (
      $root//p,
      for $a in $root//* return
        if ($a="T2")
          then "T2"
        else "T3"
    )
  )
</a>
```

D.3 User Defined Queries

```
<result>
  for $bib in $root/bib return
    if (exists $bib/book)
    then
      for $book in $bib/book return $book/title
    else
      for $book2 in $bib/book2 return $book2/author
</result>
```

```
<result>
  for $bib in $root/bib return
    for $book in $bib/book return
      if (exists $book/test)
      then <match> $book/title </match>
      else ()
</result>
```

```
<result>
  for $book in $root//book return
    $book
</result>
```

```
<result>
  for $title in $root//title return
    $title
</result>
```

```
<result>
  (
  $root,
  for $book in $root//book return
    if (exists $book/test)
    then $book
    else ()
  )
</result>
```

```
<result>
  for $bib in $root/bib return
    for $book in $bib/book return
      <match>
        (
          <titlematch> $book/title </titlematch>,
          <authormatch> $book/author </authormatch>
        )
      </match>
</result>
```

```
<result>
  for $bib in $root/bib return
    for $book1 in $bib/book return
      for $book2 in $bib/book return
        if ($book1/author=$book2/author)
          then
            <match>
              ( $book1/title, $book2/title )
            </match>
          else ()
</result>
```

```
<result>
  for $bib in $root/bib return
    for $book1 in $bib/book return
      for $book2 in $bib/book return
        if ($book1/author=$book2/author and
            (not $book1/title=$book2/title))
          then
            <match>
              ( $book1/title, $book2/title )
            </match>
          else ()
</result>
```

```
<result>
  for $bib in $root//bib return
    for $book in $bib//book return
      if ($book//author="AUTHOR1")
        then <match> $book/title </match>
        else <nomatch> $book/title </nomatch>
</result>
```

E XMark Test Queries

```
<query1>
for $site in $root/site return
  for $people in $site/people return
    for $person in $people/person return
      if ($person/person_id="person0")
      then <result>$person/name</result>
      else ()
</query1>
```

```
<query6>
for $auction in $root/auction return
  for $site in $auction//site return
    for $regions in $site/regions return
      $regions//item
</query6>
```

```
<query8>
for $site in $root/site return
  for $people in $site/people return
    for $person in $people/person return
      <item>
        (<person>$person/name</person>,
        <items_bought>
          for $site2 in $root/site return
            for $cas in $site2/closed_auctions return
              for $ca in $cas/closed_auction return
                for $buyer in $ca/buyer return
                  if ($buyer/buyer_person=
                    $person/person_id)
                  then <result>$ca</result>
                  else ()
          </items_bought>
        )
      </item>
</query8>
```

```
<query13>
for $site in $root/site return
  for $regions in $site/regions return
    for $australia in $regions/australia return
      for $item in $australia/item return
        <item>
          (
            <name>$item/name</name>,
            <desc>$item/description</desc>
          )
        </item>
</query13>
```

