Efficient Main Memory-based XML Stream Processing

Dissertation
zur Erlangung des Grades
der Doktorin der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt von
Stefanie Scherzinger

Saarbrücken
February 17, 2008

Tag des Kolloquiums:	10.01.2008
Dekan:	Prof. Dr. Thorsten Herfet
Vorsitzender der Prüfungskommission:	Prof. Dr. Andreas Zeller
1. Berichterstatter:	Prof. Dr. Christoph Koch
2. Berichterstatter:	Prof. Dr. Gerhard Weikum
Akademischer Mitarbeiter:	Dr. Ralf Schenkel

Acknowledgments

I thank my advisor Christoph Koch for teaching me about research and sharing his conviction that it is worth striving for the ultimate solution to a problem. I further thank the members of my thesis committee Gerhard Weikum, Andreas Zeller, and Ralf Schenkel.

My colleagues Lublena Antova, Dan Olteanu, and Michael Schmidt at the *Saarland University Database Group*, and my collaborators Bernhard Stegmaier, Alfons Kemper, and Nicole Schweikardt have been very helpful at various phases during my Ph.D. Their technical input was always excellent, and it has been a great experience working with them. I owe Christina Fries for her fantastic coffee-making skills and her cheerful support.

I owe thanks to Gerti Kappel and Beate List, and all my former colleagues from the *Women's Institute of Internet Technologies* at the *Vienna University of Technology*. I thank Johannes Gehrke and the database group at *Cornell University* for their hospitality during my three-month stay in Ithaca.

Words cannot express my gratitude towards my family. I also thank my friends Sebastian Schöning, Thomas Zimmermann, Ralf Osbild, Arno Eigenwillig, Verena Lehmann, and Anne Krieg.

Short Abstract

Applications that process XML documents as files or streams are naturally main-memory based. This makes main memory the bottleneck for scalability. This doctoral thesis addresses this problem and presents a toolkit for effective buffer management in main memory-based XML stream processors.

XML document projection is an established technique for reducing the buffer requirements of main memory-based XML processors, where only data relevant to query evaluation is loaded into main memory buffers. We present a novel implementation of this task, where we use string matching algorithms designed for efficient keyword search in flat strings to navigate in tree-structured data.

We then introduce an extension of the XQuery language, called *FluX*, that supports event-based query processing. Purely event-based queries of this language can be executed on streaming XML data in a very direct way. We develop an algorithm to efficiently rewrite XQueries into FluX. This algorithm is capable of exploiting order constraints derived from schemata to reduce the amount of buffering in query evaluation.

During streaming query evaluation, we continuously purge buffers from data that is no longer relevant. By combining static query analysis with a dynamic analysis of the buffer contents, we effectively reduce the size of memory buffers.

We have confirmed the efficacy of these techniques by extensive experiments and by publication at international venues.

To compare our contributions to related work in a systematic manner, we contribute an abstract framework for XML stream processing. This framework allows us to gain a greater-picture view over the factors influencing the main memory consumption.

Kurzzusammenfassung

Anwendungen, die XML-Dokumente als Dateien oder Ströme verarbeiten, sind natürlicherweise hauptspeicherbasiert. Für die Skalierbarkeit wird der Hauptspeicher damit zu einem Engpass. Diese Doktorarbeit widmet sich diesem Problem, zu dessen Lösung sie Werkzeuge für eine effektive Pufferverwaltung in hauptspeicherbasierten Prozessoren für XML-Datenströme vorstellt.

Die Projektion von XML-Dokumenten ist eine etablierte Methode, um den Pufferverbrauch von hauptspeicherbasierten XML-Prozessoren zu reduzieren. Dabei werden nur jene Daten in den Hauptspeicherpuffer geladen, die für die Anfrageauswertung auch relevant sind. Wir präsentieren eine neue Implementierung dieser Aufgabe, wobei wir Algorithmen zur effizienten Suche in flachen Zeichenketten einsetzen, um in baumartig strukturierten Daten zu navigieren.

Danach stellen wir eine Erweiterung der XQuery-Sprache vor, genannt *FluX*, welche eine ereignisbasierte Anfragebearbeitung erlaubt. Anfragen, die nur ereignisbasierte Konstrukte benutzen, können direkt über XML-Datenströmen ausgewertet werden. Dazu entwickeln wir einen Algorithmus, mit dessen Hilfe sich XQuery-Anfragen effizient in FluX übersetzen lassen. Dieser benutzt Ordnungsinformationen aus Datenschemata, womit das Puffern in der Anfragebearbeitung reduziert werden kann.

Während der Verarbeitung des Datenstroms bereinigen wir laufend den Hauptspeicherpuffer von solchen Daten, die nicht länger relevant sind. Eine nachhaltige Reduzierung der Größe von Hauptspeicherpuffern gelingt durch die Kombination der statischen Anfrageanalyse mit einer dynamischen Analyse der Pufferinhalte.

Die Effektivität dieser Puffermanagement-Techniken erfährt ihre Bestätigung in umfangreichen Experimenten und internationalen Publikationen.

Für einen systematischen Vergleich unserer Beiträge mit der aktuellen Literatur entwickeln wir ein abstraktes System zur Modellierung von Prozessoren zur XML-Stromverarbeitung. So können wir die spezifischen Faktoren herausgreifen, die den Hauptspeicherverbrauch beeinflussen.

Zusammenfassung

Während des letzten Jahrzehnts hat sich in Wirtschaft und Industrie die *Extended Markup Language* (XML) als das Datenformat erster Wahl etabliert. Die Kommunikation auf Basis von XML-Technologien wird vor allem durch offene Standards und die Verfügbarkeit von Schemata begünstigt. Im Zuge der Verbreitung des XML-Formates hat insbesondere die Entwicklung von Anfragesprachen für XML-Daten erhöhte Aufmerksamkeit erhalten.

Zu den etablierten XML-Anfragesprachen zählen heute die W3C-Standards XPath und XQuery. Während mit XPath nur Boolsche Anfragen oder die Selektion von Knoten möglich sind, ist XQuery eine turingvollständige Programmiersprache, die auf XPath aufbaut. Mit der fortlaufenden Verbreitung dieser Sprachen ist es von zunehmender Wichtigkeit, effiziente Techniken für Ihre Auswertung zu entwickeln.

Dabei verdienen Szenarien, in denen XML-Dokumente als Dateien oder Ströme verarbeitet werden, besonderes Augenmerk. Diese Szenarien zeichnen sich dadurch aus, dass die Daten kontinuierlich eingehen, mit einer hohen Datenübertragungsrate und über lange Zeiträume hinweg.

Prozessoren für die Verarbeitung von XML-Datenströmen sind üblicherweise hauptspeicherbasiert. Für die Skalierbarkeit wird der Hauptspeicher dadurch zu einem Engpass. Zwar brauchen gängige Prozessoren für die Evaluierung von XPath-Ausdrücken auf XML-Datenströmen nur wenig Speicher; was aber Datentransformationen mit XQuery-Anfragen angeht, ist ein erheblicher Speicherverbrauch generell nicht vermeidbar. Es ist daher dringlich, Algorithmen für die Auswertung von XQuery zu entwickeln, die sparsam mit den verfügbaren Ressourcen umgehen und die Puffergrößen minimieren.

Im Idealfall wird die Pufferverwaltung in einem XQuery-Prozessor folgende Kriterien erfüllen: (1) Es werden nur solche Daten in den Puffer geladen, die für die Anfrageauswertung auch relevant sind. (2) Darüber hinaus sollte die Pufferverwaltung Daten nicht länger als nötig puffern, und (3) redundante Pufferinhalte vermeiden. Doch allein um das erste Kriterium zu erfüllen, müsste ein solches System in der Lage sein, XQuery-Anfragen auf Erfüllbarkeit zu testen. Dies ist jedoch ein unentscheidbares Problem.

Diese Doktorarbeit widmet sich der Entwicklung von Werkzeugen für eine effektive Pufferverwaltung in der Verarbeitung von XML-Datenströmen. Dabei streben wir es an, die Kriterien (1) bis (3) nach Möglichkeit zu erfüllen.

Das erste Kriterium erfordert es, möglichst nur relevante Daten in den Puffer zu laden. Dazu projezieren wir XML-Datenströme, bevor Anfragen auf den Daten ausgeführt werden. Die Projektion von XML-Dokumenten ist eine etablierte Methode, und bereits in diversen XQuery-Prozessoren integriert. Der generelle Ansatz beruht auf einer statischer Anfrageanalyse wonach Daten schon beim Parsen aus dem Eingabestrom entfernt werden, wenn sie für die Anfragebearbeitung zweifelsfrei irrelevant sind. Wir präsentieren eine neue Implementierung dieser Technik wobei wir Algorithmen zur effizienten Suche nach Schlüsselwörtern in flachen Zeichenketten einsetzen, um in baumartig strukturierten Daten zu navigieren. Anders als existierende Implementierungen betrachtet unser Algorithmus nur Bruchteile der Eingabe und weist einen sehr ökonomischen Verbrauch von Hauptspeicher und Rechenzeit auf. Bereits für Probleme niedriger Komplexität, wie das Filtern von Dokumenten nach XPath-Anfragen, erreichen wir in unseren Experimenten Beschleunigungen von zwei

Größenordnungen. Selbst im Vergleich zu leistungsfähigen SAX-Parsern, wie sie in praktisch allen Konkurrenz-Systemen eingesetzt werden, ist unser Ansatz bis zu zehnmal schneller.

Um die Menge der Daten, die in den Hauptspeicher-Puffer geladen werden, noch weiter zu reduzieren, analysieren wir Anfragen im Hinblick darauf, welche Teile direkt auf dem XML-Strom ausgewertet werden können (ohne dafür Daten zu puffern). Dazu führen wir eine Erweiterung der XQuery-Sprache ein, genannt *FluX*, welche eine ereignisbasierte Anfragebearbeitung erlaubt. Anfragen, die nur ereignisbasierte Konstrukte benutzen, können direkt auf XML-Datenströmen ausgewertet werden. Wir entwickeln einen Algorithmus um XQuery-Anfragen effizient nach FluX zu übersetzen. Dieser ist in der Lage, Schema-Informationen einzubeziehen, um das Puffern noch weiter zu reduzieren. Wir zeigen in unseren Experimenten, dass der FluX Ansatz den Speicherverbrauch deutlich reduziert. Es werden auch jene Fälle erkannt, in denen komplette Anfragen aus dem XMark-Benchmark direkt auf dem XML-Datenstrom ausgewertet werden können.

In dem zweiten Kriterium für effektives Puffermanagement wird gefordert, Daten nicht länger als nötig zu puffern. Während der Stromverarbeitung bereinigen wir daher Hauptspeicherpuffer fortwährend von Daten, die nicht länger relevant sind. Indem wir statische Anfrageanalyse mit einer dynamischen Analyse der Pufferinhalte kombinieren, reduzieren wir die Größe von Hauptspeicherpuffern nachhaltig. Unser Ansatz ähnelt der automatischen Speicherbereinigung in Programmiersprachen, denn jeder gepufferte Knoten prüft, ob er für die Anfrageauswertung noch relevant ist. Dies geschieht durch einen Relevanz-Zähler, der während der Anfrageauswertung kontinuierlich herabgesetzt wird. Das Herabsetzen geschieht zu Zeitpunkten, die in statischer Analyse bestimmt werden. Dadurch werden Knoten bereits dann zur Laufzeit aus dem Puffer gelöscht, sobald sie für die Anfragebearbeitung nicht mehr relevant sind, selbst wenn sie in internen Datenstrukturen noch referenziert werden.

Wie unsere Experimente belegen, erreichen wir so einen erheblich geringeren Speicherverbrauch als Systeme, deren Pufferverwaltung allein auf statischer Analyse beruht. Sofern Anfragen frei von Komposition sind, können wir sogar sicherstellen, dass Knoten aus dem XML-Eingabedokument nicht redundant gepuffert werden. Damit erfüllen wir auch das dritte Kriterium für dieses XQuery-Fragment.

Die Effektivität und Skalierbarkeit unserer Techniken wurden in umfangreichen Experimenten mit realistischen Daten und offiziellen Benchmarks bestätigt. Weiterhin wurden unsere Lösungen auf anerkannten Fachtagungen vorgestellt.

Um unsere Beiträge systematisch mit der bestehenden Literatur zu vergleichen, präsentieren wir in dieser Arbeit des Weiteren ein abstraktes System zur Modellierung von Prozessoren zur XML-Stromverarbeitung. Dieses System beruht auf Termersetzung, mit Termen als den zentralen syntaktischen Objekten. Unser System bietet eine umfassende Sicht auf Prozessoren zur XML-Stromverarbeitung, die sowohl die Struktur des Stroms als Sequenz von öffnenden und schließenden XML-Klammern erfasst als auch die abstrakte Sicht von Pufferinhalten als Bäume ermöglicht. Mit diesem Formalismus modellieren wir mehrere XML-Prozessoren. Dabei spannen wir den Bogen von einfachen Stromübersetzern bis hin zu XQuery-Prozessoren. So können wir speziell jene Faktoren eingehend betrachten, die den Hauptspeicherverbrauch in der XML-Stromverarbeitung beeinflussen.

640K should be enough for anyone.

Attributed to Bill Gates.

Contents

Part I

Introduction

1 Motivation and Outline

Challenges in XML Stream Processing

Within the last decade, industry has converged to XML as the data exchange format of choice. Communication based on XML technology is fostered by open standards and the ready availability of schemas. Along with the proliferation of the XML data format, the development of query languages dedicated to XML processing has received much attention for several reasons.

- XML query languages avoid the impedance mismatch that occurs when XML data is processed with XML-unrelated programming languages.
- Declarative query languages lend themselves to automatic optimization, whereas procedural programs require manual tuning.
- Composable and statically typed XML query languages are a good basis for building robust protocols for data exchange.

Among the most established XML query languages are the W3C standards XPath [113] and XQuery [114]. While XPath only allows for Boolean or node-selecting queries, XQuery is a Turing-complete language that comprises XPath. As more developers use these languages, it becomes increasingly important to devise well-principled machineries for their execution.

In recent years, research and industry have delivered various systems for XQuery processing, both disk-backed and main memory-based. Typically, these systems operate in two phases, as they load the complete input before querying it. This happens in XML databases where XML data is stored persistently, but also in main memory-based query engines such as the IPSI, Saxon, and QizX XQuery processors [40, 90, 93]. These systems load the entire input into main memory buffers. Consequently, they are severely self-limited as to the amount of input they can handle.

Yet in most cases of XML data exchange we face *streaming scenarios*, where this two-phase approach is impractical. In streaming scenarios, the data arrives continuously, at a high rate, and over long periods of time. A priori, we take a general point of view and make no assumptions about the structure of the stream. In doing so, we differ from works where the stream is assumed to consist of a sequence of XML documents. Frequently, each single document is deemed small enough to fit into main memory (c.f. [6, 17, 53]). Also, we do not presume the existence of punctuations [104]. Rather, we regard an XML stream as a single coherent XML document of (theoretically) infinite size.

In this setting, it is impractical to load the entire input before querying. Even if the input can be stored, users most likely cannot afford the time required for loading, as streams arrive over long periods of times. Also, users may not even care for persistent storage of the data, e.g. in an XML database, if its relevance expires within moments of its arrival. Out of this reason, *main memory*-based query processors are frequently used in streaming scenarios. Today, there exists a variety of main memory-based query engines for XML stream processing, which we can broadly classify as either automata- or algebra-based.

For *restricted* fragments of XPath, engines that use automata for stream processing have been shown to scale up to large inputs [85,87], and even to high workloads of queries [35,87]. Automata encodings have also been developed for larger XQuery fragments [63, 76]. In the XSM machine [76], a fragment with nested for-loops and where-conditions is evaluated using a network of transducers. While automata-based approaches are elegant and efficient, it is doubtful whether they can generalize to full XQuery. Automata operate on a low level of abstraction, and may not lend themselves naturally for integration with algebraic approaches to query optimization.

Most query processors that evaluate full XQuery, or comprehensive fragments thereof, are based on some form of query algebra. Some of these algebras also provide streaming operators [43,44,70,92], which may even be implemented using automata (e.g. [101]). The main limitation of systems that evaluate larger XQuery fragments is their high memory footprint. In fact, for XML data transformations, as opposed to just XPath filtering, the need for substantial main memory buffers cannot be avoided in general [12]. While there are satisfactory solutions for the evaluation of XPath fragments over XML streams, the evaluation of XQuery over XML streams is still a largely unresolved issue. In existing XQuery processors, main memory remains the major bottleneck.

Ideally, the buffer manager of a streaming XQuery engine will (1) only load 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. Yet already a system optimal for (1) would have to be able to check satisfiability of XQuery expressions, an undecidable problem (c.f. [14]).

When devising XQuery engines that treat buffering as an optimization target, we must thus resort to a best-effort approach. In assessing buffer management algorithms, our optimization target is the high watermark of main memory consumption, yet the overall query evaluation time must not be neglected.

Finally, we desire that our techniques can be blended naturally with existing, possibly algebra-based approaches to XML query evaluation and optimization.

Contributions

This dissertation is dedicated to the efficient processing of XML streams by means of main memory-based query engines. Our main focus is on XQuery evaluation. Current XQuery processors are severely limited by their high memory footprint, and hence cannot scale to XML streams.

The objective of this dissertation is twofold: First, we present a general framework for XML stream processing. The emphasis in developing such a framework is on providing a comprehensive view on query evaluation over XML streams, rather than concrete algorithms. Second, we deliver practical buffer management techniques for XQuery engines. We have verified these techniques by extensive experiments and by publications at international conferences and workshops. We outline our principal contributions in the following.

An Abstract Framework for XML Stream Processing

In relational databases, we assume that the database state is too large to be stored in main memory. The state is thus stored on disk, and disk I/O is generally regarded as the dominating matter of expense. In XML stream processing, we deal with a different setup.

From a data-centric point of view, the basic infrastructure consists of a *read-only input tape* (or input stream) of XML tokens, and a *write-only output tape* (or output stream) of XML tokens. On both tapes, a cursor – which can only move forward – marks the current read or write position. A *physical query plan*, the contents of the input tape, and the buffer contents specify a sequence of operations that compute the query result. Finally, a *main memory buffer* stores data for future reference.

Then data that is not buffered when it is first read in the input, or data that is purged from the buffer, is irretrievably lost. Hence, buffer management must take care that all relevant data is buffered, and that it is not discarded prematurely. At the same time, an effective buffer manager will attempt to keep the memory footprint small.

In a high-level discussion of concepts for buffer minimization, a specific query algebra may bias us against certain optimizations. Out of this consideration, we have developed a formal framework for XML stream processing. Our framework is based on term rewriting [33, 50], and uses terms as syntactic objects. The processing of the input, the management of the buffer contents. and the writing of tokens to the output are specified using term rewriting rules, which is the mapping defined for manipulating terms.

Our framework provides a novel and integrating view on typical XML stream processing tasks. We use it to model two systems for XML stream processing. First, we model XML stream pushdown transducers (termed *XML-DPDT*s in [69]). These transducers are representatives of a class of important stream processing tasks, comprising basic operations such as checking well-formedness and the validation of XML streams against DTDs. Yet these transducers can also execute XPath filtering tasks [6, 36, 84] or perform XML document projection (c.f. [15, 77]). They are even able to execute highly scalable XML stream transformations, as specified by attribute grammar formalisms (c.f. [69]). Second, we model XQuery engines in our framework, and illustrate techniques for streaming query evaluation and efficient buffer management.

We will thus consult our framework in exploring the design space of XML stream processors with a low memory footprint, and for gaining an understanding of the factors influencing memory consumption.

Buffer Management Algorithms

The following buffer management algorithms are proposed in this dissertation. These techniques can be implemented in isolation, but they are also composable, and thus form a comprehensive toolkit of buffer management techniques.

XML prefiltering. One of the earliest works to reduce the memory footprint is XML document projection, a prefiltering technique first realized in the Galax XQuery engine [77]. Based on a static analysis of the query, all data that is certainly irrelevant to query evaluation is immediately discarded without further processing. This approach can greatly reduce the main memory requirements of query processors.

In this dissertation, we take on a new approach to implementing XML prefiltering. Our technique takes string matching algorithms designed for efficient keyword search in flat strings into the second dimension, to navigate in tree structured data. Different from existing schemes, we usually process only fractions of the input and get by with very economical consumption of main memory and processing time. Our experiments reveal that, already on low-complexity problems such as XPath filtering, in-memory query engines can experience speedups by two orders of magnitude. Even in comparison to bare input tokenization by industrial-strength SAX parsers, which parse the input data in virtually all competing applications, our system is up to ten times faster.

Event-based XQuery processing. We introduce an extension of the XQuery language, called *FluX*, that supports event-based query processing and the conscious handling of main memory buffers. Purely event-based queries of this language can be executed on streaming XML data in a very direct way. We then develop an algorithm that statically rewrites XQueries into the FluX language. This algorithm uses DTD order constraints to schedule event handlers and to thus minimize the amount of buffering required for evaluating a query.

Effective buffer purging. We further propose a novel buffer purging algorithm which combines static and dynamic analysis to keep main memory consumption low. This is achieved by a technique that we call *active garbage collection*, which timely purges buffers at runtime. Our approach is strongly related to garbage collection via reference counting, as each node in the buffer keeps track whether it is still relevant to the remaining query evaluation. While a traditional garbage collector frees memory based on reference counts, i.e. based on dynamic analysis alone, our approach additionally exploits data access patterns that are derived in static query analysis. That is, we purge nodes from buffers once they are no longer relevant for query evaluation, even if they are still referenced in internal data structures.

For the techniques outlined above, we have built prototype systems for a practical fragment of XQuery. The experimental results demonstrate the significant impact of our algorithms on main memory consumption. While improving the memory footprint is our main objective, this typically also results in more time-efficient algorithms. Indeed, in our experiments we generally also observe improvements in running time.

Structure

This dissertation is divided into four parts. In the first part, we have so far motivated XML stream processing, and have outlined our contributions. Related work is discussed in Chapter 2. In Chapter 3, we present technical background and define a practical XQuery fragment. We further introduce our own approach to XML document projection.

We define an abstract framework for XML stream processing in the second part. This framework is defined in the manner of term rewriting systems in Chapter 4. In Chapter 5, we model two query engines. The first is a strictly scalable XML transducer which can execute stream processing tasks such as XML document projection, validation against DTDs, and even a restricted class of stream transformations. The second query engine is an in-memory XQuery processor implementing a powerful XQuery fragment. In Chapter 6, we discuss various approaches to building buffer-conscious XML stream processors. At this point, our goal is to provide intuition and motivation, rather than algorithms. These will be delivered in the third part.

The third part presents concrete buffer management algorithms. We present our implementation of XML prefiltering as a string matching problem in Chapter 7. In Chapter 8, the static scheduling of streaming query operators is explored, both in the absence and presence of a schema. In combining static and dynamic buffer management, we develop a garbage collector for XQuery processors in Chapter 9. For each of these techniques, we have conducted extensive experiments, as described in the corresponding chapters.

In the fourth part, we first summarize our contributions. We then point out opportunities for future research, and discuss how the techniques developed here may foster further research.

The Appendices contains details on queries and data used in our experiments, and a list of publications that were written in context of this dissertation.

2 Related Work

Over the past years, there has been growing interest in XML stream processing. In this chapter, we discuss our contributions to this field in the context of related work. We begin by describing the state-of-the-art. There currently co-exist two notions of XML streams, namely that of a continuous stream of multiple XML documents, called *packets* or *messages*, and that of a stream encoding a single coherent XML document. We refer to streams of the first kind as *packet streams* or *multi-document* streams, and to streams of the second kind as *single-document* streams. For multi-document streams, we assume that a stream consists of self-contained units that each fit into main memory, while we do not make this assumption for single-document streams.

In Sections 2.1 and 2.2, we discuss systems and query languages for processing these two kinds of XML streams. Section 2.3 is dedicated to the query language XQuery. We discuss the growing interest in this language and compare two processing models. The first is a two-phase approach, where the complete input is loaded prior to query evaluation. The second is a streaming approach, where the query is evaluated incrementally on the input stream. We then discuss techniques for XQuery optimization. In particular, we focus on reducing the memory footprint of main memory-based XQuery engines.

2.1 Multi-Document Streams

In traditional publish-subscribe scenarios, streams are “flat” and consist of attribute-value pairs or tuples. A variety of data stream management systems (DSMSs) have been introduced, such as the STREAM system [8, 82], the AURORA project [10], TelegraphCQ [72], NiagaraST [83], the StreamGlobe project [73], and SASE [118]. Yet when the data format is XML, richer messages and queries increase the complexity of stream processing accordingly.

In *XML publish-subscribe scenarios* [6, 17, 53], it is generally assumed that the input consists of a sequence of self-contained XML documents or packets, where each individual packet is small enough to fit into main memory. The packets can then be processed one at-a-time, where several queries are evaluated concurrently on a packet. Depending on the application, queries of different expressiveness are evaluated. We distinguish between applications that evaluate queries against single packets from those that evaluate queries specified over several packets. In both cases, the optimization target is to share work when queries are evaluated concurrently. In the context of XML, this also concerns the evaluation of structural joins (regarding the relationships of nodes in the XML document tree), which must be evaluated in addition to value-based joins. We also touch on this topic in the subsequent section, when we discuss XML database techniques.

Queries against single packets. In *packet routing*, a router forwards incoming packets depending on client subscriptions [6, 98]. Subscriptions can be specified using the XPath language for navigation inside XML documents. A packet is forwarded to a subscriber if one of his or her XPath subscriptions is matched by the packet. Commonly, these XPath queries are evaluated as Boolean queries. For restricted fragments of XPath, typically using the child- and descendant axis, value-based predicates, and the wildcard "*", high workloads of queries can be compiled into a single pushdown transducer. This transducer can be implemented as a main memory-based application, which requires little memory and yet scales up to high workloads of queries [6, 27, 36, 53, 54, 87].

For the purpose of building efficient routing infrastructures, queries may be clustered based on selectivity estimates, e.g. as in [26]. So we search for similarities between queries for the purpose of sharing work in query evaluation. This also holds for the BEA XQuery engine [49], where XQuery is evaluated against single XML packets. The BEA engine is capable of caching intermediate results for the evaluation of future queries [37].

Queries spanning several packets. As in traditional data stream management systems, in XML DSMSs we can formulate continuous queries by means of time constraints, such as queries with window-based joins. As queries are posted over several packets, we talk of *inter-document querying*. CayugaX [59] is such a system. Like the majority of its competitors, CayugaX has a relational backend to accelerate join processing for high workloads of queries.

2.2 Single-Document Streams

In an alternative view, an XML stream is seen as a single coherent XML document. We refer to this notion as single-document streams. A priori, no assumptions are made about whether parts of the input fit into main memory. This is also the view taken in this dissertation. For the remainder of this dissertation, by *XML streams* we will always refer to single-document XML streams.

Apart from data that occurs naturally in this form, such as in end-to-end data exchange of large XML files, there is also the case of ad-hoc processing of large XML files that reside on disk. It is adequate to process this data in streaming form as well, as a single sequential scan over the input is preferable to non-sequential data access on disk.

The majority of systems processing single-document streams are completely main memory-based, and evaluate single queries [44, 76, 85]. It seems that multi-query evaluation for single-document streams is a largely unexplored topic. We next categorize existing systems depending on whether they only evaluate node-selecting queries, or whether they can actually encode data transformations. We also distinguish between query languages designed for XML streams, and languages that have been proposed independently.

Node selecting queries. In packet routing, XPath expressions are evaluated as Boolean queries. In processing single-document streams, XPath expressions are evaluated as node-selecting queries. Before output can be produced, parts of the input may have to be buffered until conditions can be checked. Consequently, these XPath processors require buffering capabilities. For instance, the SPEX XPath processor [85] evaluates a practical XPath fragment over XML streams. Its backend is a network of pushdown transducers, with one Turing machine dedicated to the task of buffering. Related projects that also evaluate node selecting queries include [13, 27, 76, 87].

XML stream transformation queries. Several query languages have been proposed for the purpose of specifying XML stream transformations. For such transformations to be scalable in the strictest sense, there is a need for query languages which can be evaluated in linear time in the size of the input, by one linear forward scan over the data. Also, at any time during query evaluation, memory consumption must be bounded w.r.t. the length of the input stream, but not the depth of the XML tree. This is motivated by the fact that we need a stack, and hence at least the expressiveness of a pushdown automaton, for even the most basic XML parsing tasks.

In earlier work, we have contributed to this class of queries a novel formalism, the XML stream attribute grammars (XSAGs) [69, 94]. XSAG queries are specified in the style of attribute grammars, e.g. based on DTDs as a grammar formalism. XSAGs can be evaluated by deterministic pushdown transducers, and adhere to the requirements for strict scalability outlined above. The XSAG formalism provides a strong intuition for which transformations can be evaluated scalably. Naturally, XSAGs can only specify queries of limited expressiveness.

In practice, it is desirable and often justified to allow for memory buffers in query evaluation, at the cost of sacrificing strict scalability. The query languages named in the following have been designed for stream processing, but are indeed Turing-complete. We characterize selected XML stream processing languages by their underlying design paradigms.

- **Schema-based languages.** As an extension to XSAGs, we have developed a full-fledged attribute grammar formalism with unrestricted Java code in the attribution functions [96]. In the tradition of tools such as Yacc [74], a parser generator translates *TransformX attribute grammars* to Java source code, which may then be compiled and executed.

- **Scripting languages.** The language design of STX [29] leans on the scripting language XSLT [115], but its execution model is targeted at stream processing. While STX seems to be supported by an active user community, it has so far received little attention in database research.

- **Functional languages.** XStream [50] is a functional programming language that is based on term rewriting. Unlike XQuery, which is also functional, XStream does not incorporate XPath expressions. In XStream evaluation, all query terms are rewritten eagerly. Consequently, the results produced by rewriting terms may have to be temporarily buffered to put the output of all terms in the correct order. Depending on the query at hand, this can lead to a considerable memory overhead. In Section 6.1 we will discuss this issue in further detail.

In the query languages above, queries are specified on a high level of abstraction, usually as tree manipulations. Authors of queries are oblivious to the single XML events of which XML streams are composed. It is then the job of the query compiler to translate queries over trees into an event-based processing of the input. This also holds when XPath expressions are compiled into pushdown automata that operate on the level of XML events, as done in XML packet routing or filtering. In Chapter 7, we present an approach that compiles path expressions to an even lower level of abstraction. Instead of processing SAX events, our projection algorithm operates on the unparsed encoding of XML streams. It largely disregards the token-structure of the input, by leveraging established string matching algorithms. The effect is a highly scalable approach to XML stream processing.

The low-level counterpart to tree-manipulating query languages are user-written applications, e.g. based on SAX parsers [79], which are frequently tedious to maintain and error-prone. Nevertheless, there have been recent advances regarding XML type-checking even for such programs [88].

To specify backends for streaming query engines, we need formal frameworks that allow for a unified view, as they support both the specification of high-level tree manipulations and the low-level handling of single events from the XML input stream. Both our formal framework for XML stream processing (which we introduce in Chapter 4) and the FluX query language (presented in Chapter 8) syntactically distinguish between both levels of abstraction.

General-purpose XML query languages. While originally not intended for XML stream processing, there has been a large effort towards the streaming evaluation of general-purpose XML query languages such as XSLT or XQuery, e.g. see [42, 44, 70, 75, 76, 97, 101, 106]. Our work focuses on streaming XQuery evaluation, and we dedicate the next section to this language.

2.3 XQuery Evaluation Techniques

We next discuss the XQuery language [114] and its areas of applications. We classify different kinds of XQuery engines by their processing model. We distinguish systems that operate in a two-phase approach of loading the data before evaluating queries, from systems that have a single-pass streaming execution model. We conclude with an overview over XQuery optimization techniques.

XQuery is a functional query language where the most prominent feature is the FWR-expression, a looping primitive. In a for-loop "for $\$x$ in α return β", the query-variable $\$x$ is bound to a sequence of nodes as computed by expression α. In a for-loop "for $\$x$ in α where χ return β", the condition χ must be additionally satisfied for a binding to occur. Then for each binding of the query-variable, the body β of the for-loop is evaluated.

This looping primitive is evidence of the functional nature of XQuery, as expression β can be evaluated in parallel for all bindings of $\$x$. XML databases commonly exploit this property in the compilation of query plans. Yet as we discuss in Chapter 6, in the context of main memory-based XQuery processors, it is sometimes preferable to evaluate queries such that each query-variable binds to only one node from the input document at-a-time.

XQuery language extensions. While XQuery has been approved as a W3C standard, there is ongoing interest in the design of this language. Recently, various language extensions have been proposed.

- **XQuery in application development.** XQuery is increasingly being used as a programming language, to avoid the impedance mismatch frequently observed when applications for XML processing are implemented with with XML-unrelated programming languages. For this purpose, dialects such as XQueryP [24] and XQuery! [52] have emerged.
- **XQuery with full-text support.** To support versatile text search in XML documents and to compute ranked results, the XQuery full-text extensions have been proposed [7,31]. This makes XQuery a candidate for XML processing tasks in information-retrieval as well (c.f. [57,103]).
- **XQuery with window-based aggregations.** While XQuery was not intended for stream processing, it has been studied intensively in this context [41–43,70,73,75,76,97,101]. Recently, stream-specific language extensions such as window-based aggregations have been explored [23,99].

We regard these efforts to further extend the XQuery language as a strong indicator that the adoption of the language is gaining momentum.

XQuery processing models. As the XQuery language claims new territories, building efficient XQuery processors is an ever so important task. In the past, a variety of processors has evolved. We discuss them regarding their suitability for stream processing.

Two-phase query evaluation. In systems implementing a two-phase approach, the data is first loaded, either into main memory [40,51,90,93] or an XML database [47,62,81], before query evaluation sets in. This has the advantage that the compilation of query plans does not have to assume a single sequential scan over the data. Hence, the full armamentarium of classic query optimization techniques, as well as join algorithms or indexes specifically targeted at XML query evaluation, can be applied.

Yet processors that load the entire input into main memory are severely self-limited as to the amount of data they can handle, as main memory is fixed on a given machine. By extending memory-based query processors with the capability to spill data to disk, their scalability can be increased [39,107]. Like full-fledged XML databases, these query processors can handle large inputs. Nevertheless, the two-phase approach is intrinsically ill-suited for processing XML streams, as already discussed in the Introduction.

Streaming query evaluation. To scale up to large inputs and guarantee low response times at the same time, a *streaming* evaluation paradigm is required. That is, queries are processed as the input is read, and before an integrated representation of the complete input is available. In essence, this is an instance of the classic database principle of pipelining, or can even be seen as an instance of the problem of processing text files *as is* [102].

Several main memory-based systems with streaming capabilities have been shown to scale to XML streams for many practical queries (e.g. [6,36,49,69, 70, 75, 76, 85, 97, 100]). Interestingly, in some efforts towards developing main memory-based XQuery engines whose original emphasis was *not* on stream processing, such as BEA's XQRL [49], it has been observed that it is nevertheless worthwhile to employ stream processing operators. Yet naturally, there are also queries with blocking operators, or descendant axes and wildcards, where little can be evaluated on-the-fly [11, 42, 70, 75].

XQuery optimization. XQuery optimization is an active research area. For instance, XQueries may be optimized using type (or schema) information [28, 64]. Additionally, there are logical optimizations [28] on the level of the query language, such as the elimination of let-expression to obtain composition-free queries [68], and various algebras [43, 45, 78, 92, 101, 116].

XQuery differs from other query languages in its sensitivity regarding order and duplicates. Sorting and duplicate elimination are inherently blocking operations, and there is work on avoiding sorts [46,56,58]. Also, the efficient evaluation of structural joins (as opposed to value-based joins) in XML databases has received a lot of attention, which lead to novel algorithms for join evaluation and join reordering, e.g. [4,20,119]. In XML databases, the computation of structural joins is often accompanied by XML-specific indexing schemes, e.g. [86, 91, 108].

In main memory-based XQuery processors, a primary optimization target is the main memory consumption. Below, we consider solutions targeted at the internal representations of XML data as well as algorithms for prefiltering the input. We contrast event-based query evaluation, which operates directly on the input stream, with query evaluation over buffered data. We also address the purging of buffers during query evaluation.

Representations of the data model. Main memory-based XML query engines represent their input using DOM-like tree datastructures [40, 77, 93, 97, 111, 121], array-based encodings [90], or they store sequences of SAX-events in buffers [49, 70]. The input stored in main memory commonly requires significantly more space than the same document residing on disk, so that main memory becomes a crucial bottleneck [15,22,70,77]. In consequence, the internal representation of XML documents has attracted attention both from academic research and renowned projects in the open-source community, ranging from query-able compressions [21, 22] to on-demand loading of subtrees from disk, while the tree is traversed in main memory [121].

XML document projection. Among the early work on XQuery buffer management is XML document projection. The idea is to employ static query analysis to load only data relevant to query evaluation into main memory buffers. The work of [77] covers full XQuery with XPath downward axes. The benefits of this approach were quickly noted, and prefiltering has since been integrated both in open-source [77, 97] and commercial XQuery engines [32]. Successor systems additionally exploit schema information and filter for predicates [15].

In Section 3.5, we introduce our own approach to XML document projection. This approach can be implemented very efficiently, as it requires only a single pass over the input. In particular, the decision whether to discard a node is already made when its opening tag is read. Without the capability to buffer data or to exploit schema knowledge, there are cases where we can actually project out more nodes than competitor systems.

Scheduling of event-based query operators. In streaming query evaluation, parts of the query may be evaluated directly on the stream, while other parts must be evaluated over buffered data. For this purpose, some query algebras have dedicated streaming operators, e.g. [44, 70, 101]. Typically, operator scheduling takes place at query compile time [44,70], yet [101] also discusses the potential of cost-based operator scheduling at runtime. In the Galax XQuery engine [44], a physical algebra for full XQuery with streaming operators is used. The approach of the FluXQuery engine from Chapter 8 was the first to mix operators for query-processing over streams and buffers. It can further be seen as more general than the Galax approach, as it does not specify the physical query plans in detail. Rather, the FluXQuery algorithm specifies *which* parts of the query are evaluated directly on the stream and *which* are evaluated over buffered data. Moreover, in doing so FluXQuery can factor in schema information.

The system from [75] also performs data dependency analysis on queries for the compilation of query execution plans. However, the focus is on computing XQuery aggregations on-the-fly, while the streaming evaluation of the other language primitives is disregarded.

Purging buffers of irrelevant data. In evaluating XQueries in a buffer-conscious manner, data that is no longer relevant for query evaluation needs to be purged from buffers. In *static* buffer management, as in the case of our FluXQuery engine (see Chapter 8) and similarly in [75], 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, avoiding that data is buffered redundantly becomes a challenge. Such situations arise if the same XML node is bound by different variables during query evaluation, e.g. as the node is required for checking a condition as well as for producing output. As we discuss in Section 6.3.1, it can be infeasible to avoid duplicate buffering based on static analysis alone.

Here, a *dynamic* approach is required. In the GCX XQuery engine (see Chapter 9), we implement a novel buffer purging algorithm. Our approach is related to garbage collection. Yet while garbage collection mechanisms have also been implemented in other main memory-based XML processors, e.g. see [50], they rely exclusively on reference counting to determine which buffer contents are no longer reachable. There is earlier work where garbage collection is employed by the database community, namely in object-oriented databases [9]. However, these earlier approaches all rely on the analysis of references between objects. In contrast, we statically analyze queries to derive data access patterns, which center around the notion of the relevance of data to query evaluation. This makes it possible to delete data from buffers once we know from the state of query evaluation that it is no longer required, even if it is still referenced inside datastructures internal to the query engine.

3 Technical Background

This chapter gives an overview over the main theoretical tools that are used in this dissertation. Throughout, we assume that all XML documents are well-formed, where we also consider empty documents to be well-formed. We abstract from attributes in the XML document model, as these can always be modeled as children of an element node. We assume that all tagnames in XML documents stem from a set *Tag*, and that all textual data in XML documents stems from a set *Char* of characters. We further assume familiarity with the basics of regular expressions, finite-state automata, and grammars (c.f. [60]).

We begin this chapter by introducing formalisms from automata theory for XML processing. In Section 3.1, we present known results on one-unambiguous regular expressions. In Section 3.2, we define a class of pushdown transducers which is expressly suited for XML processing. This formalism captures a large class of XML filtering tasks, and even certain XML stream transformations. We review DTDs as a grammar formalism for XML documents in Section 3.3.

In Section 3.4, we define a syntactic XQuery fragment along with some terminology and rules for query normalization. In Section 3.5, we define our notion of XML document projection based on static query analysis. Finally, we introduce the concept counting the number of ways in which a path is matched by a node in an XML document (see Section 3.6).

3.1 One-Unambiguous Regular Expressions

We assume the usual notion of regular expressions. By $L(\rho)$, we denote the language defined by the regular expression ρ, and by $symb(\rho)$ the set of atomic symbols that occur in ρ. By a *marking* of a regular expression ρ over an alphabet Σ, we denote a regular expression ρ' where each occurrence of an atomic symbol in ρ is replaced by the symbol with its position among the atomic symbols of ρ added as subscript. For instance, the marking of $(a \cup b)^*.a.a^*$ is $(a_1 \cup b_2)^*.a_3.a_4^*$. The reverse of a marking, indicated by $^\#$, is obtained by dropping the subscripts.

Ambiguity in regular languages. A regular expression ρ is called *ambiguous* [16] if there are two words $w_1, w_2 \in L(\rho')$ such that $w_1 \neq w_2$ but $w_1^\# = w_2^\#$. A regular expression is called *unambiguous* if it is not ambiguous.

Example 3.1 The language defined by the regular expression $\rho = (a \cup b)^*.a.a^*$ is ambiguous because the language defined by its marking $\rho' = (a_1 \cup b_2)^*.a_3.a_4^*$ contains the words $a_1.a_3$ and $a_3.a_4$, which correspond to the word aa in $L(\rho)$.□

Definition 3.1 ([19]) Let ρ be a regular expression, ρ' its marking, and $\Sigma' = \mathit{symb}(\rho')$ the marked alphabet used by ρ'. Then ρ is called *one-ambiguous* iff there are words u, v, w over Σ' and symbols $x \neq y \in \Sigma'$ with $x^\# = y^\#$ such that $uxv, uyw \in L(\rho')$. A regular expression is called *one-unambiguous* if it is not one-ambiguous. □

Intuitively, a one-unambiguous regular expression ρ allows us to determine which atomic symbol in ρ matches the next symbol from an input word $w \in L(\rho)$ while we parse w from left to right with a lookahead of one token.

Example 3.2 Consider the regular expression $\rho = a^*.a$ and its marking $\rho' = a_1^*.a_2$. Let $u = a_1$, $x = a_2$, $v = \epsilon$, $y = a_1$, and $w = a_2$. Clearly, $uxv = a_1a_2$ and $uyw = a_1a_1a_2$ are both words of $L(\rho')$, so ρ is one-ambiguous. The equivalent regular expression $a.a^*$ is one-unambiguous. □

We next introduce Glushkov automata, which are finite-state automata that lend themselves nicely for checking one-unambiguity of regular expressions.

Glushkov automata. For a given regular language L over alphabet Σ, the set *first*(L) consists of precisely those symbols x such that there is a word w with $xw \in L$. For each $x \in \Sigma$, the set *follow*(L, x) consists of those symbols y such that there are words v, w with $vxyw \in L$. Finally, *last*(L) consists of those symbols x such that there is a word w with $wx \in L$.

Definition 3.2 ([19]) Let ρ be a regular expression and let ρ' be its marking. The *Glushkov automaton* of ρ is the nondeterministic finite-state automaton $\mathcal{G}(\rho) = (Q, \mathit{symb}(\rho), \delta, q_0, F)$ with

- $Q = \mathit{symb}(\rho') \cup \{q_0\}$, i.e. the states of $\mathcal{G}(\rho)$ contain the marked symbols in ρ' and a new initial state q_0.
- For each $a \in \mathit{symb}(\rho)$, $\delta(q_0, a) = \{x \mid x \in \mathit{first}(\rho') \wedge x^\# = a\}$.
- For $x \in \mathit{symb}(\rho')$, $a \in \mathit{symb}(\rho)$, $\delta(x, a) = \{y \mid y \in \mathit{follow}(\rho', x) \wedge y^\# = a\}$.
- and $F = \mathit{last}(\rho') \cup \{q_0 \mid \epsilon \in L(\rho)\}$. □

The Glushkov automaton for a regular expression has no transitions that lead into the initial state. Any two transitions that lead into the same state have identical labels [19], a property called *homogeneity* [25]. A state in a homogeneous automaton is *a-labeled* if its incoming transitions carry label a.

Example 3.3 The automata from Figure 3.1 are all homogeneous. All states a_1 and a_2 are a-labeled, while state b_2 is b-labeled. □

Proposition 3.1 ([19]) *Let ρ be a regular expression. Then ρ is one-unambiguous iff its Glushkov automaton $\mathcal{G}(\rho)$ is deterministic.*

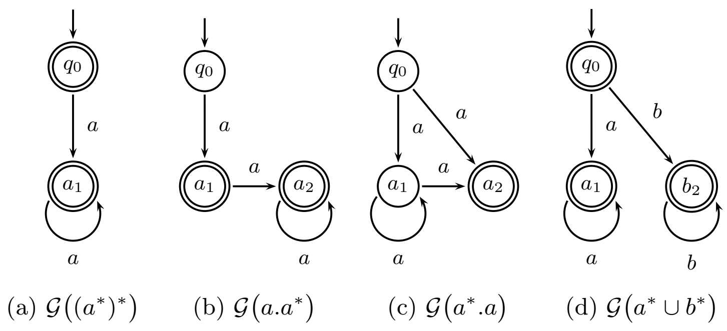


Figure 3.1: *Glushkov automata.*

It follows from this proposition and the definition of Glushkov automata that deterministic finite-state automata (DFAs) can be constructed from one-unambiguous regular expressions in just quadratic time [19].

Example 3.4 We consider the regular expressions $(a^*)^*$, $a.a^*$, and $a^* \cup b^*$. They each are one-unambiguous as the corresponding Glushkov automata in Figures 3.1(a), (b), and (d) are deterministic. However, the Glushkov automaton of the regular expression $a^*.a$ in Figure 3.1(c) has several distinct transitions leading from states q_0 and a_1 under input symbol a, so it is not deterministic. It follows from the above Proposition that $a^*.a$ is not one-unambiguous. $\square$

Order constraints. We next define a set of *order constraints* to be checked on regular expressions. In [71], we have shown how these constraints can be computed efficiently for one-unambiguous regular expressions.

In the following let ρ be a regular expression over alphabet Σ. Given a word w, let w_i denote its i-th symbol.

Definition 3.3 We define a binary relation $Ord_\rho \subseteq \Sigma \times \Sigma$ such that for $a, b \in \Sigma$, $Ord_\rho(a, b) \;:\Leftrightarrow\; \nexists w \in L(\rho) : w_i = b \wedge w_j = a \wedge i < j.$ $\square$

Intuitively, order constraint $Ord_\rho(a, b)$ holds if there is no word in $L(\rho)$ in which a symbol a is preceded by a symbol b. In other words, all a-symbols occur before all b-symbols.

Example 3.5 Let $\rho = (a^*.b.c^*.(d|e^*).a^*)$. Then we have $Ord_\rho(b, c)$, $Ord_\rho(c, d)$, and $Ord_\rho(c, e)$, but $\neg Ord_\rho(a, c)$. Relation Ord_ρ is transitive, so $Ord_\rho(b, d)$. $\square$

Definition 3.4 Let ρ be a regular expression and let S be a set of symbols. Then for each word $u = u_1 \dots u_n \in \Sigma^*$, we define

$$\begin{aligned} Past_{\rho,S}(u) \quad &:\Leftrightarrow \quad \forall w \in \Sigma^* : uw \in L(\rho) \rightarrow \nexists\, i \;:\; w_i \in S \\ \textit{first-past}_{\rho,S}(u) \quad &:\Leftrightarrow \quad Past_{\rho,S}(u) \;\wedge\; \big(n > 0 \rightarrow \neg Past_{\rho,S}(u_1 \dots u_{n-1})\big). \end{aligned}$$ $\square$

Intuitively, when processing a word uw in $L(\rho)$ from left to right, if condition $\textit{first-past}_{\rho,S}(u)$ holds, then the moment when we read the last symbol of u is the earliest possible time at which we know that none of the symbols in S can be seen anymore until the end of the word uw.

3.2 XML Stream Pushdown Transducers

We next define *XML-DPDT*s, as introduced in our earlier work [69]. These are deterministic pushdown transducers with a restricted stack discipline that is natural in the context of XML stream processing. With *XML-DPDT*s, the size of the stack is bounded by the maximum depth of the incoming document tree. Items are pushed on the stack for reading opening tags, and are popped from the stack for matching closing tags.

We first introduce deterministic pushdown transducers (*DPDT*s) as deterministic pushdown automata with output and which accept by empty stack. As with pushdown automata [3], the *DPDT*s accepting by empty stack are equivalent to the *DPDT*s accepting by final state.

Definition 3.5 A *deterministic pushdown transducer* is a tuple

$$\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$$

where Q is a finite set of states, Σ, Γ, and Δ are the finite alphabets for input tape, stack, and output tape respectively, δ is the partial transition function

$$\delta \; : \; Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^* \times \Delta^*,$$

q_0 denotes the initial state, and Z_0 the initial stack symbol. For each $q \in Q$ and $X \in \Gamma$ such that $\delta(q, \epsilon, X)$ is defined, $\delta(q, a, X)$ is undefined for all $a \in \Sigma$. A transition $\delta(q, \epsilon, X)$ is an *ϵ-transition.* A DPDT without ϵ-transitions is *ϵ-free.*

We define a run of $\mathcal{T}$ by means of *instantaneous descriptions* (IDs), which describe the configurations of a DPDT at a given instant. An ID is a quadruple

$$(q, w, \alpha, o) \;\in\; Q \times \Sigma^* \times \Gamma^* \times \Delta^*$$

where q is a state, w is the remaining input, α a string of stack symbols denoting the current stack, and o the output generated so far. We make a transition

$$(q, aw, X\alpha, o) \vdash (q', w, \gamma\alpha, o\sigma)$$

if $\delta(q, a, X) = (q', \gamma, \sigma)$, where $a \in \Sigma \cup \{\epsilon\}$, $X \in \Gamma$, $\alpha \in \Gamma^*$, $q' \in Q$, and $\sigma \in \Delta^*$. Here, $\gamma \in \Gamma^*$ is the string of stack symbols which replace X on top of the stack. For $\gamma = \epsilon$, the stack is popped, whereas for $\gamma = X$, the stack remains unchanged. If $\gamma = YX$, then Y is pushed on top of X.

Let $\vdash^*$ be the reflexive and transitive closure of $\vdash$. $\mathcal{T}$ accepts an input word $w \in \Sigma^*$ by empty stack if $(q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, o)$ for $q \in Q$ and $o \in \Delta^*$. We say o is the output for input w. The language accepted by a DPDT $\mathcal{T}$, denoted $L(\mathcal{T})$, is the set of strings accepted by $\mathcal{T}$. □

For a set S, we use $S^{\leq 2}$ as a shortcut for $\{\epsilon\} \cup S \cup (S \times S)$.

Definition 3.6 Let the input alphabet

$$\Sigma = \{\langle t \rangle \mid t \in \mathit{Tag}\} \cup \{\langle /t \rangle \mid t \in \mathit{Tag}\} \cup \mathit{Char}$$

consist of matching XML opening and closing tags, as well as character symbols.

An *XML-DPDT* is a *DPDT* $\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$ where $\Gamma = \{Z_0\} \cup (\mathit{Tag} \times \Gamma')$, so the stack alphabet consists of stack start symbol Z_0, and pairs of tags and symbols from some set Γ'. The transition function

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^{\leq 2} \times \Delta^*$$

is restricted as follows:

1. In the very first transition, the initial stack symbol is replaced; we require

   $$\delta\big(q_0, \langle t\rangle, Z_0\big) = \big(p, (t, Y), \sigma\big)$$

   for $\langle t\rangle \in \Sigma$, $p \in Q$, $(t, Y) \in \Gamma$, and $\sigma \in \Delta^*$.

2. For all other configurations of $q \in Q$ and $X \in \Gamma$, a symbol is pushed on the stack when an opening tag is read from the input stream. We require

   $$\delta\big(q, \langle t\rangle, X\big) = \big(p, (t, Y)X, \sigma\big)$$

   for $\langle t\rangle \in \Sigma$, $p \in Q$, $(t, Y) \in \Gamma$, and $\sigma \in \Delta^*$.

3. A symbol is popped from the stack when a matching closing is encountered in the input stream, so

   $$\delta\big(q, \langle/t\rangle, (t, Y)\big) = \big(p, \epsilon, \sigma\big)$$

   for $p, q \in Q$, $\langle/t\rangle \in \Sigma$, $(t, Y) \in \Gamma$, and $\sigma \in \Delta^*$. □

Example 3.6 The following *XML-DPDT* accepts only XML documents where all nodes are labeled differently from their parents. To communicate that a document has been accepted, the transducer outputs "⟨*accept*⟩⟨/*accept*⟩", otherwise it outputs "⟨*reject*⟩⟨/*reject*⟩".

We define this *XML-DPDT* for the set of states $\{q_0, q_1, q_2\}$. The idea in defining the transition function is the following. The *XML-DPDT* will start in state q_0, and enter state q_1 for the first opening tag in the input. It remains in state q_1 unless a node is encountered that carries the same label as its parent. Then the designated error state q_2 is entered. We will define no outgoing transitions for this state. In order to compare the labels of nodes with those of their parents, we push labels on the stack. To this end, we define the stack alphabet $\Gamma = \{Z_0\} \cup (\mathit{Tag} \times \{q_0, q_1, q_2\})$ for the initial stack symbol Z_0. Then we can store both the label of the parent node and the state entered when processing its opening tag on the stack.

We now define the transducer transitions accordingly. For each tagname t, we define an initial transition

$$\delta(q_0, \langle t\rangle, Z_0) = (q_1, (t, q_0), \epsilon)$$

that processes the first opening tag in the input, and replaces the initial stack symbol by the tagname and the initial state.

For all opening tags encountered in state q_1, we check whether the tagname differs from the tagname of the parent node, which is stored on the stack. We next consider all states q that are either q_0 or q_1.

- Let us first consider the case where the current node has the same label as its parent. For all tagnames t, we define a transition

$$\delta(q_1, \langle t\rangle, (t, q)) = (q_2, (t, q_2)(t, q), \langle \mathit{reject}\rangle\langle/\mathit{reject}\rangle).$$

  The *XML-DPDT* outputs "$\langle \mathit{reject}\rangle\langle/\mathit{reject}\rangle$" and enters the error state q_2. No outgoing transitions are defined for q_2, so the stack cannot be emptied anymore. As *XML-DPDT*s accept by empty stack, the input is rejected.

- For all distinct tagnames a and b, we define a transition

$$\delta(q_1, \langle a\rangle, (b, q)) = (q_1, (a, q_1)(b, q), \epsilon)$$

  where the current tagname and state are pushed onto the stack.

Transitions reading character data leave the current stack and state unchanged. So for all characters c and all states q, we define $\delta(q, c, X) = (q, X, \epsilon)$.

It remains to specify the transitions for reading closing tags. This is also done for all tagnames t:

- When opening and closing tags do not match, the transducer rejects the input. For state q either q_0 or q_1 and distinct tagnames a and b, we define the transition $\delta(q_1, \langle/a\rangle, (b, q)) = (q_2, \epsilon, \langle \mathit{reject}\rangle\langle//\mathit{reject}\rangle)$.

- We distinguish the case when the last token from the input stream is read, which is the case when state q_0 is on top of the stack. With executing transition $\delta(q_1, \langle/t\rangle, (t, q_0)) = (q_1, \epsilon, \langle \mathit{accept}\rangle\langle/\mathit{accept}\rangle)$ the stack becomes empty and the transducer accepts the input.

- For all other closing tags, we define $\delta(q_1, \langle/t\rangle, (t, q_1)) = (q_1, \epsilon, \epsilon)$.

Given the input "$\langle a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle$", we show a run of this transducer below.

$$\begin{array}{llrrl}
 & (q_0, & \langle a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle, & Z_0, & \epsilon) \\
\vdash & (q_1, & \langle b\rangle\langle/b\rangle\langle/a\rangle, & (a, q_0), & \epsilon) \\
\vdash & (q_1, & \langle/b\rangle\langle/a\rangle, & (b, q_1)(a, q_0), & \epsilon) \\
\vdash & (q_1, & \langle/a\rangle, & (a, q_0), & \epsilon) \\
\vdash & (q_1, & \epsilon, & \epsilon, & \langle \mathit{accept}\rangle\langle/\mathit{accept}\rangle)
\end{array}$$

Note that the depth of the *XML-DPDT* stack is bounded by the depth of the input document tree. □

3.3 Document Type Definitions

Document type definitions (DTDs) [117] are deterministic context-free grammars for defining XML documents. We assume familiarity with common grammar formalisms [60], and merely introduce DTDs on a syntactic level. Note that in DTDs, we do not distinguish between nonterminals and terminals.

Syntax. DTDs are grammars of the form

`<!DOCTYPE` *root element* `[` *element definitions* `] >`

where the *root element* specifies the grammar start symbol. The element definitions are specified as a list. Each single element definition in this list is of the form "`<!ELEMENT` *tagname* `(`ρ`)>`" with a tagname and a one-unambiguous regular expression ρ over tagnames and the token `#PCDATA`. Token `#PCDATA` denotes character content. The designated term `EMPTY` indicates that ρ is the empty regular expression. The operators allowed in specifying ρ are parentheses, the union "`|`", comma for concatenation, the Kleene star "`*`" for any number of occurrences, and "`+`" denoting at least one occurrence. The intuition for an element definition is that the tagname specifies the label of a given node, while the regular expression defines the possible labels and order of its children.

```
<!DOCTYPE site [
   <!ELEMENT site       (regions)>
   <!ELEMENT regions    (africa, asia, australia)>
   <!ELEMENT africa     (item*)>
   <!ELEMENT asia       (item*)>
   <!ELEMENT australia  (item*)>
   <!ELEMENT item  (location,name,payment,description,
                  shipping,incategory+)>
   <!ELEMENT location    (#PCDATA)>
   <!ELEMENT name        (#PCDATA)>
   <!ELEMENT payment     (#PCDATA)>
   <!ELEMENT description (#PCDATA)>
   <!ELEMENT shipping    (#PCDATA)>
   <!ELEMENT incategory   EMPTY>  ...  ]>
```

Figure 3.2: *Excerpt from a DTD defining auction data.*

Example 3.7 Figure 3.2 shows an excerpt of a non-recursive DTD that defines transactions in an online auction, in the style of the XMark benchmark [122]. Note that any XML document adhering to this DTD contains a site-node as top-level node. □

Validation. We denote the language defined by a DTD D by $L(D)$. We say an XML document is *valid* w.r.t. a DTD D if it is in $L(D)$. As $L(D)$ is a deterministic context-free language, validation can be realized by deterministic pushdown automata, or the *XML-DPDT*s previously introduced. Of course, for validation against non-recursive DTDs, finite-state automata are sufficient.

3.4 A Composition-free Fragment of XQuery

We study fragments of composition-free XQuery [67]. This implies that during query evaluation, query-variables only bind to nodes from the input document tree. The first query fragment, called XQ, comprises queries with the XPath

$$\begin{array}{rcl}
XQ & ::= & \langle a \rangle q \langle /a \rangle \\
q & ::= & () \mid \langle a \rangle q \langle /a \rangle \mid var \mid var/axis :: \nu \mid (q,\ \ldots\ , q) \\
 & \mid & \text{for } var \text{ in } var/axis :: \nu \text{ [where } cond\text{] return } q \\
 & \mid & \text{if } cond \text{ then } q \text{ else } q \mid string \\
cond & ::= & \text{true()} \mid \text{false()} \mid \text{exists } var/axis :: \nu \mid (cond) \\
 & \mid & cond \text{ and } cond \mid cond \text{ or } cond \mid \text{not } cond \\
 & \mid & operand\ RelOp\ operand \\
 & \mid & \text{some } var \text{ in } var/axis :: \nu \text{ satisfies } cond \\
operand & ::= & var \mid var/axis :: \nu \mid string \\
axis & ::= & \text{child} \mid \text{descendant} \\
\nu & ::= & a \mid * \mid \text{text()} \mid \text{node()} \\
RelOp & ::= & \leq \mid < \mid = \mid \geq \mid >
\end{array}$$

Figure 3.3: *The XQuery fragment XQ.*

axes *child* and *descendant* and the node tests for tagnames, the wildcard "$*$", and further "text()" and "node()". The second fragment, which we call XQ^-, is a subset of XQ. It is restricted to the XPath axis *child* and node tests for tagnames only.

With nested for-loops, conditionals, and joins, these fragments nevertheless cover many practical queries: While we only allow single-step XPath expressions, e.g. of the form $x/a but not $x/a/b, many cases of multi-step XPath expressions can be rewritten into equivalent nested for-loops with single-step expressions. Further, we argue that many practical queries can be specified without composition. Only a handful of the queries from the XQuery Use Cases actually employ composition [123], and in the majority of the XMark queries [122], composition can be removed using the algorithm from [68].

Syntax. Figure 3.3 shows the abstract syntax of our XQuery fragment XQ where *var* ranges over the set of XQuery variables and a over the set of tagnames. In the following, we frequently refer to variables in XQuery expressions as *query-variables*, or merely *variables* when the context is clear. By *string* we denote string constants that are enclosed in quotation marks.

The fragment XQ^- is obtained by restricting *axis* to the XPath axis *child*, and the node test ν to tests for tagnames only. In the following, we assume the XQuery fragment XQ unless we state otherwise.

For syntactic convenience, we make use of the XPath abbreviations "/" and "//" for the child- and descendant axis when specifying XQueries.

Normalization. To simplify static query analysis, we consider only a reduced number of syntactic constructs. We normalize queries in two steps.

(1) By renaming query-variables, we ensure that each for-loop or some-expression binds a variable of a different name. In particular, we rename query-variables such that no variable uses the reserved name "$root". (2) We then apply the rewriting rules from Figure 3.4 until no further change is possible. The rewriting rules are specified in the tradition of inference rules, and read as follows. If a query expression matches the premise of a rule, it is rewritten

$$(n1) \qquad \frac{\text{for } \$x \text{ in } \sigma \text{ where } \chi \text{ return } \beta}{\text{for } \$x \text{ in } \sigma \text{ return } (\text{if } \chi \text{ then } \beta \text{ else } (\,))}$$

$$(n2) \qquad \frac{\$y/\mathit{axis} :: \nu}{\text{for } \$x_0 \text{ in } \$y/\mathit{axis} :: \nu \text{ return } \$x_0} \ (x_0 \ \mathit{new})$$

$$(n3) \qquad \frac{\text{exists } \$y/\mathit{axis} :: \nu}{\text{some } \$x_0 \text{ in } \$y/\mathit{axis} :: \nu \text{ satisfies true()}} \ (x_0 \ \mathit{new})$$

$$(n4) \qquad \frac{\$y/\mathit{axis} :: \nu \ \mathit{RelOp} \ \alpha}{\text{some } \$x_0 \text{ in } \$y/\mathit{axis} :: \nu \text{ satisfies } (\$x_0 \ \mathit{RelOp} \ \alpha)} \ (x_0 \ \mathit{new})$$

$$(n5) \qquad \frac{\alpha \ \mathit{RelOp} \ \$y/\mathit{axis} :: \nu}{\text{some } \$x_0 \text{ in } \$y/\mathit{axis} :: \nu \text{ satisfies } (\alpha \ \mathit{RelOp} \ \$x_0)} \ (x_0 \ \mathit{new})$$

$$(n6) \qquad \frac{/\mathit{axis} :: \nu}{\$\text{root}/\mathit{axis} :: \nu}$$

***Figure 3.4**: XQuery normalization rules.*

$$\begin{array}{rcl}
Q' & ::= & \langle a \rangle \ q' \langle /a \rangle \\
q' & ::= & () \mid \langle a \rangle q' \langle /a \rangle \mid \mathit{var} \mid (q', \ \ldots \ , q') \\
 & \mid & \text{for } \mathit{var} \text{ in } \mathit{var}/\mathit{axis} :: \nu \text{ return } q' \\
 & \mid & \text{if } \mathit{cond}'a \text{ then } q' \text{ else } q' \ \mid \ \mathit{string} \\
\mathit{cond}' & ::= & \text{true()} \ \mid \ \text{false()} \ \mid \ \mathit{operand}' \ \mathit{RelOp} \ \mathit{operand}' \\
 & \mid & \text{some } \mathit{var} \text{ in } \mathit{var}/\mathit{axis} :: \nu \text{ satisfies } \mathit{cond}' \\
 & \mid & \mathit{cond}' \text{ and } \mathit{cond}' \ \mid \ \mathit{cond}' \text{ or } \mathit{cond}' \ \mid \ \text{not } \mathit{cond}' \\
\mathit{operand}' & ::= & \mathit{var} \ \mid \ \mathit{string}
\end{array}$$

***Figure 3.5**: The normalized XQuery fragment XQ.*

as specified in the conclusion of this rule. In doing so, it may be necessary to introduce fresh query-variables.

Rule $n1$ translates where-conditions into if-then-else statements with an empty else-part. Rule $n2$ introduces for-loops for XPath expressions that produce output. Note that a fresh query-variable is introduced. Likewise, we rewrite XPath expressions inside conditionals using some-expressions and introducing new variables, as done by rules $n3$ through $n5$.

In rule $n6$, we translate absolute path expression into references to an implicit query-variable named "$root". As a syntactical convention, we will hide the effect of applying this step when showing example queries. The intention behind this convention is that example queries can then be directly evaluated by any XQuery processor implementing the XQuery standard.

Note that the result of normalization is not necessarily unique, but that normalized queries are always equivalent to the original query.

The abstract syntax of normalized queries is shown in Figure 3.5. The productions for *axis*, *RelOp*, and node test ν are defined as in Figure 3.3.

```
<results>
{ for $bib in /bib return
  (for $b in $bib/book
   where ( exists($b/price) )
   return $b,


   for $a in $bib/article
   where (some $x1 in $bib/book
          satisfies
            ($a/author=$x1/editor))
   return $a ) }
</results>
```

(a)

```
<results>
{ for $bib in $root/bib return
  ( for $b in $bib/book return
    if (some $x2 in $b/price
        satisfies true())
    then $b else ( ),
    for $a in $bib/article return
    if (some $x1 in $bib/book
        satisfies
          (some $x3 in $x1/editor
           satisfies
             (some $x4 in $a/author
              satisfies ($x3=$x4))))
    then $a else ( ) ) }
</results>
```

(b)

***Figure 3.6**: Translation of query (a) into normal form (b).*

Example 3.8 In normalizing the query from Figure 3.6(a), we can obtain the query from Figure 3.6(b). □

Query-variables. We next introduce some terminology for query-variables. We assume that all queries are normalized.

Free and bound variables. We introduce the common notions of bound and free query-variables in analogy to the free variables of a formula in first-order logic. In the table in Figure 3.7, we inductively define free and bound query-variables in normalized XQuery expressions. The first column describes the XQuery expression. In the second and third columns, we state how the free and bound variables are determined for the given expression. The last row concerns all XQuery expressions that have not been explicitly listed.

expression α	$\mathit{freeVar}(\alpha)$	$\mathit{boundVar}(\alpha)$
$\langle a\rangle q\langle/a\rangle$	$\mathit{freeVar}(q)$	$\mathit{boundVar}(q)$
$\$x$	$\{\$x\}$	$\emptyset$
for $\$x$ in $\$y/\pi$ return q	$\{\$y\} \cup \mathit{freeVar}(q) \setminus \{\$x\}$	$\{\$x\} \cup \mathit{boundVar}(q)$
if χ then q_1 else q_2	$\bigcup_{e\in\{\chi,q_1,q_2\}} \mathit{freeVar}(e)$	$\bigcup_{e\in\{\chi,q_1,q_2\}} \mathit{boundVar}(e)$
some $\$x$ in $\$y/\pi$ satisfies χ	$\{\$y\} \cup \mathit{freeVar}(\chi) \setminus \{\$x\}$	$\{\$x\} \cup \mathit{boundVar}(\chi)$
o_1 *RelOp* o_2	$\bigcup_{e\in\{o_1,o_2\}} \mathit{freeVar}(e)$	$\bigcup_{e\in\{o_1,o_2\}} \mathit{boundVar}(e)$
χ_1 and χ_2	$\bigcup_{e\in\{\chi_1,\chi_2\}} \mathit{freeVar}(e)$	$\bigcup_{e\in\{\chi_1,\chi_2\}} \mathit{boundVar}(e)$
χ_1 or χ_2	$\bigcup_{e\in\{\chi_1,\chi_2\}} \mathit{freeVar}(e)$	$\bigcup_{e\in\{\chi_1,\chi_2\}} \mathit{boundVar}(e)$
not χ	$\mathit{freeVar}(\chi)$	$\mathit{boundVar}(\chi)$
. . . all others. . .	$\emptyset$	$\emptyset$

***Figure 3.7**: Free and bound query-variables.*

Example 3.9 Given query Q from Figure 3.6(b), $\mathit{freeVar}(Q) = \{\$\mathrm{root}\}$ and $\mathit{boundVar}(Q)$ comprises all other query-variables occurring in Q. Let us now consider the subexpression which spans lines 3 through 6. Here, \$bib is the only free variable, and the variables \$b and \$x2 are bound. □

Parent variables. We assume an XQuery with a subexpression β. Then the parent variable of β is determined by the least ancestor expression in the query parse tree which is either a for-loop or a some-expression binding a variable $\$x$. Then $\$x$ is the parent variable of expression β, which we denote by $\mathit{parentVar}(\beta) = \$x$. If an expression has no for-loops or some-expressions among its ancestor expressions, then variable \$root is its parent variable.

Example 3.10 Consider the XQuery in Figure 3.6(b). For the query expression spanning lines 1 through 15, the parent variable is \$root. The same holds for the expression spanning lines 2 trough 14. The parent variable of the expression from lines 3 through 6 is \$bib, and the parent variable for the expression from lines 4 and 5 is \$b. The subexpressions "`$b`" and "`( )`" in line 6 both have \$b as parent variable. □

Variable lineage. Given a query from our fragment, we define the concept of *lineage paths* between two distinct variables $\$x$ and $\$y$. Variable lineage is derived from normalized queries in a closure computation:

1. For each for-loop "for $\$x$ in $\$y/\pi$ return α" or a some-expression "some $\$x$ in $\$y/\pi$ satisfies α", we define $\mathit{lineage}(\$y, \$x) = \$y/\pi$.

2. Next, we repeatedly consider each pair of variables $\$x$ and $\$y$ such that $\mathit{lineage}(\$y, \$x)$ is defined, If there exists a for-loop "for $\$z$ in $\$x/\pi$ return α" or some-expression "some $\$z$ in $\$x/\pi$ satisfies α", then we define $\mathit{lineage}(\$y, \$z) = \mathit{lineage}(\$y, \$x)/\pi$.

Example 3.11 Given the query from Figure 3.6(a), we derive lineage paths for selected variables as summarized in the table below. Columns separate different variable names for $\$y$, and rows distinguish different variable names $\$x$.

$\$x$	*lineage*(\$root,$\$x$)	*lineage*(\$bib,$\$x$)	*lineage*(\$b,$\$x$)	*lineage*(\$a,$\$x$)
\$bib	\$root/bib			
\$b	\$root/bib/book	\$bib/book		
\$x2	\$root/bib/book/price	\$bib/book/price	\$b/price	
\$a	\$root/bib/article	\$bib/article		
\$x1	\$root/bib/book	\$bib/book		
\$x3	\$root/bib/book/editor	\$bib/book/editor		
\$x4	\$root/bib/article/author	\$bib/article/author		\$a/author

For instance, the table encodes that $\mathit{lineage}(\$\text{bib}, \$\text{b})$ =\$bib/book and further that $\mathit{lineage}(\$\text{bib}, \$\text{x2})$ = \$bib/book/price. The entries for the empty cells in the table are not defined. □

Semantics. *XQ* queries are evaluated according to the usual XQuery semantics [116]. We further assume the designated query-variable \$root, which will bind to the root of the XML input document.

Definition 3.7 An *XQuery* is an *XQ* expression in which all free variables except for the special variable \$root corresponding to the root of the document are bound. □

3.5 XML Document Projection

XML document projection is an established technique to reduce the memory consumption of main memory-based XML processors. It was first implemented for the Galax XQuery processor [77]. The idea is to load only data relevant to query evaluation into main memory buffers. This prefiltering of XML documents is based on projection paths. The projection paths are extracted in static query analysis, and capture a superset of the data that is actually relevant for query evaluation. In reverse, this means that all input data that is not captured by projection paths is irrelevant and can be filtered out.

We next define our own notion of projection paths and their semantics. As our XQuery fragment is more restricted than the one covered in [77], our path extraction algorithm is easier to define. Our projection semantics can be implemented such that prefiltering occurs in a single scan over the input, even for queries with the descendant axis, as possible in [15] but not in [77]. At the same time, our approach often allows us also to discard nodes in the document tree which cannot be discarded by other projection algorithms [15, 77].

3.5.1 Projection Paths

In introducing projection paths, we lean on the concept from [77] and the standard XPath notation. The syntax of projection paths is defined by the grammar below, with XPath axes *axis* and node tests ν as defined in Figure 3.3. We use the flag "`#`" to indicate that the descendants of selected nodes are also relevant to query evaluation. The semantics of this flag is equivalent to the relative XPath step expression `./descendant-or-self::node()`.

$$\begin{array}{rcl} \mathit{ProjectionPath} & ::= & /\mathit{SimplePath} \mid /\mathit{SimplePath}\texttt{\#} \\ \mathit{SimplePath} & ::= & \mathit{axis} :: \nu \mid \mathit{SimplePath}/\mathit{SimplePath} \end{array}$$

We next define the extraction of projection paths from XQueries in static analysis. Note that projection paths may also be extracted from other XML transformation languages, such as XSLT [115]. For our XQuery fragment, we associate projection paths with query-variables. The mapping from query-variables to projection paths is denoted by the function *pp*. The following definition uses the notion of variable lineage, as introduced in Section 3.4.

Definition 3.8 Let Q be a normalized XQuery and let $\$x$ be a query-variable such that $\mathit{lineage}(\$\mathrm{root}, \$x)$ is defined. Let $\mathit{lineage}(\$\mathrm{root}, \$x)$ be $\$\mathrm{root}/\pi$ for some path expression π. Then the *projection path associated with* $\$x$, denoted $\mathit{pp}(\$x)$, is $/\pi\texttt{\#}$ if $\$x$ occurs in an output expression or a value comparison inside a condition, and $/\pi$ otherwise. □

By convention, we simplify all projection paths of the form "$/\pi/\mathit{axis}::\,\mathrm{text}()\texttt{\#}$" to "$/\pi/\mathit{axis}::\,\mathrm{text}()$".

Example 3.12 We extract projection paths from the query from Figure 3.6(b). Figure 3.8 shows the function *pp* in tabular notation, where we list the query-variables in order of their specification. □

$\$x$	$pp(\$x)$
\$bib	/bib
\$b	/bib/book#
\$x2	/bib/book/price
\$a	/bib/article#
\$x1	/bib/book
\$x3	/bib/book/editor#
\$x4	/bib/book/article#

Figure 3.8*: Projection path mapping of Example 3.12.*

3.5.2 Automata-based Path Matching

In streaming scenarios, we commonly use pushdown automata to match paths against the XML stream [6,36,54]. These automata operate on a low level of abstraction which makes it difficult to track the progress in path matching. Apart from which paths have been matched, it may be necessary to know which paths can still be matched by the descendants of a given node. Out of this motivation, we introduce path matching automata that make the information represented by each state is explicit. This allows us to provide a concise definition of our projection semantics. The formal basis is an extension of the established parsing technique of Earley-style dotted items [38,55] to path matching.

Definition 3.9 Let P be a projection path. A *dotted path* is an expression of the form $P \to \alpha \bullet \beta$, with α and β expressions such that $P = \alpha\beta$. □

Intuitively, the dot separates the matched part of the input from the unmatched part. When processing the XML document tree top-down, we shift the dot in the dotted path whenever we move from a parent-node to a child-node. This process is specified by the path matching rules introduced next.

Path matching rules. In matching a path, the dot is initially placed in front of the projection path. So we begin with an *initial path* of the form $P \to \bullet\alpha\beta$. Figure 3.9 contains the exemplary rules for shifting the dot. A rule $A \overset{a}{\Rightarrow} B$ shows how the dotted path A is changed to the dotted path B when we descend to a child node with label a. We say that the rule *applies* for node a. Rules of the form $A \overset{\texttt{\#PCDATA}}{\Rightarrow} B$ apply to text nodes. Rules r_1 through r_8 concern element nodes, while rules r_9 through r_{14} are dedicated to text nodes. Rule r_{15} applies to both kinds of nodes.

In matching the descendant-axis, we may need to consider two possibilities. Suppose the current dotted path is $P \to \alpha \bullet \texttt{/descendant}::a\ \beta$ and we see an a-labeled node in the input. Then this can be the desired match, and we shift the dot to $P \to \alpha\texttt{/descendant}::a \bullet \beta$, or the match is still to come and the dot remains at its position. This is expressed by rule r_4.

Recognizing rules. We refer to the rules $r_1 - r_4$, r_6, r_8, r_9, r_{10}, $r_{13} - r_{15}$ as *recognizing rules*. Intuitively, these are the rules where we recognize nodes in the input that the dotted path expects to match next. In Figure 3.9, we have highlighted these rules by bold rule identifiers.

$$
\begin{array}{llcl}
\mathbf{r_1}: & P \to \alpha \bullet \texttt{/child} :: a\ \beta & \overset{a}{\Rightarrow} & P \to \alpha\texttt{/child} :: a \bullet \beta \\
\mathbf{r_2}: & P \to \alpha \bullet \texttt{/child} :: *\ \beta & \overset{a}{\Rightarrow} & P \to \alpha\texttt{/child} :: * \bullet \beta \\
\mathbf{r_3}: & P \to \alpha \bullet \texttt{/child} :: \texttt{node()}\ \beta & \overset{a}{\Rightarrow} & P \to \alpha\texttt{/child} :: \texttt{node()} \bullet \beta \\
\mathbf{r_4}: & P \to \alpha \bullet \texttt{/descendant} :: a\ \beta & \overset{a}{\Rightarrow} & P \to \alpha\texttt{/descendant} :: a \bullet \beta, \\
 & & & P \to \alpha \bullet \texttt{/descendant} :: a\ \beta \\
r_5: & P \to \alpha \bullet \texttt{/descendant} :: b\ \beta & \overset{a}{\Rightarrow} & P \to \alpha \bullet \texttt{/descendant} :: b\ \beta \\
\mathbf{r_6}: & P \to \alpha \bullet \texttt{/descendant} :: *\ \beta & \overset{a}{\Rightarrow} & P \to \alpha\texttt{/descendant} :: * \bullet \beta, \\
 & & & P \to \alpha \bullet \texttt{/descendant} :: *\ \beta \\
r_7: & P \to \alpha \bullet \texttt{/descendant} :: \texttt{text()}\ \beta & \overset{a}{\Rightarrow} & P \to \alpha \bullet \texttt{/descendant} :: \texttt{text()}\ \beta \\
\mathbf{r_8}: & P \to \alpha \bullet \texttt{/descendant} :: \texttt{node()}\ \beta & \overset{a}{\Rightarrow} & P \to \alpha\texttt{/descendant} :: \texttt{node()} \bullet \beta, \\
 & & & P \to \alpha \bullet \texttt{/descendant} :: \texttt{node()}\ \beta \\
\mathbf{r_9}: & P \to \alpha \bullet \texttt{/child} :: \texttt{text()}\ \beta & \overset{\texttt{\#PCDATA}}{\Rightarrow} & P \to \alpha\texttt{/child} :: \texttt{text()} \bullet \beta \\
\mathbf{r_{10}}: & P \to \alpha \bullet \texttt{/child} :: \texttt{node()}\ \beta & \overset{\texttt{\#PCDATA}}{\Rightarrow} & P \to \alpha\texttt{/child} :: \texttt{node()} \bullet \beta \\
r_{11}: & P \to \alpha \bullet \texttt{/descendant} :: a\ \beta & \overset{\texttt{\#PCDATA}}{\Rightarrow} & P \to \alpha \bullet \texttt{/descendant} :: a\ \beta, \\
r_{12}: & P \to \alpha \bullet \texttt{/descendant} :: *\ \beta & \overset{\texttt{\#PCDATA}}{\Rightarrow} & P \to \alpha \bullet \texttt{/descendant} :: *\ \beta \\
\mathbf{r_{13}}: & P \to \alpha \bullet \texttt{/descendant} :: \texttt{text()}\ \beta & \overset{\texttt{\#PCDATA}}{\Rightarrow} & P \to \alpha\texttt{/descendant} :: \texttt{text()} \bullet \beta, \\
 & & & P \to \alpha \bullet \texttt{/descendant} :: \texttt{text()}\ \beta \\
\mathbf{r_{14}}: & P \to \alpha \bullet \texttt{/descendant} :: \texttt{node()}\ \beta & \overset{\texttt{\#PCDATA}}{\Rightarrow} & P \to \alpha\texttt{/descendant} :: \texttt{node()} \bullet \beta, \\
 & & & P \to \alpha \bullet \texttt{/descendant} :: \texttt{node()}\ \beta \\
\mathbf{r_{15}}: & P \to \alpha \bullet \texttt{\#} & \overset{a,\texttt{\#PCDATA}}{\Rightarrow} & P \to \alpha \bullet \texttt{\#}
\end{array}
$$

***Figure 3.9**: Earley-style path matching rules for distinct tagnames a and b.*

Path sets. Starting with a set of initial paths at the root of the input, and applying the rules for matching moves, we obtain sets of dotted paths which we refer to as *path sets*. We illustrate this with a simple example where we abbreviate XPath axes child and descendant with "/" and "//".

Example 3.13 Consider the projection path `/a//b` and a path of nodes labeled a, b, c, and b from the root of an XML document to a b-labeled descendant. We then observe the following succession of path sets.

$$
\{P \to \bullet\texttt{/a//b}\} \overset{a}{\rightsquigarrow} \{P \to \texttt{/a} \bullet \texttt{//b}\} \overset{b}{\rightsquigarrow} \{P \to \texttt{/a//b}\bullet, P \to \texttt{/a} \bullet \texttt{//b}\}
$$
$$
\overset{c}{\rightsquigarrow} \{P \to \texttt{/a} \bullet \texttt{//b}\} \overset{b}{\rightsquigarrow} \{P \to \texttt{/a//b}\bullet, P \to \texttt{/a} \bullet \texttt{//b}\}
$$

The initial path set is $\{P \to \bullet\texttt{/a//b}\}$. For reading an a-labeled node (denoted $\overset{a}{\rightsquigarrow}$), we shift the dot by one position and obtain the path set $\{P \to \texttt{/a} \bullet \texttt{//b}\}$. We read a b-labeled node, and the path set develops into $\{P \to \texttt{/a//b}\bullet,\ P \to \texttt{/a} \bullet \texttt{//b}\}$. For reading a c-labeled node, we obtain the path set $\{P \to \texttt{/a} \bullet \texttt{//b}\}$. There is no rule for dotted path $P \to \texttt{/a//b}\bullet$ and input symbol c, so this path cannot be developed any further. In matching the second b-labeled node, we expand the path set to $\{P \to \texttt{/a//b}\bullet,\ P \to \texttt{/a} \bullet \texttt{//b}\}$. $\square$

Path-matching automata. Given a finite set of projection paths, we statically compute a deterministic finite-state automaton (DFA), the *path-matching automaton*. In this DFA, the states are path sets. The initial state is the path

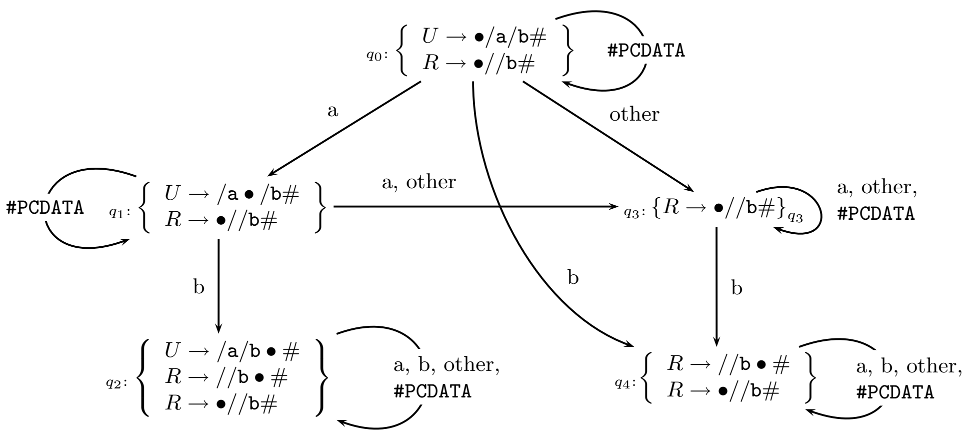

q_0: $\{U \to \bullet$/a/b#, $R \to \bullet$//b#$\}$
#PCDATA
a
other
#PCDATA
q_1: $\{U \to$ /a $\bullet$ /b#, $R \to \bullet$//b#$\}$
a, other
q_3: $\{R \to \bullet$//b#$\}_{q_3}$
a, other,
#PCDATA
b
b
b
q_2: $\{U \to$ /a/b $\bullet$ #, $R \to$ //b $\bullet$ #, $R \to \bullet$//b#$\}$
a, b, other,
#PCDATA
q_4: $\{R \to$ //b $\bullet$ #, $R \to \bullet$//b#$\}$
a, b, other,
#PCDATA

***Figure 3.10**: Path-matching automaton.*

set where all projection paths occur as initial paths. We successively derive all possible states and transitions in a closure algorithm [55], where we define transitions for all tagnames that explicitly occur in the projection paths. With the special label "other" we capture all other tagnames, and we use the label "#PCDATA" for textual data. The empty path set is a sink state.

Example 3.14 Figure 3.10 shows the path-matching automaton for the projection paths $U =$ /a/b# and $R =$ //b#. The initial state is q_0. □

Size of projection automata. For a finite set of projection paths, there are exponentially many path sets (and thus DFA states) in principle. Yet we may expect the automata to be small in practice. First of all, the number of projection paths extracted from XQueries is usually small. This certainly holds for the queries from the XMark or XBench benchmarks [120,122], or the XQuery Use Cases [123]. Moreover, paths that are defined w.r.t. the same variable in the query also share a common prefix. Consequently, certain dotted paths coexist in path sets, while paths sets such as $\{$/a $\bullet$ //b, /c $\bullet$ //b$\}$ are not possible, as a parent node is either labeled a or c. Likewise, a set $\{$/a/text()$\bullet$, /a/b$\bullet\}$ can be outruled. If we indeed should observe a large number of possible states, we can always construct the states lazily at runtime, which keeps the amount of states manageable for realistic XML documents [54].

3.5.3 Projection Semantics

We define XML document projection as a mapping between XML documents. Unless the projection preserves all data relevant for query evaluation, query evaluation on the original and the projected document may produce different results. This invalid behavior must be averted. We shortly formulate this requirement as *projection-safety.*

We can view projection paths as XPath expressions if we interpret the flag # as the XPath step expression /descendant-or-self::node(). Now given an

XML document T, we assume the usual XPath semantics [113] to evaluate a projection path p on T, by which we compute a list of XML documents and strings. We denote this evaluation according to the XPath semantics by $[\![p]\!](T)$, and the resulting lists by $[i_1, \ldots, i_n]$, where the i_j are the entries. To compare the evaluation of XPath expressions over XML documents and their projections, we define an equality relation over such lists. This relation captures the idea that from the viewpoint of XPath evaluation, the original and the projected document cannot be distinguished. We additionally factor in the possibility that in query evaluation, a node may be required without its descendants, e.g. as we only perform an existence check. This leads us to the following definition of *top-level equality* in comparing the results of XPath evaluation.

Definition 3.10 Let L_1 and L_2 be two lists of XML documents and strings. We say L_1 and L_2 are *top-level equal* iff

- L_1 and L_2 have the same length, and
- the ith elements of L_1 and L_2 are either two equal strings or two XML documents trees where the root nodes have the same label. □

Example 3.15 Let s be a fixed string. Then the lists $[\langle a\rangle b\langle/a\rangle, s]$, $[\langle a\rangle c\langle/a\rangle, s]$, and $[\langle a\rangle\langle/a\rangle, s]$ are pairwise top-level equal. □

Definition 3.11 Let $\mathcal{P}$ be a set of projection paths, and let T_1 and T_2 be XML documents. Then T_1 and T_2 are *top-level indistinguishable w.r.t.* $\mathcal{P}$ if for all projection paths p in $\mathcal{P}$, $[\![p]\!](T_1)$ and $[\![p]\!](T_2)$ are top-level equal. □

Definition 3.12 Let $\mathcal{P}$ be a set of projection paths. Let f be a mapping between XML documents. Then function f is *projection-safe w.r.t.* $\mathcal{P}$ if for all XML documents T, T and $f(T)$ are top-level indistinguishable w.r.t. $\mathcal{P}$. □

We now define the notion of relevant nodes based on projection paths. These are the nodes that are preserved in XML document projection.

Definition 3.13 Let $\mathcal{P}$ be a set of projection paths. Let $A(\mathcal{P})$ be the path-matching automaton computed from $\mathcal{P}$, and let c be a node in an XML document. Then the path of ancestor nodes from the root of the document to the parent of c defines a path from the initial state of $A(\mathcal{P})$ to some state S.

1. If c is an element node, then c *is relevant according to* $\mathcal{P}$ if state S meets one of the following conditions:
   (a) S contains a dotted path $P \rightarrow \alpha \bullet \beta$, where a recognizing rule applies for the label of node c,
   (b) S contains two dotted paths of the form $P \rightarrow \alpha \bullet /\texttt{child}::b\,\beta$ and $P' \rightarrow \alpha' \bullet /\texttt{descendant}::b\,\beta'$,
   (c) S contains a dotted path $P \rightarrow \alpha \bullet /\texttt{child}::\texttt{text}()\,\beta$ or a dotted path $P \rightarrow \alpha \bullet /\texttt{descendant}::\texttt{text}()\,\beta$.
2. If c is a text node, then c *is relevant according to* $\mathcal{P}$ if state S contains a dotted path $P \rightarrow \alpha \bullet \beta$ where a recognizing rule applies for `#PCDATA`. □

We briefly discuss this definition. Intuitively, condition (1a) tags all element nodes as relevant for which there is a progress in path matching. Conditions (1b) and (1c) cover special cases where ancestor-descendant relationships must be preserved, and which we illustrate with Examples 3.16 and 3.17 below. In turn, condition (2) concerning text nodes is straightforward.

In defining a projection function below, we preserve all nodes from the input that are relevant according to the definition above. By default, we also preserve the top-level element node to ensure that the output of projection is well-formed.

Definition 3.14 Let $\mathcal{P}$ be a set of projection paths and let T be an XML document. Then the *projection* $f_{\mathcal{P}}(T)$ is the XML document which contains only the nodes from T that are relevant according to $\mathcal{P} \cup \{/*\}$, with the ancestor-descendant and following-relationships as in T. □

Example 3.16 We are given the set of projection paths $P = \{$`/a/b#`, `//b#`$\}$. We then consider the XML document $\langle a\rangle\langle c\rangle\langle b/\rangle\langle/c\rangle\langle/a\rangle$. Then projection $f_{\mathcal{P}}$ again yields the input document, as all nodes are relevant according to $\mathcal{P} \cup \{/*\}$.

If we discard the c-labeled node, we obtain the document $\langle a\rangle\langle b/\rangle\langle/a\rangle$. Then projection safety is violated for the following reason. Let us consider the projection path `/a/b#`. In evaluating this path on the input $\langle a\rangle\langle c\rangle\langle b/\rangle\langle/c\rangle\langle/a\rangle$ according to the XPath semantics, we obtain the empty list. Yet for evaluation on the XML document $\langle a\rangle\langle b/\rangle\langle/a\rangle$, we obtain a non-empty list. These lists are not top-level equal, so the input and the projection are not top-level indistinguishable w.r.t. $\mathcal{P}$, as required by Definition 3.12. □

Example 3.17 In projecting nodes with mixed content, special care must be taken. The XQuery below checks whether the input document contains molecules with the textual content "H" for hydrogen.

```
<results>
{ for $x in //molecule return
  if ( some $y in $x/text() satisfies ($y = "H") )
  then "hydrogen" else () }
</results>
```

We extract the projection paths `//molecule` and `//molecule/text()` for the query-variables $\$x$ and $\$y$ respectively. Given the input

$$T_1 = \langle \mathit{molecule}\rangle H\langle \mathit{sub}\rangle 2\langle/\mathit{sub}\rangle O\langle/\mathit{molecule}\rangle,$$

then $f_{\mathcal{P}}(T_1)$ computes the projection "$\langle \mathit{molecule}\rangle H\langle \mathit{sub}\rangle\langle/\mathit{sub}\rangle O\langle/\mathit{molecule}\rangle$" where the *sub*-labeled node is not discarded. If we discard this node, we obtain the document $T_2 = \langle \mathit{molecule}\rangle HO\langle/\mathit{molecule}\rangle$. Now, projection safety is violated. Evaluating path `//molecule/text()` according to the XPath semantics on the original input yields ["H", "O"], i.e. a list of two strings. Yet the evaluation over document T_2 yields a list with the single string "HO". These lists are not top-level equal, so projection-safety is violated.

Correspondingly, the evaluation of the XQuery on T_1 produces a different result from the evaluation on T_2. □

The lemma below states the correctness of this approach.

Lemma 3.1 *Function $f_{\mathcal{P}}$ is projection-safe w.r.t. $\mathcal{P}$.*

In the following, we show how XML document projection can be implemented in accordance with the lemma above, such that only a single pass over the input document is required.

3.5.4 A Single-Pass Implementation

We implement XML projection by compiling our projection semantics into an *XML-DPDT*. This guarantees an execution that is scalable in the strictest sense. The projection automata defined in the following are an intermediary datastructure in our compilation to *XML-DPDT*s.

Projection automata. Given a path-matching automaton and the definition of node-relevance from above, it is straightforward to assign automaton transitions with labels, where label "R" denotes that a node is relevant to query evaluation, while label "I" classifies a node as irrelevant.

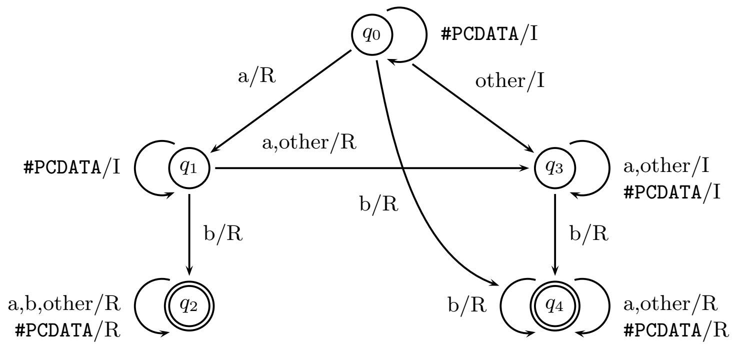


***Figure 3.11**: Projection automaton.*

Example 3.18 The projection automaton for the path-matching automaton from Figure 3.10 is shown in Figure 3.11. All transitions for reading text nodes are summarized by the label `#PCDATA`. The states q_0 through q_4 denote the same states as in Figure 3.10. □

Compilation to *XML-DPDT*s. By extending projection-automata with a stack, we obtain the necessary expressive power to process XML documents. In particular, we resort to the *XML-DPDT* formalism introduced earlier.

The following compilation of an *XML-DPDT* $\mathcal{T}_\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$ from a projection automaton assumes that the first token read in an XML document is an opening tag. We consider the transitions of the projection automaton one-by-one, and define corresponding *XML-DPDT* transitions. For simplicity, we assume that the set of possible tagnames is known in advance, so we can rewrite all transitions under label "other" by corresponding tagnames. In a practical implementation, we would of course retain transitions under the label "other", and match tagnames read in the input correspondingly.

Further, we assume that textual passages are read one character at-a-time.

- For each transition $q_0 \overset{a/R}{\to} q$ from the initial state of the projection automaton with some tagname a and state q, we define an *XML-DPDT*-transition which copies the input token to the output, namely

  $$\delta(q_0, \langle a \rangle, Z_0) = (q, (a, q_0), \langle a \rangle).$$

  For each transition of the form $q_0 \overset{a/I}{\to} q$, no output is produced.

  $$\delta(q_0, \langle a \rangle, Z_0) = (q, (a, q_0), \epsilon).$$

- Next, we consider all transitions $q \overset{a/R}{\to} p$ for some states q, p and tagname a. For all stack symbols $X \neq Z_0$, we define transitions

  $$\delta(q, \langle a \rangle, X) = (p, (a, q)X, \langle a \rangle) \quad \text{and} \quad \delta(p, \langle /a \rangle, (a, q)X) = (q, X, \langle /a \rangle).$$

  We proceed analogously for transitions $q \overset{a/I}{\to} q$, but produce no output.

- Finally, we consider character data. Note that all `#PCDATA`-transitions in projection automata are self-loops. For each transition $q \overset{\texttt{#PCDATA}/R}{\to} p$ and a character symbol c, we define $\delta(q, c, X) = (q, X, c)$ where this character is output and the stack remains unchanged. For transitions of the form $q \overset{\texttt{#PCDATA}/I}{\to} p$ we define analogous transitions, with the difference that no output is produced.

The pushdown-automaton such constructed is deterministic and coherent with the definition of *XML-DPDT*s.

Correctness properties. Let $\mathcal{P}$ be a set of projection paths. By $\mathcal{T}_{\mathcal{P}}$ we denote the *XML-DPDT* constructed from the projection automaton for the projection paths $\mathcal{P} \cup \{\texttt{/*}\}$. Then $\mathcal{T}_{\mathcal{P}}$ implements function $f_{\mathcal{P}}$ from Definition 3.14: Due to the projection path "`/*`", the opening- and closing tags of the topmost element node are preserved. If an element node is relevant, then both its opening- and closing tags are preserved, while both tags are ignored otherwise. Also, the ancestor-descendant relationships between relevant nodes are preserved, so that the output tags are properly nested.

Hence, for all well-formed XML documents, $\mathcal{P}$ preserves all nodes relevant according to $\mathcal{P} \cup \{/*\}$, along with ancestor-descendant and following-relationships between these nodes.

3.6 Cardinality of Path Matches

In path matching, we may not only be interested in *which* paths are matched by a node by the XPath semantics [113], but also in how many different ways a path is matched. For instance, consider the XPath expression `//a//b` and the input $\langle a \rangle \langle a \rangle \langle b/ \rangle \langle /a \rangle \langle /a \rangle$. Then the b-labeled node matches the path in two ways, once for each a-labeled ancestor. By extending the path sets from the previous section to multisets, we keep track of the *cardinality* of matches.

Example 3.19 We match the path `//a//b` using Earley-style dotted paths with a bag semantics. The initial path set is $S_0 = \{\bullet\texttt{//a//b}\}$. Processing an a-labeled node, we obtain the multiset

$$A = \{\bullet\texttt{//a//b}, \texttt{//a}\bullet\texttt{//b}\}.$$

Given a further a-labeled node, multiset A develops into the multiset

$$B = \{\bullet\texttt{//a//b}, \texttt{//a}\bullet\texttt{//b}, \texttt{//a}\bullet\texttt{//b}\},$$

with two occurrences of the dotted path instance $\texttt{//a}\bullet\texttt{//b}$. For reading a b-labeled node next, we obtain the multiset

$$C = \{\bullet\texttt{//a//b}, \texttt{//a}\bullet\texttt{//b}, \texttt{//a}\bullet\texttt{//b}, \texttt{//a//b}\bullet, \texttt{//a//b}\bullet\},$$

where the path `//a//b` is matched by this node in two ways. □

Counting paths. We abbreviate path multisets using numeric counters where dots can be placed in paths. A projection path $\texttt{/s}_1\texttt{/.../s}_n$ is then encoded as a *counting path* ${}_{[i_0]}\texttt{/s}_1{}_{[i_1]}\texttt{/.../s}_n{}_{[i_n]}$ where $i_0, i_1, \ldots, i_n$ are the numeric counters. Projection paths with the flag `#` at the tail are encoded accordingly, so $\texttt{/s}_1\texttt{/.../s}_n\texttt{\#}$ is encoded as ${}_{[i_0]}\texttt{/s}_1{}_{[i_1]}\texttt{/.../s}_n{}_{[i_n]}\texttt{\#}$. We refer to counter i_n as the *tail counter*. For instance, the counting path representation of multiset A from the previous example is $\{{}_{[1]}\texttt{//a}_{[0]}\texttt{//b}_{[0]}\}$, while C is encoded as $\{{}_{[1]}\texttt{//a}_{[2]}\texttt{//b}_{[2]}\}$.

Given a set of dotted paths with set-semantics, we mentioned in the previous section that we can statically compute a DFA in a closure algorithm [55]. With counting paths, the closure would not reach a fixpoint if paths contain the descendant axis. Instead, we compute the automata lazily at runtime. We begin with the initial path set. For each initial dotted path $P \to \bullet\texttt{/s}_1\texttt{/.../s}_n$, we define an initial counting path $P \to {}_{[1]}\texttt{/s}_1{}_{[0]}\texttt{/.../s}_n{}_{[0]}$. In matching counting paths, the tail counter is treated differently. This distinction is evident in the counter update rules in Figure 3.12. These read as follows. Let S be the current set of counting paths. Then reading a node labeled a, we update each counting path P in S by resetting the counters from right to left by rewriting $[\![P]\!]^a_{tail}$. The update rules first reset the tail counter, and then proceed to the remaining counters in the "body" of the counting path.

Example 3.20 We recompute the path sets from Example 3.19 using the update rules for counting paths. We instantiate the initial set

$$\{\ [\![\ [\![\ [\![\ {}_{[1]}\texttt{//a}_{[0]}\texttt{//b}_{[0]}\ ]\!]^a_{tail}\ ]\!]^a_{tail}\ ]\!]^b_{tail}\ \}$$

for the sequence of nodes a, a, and b. We first update the innermost counting path from right to left for reading an a-labeled node, and obtain the paths from multi-set A from the previous example.

$$[\![{}_{[1]}\texttt{//a}_{[0]}\texttt{//b}_{[0]}]\!]^a_{tail} = [\![{}_{[1]}\texttt{//a}_{[0]}]\!]^a_{body}\texttt{//b}_{[0]} = [\![{}_{[1]}]\!]^a_{body}\texttt{//a}_{[1]}\texttt{//b}_{[0]} = {}_{[1]}\texttt{//a}_{[1]}\texttt{//b}_{[0]}$$

Next, we update the counting path for the second a-labeled node which yields the paths from multiset B.

$$[\![{}_{[1]}\texttt{//a}_{[1]}\texttt{//b}_{[0]}]\!]^a_{tail} = [\![{}_{[1]}\texttt{//a}_{[1]}]\!]^a_{body}\texttt{//b}_{[0]} = [\![{}_{[1]}]\!]^a_{body}\texttt{//a}_{[2]}\texttt{//b}_{[0]} = {}_{[1]}\texttt{//a}_{[2]}\texttt{//b}_{[0]}$$

For distinct tagnames a and b, and counter variables x and y:

$$\begin{array}{rcl}
[\![\alpha_{[x]}\texttt{\#}]\!]^{a}_{tail} & = & [\![\alpha_{[x]}]\!]^{a}_{body}\texttt{\#} \\
[\![\alpha_{[x]}\texttt{/child::}\nu_{[y]}]\!]^{a}_{tail} & = & \begin{cases} [\![\alpha_{[0]}]\!]^{a}_{body}\texttt{/child::}\nu_{[x]} & \nu \in \{a, \texttt{*}, \texttt{node()}\} \\ [\![\alpha_{[0]}]\!]^{a}_{body}\texttt{/child::}\nu_{[0]} & \text{otherwise} \end{cases} \\
[\![\alpha_{[x]}\texttt{/descendant::}\nu_{[y]}]\!]^{a}_{tail} & = & \begin{cases} [\![\alpha_{[x]}]\!]^{a}_{body}\texttt{/descendant::}\nu_{[x]} & \nu \in \{a, \texttt{*}, \texttt{node()}\} \\ [\![\alpha_{[x]}]\!]^{a}_{body}\texttt{/descendant::}\nu_{[0]} & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
[\![\epsilon_{[x]}]\!]^{a}_{body} & = & {}_{[x]} \\
[\![\alpha_{[x]}\texttt{/child::}\nu_{[y]}]\!]^{a}_{body} & = & \begin{cases} [\![\alpha_{[0]}]\!]^{a}_{body}\texttt{/child::}\nu_{[x+y]} & \nu \in \{a, \texttt{*}, \texttt{node()}\} \\ [\![\alpha_{[0]}]\!]^{a}_{body}\texttt{/child::}\nu_{[y]} & \text{otherwise} \end{cases} \\
[\![\alpha_{[x]}\texttt{/descendant::}\nu_{[y]}]\!]^{a}_{body} & = & \begin{cases} [\![\alpha_{[x]}]\!]^{a}_{body}\texttt{/descendant::}\nu_{[x+y]} & \nu \in \{a, \texttt{*}, \texttt{node()}\} \\ [\![\alpha_{[x]}]\!]^{a}_{body}\texttt{/descendant::}\nu_{[y]} & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
[\![\alpha_{[x]}\texttt{\#}]\!]^{\texttt{\#PCDATA}}_{tail} & = & [\![\alpha_{[x]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{\#} \\
[\![\alpha_{[x]}\texttt{/child::}\nu_{[y]}]\!]^{\texttt{\#PCDATA}}_{tail} & = & \begin{cases} [\![\alpha_{[0]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/child::}\nu_{[x]} & \nu \in \{\texttt{text()}, \texttt{node()}\} \\ [\![\alpha_{[0]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/child::}\nu_{[0]} & \text{otherwise} \end{cases} \\
[\![\alpha_{[x]}\texttt{/descendant::}\nu_{[y]}]\!]^{\texttt{\#PCDATA}}_{tail} & = & \begin{cases} [\![\alpha_{[x]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/descendant::}\nu_{[x]} & \nu \in \{\texttt{text()}, \texttt{node()}\} \\ [\![\alpha_{[x]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/descendant::}\nu_{[0]} & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
[\![\epsilon_{[x]}]\!]^{\texttt{\#PCDATA}}_{body} & = & {}_{[x]} \\
[\![\alpha_{[x]}\texttt{/child::}\nu_{[y]}]\!]^{\texttt{\#PCDATA}}_{body} & = & \begin{cases} [\![\alpha_{[0]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/child::}\nu_{[x+y]} & \nu \in \{\texttt{text()}, \texttt{node()}\} \\ [\![\alpha_{[0]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/child::}\nu_{[y]} & \text{otherwise} \end{cases} \\
[\![\alpha_{[x]}\texttt{/descendant::}\nu_{[y]}]\!]^{\texttt{\#PCDATA}}_{body} & = & \begin{cases} [\![\alpha_{[x]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/descendant::}\nu_{[x+y]} & \nu \in \{\texttt{text()}, \texttt{node()}\} \\ [\![\alpha_{[x]}]\!]^{\texttt{\#PCDATA}}_{body}\texttt{/descendant::}\nu_{[y]} & \text{otherwise} \end{cases}
\end{array}$$

***Figure 3.12**: Updating counting paths.*

Finally, we perform the updates for the b-labeled node.

$$[\![{}_{[1]}\texttt{//a}_{[2]}\texttt{//b}_{[0]}]\!]^{b}_{tail} = [\![{}_{[1]}\texttt{//a}_{[2]}]\!]^{b}_{body}\texttt{//b}_{[2]} = [\![{}_{[1]}]\!]^{b}_{body}\texttt{//a}_{[2]}\texttt{//b}_{[2]} = {}_{[1]}\texttt{//a}_{[2]}\texttt{//b}_{[2]}$$

The result is the encoding of multiset C using counting paths. □

Let $c_1, c_2, \ldots, c_n$ be a sequence of tagnames, then we introduce $[\![P]\!]^{c_1 c_2 \ldots c_n}_{tail}$ as syntactic sugar for $[\![\ldots[\![[\![P]\!]^{c_1}_{tail}]\!]^{c_2}_{tail}\cdots]\!]^{c_n}_{tail}$.

Definition 3.15 Let P be a projection path and let P_0 its initial counting path. Let $c_1, \ldots, c_n$ be a sequence of labels that describe a path of descendants, starting from the root of an XML document to a node labeled c_n. We say P is matched by c_n exactly k-times if the tail counter of $[\![P_0]\!]^{c_1 \ldots c_n}_{tail}$ has value k. □

Part II

Abstract Framework

4 Framework Definition

In application development, decisions made early on in the design process, such as the choice of datastructures or methods for manipulating them, can have far-reaching implications for the subsequent states of program development, and even the runtime-behavior. Out of this motivation, we propose an abstract framework for modeling main memory-based XML stream processors. Our intention is that developers may model different system setups, for the purpose of learning about their characteristics early on in the design stage.

In Section 4.1, we explain the motivation in designing this framework, and give an overview over the framework components. The formal definitions are provided in Section 4.2. While ease-of-use and transparency cannot be justified by our claims alone, we do believe that our framework lends itself nicely to model and discuss various design decisions in building applications for XML stream processing. To affirm this claim, we model several example applications in Section 4.3 and in the following chapters.

4.1 Outline

The formal part of our framework is motivated by the following considerations. Any formalism for XML processing must be capable of representing various tree datastructures. For one, XML documents are generally viewed as trees, yet queries can also be parsed into query parse trees. These are commonly the basis for static query analysis. Further, the operators in query algebras are typically assembled into tree-shaped query plans. Hence, we require a *tree manipulation language*. While other systems for XML processing are well capable of expressing tree manipulations (e.g. [50, 61]), in XML stream processing, we need to also capture low-level tasks such as reading single XML events from the input stream and also writing them to the output.

In this thesis, we develop a novel framework that is capable of managing this tightrope walk between low-level event handling and high-level tree manipulations. Our formalism is designed in the tradition of term rewriting systems (e.g. see [33]). Rewriting systems consist of directed equations, and computations are performed by repeatedly replacing subtrees in terms. Our framework provides means for modeling the basic infrastructure of XML stream processing systems, as sketched in Figure 4.1.

The framework consists of a *read-only input tape* (or input stream) of XML

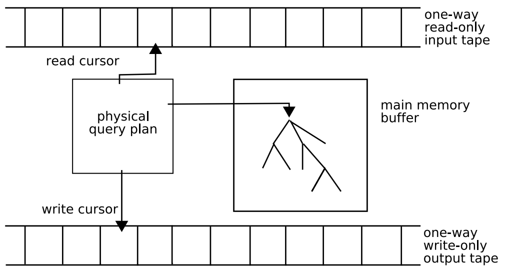
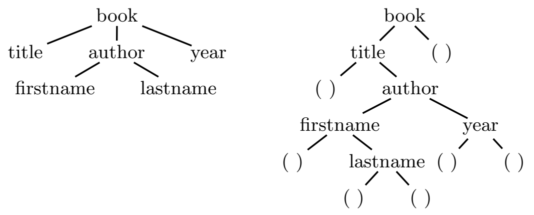


Figure 4.1: *Infrastructure for XML stream processing.*



```
book[
  title[()]
  author[
    firstname[()]
    lastname[()] ()]
  year[()] ()] ()
```

(a) Unranked tree. (b) Binary tree encoding. (c) Inlined term notation.

Figure 4.2: *Equivalent tree representations.*

tokens, and a *write-only output tape* (or output stream) of XML tokens. On both tapes, a cursor which can only move forward marks the current read or write position. A *main memory buffer* stores data for future reference. Finally, a *physical query plan*, the contents of the input tape, and the buffer contents specify a sequence of operations that compute the query result.

In our framework, the syntactic objects for events from the input- or output-tape, for buffer contents, and the query plan are *terms*. The processing of the input, the management of buffer contents, and the writing of tokens to the output are specified by *term rewriting rules*, the means for manipulating terms. We proceed with an exemplary overview of our framework, and provide the formal definitions in the next section.

Binary tree encoding. XML documents encode node-labeled, ordered trees. For simplicity, we disregard text values, attributes, and comments, features which can be modeled as parts of the tree. We assume that all node labels stem from the set of tagnames *Tag*. We model XML documents as *binary* trees, using an established first-child next-sibling encoding (e.g. [50, 66]). In Figures 4.2(a) and (b), we show two views of the same XML document, as an unranked and as a binary tree. Independent of this encoding, the title, author, and year nodes are siblings according to the XML document model.

The terms for the binary tree encoding are constructed from the simple grammar "$\tau ::= (\ ) \mid t[\tau]\tau$" where "( )" is the *empty node* (also called *empty term*) and t is a tagname. Then $t[e_1]e_2$ is an element node with label t and content e_1, which is followed by e_2. In specifying terms, we will frequently switch between the view of terms as trees (as in Figure 4.2(b)) and the inlined term notation (as in Figure 4.2(c)).

Term rewriting. A stream processing system is specified by an initial term, called the *start-term*, and a set of term rewriting rules for manipulating this term. We introduce rewriting rules informally at this point, and provide concise definitions in the next section. Term rewriting rules are expressions of the form

$$\textit{replacement} \quad \leftarrow \quad \textit{template} \quad \{\ \textit{action}\ \}.$$

At runtime, the system has a single *query-term* (obtained by rewriting the start-term). If the query-term is matched by the *template* of a rewriting rule, then the *action* is executed and the query-term is modified according to the *replacement*. We specify the details of matching later on.

Modeling I/O. We next specify a rule to read a token from the input tape. Here, we presume an XML stream processing infrastructure as sketched in Figure 4.1. We assume that the current query-term is matched by template τ, and that there is a term rewriting rule

$$x \quad \leftarrow \quad \tau \quad \{\ \text{newVar } x;\ x := \text{read}()\ \}. \tag{4.1}$$

Then the query-term is replaced by the next token from the input stream. This is done in several steps. As the query-term is matched by the template τ, the action is executed, and a new variable x is defined. We refer to these variables as *buffer-variables.* When this variable is assigned with the value of "read()", the next token is read from the input tape, and the input cursor advances by one input token. The left-hand side of the rewriting rule specifies that the query-term is now replaced by the value of variable x.

The following rules match the query-terms "$\langle t\rangle$" or "$\langle/t\rangle$" for a given tagname. In templates, we underline all terms except variables, to emphasize which parts must be matched by query-terms. In applying these rules, we write the matched XML tag to the output and reduce the query-term to "\$".

$$\$ \quad \leftarrow \quad \underline{\langle t\rangle} \quad \{\ \text{write}(\ \langle t\rangle\ )\ \} \tag{4.2}$$

$$\$ \quad \leftarrow \quad \underline{\langle/t\rangle} \quad \{\ \text{write}(\ \langle/t\rangle\ )\ \} \tag{4.3}$$

Example 4.1 We assume a system defined by the start-term "\$" and the term rewriting rule $r_{4.1}$ for $\tau = \$$. We further assume that rules $r_{4.2}$ and $r_{4.3}$ are instantiated for all tagnames t. This system simply copies the input to the output, one token at-a-time.