```
<query20>
for $site in $root/site return
  for $people in $site/people return
    for $person in $people/person return
      if (exists $person/income)
      then ()
      else $person
</query20>
```

References

- [1] M. Altinel and M. Franklin. “Efficient Filtering of XML Documents for Selective Dissemination of Information.”. In *Proc. ICDE’00*, pages 53–64, 2000.
- [2] 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.
- [3] M. Benedikt, W. Fan, and F. Geerts. “XPath Satisfiability in the Presence of DTDs”. In *Proc. PODS*, pages 25–36, 2005.
- [4] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Reference implementation of “Type-Based XML Projection (VLDB 2006)”, 2006. <http://www.lri.fr/~kn>.
- [5] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. “Type-Based XML Projection”. In *Proc. VLDB’06*, 2006.
- [6] S. Bressan, B. Catania, Z. Lacroix, Y. G. Li, and A. Maddalena. “Accelerating Queries by Pruning XML Documents”. *TKDE*, 54(2):211–240, 2005.
- [7] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. “The XML Stream Query Processor SPEX”. In *icde2005*, pages 1120–1121, 2005.
- [8] Y. Diao, P. Fischer, M. J. Franklin, and R. To. “YFilter: Efficient and Scalable Filtering of XML Documents.”. In *Proc. ICDE’02*, 2002.
- [9] “DTD Tutorial”. <http://www.w3schools.com/dtd/default.asp>.
- [10] L. Fegaras, R. Dash, and Y. Wang. “A Fully Pipelined XQuery Processor.”. In *XIME-P*, 2006.
- [11] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. “Query Processing of Streamed XML Data”. In *Proc. CIKM 2002*, pages 126–133, 2002.
- [12] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. “The BEA/XQRL Streaming XQuery Processor”. In *Proc. VLDB’03*, pages 997–1008, 2003.

- [13] “The FluXQuery Engine”, 2004. <http://www-db.cs.uni-sb.de/~scherzin/FluXQuery.html>.
- [14] “Galax”. <http://www.galaxquery.org/>.
- [15] G. Gottlob, C. Koch, and R. Pichler. “XPath Processing in a Nutshell”. *ACM SIGMOD Record*, 32(2):21–27, 2003.
- [16] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. “Processing XML Streams with Deterministic Automata”. In *Proc. ICDT’03*, pages 173–189, 2003.
- [17] M. Grohe, C. Koch, and N. Schweikardt. “Tight Lower Bounds for Query Processing on Streaming and External Memory Data”. In *Proc. ICALP’05*, pages 1076–1088, 2005.
- [18] S. Helmer, C.-C. Kanne, and G. Moerkotte. “XQuery Processing in Natix with an Emphasize on Join Ordering”. In *XIME-P*, 2004.
- [19] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Pappas, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. “TIMBER: A Native XML Database”. In *Proc. VLDB’02*, pages 274 – 291, 2002.
- [20] C. Koch. “On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values”. In *Proc. PODS’05*, pages 84–97, 2005.
- [21] C. Koch. “On the role of composition in XQuery”. In *WebDB*, pages 37–42, 2005.
- [22] 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*.
- [23] 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.
- [24] X. Li and G. Agrawal. “Efficient evaluation of XQuery over streaming data”. In *Proc. VLDB’05*, pages 265–276, 2005.

- [25] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. “A Transducer-Based XML Query Processor”. In *Proc. VLDB’02*, pages 227–238, 2002.
- [26] S. Madden and M. J. Franklin. “Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data”. In *Proc. ICDE’02*, 2002.
- [27] A. Marian and J. Siméon. “Projecting XML Documents”. In *Proc. VLDB’03*, pages 213–224, 2003.
- [28] “MonetDB/XQuery”. <http://monetdb.cwi.nl/XQuery/>.
- [29] D. Olteanu. “Forward Node-Selecting Queries Over Trees”. *TODS*, 2006. *Conditionally accepted*.
- [30] D. Olteanu. “SPEX: Streamed and Progressive Evaluation of XPath”. In *TKDE*, 2006. *Conditionally accepted*.
- [31] D. Olteanu, T. Furche, and F. Bry. “An Efficient Single-Pass Query Evaluator for XML Data Streams”. In *Proc. 19th Annual ACM Symposium on Applied Computing (SAC)*, Cypros, Mar. 2004. Technical Report.
- [32] D. Olteanu, T. Furche, and F. Bry. ”Evaluating Complex Queries against XML streams with Polynomial Combined Complexity”. In *Proc. BNCOD 2004*, pages 31–44, 2004.
- [33] D. Olteanu et al. “XPath: Looking Forward”. In *EDBT ’02: Proceedings of the Workshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 109–127, 2002.
- [34] D. Olteanu et al. “An Evaluation of Regular Path Expressions with Qualifiers against XML Streams”. In *Proc. ICDE’03*, page 702, 2003.
- [35] F. Peng and S. S. Chawathe. “XPath Queries on Streaming Data”. In *Proc. SIGMOD 2003*, pages 431–442, 2003.
- [36] X. Peng, R. Brazile, and K. M. Swigger. “Extending XML Document Projection for Data Integration”. In *Information Reuse and Integration, Conf. 2005. IRI -2005 IEEE International Conference on.*, pages 138–142, 2005.
- [37] “Qizx/open”. <http://www.axyana.com/qizxopen/>.
- [38] R. Ramakrishnan and J. Gehrke. “*Database Management Systems (3rd edition)*”. McGraw-Hill Higher education, 2003.

- [39] Saarland University Database Group. “Stammvorlesung Database Systems”. <http://www-db.cs.uni-sb.de/teaching/dbs05/>.
- [40] Saarland University Database Group. “Stammvorlesung Database Systems - XQuery Test Cases”. <http://www.infosys.uni-sb.de/teaching/dbs05/testqueries.html>.
- [41] “Saxon”. <http://saxon.sourceforge.net/>.
- [42] H. Su, E. A. Rundensteiner, and M. Mani. “Semantic Query Optimization for XQuery over XML Streams”. In *Proc. VLDB*, pages 277–288, 2005.
- [43] “Tamino - Native XML Management”. <http://www.softwareag.com/corporate/products/tamino/default.asp>.
- [44] P. R. Wilson. “Uniprocessor Garbage Collection Techniques”. In *Proc. IWMM'92*, pages 1–42, 1992.
- [45] World Wide Web Consortium. “Document Object Model (DOM)”. <http://www.w3.org/DOM/>.
- [46] World Wide Web Consortium. “W3C Use Case XMP”. <http://www.w3.org/TR/2003/WD-xquery-use-cases-20031112/>.
- [47] World Wide Web Consortium. “XML Path Language (XPath 1.0)”. <http://www.w3.org/TR/1999/REC-xpath-19991116>.
- [48] World Wide Web Consortium. “XML Path Language (XPath 2.0)”. <http://www.w3.org/TR/2006/CR-xpath20-20060608/>.
- [49] World Wide Web Consortium. “XML Query (XQuery)”. <http://www.w3c.org/XML/query/>.
- [50] World Wide Web Consortium. “XML Schema”. <http://www.w3.org/XML/Schema>.
- [51] World Wide Web Consortium. “XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Candidate Recommendation 8 June 2006”. <http://www.w3.org/TR/query-algebra/>.
- [52] “Xindice - An open source XML native XML database system”. <http://www.softwareag.com/corporate/products/tamino/default.asp>.
- [53] “XMark”. <http://monetdb.cwi.nl/xml/>.