We visualize some steps in Figure 4.3, where we lean on the graphical representation from Figure 4.1 to point out the correspondences. In particular, the query-term corresponds to the physical query plan, and a memory buffer is not required in this example. Step (a) shows the start-term and the input cursor which is positioned on the first input token. In applying rule 4.1, the query-term is substituted by the first input token, and the cursor advances in the transition to step (b). In applying rule 4.2, the query-term is replaced by the term "\$", and the token "$\langle t\rangle$" is written to the output-tape in step (c). Now, rule 4.1 can again be applied to copy the next input token. □

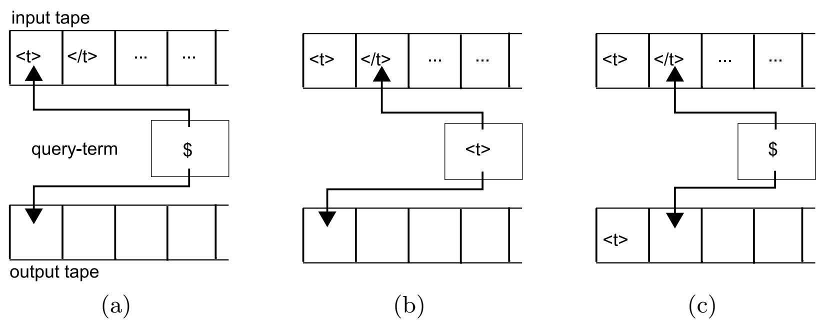


Figure 4.3: *Snapshots of Example 4.1.*

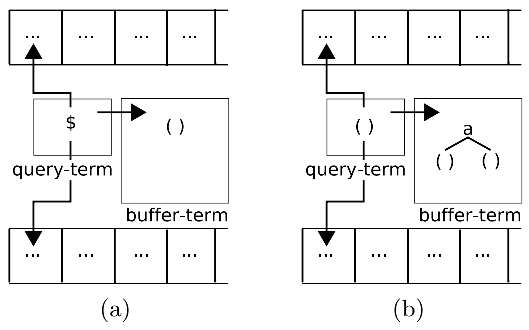


Figure 4.4: *Snapshots of Example 4.2.*

Modeling buffering. We next consider the issue of buffering. We have already shown that inside actions, variables can be assigned with input tokens. In fact, we can assign variables with arbitrary terms, such as binary-tree encodings of XML documents. Our framework provides a global buffer-variable **root**. By assigning terms to this variable, we can conveniently store terms for future use. We refer to the term stored by **root** as the *buffer-term*. When building XML stream processors, we commonly use the query-term and the buffer-term in analogy to the physical query plan and the main memory buffer from Figure 4.1. This is indicated by the layout of Figure 4.4, where the input tape is shown on top, and the output tape on the bottom.

Let us consider an example. We assume that the query-term is matched by template τ. To assign the buffer-term the tree "a[( )]( )", we specify a rule

$$\tau' \quad \leftarrow \quad \tau \quad \{\, \textbf{root} := \mathrm{a}[(\ )](\ )\,\} \tag{4.4}$$

in which the buffer-term is assigned within the action, and where τ' defines some replacement.

Example 4.2 In Figure 4.4, we show two snapshots in applying rule 4.4 for $\tau = \$$ and $\tau' = (\ )$. Initially, the buffer-term is the empty term, and the start-term is "\$". In applying the rewriting rule, the query-term is replaced by the empty term, and the buffer-term is replaced by the binary tree encoding of a single node labeled a. □

The buffer-term can be accessed and modified at later stages during term rewriting. For instance, to apply a function f to the buffer-term, we reassign the **root**-variable inside an action as sketched below.

$$\tau' \quad \leftarrow \quad \tau \quad \{\ \mathbf{root} := \mathrm{f}(\mathbf{root})\ \}$$

To check conditions on the buffer-term, the **root**-variable is made part of the query-term. Assume that function “even” rewrites a term to “true” if a binary tree has an even number of nodes, and to “false” otherwise. Then we instantiate the start-term “even(**root**)” to compute the result. Conceptually, the buffer-term is copied (in terms of call-by-value) into the query-term, so the buffer-term remains unchanged.

Remark 4.1 This call-by-value approach of copying the buffer-term into the query-term would be prohibitively expensive regarding the main memory consumption in practice. Already in the example above, instantiating “even(**root**)” doubles the main memory requirements of the query- and the buffer-term. However, we take care to only apply functions to **root** within query-terms that rewrite this term to single unary term (e.g. “true” or “false”), and which proceed in a single pass. This has the advantage that subtrees can be shared between the query-term and the buffer-term, as done in [50] and as discussed later in Example 4.10.

4.2 Syntax and Semantics

We next define the syntax and semantics for modeling stream processing systems in our framework.

Definition 4.1 The syntax of a *term* τ is defined in by the grammar

$$\tau \quad ::= \quad (\ ) \ \mid\ a[\tau]\tau \ \mid\ x \ \mid\ f^n(\tau_1, \ldots, \tau_n) \ \mid\ \langle a \rangle \ \mid\ \langle /a \rangle \ \mid\ \langle / \rangle.$$

The constructs “( )” and “$a[\tau]\tau$” define binary trees, where a is a tagname in *Tag*. Meta-variable x ranges over an infinite set of variables. The meta-variable f^n ranges over a set of symbols of arity n (with $n \geq 0$). Among XML events, we define opening tags, closing tags, and the implicit token “$\langle / \rangle$” to represent the end of the input stream.

A *term rewriting rule* is a statement of the form

$$\textit{replacement} \quad \leftarrow \quad \textit{template} \qquad \big[\ \{\ \textit{action}\ \}\ \big]$$

where *template* and *replacement* are terms. The *action* is an optional expression defined according to the grammar

$$\textit{action} \quad ::= \quad \mathrm{newVar}\ x \ \mid\ x{:=}\tau \ \mid\ x{:=}\ \mathrm{read}() \ \mid\ \mathrm{write}(\textit{event}) \ \mid\ \textit{action};\ \textit{action}$$

where meta-variable x ranges over variable names and τ is a term. □

Within a rule, we require that the variable names occurring in the template are disjoint from those declared in actions. Moreover, the replacement term can only contain variables that occur in either the template or the action of a rule. The same variable name cannot occur multiple times in a template, and the global variable **root** is not allowed to occur in templates. Finally, we make it a convention to underline all terms except variables in templates.

We are now in the position to define the syntax of XML stream processors in our framework.

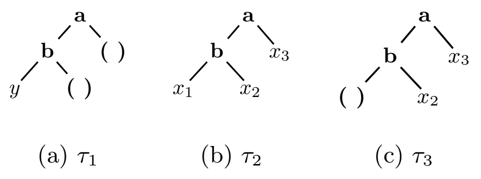


Figure 4.5*: Terms with buffer-variables y, x_1, x_2, and x_3.*

Definition 4.2 An XML stream processor $\mathcal{F} = (s, R)$ is a tuple of an initial term s, called the *start-term*, and a set R of term rewriting rules. □

The term rewriting process starts with the start-term as the query-term. When a template of a rewriting rule is matched by a subterm of the query- or the buffer-term, we proceed as follows. First, the action is executed, and then the query-term is rewritten according to the replacement. The rewriting process stops when no more rules can be applied. We formalize this mechanism below.

A *substitution* σ is a mapping from variables to terms or the special symbol "$\perp$", and written out as $\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}$. To access τ_i, we write $\sigma(x_i)$. At times, we interpret substitutions as variable bindings. Then a substitution $\sigma(x) = \perp$ denotes that the variable x is defined, but not yet bound to any term.

Substitutions play a role when applying term rewriting rules. If there exists a substitution σ embedding a template into a term, we say that the term is *matched* by the template with substitution σ. The matched part of the term is called a *reducible expression*, or *redex* for short. When the redex is replaced as defined by the replacement-term, then this again involves applying a substitution. Below, we define the process of updating terms, and then illustrate the idea using an example.

Definition 4.3 Let τ be a term and let σ be a substitution. In *updating term τ for substitution σ*, each variable x occurring in τ for which $\sigma(x)$ is a term is replaced by $\sigma(x)$. We denote the updated term by $\tau\big[\sigma\big]$. □

Example 4.3 We consider the terms from Figure 4.5 and assume a rewriting rule with template τ_2. Then term τ_1 is matched by the template τ_2 with the substitution $\sigma = \{x_1 \mapsto y, x_2 \mapsto (\ ), x_3 \mapsto (\ )\}$, so $\tau_2\big[\sigma\big] = \tau_1$. If τ_2 occurs as a replacement-term, then updating term τ_2 for substitution σ again yields the term τ_1. Moreover, updating τ_2 for $\sigma' = \{x_1 \mapsto (\ ), x_2 \mapsto \perp\}$ yields term τ_3. Note that variable x_2 is not replaced, as $\sigma'(x_2)$ does not map to a term, but to the designated symbol $\perp$. □

We introduce the notion of configurations to describe an XML stream processor at any given instant. In the initial configuration, the start-term is the query-term and only the global variable **root** is defined (but not initialized). At this point, no input has been consumed, and no output has been produced yet.

Definition 4.4 A *configuration* of an XML stream processor $\mathcal{F} = (s, R)$ is a quadruple (q, ω, σ, o) where q is the current query-term, ω is the sequence of XML tokens still to be read from the input tape, σ is a substitution, and o is the output generated so far. The *initial configuration* given an XML input document w is $(s, w\langle/\rangle, \{\textbf{root} \mapsto \perp\}, \epsilon)$. □

Semantics of actions. We consider the effect of executing actions on a configuration. We write $(q, \omega, \sigma, o) \vdash^c (q', \omega', \sigma', o')$ to denote that executing the single command c transfers the configuration (q, ω, σ, o) to $(q', \omega', \sigma', o')$. In particular, we define the following transitions where we assume that σ is defined as $\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}$ and that for all substitutions $\hat{\sigma}$, $\bot\big[\hat{\sigma}\big] = \bot$.

$$(q,\ \omega,\ \sigma,\ o) \overset{\text{newVar } x}{\vdash} (q,\ \omega,\ \sigma \cup \{x \mapsto \bot\},\ o) \tag{4.5}$$

$$\begin{aligned}(q,\ \omega,\ \sigma,\ o) \overset{x_i:=\tau}{\vdash}\ & (q,\ \omega,\ \sigma', o) \text{ where} \\ & \sigma' = \{x_i \mapsto \tau\big[\sigma\big]\} \cup \{x_j \mapsto \tau_j\big[\{x_i \mapsto \tau\big[\sigma\big]\}\big] \mid j \neq i\}\end{aligned} \tag{4.6}$$

$$\begin{aligned}(q,\ t\omega,\ \sigma,\ o) \overset{x_i:=\text{read}()}{\vdash}\ & (q,\ \omega,\ \sigma',\ o) \text{ where} \\ & t \in \{\langle a\rangle, \langle/a\rangle \mid a \in \mathit{Tag}\} \cup \{\langle/\rangle\} \text{ and} \\ & \sigma' = \{x_i \mapsto t\} \cup \{x_j \mapsto \tau_j\big[\{x_i \mapsto t\}\big] \mid j \neq i\}\end{aligned} \tag{4.7}$$

$$\begin{aligned}(q,\ \omega,\ \sigma,\ o) \overset{\text{write}(t)}{\vdash}\ & (q,\ \omega,\ \sigma,\ ot) \text{ where} \\ & t \in \{\langle a\rangle, \langle/a\rangle \mid a \in \mathit{Tag}\} \cup \{\langle/\rangle\}\end{aligned} \tag{4.8}$$

In transition 4.5, a new variable x is defined. If the variable name x should already be in use, we rename variables to resolve *name conflicts*. In transition 4.6, the value of a variable x_i is assigned. This affects not only this variable, but possibly also the terms assigned to the other variables, which need to be updated for the new value of x_i. Reading a token from the input and assigning it as the value of a variable is captured by transition 4.7. Finally, writing an XML event to the output is described by transition 4.8.

Example 4.4 We assume a configuration $(q, \omega, \{\mathbf{root} \mapsto (\ )\}, o)$ for some query-term q, input ω, and output o, and execute the command "**root**:=f(**root**)".

$$\begin{aligned} & (q,\ \omega,\ \{\mathbf{root} \mapsto (\ )\},\ o) \\ \overset{\mathbf{root}:=f(\mathbf{root})}{\vdash}\ & (q,\ \omega, \{\mathbf{root} \mapsto f(\mathbf{root})\big[\{\mathbf{root} \mapsto (\ )\}\big]\}) \\ =\ & (q,\ \omega, \{\mathbf{root} \mapsto f(\ (\ )\ )\})\end{aligned}$$

Thus, the buffer-term is changed from "( )" to "f(( ))". □

Example 4.5 We assume a configuration $(q, \omega, \{\mathbf{root} \mapsto a[x_1](\ ), x_1 \mapsto \bot\}, o)$, and execute the command "x_1:=( )".

$$\begin{aligned} & (q,\ \omega,\ \{\mathbf{root} \mapsto a[x_1](\ ), x_1 \mapsto \bot\},\ o) \\ \overset{x_1:=(\ )}{\vdash}\ & (q,\ \omega,\ \{\mathbf{root} \mapsto a[x_1](\ )\big[\{x_1 \mapsto (\ )\}\big], \\ & \qquad x_1 \mapsto (\ )\big[\{\mathbf{root} \mapsto a[x_1](\ ), x_1 \mapsto \bot\}\big]\},\ o) \\ =\ & (q,\ \omega,\ \{\mathbf{root} \mapsto a[(\ )](\ ), x_1 \mapsto (\ )\}, o)\end{aligned}$$

In assigning a term to x_1, the value of the **root**-variable changes as well. □

Let $\alpha = s_1; \ldots; s_n$ be a sequence of statements, then by $c_1 \vdash^\alpha c_{n+1}$ we summarize the execution of these statements on configuration c_1 by $c_1 \vdash^{s_1} c_2 \vdash^{s_2} \ldots c_n \vdash^{s_n} c_{n+1}$.

Semantics of term rewriting rules. We denote the application of a term rewriting rule r to a configuration c as $c \vdash_r d$ where d is the new system configuration. In the application of rules, we identify three cases. The first two cases concern rewriting rules that are free of side-effects, as they have no action. We apply them to the query-term or the buffer-term. The third case involves rules with actions, which we apply to query-terms only.

Let r be a term rewriting rule and let (q, ω, σ, o) be a configuration. Then we distinguish the following cases.

1. Rule r is of the form $\rho \leftarrow \tau$ and template τ is matched by a subtree t of query-term q with substitution σ_τ, so the inlined term notation of q is of the form utv for some strings u and v. Then

$$(q,\ \omega,\ \sigma,\ o)\ \ \vdash^r\ \ (u(\rho\big[\sigma \cup \sigma_\tau\big])v\ ,\ \omega,\ \sigma,\ o).$$

2. Rule r is of the form $\rho \leftarrow \tau$ and template τ is matched by a subtree t of buffer-term $\sigma(\mathbf{root})$ with substitution σ_τ, so the inlined term notation of $\sigma(\mathbf{root})$ is of the form utv. Then

$$(q,\ \omega,\ \sigma,\ o)\ \ \vdash^r\ \ (q,\ \omega,\ \{\mathbf{root} \mapsto u(\rho\big[\sigma \cup \sigma_\tau\big])v\},\ o).$$

3. Rule r is of the form $\rho \leftarrow \tau\ \{\alpha\}$ and template τ is matched by a subtree t of query-term q with substitution σ_τ, so the inlined term notation of q is of the form utv.

   Substitution σ_τ may map variables to variables. As variables can be reassigned in actions, we need to address the issue of *variable aliasing.*

   (a) We define a new substitution $\hat{\sigma}$ that contains all mappings of variables to terms from σ_τ. It further ignores all mappings of variables to the **root**-variable, and for all other mappings "$\sigma(x) = y$" between variables introduces a mapping "$\hat{\sigma}(y) = \bot$". This defines variable y in the environment, but leaves it unassigned. We formalize this below, where we assume that $\sigma_\tau = \{z_1 \mapsto \psi_1, \ldots, z_n \mapsto \psi_n\}$.

$$\begin{aligned}&\hat{\sigma} = \{z_i \mapsto \psi_i \mid \psi_i \text{ is not a variable}\}\\ &\cup\{\psi_i \mapsto \bot \mid \text{ex. } z_j \text{ s.t. } \sigma_\tau(z_j) = \psi_i,\ \psi_i \text{ is a variable},\ \psi_i \neq \mathbf{root}\}\end{aligned}$$

   (b) For each mapping $\sigma_\tau(z_i) = \psi_i$ such that ψ_i is a variable, we replace all occurrences of z_i in action α and replacement-term ρ by ψ_i. This yields action α' and replacement-term ρ'.

   Then we execute the action by $(q,\ \omega,\ \sigma \cup \hat{\sigma},\ o) \vdash^{\alpha'} (q,\ \omega',\ \sigma',\ o')$, and define the transition

$$(q,\ \omega,\ \sigma,\ o)\ \ \vdash^r\ \ (u(\rho'[\sigma'])v,\ \omega',\ \{\mathbf{root} \mapsto \sigma'(\mathbf{root})\},\ o').$$

In the first case, only the query-term is updated, but the input- and output tapes, as well as the buffer-term, remain unaffected. In the second case we only modify the buffer-term. The third case can affect all components of the system configuration. Note that we restrict all I/O-related side-effects to actions. Commands within actions are executed in the order of their specification, which guarantees a deterministic order for reading tokens from the input and for generating output.

Example 4.6 We formally define the XML stream processor from Example 4.1 that copies the input to the output as $\mathcal{F} = (\$, R)$ where R consists of the rewriting rules 4.1 through 4.3 instantiated for all tagnames. The initial configuration for the input "$\langle a\rangle\langle/a\rangle$" is $(\$,\ \langle a\rangle\langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon)$.

Then the template of rule $r_{4.1}$ is matched by the query-term for the empty substitution, as the template consists of a single constant term. We execute the action of rule $r_{4.1}$,

$$\begin{array}{cl} & (\$,\ \langle a\rangle\langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon) \\ \overset{\text{newVar } x}{\vdash} & (\$,\ \langle a\rangle\langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot,\ x \mapsto \bot\},\ \epsilon) \\ \overset{x:=\text{read}()}{\vdash} & (\$,\ \quad \langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot,\ x \mapsto \langle a\rangle\},\ \epsilon) \end{array}$$

and update the query-term to "$x\big[\{\mathbf{root} \mapsto \bot,\ x \mapsto \langle a\rangle\}\big] = \langle a\rangle$". In summary,

$$(\$,\ \langle a\rangle\langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon) \quad \vdash^{r_{4.1}} \quad (\langle a\rangle,\ \langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon).$$

Now the template of rule $r_{4.2}$ is matched by the query-term, again with an empty substitution. Below, we execute the action

$$\begin{array}{cl} & (\langle a\rangle,\ \langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon) \\ \overset{\text{write}(\langle a\rangle)}{\vdash} & (\langle a\rangle,\ \langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle) \end{array}$$

and update the query-term to "\$". We summarize this step below.

$$(\langle a\rangle,\ \langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon) \quad \vdash^{r_{4.2}} \quad (\$,\ \langle/a\rangle\langle/\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle)$$

We fast-forward to the configuration $(\langle/\rangle,\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle\langle/a\rangle)$. As no further rewriting rule applies, the rewriting process terminates. □

4.3 Modeling Basic Applications

We model several applications in our framework that cover basic functionalities such as I/O, buffering the input, and serializing binary trees into XML events. Parts of these encodings will be re-used in the succeeding chapters, where we model more complex XML stream processors.

In modeling, we make use of the following constructs for syntactic sugaring. These constructs can all be expressed within our framework.

Lists. We define lists of terms according to the grammar "*list* ::= [ ] | τ :: *list*". Here, "[ ]" denotes the empty list, the function symbol "::" is the operator for list concatenation, and τ is a term. We commonly abbreviate lists such as $a :: b :: c :: d$ by $[a, b, c, d]$, where $[x]$ is equivalent to $x :: [\,]$.

Tree concatenation. We concatenate binary tree representations of XML documents as shown below and in [50]. The rules are defined for all tagnames a.

$$\begin{array}{rcl} a[e_1]\,\text{concat}(e_2,\, e_3) & \leftarrow & \underline{\text{concat}}(\ \underline{a}[e_1]e_2,\, e_3\ ) \\ e & \leftarrow & \underline{\text{concat}}(\ \underline{(\ )},\, e) \\ e & \leftarrow & \underline{\text{concat}}(\ e,\, \underline{(\ )}) \end{array}$$

4.3.1 Checking Well-Formedness

We specify an application that checks whether the input is well-formed. While this is a very basic task, it lends itself nicely to illustrate the workings of our framework. To match opening- and closing tags, the application maintains a stack of tagnames. This stack is modeled as a list within the query-term, where the leftmost item in the list represents the top of the stack. The application will output "$\langle wf\rangle\langle /wf\rangle$" if the input is well-formed, and "$\langle mf\rangle\langle /mf\rangle$" otherwise.

For all distinct tagnames a and b:

$$
\begin{array}{rcll}
\text{check}(\square,\ [\ ]) & \leftarrow & \underline{\$} & \qquad (4.9)\\
\text{check}(x,\ S) & \leftarrow & \underline{\text{check}}(\ \underline{\square},\ S) & \{\ \text{newVar}\ x;\ x := \text{read}()\ \} \qquad (4.10)\\
\text{check}(\square,\ a :: S) & \leftarrow & \underline{\text{check}}(\underline{\langle a\rangle},\ S) & \qquad (4.11)\\
\text{check}(\square,\ S) & \leftarrow & \underline{\text{check}}(\underline{\langle /a\rangle},\ \underline{a}::S) & \qquad (4.12)\\
(\ ) & \leftarrow & \underline{\text{check}}(\underline{\langle /\rangle},\ \underline{[\ ]}) & \{\ \text{write}(\langle wf\rangle);\ \text{write}(\langle /wf\rangle)\ \}\ (4.13)\\
(\ ) & \leftarrow & \underline{\text{check}}(\langle /\rangle,\ \underline{a}::S) & \{\ \text{write}(\langle mf\rangle);\ \text{write}(\langle /mf\rangle)\ \}(4.14)\\
(\ ) & \leftarrow & \underline{\text{check}}(\underline{\langle /b\rangle},\ \underline{a}::S) & \{\ \text{write}(\langle mf\rangle);\ \text{write}(\langle /mf\rangle)\ \}(4.15)
\end{array}
$$

Figure 4.6: *Term rewriting rules for checking well-formedness.*

The system is defined for the start-term "\$" and the rewriting rules from Figure 4.6, which we instantiate for all distinct tagnames a and b. Rule $r_{4.9}$ specifies that the start-term "\$" is replaced by a binary function "check" that takes as arguments the current input token (or placeholder symbol "$\square$") and the current stack. We define the placeholder symbol as a nullary function, and use it to signal that the current input token has been consumed. Accordingly, rule $r_{4.10}$ specifies that if the check-function carries the placeholder "$\square$" as its first argument, then the next input symbol is read.

Rules $r_{4.11}$ and $r_{4.12}$ push the tagname on the stack for reading an opening tag and pop a tagname from the stack for reading a matching closing tag. The input is well-formed if all opening- and closing tags match by the time that we reach the end of the input stream. This is captured by rule $r_{4.13}$. Otherwise, the input is malformed. This case is handled by rules $r_{4.14}$ and $r_{4.15}$.

Example 4.7 We show a run of the XML stream processor $\mathcal{F} = (\$, R)$ where R captures the rules from Figure 4.6 instantiated for all tagnames. We consider the input "$\langle c\rangle\langle /d\rangle$" and show the rewriting steps in Figure 4.7.

The initial configuration is $(\$, \langle c\rangle\langle /d\rangle\langle /\rangle, \{\mathbf{root} \mapsto \bot\}, \epsilon)$. The processor consumes the first input token $\langle c\rangle$ and pushes tagname c onto the stack. When it reads the next token $\langle /d\rangle$, it realizes that the tagnames for the opening- and the closing-tag do not match. Consequently, the output "$\langle mf\rangle\langle /mf\rangle$" is generated, as specified by the last configuration $((\ ), \langle /\rangle, \{\mathbf{root} \mapsto \bot\}, \langle mf\rangle\langle /mf\rangle)$. $\square$

4.3.2 Loading the Buffer

We specify a function that loads the complete input into main memory. Our formalization draws on ideas from XStream [50]. While the general approach is the same, the XStream formalism cannot express the handling of *single* XML tokens from the input stream. XStream queries only specify tree manipulations. In

$$(\$,\ \langle c\rangle\langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon) \qquad \text{match } r_{4.9} \text{ with empty substitution}$$

$$\vdash_{r_{4.9}} (\text{check}(\Box,\ [\,]),\ \langle c\rangle\langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon) \qquad \text{match } r_{4.10} \text{ with } \{S\mapsto[\,]\}$$

execute action of rule $r_{4.10}$:

$$(\text{check}(\Box,\ [\,]),\ \langle c\rangle\langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto[\,]\},\ \epsilon)$$

$$\overset{\text{newVar } x}{\vdash} (\text{check}(\Box,\ [\,]),\ \langle c\rangle\langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto[\,], x\mapsto\bot\},\ \epsilon)$$

$$\overset{x:=\text{ read()}}{\vdash} (\text{check}(\Box,\ [\,]),\ \langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto[\,], x\mapsto\langle c\rangle\},\ \epsilon)$$

update for $\sigma_{4.10} = \{\mathbf{root}\mapsto\bot, S\mapsto[\,], x\mapsto\langle c\rangle\}$:

$$\vdash_{r_{4.10}}(\text{check}(x,\ S)\big[\sigma_{r_{4.10}}\big],\ \langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon)$$

$$= \quad (\text{check}(\langle c\rangle,\ [\,]),\ \langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon) \qquad \text{match } r_{4.11} \text{ with } \{S\mapsto[\,]\}$$

$$\vdash_{r_{4.11}}(\text{check}(\Box,\ c{::}S)\big[\{\mathbf{root}\mapsto\bot, S\mapsto[\,]\}\big],\ \langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon)$$

$$= \quad (\text{check}(\Box,\ c{::}[\,]),\ \langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon) \qquad \text{match } r_{4.10} \text{ with } \{S\mapsto c{::}[\,]\}$$

execute action of rule $r_{4.10}$:

$$(\text{check}(\Box,\ c{::}[\,]),\ \langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto c{::}[\,]\},\ \epsilon)$$

$$\overset{\text{newVar } x}{\vdash} (\text{check}(\Box,\ c{::}[\,]),\ \langle /d\rangle\langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto c{::}[\,], x\mapsto\bot\},\ \epsilon)$$

$$\overset{x:=\text{ read()}}{\vdash} (\text{check}(\Box,\ c{::}[\,]),\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto c{::}[\,], x\mapsto\langle /d\rangle\},\ \epsilon)$$

update for $\sigma_{4.10(b)} = \{\mathbf{root}\mapsto\bot, S\mapsto c{::}[\,], x\mapsto\langle /d\rangle\}$:

$$\vdash_{r_{4.10}}(\text{check}(x,\ S)\big[\sigma_{r_{4.10(b)}}\big],\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon)$$

$$= \quad (\text{check}(\langle /d\rangle,\ c{::}[\,]),\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \epsilon) \qquad \text{match } r_{4.15} \text{ for } \{S\mapsto[\,]\}$$

execute action of rule $r_{4.15}$:

$$(\text{check}(\langle /d\rangle,\ c{::}[\,]),\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto[\,]\},\ \epsilon)$$

$$\overset{\text{write}(\langle mf\rangle)}{\vdash} (\text{check}(\langle /d\rangle,\ c{::}[\,]),\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto[\,]\},\ \langle mf\rangle)$$

$$\overset{\text{write}(\langle /mf\rangle)}{\vdash} (\text{check}(\langle /d\rangle,\ c{::}[\,]),\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot, S\mapsto[\,]\},\ \langle mf\rangle\langle /mf\rangle)$$

update for $\sigma_{4.15} = \{\mathbf{root}\mapsto\bot, S\mapsto[\,]\}$:

$$\vdash_{r_{4.15}}((\ )\big[\sigma_{4.15}\big],\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \langle mf\rangle\langle /mf\rangle)$$

$$= \quad ((\ ),\ \langle /\rangle,\ \{\mathbf{root}\mapsto\bot\},\ \langle mf\rangle\langle /mf\rangle)$$

Figure 4.7: *Steps in checking well-formedness (Example 4.7).*

contrast, our framework allows us to specify both low-level event-handling and higher-level tree manipulations. The intuition is the following. We incrementally construct a binary tree in the main memory buffer, where buffer-variables are placeholders for parts that have not yet been loaded. To assemble this tree, the loader uses a stack of variables. As in the previous application, the stack is modeled as a list and is maintained inside the query-term.

We design the loader so that it processes a list of XML tokens, with the intention of re-using the loader within larger systems, where it will process the output of other functions. The loader assumes that the input is well-formed and takes two lists as arguments. The first is the input-list of XML tokens,

For all tagnames a:

$$\begin{array}{rcll}
\text{load}([x],\, S) & \leftarrow & \underline{\text{load}}(\underline{[\;]},\, S) & \{\ \text{newVar } x;\ x := \text{read}()\ \} \quad (4.16)\\
\text{load}(E,\, x_1 :: x_2 :: S) & \leftarrow & \underline{\text{load}}(\underline{\langle a\rangle}{::}\ E,\, x_0 :: S) & \{\ \text{newVar } x_1; \quad (4.17)\\
 & & & \quad \text{newVar } x_2;\\
 & & & \quad x_0 := a[x_1]x_2\ \}\\
\text{load}(E,\, S) & \leftarrow & \underline{\text{load}}(\underline{\langle /a\rangle}{::}\ E,\, x_0 :: S) & \{\ x_0 := (\;)\ \} \quad (4.18)\\
(\;) & \leftarrow & \underline{\text{load}}([\underline{\langle/\rangle}],\, [x_0]) & \{\ x_0 := (\;)\ \} \quad (4.19)
\end{array}$$

***Figure 4.8**: Term rewriting rules for loading the input.*

and the second is a variable-list modeling a stack. The system is specified in our framework as $\mathcal{F} = (\text{load}([\;],[\textbf{root}]), R)$ with the rewriting rules R from Figure 4.8 instantiated for all tagnames.

Whenever the input-list is empty, rule $r_{4.16}$ fetches the next XML token from the input. For each tagname a, we specify the rules $r_{4.17}$ and $r_{4.18}$. For the opening tag $\langle a\rangle$, we create a new node in the binary tree, by replacing the variable x_0. Details on the first child and next sibling of the new node are unknown, so we insert fresh buffer-variables x_1 and x_2 into the binary tree, which we also store on the stack. For reading the closing tag $\langle /a\rangle$, the variable-stack is popped and a leaf node is inserted in the binary tree. Rule $r_{4.19}$ handles the end of the input and finishes the rewriting process.

Example 4.8 Figure 4.9 shows snapshots in loading an input document. The output tape is not affected, and hence not shown. In step (a), we are about to process the token $\langle b\rangle$, which is contained in the input-list of the loader. The buffer term already contains the a-labeled parent node (in binary tree notation), and variables x_1 and x_2 mark the unknown parts of the tree. Step (b) shows the effect of processing token $\langle b\rangle$ to the buffer-term and the variable-stack. Steps (c) and (d) show the processing of the matching closing tag, which still has to be fetched from the input tape.

In Figure 4.10, we specify the formal steps for processing the XML snippet "$\langle a\rangle\langle b\rangle\langle /b\rangle$", where ϵ denotes the empty input. We do not show the details of executing the action for rule $r_{4.16}$, as this is very similar to the previous examples. Mind that the notations x::[ ] and $[x]$ both describe a list with the single element x, and are treated equally. This example further illustrates the aliasing of buffer-variables, as specified in the previous section. We have marked the configurations that correspond to snapshots (a) through (d). □

4.3.3 Serializing Binary Trees

We define a set of rules in Figure 4.11 that translate a binary tree encoding of an XML document into XML events on the output tape. We call this function a serializer. The serializer traverses the binary tree in depth-first left-to-right order. In this traversal, each node is visited twice. In the first visit, the corresponding opening tag is written to the output, and in the second visit, the closing tag. Accordingly, we define two functions, "output↓" and "output↑", and begin with the downward traversal, as stated in rule $r_{4.20}$. A binary tree encoding τ is then serialized by instantiating "output(τ)".

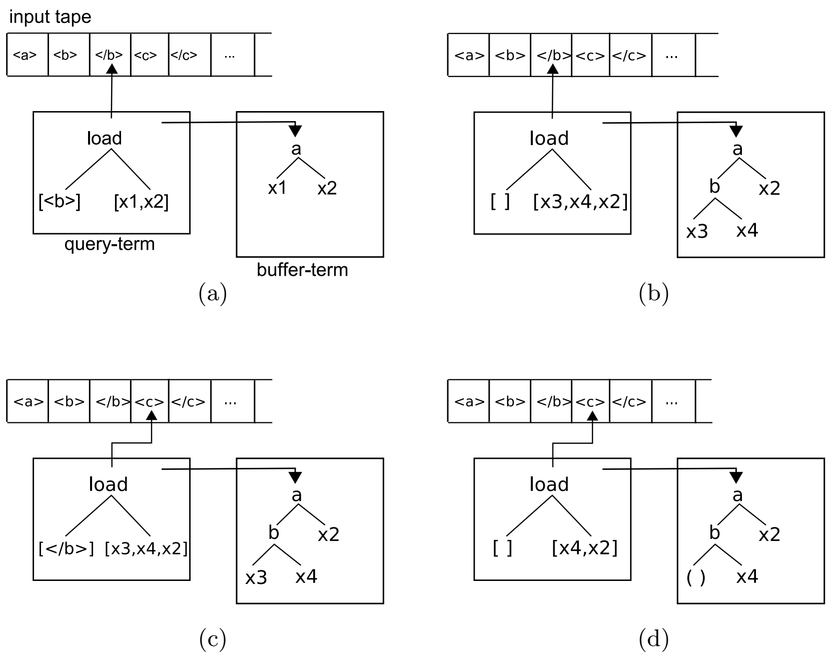


***Figure 4.9**: Snapshots in loading an XML document (Example 4.8).*

Note that the output-functions consume the binary tree, and rewrite it to the term "output↑". As discussed in Remark 4.1, such functions lend themselves nicely for subtree sharing between the query-term and the buffer-term.

Example 4.9 In Figure 4.12, we serialize the binary tree encoding "a[( )]( )" to the output tape. For the sake of clarity, we do not make the execution of actions explicit. Note that in the first application of rule $r_{4.23}$, only the subterm "output↓(( ))" of the query-term is matched by the template. Correspondingly, only this subterm is modified. □

Example 4.10 We assume that the buffer-term is the binary tree from Figure 4.2(b) and instantiate the start-term "output(**root**)". In a call-by-value implementation, the contents of the root-variable are copied and inserted into the query-tree. This practically doubles the main memory requirements. However, we can effectively share subtrees when implementing such a system. This reduces the main memory consumption, and also avoids redundant computations when shared subtrees are modified.

Figure 4.13 depicts a sequence of steps in applying the serializer rewriting rules with subtree sharing. Different from our convention so far, we draw the query-term on top of the buffer-term, to better visualize the sharing of subtrees.

We begin with step (a), where the output-function is instantiated with the buffer-variable **root** as its argument. Using pointers (indicated by arrows), the query-term and the buffer-term share a subtree, namely the complete buffer-term. We then apply the rewriting rules and output the token ⟨*book*⟩. The resulting query-term is depicted in step (b), where we now can share two smaller subtrees with the buffer-term. In the transition to step (c), the opening tag ⟨*title*⟩

$(\mathrm{load}([\,], [\mathbf{root}]),\ \langle a\rangle\langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon)$ match $r_{4.16}$ with $\{S \mapsto [\mathbf{root}]\}$

$\vdash_{r_{4.16}}$ $(\mathrm{load}([\langle a\rangle],\ [\mathbf{root}]),\ \langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon)$

match $r_{4.17}$ with $\{E \mapsto [\,], x_0 \mapsto \mathbf{root}, S \mapsto [\,]\}$,
execute action for $q_{4.17} = \mathrm{load}([\langle a\rangle],\ [\mathbf{root}])$, with variable x_0 as an alias for **root**:

$$(q_{4.17},\ \langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto \bot, E \mapsto [\,], S \mapsto [\,]\},\ \epsilon)$$

$$\overset{\mathrm{newVar}\ x_1}{\vdash} (q_{4.17},\ \langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto \bot, E \mapsto [\,], S \mapsto [\,], x_1 \mapsto \bot\},\ \epsilon)$$

$$\overset{\mathrm{newVar}\ x_2}{\vdash} (q_{4.17},\ \langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto \bot, E \mapsto [\,], S \mapsto [\,], x_1 \mapsto \bot, x_2 \mapsto \bot\},\ \epsilon)$$

$$\overset{\mathbf{root}:=a[x_1]x_2}{\vdash} (q_{4.17},\ \langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto a[x_1]x_2, E \mapsto [\,], S \mapsto [\,], x_1 \mapsto \bot, x_2 \mapsto \bot\},\ \epsilon)$$

update for $\sigma_{4.17} = \{\mathbf{root} \mapsto a[x_1]x_2, E \mapsto [\,], S \mapsto [\,], x_1 \mapsto \bot, x_2 \mapsto \bot\}$

$\vdash_{r_{4.17}}$ $(\mathrm{load}(E, x_1{::}x_2{::}S)\big[\sigma_{4.17}\big],\ \langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto a[x_1]x_2\},\ \epsilon)$

$=$ $(\mathrm{load}([\,], [x_1, x_2]),\ \langle b\rangle\langle/b\rangle,\ \{\mathbf{root} \mapsto a[x_1]x_2\},\ \epsilon)$ match $r_{4.16}$ with $\{S \mapsto [x_1, x_2]\}$

$\vdash_{r_{4.16}}$ $(\mathrm{load}([\langle b\rangle], [x_1, x_2]),\ \ \langle/b\rangle,\ \{\mathbf{root} \mapsto a[x_1]x_2\},\ \epsilon)$, d – **step (a) in Figure 4.9**

match $r_{4.17}$ for substitution $\{E \mapsto [\,], x_0 \mapsto x_1, S \mapsto [x_2]\}$,
execute action for $q_{4.17(b)} = \mathrm{load}([\langle b\rangle], [x_1, x_2])$, with variable x_0 as an alias for x_1,
resolve name conflict by renaming variables x_1, x_2 in term rewriting rule to x_3, x_4:

$$(q_{4.17(b)},\ \langle/b\rangle,\ \{\mathrm{root} \vdash a[x_1]x_2, E \mapsto [\,], S \mapsto [x_2], x_1 \mapsto \bot\},\ \epsilon)$$

$$\overset{\mathrm{newVar}\ x_3}{\vdash} (q_{4.17(b)},\ \langle/b\rangle,\ \{\mathrm{root} \vdash a[x_1]x_2, E \mapsto [\,], S \mapsto [x_2], x_1 \mapsto \bot, x_3 \mapsto \bot\},\ \epsilon)$$

$$\overset{\mathrm{newVar}\ x_4}{\vdash} (q_{4.17(b)},\ \langle/b\rangle, \{\mathrm{root} \vdash a[x_1]x_2, E \mapsto [\,],$$
$$S \mapsto [x_2], x_1 \mapsto \bot, x_3 \mapsto \bot, x_4 \mapsto \bot\},\ \epsilon)$$

$$\overset{x_1:=b[x_3]x_4}{\vdash} (q_{4.17(b)},\ \langle/b\rangle, \{\mathrm{root} \vdash a[x_1]x_2, \big[\{x_1 \mapsto b[x_3]x_4\}\big],$$
$$E \mapsto [\,], S \mapsto [x_2], x_1 \mapsto b[x_3]x_4, x_3 \mapsto \bot, x_4 \mapsto \bot\},\ \epsilon)$$

$$= (q_{4.17(b)},\ \langle/b\rangle, \{\mathrm{root} \vdash a[\ b[x_3]x_4\ ]x_2, E \mapsto [\,], S \mapsto [x_2],$$
$$x_1 \mapsto b[x_3]x_4, x_3 \mapsto \bot, x_4 \mapsto \bot\},\ \epsilon)$$

update for $\sigma_{4.17(b)} = \{\mathbf{root} \vdash a[\ b[x_3]x_4\ ]x_2, E \mapsto [\,], S \mapsto [x_2],$
$x_1 \mapsto b[x_3]x_4, x_3 \mapsto \bot, x_4 \mapsto \bot\}$

$\vdash_{r_{4.17}}$ $(\mathrm{load}(E, x_3{::}x_4{::}\mathrm{S})\big[\sigma_{4.17(b)}\big],\ \langle/b\rangle,\ \{\mathbf{root} \mapsto a[\ b[x_3]x_4\ ]x_2\},\ \epsilon)$

$=$ $(\mathrm{load}([\,], [x_3, x_4, x_2]),\ \langle/b\rangle,\ \{\mathbf{root} \mapsto a[\ b[x_3]x_4\ ]x_2\},\ \epsilon)$, – **step (b) in Figure 4.9**

$\vdash_{r_{4.16}}$ $(\mathrm{load}([\langle/b\rangle], [x_3, x_4, x_2]),\ \epsilon,\ \{\mathbf{root} \mapsto a[\ b[x_3]x_4\ ]x_2\},\ \epsilon)$, – **step (c) in Figure 4.9**
match $r_{4.18}$ for $\{E \mapsto [\,], x_0 \mapsto x_3, S \mapsto [x_4, x_2]\}$,
execute action with variable x_0 as an alias for x_3 and $q_{4.18} = \mathrm{load}([\langle/b\rangle], [x_3, x_4, x_2])$:

$$(q_{4.18},\ \epsilon,\ \{\mathbf{root} \mapsto a[\ b[x_3]x_4\ ]x_2, E \mapsto [\,], S \mapsto [x_4, x_2], x_3 \mapsto \bot\},\ \epsilon)$$

$$\overset{x_3:=(\,)}{\vdash} (q_{4.18},\ \epsilon,\ \{\mathbf{root} \mapsto a[\ b[x_3]x_4\ ]x_2\ \big[\{x_3 \mapsto (\,)\}\big],$$
$$E \mapsto [\,], S \mapsto [x_4, x_2], x_3 \mapsto (\,)\},\ \epsilon)$$

$$= (q_{4.18},\ \epsilon,\ \{\mathbf{root} \mapsto a[\ b[(\,)]x_4\ ]x_2\quad E \mapsto [\,], S \mapsto [x_4, x_2], x_3 \mapsto (\,)\},\ \epsilon)$$

update for $\sigma_{4.18}\ =\ \{\mathbf{root} \mapsto a[\ b[(\,)]x_4\ ]x_2\quad E \mapsto [\,], S \mapsto [x_4, x_2], x_3 \mapsto (\,)\}$

$\vdash_{r_{4.18}}$ $(\mathrm{load}(E, S)\big[\sigma_{4.18}\big],\ \ \epsilon,\ \{\mathbf{root} \mapsto a[\ b[(\,)]x_4\ ]x_2\},\ \epsilon)$

$=$ $(\mathrm{load}([\,], [x_4, x_2]),\ \ \epsilon,\ \{\mathbf{root} \mapsto a[\ b[(\,)]x_4\ ]x_2\},\ \epsilon)$ – **step (d) in Figure 4.9**

***Figure 4.10**: Steps in loading an XML document (Example 4.8).*

For all tagnames a:

$$\begin{array}{rcll}
\text{output}{\downarrow}(e) & \leftarrow & \underline{\text{output}}(e) & \qquad (4.20)\\
a[\ \text{output}{\downarrow}(e_1)\ ]e_2 & \leftarrow & \underline{\text{output}{\downarrow}}(\ \underline{a}[e_1]e_2\ ) \quad \{\ \text{write}(\langle a\rangle)\ \} & \qquad (4.21)\\
\text{output}{\downarrow}(e) & \leftarrow & \underline{a}[\ \underline{\text{output}{\uparrow}}\ ]e \quad \{\ \text{write}(\langle /a\rangle)\ \} & \qquad (4.22)\\
\text{output}{\uparrow} & \leftarrow & \underline{\text{output}{\downarrow}}(\ \underline{(\ )}\ ) & \qquad (4.23)
\end{array}$$

***Figure 4.11**: Term rewriting rules for serializing a binary tree.*

$$\begin{array}{lll}
 & (\text{output}(\ \text{a}[(\ )](\ )\ ),\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon) & \text{match } r_{4.20} \text{ with } \{e \mapsto \text{a}[(\ )](\ )\}\\
\vdash_{r_{4.20}} & (\text{output}{\downarrow}(e)\big[\{\mathbf{root} \mapsto \bot, e \mapsto \text{a}[(\ )](\ )\}\big],\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon) & \\
= & (\text{output}{\downarrow}(\ \text{a}[(\ )](\ )\ ),\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \epsilon) & \text{match } r_{4.21} \text{ for } \{e_1 \mapsto (\ ), e_2 \mapsto (\ )\}\\
\vdash_{r_{4.21}} & (\text{a}[\ \text{output}{\downarrow}(e_1)\ ]e_2\ \big[\{\mathbf{root} \mapsto \bot, e_1 \mapsto (\ ), e_2 \mapsto (\ )\}\big],\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle) & \\
= & (\text{a}[\ \text{output}{\downarrow}(\ (\ )\ )\ ](\ ),\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle) & \text{match } r_{4.23} \text{ for empty substitution}\\
\vdash_{r_{4.23}} & (\text{a}[\ \text{output}{\uparrow}\ ](\ ),\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle) & \text{match } r_{4.22} \text{ for } \{e \mapsto (\ )\}\\
\vdash_{r_{4.22}} & (\text{output}{\downarrow}(\ e\ )\big[\{\mathbf{root} \mapsto \bot, e \mapsto (\ )\}\big],\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle\langle /a\rangle) & \\
= & (\text{output}{\downarrow}(\ (\ )\ ),\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle\langle /a\rangle) & \text{match } r_{4.23} \text{ for empty substitution}\\
\vdash_{r_{4.23}} & (\text{output}{\uparrow},\ \epsilon,\ \{\mathbf{root} \mapsto \bot\},\ \langle a\rangle\langle /a\rangle) &
\end{array}$$

***Figure 4.12**: Steps in serializing a binary tree (Example 4.9).*

is output, and we now share three subtrees. Next, we consider step (d). The output-function moves up one level in the binary tree, and proceeds to process the author-node. In the transition to step (e), the closing tag ⟨/*title*⟩ is output.

We observe that the size of the query-term is bounded by the depth of the buffer-term. Subtree-sharing is straightforward to realize in this case. For more details on this technique, we refer to the XStream system [50]. □

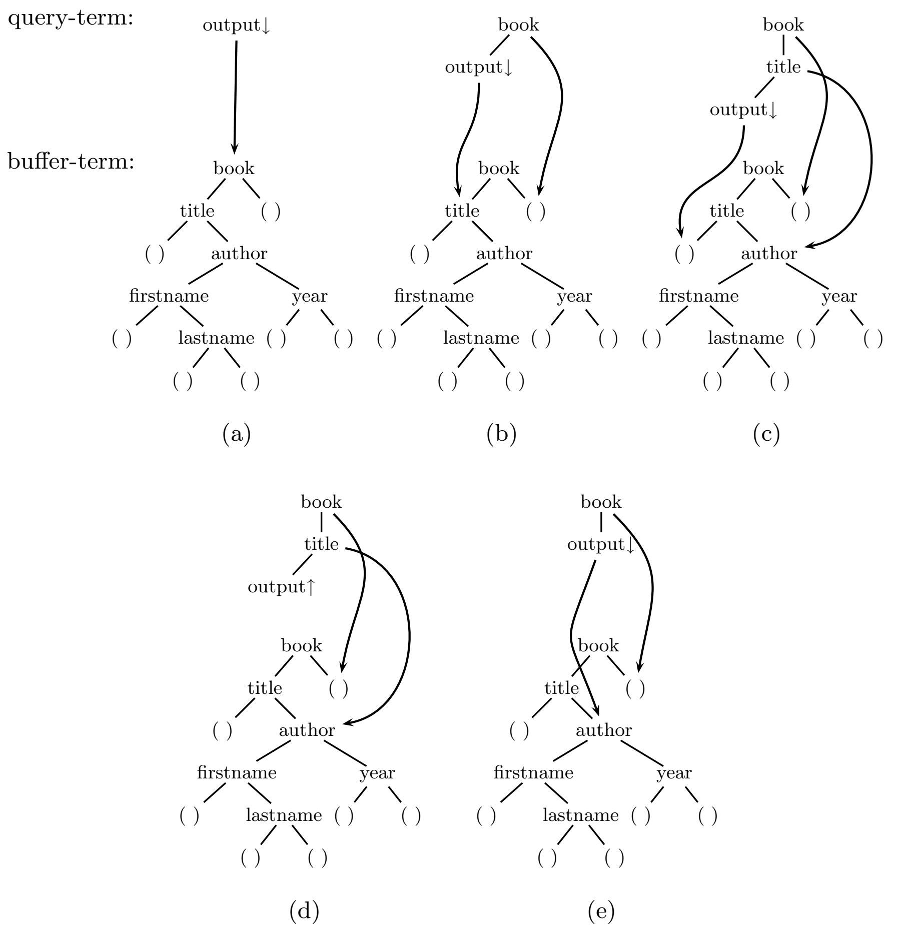


Figure 4.13: *Subtree sharing (Example 4.10).*

5 Modeling XML Query Engines

In this chapter, we model two XML processors in our abstract framework. The first, discussed in Section 5.1, is an encoding of XML pushdown transducers. This allows for the memory-efficient implementation of a range of important XML stream processing tasks. In Section 5.2, we further model a main memory-based XQuery engine that evaluates our XQuery fragment XQ. We construct this engine according to the two-phase approach of first loading the complete input into a buffer before executing the query. In the next chapter, we will discuss strategies to improve on this processing model with respect to XML stream processing.

5.1 XML Pushdown Transducers

In Section 3.2, we have introduced *XML-DPDT*s as deterministic pushdown transducers for XML stream processing. The *XML-DPDT* formalism is capable of encoding a range of important XML stream processing tasks. As sketched in Chapter 3, we may use *XML-DPDT*s for the purpose of validation against a DTD, or to execute the projection algorithm from Section 3.5. Yet *XML-DPDT*s can also encode certain XML stream transformations. While pushdown transducers can be evaluated very efficiently over XML streams, their manual specification is cumbersome and error-prone. Consequently, users may be little inclined to actually encode XML stream transformations in this formalism. Out of this consideration, we have developed a higher-level querying-formalism based on attribute grammars. These *XML Stream Attribute Grammars* (XSAGs) have precisely the same expressiveness as *XML-DPDT*s, so they can be compiled into pushdown transducers. Thus, XSAGS can easily be executed in our framework as well. We regard it as their strong point that they provide an intuition for which queries can, and cannot, be evaluated scalably on streams. We refer to [69] for our earlier work on the XSAG formalism.

We next show how *XML-DPDT*s can be modeled in our framework.

Modeling *XML-DPDT*s. Given an *XML-DPDT*, we encode its transition function using a term-manipulating function "delta". Further below, we will also introduce a function "xdpdt" that is responsible for I/O. The motivation for separating these two functionalities is that we want to re-use the definition of function "delta" in the next chapter.

$$
\begin{array}{rcll}
\sigma_1{::}\ldots{::}\sigma_n{::}\text{delta}(p,\ \Box,\ [(t,Y)]) & \longleftarrow & \underline{\text{delta}}(\underline{q_0},\ \underline{\langle t\rangle},\ \underline{[\,]}) & (5.1)\\
\sigma_1{::}\ldots{::}\sigma_n{::}\text{delta}(p,\ \Box,\ (t,Y){::}(a,B){::}S) & \longleftarrow & \underline{\text{delta}}(\underline{q},\ \ \underline{\langle t\rangle},\ (a,B){::}S) & (5.2)\\
\sigma_1{::}\ldots{::}\sigma_n{::}\text{delta}(p,\ \Box,\ S) & \longleftarrow & \underline{\text{delta}}(\underline{q},\ \underline{\langle /t\rangle},\ \underline{(t,Y)}{::}S) & (5.3)\\
\langle / \rangle & \longleftarrow & \underline{\text{delta}}(\underline{q},\ \underline{\langle / \rangle},\ \underline{[\,]}). & (5.4)\\
\text{xdpdt}(\ \text{delta}(q,\ x,\ S)\ ) & \longleftarrow & \underline{\text{xdpdt}}(\ [\underline{\text{delta}}(q,\ \underline{\Box},\ S)]\ ) & (5.5)\\
 & & \qquad\{\ \text{newVar}\ x;\ x := \text{read}()\ \} & \\
\text{xdpdt}(L) & \longleftarrow & \underline{\text{xdpdt}}(\ \underline{\langle t\rangle}{::}\ L)\quad \{\ \text{write}(\langle t\rangle)\ \} & (5.6)\\
\text{xdpdt}(L) & \longleftarrow & \underline{\text{xdpdt}}(\underline{\langle /t\rangle}{::}\ L)\quad \{\ \text{write}(\langle /t\rangle)\ \} & (5.7)\\
(\ ) & \longleftarrow & \underline{\text{xdpdt}}(\ [\underline{\langle / \rangle}]\ )\qquad \{\ \text{write}(\langle / \rangle)\ \} & (5.8)
\end{array}
$$

***Figure 5.1**: Term rewriting rules for modeling a XML-DPDT.*

Encoding the transition function. We assume that *XML-DPDT*s are without epsilon-transitions, as these can always be removed [69]. Given the specification $\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$ of an *XML-DPDT*, we now define the function "delta". This function (just like the *XML-DPDT* transition function) takes the current state, the input symbol, and the stack contents as parameters. The stack is realized using a list. We define nullary functions to model the states and stack symbols of $\mathcal{T}$.

For each initial transition of the form $\delta(q_0, \langle t\rangle, Z_0) = (p, (t, Y), \sigma_1 \ldots \sigma_n)$ (where the σ_i are output tokens), we define a term rewriting rule according to rule $r_{5.1}$ in Figure 5.1. When the input token $\langle t\rangle$ is consumed, we replace it with the placeholder term "$\Box$". Any output produced by the *XML-DPDT* is encoded as a list of XML tokens, where the last element is the rewritten delta-function. If the transition produces no output, we produce no prefix.

All remaining transitions of the form $\delta(q, \langle t\rangle, X) = (p, (t, Y)X, \sigma_1 \ldots \sigma_n)$ are encoded analogously, with the difference that we push a tuple onto the stack (see rule $r_{5.2}$). Note that the template-term requires that the stack is not empty. As we do not consider character data in our framework, we need not specify rewriting rules for handling them. For reading closing tags, the stack is popped if the closing tag matches the corresponding opening tag, as *XML-DPDT*s check the well-formedness of their input. So for each transition $\delta(q, \langle /t\rangle, (t, Y)) = (p, \epsilon, \sigma_1 \ldots \sigma_n)$, we devise a rule according to $r_{5.3}$.

Finally, we model the acceptance of *XML-DPDT*s by empty stack. For each state q, we thus instantiate rule $r_{5.4}$.

Encoding I/O. Function "xdpdt" reads the input one token at-a-time and feeds this token into the delta-function. It also takes care that output computed by delta is immediately written to the output tape. We again refer to Figure 5.1 for the rewriting rules. Rule $r_{5.5}$ reads a new token once "delta" has consumed its input, indicated by the placeholder "$\Box$". Rules $r_{5.6}$ and $r_{5.7}$ ensure that the token stream produced by the *XML-DPDT* is written to the output tape. The rewriting terminates when the *XML-DPDT* has accepted its input, see rule $r_{5.8}$.

Putting it all together. We model an *XML-DPDT* in our framework as an application $\mathcal{F} = (\text{xdpdt}(\ [\text{delta}(q_0,\ \Box,\ [\,])]\ ), R)$, where the term rewriting

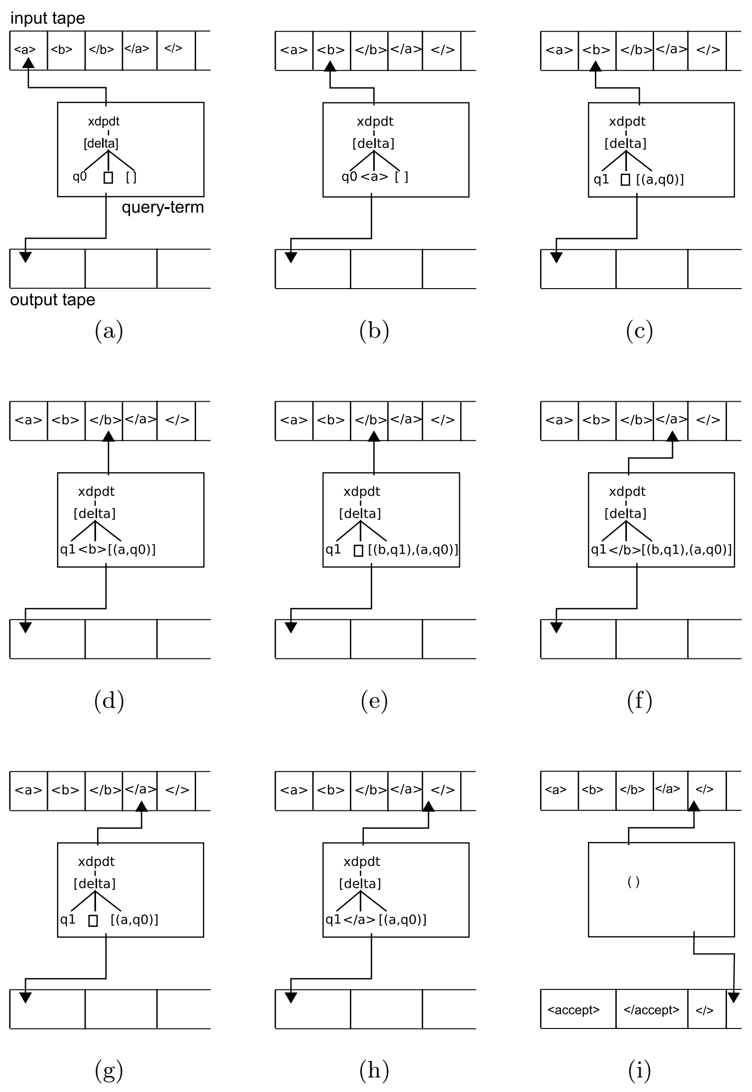
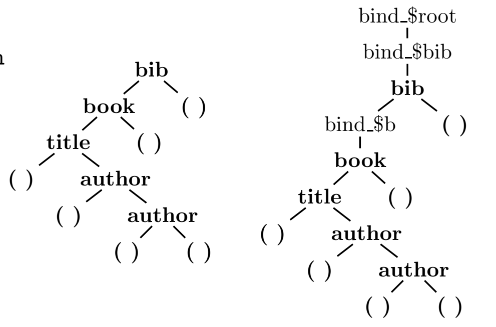


Figure 5.2: *Snapshots of executing an XML-DPDT (Example 5.1).*

rules R are obtained as described above.

Example 5.1 We model the *XML-DPDT* from Example 3.6 and execute it for the input "$\langle a\rangle\langle b\rangle\langle /b\rangle\langle /a\rangle$". In Figure 5.2, we show snapshots of the system at runtime. Snapshot (a) shows the start-term, yet not the buffer contents, as the buffer-term is not accessed. The xdpdt-function reads the first input token and feeds it to the delta-function. The delta-function pushes the current tagname and state on the stack, and we arrive in snapshot (b). The stack grows and shrinks as the tags for the b-labeled node are processed (steps (c) through (f)). Upon reading the opening tag of the second a-labeled node in document order (see step h), the input is rejected, as this node has the same label as its parent node. We terminate with step (i) and the term "( )" as the query-term. □

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



(a) XQuery expression. (b) Buffer-term. (c) Variable bindings.

Figure 5.3: *XQuery and buffer-terms.*

5.2 XQuery Evaluation

We next model an XQuery processor which uses unbounded buffers, so different from *XML-DPDT*s, scalability cannot be guaranteed. Our query processor evaluates the XQuery fragment XQ from Section 3.4 and operates in a two-phase approach. In the first phase, the input is loaded into a main memory buffer, and in the second, the query is evaluated on the buffered data. For instance, the IPSI, QizX, and Saxon XQuery engines operate in this mode [40, 90, 93].

Query fragment. As we only consider XML documents with element nodes in our framework, we restrict ourselves to queries with the node test "*" and the test for tagnames. In the absence of text nodes, the XPath node tests "`node()`" and "*" can be considered equivalent. Additionally, we restrict comparisons to equality checks (rather than less-than or greater-than comparisons).

Components. Our query engine will consist of several components, among them a *loader* and a *serializer* as introduced in the previous chapter. Further, it requires a *variable environment* relating query-variables to nodes in the buffer-term, and a mechanism for *checking conditions* on the buffer-term. We address these issues next, with the exception of the loader and serializer, which we have already introduced. This is followed by a systematic compilation of XQueries into term rewriting rules, so that they may be evaluated in our framework.

5.2.1 Variable Environment

We model variable environments, i.e. bindings of query-variables to XML document nodes, by introducing function symbols into the buffer-term, which mark the bound node. To avoid confusion when referring to variables, we will make it clear whether we refer to buffer-variables (from our framework) or query-variables (occurring in XQueries).

Example 5.2 Consider the query from Figure 5.3(a), and the buffer-term from Figure 5.3(b). In evaluating the for-loops for query-variables \$bib and \$b, these variables are bound to nodes in the buffered tree. This is modeled by introducing function symbols "bind_$bib" and "bind_$b" into the buffer-term, above the node to which they are bound. This is visualized in Figure 5.3(c). The designated query-variable $root binds the root of the buffered tree. □

Approach. We assume a set of functions for the manipulation of variable bindings. Consider the evaluation of a for-loop of the form "for $\$x$ in $\$y/\pi$ return α", where query-variable $\$y$ has already been bound to a node in the buffer-term. To evaluate this expression, we first check whether a binding for $\$x$ is possible. If this is the case, then we bind the variable $\$x$ to the first such node in document order, then we check whether another binding is possible. If this is the case, the variable is bound anew and the body of the for-loop is evaluated for this binding as well. This is repeated while new bindings can be found.

We only describe the functions for manipulating the variable environment informally, as their specification is quite straightforward yet technically involved. For modeling the variable environment, we construct buffer-terms according to the grammar

$$\beta \quad ::= \quad (\,) \;\mid\; a[\beta]\beta \;\mid\; x \;\mid\; \text{bind\_\$}y(\beta)$$

where a is a tagname, x ranges over buffer-variable names, $\$y$ ranges over query-variable names, and "bind_$\$y$" is a unary function.

We define two types of functions over buffer-terms. Functions of the first type copy the buffer-term into the query-term, and then perform a single pass over this tree. These functions lend themselves nicely for subtree sharing (see Remark 4.1). We use this approach for checking conditions on the buffer-term, or for selecting nodes bound to a given query-variable.

The functions of the second type modify the buffer-term. For this purpose, we define rules of the form

$$\tau' \quad \leftarrow \quad \tau \qquad \{\ldots\ \textbf{root} := f(\textbf{root});\ \ \ldots\ \}$$

where buffer-variable **root** is reassigned with the result of rewriting $f(\textbf{root})$. Along this line, we will shortly introduce a function "remove_binding($\$x$,**root**)". This function removes the binding for query-variable $\$x$ from the buffer-term.

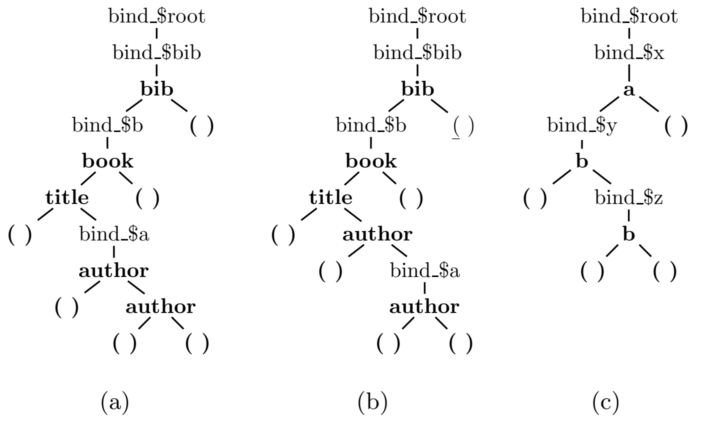


Figure 5.4: *Buffer-terms with variable bindings.*

Selecting bound nodes. We assume a function “select” that prunes a copy of the buffer-term in the query-term down to the first node in document order to which a given variable is bound. Function “select($\$x, \tau$)” assumes that variable $\$x$ is already bound in term τ.

Example 5.3 Let the term from Figure 5.4(a) be the buffer-term. Below, we apply the select-function for two different query-variables.

select($b,**root**) = **book**[**title**[( )] bind_$a(**author**[( )] **author**[( )]( ))]( )
select($a,**root**) = **author**[( )]( )

Note that the following-siblings of the selected node are not returned, as is evident for select($a,**root**). At the same time, the variable bindings within the selected tree are preserved, such as in select($b,**root**). □

Removing variable bindings. Given a buffer-term τ, we remove the bindings of a query-variable $\$x$ with function “remove_binding($\$x$, τ)”. To remove all variable-bindings from τ, we use function “remove_all_bindings(τ)”.

Example 5.4 Given the buffer-term from Figure 5.4(a), rewriting the term “remove_binding($a, **root**)” yields the term shown in Figure 5.3(c), while “remove_all_bindings(**root**)” yields the buffer-tree from Figure 5.3(b). □

Example 5.5 In evaluating queries, we will frequently output nodes to which query-variables are bound. Let us assume that the query-variable $\$y$ is bound in a term τ. To serialize the node to which $x is bound to the output tape, we instantiate the term “output(remove_all_bindings(select($\$y, \tau$)))”. □

Existence checks. We define two existence checks which both return the Boolean values “true” or “false” as constant terms. Function “exists($\$x/\pi, \tau$)” returns true, if there is a match in term τ for the relative XPath expression π with the node bound by $\$x$ as its context node.

Example 5.6 We consider the query from Figure 5.3(a) and the buffer-term from Figure 5.3(c). The term “exists($b/author,**root**)” checks whether there is a binding for variable $a within the given book. This is the case, so term rewriting yields the term “true”. If we instantiate “exists($b/year, **root**)”, we obtain the term “false”, as no such node exists in the buffer-tree. □

Function “has_next($\$x/\pi, \$y, \tau$)” assumes that query-variables $\$x$ and $\$y$ are both bound in term τ. The function returns “true” if a further binding for $\$y$ is possible, to a node that is matched by $\$x/\pi$, but occurs after the node to which $\$y$ is currently bound (in document order).

Example 5.7 We continue with Example 5.6. Then “has_next($b/author, $a, **root**)” yields “true”, while “has_next($bib/book, $b, **root**)” yields “false”. □

Binding nodes. Function “bind_first($\$x/\pi, \$y, \tau$)” assumes that the term “exists($\$x/\pi, \tau$)” can be rewritten to “true”. That is, there is at least one node in the buffer-term that is matched by path expression $\$x/\pi$. Then the function “bind_first($\$x/\pi$, $\$y$, τ)” modifies the term τ such that query-variable $\$y$ binds the first such node in document order.

Example 5.8 We again consider the query from Figure 5.3(a), and the buffer-term from Figure 5.3(c). In Example 5.6, we have already checked that a binding for query-variable \$a from the for-loop over authors is possible. Then the statement "**root** := bind_first(\$b/author, \$a, **root**)" computes the buffer-term from Figure 5.4(a), where \$a is bound to the first author-node in document order. □

The function "bind_next($\$x/\pi, \$y, \tau$)" assumes that "has_next($\$x/\pi, \$y, \tau$)" yields "true". Then "bind_next" locates the closest node in document order that matches $\$x/\pi$ and that is a follower of the node that is currently bound by $\$y$. The old binding of $\$y$ is released, and the query-variable now binds the newly located node.

Example 5.9 We resume the previous example, where we have already checked that "has_next(\$b/author, \$a, **root**)" evaluates to true for the buffer-term from Figure 5.4(a). Reassigning the **root**-variable with "bind_next(\$b/author, \$a, **root**)" yields the buffer-term from Figure 5.4(b). □

Comparing nodes. Let $\$x$ and $\$x'$ be two query-variables that are bound in term τ. Then "compare($\$x, \$x', \tau$)" returns true if the nodes bound by these variables in term τ are structurally equal (according to the XML document model). In comparing nodes in the buffer-term, we ignore variable bindings.

Example 5.10 We once more assume the buffer-term from Figure 5.4(c). where "compare($\$x, \$y, \mathbf{root}$)" yields false, and "compare($\$y, \$z, \mathbf{root}$)" yields true. □

Conditionals. For evaluating conditionals, we introduce the general term rewriting rules from Figure 5.5.

$$
\begin{array}{rcl@{\qquad}rcl}
e_1 & \leftarrow & \underline{\text{if\_then\_else}}(\underline{\text{true}}, e_1, e_2) & \text{true} & \leftarrow & \underline{\text{not}}(\underline{\text{false}}) \\
e_2 & \leftarrow & \underline{\text{if\_then\_else}}(\underline{\text{false}}, e_1, e_2) & \text{false} & \leftarrow & \underline{\text{not}}(\underline{\text{true}}) \\
\text{true} & \leftarrow & \underline{\text{or}}(\underline{\text{true}}, e) & \text{false} & \leftarrow & \underline{\text{and}}(\underline{\text{false}}, e) \\
\text{true} & \leftarrow & \underline{\text{or}}(e, \underline{\text{true}}) & \text{false} & \leftarrow & \underline{\text{and}}(\text{e}, \underline{\text{false}}) \\
\text{false} & \leftarrow & \underline{\text{or}}(\underline{\text{false}}, \underline{\text{false}}) & \text{true} & \leftarrow & \underline{\text{and}}(\underline{\text{true}}, \underline{\text{true}})
\end{array}
$$

***Figure 5.5**: Term rewriting rules for evaluating conditionals.*

5.2.2 Query Compilation

We next compile XQueries from our fragment XQ into a term rewriting system.

Query compilation. Given a normalized XQuery Q, the compilation proceeds top-down in a single pass over the query parse tree, compiling query expressions into terms and term rewriting rules. Each query subexpression q with k free query-variables has access to a binary tree encoding of the current environment (denoted *env*), where the k free query-variables are bound. In our compilation, this is denoted by $[\![q]\!]_k(\mathit{env})$. We begin with $[\![Q]\!]_1(\mathbf{root})$, where

$[\![\langle a\rangle q\langle/a\rangle]\!]_k(e)$ / $\langle a[\ [\![q]\!]_k(e)\ ](\ ),\ \emptyset\rangle$

$[\![(\ )]\!]_k(e)$ / $\langle(\ ),\ \emptyset\rangle$

$[\![\$x]\!]_k(e)$ / $\langle\text{remove\_all\_bindings}(\text{select}(\$x, e)),\ \emptyset\rangle$

$[\![(q_1, \ldots, q_n)]\!]_k(e)$ /
$\langle\text{concat}\big(\ [\![q_1]\!]_k(e),\ \text{concat}([\![q_2]\!]_k(e), \ldots \text{concat}([\![q_{n-1}]\!]_k(e), [\![q_n]\!]_k(e)))\ \big),\ \emptyset\rangle$

$[\![\text{for } \$x \text{ in } \$y/\pi \text{ return } \alpha]\!]_k(e)$ /
$\langle\text{for}_{\$x}(e),\ \big\{\text{check\_first\_binding}_{\$x}(\text{exists}(\$y/\pi, z), z) \quad\leftarrow\quad \underline{\text{for}_{\$x}}(z),$
$\text{concat}([\![q]\!]_{k+1}(z'), \text{check\_next\_binding}_{\$x}(\text{has\_next}(\$y/\pi, \$x, z'), z'))$
$\quad\leftarrow\quad \underline{\text{check\_first\_binding}_{\$x}}(\underline{\text{true}}, z)\ \ \{\text{newVar } z';\ z'{:=}\text{bind\_first}(\$y/\pi, \$x, z)\},$
$\text{concat}([\![q]\!]_{k+1}(z'), \text{check\_next\_binding}_{\$x}(\text{has\_next}(\$y/\pi, \$x, z'), z'))$
$\quad\leftarrow\quad \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{true}}, z)\ \ \{\text{newVar } z';\ z'{:=}\text{bind\_next}(\$y/\pi, \$x, z)\},$
$(\ ) \quad\leftarrow\quad \underline{\text{check\_first\_binding}_{\$x}}(\underline{\text{false}}, z),$
$(\ ) \quad\leftarrow\quad \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{false}}, z)\big\}\rangle$

$[\![\text{if } \chi \text{ then } \alpha \text{ else } \beta]\!]_k(e)$ / $\langle\text{if\_then\_else}([\![\chi]\!]_k(e), [\![\alpha]\!]_k(e), [\![\beta]\!]_k(e)),\ \emptyset\rangle$

$[\![\text{some } \$x \text{ in } \$y/\pi \text{ satisfies } \chi]\!]_k(e)$ /
$\langle\text{some}_{\$x}(e),\ \big\{\text{check\_first\_binding}_{\$x}(\text{exists}(\$y/\pi, z), z) \quad\leftarrow\quad \underline{\text{some}_{\$x}}(z),$
$\text{or}([\![q]\!]_{k+1}(z'), \text{check\_next\_binding}_{\$x}(\text{has\_next}(\$y/\pi, \$x, z'), z'))$
$\quad\leftarrow\quad \underline{\text{check\_first\_binding}_{\$x}}(\underline{\text{true}}, z)\ \ \{\text{newVar } z';\ z'{:=}\text{bind\_first}(\$y/\pi, \$x, z)\},$
$\text{or}([\![q]\!]_{k+1}(z'), \text{check\_next\_binding}_{\$x}(\text{has\_next}(\$y/\pi, \$x, z'), z'))$
$\quad\leftarrow\quad \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{true}}, z)\ \ \{\text{newVar } z';\ z'{:=}\text{bind\_next}(\$y/\pi, \$x, z)\},$
$\text{false} \quad\leftarrow\quad \underline{\text{check\_first\_binding}_{\$x}}(\underline{\text{false}}, z),$
$\text{false} \quad\leftarrow\quad \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{false}}, z)\big\}\rangle$

$[\![\$x = \$y]\!]_k(e)$ / $\langle\text{compare}(\$x, \$y, e),\ \emptyset\rangle$

$[\![\text{true}()]\!]_k(e)$ / $\langle\text{true},\ \emptyset\rangle$

$[\![\text{false}()]\!]_k(e)$ / $\langle\text{false},\ \emptyset\rangle$

Figure 5.6: *XQuery compilation rules $[\![q]\!]_k(e)/\langle\tau, R\rangle$ for eager evaluation.*

query-variable \$root as the single free query-variable is bound to the buffer-term, which is accessible via buffer-variable **root**.

In Figure 5.6, we specify rules of the form "$[\![q]\!]_k(env)/\langle\tau, R\rangle$", where a query expression with k free variables that is of the form q is replaced by the term τ, and a set of term rewriting rules is generated according to pattern R. The compilation proceeds until no more terms or rules can be generated.

Putting it all together. Phase one of query evaluation, the loading of the input, is initiated by the start-term "load([ ],[**root**])". Loading proceeds as specified in the previous chapter. By the time that the input has been loaded, the start-term has been reduced to the empty term. We then proceed with phase two and start the query evaluation by the rule

$$\text{output}([\![Q]\!]_1(\textbf{root})) \quad\leftarrow\quad \underline{(\ )} \quad \{\ \textbf{root} := \text{bind\_\$root}(\textbf{root})\ \}.$$

That is, the implicit query-variable **\$root** is bound to the root of the buffer-term in executing the action. The replacement term specifies that any results produced in XQuery evaluation are to be immediately serialized to the output. The following example illustrates the compilation of a simple query.

Example 5.11 Given the XQuery

```
<x>{ for $b in //book return  <y>{$b}</y> }</x>
```

we begin with the start-term "load([ ],[**root**])", and compile the rules below.

$\text{output}(x[\text{for}_{\$b}(\mathbf{root})](\,)) \quad \leftarrow \quad \underline{(\,)}$
$\text{check\_first\_binding}_{\$b}(\text{exists}(\$\text{root}//\text{book}, z), z) \quad \leftarrow \quad \underline{\text{for}_{\$b}}(z),$
$\text{concat}(y[\text{remove\_all\_bindings}(\text{select}(\$b, z'))](\,),$
$\quad \text{check\_next\_binding}_{\$b}(\text{has\_next}(\$\text{root}//\text{book}, \$b, z'), z'))$
$\quad \leftarrow \quad \underline{\text{check\_first\_binding}_{\$b}}(\underline{\text{true}}, z) \ \{\text{newVar } z';\ z' {:=} \text{bind\_first}(\$\text{root}//\text{book}, \$b, z)\},$
$\text{concat}(y[\text{remove\_all\_bindings}(\text{select}(\$b, z'))](\,),$
$\quad \text{check\_next\_binding}_{\$b}(\text{has\_next}(\$\text{root}//\text{book}, \$b, z'), z'))$
$\quad \leftarrow \quad \underline{\text{check\_next\_binding}_{\$b}}(\underline{\text{true}}, z) \ \{\text{newVar } z';\ z' {:=} \text{bind\_next}(\$\text{root}//\text{book}, \$b, z)\},$
$(\,) \leftarrow \quad \underline{\text{check\_first\_binding}_{\$b}}(\underline{\text{false}}, z),$
$(\,) \leftarrow \quad \underline{\text{check\_next\_binding}_{\$b}}(\underline{\text{false}}, z).$ □

Discussion. Due to the functional nature of XQuery, all bindings of a query-variable can be evaluated in parallel. We refer to this mode as *eager*. Eager evaluation has been realized in various XML processors, even processors designed for XML stream processing [42, 50]. The queries compiled according to the rules from Figure 5.6 evaluate query expressions eagerly, and use concatenation to ensure that output is produced in the correct order.

The problem with this approach is that the main memory footprint can be unnecessarily high, as intermediate results must be stored until it is their turn for output. The following example illustrates this point.

Example 5.12 Let us consider the query from Example 5.11 and a buffered XML document with three books, as sketched in Figure 5.7(a). We abbreviate the subtrees of books by triangles. Once the input has been loaded, query evaluation begins with the term from Figure 5.7(b). We mark up terms that form binary tree encodings in bold font.

Subfigures (b) through (f) show a sequence of possible snapshots of the query-term in term rewriting. All possible bindings for query-variable \$b are located and evaluated in parallel. In snapshot (b), we begin with the evaluation of the for-loop over books. That is, we check whether a binding for variable \$b is possible (step (c)), using the exists-function. As this existence check evaluates to true (step (d)), the first matching book-node in document order is bound by query-variable \$b. We begin to evaluate the body of the for-loop for this binding, but also search for further bindings (see step (e)). Altogether, three bindings of variable \$b are possible, and can be evaluated in parallel.

In snapshot (f), the tags "$\langle x\rangle\langle y\rangle$" have already been output, and the first book is about to be output. Snapshot (g) shows an even further advanced situation. Yet as long as the first book has not yet been serialized, the second and third book must be stored in the query-term until it is their turn to be serialized to the output. □

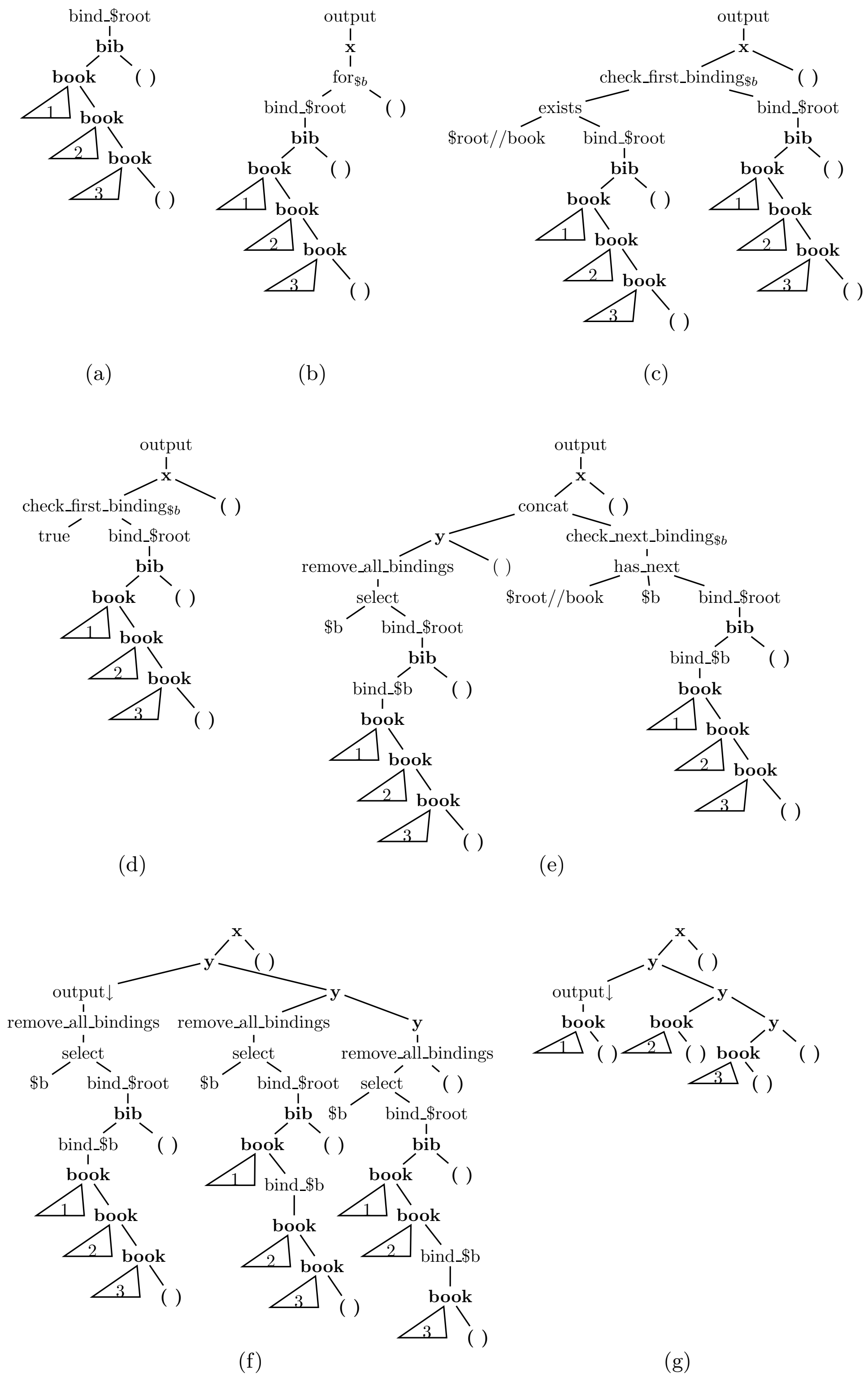


Figure 5.7: (a) A buffer-term and (b)-(g) query-terms (Example 5.12).

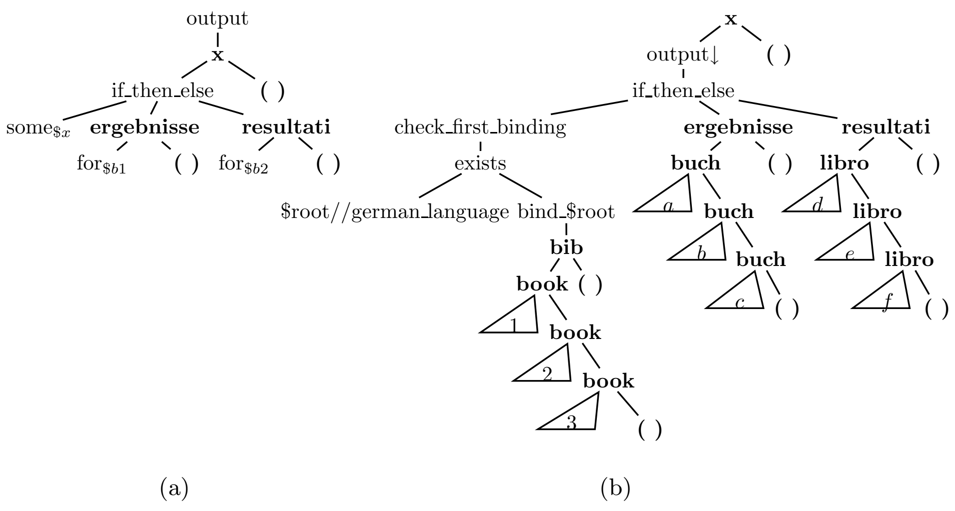


Figure 5.8: *Snapshots of query-terms (Example 5.13).*

As pointed out in the previous example, query evaluation requires sufficient memory to buffer intermediate results. Frequently, this memory overhead can be diminished by subtree sharing. But as Example 5.13 illustrates, there are queries where subtree sharing between intermediate results is of no effect.

Example 5.13 The following query makes the query output dependent on a condition. If the condition is satisfied, then element nodes are relabeled in German, otherwise in Italian.

```
<x>
{ if ( some $x in //german_language satisfies true() )
  then <ergebnisse>
       { for $b1 in //book return
          <buch>
           {( for $t in $b1/title return <titel></titel>,
              for $a in $b1/year  return <jahr></jahr> )}
          </buch> }
       </ergebnisse>
  else <risultati>
       { for $b2 in //book return
          <libro>
           {( for $t in $b2/title return <titolo></titolo>,
              for $a in $b2/year  return <anno></anno> )}
         </libro> }
   </risultati> }
</x>
```

If this query is evaluated in our framework, then query evaluation (after the input has been loaded) begins with the query-term from Figure 5.8(a). We assume the buffer-term from Figure 5.7(a). Further, we assume that this tree contains no node satisfying the condition and that the subtrees labeled one through tree are structurally different (and hence cannot be shared).

In our framework, there is no predefined order in which term rewriting rules are applied. We may obtain the intermediate query-term from Figure 5.8(b) where the condition has not yet been evaluated, but the results of the alternative execution paths have already been computed. Note that the indexes a through g in the figure denote structurally different subtrees. In this query-term, subtree sharing between the trees produced by the then- and else-path is only possible for leaf nodes in the binary tree representation. Thus, the savings are marginal and even with subtree sharing, the eager evaluation of this query requires at least enough memory to buffer the intermediate results.

Moreover, we point out that if we deviate from the two-phase model and evaluate the query while the input is being read, the problem remains. The reason is that the condition cannot be evaluated to false until the last token from the input stream has been read. □

Our solution to this problem is a *sequential* (or *lazy*) approach to query evaluation, as opposed to the *eager* approach introduced above. In sequential query evaluation, each query-variable binds to one node in the buffered XML document at-a-time. Throughout query evaluation, only one for-loop is currently responsible for producing output, so no intermediate results accumulate. We introduce this alternative mode of query evaluation, along with further techniques for reducing main memory consumption, in the next chapter.

6 Buffer-Conscious XML Stream Processing

In main memory-based XQuery engines like those modeled in the previous chapter, queries are evaluated on the buffered input tree. Buffer management in this setup is rather simple. It consists of loading the complete input prior to query evaluation, and discarding the buffer contents when query evaluation has finished. Yet this thwarts any chances at scalability, as the complete input has to fit into main memory. Moreover, we have shown how an eager approach to query evaluation, where several bindings of a query-variable are evaluated in parallel, can cause additional memory overhead.

In this chapter, we isolate key tasks in reducing the main memory footprint. Our solutions are fully composable, and selected methods will be implemented as prototype systems in the succeeding chapters. We introduce a sequential approach to query evaluation in Section 6.1 and discuss its suitability for streaming scenarios. We then evaluate queries on-the-fly, while the input is read. In particular, we discuss the evaluation of parts of the query directly on the input stream, instead of on buffered data. This results in a scheduling of event-based query operators, and is covered in Section 6.2. In this context, we point out how the preemptive purging of buffers during query evaluation can effectively reduce the main memory requirements. Preemptive buffer management is discussed in Section 6.3. We finally summarize our observations in Section 6.4.

6.1 Sequential XQuery Evaluation

In a buffer-conscious XQuery evaluation in our framework, we want to influence the order in which terms are rewritten. We motivate this idea below, and specify a systematic compilation of XQueries such that queries are evaluated sequentially. This stands in contrast to the eager evaluation from the previous chapter. We conclude this section with a discussion of the benefits of this approach in the context of XML stream processing.

Motivation. Let us consider the query from Figure 5.3(a) and the buffer-term from Figure 5.3(b). Regarding the current state of query evaluation, we assume that the query-variables \$root, \$bib, and \$b have already been bound, as shown in Figure 5.3(c). Next, the for-loops over titles and authors are evaluated. Let $q_{\$t}$ and $q_{\$a}$ be the terms for evaluating these for-loops. Rewriting these terms will produce binary trees that encode the result of evaluating the for-loops. These trees are then serialized to the output. For instance, in eager evaluation we instantiate the query-term

$$\text{output}(\ \text{concat}(q_{\$t}, q_{\$a})\ ),$$

where we evaluate both for-loops, and concatenate the results.

Note that our framework does not provide means for imposing an order in which term rewriting rules are applied. Yet the size of the query-term, and thus the total memory consumption, depend on this order. If we are "lucky", term $q_{\$t}$ is rewritten before term $q_{\$a}$, but term rewriting can also proceed the other way round. Consequently, we may obtain the query-terms

$$\text{“output(}\mathbf{title}[(\ )]q_{\$a}\text{)”} \quad \text{or} \quad \text{“output(concat(}q_{\$t},\ \mathbf{a}[(\ )]\mathbf{a}[(\ )](\ ))\text{)”}$$

as intermediary results. In the first case, the opening- and closing-tag for the title-node can be output before the term $q_{\$a}$ is rewritten. This occurs at little memory overhead. In the second case, the author-nodes must remain buffered until it is their turn for output, which is *after* the title has been output. While subtree sharing can diminish this overhead, we have already shown that there are queries where subtree sharing is of virtually no effect.

We solve this problem by enforcing a sequential evaluation of the for-loops. We apply a common approach from functional programming for enforcing lazy query evaluation in eager programming languages. We instantiate the query-term $q_{\$a}$ only after the term $q_{\$t}$ has been rewritten. In the context of XQuery evaluation, this will be the case when the *complete* binary tree produced by $q_{\$t}$ has been serialized to the output. To detect this moment, we introduce a function "$\text{delay}_{\$t}(\tau)$" that yields the nullary term "$\text{finished}_{\$t}$" once term τ has been serialized. We rewrite the query-term "$\text{output}(\text{delay}_{\$t}(q_{\$t}))$" to the term "$\text{output}{\downarrow}(\text{finished}_{\$t})$", and then apply the term rewriting rule "$q_{\$a} \leftarrow \underline{\text{finished}_{\$t}}$" to instantiate term $q_{\$a}$ for the for-loop over authors.

Auxiliary functions. We introduce several auxiliary functions for enforcing sequential query evaluation. In resuming the previous example, we begin with the query-term "$\text{output}(\text{delay}_{\$t}(q_{\$t}))$". Rewriting the term $q_{\$t}$ yields a binary tree. We place a marker "$\text{end}_{\$t}$" to the rightmost leaf-node of this tree, using the function "$\text{findend}_{\$t}$" to determine this position. Independently, the output-function traverses this tree in document order. Once it reaches the marker "$\text{end}_{\$t}$", we know for sure that the result of the for-loop over titles has been output. The marker is now replaced by the constant term "$\text{finished}_{\$t}$", upon which we instantiate $q_{\$a}$.

In Figure 6.1, we show snapshots of how the query-term changes over time, and point out when output is produced. The "$\text{findend}_{\$t}$"-function moves to the rightmost child of the title-tree, where it positions the marker "$\text{end}_{\$t}$". This marker stays put until it is met by the "output"-function. Then the query-term is replaced by the term for evaluating the for-loops over authors.

6.1.1 Enforcing Evaluation Order

We next formalize the rules for enforcing sequential evaluation. We use identifiers (or *indexes*) as subscripts in function names to allow for the sequential

$$\begin{array}{ll}
 & \text{output(delay}_{\$t}(q_{\$t})\text{)} \\
\rightsquigarrow & \text{output}{\downarrow}\text{(findend}_{\$t}(q_{\$t})\text{)} \qquad \text{– evaluate } q_{\$t} \text{ to } \mathbf{title}[\text{( )}]\text{( )} \\
\rightsquigarrow & \text{output}{\downarrow}\text{(findend}_{\$t}\text{(}\mathbf{title}[\text{( )}]\text{( )))} \\
\rightsquigarrow & \text{output}{\downarrow}\text{(}\mathbf{title}[\text{( )}]\text{ findend}_{\$t}\text{(( )))} \quad \text{– position end-marker end}_{\$t} \\
\rightsquigarrow & \text{output}{\downarrow}\text{(}\mathbf{title}[\text{( )}]\text{ end}_{\$t}\text{)} \\
\rightsquigarrow & \text{title[output}{\downarrow}\text{(( ))] end}_{\$t} \quad \text{– output token “}\langle t\rangle\text{”} \\
\rightsquigarrow & \text{title[output}{\uparrow}\text{] end}_{\$t} \qquad \text{– output token “}\langle /t\rangle\text{”} \\
\rightsquigarrow & \text{output}{\downarrow}\text{(end}_{\$t}\text{)} \\
\rightsquigarrow & \text{output}{\downarrow}\text{(finished}_{\$t}\text{)} \qquad \text{– instantiate term } q_{\$t} \\
\rightsquigarrow & \text{output}{\downarrow}\text{(} q_{\$a}\text{)}
\end{array}$$

Figure 6.1: *Snapshots of the query-term in scheduling output.*

scheduling of several terms. As the input queries are known in advance, we can assume a finite set of indexes for each XQuery. Given two terms τ_i and τ_j with indexes i and j that produce binary trees, we want to instantiate term τ_j only after all results of term τ_i have been written to the output. We further assume that the query-term has a subtree of the form "output↓(delay$_i(\tau_i)$)". To position marker "finished$_i$" as the rightmost leaf of the binary tree computed by τ_i, we define the rules below for all tagnames a.

$$\begin{array}{rcl}
\text{findend}_i(e) & \leftarrow & \underline{\text{delay}_i}(e) \\
a[e_1]\ \text{findend}_i(e_2) & \leftarrow & \underline{\text{findend}_i}(\ \underline{a}[e_1]e_2\ ) \\
\text{end}_i & \leftarrow & \text{findend}_i(\ \underline{(\ )}\ )
\end{array}$$

As the set of indexes is finite, for each permutation π of a subset $\{i_1, \ldots, i_n\}$ of indexes and some term τ, we can precompute all nestings of findend-functions around term τ of the form

$$\text{findend}_{\pi(i_1)}(\text{findend}_{\pi(i_2)}(\ldots\ \text{findend}_{\pi(i_n)}(\tau)\ \ldots)).$$

For all such nestings, we define a rule that instantiates the term τ_j once the results of term τ_i have been output. This is done in two steps. First, the end-marker is replaced by the token "finished".

$$\begin{array}{l}
\text{output}{\downarrow}\big(\ \text{findend}_{\pi(i_1)}(\ldots\ \text{findend}_{\pi(i_n)}(\text{finished}_i)\ \ldots)\big) \\
\qquad\qquad \leftarrow \underline{\text{output}{\downarrow}}\big(\ \underline{\text{findend}_{\pi(i_1)}}(\ldots\ \underline{\text{findend}_{\pi(i_n)}}(\underline{\text{end}_i})\ \ldots)\big).
\end{array}$$

Then, term τ_j is instantiated by the rule

$$\tau_j \quad \leftarrow \quad \underline{\text{finished}_i}.$$

Example 6.1 The rewriting steps of the query-term "output(delay$_{\$t}(q_{\$t})$)" in Figure 6.1 are in accordance with the rules specified above. □

6.1.2 Query Compilation

We next compile queries for sequential query evaluation. In doing so, we re-use the auxiliary functions for maintaining variable environments (see Section 5.2.1).

In the upcoming compilation, instead of maintaining several environments simultaneously, we manage a *single* variable environment as the current variable environment. This environment is stored in the buffer-term, hence the environment is no longer a parameter in query compilation. As motivated before, we assign identifiers to query subexpressions. These identifiers also function as parameters in the compilation.

Let Q be a normalized XQuery. We again implement a two-phase evaluation approach, and begin with start term "load([ ],[**root**])". We also define the rule

$$\text{output}([\![Q]\!]_1) \quad \leftarrow \underline{(\ )}$$

where Q is the given query, and index 1 denotes the identifier of this query expression. The compilation proceeds according to Figure 6.2, which contains mappings of the form $[\![q]\!]_i/\langle\tau, R\rangle$. If the current query subexpression with identifier i is of the form q, then it is replaced by the term τ, and a set of term rewriting rules is defined according to R. Again, the compilation proceeds until no more subexpressions can be rewritten.

Example 6.2 We again consider the query

```
<x>{ for $b in //book return  <y>{$b}</y> }</x>
```

from Example 5.11, where this query was compiled for eager evaluation. Now, we obtain the same start term "load([ ],[**root**])" as in eager compilation, yet different rewriting rules, as shown below.

$$
\begin{array}{l}
\text{output}(x[\text{for}_{\$b}](\ )) \quad \leftarrow \quad \underline{(\ )}, \\
\text{check\_first\_binding}_{\$b}(\ \text{exists}(\$\text{root}//\text{b}, \textbf{root})\ ) \qquad \leftarrow \quad \underline{\text{for}_{\$b}}, \\
\text{delay}_{\$b}(\ y[\text{remove\_all\_bindings}(\text{select}(\$\text{b}, \textbf{root}))](\ )\ ) \\
\qquad \leftarrow \quad \underline{\text{check\_first\_binding}_{\$b}(\underline{\text{true}})} \quad \{\textbf{root}{:=}\text{bind\_first}(\$\text{root}//\text{b}, \$\text{b}, \textbf{root})\}, \\
(\ ) \qquad \leftarrow \quad \underline{\text{check\_first\_binding}_{\$b}(\underline{\text{false}})}, \\
\text{check\_next\_binding}_{\$b}(\ \text{has\_next}(\$\text{root}//\text{b}, \$\text{b}, \textbf{root})\ ) \leftarrow \underline{\text{finished}_{\$b}}, \\
\text{delay}_{\$b}(\ y[\text{remove\_all\_bindings}(\text{select}(\$\text{b}, \textbf{root}))](\ )\ ) \\
\qquad \leftarrow \underline{\text{check\_next\_binding}_{\$b}(\underline{\text{true}})} \quad \{\textbf{root}{:=}\text{bind\_next}(\$\text{root}//\text{b}, \$\text{b}, \textbf{root})\}, \\
(\ ) \qquad \leftarrow \underline{\text{check\_next\_binding}_{\$b}(\underline{\text{false}})} \quad \{\textbf{root}{:=}\text{remove\_binding}(\$\text{b}, \textbf{root})\}.
\end{array}
$$

Figure 6.3 shows snapshots in evaluating this query on the buffer-term from Figure 5.7(a). In snapshots (a) and (b), query-variable $b has already been bound to the first book-node in document order. This node is serialized to the output. Then the variable is bound to the second book, which is also output (snapshots (c) and (d)). Finally, the variable is bound to the third book (snapshot (e)). After this book as been output, query evaluation terminates with writing the closing tag $\langle/x\rangle$ to the output. □

6.1.3 Discussion

There are several systems for XML stream processing that evaluate queries eagerly [41, 42, 50]. This is possible due to the functional nature of XQuery, which allows that all bindings for a for-loop can be evaluated in parallel.

$$\llbracket \langle a\rangle q \langle a\rangle \rrbracket_i \;/\; \langle a[\llbracket q\rrbracket_j](\,),\ \emptyset\rangle \quad \text{with new identifier } j,$$

$$\llbracket (\,) \rrbracket_i \;/\; \langle (\,),\ \emptyset\rangle$$

$$\llbracket \$x \rrbracket_i \;/\; \langle \text{remove\_all\_bindings}(\text{select}(\$x, \mathbf{root})),\ \emptyset\rangle$$

$$\begin{array}{l}
\llbracket (q_1, \ldots, q_n) \rrbracket_i \;/ \\
\quad \langle \text{delay}_{i_{n-1}}(\text{delay}_{i_{n-2}}(\ \ldots \text{delay}_{i_1}(\ \llbracket q_1\rrbracket_{i_n}\ ) \ldots\ )), \\
\qquad \{\ \begin{array}{lcl}
\llbracket q_2\rrbracket_{i_{n+1}} & \leftarrow & \underline{\text{finished}_{i_1}}, \\
\llbracket q_3\rrbracket_{i_{n+2}} & \leftarrow & \underline{\text{finished}_{i_2}}, \\
\ldots & & \\
\llbracket q_{j+1}\rrbracket_{i_{n+j}} & \leftarrow & \underline{\text{finished}_{i_j}}, \\
\ldots & & \\
\llbracket q_n\rrbracket_{i_{2n-i}} & \leftarrow & \underline{\text{finished}_{i_{n-1}}}\ \}\rangle
\end{array} \\
\qquad \text{with new identifiers } i_1, \ldots, i_n, i_{n+1}, \ldots, i_{2n-1},
\end{array}$$

$$\begin{array}{l}
\llbracket \text{for } \$x \text{ in } \$y/\pi \text{ return } q \rrbracket_i \;/ \\
\quad \langle \text{for}_{\$x},\ \{\text{check\_first\_binding}_{\$x}(\text{exists}(\$y/\pi, \mathbf{root})) \leftarrow \underline{\text{for}_{\$x}}, \\
\qquad\qquad \begin{array}{lcl}
\text{delay}_i(\llbracket q\rrbracket_j) & \leftarrow & \underline{\text{check\_first\_binding}_{\$x}}(\underline{\text{true}}) \\
 & & \{\ \mathbf{root}{:=}\text{bind\_first}(\$y/\pi, \$x, \mathbf{root})\ \}, \\
(\,) & \leftarrow & \underline{\text{check\_first\_binding}_{\$x}}(\underline{\text{false}}), \\
\multicolumn{3}{l}{\text{check\_next\_binding}_{\$x}(\text{has\_next}(\$y/\pi, \$x, \mathbf{root})) \leftarrow \underline{\text{finished}_i},} \\
\text{delay}_i(\llbracket q\rrbracket_j) & \leftarrow & \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{true}}) \\
 & & \{\ \mathbf{root}{:=}\text{bind\_next}(\$y/\pi, \$x, \mathbf{root})\ \}, \\
(\,) & \leftarrow & \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{false}}) \\
 & & \{\ \mathbf{root}{:=}\text{remove\_binding}(\$x, \mathbf{root})\ \}\ \}\rangle
\end{array} \\
\quad \text{with new identifier } j
\end{array}$$

$$\begin{array}{l}
\llbracket \text{if } \chi \text{ then } \alpha \text{ else } \beta \rrbracket_i \;/ \\
\quad \langle \text{check\_condition}_i(\llbracket \chi\rrbracket_{j_\chi}), \\
\qquad \{\ \llbracket \alpha\rrbracket_{j_\alpha} \leftarrow \underline{\text{check\_condition}_i}(\underline{\text{true}}), \llbracket \beta\rrbracket_{j_\beta} \leftarrow \underline{\text{check\_condition}_i}(\underline{\text{false}})\ \}\rangle \\
\quad \text{with new identifiers } j_\chi,\ j_\alpha, \text{ and } j_\beta
\end{array}$$

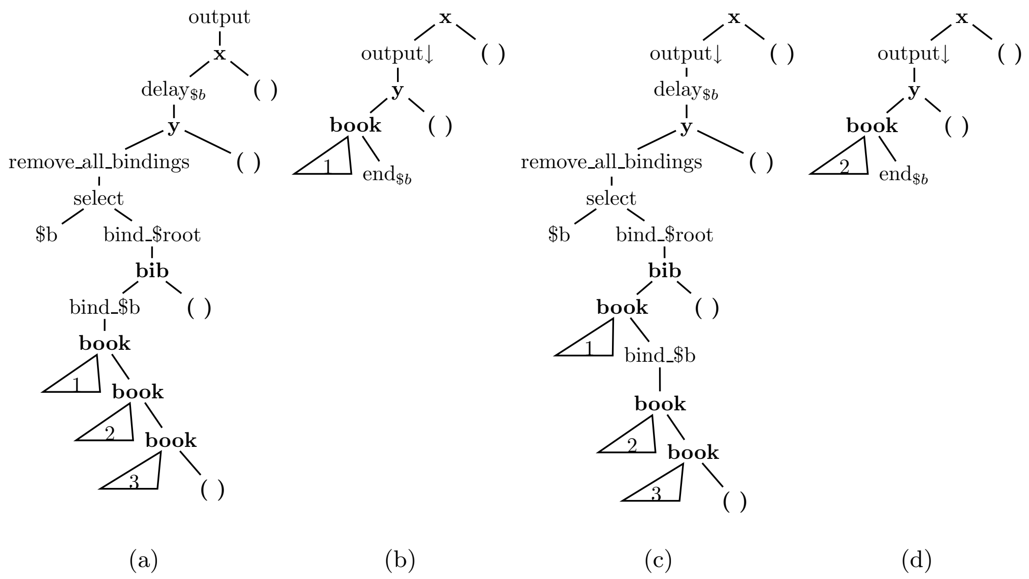
$$\begin{array}{l}
\llbracket \text{some } \$x \text{ in } \$y/\pi \text{ satisfies } \chi \rrbracket_i \;/ \\
\quad \langle \text{some}_{\$x},\ \{\text{check\_first\_binding}_{\$x}(\text{exists}(\$y/\pi, \mathbf{root})) \leftarrow \underline{\text{some}_{\$x}}, \\
\qquad\qquad \begin{array}{lcl}
\text{check\_condition}_{\$x}(\llbracket \chi\rrbracket_j) & \leftarrow & \underline{\text{check\_first\_binding}_{\$x}}(\underline{\text{true}}) \\
 & & \{\ \mathbf{root}{:=}\text{bind\_first}(\$y/\pi, \$x, \mathbf{root})\ \}, \\
\text{true} & \leftarrow & \underline{\text{check\_condition}_{\$x}}(\underline{\text{true}}), \\
\multicolumn{3}{l}{\text{check\_next\_binding}_{\$x}(\text{has\_next}(\$y/\pi, \$x, \mathbf{root})) \leftarrow \underline{\text{check\_condition}_{\$x}}(\underline{\text{false}}),} \\
\text{check\_condition}_{\$x}(\llbracket \chi\rrbracket_j) & \leftarrow & \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{true}}) \\
 & & \{\ \mathbf{root}{:=}\text{bind\_next}(\$y/\pi, \$x, \mathbf{root})\ \}, \\
\text{false} & \leftarrow & \underline{\text{check\_next\_binding}_{\$x}}(\underline{\text{false}}) \\
 & & \{\ \mathbf{root}{:=}\text{remove\_binding}(\$x, \mathbf{root})\ \}\ \}\rangle
\end{array} \\
\quad \text{with new identifier } j
\end{array}$$

$$\llbracket \$x = \$y \rrbracket_i \;/ \langle \text{compare}(\$x, \$y, \mathbf{root}),\ \emptyset\rangle$$

$$\llbracket \text{true}() \rrbracket_i \;/ \langle \text{true},\ \emptyset\rangle$$

$$\llbracket \text{false}() \rrbracket_i \;/ \langle \text{false},\ \emptyset\rangle$$

Figure 6.2: *XQuery compilation rules $\llbracket q\rrbracket_i/\langle \tau, R\rangle$ for sequential evaluation.*



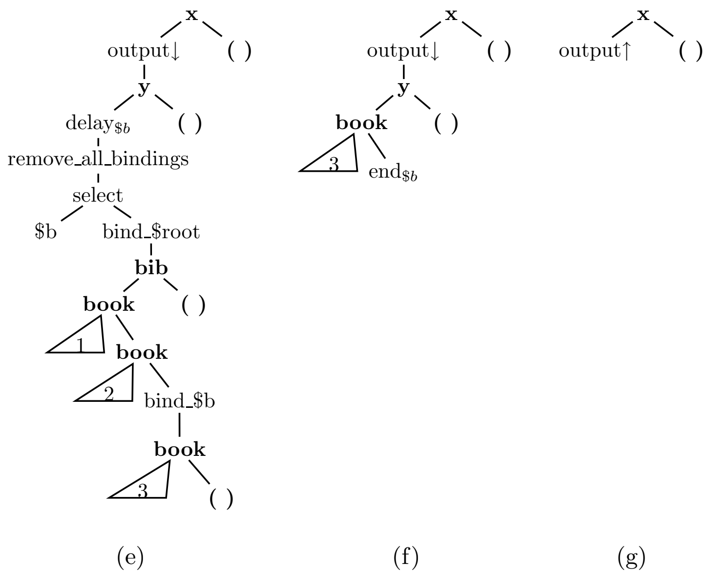


Figure 6.3: *Snapshots of query- and buffer-terms (Example 6.2).*

The trade-off between *lazy* and *eager* evaluation is a fully discussed subject in the field of programming languages [55]. Regarding runtime, the same arguments apply here, namely that lazy (or sequential) evaluation avoids the computation of intermediate results that may not be required after all. In the context of XML stream processing, our main concern is the memory required for storing the intermediate results produced by eager evaluation.

With sequential query evaluation, we can actually guarantee for our query fragment XQ that no intermediate query results need to be buffered for producing output in the correct order. Thus, there is the same worst-case high watermark for all queries over a given input document, and data is not buffered redundantly. In contrast, we cannot give such guarantees in eager evaluation, where we may have to buffer intermediate results as well. In the eager XML stream processing system of [50], garbage collection is combined with subtree sharing to reduce the main memory overhead, and to avoid redundant buffering. However, there are queries where little can be gained by these techniques, as shown in Example 5.13.

In summary, when main memory consumption is our prime optimization target, as is the case in XML stream processing, a sequential query evaluation is to be preferred. Note that in our discussions of runtime aspects, in particular the computation of joins in Part IV, we will relativize this point to some extent.

6.2 Streaming XQuery Evaluation

For the sake of scalability, we deviate from the two-phase evaluation model presented so far, where query evaluation only sets in after the complete input has been read. In this section, we discuss the sequential evaluation of XQueries, or parts thereof, *on-the-fly*, while the input stream is being read.

In the following, we present two approaches. In the first, we statically schedule dedicated streaming query operators. In the second, we do not have dedicated operators available, but query evaluation is interleaved with reading the input and the purging of buffers. The overall effect is also an on-the-fly query evaluation. We introduce these two approaches in Sections 6.2.1 and 6.2.2 respectively. In Section 6.2.3, we discuss the tradeoffs involved. We emphasize the necessity of purging buffers preemptively, which is the topic of the next section.

Motivation. There are several main memory-based XQuery processors with streaming operators in their query algebras (e.g. [44, 70, 101]). These operators are event-based, and are evaluated directly over the events in the input stream, rather than over buffer datastructures. The FluXQuery engine, which we will also present in Chapter 8, is a forerunner in this field. Below, we outline the basic idea using an example query. Then we sketch how the FluXQuery engine proceeds for this query, using our abstract framework as a modeling language.

Example 6.3 Consider the XQuery below

```
<results>{
   for $bib in /bib return
     for $b in $bib/book
     return  <book>{ ($b/title, $b/author, $b/year) }</book> }
</results>
```

which outputs the title, author, and year children of each book node, in this order. In two-phase XQuery evaluation, the for-loop over book nodes is evaluated over buffered data. When streaming operators are available, we can design an alternative query execution plan that processes one book at-a-time, while the input is read. Then each opening tag $\langle book \rangle$ that is part of a node reachable via the path `/bib/book` is directly written to the output. All tokens that are part of book titles are output the instance that they appear in the input stream. In the meantime, the author nodes and year nodes of books are buffered. When the tag $\langle /book \rangle$ is encountered, the query expressions "`$b/author`" and "`$b/year`" are evaluated over the buffered data. The buffer contents can now be purged, as they are no longer relevant for the remaining query evaluation. At this point, the closing tag for the book-node is written to the output.

Thus, the titles need not be buffered at all. At each moment during query evaluation, at most the author and year children of a single book are stored in buffers. In contrast, in evaluating the complete query in a two-phase approach, the titles, author, and year children of *all* book nodes are buffered. □

6.2.1 Static Scheduling of Streaming Operators

To the best of our knowledge, in all existing XQuery engines with streaming operators, operator scheduling takes place at query compile time [44, 70] (though [101] discusses the potential of operator scheduling at runtime). We refer to the operators that process buffered data as *buffer* operators.

For static operator scheduling, we need a formalism that makes the distinction between streaming and buffer operators explicit. In the Galax XQuery engine [44], a physical query algebra for full XQuery with streaming operators is used. Our FluX approach is more general as it does not specify *how* query evaluation over buffers is realized. Rather, it identifies *which* parts of the query can be directly evaluated over the input stream, and *which* must be evaluated over buffers. In the following, we present ideas from the FluX approach.

The FluX approach. The FluX language extends our XQuery fragment XQ^- with the *process-stream* construct, which specifies the parts of the query that are evaluated directly on the input stream. We introduce the FluX language on an intuitive level with the following example.

Example. The query from Example 6.3 is phrased as a FluX query in Figure 6.4. A "process-stream $\$x$" expression in FluX consists of a number of *handlers* which process the children of the XML node bound by variable $\$x$ from left to right. In sequential query evaluation, query variable $\$x$ binds to one node at-a-time, which we refer to as the *context node.* Let us consider the list of handlers defined in Figure 6.4 for the context node represented by variable \$root. The handler "on-first-past()" fires when the context of the root variable is first entered. So in the example above, the opening tag $\langle results \rangle$ is output before any input data has been read. Equivalently, the handler "on-first-past(*)" fires when the context of the variable is left, so the closing tag $\langle results \rangle$ is output at the end of the stream.

An "on a"-handler fires for each child of the context node that is labeled a. When the handler fires, its associated query expression is executed. Here, the

```
{ process-stream $root:
     on-first-past() return <results>,
     on bib as $bib return
        { process-stream $bib:
             on book as $b return
              {  process-streams $b:
                    on-first-past() return <book>,
                    on title as $title return $title,
                    on-first-past(*) return $b/author,
                    on-first-past(*) return $b/year,
                    on-first-past(*) return </book> } }
     on-first-past(*) return </results> }
```

Figure 6.4: *A FluX expression.*

"on bib"-handler fires when the opening tag for a bib-node is read. Then this node becomes the context node for variable $bib. Next, we check which of the the handlers defined for the "process-stream $bib"-expression fire.

All children of the context node that are not labeled "book" are simply ignored. When the opening tag for a book-node is read, the "on book"-handler fires. The book-node becomes the context node of variable $b. The interplay of event-handlers for the $b-variable can be summarized as follows. Up front, the opening tag ⟨*book*⟩ is output. Each title is output directly when read in the input. Once the context of the variable $b is left, i.e. the closing tag ⟨/*book*⟩ is read in the input stream, the "on-first-past(*)" event handlers fire. The handlers fire in order of their specification, and their subexpressions are evaluated over buffered data. So first the author children are retrieved from the buffer and output, then the year children. It is the task of the buffer manager to gather this data while the book-node is being parsed. The FluX language does not make these details of buffer management explicit, as they are really an orthogonal issue. However, the compilation of queries into FluX does ensure that the buffer manager has enough time to fill buffers before their contents are accessed in query evaluation. In Chapter 8, where we present the details on FluX, we formulate this as a *safety* requirement. Finally, when the third "on-first-past(*)"-handler is executed, the closing tag ⟨/*book*⟩ is output.

FluX expressions in the abstract framework. The formal syntax and semantics of the full FluX language are specified in Chapter 8. For now, we argue on the level of intuition, and show how a subset of FluX expressions can be modeled in our framework. Note that we disregard all issues of buffer management for now, such as loading data into buffers or purging data from buffers. The following section will fill in these gaps.

The idea is to represent a FluX expression as a query-term, and to pass the XML events from the input stream from one query subexpression to the next. In particular, the stream is passed from event-handler to event-handler, where each handler can fire in turn. We first introduce two auxiliary functions for processing single XML events, before we encode process-stream statements.

Auxiliary macros. The auxiliary functions specified in Figure 6.5 process single XML events. When the input is read, the single events from the input will be fed to these functions. Macro "skipnode" skips a subtree, while macro "outputnode" copies a subtree to the output. Both functions use a stack to determine the scopes of subtrees. Again, we implement stacks as lists and use the term "□" to signal that an input token has been consumed (see Section 4.3).

For all tagnames a and b:

$$\begin{array}{rcll}
\text{skip}(\square, [a]) & \leftarrow & \underline{\text{skipnode}}(\underline{\langle a\rangle}) & \\
\text{skip}(\square, a :: S) & \leftarrow & \underline{\text{skip}}(\underline{\langle a\rangle}, S) & \\
\text{skip}(\square, x :: S) & \leftarrow & \underline{\text{skip}}(\underline{\langle /a\rangle}, \underline{a} :: \underline{b} :: S) & \\
\square & \leftarrow & \text{skip}(\underline{\langle /a\rangle}, \underline{a} :: \underline{[\,]}) & \\
\text{out}(\square, [a]) & \leftarrow & \underline{\text{outputnode}}(\underline{\langle a\rangle}) & \{\text{write } \langle a\rangle\} \\
\text{out}(\square, a :: S) & \leftarrow & \underline{\text{out}}(\underline{\langle a\rangle}, S) & \{\text{write } \langle a\rangle\} \\
\text{out}(\square, x :: S) & \leftarrow & \underline{\text{out}}(\underline{\langle /a\rangle}, \underline{a} :: \underline{b} :: S) & \{\text{write } \langle /a\rangle\} \\
\square & \leftarrow & \underline{\text{out}}(\underline{\langle /a\rangle}, \underline{a} :: \underline{[\,]}) & \{\text{write } \langle /a\rangle\}
\end{array}$$

***Figure 6.5**: Stream processing macros.*

Rewriting rules. For the sake of simplicity, we assume that the input stream encodes a non-recursive XML document. This saves us some technical overhead. In the compilation of a FluX expression f into our framework, we use the notation $[\![f]\!]^{\mathit{flux}}$. Then a FluX expression of the form

$$[\![\text{ on } a \text{ as } \$y \text{ return } \{ \text{ process-stream } \$y\text{: } \mathit{handler}_1, \ldots, \mathit{handler}_n \text{ } \} \text{ }]\!]^{\mathit{flux}}$$

is compiled into a query-term as shown below,

$$\begin{tikzcd}
 & \text{on\_}a\text{\_as\_}\$y\text{\_return} \arrow[d, dash] & \\
 & \text{process-stream\_}\$y \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
{[\![\mathit{handler}_1]\!]^{\mathit{flux}}} & \ldots & {[\![\mathit{handler}_n]\!]^{\mathit{flux}}}
\end{tikzcd}$$

where the event handlers $\mathit{handler}_1, \ldots, \mathit{handler}_n$ are compiled as follows.

- An event handler of the form $[\![$on-first-past() return $\alpha]\!]^{\mathit{flux}}$ is translated into a term "on-first-past()($[\![\alpha]\!]^{\mathit{flux}}$)".
- An event handler of the form $[\![$on-first-past(*) return $\alpha]\!]^{\mathit{flux}}$ is translated into a term "on-first-past(*)($[\![\alpha]\!]^{\mathit{flux}}$)".
- An event handler of the form "on b as $\$z$ return $\$z$", is translated into a term "on_b_as_$z_return(return_$z)".
- An event handler of the form "on b as $\$z$ return {process-stream $\$z$: h}" is rewritten recursively.

Each event handler subexpressions α is also rewritten. If α is a sequence of opening and closing tags, then we substitute it with the term "emit_α". Otherwise, if α is an XQuery expression, we replace it by a term "expr_α".

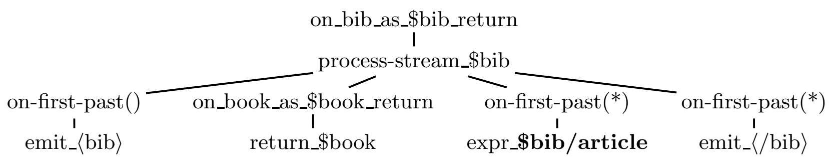


***Figure 6.6**: A query-term encoding a FluX expression (Example 6.4).*

Example 6.4 We now resort to a smaller example query, for which we discuss the compilation in detail. The XQuery sub-expression

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

first retrieves books, then articles. Its FluX version

```
on bib as $bib return
  { process-stream $bib:
      on-first-past() return <bib>,
      on book as $book return $book,
      on-first-past(*) return $bib/article,
      on-first-past(*) return </bib> }
```

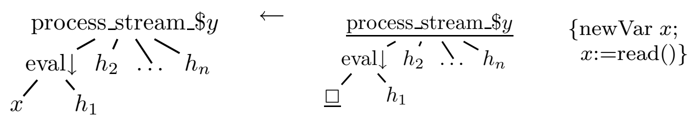
specifies that books are output the instance they are read in the input stream. Once the complete input has been read, articles are retrieved from the buffer. We represent this FluX expression as the query-term from Figure 6.6. □

Processing the input stream. We next encode the eval-function. The eval-function reads events from the input stream and passes them along to the event handlers. It takes two arguments, an input token and the currently active query expression. The eval-function traverses the encoded FluX query in depth-first order, using functions "eval↑" and "eval↓".

Tokens are read from the input by the rules from Figure 6.7. The eval-function takes care of evaluating the event handlers in sequence of their specification (see Figure 6.8). For each child of the context node, the event handlers are checked one by one. The rules for evaluating handlers are shown in Figures 6.9 and 6.10, and are discussed further below. The "on-first"-handlers fire only once for each context node. After they have fired, they are rewritten to the empty term "( )". If no event handler fires, then the current child node is skipped entirely, using the auxiliary function "skipnode". Finally, when the closing tag of the context node is read, the handlers are checked one more time, so that "on-first-past(*)"-handlers can fire. Then, the stream is passed "upwards" to the context of the parent variable. At this point, all handlers (which may have changed when they fired) are restored. This is done by the last rule in Figure 6.8, and ensures that the set of event handlers is complete when query-variable $\$x$ is bound anew.

We now look closer at the handler subexpression when an event handler actually fires. In the rule for query-term "eval↓(e, emit_$t_1 \dots t_n$)" in Figure 6.9, the expression $t_1 \dots t_n$ represents a sequence of opening and closing tags. When the attached event handler fires, these tokens are written to the output tape.

If the query-term for an event handler is of the form "eval↓(e, expr_q)", then XQuery expression q is evaluated when the event hander fires. In particular, this



$$\text{eval}\!\downarrow\!(\text{out}(x, S), e) \quad \leftarrow \quad \underline{\text{eval}\!\downarrow}(\underline{\text{out}}(\underline{\Box}, S), e) \qquad \{\text{ newVar } x;\ x{:=}\text{read}()\ \}$$
$$\text{eval}\!\downarrow\!(\text{skip}(x, S), e) \quad \leftarrow \quad \underline{\text{eval}\!\downarrow}(\underline{\text{skip}}(\underline{\Box}, S), e) \qquad \{\text{ newVar } x;\ x{:=}\text{read}()\ \}$$

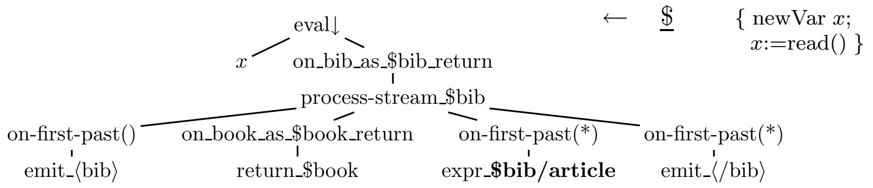
Figure 6.7*: Term rewriting rules for reading the input in FluX evaluation.*

expression is evaluated over the buffered data. This is modeled by compiling the XQuery expression into our framework. Here, we can choose between the compilation for eager or sequential query evaluation. This decision is orthogonal to the choice of *which* parts of the query are evaluated over buffered data. To keep memory consumption low, we pursue the sequential mode of query evaluation in the following. The "eval" function stalls until the XQuery expression has been evaluated, i.e. it has been reduced to the term "output↑". This is captured by the last two rules in Figure 6.9.

In Figure 6.9, we further show the treatment of "on-a"-event handlers. If these handlers require that the current context node is output (encoded by term "return_$y"), then this is done on-the-fly using the function "outputnode". This function, defined in Figure 6.5, directly copies the XML events encoding the current context node from the input to the output. The eval-function won't proceed unless the complete node has been serialized. This ensures that output is generated in a deterministic fashion. Moreover, the closing tag of the node is reinserted into the input stream, so that superordinate query expressions receive the information that the current node has already been consumed. In case an "on-a"-event handler carries a "process-stream"-statement as a subexpression, it is rewritten recursively.

The rules in Figure 6.10 concern the treatment of "on-first-past()" and "on-first-past(*)"-event handlers. The former fire for the opening-tag of the context node, the latter for the closing tag. As these event handlers fire only once for each context node, their query-terms are rewritten to the empty term.

Example 6.5 We consider the evaluation of the FluX expression from Example 6.4 for the input stream "*⟨bib⟩⟨book⟩⟨/book⟩⟨article⟩⟨/article⟩⟨/bib⟩*". We begin with the start-term "$" and define the term rewriting rule which initiates evaluation of the term from Figure 6.6.



In Figures 6.11 and 6.12, we show snapshots in evaluating the XML document from above. Snapshot (a) depicts the situation just after the start-term "$"

For a given FluX expression "on a as $\$y$ return {process-stream $\$y$: $h_1^{\$y}, \ldots, h_n^{\$y}$}", and distinct tagnames a and b:

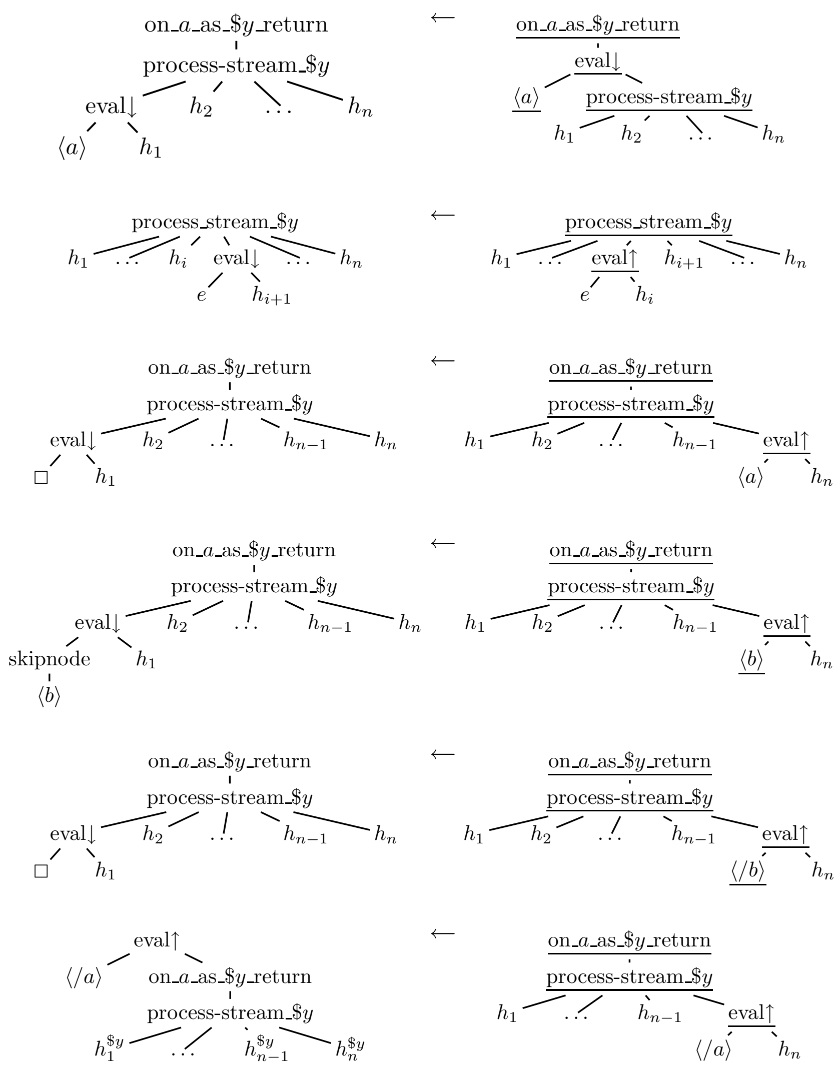


***Figure 6.8**: Evaluating FluX "process-stream"-statements.*

For distinct tagnames a and b, query-variables $\$y$, XML events $t_1, \ldots, t_n$, XQuery expression q and a new identifier j:

$$\begin{array}{rcl}
\text{on\_}a\text{\_as\_\$}y\text{\_return}(\ \text{eval}{\downarrow}(\ \langle a\rangle, e)\ ) & \leftarrow & \underline{\text{eval}{\downarrow}}(\ \underline{\langle a\rangle},\ \underline{\text{on\_}a\text{\_as\_\$}y\text{\_return}}(e)\ ) \\
\text{on\_}a\text{\_as\_\$}y\text{\_return}(\ \text{eval}{\downarrow}(\langle /a\rangle, e)\ ) & \leftarrow & \underline{\text{eval}{\downarrow}}(\ \underline{\langle /a\rangle},\ \underline{\text{on\_}a\text{\_as\_\$}y\text{\_return}}(e)\ ) \\
\text{eval}{\uparrow}(\ \langle b\rangle,\ \text{on\_}a\text{\_as\_\$}y\text{\_return}(e)\ ) & \leftarrow & \underline{\text{eval}{\downarrow}}(\ \underline{\langle b\rangle},\ \underline{\text{on\_}a\text{\_as\_\$}y\text{\_return}}(e)\ ) \\
\text{eval}{\uparrow}(\ \langle /b\rangle,\ \text{on\_}a\text{\_as\_\$}y\text{\_return}(e)\ ) & \leftarrow & \underline{\text{eval}{\downarrow}}(\ \underline{\langle /b\rangle},\ \underline{\text{on\_}a\text{\_as\_\$}y\text{\_return}}(e)\ ) \\
\\
\text{on\_}a\text{\_as\_\$}y\text{\_return}(& & \\
\text{eval}{\downarrow}(\text{outputnode}(e), \text{return\_\$}y)\ ) & \leftarrow & \underline{\text{on\_}a\text{\_as\_\$}y\text{\_return}}(\ \underline{\text{eval}{\downarrow}}(\ e, \underline{\text{return\_\$}y})\ ) \\
\text{eval}{\uparrow}(\ \langle /a\rangle, & & \\
\text{on\_}a\text{\_as\_\$}y\text{\_return}(\text{return\_\$}y)\ ) & \leftarrow & \underline{\text{on\_}a\text{\_as\_\$}y\text{\_return}}(\ \underline{\text{eval}{\downarrow}}(\underline{\square}, \underline{\text{return\_\$}y})\ ) \\
\\
\text{eval}{\uparrow}(E,\ \text{emit\_}t_1 \ldots t_n) & \leftarrow & \underline{\text{eval}{\downarrow}}(\ e,\ \underline{\text{emit\_}t_1 \ldots t_n}\ ) \\
 & & \{\text{write } t_1;\ \ldots \text{write } t_n\} \\
\text{eval}{\downarrow}(\ e,\ \text{output}([\![q]\!]_j)\ ) & \leftarrow & \underline{\text{eval}{\downarrow}}(\ e,\ \underline{\text{expr\_}q}\ ) \\
\text{eval}{\uparrow}(\ e,\ (\ )\ ) & \leftarrow & \underline{\text{eval}{\downarrow}}(\ e,\ \underline{\text{output}{\uparrow}}\ )
\end{array}$$

Figure 6.9: *FluX evaluation rules.*

has been replaced according to the rewriting rule above. In the same step, the first token is read from the input stream.

In snapshot (b), we start evaluating the handlers for the FluX statement "process-stream \$bib". Here, the "on-first-past()"-statement can be evaluated. The terms for these statements are rewritten to the empty term, so that they won't fire within the same context node anymore. Also, the token $\langle bib\rangle$ is output. Only the first handler fires for this input token, and so the eval-function transports the input token $\langle bib\rangle$ to the last handler (snapshots (c) and (d)).

Next, the input token $\langle book\rangle$ is passed along the event handlers. The second handler fires (see snapshots (e) through (g)), and the entire book-node is directly output as read in the input stream.

The next input token is the opening tag for the article-node. No event handler fires, and we reach the final event handler, as depicted in snapshot (h). This node must be discarded, so that the next child of the bib-node can be processed. This job is done by the function "skipnode", as depicted in snapshot (i). The bib-node has no more children, and so the next input token is the closing tag of the bib-node itself (snapshot (j)). This token is passed to the handlers, so that the "on-first-past(*)"-handlers may fire. In snapshots (k) and (l), the article-nodes are output. When the last handler in this list fires, the closing tag $\langle /bib\rangle$ is written to the output.

In snapshots (m) and (n), we finish the evaluation of the current context node. Note that in the final snapshot, the handlers are restored so that they can fire properly should there be a further context node. □

For a given FluX expression “on a as \$$y$ return {process-stream \$$y$: $h_1^{\$y}$, …, $h_n^{\$y}$}”, and distinct tagnames a and b:

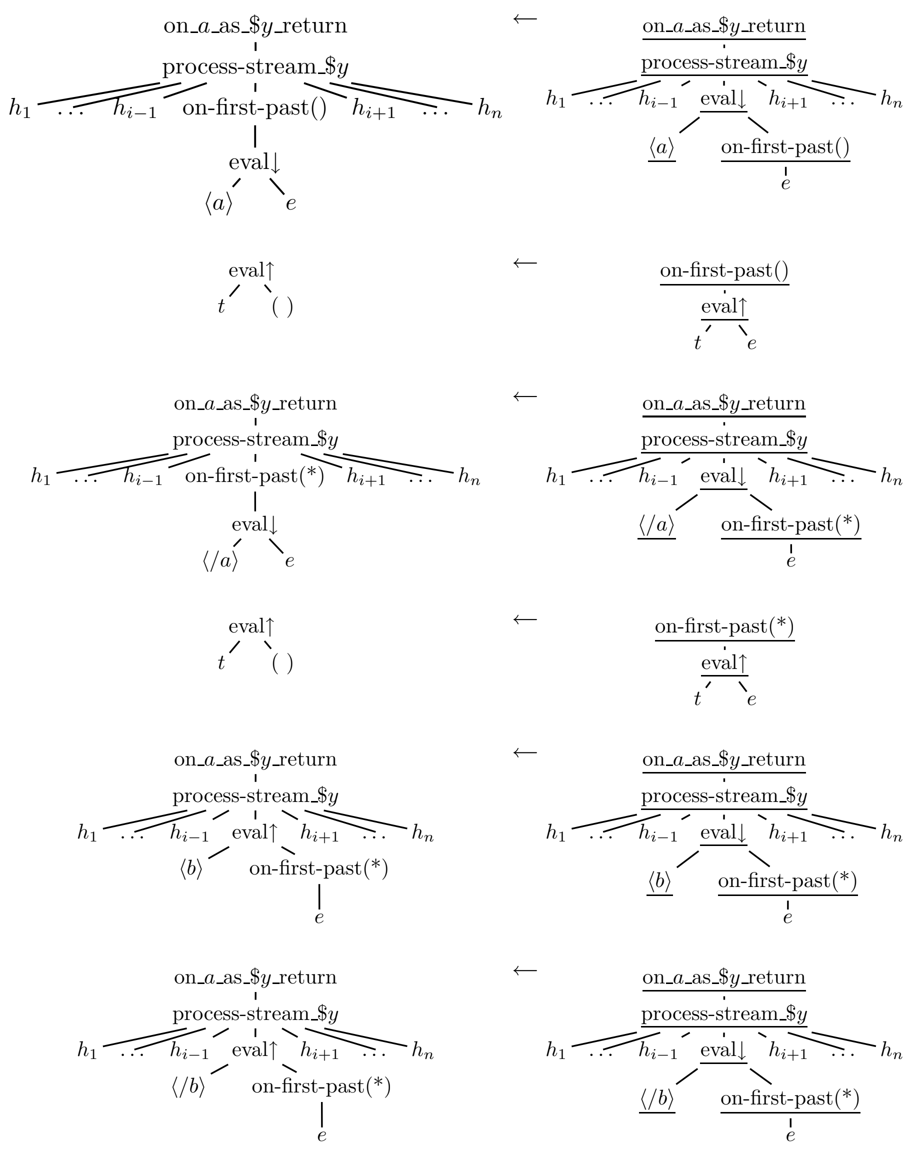


***Figure 6.10**: Evaluating FluX “on-first-past”-statements.*

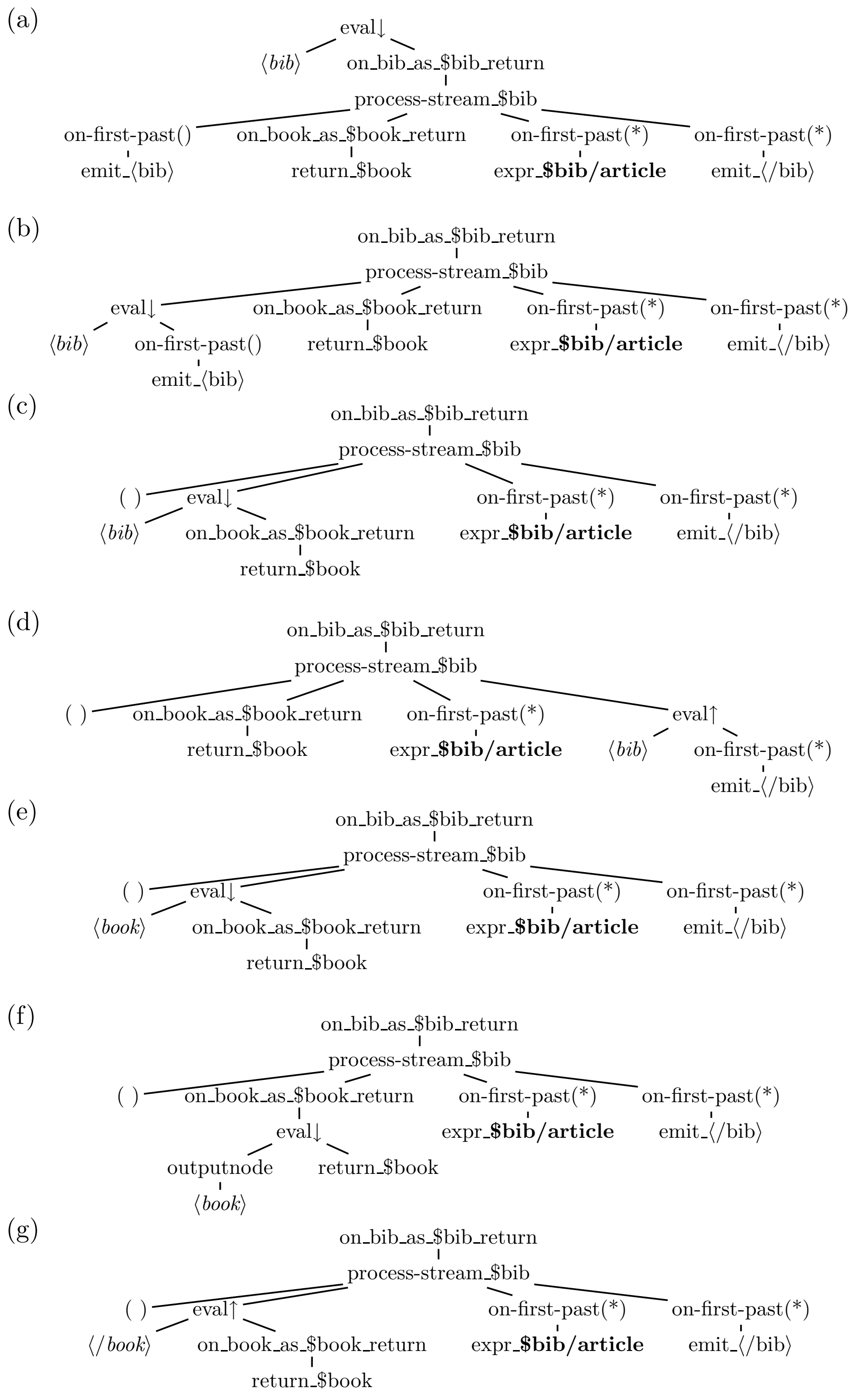


***Figure 6.11**: Snapshots of query-terms from Example 6.5.*

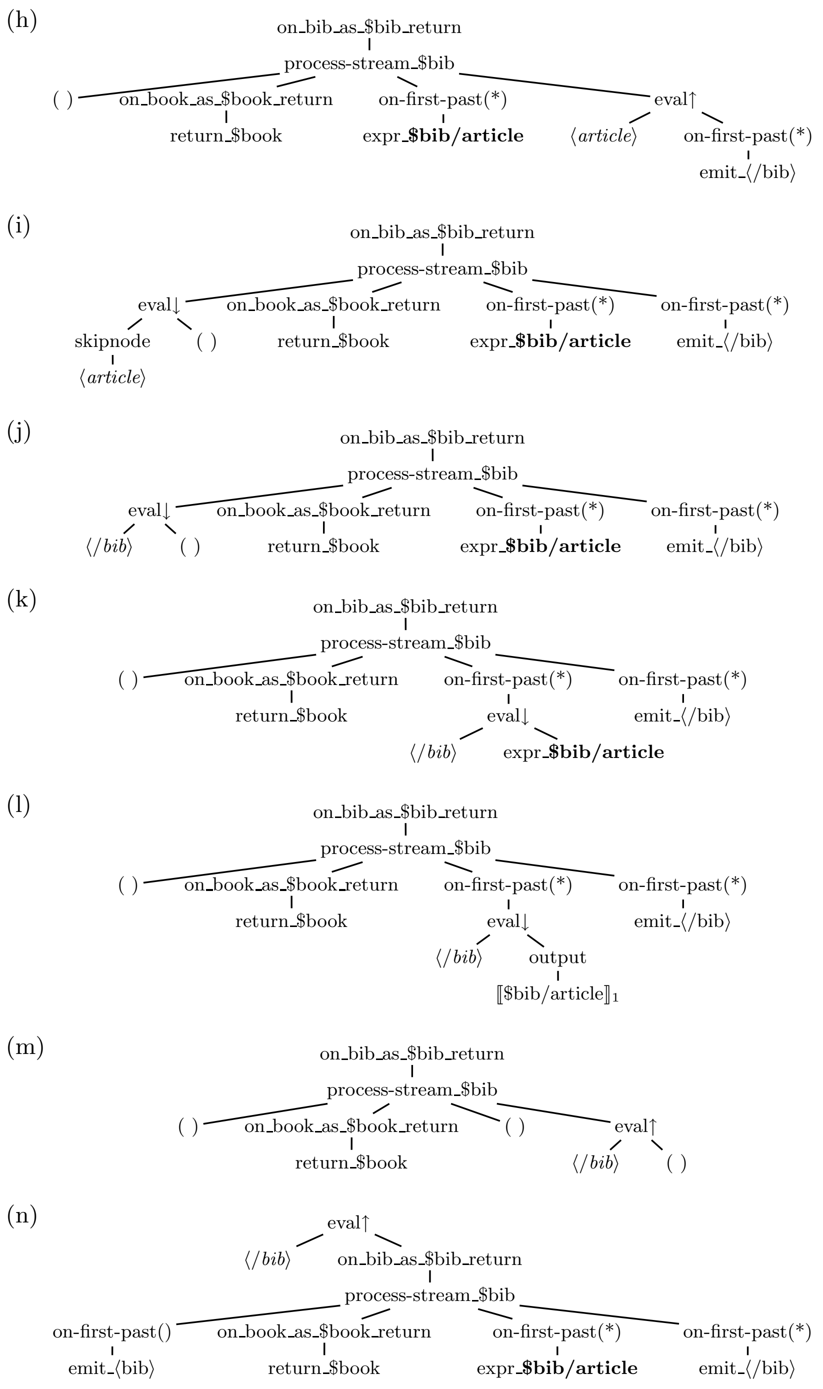


***Figure 6.12**: Snapshots of query-terms from Example 6.5 (continued).*

```
{ process-stream $root:
     on-first-past() return <results>,
     on bib as $bib return
        { process-stream $bib:
            on book as $book return
               process-stream $book:
                 on-first-past(*) return
                     if exists($book/price)
                     then <book>{ $b/title, $b/price }</book>
                     else () }
     on-first-past(*) return </results> }
```

Figure 6.13: *A FluX query.*

6.2.2 Incremental Query Evaluation over Buffered Data

Given a sequential XQuery processor as specified in Section 6.1, we can achieve a streaming query evaluation without introducing dedicated streaming operators. To this end, we evaluate the query on the *partially* buffered input tree. One input token after the other is loaded, and query evaluation proceeds as far as possible in each step. A preemptive buffer manager purges nodes from the buffer that are no longer required. The effect is an interleaving of loading and query evaluation; while the query is completely evaluated over buffered data, the buffered tree is continuously loaded and pruned.

For the query from Example 6.3, we achieve that all book titles are only buffered one node at-a-time. That is, a title is buffered, output, and purged from the buffer. Then the next title, if it exists, is processed in this manner. At most, one book title, the author, and the year children of the current book are buffered during query evaluation. The memory footprint of this approach is sightly higher than in static operator scheduling for this query (see Example 6.3), where the title nodes can be output without first buffering them.

There are also queries where this approach requires less buffering than systems that rely on static operator scheduling, as we show next.

Example 6.6 Consider the XQuery below which outputs the title and price information of all books that have prices. The static compilation into FluX yields the expression from Figure 6.13.[1]

```
<results>
{ for $bib in /bib return
    for $b in $bib/book
    where exists($b/price)
     return <book>{ ($b/title, $b/price) }</book> }
</results>
```

In evaluating this query, we wait for an opening-tag of a book to appear in the input stream. Afterwards, we delay processing the children of the book-node until closing tag $\langle/book\rangle$ has been read. This leaves enough time to buffer all title

[1] We have simplified the syntax of the FluX language for better legibility. This FluX query differs in minor details from the result of the compilation algorithm from Chapter 8.

and price nodes. In fact, a book may have a single price node as its last child, or no price at all. From the viewpoint of main memory consumption, this is the worst-case scenario, as all title nodes must be buffered until the complete book has been read. However, in the best-case scenario, the first child of a book node is a price node. In principle, the title children can be output directly, and only the price nodes need to be buffered. The incremental query evaluation over the buffered data can exploit the best-case scenario. Yet static operator scheduling cannot react to this scenario without any additional knowledge about the input, e.g. in form of schema information. □

6.2.3 Discussion

In the previous chapter, we have built XQuery engines that operate in a two-phase approach of loading the input prior to query evaluation. In this section, we have introduced two streaming variants that lower main memory consumption by abandoning the two-phase principle when possible.

The first approach statically schedules dedicated streaming query operators. That is, parts of the query are evaluated directly on the input stream, without intermediate buffering. The remaining parts of the query are evaluated over buffered data, and the choice of how to implement this is not restricted any further. For instance, both eager and sequential evaluation are possible. As an example of static scheduling, we have introduced the FluX approach, and modeled the evaluation of FluX queries in our framework. We will present FluX in greater detail in Chapter 8, where we also show how schema knowledge can be exploited in the compilation of FluX queries.

The second approach does not require dedicated streaming operators. The queries are evaluated over the buffered data, which is incrementally loaded and pruned. Again, both the eager and the sequential query evaluation modes are an option, and systems adhering to either have been implemented for XML stream processing (e.g. [97] and [41, 42, 50] respectively).

In both cases, the success in reducing the main memory consumption depends on the effectiveness of an algorithm that purges data from buffers once it is no longer relevant. The following section is dedicated to this topic.

6.3 Purging Main Memory Buffers

The decision when to purge data from buffers can be made statically or dynamically. The way in which this is realized has an effect on whether data is buffered redundantly, i.e. the same node is buffered multiple times. We introduce a technique for static buffer purging in Section 6.3.1, and an approach where buffers are purged based on static *and* dynamic analysis in Section 6.3.2. In Section 6.3.3, we discuss both approaches regarding their effectiveness in reducing main memory consumption and the effort for implementing them.

6.3.1 Static Buffer Purging

In a very basic form of buffer management, the buffer contents are not cleared before query evaluation has finished. Yet XQuery engines cannot scale to streams unless buffers are purged *preemptively*, i.e. during query evaluation.

Static buffer purging in the FluX approach. We now consider static preemptive buffer purging, and base our discussion on the FluX approach, as introduced in the previous section. We refer to the variables defined in "on a as $x"-expressions as *FluX*-variables, and all other variables as *XQuery*-variables. During query evaluation, each FluX-variable binds to one node from the input at-a-time, its *context node.* With each FluX-variable, we associate a separate buffer. The lifetime of this buffer is aligned with the scope of this variable.

In Chapter 8, we will introduce an algorithm that compiles XQueries into equivalent FluX queries. This algorithm assumes that all XQuery subexpressions are normalized. For each FluX-variable $x and each XQuery-variable $y such that the lineage between $x and $y is defined, we then define the projection path *pp*($y) as *pp*($y) := *lineage*($x, $y) (see Sections 3.4 and 3.5.1).

At runtime, we simultaneously perform XML document projection for all FluX-variables. That is, the data relevant according to the projection paths associated with a FluX-variable $x is copied into the buffer associated with $x. If a node is relevant according to the projection paths of several FluX-variables, it is copied into each of their buffers.

We start filling a buffer each time the scope of its associated FluX-variable $x is entered during query evaluation. This is the case when the opening tag of an context node of $x is read in the input. When the matching closing tag is read, and after the process-stream statement for variable $x has been executed, the contents of this buffer are discarded.

Example 6.7 We consider the FluX expression from Figure 6.4 with the FluX-variables $root, $bib, and $b. We associate a buffer and projection paths with each FluX-variable. After query normalization, we extract the projection paths `$b/author#` and `$b/year#` for FluX-variable $b. These paths identify the data relevant for query evaluation within the process-stream statement of variable $b. For the other variables, no projection paths can be extracted from the query, so the associated buffers remain empty. Let us now consider the input stream

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

from the viewpoint of buffer management. When the opening tag of the book-node is read, prefiltering for the projection paths `$b/author#` and `$b/year#` sets in. The tags for the author-node are copied into the buffer associated with $b. Meanwhile, the streaming query operators are evaluated. When the closing tag for the book-node is read, the buffer contents for variable $b have been collected, and the query evaluation over this buffered data can set in. When the scope of this variable is left, the buffered data associated with $b is no longer relevant for the remaining query evaluation. Hence, its contents are discarded. □

Redundant buffers contents. We now model static buffer purging in our framework. Basically, we synchronize the lifetime of buffer contents with the scopes of the associated FluX-variables, so little runtime overhead is inflicted for buffer purging. Moreover, the buffer preemption points are safe insofar as only data no longer relevant to query evaluation is purged. On the downside, we risk buffering data redundantly, as we discuss in the following example.

Example 6.8 The XQuery in Figure 6.14(a) outputs all books that have price-information, followed by all articles where an author has also edited a book.

```
<results>
{ for $bib in /bib return
 ( for $b in $bib/book
   where ( exists($b/price) )
   return $b,
   for $a in $bib/article
   where
    ( some $b2 in $bib/book
      satisfies
       ($b2/editor = $a/author) )
   return $a ) }
</results>
```

(a) XQuery.

```
{ process-streams $root:
   on-first past() return <results>,
   on bib as $bib return
    { process-stream $bib:
       on book as $b return
        { process-stream $b:
           on-first past(*) return
              if ( $b/price )
              then $b else () },
           on-first past(*) return
              for $a in $bib/article
              where
               (some $b2 in $bib/book
                satisfies
                 ($b2/editor = $a/author))
              return $a  },
   on-first past(*) return  </results> }
```

(b) FluX query.

Figure 6.14: *An XQuery expression and its FluX equivalent.*

The FluX expression for this query is shown in subfigure (b). It specifies the evaluation of one book at-a-time, while *all* articles are processed as buffered data, once the complete input has been read.

In static analysis, we normalize all XQuery subexpressions of the FluX query, and extract projection paths for the FluX-variables. For FluX-variable $bib, we extract `$bib/book`, `$bib/book/editor#`, further `$bib/article/author#` and `$bib/article#`. For FluX-variable $b, we extract `$b/price` and `$b#`.

Let us now assume that we read an input stream starting with the sequence of events "$\langle bib\rangle\langle book\rangle\langle editor\rangle$". Then the book-node and its editor-child are stored in the buffer associated with variable $b, as this data has to be buffered as part of the current book node. However, this data is also stored in the buffer associated with variable $bib, as it is relevant for evaluating the for-loop over articles. Hence, the editor nodes are buffered redundantly. □

Modeling static buffer purging. In Section 6.2.1, we have modeled a query engine based on the FluX language in our framework, where we assume that XQuery expressions are evaluated sequentially. We model the buffer management part next. We show how buffers are filled, and how they are purged.

Filling buffers. In static analysis of FluX queries, we proceed as described above. For each FluX-variable, we extract projection paths. These are compiled into *XML-DPDT*s that implement XML document projection, and can be modeled in our framework (see Sections 3.5.4 and 5.1).

Let $\$y_1, \ldots, \$y_n$ be the FluX-variables in a FluX query F, and let the states $q_0^{\$y_1}, \ldots, q_0^{\$y_n}$ be the initial states of the *XML-DPDT*s projecting for these variables. Then the start-term "$" is replaced according to the term rewriting

For a given FluX expression "on a as \$$y$ return {process-stream \$$y$: $h_1^{\$y}$, …, $h_n^{\$y}$}":

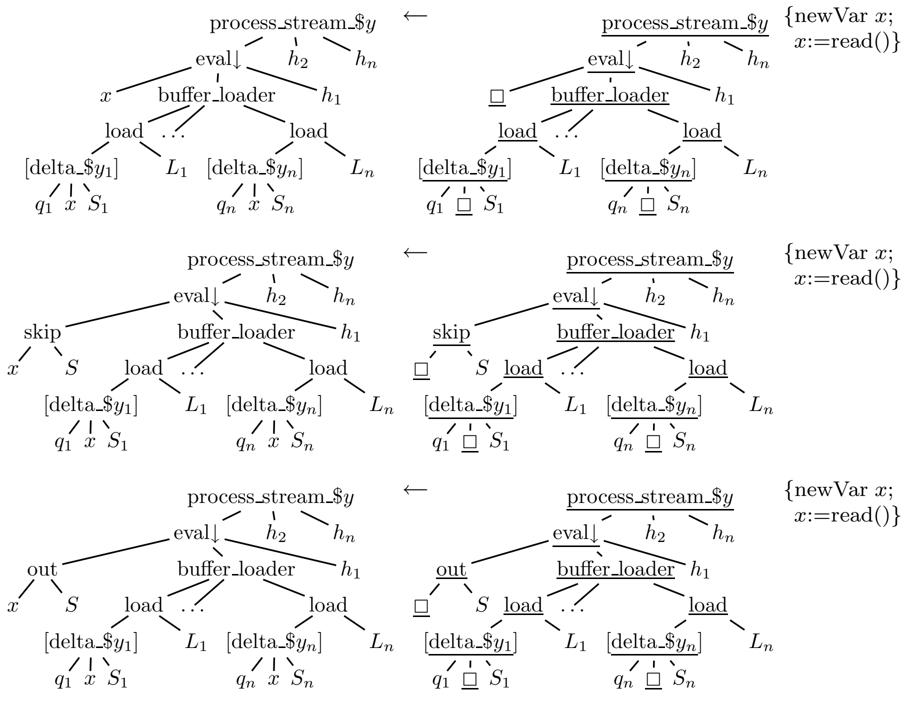
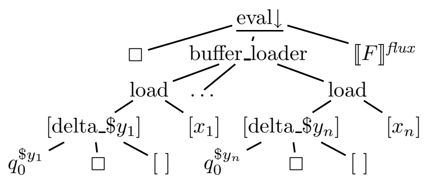


***Figure 6.15**: Prefiltering the input in static buffer management.*

rule below, where "$[\![F]\!]^{flux}$" denotes the term representation of FluX query F.



$\leftarrow$ $\underline{\$}$ { newVar x_1; …; newVar x_n;
root:=
buffer_partition(bind_\$$y_1$($x_1$),
…, bind_\$$y_n$($x_n$)) }

This rule reads as follows. In the action, we define buffer-variables to allocate a buffer partition inside the buffer-term for each FluX-variable. An n-ary function symbol "buffer_partition" separates these buffer partitions. The function "buffer_loader" dispatches the loading of data into buffers. Each input token is fed to the *XML-DPDT*s, denoted by the delta-functions, which prefilter the input. If an *XML-DPDT* preserves a token, the attached loader stores it in the designated buffer. Loaders were defined back in Section 4.3.2.

Each loader carries its own variable stack, which addresses a specific buffer partition. The input token is also processed by the eval-function. The next token is read from the input stream only when loading and query evaluation for this token are finished.

We redefine the eval-function from the previous section by adding a function "buffer_loader" as an additional parameter. In particular, we redefine all terms of the form "eval↑(e_1, e_2)"or "eval↓(e_1, e_2)" in rewriting rules by introducing a

For a given FluX expression "on a as \$$y$ return {process-stream \$$y$: $h_1^{\$y}$, …, $h_n^{\$y}$}":

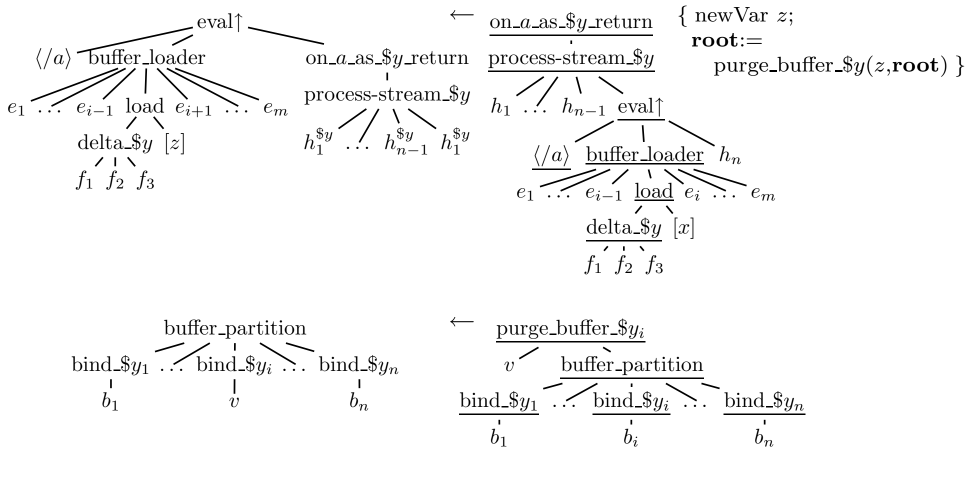


Figure 6.16: Purging buffers at the end of variable scopes.

fresh variable name x, and obtain terms "eval↑(e_1, x, e_2)" or "eval↓(e_1, x, e_2)". Further, we substitute the rules for reading input tokens from Figure 6.7 with the rules in Figure 6.15. Again, we use the designated symbol "□" to signal that the input token has been processed. This stalls all actions until the next input token has been read.

Purging buffers. To purge the buffer associated with a FluX-variable \$$y$, we redefine the bottom-most rule from Figure 6.8 as shown in Figure 6.16. When the scope of a variable is left, we define a a fresh buffer-variable z, by which we override the buffer partition for query-variable \$$y$. This discards all data previously buffered for \$$y$, while the buffer partitions of the other variables remain unchanged. The purging function is specified in the same figure.

Example 6.9 We now model the buffer management aspects of evaluating the FluX query from Example 6.8 on the input stream prefix

⟨bib⟩⟨book⟩⟨title⟩⟨/title⟩⟨price⟩⟨/price⟩⟨/book⟩.

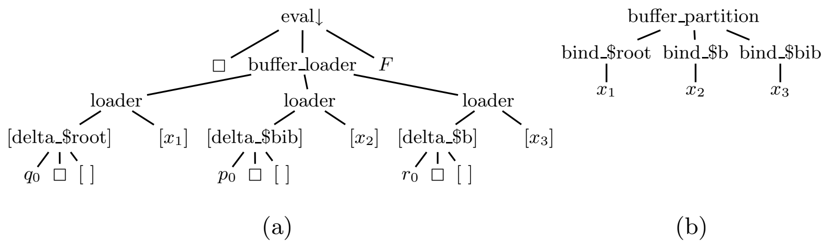
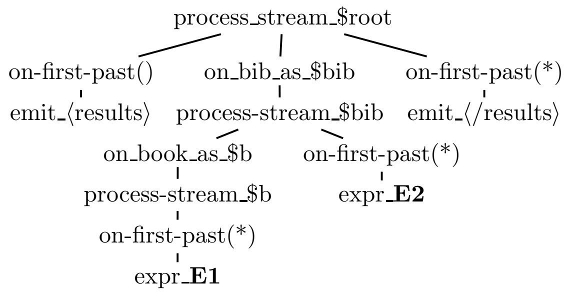


Figure 6.17: Query-term (a) and buffer-term (b) from Example 6.9.

Initially, the start-term "\$" is replaced and we obtain the buffer- and query-terms shown in Figure 6.17. The buffer-term has separate partitions for the FluX-variables \$root, \$bib, and \$b. So far, the buffer partitions only contain variable bindings to buffer-variables.

In the query-term, the *XML-DPDT*s encoded by functions "delta_\$root", "delta_\$bib", and "delta_\$b" project the input into the buffer partitions of FluX-variables. Terms q_0, p_0, and r_0 denote the initial transducer states.

By F, we abbreviate the term representation of the FluX query, which is shown below. The abbreviations **E1** and **E2** represent XQuery expressions, with **E1** in place of "if (\$b/price) then \$b else ( )", and **E2** for the expression "for \$a in \$bib/article".



Term encoding of a FluX query.

By the rules from Figure 6.15, the first input token ⟨*bib*⟩ is read, and used to substitute the placeholder term "□". The input token is then processed by the *XML-DPDT*s and also by the event handlers.

In Figure 6.18(a), we show the query-term in an advanced state, where the event-handler "on bib as \$bib" is about to fire. The *XML-DPDT*s are still in their initial states and have not yet processed the first input token. We fast-forward to the moment when the token ⟨*book*⟩ has been read in the input stream, as illustrated by the snapshot in Figure 6.18(b). The "on book"-handler is about to fire, and the *XML-DPDT*s have already changed their state and stack for processing the opening tag of the bib-node. We zoom in on the *XML-DPDT*s in Figure 6.19(a). The delta-functions have processed their input token. The *XML-DPDT*s for variables \$bib and \$b preserve this token in XML document projection, and in the next step, the attached loaders create book-nodes in their buffer partitions. The resulting *XML-DPDT* configurations and the buffer-term are shown in Figures 6.19(b) and (c) respectively.

When token ⟨/*book*⟩ is read, the buffers for FluX-variables \$bib and \$b are updated and the "on-first past(*)"-handler for variable \$b fires. Consequently, the XQuery expression "if (\$b/price) then \$b else()" is evaluated over the buffer-partition for variable \$b. Upon completion, the scope of variable \$b is left. The buffer partition for this variable is purged, and replaced with a fresh buffer-variable y. This variable is also inserted in the stack of the loader for variable \$b, in accordance with Figure 6.16. This leaves us with the query-term from Figure 6.18(c) and the buffer-term shown in Figure 6.19(d).

The evaluation proceeds with checking the event handlers of variable \$bib for the input token ⟨/*book*⟩. □

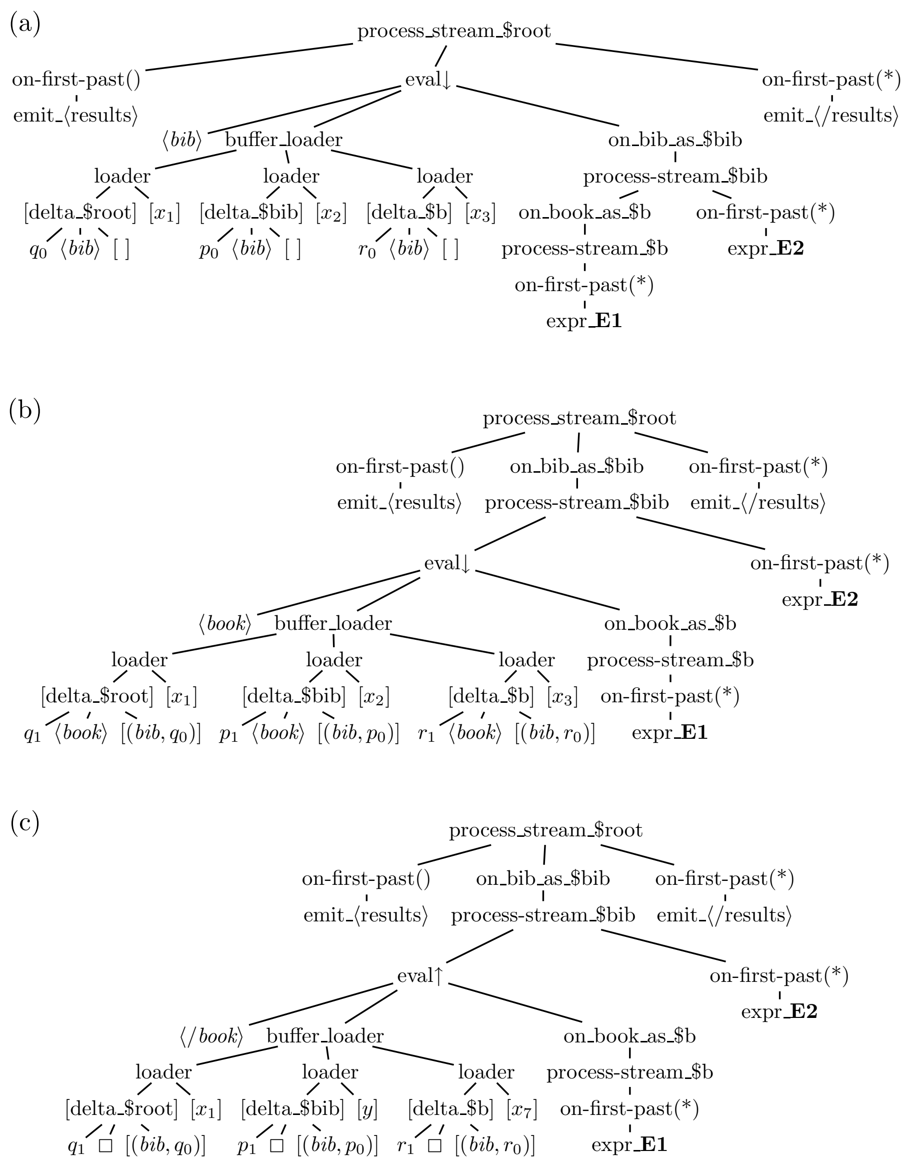


Figure 6.18: *Snapshots of query-terms (Example 6.9).*

6.3.2 Dynamic Buffer Purging

We next introduce a buffer purging algorithm that relies both on static and on dynamic analysis. For the XQuery fragment XQ considered in this thesis, we can actually guarantee that no redundant buffering occurs. The approach centers around the the notion of the relevance of data to query evaluation, and we can draw on the concepts developed for XML document projection in Section 3.5. When loading nodes into the buffer, we compute the number of query subexpressions for which each node is relevant. These so-called *relevance*

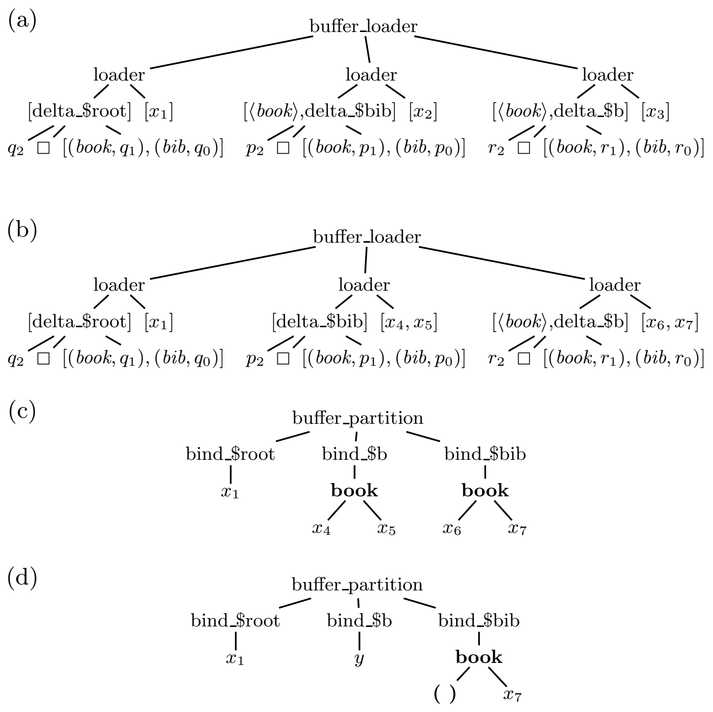

(a)
buffer_loader
loader
[delta_$root] [$x_1$]
q_2 □ [$(book, q_1), (bib, q_0)$]
[⟨book⟩,delta_$bib] [$x_2$]
p_2 □ [$(book, p_1), (bib, p_0)$]
[⟨book⟩,delta_$b] [$x_3$]
r_2 □ [$(book, r_1), (bib, r_0)$]
(b)
[delta_$bib] [$x_4, x_5$]
[⟨book⟩,delta_$b] [$x_6, x_7$]
(c)
buffer_partition
bind_$root
bind_$b
bind_$bib
book
x_1 x_4 x_5 x_6 x_7
(d)
y
( )

***Figure 6.19**: Snapshots of buffer-terms (Example 6.9).*

counts are assigned to each buffered node, a process that we refer to as *tagging*. All relevant nodes are stored in a single buffer. Hence, we no longer buffer data separately (and possibly redundantly) for different variables. During query evaluation, this *relevance count* of nodes is decreased. Once a node is no longer relevant to query evaluation, it is purged from the buffer.

This strategy is reminiscent of garbage collection [110], 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. The approach introduced here is related, yet as we will see, *relevance* counts are assigned to nodes only once during loading, and are decremented continuously during query evaluation. In the original reference counting, increments and decrements may be interleaved.

Moreover, in decrementing reference counts we exploit knowledge about data access patterns that are derived in static query analysis. In contrast, classic garbage collection is only based on dynamically computed reference counts.

Criteria for buffer purging. The conditions for removing a node from the buffer are that (1) the node itself is not tagged, (2) none of its descendants are

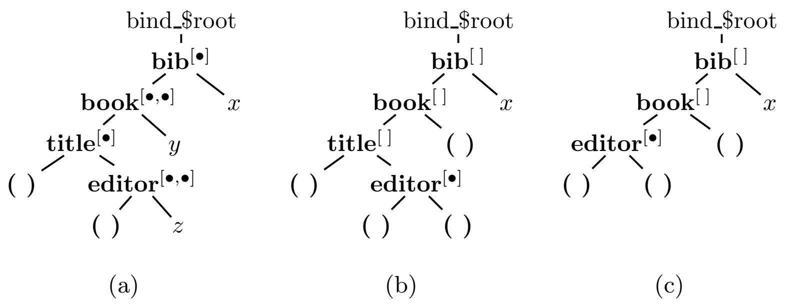


Figure 6.20: *Buffer snapshots of Example 6.10.*

tagged, and (3) the node is "finished", i.e. we have already read its closing tag in the input stream.

Overview. In this section, we present the general ideas, and model them in our framework. We assume a streaming XQuery engine in the style of Section 6.2.2, where queries are evaluated incrementally over the buffered input. This setup, in combination with dynamic buffer purging, has been successfully implemented in the GCX XQuery engine, and is discussed in Chapter 9. We point out that the techniques described here could also be made to work with streaming query engines that employ static operator scheduling, as described in Section 6.2.1. Then all steps described here only apply to the query expressions that are evaluated over buffered data.

The following example motivates the idea that each buffered node keeps track of its future relevance to query evaluation via its relevance count.

Example 6.10 We consider the XQuery from Figure 6.14(a) and the input stream prefix "$\langle bib\rangle\langle book\rangle\langle title/\rangle\langle editor/\rangle\ldots$". In sequential XQuery evaluation, books are processed one at-a-time. Then each book-node must be inevitably buffered at least until it is clear whether it contains a price node or not. The buffer-term is shown in binary tree notation in Figure 6.20(a), where the buffer-variables x, y, and z mark parts of the input that have not been processed yet. For now, we ignore the annotations to nodes in this figure.

For the given query, we extract a set of projection paths (after query normalization), among them `/bib`, `/bib/book` for binding the variable in the for-loop over books, `/bib/book#` for copying books to the output, and further the path `/bib/book/editor#` to compare book editors in the for-loop over articles.

Then all buffered nodes are tagged with a *dot*, which represents a single relevance count unit, for each time that they match a projection path. This is depicted in Figure 6.20(a). As some nodes match several projection paths, they are tagged with several dots, such as the book- and the editor-node. The total number of dots assigned to a node defines its relevance count.

We assume that the closing-tag for the book node is the next token read from the input stream. Now, one iteration of the for-loop over books can be evaluated, and all dots that were assigned for this loop-iteration are removed. In particular, the book-node loses two relevance counts, for the projection paths `/bib/book` and `/bib/book#` that originate from this for-loop. Also, the title- and editor-nodes lose one count each. As a result, we obtain the buffer contents from

Figure 6.20(b). Note that the buffer variables y and z have been replaced by empty terms, as we have meanwhile read further tokens from the input stream.

As the relevance counts have changed, we search the buffer-term for nodes to purge. In this example, the title-node can be eliminated from the buffer, as this node and none of its descendants are tagged. (Note that despite the binary tree notation, the title- and editor-node are actually siblings in the XML document).

A snapshot of the buffer after garbage collection is shown in Figure 6.20(c). The buffer contains all data that has been read so far, and that is still relevant for evaluating the for-loop over article-nodes. These are the nodes relevant according to the projection path `/bib/book/editor#`. □

Modeling relevance counts. We design our XQuery processor such that the tokenized input stream is prefiltered and loaded into the buffer. At the same time, we tag the buffered nodes with relevance counts. The means for modeling this in our framework are introduced below.

Counting paths. We model counting paths and their updates, as introduced in Section 3.6. We encode a counting path with n step expressions of the form "$_{[x_0]}/s_{1[x_1]}/s_{2[x_2]}/\ldots/s_{n[x_n]}$" as a term where L_i is a list containing exactly x_i dots. Dots and step expressions are modeled by nullary functions. Using a function symbol "path$_n$" of arity $2n+1$, we define the counting path from above as "$\text{path}_n(L_0, s_1, L_1, s_2, L_2, \ldots, s_n, L_n)$".

Example 6.11 We encode the counting path "$_{[1]}$`//a`$_{[2]}$`//b`$_{[0]}$" in our framework as the term "$\text{path}_2([\bullet], \texttt{//a}, [\bullet,\bullet], \texttt{//b}, [\,])$". □

We can also integrate the update rules for counting paths, as introduced in Figure 3.12, in our framework. To this end, we define an update-function that takes two parameters, a tagname a of the node that is matched, and a counting path p that is to be updated. An update is then initiated by the term "update(a, p)". Note that in Figure 3.12, we actually introduce two update functions to handle tail-counters differently. In specifying term rewriting rules, we can easily distinguish the tails of counting paths from their bodies. Hence, we introduce only a single update function in our framework.

We show the rules for matching the XPath step `//a` in Figure 6.21, with the operator "::" for list concatenation. By concatenating lists of dots, we sum up relevance counts. The remaining update rules can be modeled accordingly.

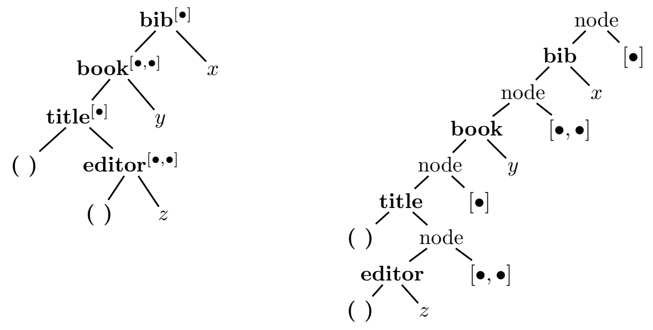
Modeling tagged nodes. Each node in the buffer-term is annotated with its relevance count, which we represent as a list of dots. For the purpose of tagging nodes, we define a binary function "node", which associates nodes and their relevance counts. A binary tree node "$\text{a}[e_1]e_2$" is encoded as "$\text{node}(\text{a}[e_1']e_2', L)$", where L is the list of dots encoding the reference count for this node, and e_1' and e_2' encode the children e_1 and the following siblings e_2 accordingly.

For instance, the tagged buffer contents shown in Figure 6.22(a) are encoded as shown in Figure 6.22(b). It is straightforward to adapt the term rewriting rules for query evaluation to this new format.

For distinct tagnames a and b, and a counting path with n step expressions:

$$\begin{aligned}
&\text{path}_n(L_0, s_1, \ldots, s_n, \text{update}(a, L_n)) \leftarrow \underline{\text{update}}(\underline{a},\ \underline{\text{path}_n}(L_0, s_1, \ldots, s_n, L_n))\\
&\text{update}(\Box,\ \text{path}_n(L_0, s_1, \ldots, s_n, L_n)) \leftarrow \underline{\text{path}_n}(\underline{\text{update}}(\underline{a}, L_0),\ s_1, \ldots, s_n, L_n)\\
&\text{path}_n(L_0, s_1, \ldots, s_{n-1}, \text{update}(a, L_{n-1}), //a, L_{n-1})\\
&\qquad \leftarrow \underline{\text{path}_n}(L_0, s_1, \ldots, s_{n-1}, L_{n-1}, \underline{//a}, \underline{\text{update}}(\underline{a}, L_n))\\
&\text{path}_n(L_0, s_1, \ldots, s_{n-1}, \text{update}(a, L_{n-1}), //b, [\,])\\
&\qquad \leftarrow \underline{\text{path}_n}(L_0, s_1, \ldots, s_{n-1}, L_{n-1}, \underline{//b}, \underline{\text{update}}(\underline{a}, L_n))\\
&\text{path}_n(L_0, s_1, \ldots, s_{i-1}, \text{update}(a, L_{i-1}), //a,\ L_{i-1} :: L_i,\ s_{i+1}, \ldots, s_n, L_n)\\
&\qquad \leftarrow \underline{\text{path}_n}(L_0, s_1, \ldots, s_{i-1}, L_{i-1}, \underline{//a}, \underline{\text{update}}(\underline{a}, L_i), s_{i+1}, \ldots, s_n, L_n)\\
&\text{path}_n(L_0, s_1, \ldots, s_{i-1}, \text{update}(a, L_{i-1}), //a,\ L_i,\ s_{i+1}, \ldots, s_n, L_n)\\
&\qquad \leftarrow \underline{\text{path}_n}(L_0, s_1, \ldots, s_{i-1}, L_{i-1}, \underline{//b}, \underline{\text{update}}(\underline{a}, L_i), s_{i+1}, \ldots, s_n, L_n)
\end{aligned}$$

***Figure 6.21**: Updating counting paths.*



(a) Abstract view of tagged nodes. (b) Encoding of tagged nodes.

***Figure 6.22**: Modeling buffer contents.*

Tagging nodes with relevance counts. Tagging buffered nodes with relevance counts is realized by function "relevance_counter$_n(x, [\![P_1]\!], \ldots, [\![P_n]\!])$", for n projection paths $P_1, \ldots, P_n$. This function operates over the buffer-term. By $[\![P_i]\!]$, we denote the encoding of a projection path as a counting path, whereas x denotes a tree node. The buffered tree is processed top-down, while it is loaded. In this traversal, each element node a is replaced by the term "node(a,L)" where L encodes the relevance count for this node.

Example 6.12 We assume the projection paths `//a` and `//a//b`, and the input stream "$\langle a\rangle\langle c\rangle\langle a\rangle\langle b\rangle\langle/b\rangle\langle/a\rangle\langle/c\rangle\langle/a\rangle$". This encodes an XML document with a single path from the root to a leaf node, via nodes labeled a, c, a, and b.

We prefilter the input for these projection paths using an *XML-DPDT*, encoded by the transition function "delta" (see Section 5.1). Then prefiltering will discard the c-labeled node as irrelevant.

We set up a system where the start-term "\$" is rewritten according to the following rule. There, q_0 is the initial state of the *XML-DPDT*, and the term "$\Box$" signals that an input token must be read.

For all tagnames a and n projection paths:

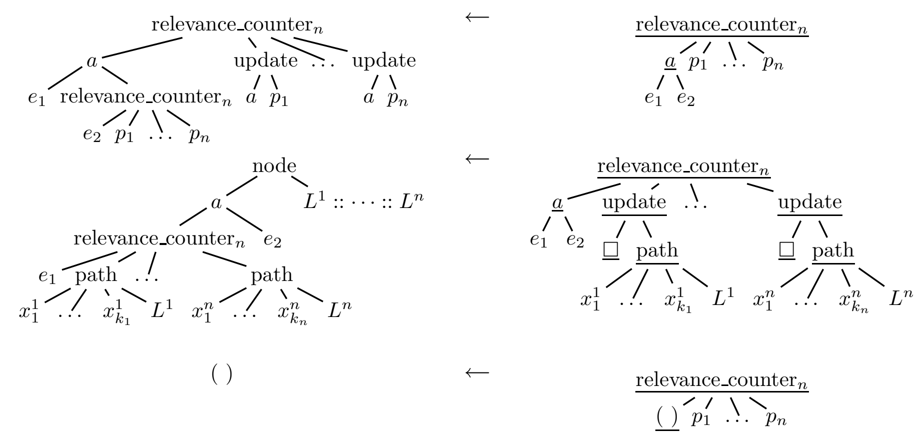
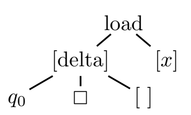


***Figure 6.23**: Tagging nodes with relevance counts.*



$\leftarrow$ $\underline{\$}$

{ newVar x;
root:= bind_\$root(
relevance_counter$_2$(x, path$_1$([•], `//a`, [ ]),
path$_2$([•], `//a`, [ ], `//b`, [ ]))) }

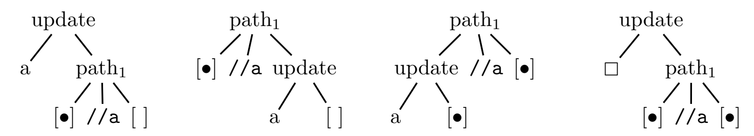
The new query-term initiates filtering and loading of the input. At the same time, relevance counts are assigned to the nodes in the buffered tree. □

In Figure 6.23, we define the tagging of nodes in the buffered tree. The first rule replicates the relevance counter to the following sibling of the current node, and starts the computation of relevances for the node itself. When all counting paths have been updated, the sum of relevance counts is assigned to the current node by the second rule. The children of the node are processed next. According to the third rule, leaf nodes "( )" are not considered.

Example 6.13 We continue with Example 6.12 and assign relevance counts to the nodes in the buffered tree. For ease of illustration, we assume that the input is already loaded, but loading and tagging can be interleaved.

Our starting point is the buffer-term from Figure 6.24(a). In Figure 6.24(b), we propagate relevance counting to the sibling of the topmost a-labeled node, and start updating the counting paths for this node.

The term applied to the sibling of the topmost a-labeled node can be directly reduced to a leaf node. We zoom in on updating one of the projection paths for the a-labeled node, and show several steps below. These read from left to right.



The second counting path is also rewritten, and we obtain the term from Figure 6.24(c). Now that the counting paths are up-to-date, we assign the relevance count "one" to the topmost a-labeled node.

In step (d), we depict the moment when we have updated the counting paths for the second a-labeled node. The second a-labeled node is also matched by

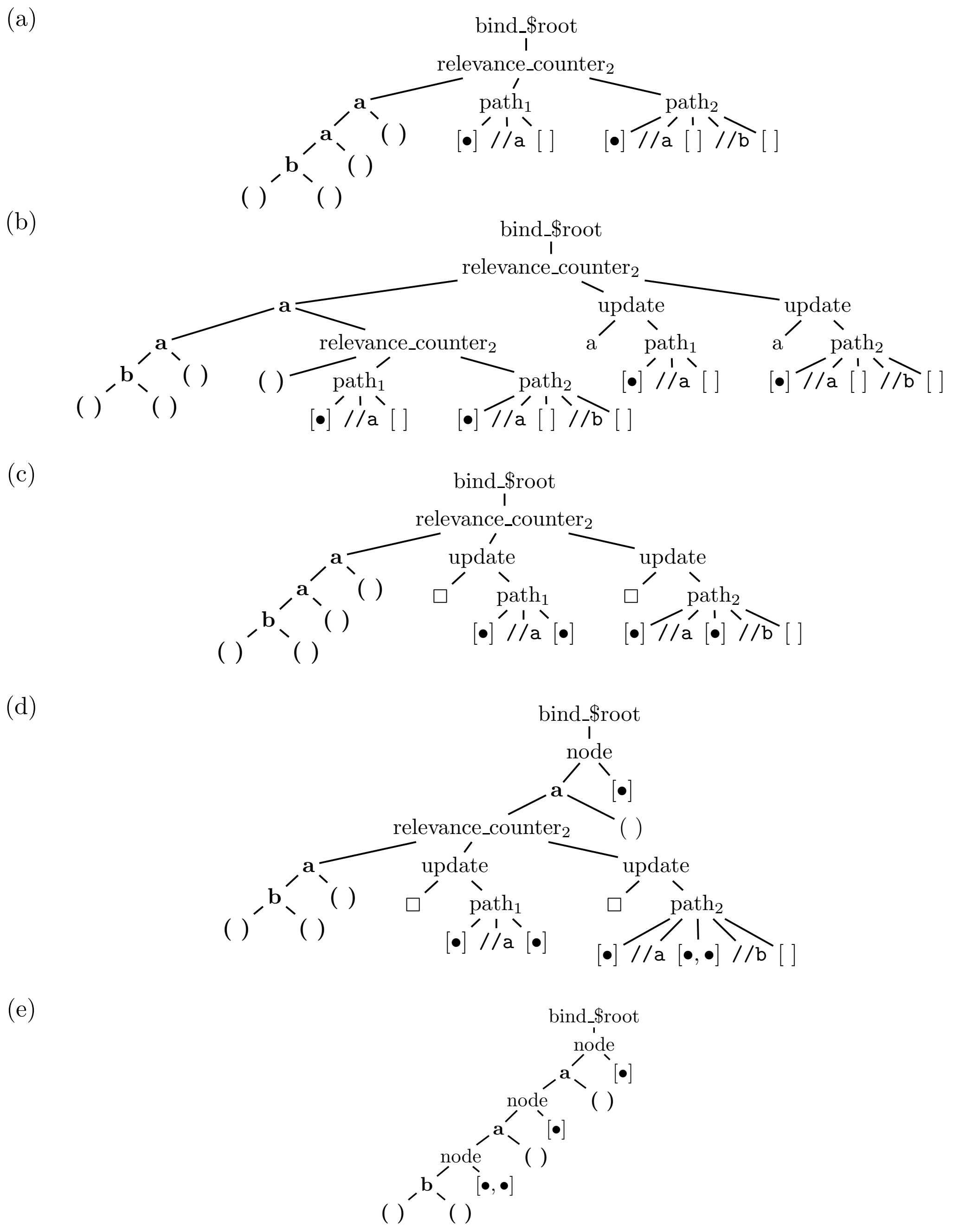


Figure 6.24: *Snapshots in tagging buffered nodes (Example 6.24).*

one projection path, and receives relevance count "one". Next, the counting paths for the *b*-labeled node are updated to two. This is shown in subfigure (d). In subfigure (e), we show the final tagged tree. □

```
<results>{
for $b in //book return
 <book>{
   for $t in $b/title
   return $t,
   for $a in $b/author
   return $a,
   for $y in $b/year
   return $y }
 </book> }

</results>
```

(a) XQuery.

```
<results>{
 for $b in //book return
  ( <book>{
      for $t in $b/title return
        ( $t, signOff($t#) ),
      for $a in $b/author return
        ( $a, signOff($a#) ),
      for $y in $b/year return
        ( $y, signOff($y#) )
    }</book>,
    signOff($b) ) }
</results>
```

(b) XQuery with signOff-statements.

Figure 6.25*: Marking buffer preemption points in XQueries.*

Decrementing relevance counts. The moments during query evaluation when relevance counts are decremented are determined at compile time. We refer to these moments as *buffer preemption points*.

Under the assumption that XQueries are evaluated sequentially, we insert signOff-statements into queries, which we position at the buffer preemption points. Intuitively, executing an statement "signOff($\$x/\pi$)" has the effect that the relevance counts of nodes in the buffer tree are decremented, provided that these nodes are reachable from the context-node of variable $\$x$ via the path π.

Example 6.14 We consider the XQuery from Figure 6.25(a), which outputs the title, author, and year information of each book. We extract the projection paths `//book`, `//book/title#`, `//book/author#`, and `//book/year#`.

Let us further assume the input stream prefix "$\langle book\rangle\langle title\rangle\langle main\_title\rangle\ldots$". In processing these tokens, the book-node, the title-node, and its descendants are each loaded into the buffer and are tagged with a relevance count of one.

We statically insert signOff-statements into the query to denote when certain nodes are no longer relevant. The resulting query is shown in Figure 6.25(b). Let us focus on the for-loop over title nodes. Once a title has been output, its relevance count is decremented. Now, the subtree rooted at this title may be deleted from the buffer (provided it does not contain book-nodes with non-zero relevance counts). The relevance counts of author- and year-nodes remain unchanged, these nodes stay in the buffer for the remaining query evaluation.

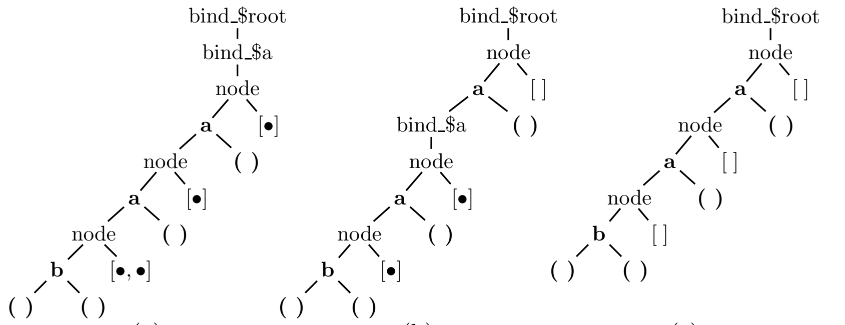
We proceed similarly for the author- and year-nodes. At the end of the scope of variable $\$b$, the relevance count of its context node is also decremented. □

Example 6.15 We consider the XQuery with signOff-statements below, from which we extract the projection paths `//a` and `//a//b`.

```
<results>{
  for $a in //a return
    ( if ( some $b in $a//b satisfies true() )
      then <a></a> else (),
      signOff($a//b), signOff($a) ) }
</results>
```

Back in Example 6.13 we have shown the buffer contents with relevance counts for a concrete input document. Below, we show some snapshots of this buffer. In snapshot (a), variable \$a is bound to the topmost *a*-labeled node. When query evaluation reaches the end of the scope of variable \$a, the signOff-statements are executed and the relevance counts are updated. As each buffered node has either a nonzero relevance count, or descendants with this property, no nodes can be purged yet by our purging criteria.[2]

Next, variable \$a is bound to the second *a*-labeled node. This is portrayed in the snapshot (b). At the end of scope of \$a, the relevance counts are again updated. The result is depicted in snapshot (c).



(a) (b) (c)

Given the final buffer contents, garbage collection can purge all nodes. □

Compilation of signOff-statements. We now model signOff-statements in our framework. We assume a binary function "decrement_count" that takes a buffered tree and a projection path (relative to a query-variable) as arguments. The semantics is that for all nodes addressed by the projection path, the relevance count is decremented. Each node is decremented as many times as it matches the projection path. We do not specify the rewriting rules for this function, and instead focus on the general picture.

We compile a statement $[\![\text{signOff}(\$x/\alpha)]\!]$ into a term "signOff($\$x/\alpha$)", for which we define a rewriting rule as shown below.

$$(\ ) \quad \leftarrow \quad \underline{\text{signOff}}(\$x/\alpha) \quad \{\ \textbf{root}\colon = \text{decrement\_count}(\textbf{root}, \$x/\alpha)\ \}$$

It remains to specify how nodes are purged from buffers.

Purging nodes from buffers. There are several options as when to invoke the garbage collection mechanism, which inflict different memory footprints and runtime behaviors. For instance, garbage collection could be invoked whenever memory consumption reaches a predefined limit. In our model, this could be a certain number of buffered nodes. Garbage collection could also sweep the buffer periodically, each time that a fixed number of input tokens has been consumed. Another alternative is to wait for the end of a variable scope, once all signOff-statements have been executed.

[2]In this particular example, the *a*-labeled node could actually be removed. Yet in general, this node may still be required for navigational purposes for other parts of the query. In order not to corrupt the buffer contents, we preserve this node.

We make in our goal to purge nodes from buffers as soon as possible. Out of this consideration, we scan the buffer for nodes that can be purged whenever a signOff-statement is executed, or a node has been entirely loaded into the buffer. In doing so, we react on every chance that the buffer contents have changed such that nodes may be purged (see Chapter 9).

6.3.3 Discussion

In this section, we have introduced two buffer purging mechanisms. The first relies on static analysis, and is intended for systems that statically schedule streaming query operators, such as the FluX approach. While this requires comparatively little building effort, it may lead to redundant buffering.

The second approach is technically more involved. It combines static and dynamic analysis, and achieves redundant-free buffering for our query fragment XQ. At compile-time, we determine the buffer preemption points when buffers are searched for nodes to purge. As we will see in Chapter 9, deriving the necessary data access patterns is a non-trivial task. At runtime, nodes that are copied into the buffer are assigned relevance counts. Nodes lose their relevance counts during query evaluation incrementally, until they can be purged. The challenge is to ensure that they are not deleted prematurely from buffers, while at the same time, we aim at timely freeing main memory resources.

We have implemented both approaches in the FluXQuery and the GCX XQuery engines, which we discuss in Chapters 8 and 9. Our experiments confirm that our buffer management techniques perform well both with regard to main memory consumption and execution time.

6.4 Summary

In this chapter, we have discussed ways to reduce the memory footprint of main memory-based XQuery processors. We make a case for sequential query evaluation to avoid buffering intermediate results. We further propose an evaluation of queries on-the-fly. We have stressed the importance of preemptive buffer management to free memory resources early on.

When XML data is processed with query engines that are disk-backed, different arguments apply. Then data access is not restricted to a single pass, and data purged from main memory buffers is not irretrievably lost. However, it turns out that our streaming execution model brings about certain problems of its own. This concerns the evaluation of joins and conditionals in particular. Here, we frequently encounter runtime problems. These problems can be overcome by borrowing from XML databases techniques. We discuss this issue in Chapter 10, among our ideas for future work.

Part III

Buffer Management Core Techniques

7 XML Prefiltering as a String Matching Problem

We present a novel approach to the problem of XML document projection, which we have defined in Section 3.5. We view XML document projection as a string matching problem. The idea is to decompose the prefiltering task into a series of string matching problems. We motivate this approach in the first section in this chapter. The runtime algorithm, which schedules the execution of individual string searches, is presented in Section 7.2. In Section 7.3, we discuss the compilation of the runtime datastructures. We further show how XML-specific jump offsets can be computed from DTDs. In Section 7.4, we present our prototype implementation, and Section 7.5 contains our extensive experiments. We discuss possible optimizations as future work in Section 7.6. We then conclude with a brief summary of our main results.

7.1 Motivation

In XML data management, we frequently face similar problems as in string matching, as we often need to detect patterns (such as a specific tagname) within XML input streams. However, to date the state-of-the-art in string matching has found little application in the acceleration of XML processing.

String matching algorithms have been subject to extensive study for more than thirty years [1, 2, 18, 30, 65, 109]. In today's algorithms, the input is not processed one character at-a-time. Rather, the Boyer-Moore [18] and the Commentz-Walter [30] algorithm rely on the insight that matching keywords from right to left lets us *skip* parts of the input. For instance, the keyword "ICDE" has four characters. Suppose the fourth character in the input is the letter "A". Then the keyword cannot be matched by the first four characters, and we can directly inspect the 8th character. If this character is the letter "C", then the pattern could be matched. Hence, we shift to the right by an offset the size of the string "DE", and again try to match the keyword from right to left.

In this chapter, we make a case for leveraging ideas from string matching for XML stream processing. We take the leap from processing flat strings to structured documents, and present a new technique for the efficient search and navigation in XML streams. What makes our approach very attractive is that it shares the advantages of established string matching algorithms: Using statically precompiled tables of fixed size, our runtime algorithm comes at little expense for the CPU and main memory resources. Moreover, it can be implemented as a streaming algorithm, where we scan the input with a fixed-size window in a single pass. Within the window held in main memory, we can locally jump back and forth, all the while trying to inspect as few characters as possible. As we confirm in our experiments, this results in significant speedups for searching in flat strings and XML data alike.

While the runtime algorithm is simple, lean, and efficient, the static analysis for computing the runtime datastructures is not trivial. In moving from flat strings to structured data, new challenges arise. When the complete XML input document has been tokenized into opening- and closing tags, e.g. using a SAX parser, it is straightforward to track ancestor-descendant relationships between nodes in the document tree. However, when we skip data, we also disregard parts of the document structure. For instance, assume we search for an occurrence of the keyword $\langle a \rangle$, and once we have found it, we search for keyword $\langle b \rangle$. Ad hoc, the relationship between the corresponding nodes is unclear. All we know for sure is that the a-labeled node is *not* a descendant of the b-labeled node.

To deal with this problem, we make use of schema information. DTDs provide us with the set of possible tagnames, parent-child relationships, as well as order and cardinality constraints. We can even take required attributes into account to compute XML-specific offsets, which let us skip parts of the input in addition to the skips performed by string matching algorithms. Based on a holistic static analysis, we decompose the task of navigating inside XML documents into multiple string matching problems, which are solved individually using established algorithms. This decomposition is robust, and computes a number of very basic lookup-tables to be used at runtime.

By only inspecting a fraction of the characters in the input, we are able to build highly scalable applications for XML stream processing. These applications exhibit a high throughput and an economical use of resources. As a proof-of-concept, we apply our technique to XML document projection. We have already introduced the concept of XML document projection in Section 3.5, and we have also discussed aspects of this technique in the previous chapter.

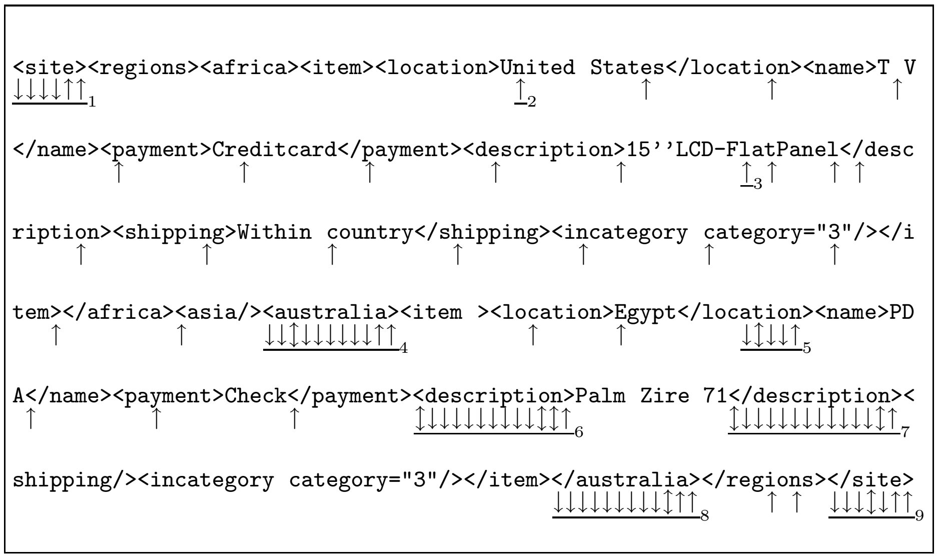
In the following, we sketch how string matching algorithms can be leveraged to accelerate XML prefiltering.

Example 7.1 We discuss XML document projection for the XQuery

```
<q>{ //australia//description }</q>.
```

As discussed in Section 3.5, we extract the projection paths `//australia` and `//australia//description#`. Additionally, we include the path `/*` to ensure that the projected document is well-formed (c.f. Definition 3.14).

We project XML documents that are valid w.r.t. the simplified XMark DTD [122] from Figure 3.2. We assume that the unspecified productions have `#PCDATA`-content, and that element nodes labeled “incategory” have a required attribute “category”. As we have excluded attributes in the definition of DTDs in Section 3.3, we refer to the official W3C specifications [117].



```
<site><regions><africa><item><location>United States</location><name>T V
</name><payment>Creditcard</payment><description>15''LCD-FlatPanel</desc
ription><shipping>Within country</shipping><incategory category="3"/></i
tem></africa><asia/><australia><item ><location>Egypt</location><name>PD
A</name><payment>Check</payment><description>Palm Zire 71</description><
shipping/><incategory category="3"/></item></australia></regions></site>
```

Figure 7.1: *XML document.*

The XML input document from Figure 7.1 adheres to this DTD. In XML document projection, we compute the projected document shown below.

```
<site><australia><description>Palm Zire 71</description>
</australia></site>
```

We now perform XML document projection using string matching algorithms. In localizing opening or closing tags in the unparsed input, we must keep in mind that tags may contain whitespaces or attributes. All tags for an element node with label t share the prefix "$\langle t$" or "$\langle /t$". The tag "$\langle t\quad /\rangle$", which contains whitespace, is valid syntax. Thus, we search for the keyword "$\langle t$" using string matching algorithms, and then locally seek the trailing "$\rangle$" or "$/\rangle$".

In Figure 7.1, we use symbol $\uparrow$ to mark characters that are checked when searching the input from left to right, and $\downarrow$ for characters that are checked from right to left. Symbol $\updownarrow$ represents both $\uparrow$ and $\downarrow$.

We begin by searching for the opening tag of the site-node, which we know must exist according to the DTD. At position **1**, we check for the keyword "$\langle$*site*" using the Boyer-Moore algorithm. The 5th character ("e") is investigated, and the match is verified from right to left. Character ">" on the right asserts that an opening tag has been found. We then output the token $\langle$*site*$\rangle$ in the course of XML document projection. Next, we are interested in tag $\langle$*australia*$\rangle$. According to the DTD, the string "$\langle$*regions*$\rangle\langle$*africa/*$\rangle\langle$*asia/*$\rangle$" with length 25 is the minimum string preceding this tag. Consequently, we can skip 25 characters, and trigger Boyer-Moore search for the keyword "$\langle$*australia*" with length 10. The search thus is resumed $25 + 10$ characters to the right, at position **2**. Up to position **3**, we check every 10th character and observe that it is not contained in the keyword. This changes when we sample the character "*l*". This character is contained in the keyword, and we shift $|\langle australia| - |\langle austral| = 2$ characters to the right. Reading character "*t*", we rule out a match and the search continues. At position **4**, we finally match and output the tag $\langle$*australia*$\rangle$.

We continue with a simultaneous search for the keywords "$\langle$*description*" and "$\langle$*/australia*" using the Commentz-Walter algorithm. Scanning for the closing tag "$\langle$*/australia*" is necessary because the DTD does not assert the existence of a description-node as a descendant of *australia*. Position **5** shows a suspect match for keyword "$\langle$*description*", which is aborted. We next match the tag $\langle$*description*$\rangle$ at position **6**. We record the start position of this token and search for "$\langle$*/description*". We jump the size of $|\langle$*/description*$|$ characters to the right, match character "$\langle$", and verify a match for $\langle$*/description*$\rangle$ at position **7**. The chunk of data beginning at the recorded start position of $\langle$*description*$\rangle$, and up to (and including) the closing tag $\langle$*description*$\rangle$ is copied verbatim to the output. We resume the search for the keywords "$\langle$*/australia*" and "$\langle$*description*". In locating these keywords, we can perform an initial jump for the string "$\langle$*shipping/*$\rangle\langle$*incategory category*=`""`$\rangle\langle$*/item*$\rangle$". As the minimum distance to the next occurrence of tag $\langle$*description*$\rangle$ is greater, this jump offset is safe. Finally, we match and output the tags $\langle$*/australia*$\rangle$ and $\langle$*/site*$\rangle$ (see positions **8** and **9**).

Even in this toy example, only about 22% of all characters need to be inspected to perform XML document projection. When processing XMark documents in the Gigabyte range (see our experiments in Section 7.5), we observe similar ratios. □

7.2 Runtime Algorithm

In this section, we further motivate the decomposition of an XML prefiltering task into multiple string matching problems. We assume that a non-recursive schema is available in form of a DTD, but we emphasize that all techniques can be extended to also handle the recursive case. We then introduce the runtime algorithm that schedules the execution of the single string matching problems, and all required runtime lookup-tables. The subsequent sections are dedicated to the static compilation of these lookup-tables.

Example. As a basic example, we project XML documents for the projection paths `/a/b#` and `/*`. Then only the top-level nodes with label a, and their b-labeled children (with their subtrees for producing output) are preserved.

We decompose the prefiltering task into multiple string matching problems, where the *frontier vocabulary* is the set of keywords that defines the current string matching problem.

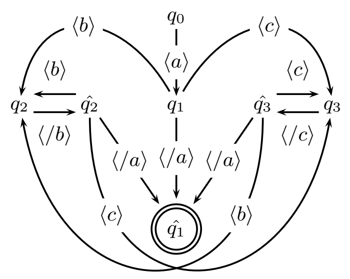
We assume that the input is valid w.r.t. the DTD from Figure 7.2(a). Then the root nodes of all input documents carry the label a. We thus start XML prefiltering with tag $\langle a\rangle$ in the frontier vocabulary. We use the Boyer-Moore algorithm for single keyword search to match token $\langle a\rangle$ in the input. Once we have located this keyword, we consider the frontier vocabulary with the tokens $\langle b\rangle$, $\langle c\rangle$, and $\langle /a\rangle$. This defines a multi-keyword search, so we use the Commentz-Walter algorithm to detect the closest match for any of these keywords. The intuition is that if the closest match is token $\langle b\rangle$, then we have found a relevant node. If we instead find token $\langle c\rangle$, we can ignore the subtree underneath, as this data is not relevant to query evaluation. Finally, if we match $\langle /a\rangle$, then we have reached the end of the parent node. This way, we recognize just enough of the document structure without parsing the complete input into tokens. It

```
<!DOCTYPE a [

  <!ELEMENT a (b|c)*>
  <!ELEMENT b (#PCDATA)>
  <!ELEMENT c (b,b?)>

]>
```

(a) DTD.



$$\begin{tikzcd}
 & & q_0 \arrow[d, "\langle a\rangle"] & & \\
q_2 \arrow[r, shift right, "\langle /b\rangle"'] & \hat{q}_2 \arrow[l, shift right, "\langle b\rangle"'] \arrow[l, bend right, "\langle b\rangle"'] \arrow[rrr, bend right, "\langle c\rangle"] \arrow[dr, "\langle /a\rangle"] & q_1 \arrow[d, "\langle /a\rangle"] \arrow[ll, bend right, "\langle b\rangle"'] \arrow[rr, bend left, "\langle c\rangle"] & \hat{q}_3 \arrow[r, shift right, "\langle c\rangle"'] \arrow[lll, bend left, "\langle b\rangle"'] \arrow[dl, "\langle /a\rangle"'] & q_3 \arrow[l, shift right, "\langle /c\rangle"'] \\
 & & \hat{q}_1 & &
\end{tikzcd}$$

(b) Runtime-automaton.

Figure 7.2: *A DTD and a runtime automaton.*

q	t	$A[q]$
q_0	$\langle a\rangle$	q_1
q_1	$\langle b\rangle$	q_2
q_1	$\langle /a\rangle$	$\hat{q}_1$
q_1	$\langle c\rangle$	q_3
q_2	$\langle /b\rangle$	$\hat{q}_2$
$\hat{q}_2$	$\langle b\rangle$	q_2
$\hat{q}_2$	$\langle c\rangle$	q_3
$\hat{q}_2$	$\langle /a\rangle$	$\hat{q}_1$
q_3	$\langle /c\rangle$	$\hat{q}_3$
$\hat{q}_3$	$\langle c\rangle$	q_3
$\hat{q}_3$	$\langle b\rangle$	q_2
$\hat{q}_3$	$\langle /a\rangle$	$\hat{q}_1$

(a) Table A.

q	$V[q]$
q_0	{ "$\langle a$" }
q_1	{ "$\langle /a$", "$\langle b$", "$\langle c$" }
$\hat{q}_1$	{ }
q_2	{ "$\langle /b$" }
$\hat{q}_2$	{ "$\langle /a$", "$\langle b$", "$\langle c$" }
q_3	{ "$\langle /c$" }
$\hat{q}_3$	{ "$\langle /a$", "$\langle b$", "$\langle c$" }

(b) Table V.

q	$J[q]$
q_0	0
q_1	0
$\hat{q}_1$	0
q_2	0
$\hat{q}_2$	0
q_3	4
$\hat{q}_3$	0

(c) Table J.

q	$T[q]$
q_0	`nop`
q_1	`copy tag`
$\hat{q}_1$	`copy tag`
q_2	`copy on`
$\hat{q}_2$	`copy off`
q_3	`nop`
$\hat{q}_3$	`nop`

(d) Table T.

Figure 7.3: *Runtime lookup-tables.*

is crucial that we also consider the keyword $\langle c\rangle$. If we only scan for tags $\langle b\rangle$ and $\langle /a\rangle$, we cannot distinguish XML documents with prefix "$\langle a\rangle\langle b\rangle\ldots$" from documents with prefix "$\langle a\rangle\langle c\rangle\langle b\rangle\ldots$". Thus, we could mistake a b-labeled child of a c-labeled node for a child of an a-labeled node.

A so-called *runtime-automaton* captures the change between frontier vocabularies. The runtime-automaton in Figure 7.2(b) has the initial state q_0 and the final state $\hat{q}_1$. We use dual states q and $\hat{q}$ for reading the opening- and the closing tag of a node. Given the current automaton state and reading position in the input, string matching algorithms locate the closest token for which an automaton transition is defined.

Four statically compiled tables provide the runtime information. We show the *runtime lookup-tables* for the current example in Figure 7.3. Table A holds the automaton transition function, mapping a state and an input token from the frontier vocabulary to the next state. Table V provides the frontier vocabulary in each state. Tags can contain attributes or whitespace, so the string search does not consider the trailing bracket of the tag, as seen in the lookup-table.

Table J stores the *initial jump offsets*, i.e. the number of positions that can be skipped when entering a new state. This is motivated in the upcoming exam-

```
$q := q_0$;  // current state
$c := 0$;    // cursor position
while cursor $c$ has not yet reached end-of-file and $q$ is not final do
  $c := c + J[q]$;    // shift cursor for initial jump offset
  if $|V[q]| = 1$
  then perform single-keyword search for token in $V[q]$   // (BM)
  else  perform multi-keyword search for tokens in $V[q]$   // (CW)
  end if
  $t :=$ the matched token from $V[q]$;
  $a :=$ tagname in token $t$;
  shift cursor $c$ to the right until reading "$\rangle$" or "$/\rangle$"   // ($\star$)
    to determine matched tag;
  if tag is an opening tag then
    assign $q := A[q, \langle a\rangle]$;
    perform action $T[q]$
  else if tag is a closing tag then
    assign $q := A[q, \langle/a\rangle]$;
    perform action $T[q]$
  else if tag is a bachelor tag then
    assign $q := A[q, \langle a\rangle]$;
    perform action $T[q]$;
    assign $q := A[q, \langle/a\rangle]$;
    perform action $T[q]$
  end if
end while
```

***Figure 7.4**: The runtime algorithm.*

ple. The actions executed in each state are stored in table T. We can perform no operation ("`nop`") or copy the current tag without or with its attributes ("`copy tag`" or "`copy tag + atts`")[1]. To output a node with its subtree, we copy the input from the start position of its opening tag up to the last position of the matching closing tag (actions "`copy on`" and "`copy off`").

Example 7.2 We consider the runtime lookup-tables in Figure 7.3. We assume that we have just located token $\langle c\rangle$ in the input, upon which we have entered state q_3. The current frontier vocabulary is specified in lookup-table V as entry $V[q_3] = \{$"$\langle/c\rangle$"$\}$. So we next search for token $\langle/c\rangle$ in the input. We know from the DTD that the c-labeled node has at least one child. The shortest string representation of this child is as "$\langle b/\rangle$". When we enter state q_3, we can thus skip four characters, as recorded in $J[q_3]$. □

The runtime algorithm. Figure 7.4 shows the runtime algorithm, which performs the context switches between different string matching problems. The current automaton state is denoted by q, and the current reading position in the input, the "cursor", by c. We iterate the following steps. We begin with an initial jump by shifting the cursor $J[q]$ positions to the right. Then we search

[1]The extraction of projection paths in Section 3.5 does not cover attributes. As the necessary extensions are straightforward, we also handle attributes in this chapter.

for the closest token from the frontier vocabulary $V[q]$. If the frontier vocabularies has only one entry, then we employ the Boyer-Moore algorithm, otherwise the Commentz-Walter algorithm. The corresponding lines in the algorithm are marked (BM) and (CW). Once a match for a keyword "$\langle t$" or "$\langle /t$" has been found, we scan to the right for the trailing strings "$\rangle$" or "$/\rangle$". If we have found an opening or a closing tag, we enter the next state and perform the associated action. In case we have found a bachelor tag $\langle a/\rangle$, we evaluate the steps for the opening tag $\langle a\rangle$ and the closing tag $\langle /a\rangle$ in immediate sequence.[2] The cursor now points to the last position with character "$\rangle$" of the matched XML token, and the iteration proceeds. For input valid w.r.t. the DTD, there is always a match when searching for the tokens from the frontier vocabulary.

There is a special case that we have omitted from the runtime algorithm for didactic simplicity. DTDs may allow tagnames that are prefixes of each other, such as "Abstract" and "AbstractText" in the Medline DTD [105]. If a string matching algorithm reports a match for token "$\langle$*Abstract*" in the input, then the runtime-algorithm must additionally ensure that it has not matched the tag $\langle$*AbstractText*$\rangle$ by mistake. This can be incorporated in the scan for the end of the tag, as marked by $(\star)$ in the algorithm.[3]

7.3 Static Compilations

In this section, we statically compute the runtime lookup-tables from projection paths and a non-recursive DTD. The static computation has the following properties. Let D be a non-recursive DTD, and let $\mathcal{P}$ be a set of projection paths. By $R_{D,\mathcal{P}}$ we denote the runtime algorithm from Figure 7.4, initialized with the lookup-tables computed for D and $\mathcal{P} \cup \{\texttt{/*}\}$, as shown below. Then for all XML documents valid w.r.t. the DTD, $R_{D,\mathcal{P}}$ executes the projection function $f_{\mathcal{P}}$ defined in Section 3.5. Thus, the projection implemented is safe.

7.3.1 Runtime-Automata

We compile a given nonrecursive DTD into a finite-state-automaton (FSA). The resulting *DTD-automata* recognize all XML documents valid w.r.t. a DTD. Ultimately, we want to associate actions with automaton states, as in lookup-table T in Figure 7.3. To this end, we use Glushkov automata, as introduced in Section 3.1. State q_0 always denotes the initial state, and we use dual states q and $\hat{q}$ to distinguish the opening- and closing tags of a node.

Example 7.3 Figure 7.5 shows the DTD-automaton for the DTD specified in Figure 7.2(a). □

We compute the runtime-automaton from a subgraph of the graph defined by the DTD-automaton. Our goal is to select a *small* subgraph, as we do not want to tokenize the complete input, but rather parse as few nodes as possible into tokens. At the same time, we need to ensure that we visit all tags that are part of relevant data, as this data needs to be preserved in XML prefiltering.

[2]At this point, we could only check for bachelor tags when permitted in the DTD.

[3]We could additionally exploit schema information and only check for ambiguous prefixes in tagnames if this is possible by the DTD.

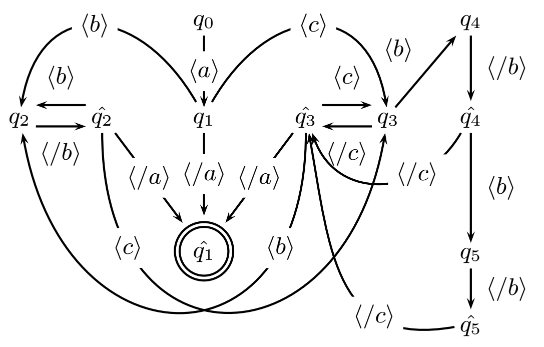


***Figure 7.5**: DTD-automaton.*

As outlined in Section 7.2, we may also have to stop over at additional nodes in order to maintain a minimum amount of orientation.

Definition 7.1 Given a homogeneous FSA D and a subset of its states S, then the *subgraph-automaton* $D_{|S}$ is the FSA with states S, and with the transitions defined as follows. For each sequence of transitions of the form $q \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n \rightarrow p$ of D where $n \geq 0$, where only states q and p are in S and state p is a-labeled, we define a transition for $D_{|S}$ from q to p with label a. A state is final in $D_{|S}$ if it is final in D. □

By construction, the subgraph-automaton $D_{|S}$ is also homogeneous.

We next introduce some basic terminology. The layout of the DTD-automaton reflects the parent-child relationships between nodes in the accepted XML documents. A state p is a *parent state* of a state q if there is a well-formed XML document where a node matched by p is a parent of a node matched by q.

Example 7.4 For the DTD-automaton in Figure 7.5, state q_0 has no parent states, but it is the parent state of the states q_1 and $\hat{q_1}$. In return, q_1 and $\hat{q_1}$ are the parent states of $q_2, \hat{q_2}$ and $q_3, \hat{q_3}$. □

For each state in a DTD-automaton (derived from a non-recursive DTD), we can determine its *ancestor path*. This is the sequence of tagnames that describe a path from the root of the document via ancestor nodes into this state.

Example 7.5 For the DTD-automaton in Figure 7.5, state q_0 has the empty ancestor path, states q_1 and $\hat{q_1}$ have the ancestor path "a", and states q_2 and $\hat{q_2}$ have the ancestor path with the tokens "a" and "b". □

We next evaluate path-matching automata on DTDs, instead of the XML documents that they define. Let $\mathcal{P}$ be a set of projection paths. We say that a state in the DTD-automaton describes nodes relevant according to $\mathcal{P}$ if the following holds. Let $a_1, \ldots, a_n$ be the ancestor path of this state, then a node in an XML document with label a_n and the path of ancestors labeled $a_1, \ldots, a_{n-1}$ is relevant according to $\mathcal{P}$ by Definition 3.13.

Example 7.6 We consider the DTD-automaton from Figure 7.5, and the projection paths $\mathcal{P}_1 = \{$`/a/b#`, `//b#`, `/*`$\}$. Then all states in the DTD-automaton with the exception of q_0 describe nodes relevant according to $\mathcal{P}_1$. In comparison, if we consider the projection paths in $\mathcal{P}_2 = \{$`/a/b#,/*`$\}$, then only the states $q_1, \hat{q_1}$ and $q_2, \hat{q_2}$ describe nodes relevant according to $\mathcal{P}_2$. □

```
Given DTD-automaton $D$ and a set of projection paths $\mathcal{P}$,
(1) choose a subset $S$ of states as follows:
    (a) $S := \{q_0\}$;
    (b) for each state $q$ in the DTD-automaton do
          if $q$ describes nodes relevant according to $\mathcal{P}$
          then add $q$ to $S$ end if
        end for
    (c) for each pair of dual states $q$ and $\hat{q}$ in S do
          let $R$ contain all states $p$ of $D$ s.t.
            there is a path from $q$ to $\hat{q}$ via $p$ in $D$;
          if $R \subseteq S$ then remove states in $R$ from $S$ end if
        end for
    (d) while there are changes to $S$ do
          if $D$ has transitions $q \to q_1 \to \cdots \to q_n \to p$
             and $q \to p_1 \to \cdots \to p_m \to p'$ for $n, m \geq 0$,
             where only $q$ and $p$ are in $S$
             and $p$ and $p'$ have the same label
          then add the parent states of $p'$ to S end if
        end while
(2) compute the subgraph automaton $D_{|S}$;
(3) determinize $D_{|S}$ to obtain the runtime-automaton;
```

***Figure 7.6**: Compilation of the runtime-automaton.*

Figure 7.6 shows the algorithm for the static compilation of the runtime-automaton, which we illustrate in the following examples.

Example 7.7 We consider the DTD-automaton from Figure 7.5 and the projection paths $\mathcal{P} = \{$`/a/b#`, `/*`$\}$. In step (1) of the algorithm, we select a set of states S for computing a subgraph-automaton. In step (a) we initialize S with the initial state of the DTD-automaton. In step (b), S is extended by the states $q_1, \hat{q_1}, q_2$, and $\hat{q_2}$. This ensures that we visit all nodes that must be preserved for query evaluation. Step (c) does not apply here. In step (d), we again extend the state set. The reason is that we observe the transitions $q_1 \to q_2$ and $q_1 \to q_3 \to q_4$ where both q_2 and q_4 are $\langle b \rangle$-labeled, but only q_1 and q_2 are contained in set S. Consequently, the state set is extended by states q_3 and $\hat{q_3}$. This ensures that the runtime-algorithm is not thrown off-track when it skips parts of the input, as we exemplified in Section 7.2. Next, the subgraph-automaton is computed (see step 3). This yields the FSA from Figure 7.3. As this FSA is already deterministic, it is also the runtime-automaton. □

7.3.2 Frontier Vocabulary and Action Table

The runtime lookup-table containing the frontier vocabulary can be directly constructed from the transitions of the runtime-automaton. The definition of the action table is more interesting. We unambiguously map an action to each state, exploiting the fact that the runtime-automaton is homogeneous, as homogeneity is preserved by determinization via subset construction [25]. The entries for the action table T are derived from the path-matching automaton. States that do not describe nodes relevant to the projection paths are assigned action "`nop`". For all other states, we consider pairs of dual states q and $\hat{q}$, for reading the opening- and closing-tag of a node. If the ancestor path of these states defines a path in the path-matching automaton into a state with a projection path of the form $P \to \alpha \bullet \#$, then we assign $T[q]$ ="`copy on`" and $T[\hat{q}]$= "`copy off`". Otherwise, we assign the action "`copy tag`" for both states, possibly also copying the attributes for the opening tag (action "`copy tag + atts`"), depending on the matched projection paths.

7.3.3 Initial Jump Offsets

We discuss the computation of the jump tables, given a runtime-automaton A and its DTD-automaton A_D, based on the notion of safe jump distances. We assume that the input is valid w.r.t. DTD D.

Safe jumps. Let A_D be a DTD automaton with states p and q such that a transition $A[p, y] = q$ is defined. A jump distance $jd[p, q]$ from state p to state q is *safe* if the following holds. Let p be the current state in validating the input document against the DTD. Then we have reached state p by a transition $A[u, x] = p$. Let j be the current reading position, which is one position *past* token x. Let j' be the position *just before* reading y, then $j + jd[p, q] \leq j'$.

The intuition is that if we now skip the next $jd[p, q]$ positions, then we must guarantee that we do not jump past position j', lest we jump too far and miss token y when scanning for it from position $j + jd[p, d]$ in the input.

If function jd maps all pairs of states to distance zero, then this solution is safe (as we never jump too far) but not effective. Below, we adapt Dijkstra's shortest-paths algorithm for the purpose of computing jump distances that are as long as possible, but that are nevertheless still safe.

ij-Dijkstra. As weight function we regard the size of the shortest XML string defining a path from p to q. We must take into account that a sequence $\langle x \rangle \langle /x \rangle$ may be encoded as a bachelor tag $\langle x/ \rangle$. For the sake of simplicity, we describe the algorithm based on the length of tagnames alone. As outlined in our examples, required attributes may be factored in when computing initial jump offsets. This extension is merely technical, and hence not covered here.

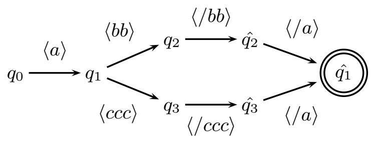
We assign a jump distance estimate $d[q]$ to each state q, which is initialized to the special symbol "∞". During the execution of the algorithm, the jump distance estimate decreases until the final jump distance has been determined. Additionally, we assign a backtrack entry $b[q]$ to each state q (initialized to zero), which contains the size of the label of state q. Recall that DTD-automata are homogeneous, so all incoming edges carry the same label.

A key operation in the Dijkstra algorithm is the relaxation applied to an edge from p to q. In relaxation, we test whether we need to consider shorter jump distance estimates from p to q, and consequently, update $d[q]$. The code in Figure 7.7 relaxes an edge $A_D[p, t] = q$. The figure also shows an adaption of Dijkstra's algorithm, where we compute the initial jump $ij[p, q]$ for all destinations q. We shall assume that for a positive integer k, "∞"$-k =$"∞".

```
**ij-Relax**(DTD-automaton $A_D$, state $p$, state $q$):
  find token $t$ such that $A_D[p,t] = q$ is defined;
  **if** $t = \langle/x\rangle$ and there is an edge $A_D[r, \langle x\rangle] = p$
  **then** $d[q] := \min(d[q],\ d[p] + 1)$;   $b[q] := |\langle/x\rangle|$
  **else**  $d[q] := \min(d[q],\ d[p] + |t|)$;  $b[q] := |t|$
  **end if**
```

```
**ij-Dijkstra**(DTD-automaton $A_D$, state $p$):
  **for each** state $q$ in $A_D$
  **do**  $d[q]$ := "$\infty$"; $b[q] := 0$ **end for**
  $d[p] := 0$;
  $Q$ is a priority queue holding all states in $A_D$
    keyed by their initial jump estimate $d$;
  **while** $Q$ not empty
    **do** remove $q$ from $Q$ s.t. $d[q]$ is minimal;
         **for each** edge $A_D[q,t] = u$ **do** ij-Relax($A_D$,$q$,$u$) **end for**
  **end while**
  **for each** state $q$ in $A_D$ **do** $ij[p,q] := d[q]$ - $b[q]$ **end for**
```

Figure 7.7: *Computation of initial jump offsets.*



(a) DTD-automaton.

state q	q_0	q_1	$\hat{q_1}$	q_2	$\hat{q_2}$	q_3	$\hat{q_3}$
$d[q]$	0	3	12	7	8	8	9
$b[q]$	0	3	4	4	5	5	6
$ij[q_0, q]$	0	0	8	3	3	3	3

(b) Computing initial jumps.

Figure 7.8: *Illustrations for Example 7.8.*

Example 7.8 We assume the DTD D shown below, and its DTD-automaton A_D from Figure 7.8(a).

```
<!DOCTYPE a [
   <!ELEMENT a (bb|ccc)>  <!ELEMENT bb EMPTY>  <!ELEMENT ccc EMPTY> ] >
```

By calling ij-Dijkstra(A_D, q_0), we compute the values shown in Figure 7.8(b). The rationale behind this table is the following. The initial jump between q_0 and q_1 is zero, as any larger jump could lead past the opening tag $\langle a\rangle$. When jumping from state q_0 towards state q_2, we must not skip more than three positions, which corresponds to the size of tag $\langle a\rangle$. Otherwise, we could miss the tag $\langle b\rangle$. To jump towards state $\hat{q_2}$, the jump is bounded by three positions as well, as the input could start with "$\langle a\rangle\langle b/\rangle$" and a jump beyond three positions could lead past the start position of the opening tag for the b-labeled node. Finally, to approach $\hat{q_1}$ without reading past the token $\langle/a\rangle$, the smallest XML snippet allowed by the DTD is "$\langle a\rangle\langle b/\rangle$", with a length of eight characters. □

Computing initial jump offsets. We compute the initial jumps given a runtime-automaton A and the DTD-automaton A_D for its DTD. First, we annotate each state in A with the set of possible matching states of A_D, denoted by $\alpha[q]$. This is done as follows. Let q_0 be the initial state of A and let p_0 be the initial state of A_D. We define $\alpha(q_0) = \{p_0\}$. Then for each state q of A where α has already been defined, and for each transition $A[q, t] = u$ for some t and u, we assign $\alpha(u)$ as states v from A_D such that there is a state $p \in \alpha(q)$, and u is reachable from p in A_D via a word wt, where w contains no symbol from $V[q]$.

Then, for each state p in A, let P be the set of states to which there is a transition from p. That is, for each $q \in P$, there is some token t such that $A[p, t] = q$ is defined. We compute the entry for $J[q]$ as $\min_{q \in P}(\{ij(a, b) \mid a \in \alpha(p), b \in \alpha(q)\})$, where $ij(a, b)$ is computed using ij-Dijkstra.

7.4 Prototype Implementation

We have implemented a prototype in C++, called *SMP*. As a technical simplification, we assume that no tags occur inside comments or CDATA-sections. Our prototype takes the projection paths and a non-recursive DTD as input. The datastructures for Boyer-Moore and Commentz-Walter string search are computed lazily at runtime, when a state of the runtime-automaton is first entered.

SMP is implemented as a streaming algorithm, where we scan the input with a fixed-size window in a single pass. We set the size of this window to eight times the systems page size. Within the window held in main memory, we can locally jump back and forth, all the while trying to quickly process the input by skipping characters. As our experiments confirm, this results in significant speedups for searching flat strings and XML data alike. In particular, both the runtime costs and the number of character comparisons of SMP are comparable with Boyer-Moore style string matching algorithms.

7.5 Experiments

Our testbed is a Core2 Duo IBM ThinkPad Z61p with a T2500 2.00GHz CPU, 1GB RAM, running Ubuntu Linux 6.06 LTS. We run Java query engines with J2RE 1.5.0_09. By *Usr*, we denote the total number of seconds used by the process used directly, and *Sys* denotes the CPU seconds used by the system on behalf of the process. We compute *CPU* workload as *Usr+Sys* divided by the total running time. To avoid warm caches, we alternately run experiments and load large dummy files into main memory.

The experiments are organized as follows. We first examine the performance characteristics of SMP on different datasets and query workloads. To this end, we run SMP as a stand-alone application. Later, we study how SMP performs in combination with in-memory XPath and XQuery engines. Finally, we contrast the throughput of SMP with that of an industrial-strength SAX parser, to demonstrate the overhead caused by input tokenization alone. We conclude with a comparison of SMP with an existing prefiltering tool that also exploits schema information, but relies on input tokenization.

7.5.1 SMP Performance Characteristics

We study the behavior of SMP for different queries, documents, and document sizes. In particular, we run experiments with the XMark [122], MEDLINE [105], and Protein Sequence [89] datasets, and their DTDs.

XMARK data. We test SMP with data from the XMark benchmark [122]. Note that the XMark DTD allows recursive lists within item descriptions. We have modified the DTD accordingly, and restrict our experiments to the XMark queries which do not address recursive lists (queries XM1–14 and XM17–20). Table 7.1 shows our results for a 5GB XMark document. To provide an idea of how SMP performs on smaller documents, we list the maximum deviation "±" (in positive or negative direction) on the 10MB, 100MB, and 1GB documents for selected values. We state the size of the projected document, and the maximum memory consumption (*Mem*). The total runtime (*Time*), the sum of *Usr* and *Sys* time, and the average *CPU* load are also listed. Time measurements include static analysis, which varies between 0.03s and 0.2s.

States is the number of states of the runtime-DFA. The value of *CW + BM* denotes the number of states for which Commentz-Walter (*CW*) or Boyer-Moore (*BM*) lookup-tables are constructed. For instance, for query XM1, the runtime-DFA has 9 states, two of which require *CW* lookup-tables, and six of which need *BM* lookup-tables.

When we scan the input, we distinguish the forward shifts performed in string pattern matching from the initial jump offsets computed by static analysis. *∅Shift Size* denotes the average size of forward shifts, which depends on the lengths of keywords in the frontier vocabularies. When we verify a potential match for a keyword, forward shifts are followed by a scan from right to left. Hence, *∅Shift Size* cannot be used to compute *Char Comp*, the percentage of character comparisons relative to the document size. *Initial Jumps* denotes the percentage of characters skipped by initial jump offsets alone. The small deviations (±) for different input sizes suggest that the XMark data generator creates documents that are very similar in their structure.

We observe that larger outputs go hand in hand with higher total processing times. For instance, prefiltering for query XM14 produces the largest output, and requires the longest running time. The *Usr+Sys* time is mainly driven by the number of character comparisons. The *CPU* load depends on the output size and the number of characters comparisons, and ranges between 11% and 21%. Thus, the system spends most of the time holding out for new data from the disk. The average size of forward shifts depends on the input and the size of the tags used in the projection paths. In evaluating query XM5, we observe comparatively large average forward shifts. Consequently, the *Usr+Sys* time is low, and SMP inspects only about 10% of the input (*Char Comp*). Overall, SMP inspects at most 23% of the input. Comparatively little can be gained by initial jump offsets in this set of experiments.

Note that queries XM2 and XM3 have identical projection paths, which is reflected in similar experimental results.

MEDLINE. We further consider the evaluation of the XPath expressions M1–5 from Figure 7.9 on 656MB of MEDLINE data [105]. In contrast to XMark, the MEDLINE data is not synthetic. To exclude trivial cases, all queries only

	XM1	XM2	XM3	XM4	XM5	XM6	XM7	XM8	XM9
Proj. Size	67.64MB	123.26MB	123.26MB	151.14MB	22.10MB	12.03MB	105.74MB	93.78MB	121.01MB
Mem	1.64MB	1.72MB	1.72MB	1.75MB	1.68MB	1.64MB	1.77MB	1.72MB	1.78MB
Time	252.48s	283.33s	281.8s	290.42s	252.35s	241.70s	256.94s	252.95s	258.93s
Usr+Sys	31.00s	41.65s	41.59s	42.40s	19.91s	29.36s	50.47s	35.91s	30.41s
CPU [%]	12.52±2.51	14.99±0.75	15.04±2.53	14.90±4.90	8.05±1.52	12.39±4.89	20.02±7.52	14.48±5.73	11.98±4.68
States (CW+BM)	9 (2 + 6)	11 (4 + 6)	11 (4 + 6)	13 (5 + 7)	9 (2 + 6)	7 (2 + 4)	11 (4 + 6)	15 (4 + 10)	25 (6 + 18)
∅ Shift Size [char]	5.72±0.02	7.62±0.01	7.62±0.01	7.65±0.01	10.83±0.04	5.17±0.00	6.55±0.04	7.42±0.00	7.50±0.05
Initial Jumps [%]	0.32±0.00	1.42±0.01	1.42±0.01	1.37±0.00	0.43±0.00	1.98±0.01	2.61±0.01	0.75±0.01	1.18±0.01
Char Comp. [%]	18.86±0.05	15.8±0.04	15.8±0.04	16.37±0.12	9.87±0.02	19.91±0.03	18.40±0.16	15.10±0.04	15.29±0.15

	XM10	XM11	XM12	XM13	XM14	XM17	XM18	XM19	XM20
Proj. Size	307.63MB	95.37MB	65.73MB	137.63MB	1357.28MB	75.44MB	21.08MB	71.22MB	38.52MB
Mem	1.96MB	1.74MB	1.74MB	1.66MB	1.64MB	1.67MB	1.69MB	1.66MB	1.67MB
Time	295.92s	256.54s	256.85s	250.35s	321.03s	255.94s	249.29s	243.67s	249.88s
Usr+Sys	54.94s	34.85s	32.40s	26.39s	53.71s	34.95s	23.54s	32.16s	31.67s
CPU [%]	18.93±2.73	13.85±4.47	12.86±2.14	10.75±3.25	17.07±2.93	13.92±1.89	9.63±0.83	13.45±1.78	12.92±4.59
States (CW+BM)	33 (10 + 22)	17 (5 + 11)	15 (5 + 9)	13 (2 + 10)	9 (2 + 6)	11 (3 + 7)	9 (3 + 5)	11 (2 + 8)	9 (3 + 5)
∅ Shift Size [char]	5.68±0.01	6.58±0.01	6.60±0.02	6.06±0.00	5.16±0.01	5.72±0.01	8.29±0.04	5.17±0.00	5.75±0.00
Initial Jumps [%]	0.16±0.01	1.85±0.01	2.00±0.00	0.13±0.00	1.35±0.01	0.32±0.00	0.80±0.01	1.64±0.01	0.59±0.01
Char Comp. [%]	22.38±0.01	17.15±0.11	16.81±0.11	17.17±0.03	21.24±0.08	18.99±0.03	12.95±0.03	20.57±0.03	18.67±0.03

***Table 7.1**: SMP benchmarks on XMark 5,000MB data.*

M1	/MedlineCitationSet//CollectionTitle
M2	/MedlineCitationSet//DataBank[DataBankName/text()="PDB"]/AccessionNumberList
M3	/MedlineCitationSet//PersonalNameSubjectList/PersonalNameSubject[
	LastName/text()="Hippocrates" or DatesAssociatedWithName="Oct2006"]
	/TitleAssociatedWithName
M4	/MedlineCitationSet//CopyrightInformation[contains(text(),"NASA")]
M5	/MedlineCitationSet/MedlineCitation[
	contains(MedlineJournalInfo//text(),"Sterilization")]/DateCompleted

Figure 7.9: *XPath queries over MEDLINE data.*

	M1	M2	M3	M4	M5
Proj. Size	0MB	0.42MB	0.34MB	0.19MB	47.4MB
Mem	1.94MB	2.01MB	2.11MB	1.99MB	2.00MB
Time	33.72s	33.62s	33.69s	33.47s	35.51s
User + Sys	2.96s	4.46s	3.02s	3.24s	4.35s
CPU [%]	9.02	13.76	9.26	9.99	12.43
States (CW + MB)	5 (1 + 1)	9 (3 + 5)	13 (4 + 4)	5 (2 + 2)	9 (3 + 5)
∅ Shift Size [char]	12.24	6.86	12.49	12.69	13.43
Initial Jumps [%]	0.00	0.00	0.00	0.01	7.61
Char Comp. [%]	8.37	14.63	8.4	8.52	9.81

Table 7.2: *SMP benchmarks on 656MB MEDLINE data.*

use paths that are satisfiable by DTD. Table 7.2 summarizes the results. Query M1 searches for nodes which are defined by the DTD, but do not occur in the input. Scanning with an average forward shift of about 12 characters, only 8.37% of all characters in the input are inspected.

In comparison to the XMark results in Table 7.1, XML prefiltering on MEDLINE data features larger average forward shifts (*∅Shift Size*), due to longer tagnames in the queries. For queries M1–M4, no significant initial jumps are possible. Upon closer inspection, it turns out that the MEDLINE DTD specifies many nodes as optional, and only required nodes can be considered for initial jumps. Yet we can observe initial jumps for query M5. In total, about 50MB of the input documents are skipped by initial jumps alone.

7.5.2 Prefiltering for XQuery and XPath Processors

We next examine how main-memory XML query engines perform when they are run in combination with SMP. We will see that the low CPU workload of SMP facilitates efficient pipelining of SMP with XPath and XQuery processors.

XQuery evaluation. We evaluate the XMark queries from before with the main memory-based XQuery processors QizX [90] and Saxon [93]. For the input tested, QizX was superior to Saxon regarding execution time and memory consumption, and so we only report on the results for QizX.

The runtime and main memory consumption are limited to one hour and 1GB. We study a sequential setup, where the prefiltered input is written back to disk. Then the total runtime includes the additional write and read.

Figure 7.10 shows the results. The graphs use log scale for the x-axis (document size) and the y-axis (time). We first discuss the runtimes for stand-alone

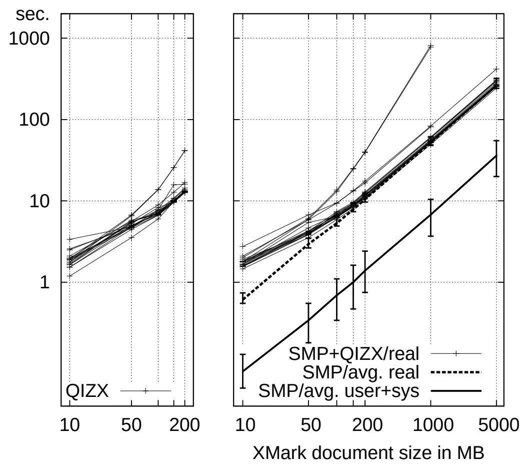


Figure 7.10: *SMP+QizX XQuery evaluation on XMark data.*

query evaluation, which are shown to the left. Without projection, QizX can successfully load the input and evaluate all queries within the memory and runtime limitations for documents up to 200MB, but fails for 1GB and 5GB documents. To the right, we plot the runtimes for evaluating SMP and QizX in sequence. On documents up to 200MB, the runtimes differ only marginally, due to the overhead of one extra write and read. When coupled with prefiltering, QizX can evaluate all queries for the 1GB document, and still 15 queries on the 5GB document. We can discern two outlier queries (XM11 and XM15) which close in on the timeout for 1GB, and fail for the 5GB document.

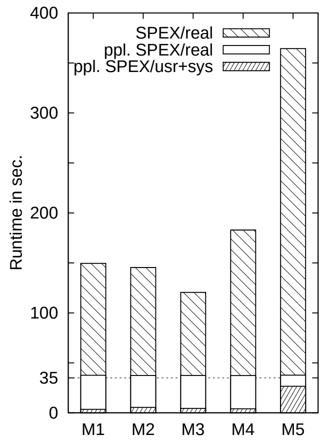
On the right, we further depict the average real time of SMP prefiltering, and the minimum and maximum values for all queries with error bars. The average user and system time is well below the real time, indicating that SMP predominantly waits for new input from the disk.

In summary, in-memory XQuery engines such as QizX can be made to scale to inputs in the Gigabyte range when coupled with SMP.

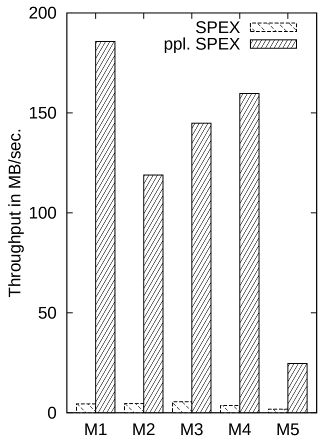
XPath in pipelining with SMP. The SPEX XPath evaluator [85] is a representative of a class of query engines such as XFilter, YFilter [35], and the XPush Machine [53], which were developed for XML stream processing. While the latter systems evaluate high workloads of queries, SPEX and SMP evaluate single queries. We expect similar results as for SPEX as when combining XFilter, YFilter, and the XPush machine with our prefiltering tool, and intend to set up additional experiments in future work.

In Figure 7.11(a), we show the runtimes for the XPath queries from Table 7.2 on MEDLINE data. We compare two scenarios. First, SPEX is run on the unprojected document, and next on the document projected by SMP, which is *piped* directly into SPEX (denoted “ppl. SPEX”).

It is remarkable that in the pipelined scenario, evaluation real time differs only marginally from the real time for prefiltering alone (see Table 7.2). In the plot, this is indicated by the 35 seconds line. In all cases, the computational effort needed for query evaluation mostly manifests in the SPEX *Usr+Sys* time



(a) Runtimes.



(b) Program throughput.

Figure 7.11*: SMP+SPEX XPath evaluation on 656MB Medline data.*

(see "ppl. SPEX/usr+sys"), rather than the real time. The projected output for query M5 is still of considerable size (47.4MB), which causes a comparatively high *Usr+Sys* time. For the other queries, SMP filters out large parts of the input, so SPEX *Usr+Sys* time is lower. We conclude that the low CPU load of SMP enables an effective interleaving of prefiltering and query evaluation.

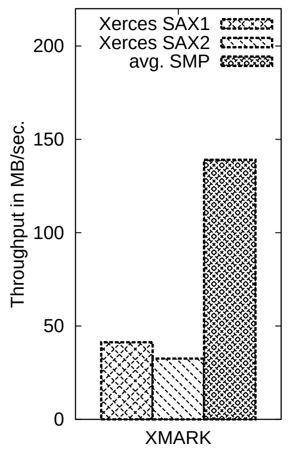
In Figure 7.11(b), we compare the throughput achieved by SPEX as a stand-alone tool with the pipelining of SMP and SPEX. In particular, we state the *program throughput*, which we derive based on the assumption that the disk is fast enough to warrant a CPU utilization of 100%. The differences in the throughput are significant, and we can reach up to 190 MB/sec for query M1. The achievable throughput for M5 is smaller, because the projected document is still large. Nevertheless, the pipelined setup is still superior.

7.5.3 XML Parsing and XML Document Projection

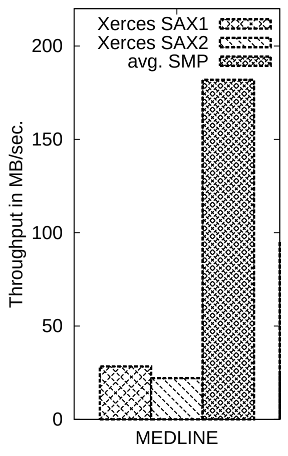
We show that on the queries and datasets studied here, SMP achieves a significantly higher throughput than industrial-strength SAX parsers. SAX parsers are used by virtually all competing approaches to tokenize the input, which suggests that these systems are inevitably inferior to SMP in terms of scalability.

Xerces. The existing approaches to XML prefiltering all process XML documents that are tokenized, e.g. by a SAX parser. We next compare the throughput of SMP and Xerces C++ [121], an efficient SAX parser. We have built a minimal application on top of the Xerces API that just parses the input into tokens. Xerces further checks the input for well-formedness by default.

The results for the XMARK (5GB) and MEDLINE datasets are visualized in Figure 7.12. The program throughput of tokenizing the input with Xerces (either using the SAX1 or the SAX2 reader) is well below the average throughput that SMP achieves in prefiltering the same data for the queries of Table 7.1 (XMark) and Table 7.2 (MEDLINE). Even pipelining SMP prefiltering and



(a) XMark data.



(b) MEDLINE data.

Figure 7.12: *Xerces vs. SMP: Program throughput in MB/sec.*

XPath evaluation with SPEX has a higher average throughput on MEDLINE data than just tokenizing the input with Xerces (c.f. Figure 7.11).

Overall, SMP is by a factor of three to nine faster than Xerces, while it performs a more complex data management task. The results confirm that the throughput achieved by our approach substantially surpasses that of projection systems that rely on a tokenization of their input.

Type-based projection. The type-based projection tool (TBP) [15] is a natural choice for comparison with SMP, as it also exploits schema knowledge, but tokenizes its complete input. To the best of our knowledge, TBP is the only operational publicly available projection tool. (We were not able to obtain a version of Galax with the projection feature mentioned in [77]). TBP performs a powerful static analysis to prefilter for additional XPath axes and even for predicates. Yet for the query workload considered here, the sizes of the projected outputs are comparable with SMP. The differences are mainly due to whitespace formatting by TBP, and the fact that SMP can sometimes discard more nodes (namely irrelevant ancestor nodes).

	Type-based Projection (OCaml)			SMP (C++)	
	Usr+Sys	Mem	Proj. Size	Mem	Proj. Size
M3	756.77s	3.36MB	26.52MB	1.72MB	24.62MB
M6	812.56s	3.36MB	2.59MB	1.64MB	2.40MB
M7	1170.03s	3.36MB	34.50MB	1.76MB	21.14MB
M19	1027.13s	3.36MB	17.92MB	1.65MB	14.23MB

Table 7.3: *Projection of 1,000MB XMark data.*

As TBP is written in OCaml, it is difficult to compare runtime results. We have tested both the byte code and the native code compilation of TBP. In the following, we only provide the results for the native code compilation, which turned out to be significantly faster. We consider the subset of XMark queries benchmarked both by SMP in this paper and by and TBP in [15], namely XM3, XM6, XM7 and XM19. Table 7.3 contains the results. Both XML filtering systems get by with an economical main memory consumption, yet the *Usr+Sys* times differ. To put our results into perspective, we point out that in computer language shootouts, OCaml programs rarely perform more than a factor of 20 worse than C++ programs compiled with *g*++ (see [5]). Typically, we may expect a gap of a factor five to ten. On the 1GB XMark document, SMP requires less than ten seconds *Usr+Sys* time (and 2MB main memory) for all queries. This is at least a factor of 90 better than the *Usr+Sys* time consumed by TBP. The program throughput of SMP is in the order of two magnitudes higher than that of TBP. This exceeds the difference one might expect by the choice of programming language. For the 10MB and 100MB documents we observe similar results, where our implementation is faster by at least a factor of 84. On the 5GB document, TBP even exceeds the time limit of one hour.

In summary, SMP is able to project the 5GB document faster than type-based projection can process 1GB. Also, SMP requires fewer CPU seconds (*Usr+Sys*) on the 5GB document than TBP needs for projecting 1GB.

7.6 Future Work

We next discuss ideas how runtime-automata may further be optimized. We show that a normalization of DTDs brings forward insights into the schema definition that generally make for leaner runtime-automata. We construct DTD-automata as Glushkov automata. As introduced in Section 3.1, Glushkov automata derived from regular expressions dedicate a state to each atomic symbol.

We propose to normalize the regular expressions in DTD-productions. We can rewrite all subexpressions A, B, and C by applying the rewriting rules below. A rule $\alpha \Rightarrow \beta$ describes that a subexpression matching pattern α is modified as specified by pattern β.

$$\texttt{(}A\texttt{|}B\texttt{),}C \Rightarrow \texttt{(}A\texttt{,}C\texttt{) | (}B\texttt{,}C\texttt{)}, \quad A\texttt{?} \Rightarrow A|\epsilon, \text{ and } A\texttt{+} \Rightarrow A\texttt{,}A\texttt{*}$$

Normalization may introduce ambiguities that are not allowed in DTDs [117]. As static analysis includes a determinization step, our analysis is oblivious to this detail. If we now perform static analysis given a normalized DTD, we may obtain an improved runtime-automaton, as motivated next.

Example 7.9 We compute the DTD-automaton D for the DTD with the productions `<!ELEMENT a(b?,c)>` `<!ELEMENT b(#PCDATA)>`, and `<!ELEMENT c(b?)>`, where a is the grammar start symbol.

Given the projection paths `/a/b` and `/*`, we compute the runtime-DFA $A(D)$ shown below. Due to determinization of the subgraph-automaton by subset-construction, the states of automaton $A(D)$ are defined as sets of states of the DTD-automaton D.

D :

$$\begin{tikzcd}
q_0 \arrow[r, "\langle a\rangle"] & q_1 \arrow[r, "\langle b\rangle"] \arrow[rrr, bend right, "\langle c\rangle"'] & q_2 \arrow[r, "\langle /b\rangle"] & \hat{q_2} \arrow[r, "\langle c\rangle"] & q_3 \arrow[r, "\langle b\rangle"] \arrow[rrr, bend right, "\langle /c\rangle"'] & q_4 \arrow[r, "\langle /b\rangle"] & \hat{q_4} \arrow[r, "\langle /c\rangle"] & \hat{q_3} \arrow[r, "\langle /a\rangle"] & \hat{q_1}
\end{tikzcd}$$

$A(D)$:

$$\begin{tikzcd}
\{q_0\} \arrow[r, "\langle a\rangle"] & \{q_1\} \arrow[r, "\langle b\rangle"] \arrow[rrr, bend right, "\langle c\rangle"'] & \{q_2\} \arrow[r, "\langle /b\rangle"] & \{\hat{q_2}\} \arrow[r, "\langle c\rangle"] & \{q_3\} \arrow[r, "\langle /c\rangle"] & \{\hat{q_3}\} \arrow[r, "\langle /a\rangle"] & \{\hat{q_1}\}
\end{tikzcd}$$

If we apply the DTD-normalization rules, the first DTD-production changes to `<!ELEMENT a((b,c)|c))`. Static analysis now produces the DTD-automaton D' and the runtime-automaton $A(D')$, as shown below.

$$\begin{tikzcd}
D' : & q_0 \arrow[r, "\langle a\rangle"] & q_1 \arrow[r, "\langle b\rangle"] \arrow[drrr, bend right, "\langle c\rangle"] & q_2 \arrow[r, "\langle /b\rangle"] & \hat{q}_2 \arrow[r, "\langle c\rangle"] & q_3 \arrow[r, "\langle b\rangle"] & q_4 \arrow[r, "\langle /b\rangle"] & \hat{q}_4 \arrow[r, "\langle /c\rangle"] & \hat{q}_3 \arrow[r, "\langle /a\rangle"] & \boxed{\hat{q}_1} \\
& & & & & q_5 \arrow[r, "\langle b\rangle"] & q_6 \arrow[r, "\langle /b\rangle"] & \hat{q}_6 \arrow[r, "\langle /c\rangle"] & \hat{q}_5 \arrow[ur, bend right, "\langle /a\rangle"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
A(D') : & \{q_0\} \arrow[r, "\langle a\rangle"] & \{q_1\} \arrow[r, "\langle b\rangle"] \arrow[rrrrr, bend right, "\langle c\rangle"] & \{q_2\} \arrow[r, "\langle /b\rangle"] & \{\hat{q}_2\} \arrow[r, "\langle /a\rangle"] & \boxed{\{\hat{q}_1\}} & \{\hat{q}_5\} \arrow[l, "\langle /a\rangle"'] & \{q_5\} \arrow[l, "\langle /c\rangle"']
\end{tikzcd}$$

The optimized runtime-DFA $A(D')$ distinguishes two cases. Assume we are in state q_1. If the opening tag of a b-labeled node is encountered before an opening tag for a c-labeled node, then we can locate the data for this node and output it. Afterwards, we need not locate the tokens for the c-labeled node, as is enforced by the unoptimized runtime-automaton $A(D)$. We thus make fewer "stopovers" when processing the input at runtime. □

In general, we may assume that reducing the number of stopovers in the runtime algorithm also improves the runtime performance. Yet without statistics on the distribution of characters in the input, it is not generally clear that the optimized automaton is indeed superior. We must therefore resort to heuristics, which we could base on experiences gained with established string matching algorithms. First, we may presume that the search for longer tagnames is more efficient. Second, we may presume that performance in string matching degenerates if keyword sets contain at least one short keyword. Based on such heuristics, we can decide whether to modify the runtime-automaton or not.

We regard the optimization of runtime-automata based on heuristics an interesting starting point for future work.

7.7 Conclusion

We have shown that established string matching techniques can be employed to efficiently implement XML document projection. This significantly increases the throughput of XPath and XQuery engines when compared to existing prefiltering approaches. The benefit of our technique becomes even more striking in pipelined scenarios, where the low CPU load of our projection tool makes an interleaving of XML prefiltering and query evaluation possible.

8 Static Operator Scheduling in Event-based Query Processing

In this chapter, we introduce an extension of the XQuery language, called *FluX*. As motivated in Section 6.2, FluX supports *event-based* query processing. Purely event-based queries of this language can be executed on streaming XML data in a very direct way, without buffering data first. We motivate the FluX language in Section 8.1, and provide a formal definition in Section 8.2. In Section 8.3, we develop an algorithm that efficiently rewrites XQueries into the event-based FluX language. This algorithm uses order constraints from a DTD to schedule event handlers and to thus minimize the amount of buffering required for evaluating a query. If a DTD is not available, we assume that no no order constraints hold. We discuss the various technical aspects of building an XQuery engine that is based on the FluX language in Section 8.4. This is complemented with an experimental evaluation of our approach in Section 8.5. In Section 8.6, we touch on ideas for extending FluX, and conclude this chapter.

8.1 Motivation

We present principled work on query optimization in the framework of XQuery which honors the special features of XML stream processing. This is the first framework for algebra-based optimization of queries on XML streams hat uses dedicated query operators that capture the spirit of stream processing, and which allows for query optimization using schema information. However, there are XQuery algebras meant for conventional query processing [45,116], and there is work on applying them in the streaming context [43]. In systems developed in succession to our work [70], parts of XQueries are evaluated directly on XML streams, also based on static query analysis [44, 75, 101]. Moreover, the problem of optimizing XQueries using a set of constraints holding in the XML data model, rather than a schema, was addressed in [34, 75].

To our knowledge, this is the first work on optimizing XQuery using schema constraints derived from DTDs. A main strength of our approach is its extensibility. Even though space limitations require us to restrict our discussion to the XQuery fragment XQ^- (see Section 3.4), our results can be generalized to even larger fragments.

We next propose the *FluX* query language, which extends XQuery by a new construct for event-based query processing called “process-stream”. FluX motivates a very direct mode of query evaluation on data streams (similar to query evaluation in XQRL [49]), and provides a strong intuition for which main memory buffers are needed in which queries. This allows for a strongly *buffer-conscious* mode of query optimization. The main focus of this chapter is on automatically rewriting XQueries into event-based FluX queries, and at the same time optimizing (reducing) the use of buffers using schema information.

XQuery and FluX. We have already sketched core ideas of FluX queries in Chapter 6.2. In the following discussion, we additionally make use of schema information, which leads to a larger set of constructs in the FluX language. Consider the following XQuery in a bibliography domain, taken from the XML Query Use Cases (see XMP Q3 in [123]) with minor adaption:

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

For each book in the bibliography, this query lists its titles and authors, grouped inside an element labeled “result”. Note that the XQuery language requires that within each result node, all titles are output before all authors.

When we evaluate this query on XML streams without schema knowledge, we cannot assume a particular order between nodes in the input. Yet even then, we can evaluate parts of the query directly on the stream. We can output the title-children inside a book-node as soon as they arrive on the stream. However, we must delay the output of the authors using a memory buffer until we reach the closing tag of the book-node. At that time, no further title-nodes can be encountered. Then we may retrieve and purge author-nodes from the buffer, and later refill the buffer with the author-nodes from the next book. We thus only need to buffer the authors of one book-node at a time, but not the titles.

Prior to this work, main memory-based XQuery engines did not exploit this fact, and rather buffered either the entire book nodes or used XML document projection to only buffer the data relevant to query evaluation. In particular, previous frameworks did not provide any means of making this seeming subtlety explicit, and for reasoning about it.

The “process-stream”-construct of FluX can express precisely the mode of query execution just described. The XQuery from before is then phrased as a FluX query[1] as shown in Figure 8.1.

A “process-stream $\$x$”-expression consists of a number of *handlers* which process the children of the context node of variable $\$x$ from left to right. An “on a”-handler fires on each child labeled a that is visited during such a traversal, executing the associated query expression. In the “process-stream $book” expression from Figure 8.1, the “on-first past(title,author)”-handler fires exactly once, as soon as the DTD implies for the first time that no further author- or title-node can be encountered among the children of a book. In the subexpression of the “on-first past(title,author)”-handler, we may freely use paths of the form $book/author or $book/title, because the nodes matching these paths cannot be encountered in the input stream anymore. Rather, we may assume that

[1]At this point, we slightly generalize the FluX syntax and allow expressions of the form $\langle t\rangle\alpha\langle/t\rangle$, where α is a “process-stream”-expression to improve the readability.

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

Figure 8.1: *FluX query compiled without schema information.*

the query engine has already buffered these matches for us. (For the rationale of buffer management in FluX query evaluation, we refer back to Chapter 6.)

Safe FluX queries. Informally, we call a query *safe* if it is guaranteed that XQuery subexpressions do not refer to paths that may still be encountered in the stream. The FluX query from before is safe. The for-loop employs the $book/author path, but is part of an on-first handler that cannot fire before all author-nodes relative to variable $book have been seen.

Let us now assume that the input stream adheres to the following DTD.

```
<!DOCTYPE bib [
    <!ELEMENT bib  (book*)>
    <!ELEMENT book ((title|author)*,price)>
    <!ELEMENT title  (#PCDATA)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT price  (#PCDATA)>   ]>
```

We again consider the FluX query from Figure 8.1, but we assume that the for-loop over authors is modified such that it iterates over book-children labeled “price”. Then this FluX query is not safe. On the firing of “on-first past(title,author)”, the price-nodes have not yet been encountered in the input stream. Consequently, no price nodes can be buffered. When the for-loop over price-nodes is evaluated, an incorrect query result is produced.

We return to the XQuery stated the beginning of this section. This query can be processed more efficiently with a schema as used in the XMP XML Query Use Cases [123], with the book-production shown below.

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

Then no buffering is required at all, because the DTD asserts that for each book, the title occurs before any authors. We denote this as an order constraint $Ord_{book}(title, author)$, as introduced in Definition 3.3. In Figure 8.2, we phrase our query in FluX such that titles and authors are directly copied to the output as they arrive on the stream.

We introduce the FluX query language in the next section. We formally define the *safe* FluX queries (under a given DTD), which are those FluX queries in which XQuery subexpressions have the usual semantics. Moreover, XQuery

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

Figure 8.2: *FluX query compiled with schema information.*

subexpressions are never executed before the data items referred to have been fully read from the stream, and may be assumed available in main memory buffers. We present an algorithm that statically schedules XQueries on streams using DTDs and that transforms them into optimized FluX queries. Further, we discuss the realization of query engines for FluX and their runtime buffer management, topics that have also been addressed in Chapter 6. We have built a prototype FluX query engine which we introduce as the *FluXQuery* engine, and which we evaluate by means of a number of experiments.

8.2 The FluX Query Language

In defining the FluX query language, we limit ourselves to queries from the XQuery fragment XQ^-. In evaluating queries over streams, we further assume a new semantics for treating strings inside queries. For example, the string "$\langle$*hello*$\rangle$" is considered valid here, but not in standard XQuery. The query "`<result>{/bib/book}</result>`" is understood in standard XQuery as a result-node with an embedded query to produce its children. In the context of XML streams, the same query is read as a sequence of three queries which output the string "$\langle$*result*$\rangle$", the /bib/book subtrees, and finally the string "$\langle$/*result*$\rangle$".

This subtlety is convenient in XML stream processing, where we output the tokens for matching tags independently. This alternative semantics is only used *internally* by the query engine, while users are oblivious to this fact. Users simply formulate queries in standard XQuery and assume the usual semantics.

8.2.1 Simple XQueries

We introduce a syntactic property of XQuery expressions that we call *simplicity.* The intuition is that output for simple XQuery subexpressions can be produced without buffering additional data. Let us motivate the notion of simple queries. Given an XQuery for-loop "for $\$x$ in $\$y/a$ return α", the question is whether the translation into a FluX expression "ps $\$y$: on a as $\$x$ return α" yields an equivalent query. FluX "on"-handlers are evaluated directly on the input stream. So at runtime, we have just read the opening tag "$\langle a\rangle$" in the input stream, and have bound the query-variable $\$x$ to this node. The question is now whether subexpression α can be evaluated immediately, without buffering any data in addition to what has already been read in the input stream so far. In particular, we want to answer this question without consulting schema information or performing non-local query analysis.

For instance, the XQuery subexpression "$\langle b\rangle 5\langle /b\rangle$" in place of α is simple, as it merely outputs a fixed string and does not require any buffering. Also, if $\alpha = \$x$ then this expression can be evaluated by directly copying the subtree of the node bound by $\$x$ from the input to the output. This requires no buffering, hence $\alpha = \$x$ is simple.

An expression $\alpha =$ "if χ then $\langle b/\rangle$" is simple provided that condition χ can be checked while the input is read. In contrast, an expression $\alpha = (\$x, \$x)$ is never simple. For a node to be output twice, it must be buffered.

We formulate a syntactic definition of simplicity of XQuery expressions. Note that this definition is not sufficient to determine whether a query subexpression can be evaluated directly on the input stream. For this purpose, we introduce the notion of query safety further below.

Definition 8.1 An XQuery expression is *simple* if it is the form "$\alpha\ \beta\ \gamma$" where

- α and γ are possibly empty sequences of strings and of expressions of the form " {if χ then s}", where χ is a condition and s is a string,
- β is either empty, "{$\$u$}", or "{if χ then $\$u$ }", for some variable $\$u$ and some condition χ, and
- if β is of the form "{$\$u$}", or "{if χ then $\$u$ }", then no atomic condition that occurs in $\alpha\,\beta$ contains the variable $\$u$. □

8.2.2 FluX Syntax and Semantics

We next define the syntax and semantics of the FluX language. We will use DTDs to extract order constraints. Order constraints for regular expressions were introduced in Section 3.1. In the following, we overload the meaning of a query-variable $\$x$ bound to an element node labeled a, by writing $\$x$ when we actually mean the DTD production identified by the element a. For example, if the DTD contains a production `<!ELEMENT` a ρ`>` for a regular expression ρ, we write $\mathit{Ord}_{\$x}(c, d)$ instead of $\mathit{Ord}_{\rho}(c, d)$, and we write $\mathit{symb}(\$x)$ instead of $\mathit{symb}(\rho)$. This static mapping from query-variables to DTD productions is straightforward for our XQuery fragment XQ^{-}, as we restrict ourselves to node tests for tagnames only, and as we do not allow the XPath descendant axis.

If no DTD is available, we simply assume that no order constraints hold.

Definition 8.2 The class of FluX expressions is the smallest set of expressions that are either *simple* or of the form

$$s\ \ \{\ \text{process-stream}\ \$y\text{:}\ \zeta\ \}\ \ s'$$

where s and s' are possibly empty strings, $\$y$ is a variable, and ζ is a comma-separated list of one or more *event handlers*. Each event handler is of one of the following two types,

1. a so-called "on-first"-handler of the form "on-first past(S) return α" where $S \subseteq \mathit{symb}(\$y)$ and α is an XQuery expression, or
2. a so-called "on"-handler of the form "on a as $\$x$ return Q" where $\$x$ is a variable, a is a tagname, and Q is a FluX expression. □

We will use "ps" as a shortcut for "process-stream", "on-first past(*)" as an abbreviation for "on-first past($symb(\$y)$)", and "on-first past()" in place of "on-first past($\emptyset$)".

Semantics of FluX. We evaluate a FluX expression "process-stream $\$y$: ζ" as follows. An event-handling statement considers the children of the node currently bound by variable $\$y$ as a list (or stream) of nodes and processes this list one node at-a-time. On processing a node v with children $t_1, \ldots, t_n$, where the labels of t_i are denoted as $label(t_i)$, we proceed as follows. For each i from 0 to $n+1$, we scan the list of event handlers $\zeta = \zeta_1, \ldots, \zeta_m$ once from the beginning to the end. In doing so, we test for each event handler ζ_j whether its event condition is satisfied, in which case the event handler ζ_j *fires* and the corresponding query expression is executed:

- A handler "on a as $\$x$ return Q" fires if $1 \leq i \leq n$ and $label(t_i) = a$.
- A handler " on-first past(S) return α" fires if $0 \leq i \leq n$ and further $first\text{-}past_{\$y,S}(label(t_1) \ldots label(t_i))$ is true. That is, for the first time while processing the children of $\$y$, no symbol of S can be encountered anymore in the input stream, or if $i = n+1$ and this event handler has not fired in any of the previous scans.

In summary, it is well possible that several events fire for a single node, in which case they are processed in the order in which the handlers occur in ζ. During the run on $t_1, \ldots, t_n$, each "on"-handler may fire zero up to several times, while each "on-first"-handler is executed exactly once.

FluX queries. In Section 3.4, we have introduced the notion of bound and free variables in XQueries. We extend this concept to FluX queries, where $freeVar(\{\text{process-stream } \$y\text{: } \zeta\})$ consists of the variable $\$y$, for each event handler in ζ of the form "on-first past(S) return α" also of the variables in $freeVar(\alpha)$, and likewise for each event handler in ζ of the form "on a as $\$x$ return Q" of the variables in $freeVar(Q) \backslash \{\$x\}$.

Note that expressions of the form "for $\$x$ in $\$y/a$ return α" and event handlers of the form "on a as $\$x$ return Q" bind the variable $\$x$, and hence remove it from the free variables of the super-expressions.

Definition 8.3 A FluX *query* is a FluX expression in which all free variables except for the special variable \$root corresponding to (the root of) the document are bound. □

8.2.3 Safe FluX Queries

We next define the notion of *safety* for FluX queries. Informally, a query is called *safe* for a given DTD if it is guaranteed that XQuery subexpressions do not refer to paths that might still be encountered in an input stream compliant with the given DTD. Let us introduce some terminology first.

In analogy to XQuery, we introduce the notion of parent variables of FluX expressions. By the *parent variable* of a FluX expression α within a FluX query, denoted *parentVar*(α), we refer to the variable bound by the closest super-expression of α, or \$root if no such variable exists.

For FluX or XQuery 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. An XQuery subexpression α of a FluX expression Q is called *maximal* if there is no XQuery expression β with $\alpha \prec \beta \preceq Q$. Note that a FluX query may contain several such maximal expressions.

We refer to normalized conditions of the form "*operand RelOp operand*" as *atomic conditions*. Given a variable \$$x$ and a FluX expression α, we identify a set of tagnames as the *dependencies* of \$$x$ in α.

$$\begin{aligned} \mathit{dependencies}(\$x,\alpha) = \{a \in \mathit{Tag} \mid \text{there exists } \$y \in \mathit{boundVar}(\alpha) :\\ (\mathit{lineage}(\$x,\$y) = \$x/a/\pi \text{ or } \mathit{lineage}(\$x,\$y) = \$x/a)\}\\ \cup \{a \in \mathit{symb}(\$x) \mid \$x \text{ occurs in an atomic condition in } \alpha\} \end{aligned}$$

Definition 8.4 A FluX query Q is called *safe* w.r.t. a given DTD iff for each subexpression "{ps \$$y$: ζ }" of Q, the following two conditions are satisfied:

1. For each handler "on-first past(S) return α" in the list ζ of handlers, the following is true:
   - $\forall\, b \in \mathit{dependencies}(\$y, \alpha)$ we have: $b \in S$ or ex. $a \in S$ s.t. $\mathit{Ord}_{\$y}(b, a)$,
   - $\forall\, \$z \in \mathit{freeVar}(\alpha)$ s.t. $\{\$z\} \preceq \alpha$ or $\{\$z/\pi\} \preceq \alpha$ (for some π) we have: $\$z = \$y$ and $\forall\, b \in \mathit{symb}(\$y)$: $b \in S$ or ex. $a \in S$ s.t. $\mathit{Ord}_{\$y}(b, a)$.
2. For each handler "on a as \$$x$ return β " in the list ζ, and for each maximal XQuery subexpression α of β, the following is true:
   - $\forall\, b \in \mathit{dependencies}(\$y, \alpha)$ we have: $\mathit{Ord}_{\$y}(b, a)$, and
   - if $\alpha = \beta$, then for all \$$u$ s.t. $\{\$u\} \preceq \alpha$ we have[2]: $\$u = \$x$:

□

It can be shown that this notion of *safety* is sufficient to ensure that main memory buffers are fully populated when they are accessed in query evaluation. Hence, a FluX query can be evaluated in a straightforward way on input streams compliant with the DTD.

Examples of safe FluX queries can be found in Chapter 6 and earlier in this chapter. The following example shows a query that is not safe.

Example 8.1 The FluX query in Figure 8.3 is not safe for a DTD with the production `<!ELEMENT book(title,author*,year)>`. This query resembles the first FluX query from Section 8.1, but the "on-first past"-handler only considers title-nodes, not author-nodes.

Now if a book-node is read in the input, the "on-first past(title)"-constraint is satisfied once its title is encountered. Yet at this time, the author-nodes for this book have not even been read yet. Consequently, evaluating the for-loop over authors yields an empty result. □

[2]Note that according to Definition 8.2, expression α must be simple in this case.

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

Figure 8.3: *A FluX query that is not safe for $Ord_{book}(title, author)$.*

8.3 Translating XQuery into FluX

In this section we address the problem of rewriting a query of our XQuery fragment XQ^- into an equivalent safe FluX query. The FluX extensions manage the event based, streaming execution of the query. In contrast, all XQuery expressions denote parts of the query that are evaluated over buffered data.

As the following example shows, every XQuery query can be transformed into a FluX query in a straightforward way.

Example 8.2 Every XQuery query α is equivalent to the FluX query

{ `ps $root:  on-first past(*) return` α }

Then the complete query is evaluated over buffered data. In particular, query evaluation sets in after the complete input has been read. This obviously gives the buffer-manager enough time to buffer all relevant data. Essentially, this corresponds to the two-phase XQuery evaluation of loading and query evaluation, as sketched in Section 5.2. □

Depending on our assumptions about the input, an XQuery can be transformed into an equivalent FluX query that can be evaluated more efficiently. To this end, we proceed in two steps. First, we transform the given XQuery query into an equivalent query in *FluX normal form.* Based on schema knowledge, this normalized query is then rewritten into an equivalent and safe FluX query.

8.3.1 FluX Normal Form

We transform XQueries into FluX normal form by applying the XQuery normalization rules from Figure 3.4 and further the rules from Figure 8.4 to push if-conditions inside for-loops and sequences.

An XQuery in FluX normal form does not contain any *conditional* for-loops, as normalization pushes conditionals inside the innermost for-loops. Moreover, for each subexpression of the form "if χ then α else β", α is either a fixed string or of the form "$\$x$", and β is always the expression "( )".

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

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

$$\frac{\text{if } \chi \text{ then (for } \$x \text{ in } \$y/\pi \text{ return } \alpha \text{) else ( )}}{\text{for } \$x \text{ in } \$y/\pi \text{ return (if } \chi \text{ then } \alpha \text{ else ( ))}}$$

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

$$\frac{\text{if } \chi \text{ then (if } \chi' \text{ then } \alpha \text{ else } \beta) \text{ else ( )}}{(\text{ if } (\chi \text{ and } \chi') \text{ then } \alpha \text{ else ( ), if } (\chi \text{ and not}(\chi')) \text{ then } \beta \text{ else ( ))}}$$

***Figure 8.4**: FluX normalization rules*

```
<bib>
{ for $bib in /bib return
    for $b in $bib/book
    where ( $b/publisher
            = "Addison-Wesley"
            and $b/year = "1991" )
    return <book>
             { $b/year  }
             { $b/title }
           </book> }
</bib>
```

(a) XQuery Q_1.

```
<bib>
{ for $bib in /bib return
    for $b in $bib/book return
     ( if χ then <book> else ( ),
       for $year in $b/year return
         if χ then $year else ( ),
       for $title in $b/title return
         if χ then $title else ( ),
       if χ then </book>  else ( )) }
</bib>
```

(b) XQuery Q_1 in FluX normal form.

***Figure 8.5**: Queries in FluX normal form.*

Example 8.3 Consider XQuery Q_1 from Figure 8.5(a), which is based on the XQuery Use Cases [123]. This queries returns books published by Addison-Wesley in 1991, including their year and title. We abbreviate the where-condition of this query as χ. Then the query Q_1' in Figure 8.5(b) is a FluX normal form of query Q_1. □

8.3.2 The FluX Compilation Algorithm

To formulate the algorithm for the static compilation of XQueries into FluX, we need some further notation. Let Σ be the set of tag names occurring in a given DTD. Let $\perp$ denote an empty list of event handlers. Given a list ζ of event handlers, we inductively define the set *hsymb*(ζ) of handler symbols for which an "on"-handler or an "on-first"-handler exists in ζ, see Figure 8.7.

The compilation algorithm is shown in Figure 8.6. This algorithm uses order constraints, as can be extracted from a DTD. If no DTD is available, we simply assume that no order constraints hold. Given an XQ^- XQuery Q in FluX normal form, we obtain a FluX query by evaluating "rewrite($root, $\emptyset$, Q)".

```
rewrite(Variable parentVar, Set⟨$\Sigma$⟩ $H$, XQuery $\beta$)
  let $\$x$ = parentVar;
  if $\{\$x\} \preceq \beta$ then
    if $\beta$ is simple and $dependencies(\$x, \beta) = \emptyset$ then
      return $\beta$
    else
      return { ps $\$x$: on-first past($*$) return $\beta$ }
    end if
  else /* $\{\$x\} \npreceq \beta$ */
    if $\beta = \beta_1\ \beta_2$ then
      $\beta_1'$ := rewrite(parentVar, $H$, $\beta_1$);
      match $\zeta_1$ such that $\beta_1'$ = { ps $\$x$: $\zeta_1$ };
      $\beta_2'$ := rewrite(parentVar, $H \cup hsymb(\zeta_1)$, $\beta_2$);
      match $\zeta_2$ such that $\beta_2'$ = { ps $\$x$: $\zeta_2$ };
      return { ps $\$x$: $\zeta_1$, $\zeta_2$ }
    else if $\beta$ is simple then
      /* e.g. $\beta$ is of the form $s$ or { if $\chi$ then $s$ } */
      return { ps $\$x$: on-first past($dependencies(\$x, \beta) \cup H$) return $\beta$ }
    else if $\beta$ is of the form { for $\$y$ in $\$z/a$ return $\alpha$ } then
      if $\$z \neq \$x$ then
        return { ps $\$x$: on-first past($dependencies(\$x, \alpha) \cup H$) return $\beta$ }
      else
        $X := \{b \in dependencies(\$x, \alpha) \cup H \mid \neg Ord_{\$x}(b, a)\}$;
        if $X \neq \emptyset$ then
        return { ps $\$x$: on-first past($X \cup \{a\}$) return $\beta$ };
        else
          $\alpha'$ := rewrite($\$y$, $\emptyset$, $\alpha$);
          return { ps $\$x$: on $a$ as $\$y$ return $\alpha'$ }
        end if
      end if
    end if /* if $\beta$ is for-expression */
  end if /* else $\{\$x\} \npreceq \beta$ */
```

Figure 8.6: Algorithm: Compiling XQuery into FluX.

$$\begin{array}{rcl}
hsymb(\bot) & := & \emptyset \\
hsymb(\zeta; \text{on } a \text{ as } \$x \text{ return } \alpha) & := & hsymb(\zeta) \cup \{a\} \\
hsymb(\zeta; \text{on-first past}(S) \text{ return } \alpha) & := & hsymb(\zeta) \cup S
\end{array}$$

Figure 8.7: Definition of handler symbols.

This algorithm is designed to produce a FluX query which (1) is safe w.r.t. the given DTD, (2) which is is equivalent to the input XQuery on all XML documents compliant with the DTD, and (3) which minimizes the amount of buffering needed for evaluating the query.

To meet goals (1) and (2), the particular order of the if-statements in the algorithm is crucial. Also, a set H of handler symbols must be passed on in recursive calls of the algorithm. One important construct for meeting goal (3) is the case distinction starting in line 19, where an "on"-handler is created provided that this is safe, and an "on-first"-handler is created otherwise.

8.3.3 Examples

We discuss the effect of our rewrite algorithm on sample queries. Queries borrowed from the XQuery Use Cases [123] are modified to work without attributes.

Example 8.4 ([123], XMP, Q2) Let us consider the XQuery Q_2, which creates a flat list of all the title–author pairs, with each pair enclosed in a result-element. We directly state the normalization Q_2' of this query.

```
1   <results>
2   { for $bib in /bib return
3       for $b in $bib/book return
4         for $t in $b/title return
5           for $a in $b/author return
6             <result> {$t} {$a} </result> }
7   </results>
```

When given a DTD that does not impose any order on book titles and authors, "rewrite($root,∅,$Q_2'$)" proceeds as follows: First, query Q_2' is decomposed into two subexpressions β_1, consisting of line 1, and β_2, consisting of lines 2–7 in the query. Then, the rewrite algorithm is recursively called for β_1 and for β_2. As β_1 is *simple*, the call for β_1 produces the result shown below.

```
{ ps root: on-first past() return <results> }
```

The call for β_2 decomposes β_2 into two subexpressions. Subexpression β_{21} consists of lines 2–6, and subexpression β_{22} consists of line 7 of Q_2'. The recursive call "rewrite($root,∅,$\beta_{21}$)" executes lines 26 and 27 of the algorithm in Figure 8.6, because β_{21} is a for-loop with parent variable $root and associated set $X = X_{\beta_{21}} = \emptyset$. Then the result

{ ps $root: on bib as $bib return α_1' }

is produced, where α_1' is computed by function call "rewrite($bib,∅,$\alpha_1$)", for the subquery α_1 of Q_2' in lines 3–6. This recursive call again executes lines 26 and 27 of the algorithm, and yields the expression $\alpha_1' =$

{ ps $bib: on book as $b return α_2' }

where α_2' is the result of "rewrite($b,∅,$\alpha_2$)" for the subquery α_2 of Q_2' in lines 4–6. As α_2 is a for-loop with parent variable $b and associated set $X = X_{\alpha_2} = \{author\}$, line 23 of the algorithm is executed, which produces expression $\alpha_2' =$

```
{ ps $b: on-first past(author,title) return α₂ }.
```

All in all, "rewrite(\$root,$\emptyset$,Q'_2)" returns the following FluX query F_2:

```
1  { ps $root:
2     on-first past() return <results>,
3     on bib as $bib return
4      { ps $bib: on book as $b return
5         { ps $b: on-first past(author,title) return
6                    for $t in $b/title return
7                      for $a in $b/author return
8                        <result> {$t} {$a} </result> } },
9     on-first past(bib) return </results> }
```

We again will refer to the "{ps \$b$\cdots$}"-expression in lines 5–8 of F_2 as α'_2. When evaluating the query F_2, the XQuery inside α'_2 will be evaluated once *all* author- and *all* title-nodes have been encountered. This gives the buffer manager enough time to buffer all relevant data.

Let us now consider the case where we are given a DTD with the production `<!ELEMENT book (author*,title*)>` where the constraint $Ord_{\texttt{book}}(author, title)$ holds. While running "rewrite(\$root,$\emptyset$,Q'_2)", we encounter the situation where $X = X_{\alpha_2} = \emptyset$ (rather than $X = \{author\}$, as we had before). Therefore, when processing the recursive call "rewrite(\$b,$\emptyset$,α_2)", lines 26–27 of the algorithm are executed, producing the result $\alpha''_2 =$

```
{ ps $b: on title as $t return
 { ps $t: on-first past(*) return
            for $a in $b/author return
              <result> {$t} {$a} </result> } }
```

Now, "rewrite(\$root,$\emptyset$,Q'_2)" yields query F'_2, which differs from F_2 in the lines 5–8, which must be replaced by the above expression α''_2.

When evaluating F'_2 on an XML document compliant with the second DTD, all author-nodes arrive before title-nodes, and can meanwhile be buffered. Encountering a title-node in the input stream invokes the following actions. The statement "on-first past(*)" delays the execution until the complete title-node has been seen. We may assume that the value of this particular node has been buffered. We can iterate over the buffer containing all collected author-nodes, each time writing the buffered titles and the current author to the output.

In contrast to the worst-case scenario above, we only buffer one title at-a-time in addition to the list of all authors. If there is more than one title, this strategy is clearly preferable. □

Example 8.5 ([123], XMP, Q1) Let us consider the query Q_1 and its normalization Q'_1 from Example 8.3. Given a DTD that does not impose any order constraints, such as a DTD with productions

`<!ELEMENT bib (book)*>` and `<!ELEMENT book (title|publisher|year)*>`

and the grammar start symbol "bib", the function call "rewrite(\$$root, \emptyset, Q'_1$)" rewrites Q'_1 into the following FluX query F_1:

```
 1  { ps $root:
 2      on-first past() return <bib>,
 3      on bib as $bib return
 4       { ps $bib: on book as $b return
 5          { ps $b:
 6              on-first past(publisher,year) return
 7                 if χ then <book> else ( ),
 8              on-first past(publisher,year) return
 9                 for $year in $b/year return
10                   if χ then $year else ( ),
11              on-first past(publisher,year,title) return
12                 for $title in $b/title return
13                   if χ then $title then ( ),
14              on-first past(publisher,year,title) return
15                   if χ then </book> else ( ) } }
16      on-first past(bib) return </bib> }
```

The "on-first" handler in lines 11–13 delays query execution until all title-nodes have been buffered and all publisher- and year-nodes have been seen.

When given a different DTD that guarantees that both $\mathit{Ord}_{book}(\mathit{year}, \mathit{title})$ and $\mathit{Ord}_{book}(\mathit{publisher}, \mathit{title})$ hold, the title-nodes can be processed in streaming fashion. The query F_1' produced by "rewrite($root,$\emptyset$,$Q_1'$)" with this new DTD differs from the above query F_1 in the subexpression in lines 11–13 which must be replaced by the expression below.

```
on title as $title return ( if χ then $title else ( ) )
```

Consequently, titles will not be buffered at all. □

Our rewrite algorithm is well capable of optimizing joins over two or more join predicates, as is demonstrated in the following example.

Example 8.6 We remain in the bibliography domain and consider the DTD with start symbol "bib" and the productions as shown below.

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

The following XQuery Q_3 retrieves those authors of articles which are co-authored by people who have also edited books:

```
<results>
{ for $bib in /bib return
    for $article in $bib/article return
      for $book in $bib/book
      where $article/author = $book/editor return
         <result> {$article/author} </result> }
</results>
```

For the remainder of this example, we abbreviate the (normalized) join-condition comparing the authors of articles with the editors of books by χ. Normalization into FluX normal form yields the following query Q_3':

```
1  <results>
2  { for $bib in /bib return
3      for $article in $bib/article return
4        for $book in $bib/book return
5          ( if χ then <result> else ( ),
6            for $author in $article/author return
7              if χ then $author else ( ),
8            if χ then </result> else ( ) ) }
9  </results>
```

When executing "rewrite($root,∅,$Q'_3$)" with the DTD above, a recursive call "rewrite($bib,∅,$\beta$)" is eventually invoked for the subexpression β of Q'_3 in lines 3–8. As β is a for-loop with parent variable $bib and associated set $X = X_\beta = \{\mathit{book}\} \neq \emptyset$, line 24 of the algorithm is executed, returning an expression of the form " {ps $bib: on-first past(book,article) $\cdots$ }". As no order constraint between article- and book-nodes holds, an "on-first"-handler ensures that all articles and books can buffered.

Altogether, "rewrite($root,∅,$Q'_3$)" produces the following FluX query F_3, where α is used as abbreviation for the for-loop over books in lines 4–8 of Q'_3:

```
1  { ps $root:
2     on-first past() return <results>,
3     on bib as $bib return
4      { ps $bib: on-first past(book,article) return
5          for $article in $bib/article return α },
6     on-first past(bib) return </results> }
```

When given a DTD which imposes an order on books and articles, e.g. by the production `<!ELEMENT bib (book*,article*)>`, we can evaluate Q'_3 by buffering only books, but processing articles in a streaming fashion.

Indeed, when executing "rewrite($root,∅,$Q'_3$)" with this new DTD, we eventually encounter the situation where set $X = X_\beta = \emptyset$, and therefore, lines 26–27 (rather than line 24, as with the previous DTD) are executed. Altogether, the new FluX query F'_3 differs from the above query F_3 in the subexpression in lines 4–5, which must be replaced by

```
4  { ps $bib: on article as $article return
5    { ps $article: on-first past(author) return α } }
```

As all book-nodes will have arrived before an article-node can be encountered, data from books is available in buffers once the first article-node is read. When processing the children of an article-node, we first buffer all author-nodes before the query can be evaluated for the current article.

During the evaluation of F'_3, we therefore only buffer the authors of a *single* article in addition to the data already stored on books, whereas the evaluation of F_3 requires the authors of *all* articles to be buffered. □

8.4 Prototype Implementation

We discuss the system architecture of *FluXQuery*, an XQuery engine built on the concept of the translation of XQueries into FluX. This query engine consists

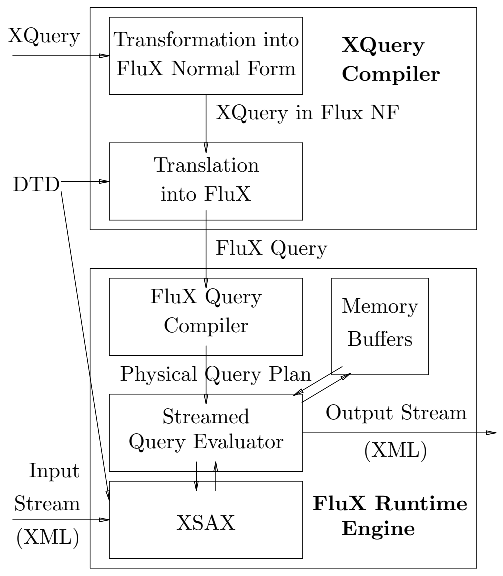


Figure 8.8: *System architecture of the FluXQuery engine.*

of the static *XQuery compiler* and the *FluX runtime engine*, as depicted in Figure 8.8. The XQuery compiler translates user queries written in XQuery into FluX queries. First, XQueries are rewritten into FluX normal form (abbreviated *FluX NF*) that reduces the number of syntactical constructs we need to handle subsequently. Next, the XQuery is translated into a FluX query by the algorithm from Figure 8.6. The second part of the FluXQuery system architecture is the runtime engine. It evaluates FluX queries as obtained by the XQuery compiler. We briefly recapitulate the degrees of freedom in evaluating FluX queries.

Design decisions. The FluX language specifies which parts of a query are evaluated in an event-based fashion. The compilation of XQuery into FluX further ensures that buffer management has enough time to fill main memory buffers with data, before this data is accessed in query evaluation. Yet the FluX language does not specify (1) which data is buffered, where it is buffered, and for how long. (2) Moreover, no assumptions are made how the query evaluation over buffered data (inside "on-first"-handlers) is realized.

In Chapter 6, we discussed several options. Regarding (1), we contrasted a purely static approach to buffer management with an approach also based on dynamic data analysis. In answer to (2), we considered eager and sequential XQuery evaluation. In our prototype, we evaluate queries sequentially and realize the static buffer management described in Section 6.3, which is coupled with XML document projection. Hence, the buffer manager only buffers data relevant to query evaluation, yet possibly redundantly.

The runtime engine is organized as follows. The *FluX query compiler* transforms a given FluX query into a physical query plan. It first extracts projection paths from the parts of the query that are evaluated on buffered data. These capture the data that needs to be buffered. Based on the projection paths, the FluX query compiler schedules query operators, such as the execution of

Query		**FluX**	Galax	AnonX
XMark XM1	5MB	**2.1s / 0MB**	13.4s / 37MB	3.4s
	10MB	**2.8s / 0MB**	29.8s / 83MB	6.7s
	50MB	**7.8s / 0MB**	- / >500MB	38.3s
	100MB	**14.0s / 0MB**	- / >500MB	-
XMark XM8	5MB	**6.8s / 1.54MB**	296.9s / 50MB	143.8s
	10MB	**17.2s / 3.16MB**	1498.3s / 100MB	534.8s
	50MB	**357.8s / 16.00MB**	- / >500MB	-
	100MB	**11566.9s / 32.25MB**	- / >500MB	-
XMark XM11	5MB	**5.6s / 374kB**	277.0s / 50MB	n/a
	10MB	**11.4s / 741kB**	1663.7s / 100MB	n/a
	50MB	**170.8s / 3.64MB**	- / >500MB	n/a
	100MB	**626.8s / 7.27MB**	- / >500MB	n/a
XMark XM13	5MB	**2.2s / 0MB**	12.8s / 38MB	3.0s
	10MB	**3.1s / 0MB**	27.2s / 73MB	5.2s
	50MB	**7.9s / 0MB**	230.1s / 344MB	88.0s
	100MB	**13.9s / 0MB**	- / >500MB	-
XMark XM20	5MB	**2.8s / 4.66kB**	13.2s / 36MB	2.5s
	10MB	**3.4s / 5.18kB**	29.7s / 80MB	6.2s
	50MB	**8.7s / 7.01kB**	- / >500MB	151.9s
	100MB	**15.4s / 7.02kB**	- / >500MB	-

***Figure 8.9**: FluXQuery runtime and memory consumption.*

"process-stream"-expressions, of conditionals, and the buffer population.

The *streamed query evaluator* uses our validating SAX parser *XSAX*, which is an extension of a standard SAX parser. XSAX produces "on-first"-events in addition to customary SAX-events. Basically, the XSAX parser works as follows. We register the DTD and all "on-first"-event handlers of the FluX query with the XSAX parser. Based on this information, the XSAX parser builds an *XML-DPDT* (see Section 3.2) for validating the input against the DTD, and for dynamically generating on-first events. As regular expressions in DTDs are one-unambiguous, these events can be computed efficiently [71]. While reading the input XML stream, the "on-first"-events are properly inserted among the generated stream of SAX events. The streamed query evaluator processes these events and delivers its output as an XML stream.

8.5 Experiments

In order to assess the merits of the FluX approach, we have experimentally evaluated our prototype query engine. The engine is implemented in JAVA, and we consider queries and data from the XMark benchmark [122]. Details on queries and data are provided in Appendix A.

We generated XMark data of the sizes 5MB, 10MB, 50MB, and 100MB. All tests were performed with the SUN JDK 1.4.2_03 and the built-in SAX parser on an AMD Athlon XP 2000+ (1.67GHz) with 512MB RAM running Linux (gentoo linux using kernel 2.6). Our query engine was implemented as described in Section 8.4. As a reference implementation, we employ the Galax query engine (V. 0.3.1). Unfortunately, we could not make the projection-feature from Galax to work (see [77]). The performance of query evaluation was studied by measuring the execution time[3] (in seconds) and maximum memory consumption (in bytes) of each engine. The memory and CPU usage of both query engines were measured by internal monitoring functions, and exclude the fixed memory consumption of the Java Virtual Machine for the FluXQuery engine.

To give a broader overview over the performance of our approach we additionally evaluate our queries with a commercial XQuery system of a major company that has to remain anonymous and will be called AnonX below. Unfortunately, we can not determine the exact memory consumption for this system, and can only state its execution time. As AnonX was not able to parse Query 11, we are not able to list the execution time.

Figure 8.9 shows the results. To evaluate most queries with inputs greater than 10MB, Galax needed more than 500MB of main memory after running for a few minutes (which caused the system to start swapping). These runs were aborted. Our prototype engine clearly outperforms Galax with respect to both execution time and memory consumption. Queries 1 and 13 are evaluated without buffering because of the order constraints imposed by the DTD. Query 20 has to buffer only a single element node at-a-time, which leads to a low memory consumption. Queries 8 and 11 perform a join on two subtrees, and therefore inevitably have to buffer elements. Nevertheless, due to our effective projection scheme, only a small fraction of the original data is buffered. The rapid increase in execution time is due to the fact that we compute joins by naive nested loops at the moment. We discuss this orthogonal but vital issue in Section 10.

The comparison of the execution times to AnonX again shows the competitiveness of our query engine. AnonX ran out of memory processing queries marked by "-" (the maximum heap size of the Java VM was set to 512MB in both cases) and hence did not give any results in this case.

Altogether, our approach seems to perform very well with respect to execution time, maximum memory consumption, and the maximum size of XML documents that can be processed.

8.6 Conclusion

In this chapter, we have presented the FluX language together with an algorithm for translating a significant fragment of XQuery into equivalent FluX queries. The primary intention of FluX is as an internal representation format for queries, rather than a language for end-users. Nevertheless, it provides a strong intuition for buffer-conscious query processing on XML streams. We provide an algorithm that uses schema information to statically schedule event-based query constructs in FluX queries, to reduce the usage of buffers.

As evidenced by our experiments, our approach dramatically increases the scalability of main memory XQuery engines, even though we think we are not yet close to exhausting this approach, neither with respect to run-time buffer management and query processing, nor query optimization.

For future work, want to extend the covered XQuery fragment. If a nonrecursive DTD is available, XQueries with descendant axes can easily be rewritten into the smaller XQuery fragment covered here. For more expressive queries, the FluX language and the compilation algorithm need to be extended accordingly.

[3]The times for query rewriting are negligible, and are hence not reported.

Another important starting point for future work is to push if-expressions, which we have moved down the query tree to obtain our normal form, back "up" the expression tree as soon as the other simplifications have been realized. First ideas in this direction are described in [71] and [99].

9 Buffer Purging based on Static and Dynamic Analysis

In this chapter, we present an effective buffer purging algorithm that is inspired by the notion of *relevance counts*. We have motivated this approach in Section 6.3.2, and will recapitulate the main ideas in Section 9.1. In Section 9.2, we provide an overview over the components of the runtime system. Section 9.3 is dedicated to the static and dynamic analysis required for assigning relevance counts to buffered nodes, and for also decrementing relevance counts. Our prototype implementation and our extensive experiments are discussed in Section 9.4. We provide a summary in Section 9.5.

9.1 Motivation

In keeping the main memory consumption low during streaming XQuery evaluation, it is essential that memory buffers are purged continuously. In other words, a *garbage collection* mechanism is required that removes data from buffers when it is no longer relevant to query evaluation.

In existing systems, buffers are purged based on either static or dynamic analysis. The query engines of [15, 43, 70, 75, 85] employ static analysis. In Section 6.3.1, we have discussed the implications of static buffer management in case of the FluXQuery engine, namely that nodes may be buffered redundantly. In contrast, the XStream engine [50] implements a dynamic approach. In XStream, buffers are purged based on garbage collection via reference counting. When a subtree is no longer referenced, it can be purged from memory.

In this chapter, we introduce a novel buffer purging algorithm that combines static and dynamic analysis, and that pro-actively purges buffers, as we explain next. Based on static query analysis, we incrementally compute a projection of the input, thus we only buffer data relevant to query evaluation. In addition, we statically infer the moments during query evaluation when buffered nodes have become irrelevant. In identifying such *buffer preemption points*, we assume a sequential query evaluation. To timely delete nodes from the buffer, a dynamic analysis takes into account the current buffer contents, the state of query evaluation, and the progress made in reading the input. Obviously, we may expect the impact of *combined static and dynamic analysis* on main memory consumption to be greater than what can be achieved by static or dynamic analysis alone.

In concept, our approach is related to garbage collection via reference counting, as each node in the buffer keeps track whether it is still relevant to the remaining XQuery evaluation. Yet instead of counting references, we rely on *relevance counts*. Intuitively, a relevance count serves as a metaphor for the future relevance of a buffered node.

While a traditional garbage collector is *passive* in the sense that it is invoked whenever there is no more space for allocating new objects, our approach is *active*. That is, we purge buffers from irrelevant nodes early on, so that both the high watermark and the average main memory consumption remain low throughout query evaluation.

The basic idea behind *active garbage collection* is clean and simple: We statically extract projection paths from the query. While reading the input, the input is prefiltered, and relevance counts are assigned to all nodes that are copied into the buffer. A node can be matched by multiple projection paths, and even multiple times, if it is used in the query by different subexpressions. The relevance count of a node captures such multiple matches. At compile-time, we determine the moments during query evaluation when relevance counts are decremented. Once the relevance count for a node has reached zero at runtime, the node becomes a candidate for removal from the buffer.

9.2 System Overview

We begin with an overview of the runtime system. We allocate a *single* buffer which contains the currently relevant projected document tree. Each node in this tree has an integer-value to keep track of its current relevance count, and a Boolean flag to mark it as *finished* or *unfinished*. Simply put, text nodes are finished the moment they are loaded into the buffer. An element node remains unfinished until we have processed its closing tag in the input stream. Only then is the node considered finished. This information is required in garbage collection, where unfinished nodes won't be purged from buffers, so as not to corrupt the tree datastructure while query-variables are still bound to this node or its descendants.

The relevance counts for each node are assigned when a node is first inserted into the buffer. To decrement relevance counts at runtime, we need a mechanism to notify the buffer manager. This is achieved by statically compiling signOff-statements into queries, which mark the preemption points where the relevance counts in the buffer are updated.

The runtime system. The runtime architecture comprises three components, the *query evaluator*, the *stream preprojector*, and the *buffer manager*. The interaction between these components is *pull-based*:

- The query evaluator executes the query sequentially until it has to *block* either because a new node is required (e.g. when a variable is bound to the next node in a 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.
- 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 requests new data from the stream preprojector until the data is available, or it

has become evident that the data does not exist (e.g. as the input is exhausted). The reception of signOff-statements triggers garbage collection.

- Once it has been activated by the buffer manager, the stream projector processes the input stream until a token relevant to query evaluation is matched. A corresponding node is inserted in the in-memory tree, and the relevance count of this node is initialized.

Via this chain of commands, the query evaluator incrementally processes the input and evaluates the query on-the-fly, over the buffered data.

Garbage collection. Active garbage collection relies on the correct assignment of relevance counts, the timely reduction of these values, and ultimately, the purging of nodes from the buffer. A buffered node can be purged if it is marked finished and if the node and its descendants have relevance count zero. In the following, we assume that the buffer contains the projected input document (as read so far) and that buffered nodes are marked unfinished or finished.

Traditional garbage collectors start searching for memory that can be freed whenever there is no more space for allocating 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 or a node is marked finished. Back in Section 6.3.2, we have also discussed alternative strategies. As the garbage collector is invoked quite often, it is desirable to restrict the search space for nodes to purge within the buffer. The garbage collector is invoked the moment that a node is marked finished, or a signOff-statement is executed. This already narrows down the set of nodes that are candidates for removal. In the former case, we only consider the node now finished, in the latter, all nodes which have been affected by a signOff-statement. If the relevance count of any of these nodes has reached zero, the garbage collector checks whether it can be deleted from the buffer. If this is possible, the garbage collection proceeds bottom-up in the buffered tree. Thus, the deletion of nodes can propagate up to the root node of the buffered tree.

The treatment of unfinished nodes in the buffer requires extra care. An unfinished node is not deleted to avoid buffer corruption. Instead, it is *marked* deleted and ultimately purged from the buffer once its closing tag is read.

9.3 Static and Dynamic Analysis

The correctness of active garbage collection relies on the interplay of (1) XML projection to load all relevant data into buffers, (2) the assignment of relevance counts to buffered nodes, and (3) decrementing these values at runtime.

Regarding (1), we implement the projection semantics introduced earlier in this thesis, for which we have already ascertained that it is correct and can be realized with little runtime overhead. As far as (2) and (3) are concerned, we can guarantee that the number of signOff-statements evaluated on each buffered node equals the relevance count initially assigned. Thus, only nonzero relevance counts are ever decremented, and no *memory leaks* can occur, as all nodes eventually reach a relevance count of zero and are purged from the buffer.

In the following, we provide the particulars of static and dynamic analysis by which we ensure these properties.

9.3.1 Assigning Relevance Counts

Given a query from our XQuery fragment XQ, we perform normalization and extract projection paths as outlined in Chapter 3. At runtime, we couple XML prefiltering and the assignment of relevance counts when loading data into buffers. XML document projection can be realized by an *XML-DPDT*, as discussed in Section 3.5.4. The result is an XML stream which encodes the projected document tree. This is the input for the *tree builder*, the module responsible for the incremental construction of the internal tree representation and the assignment of relevance counts to buffered nodes.

We compute relevance counts based on counting paths, as introduced in Section 3.6. Initially, all projection paths are represented as initial counting paths, and are pushed on a stack. For each opening tag or text event in the input stream, a node is inserted in the in-memory tree representation. At this point, all irrelevant input has already been filtered out. Then the set of counting paths on top of the stack is copied, updated, and pushed on top of the stack. The total number of times a projection path is matched by this node is assigned as the relevance count of this node. For closing tags, the stack is popped accordingly.

Example 9.1 We consider the normalized query from Figure 3.6(b). In Example 3.12, we have shown the mapping *pp* between query-variables and the associated projection paths. We discuss how the tree builder proceeds for the input stream prefix "$\langle bib\rangle\langle book\rangle\langle price\rangle\langle/price\rangle$".

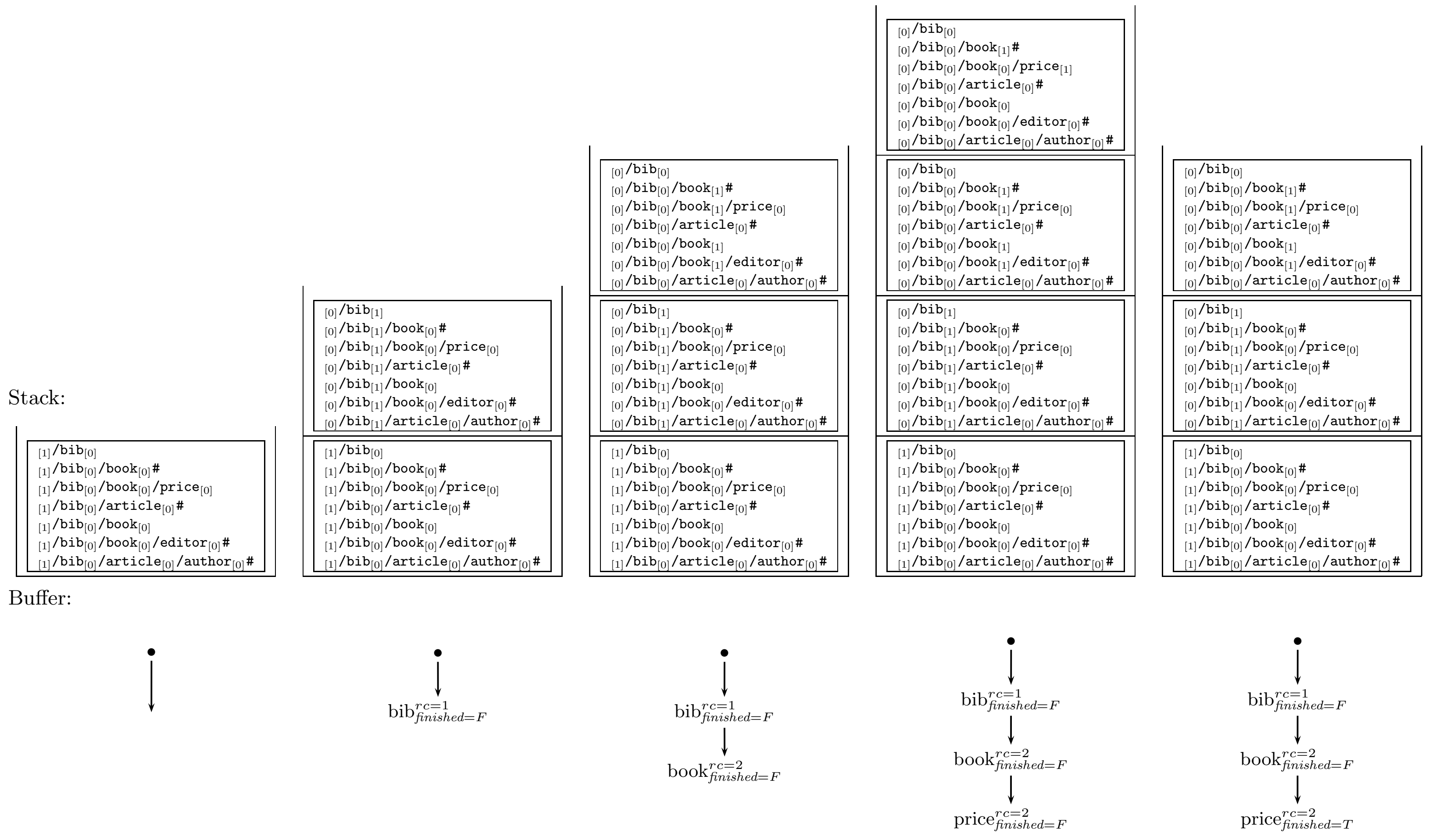
In Figure 9.1, we show the current contents of the stack and the buffer. In step (a), the stack holds the initial counting paths and the buffer only contains an unnamed root node. In step (b), we process the first input token $\langle bib\rangle$. We update each counting path P on top of the stack by computing $[\![P]\!]^{bib}_{tail}$ according to the rules from Figure 3.12. Then we push the updated counting paths on the stack, and insert a new node in the main memory buffer. In our example, this node is assigned relevance count 1 (denoted rc), as the projection path /bib is matched once for the bib-node. The node is marked unfinished, as its closing tag has not yet been read.

The next input token is $\langle book\rangle$, with the stack and buffer as shown in step (c). The buffered node obtains relevance count two, as two projection paths are matched. The effect for the opening tag of the price-node is similar, and portrayed in step (d). When we read the closing tag $\langle/price\rangle$, we mark the buffered price-node as finished and pop the stack. □

9.3.2 Decrementing Relevance Counts

In static analysis, we further compute the buffer preemption points for garbage collection. We first introduce some terminology.

Given a query, we compute the *data scope* of query-variables. Intuitively, this is a variable whose scope defines how long the data captured by $pp(\$x)$ remains relevant for query evaluation. In the previous example, the data associated with variable \$b is captured by the projection path *pp*(\$b). Once the scope of variable \$b ends, this data is no longer accessed by query expressions within the scope of \$b. Hence, variable \$b defines its own data scope. On the other hand, the data associated with variables \$x1 and \$x3 is captured by *pp*(\$x1) and *pp*(\$x3). This data must remain buffered at least as long as there are

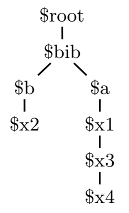


(a) Initial state. (b) Reading ⟨*bib*⟩. (c) Reading ⟨*book*⟩. (d) Reading ⟨*price*⟩. (e) Reading ⟨*/price*⟩.

Figure 9.1: *Computing reference counts in Example 9.1.*

article-nodes to process, as we access this data within the for-loop over articles. Hence, the data scope of these variables is defined by the scope of variable \$bib.

Variable trees. Given a normalized query, its variable tree is extracted as follows. For each variable $\$x$ in the query (including the implicit variable \$root), we define a node $v_{\$x}$ in the variable tree. The parent-child relationships between nodes in the variable tree mirror the nesting of for- and some-expressions in the query. Variable trees are unordered.



(a) Variable tree.

$\$x$	*data_scope*($\$x$)
\$root	\$root
\$bib	\$bib
\$b	\$b
\$x2	\$b
\$a	\$a
\$x1	\$bib
\$x3	\$bib
\$x4	\$a

(b) Data scopes.

***Figure 9.2**: Data scope of variables.*

Example 9.2 For the query from Figure 3.6(b) we extract the variable tree shown in Figure 9.2(a). The variable tree reflects that variables \$bib and \$b are defined in nested for-loops, and that variable \$x2 is defined in a some-expression that is nested within the for-loop for variable \$b.

Variables \$a and \$x3 are not in a parent-child relationship in the variable tree, as a some-expression is nested between their declarations. □

Data scopes. The variable tree of a query is the basis for defining the data scope function. The data scope function is a mapping between the query-variables, which we denote by *data_scope*. Intuitively, let $\$x$ be a query-variable, then the data buffered for all variables in $\mathit{data\_scope}^{-1}(\$x)$ must not be purged from buffers while $\$x$ is still bound.

We now discuss the computation of data scopes. Given a variable tree, we first compute a subset of nodes V_{ds}. These are the variables that define their own data scope. By default, $\$\text{root} \in V_{ds}$. Moreover, let $\$x$ be a query-variable. Then $\$x$ is in V_{ds} if two conditions are met. The parent node $\$y$ of $\$x$ in the variable tree is also in V_{ds} and there is an expression of the form "for $\$x$ in $\$y/\alpha$ return β" in the query. Then the data-scope of a query-variable is defined as

$$\mathit{data\_scope}(\$x) = \begin{cases} \$x & \text{if } \$x \in V_{ds} \\ ds(\mathit{parentVar}(\$x)) & \text{otherwise.} \end{cases}$$

Example 9.3 We continue with our query from Example 9.1 with the variable tree in Figure 9.2(a). The data-scope mapping is listed in Figure 9.2(b). □

Semantics of signOff-statements. A signOff-statement is an expression of the form signOff(α) where expression α is either a query-variable $\$x$ or $\$x/\pi$

```
Given a normalized query $Q$, we insert signOff-statements:
(1) rewrite expression $\alpha$ in query $Q = \langle a\rangle\alpha\langle/a\rangle$:
  for each variable $\$z$ in $data\_scope^{-1}(\$\text{root}) \setminus \{\$\text{root}\}$ do
    if $pp(\$z)$ carries the flag "#"
    then $\alpha$ is replaced by ($\alpha$, signOff($lineage(\$\text{root}, \$z)$#))
    else $\alpha$ is replaced by ($\alpha$, signOff($lineage(\$\text{root}, \$z)$)) end if end for
(2) rewrite expression $\alpha$ in all for-loops "for $\$x$ in $\$y/\sigma$ return $\alpha$":
  for each variable $\$z$ in $data\_scope^{-1}(\$x)$ do
    if $pp(\$z)$ carries the flag "#"
    then $\alpha$ is replaced by ($\alpha$, signOff($lineage(\$x, \$z)$#))
    else $\alpha$ is replaced by ($\alpha$, signOff($lineage(\$x, \$z)$)) end if end for
(3) rewrite expressions $\alpha$ and $\beta$ in all if-statements "if $\chi$ then $\alpha$ else $\beta$":
  for each variable $\$x$ in $boundVar(\beta)$ do
    if for all $\$y$ in $freeVar(\beta)$: $\$x \notin data\_scope^{-1}(\$y)$
    then
      $\$y := parentVar(\beta)$;
      if $pp(\$z)$ carries the flag "#"
      then $\alpha$ is replaced by ($\alpha$, signOff($lineage(\$y, \$z)$#))
      else $\alpha$ is replaced by ($\alpha$, signOff($lineage(\$y, \$z)$)) end if
    end if end for
(4) repeat step (3) with the roles of $\alpha$ and $\beta$ reversed;
```

***Figure 9.3**: Algorithm: Compiling queries with signOff-statements.*

with a projection path π. The semantics is the following. At runtime, let $\$x$ be a variable bound in the current environment, and let $n_{\$x}$ denote the node in the buffer to which this variable binds. Then we proceed as follows.

- Executing "signOff($\$x$)" decrements the relevance count of $n_{\$x}$ by one.
- In executing a statement "signOff($\$x/\pi$)", we consider each node in the buffer that is matched by $\$x/\pi$. If path expression $\$x/\pi$ is matched k-times by this node (c.f. Definition 3.15), then we decrease the relevance count of this node by k.

Note that in the second case, query evaluation blocks until node $n_{\$x}$ is marked finished. The rationale behind this step is that by then, all nodes matched by the path $\$x/\pi$ are available in the buffer, so these nodes cannot "escape" garbage collection.

Compilation of queries with signOff-statements. Given a normalized query, the algorithm in Figure 9.3 statically inserts signOff-statements. In steps (1) and (2) of the algorithm, we consider variable $root, as well as all variables bound in for-loops. At the end of the scope of a variable $\$x$, we emit signOff-statements for all variables with $\$x$ as their data scope. At this point, we distinguish whether relevance counts are also decremented for descendants of nodes, as indicated by flag "#". In step (3), we handle if-statements. Once we enter an alternative execution path at runtime, e.g. as the condition is true, then the relevance counts assigned for nodes in the other execution path are

```
<results>
{ for $bib in /bib return
  ( for $b in $bib/book return
    ( if ( some $x2 in $b/price satisfies true() )
      then $b else (),
      signOff($b#),            (: introduced for $b   :)
      signOff($b/price) ),     (: introduced for $x2  :)

    for $a in $bib/article return
    ( if ( some $x1 in $bib/book satisfies
           ( some $x3 in $x1/editor satisfies
             ( some $x4 in $a/author satisfies ( $x3 = $x4 ) ) ) )
      then $a else (),
      signOff($a#),                   (: introduced for $a   :)
      signOff($a/author#) ),          (: introduced for $x4  :)

    signOff($bib),                    (: introduced for $bib :)
    signOff($bib/book),               (: introduced for $x1  :)
    signOff($bib/book/editor#)  ) }   (: introduced for $x3  :)
</results>
```

Figure 9.4: *XQuery with signOff-statements.*

also decremented. Otherwise, we could create *memory leaks*, i.e. nodes whose relevance count is never decremented.

The first example below only requires the first two steps, while the second example illustrates the third step of the algorithm.

Example 9.4 We again consider the query from Figure 3.6. The rewritten query is shown in Figure 9.4. We have omitted redundant parenthesis, and have annotated each signOff-statement with the variable for which it has been introduced using XQuery comments (encapsulated in "`(:`" and "`:)`"). □

In the following, we consider yet another query, where the treatment of if-statements comes into play.

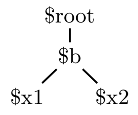
Example 9.5 The following query outputs the titles of books if these books have no editors.

```
<results>
{ for $b in //book return
  if ( not( some $x1 in $b/editor satisfies true() ) )
  then for $x2 in $b/title return $x2
  else () }
</results>
```

We extract the projection paths, the variable tree, and the data scopes.

$\$x$	$pp(\$x)$	$data\_scope(\$x)$
$root		$root
$b	//book	$b
$x1	//book/editor	$b
$x2	//book/title#	$x2

Variable relationships.



Variable tree.

Based on this information, we compile signOff-statements into the query.

```
  <results>
  { for $b in //book return
    ( if ( not( some $x1 in $b/editor satisfies true() ) )
      then for $x2 in $b/title return ($x2, signOff($x2#) ) (: see $x2 :)
      else signOff($b/title#),                  (: introduced for $x2 :)

      signOff($b),                              (: introduced for $b  :)
      signOff($b/editor)  ) }                   (: introduced for $x1 :)
</results>
```

Assume we are processing a book-node. Then the book titles are buffered because they may have to be output. Yet once an editor is encountered, the condition evaluates to false. Hence, we enter the else-path. While no output is produced in the else-path, the relevance counts assigned to title-nodes must nevertheless be decremented. Otherwise, we would create memory leaks in form of nodes that are not purged from buffers, even though they won't be accessed in query evaluation any more. The evaluation of signOff-statements stalls until the book-node to which variable $b is bound is finished. By ensuring that the relevance counts of all titles are decremented, memory leaks cannot occur. □

In the example above, once the else-part is entered, title-nodes of the current book need not be loaded into the buffer anymore. However, our system uses a static approach to XML projection, and cannot "unsubscribe" or change its subscription of query-relevant nodes. We discuss this problem among our ideas for future work in Chapter 10.

9.4 Experiments

We have implemented active garbage collection for a prototype XQuery engine, the GCX system. GCX is implemented in C++ which, unlike garbage collected languages, gives direct control over memory allocation and deallocation. This is crucial when designing a query engine with low memory consumption. Below, we discuss our prototype implementation and our experiments.

9.4.1 Prototype Implementation

Our buffer datastructure is a tree, with parent-child and next-sibling pointers between nodes. We have merged the components for XML document projection and the tree builder into one module, exploiting synergy effects arising from a single stack. The states of the projection automaton now contain counting paths, which are computed lazily [53]. Note that instead of assigning relevance counts, our prototype assigns *roles* (see [97]), which are simply relevance counts that are named by the query-variables for which they are introduced.

Whenever nodes are matched by the "#"-flag in projection paths, we assign *aggregate roles* to avoid the overhead for assigning and decrementing each single node in a subtree. The garbage collection mechanism has been adapted accordingly. While the original path extraction mechanism in [97] treats existential checks differently, this is of no consequence for the experiments conducted here.[1]

The earlier approach uses static query rewriting to ensure that if-statements do not occur outside for-loops, which simplifies static analysis. This tends to increase the size of queries, and can cause a runtime overhead due to additional relevance counts that are assigned and decremented. We manage to avoid this in the approach featured here. Yet this is not reflected in the experiments, as the XMark benchmark queries do not employ nested if-then-else expressions.

To further reduce the runtime overhead for garbage collection, projection paths can sometimes be pruned, so that redundant paths are eliminated. That way, nodes receive smaller relevance counts (or are assigned fewer roles), which also leads to fewer decrements (or role updates) at runtime.

9.4.2 Experimental Results

We consider XMark documents between 10MB and 200MB in size. Details on the data and queries are provided in Appendix A. The benchmarks are carried out on a 3GHz CPU Intel Pentium IV with 2GB RAM, running SuSe Linux 10.0. We use J2RE v1.4.2 for running Java-based programs.

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

- The FluXQuery engine (implemented in Java, see Chapter 8) is the most natural choice for a reference implementation. FluXQuery is also a main-memory XQuery engine geared towards XML stream processing, and it implements a similar XQuery fragment. FluXQuery can exploit schema information, and was provided the XMark DTD in our experiments.

- The in-memory query engines Galax [51] (OCaml), QizX/open v1.1 [90] (Java), and Saxon v8.7.1 [93] (Java) implement full XQuery. While Galax version 0.6.8 has not been designed with XML stream processing in mind, it is often consulted in XQuery benchmarks. For this reasons, comparative experiments are also included here

Unfortunately, there are only few implementations of streaming XQuery engines publicly available. This makes it difficult to set up extensive comparative experiments. Acting from this necessity, we further consider experiments with MonetDB v4.12.0 with XQuery v0.12.0 [81], a mature XML database system. As a secondary-storage implementation, MonetDB uses index structures to speed up query evaluation, which is not done by the GCX engine. Also, MonetDB XQuery stores the entire data physically before query evaluation. To account for the fact that GCX and the other main memory engines read the complete input document for each query evaluation, we force the MonetDB server to reload the complete document in each run.

The focus of our experiments is primarily on main memory consumption, but we also consider the query execution time. Main memory consumption is measured with the Linux *top* command. For each system and query we set a timeout of one hour. Figure 9.5 shows the results of our experiments. For each system and size of the input document, we measure the high watermark of

[1]In [97] we also consider projection paths of the form /bib/book/price[1] for existence checks in XQuery conditionals. As we only check for the existence of a certain path, it is sufficient to consider only the first witness for such a path as relevant to query evaluation.

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

***Figure 9.5**: GCX benchmark results over XMark data.*

non-swapped memory consumption, and the total query evaluation time. The value "n/a" indicates that the query cannot be expressed in the language supported by the specific engine, while "-" denotes failure, as caused by segmentation faults. With the Java-based engines, we observe that due to effects caused by automatic memory management and the Java Virtual Machine, memory consumption often increases with the document size, even though the amount of data buffered is bounded by a small constant (e.g. for FluXQuery).

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

For queries $Q1$, $Q6$, $Q13$ and $Q20$, the memory consumption of our prototype is independent of the input stream size. Little has to be buffered, 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 still manages to evaluate this query with low main memory consumption. As in FluXQuery, 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 which can be improved with standard database techniques. We resume this discussion in Section 10.

In summary, the experiments confirm that buffer management via active garbage collection performs well both w.r.t. main memory consumption and execution time. For a large class of queries, our prototype can even outperform the FluXQuery engine, which exploits schema information.

9.5 Conclusion

In this chapter, we have discussed the first buffer manager for streaming XQuery engines which employs *static and dynamic* analysis to reduce main memory consumption. The approach is based on the notion of relevance counts that are assigned to buffered nodes. These counts serve as a metaphor for the relevance of a node to query evaluation. We have shown how counts are initialized, how they are decremented, and when nodes can be ultimately purged from buffers.

In summary, active garbage collection has proved to be an effective buffer purging technique. Our prototype implementation shows the significant impact of active garbage collection on main memory consumption and query evaluation time. As we have argued in Chapter 6, we can achieve redundant-free buffering for our query fragment. As confirmed by our experiments with XMark data and queries, *combined* static and dynamic analysis can achieve a lower memory consumption than systems which exclusively rely on static buffer management.

Part IV

Synopsis

10 Conclusion and Summary

This dissertation is dedicated to XML stream processing in main memory-based query engines. The memory consumption of these systems constitutes a major bottleneck. This particularly holds for systems that allow for data transformations via powerful declarative query languages such as XQuery, rather than mere XPath filtering. In Part II of this dissertation, we have presented an abstract framework for specifying XML stream processors. By modeling various system setups, we are able to gain a greater-picture view on the factors influencing the memory footprint during query evaluation. In Part III, we have presented buffer management algorithms that employ static and dynamic analysis. We next review the targets of our analysis techniques.

Static analysis techniques. We have studied main memory-based XQuery processors that load the input for the evaluation of single queries. As the query is known in advance to loading the data, this allows for static query analysis regarding the relevance of data to query evaluation, order constraints, and possible buffer preemption points.

Data relevance. An established technique for reducing the main memory consumption in the context of this work is XML document projection. Given a query, we statically extract the information *which* data is relevant for query evaluation. At runtime, we then filter out irrelevant data. In this dissertation, we contribute a new approach to implementing XML prefiltering, where we combine ideas from string pattern matching with schema knowledge. This leads to significant runtime speedups at low memory costs (see Chapter 7).

Order constraints. We further derive order constraints from queries. This concerns the order in which data from the input is required for producing output in query evaluation. Schema knowledge, if available, describes the order of data in the input. If this order coincides, we can evaluate parts of the query directly on the input stream. The remaining parts of the query are then evaluated on buffered data. Even if no schema is available, many practical queries can be partly evaluated on the stream, which can significantly reduce the main memory consumption (see Chapter 8).

Buffer preemption points. If we assume a particular mode of query execution, we can derive data access patterns from queries. In particular, we can determine the moments when certain parts of the input may no longer be relevant to query evaluation. We refer to these moments as *buffer preemption points*. In combination with runtime analysis, we have built a garbage collection algorithm that efficiently purges buffers during query evaluation (see Chapter 9).

Dynamic analysis techniques. The XQuery engines developed in this thesis also rely on dynamic analysis. At runtime, we can analyze the state of both the input and of the buffered data.

Analysis of input data. When streaming query operators are scheduled statically, they must be synchronized with the arrival of tokens at runtime. In Chapter 8, we make use of automata to efficiently analyze the input, and to generate events which communicate that certain XML tokens won't be witnessed anymore. These automata validate the input against DTDs in order to issue these events as early as possible.

As a prerequisite to active garbage collection (see Chapter 9), nodes are tagged with a relevance count when they are loaded into main memory buffers. This relevance count denotes the number of XQuery subexpressions for which a node is considered relevant. If XQueries contain wildcards or transitive XPath expressions, relevance counts must be computed dynamically, which we also realize using automata-based techniques.

Analysis of buffered data. In active garbage collection, we search the buffer at runtime for nodes that can be purged. If buffers are to contain no redundant data, then the decision when to purge nodes from buffers cannot be based on static analysis alone (see Section 6.3.1). Out of this motivation, we have developed a dynamic approach based on garbage collection.

It is commonly regarded as one of the advantages of reference counting that this garbage collection mechanism works locally and incrementally. This also holds for our garbage collector based on *relevance* counting, and our experiments confirm the scalability of our approach.

The buffer management techniques presented in this dissertation effectively *combine* static and dynamic analysis. Moreover, they are composable, not just with each other but also with other approaches for XQuery evaluation. For instance, the static scheduling of streaming query operators in the FluXQuery engine makes no assumptions about the implementation of query evaluation over buffered data. For the buffered data, we can again employ active garbage collection. The garbage collection mechanism presented here can be extended to any query evaluation plan as long as we can identify the moments when the evaluation of certain subexpressions has finished. The assumption that query evaluation proceeds in phases is justified in XML stream processing.

The extension of our techniques is the topic of the next section, where we discuss future work.

```
<results>
{ if ( ... some German seller offers item x ...)
  then <German_sellers>
       { ...names of German sellers offering item x... }
       </German_sellers>
  else <international_sellers>
       { ...names and addresses of all sellers offering item x... }
       </international_sellers>  }
</results>
```

Figure 10.1: *Sketch of an XQuery expression (Example 10.1).*

Future Work

A natural step for future work is to extend the supported XQuery fragment, to cover composition as well as aggregation. In part, XQuery aggregation in the context of XML streams has been featured elsewhere [23, 75, 99]. Of course, the interplay with our techniques, such as active garbage collection, is still an issue worth inspecting. Along this line, it is interesting to explore how our techniques can be applied to other XML query languages, for instance, stylesheet-based languages that use XPath such as XSLT [115] or STX [29], or functional languages such as XDuce [61] or XStream [50] that do not provide path expressions as syntactic constructs.

Moreover, we believe that we have not yet tapped the full potential of the analysis techniques employed in this thesis. Below, we sketch two starting points for further research. The first concerns the evaluation of conditionals in XQuery, and the second the representation of XML data inside main memory buffers.

Conditionals in XQuery. XQuery has two constructs for evaluating conditionals, namely where-clauses and if-statements. Surprisingly, in typical XQuery benchmarks [80, 120, 122] and the XQuery Use Cases [123], few if-statements occur. If they do occur, then the else-part is usually empty, or the alternative branches compute fixed output.

We may suspect that the lack of if-statements in benchmarks is partly responsible for the fact that query optimization for such queries has been neglected so far. However, we may reasonably assume that developers make use of the if-construct, even more so as XQuery is increasingly employed as a programming language. The example below illustrates that if-statements are not trivial from the perspective of buffer management.

Example 10.1 We process an XML stream of auction data with information on items, auctions, buyers, and sellers. We want to buy a specific item, and want to save on the shipping costs. We can save costs by buying the item from sellers close by. So if people with an address in Germany offer this item, we want to list these sellers. If no German sellers offer this item, we want the list of all sellers of this item and their addresses. In Figure 10.1, we show a rough sketch of such a query.

In XML document projection, we preserve the data required for evaluating the condition, the then- and the else-part. All other input data is discarded. We require the addresses of sellers for the else-part, but not for the then-part.

All existing systems for XML document projection rely on purely static query analysis to extract the projection paths describing query-relevant data. Yet in streaming query evaluation, we may realize early on that a condition is satisfied. At this point, it is desirable to focus on processing the then-part only. In particular, we want to purge all data in buffers that is exclusively required for evaluating the else-part (such as the addresses of sellers). More so, we want to immediately stop buffering this data. Yet the GCX query engine introduced in Chapter 9 is not capable of changing is objectives in XML document projection. The projection paths are computed once in static analysis, and remain unchanged during query evaluation.

What is required is a *dynamic* approach to XML document projection, where the query engine can notify XML stream prefiltering that certain data is no longer considered relevant for query evaluation. While there is work on dynamic publish-subscribe systems for XPath filtering and routing, to the best of our knowledge, there is no work on dynamic XML document projection. In fact, correctly solving this problem is by no means trivial for XQuery, and requires a careful analysis of the current state of query evaluation. □

Bulk bypassing. In XML document projection, we operate on the insight that in the evaluation of many queries, large parts of the input are actually irrelevant. However, there are also queries where little can be gained by projection. We have observed in [95] that this concerns queries that require large parts of the input only for generating output. Often, large chunks of data are written to the output as read in the input, without traversing their tree-structure. Such "bulk" data may be stored and treated differently from data that is actually traversed in query evaluation. In [95], we present preliminary work on bulk bypassing in XQuery engines. This comprises a technique to recognize bulk data, which can be coupled with XML document projection. We show that bulk data arises in practice, and discuss ideas along the line of bulk-bypassing in main memory-based XQuery engines.

Outlook

This dissertation partly leverages ideas from other fields of computer science, such as string matching and garbage collection, and applies them successfully to XML stream processing. With respect to the larger field of research in data management, we hope that our techniques may be adopted in other areas as well. Below, we elaborate on how this could come about.

Data-specific garbage collectors. In developing our active garbage collector, we have applied the idea of garbage collection via reference counting in a new domain. There is earlier work where garbage collection strategies have been employed by the database community, namely in object-oriented databases [9]. However, these earlier approaches rely on the purely dynamic analysis of references between objects. In contrast, we also statically analyze queries to derive data access patterns.

This raises the question whether other main memory-based query engines may also benefit from garbage collection schemes that are specifically tailored to the datastructure and queries at hand, for instance when evaluating queries on RDF data [112]. The additional challenge, apart from analyzing a different query language, lies in the fact that graphs rather than trees must be handled.

XML stream processing versus XML databases. Throughout this thesis, we have made a clear distinction between query evaluation over XML streams and in XML databases. However, much can be gained on both sides by tearing down these fences.

Leveraging XML database techniques for XML stream processing. In our discussions, we have argued strongly in favor of lazy (or sequential) XQuery evaluation, in the interest of keeping the main memory consumption low. In our prototype implementations, value-based joins are thus realized as simple nested-loops joins. Evaluating these joins comes at no extra main memory overhead, but as can be expected, the runtime performance suffers considerably for larger inputs.

By investing buffer space for building indexes, we can accelerate join processing, and thus the overall runtime performance of our systems. This is the classic tradeoff between main memory and runtime. To some part, we can use standard database techniques for join processing, yet with the constraint that the construction of indexes must proceed with little runtime and memory overhead.

Leveraging XML stream processing techniques for XML databases. In compiling query plans in XML databases, we are not technically limited to scanning the data in document order, as we are in stream processing. However, it is not uncommon that nodes are nevertheless physically stored in this order, due to clustering and indexing schemes (e.g. [48]). Then performing a full table scan bears close resemblance to accessing nodes in XML stream processing: Parent nodes are encountered before their children, and all nodes are encountered before their following siblings.

This suggests the idea of employing streaming techniques in accessing relational XML data that is clustered in document order. In particular, we can apply the FluX compilation algorithm in generating query execution plans. Then the streaming part of FluX queries can be evaluated in pipelining fashion, and the remaining parts are compiled as usual. For queries that are nonblocking, or only contain locally confined joins, this approach can lead to cheaper query plans, as we avoid the computation of structural joins for those parts that are evaluated in streaming fashion.

Part V

Appendix

A Benchmark Data and Queries

In experiments with the GCX and FluXQuery prototypes, we made the following modifications to the XMark benchmark data and queries [122].

- We consider XML without attributes, and rewrite all attributes to subelements. For instance, token `<book id="1">` is encoded by the sequence `<book><book_id>1</book_id>`. The XMark DTD is adapted accordingly.
- Our query fragment does not support aggregation. Hence, we modified selected XMark queries, as listed below.

Queries. In the following we list the queries. Note that query Q11 uses arithmetic operators, which are not defined in the XQuery fragment XQ defined in Section 3.4, but which are nevertheless implemented in the FluXQuery engine (see Chapter 8).

Q1

```
<query1> {
   for $site in /site return
     for $people in $site/people return
       for $person in $people/person
       where($person/person_id="person0")
       return <result> { $person/name } </result> }
 </query1>
```

Q6

```
<query6> {
   for $site in //site return
     for $regions in $site/regions return $regions//item }
 </query6>
```

Q8

```
<query8> {
   for $site in /site return
     for $people in $site/people return
       for $person in $people/person return
         <item> {
           <person>{ $person/name }</person>,

              <items_bought> {
               for $site2 in /site return
                 for $cas in $site2/closed_auctions return
                   for $ca in $cas/closed_auction return
```

```
                    for $buyer in $ca/buyer
                    where($buyer/buyer_person=$person/person_id)
                    return <result> { $ca } </result> }
           </items_bought> }
         </item> }
</query8>
```

Q11

```
<query11> {
  for $s in /site return
    for $people in $site/people return
      for $p in $people/person return
        <items>
        { $p/name }
        { for $s2 in /site return
            for $oas in $s2/open_auctions return
              for $o in $oas/open_auction
              where ( some $prf in $p/profile satisfies
                      ( $prf/income > (5000 * $o/initial) ) )
              return $o/open_auction_id }
        </items> }
</query11>
```

Q13

```
<query13> {
  for $site in /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>
```

Q20

```
<query20> {
  for $site in /site return
    for $people in $site/people return
      for $person in $people/person
      where (fn:not(fn:exists($person/person_income)))
      return $person }
</query20>
```

Note: The path `$person/person_income` is unsatisfiable by the DTD, hence all persons qualify for output in this query.

B Publications

- Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier: *Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams.* In Proc. VLDB 2004.
- Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. *FluXQuery: An Optimizing XQuery Processor for Streaming XML Data.* In Proc. VLDB 2004. Demo paper.
- Stefanie Scherzinger, Alfons Kemper: *Syntax-directed Transformations of XML Streams.* In Proc. PLAN-X 2005.
- Christoph Koch, Dan Olteanu, and Stefanie Scherzinger: *Building a Native XML-DBMS as a Term Project in a Database Systems Course.* In Proc. XIME-P 2006.
- Christoph Koch, Stefanie Scherzinger: *Attribute Grammars for Scalable Query Processing on XML Streams.* VLDB Journal, p. 317–342, Volume 16, Number 3 / July 2007.
- Michael Schmidt, Stefanie Scherzinger, and Christoph Koch. Combined Static and *Dynamic Analysis for Effective Buffer Minimization in Streaming XQuery Evaluation.* In Proc. ICDE 2007.
- Stefanie Scherzinger. *Bulk Data in Main Memory-based XQuery Evaluation.* In Proc. XIME-P 2007.
- Christoph Koch, Stefanie Scherzinger, and Michael Schmidt. *The GCX System: Dynamic Buffer Minimization in Streaming XQuery Evaluation.* In Proc. VLDB 2007. Demo paper.
- Christoph Koch, Stefanie Scherzinger, and Michael Schmidt. *XML Prefiltering as a String Matching Problem.* To appear in Proc. ICDE 2008.

List of Figures

Bibliography

[1] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. 1990.

[2] A. V. Aho and M. J. Corasick. "Efficient string matching: An Aid to bibliographic search". *CACM*, 18(6):333–340, 1975.

[3] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. I: Parsing*, volume 1. Prentice-Hall, 1972.

[4] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. "Structural Joins: A Primitive for Efficient XML Query Pattern Matching". In *Proc. ICDE'02*, pages 141–152, 2002.

[5] Alioth Debian.org. "Computer Language Shootout", 2007. http://shootout.alioth.debian.org/.

[6] M. Altinel and M. Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In *Proc. VLDB'00*, pages 53–64, 2000.

[7] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. "Texquery: A Full-Text Search Extension to XQuery". In *Proc. WWW'04*, pages 583–594, 2004.

[8] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. "STREAM: The Stanford Stream Data Manager". *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[9] S. Ashwin, P. Roy, S. Seshadri, A. Silberschatz, and S. Sudarshan. "Garbage Collection in Object Oriented Databases Using Transactional Cyclic Reference Counting". In *Proc. VLDB'97*, pages 366–375, 1997.

[10] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. F. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. B. Zdonik. "Retrospective on Aurora". *VLDB J.*, 13(4):370–383, 2004.

[11] 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.

[12] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. "Buffering in Query Evaluation over XML Streams". In *Proc. PODS'05*, pages 216–227, 2005.

[13] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. "Streaming XPath Processing with Forward and Backward Axes". In *Proc. ICDE'03*, pages 455–466, 2003.

[14] M. Benedikt, W. Fan, and F. Geerts. "XPath Satisfiability in the Presence of DTDs". In *Proc. PODS'05*, pages 25–36, 2005.

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

[16] R. Book, S. Even, S. Greibach, and G. Ott. "Ambiguity in Graphs and Expressions". *IEEE TC*, **20**(2):149–153, Feb. 1971.

[17] S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. "A Query Algebra for Fragmented XML Stream Data". In *Proc. DBPL'03*, pages 195–215, 2003.

[18] R. S. Boyer and J. S. Moore. "A Fast String Searching Algorithm". *CACM*, 20(10):762–772, 1977.

[19] A. Brüggemann-Klein and D. Wood. "One-Unambiguous Regular Languages". *Information and Computation*, **142**(2):182–206, 1998.

[20] N. Bruno, N. Koudas, and D. Srivastava. "Holistic Twig Joins: Optimal XML Pattern Matching". In *Proc. SIGMOD'02*, pages 310–321, 2002.

[21] P. Buneman, M. Grohe, and C. Koch. "Path Queries on Compressed XML". In *Proc. VLDB'03*, pages 141–152, 2003.

[22] G. Busatto, M. Lohrey, and S. Maneth. "Efficient Memory Representation of XML Documents". In *Proc. DBPL'05*, pages 199–216, 2005.

[23] I. Carabus, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. "Extending XQuery with Window Functions". In *Proc. VLDB'07*, pages 75–86, 2007.

[24] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, and J. Robie. "XQueryP: Programming with XQuery". In *Proc. XIME-P'06*, 2006.

[25] J.-M. Champarnaud. "Subset Construction Complexity for Homogeneous Automata, Position Automata and ZPC-Structures". *Theor. Comput. Sci.*, 267(1-2):17–34, 2001.

[26] R. Chand, P. Felber, and M. N. Garofalakis. "Tree-Pattern Similarity Estimation for Scalable Content-based Routing". In *Proc. ICDE'07*, pages 1016–1025, 2007.

[27] Y. Chen, S. B. Davidson, and Y. Zheng. "An Efficient XPath Query Processor for XML Streams". In *Proc. ICDE'06*, page 79, 2006.

[28] B. Choi, M. Fernandez, and J. Simeon. "The XQuery Formal Semantics: A Foundation for Implementation and Optimization". In *Technical Report MS-CIS-02-25, University of Pennsylvania*, 2002.

[29] P. Cimprich and O. B. et al. "Streaming Transformations for XML (STX)", 2004. Available at http://stx.sourceforge.net.

[30] B. Commentz-Walter. "A String Matching Algorithm Fast on the Average". In *Proc. ICALP'79*, pages 118–132, 1979.

[31] E. Curtmola, S. Amer-Yahia, P. Brown, and M. Fernández. "GalaTex: A Conformant Implementation of the XQuery Full-Text Language". In *Proc. WWW'05*, pages 1024–1025, 2005.

[32] DataDirect. "DataDirect XQuery". http://www.datadirect.com.

[33] N. Dershowitz and J.-P. Jouannaud. "Rewrite systems". *Handbook of theoretical computer science (vol. B): Formal models and semantics*, pages 243–320, 1990.

[34] A. Deutsch and V. Tannen. "Reformulation of XML Queries and Constraints". In *Proc. ICDT'03*, pages 225–241, 2003.

[35] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. "Path Sharing and Predicate evaluation for High-Performance XML Filtering". *ACM TODS*, 28(4):467–516, 2003.

[36] Y. Diao, P. Fischer, M. J. Franklin, and R. To. "YFilter: Efficient and Scalable Filtering of XML Documents". In *Proc. ICDE'02*, pages 341–342, 2002.

[37] Y. Diao, D. Florescu, D. Kossmann, M. J. Carey, and M. J. Franklin. "Implementing Memoization in a Streaming XQuery Processor". In *Proc. XSym'04*, pages 35–50, 2004.

[38] J. Earley. "An Efficient Context-free Parsing Algorithm". *CACM*, 13(2):94–102, 1970.

[39] R. Edwards and S. Hope. "Persistent DOM: An Architecture for XML Repositories in Relational Databases". In *Proc. IDEAL'00*, pages 416–421, 2000.

[40] P. Fankhauser, T. Groh, and S. Overhage. "XQuery by the Book: The IPSI XQuery Demonstrator". In *Proc. EDBT'02*, pages 742–744, 2002.

[41] L. Fegaras. "The joy of SAX". In *Proc. XIME-P'04*, 2004.

[42] L. Fegaras, R. Dash, and Y. Wang. "A Fully Pipelined XQuery Processor". In *Proc. XIME-P'06*, 2006.

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

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

[45] M. Fernandez, J. Siméon, and P. Wadler. "A Semi-monad for Semi-structured Data". In *Proc. ICDT'01*, pages 263–300, 2001.

[46] M. F. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. "Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions". In *Proc. DEXA'05*, pages 554–563, 2005.

[47] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. "Natix: A Technology Overview". In *Proc. Web, Web-Services, and Database Systems'02*, pages 12–33, 2002.

[48] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proc. WebDB'00*, pages 125–136, 2000.

[49] 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.

[50] A. Frisch and K. Nakano. "Streaming XML Transformation Using Term Rewriting". In *Proc. PLAN-X'07*, pages 2–13, 2007.

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

[52] G. Ghelli, C. Re, and J. Simeon. "XQuery!: An XML Query Language with Side Effects". In *Proc. EDBT Workshops'06*, pages 178–191, 2006.

[53] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML streams with deterministic automata and stream indexes". *TODS*, 29(4):752–788, 2004.

[54] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. ICDT'03*, pages 173–189, 2003.

[55] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Modern Compiler Design.* John Wiley & Sons, 2000.

[56] T. Grust, J. Rittinger, and J. Teubner. "eXrQuy: Order Indifference in XQuery". In *Proc. ICDE'07*, pages 226–235, 2007.

[57] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. "XRANK: Ranked Keyword Search over XML Documents". In *Proc. SIGMOD'03*, pages 16–27, 2003.

[58] J. Hidders and P. Michiels. "Avoiding Unnecessary Ordering Operations in XPath". In *Proc. DBPL'03*, pages 54–70, 2003.

[59] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. “Massively multi-query join processing in publish/subscribe systems”. In *Proc. SIGMOD’07*, pages 761–772, 2007.

[60] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA USA, 1979.

[61] H. Hosoya and B. C. Pierce. “XDuce: A statically typed XML processing language”. *ACM TOIT*, 3(2):117–148, 2003.

[62] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. “TIMBER: A native XML database”. *The VLDB Journal*, 11(4):274–291, 2002.

[63] V. Josifovski, M. Fontoura, and A. Barta. “Querying XML streams”. *The VLDB Journal*, 14(2):197–210, 2005.

[64] H. Katz, D. Chamberlin, M. Kay, P. Wadler, and D. Draper. *“XQuery from the Experts: A Guide to the W3C XML Query Language”*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[65] D. E. Knuth, J. James H. Morris, and V. R. Pratt. “Fast Pattern Matching in Strings”. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[66] C. Koch. “Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach”. In *Proc. VLDB’03*, pages 249–260, 2003.

[67] C. Koch. “On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values”. In *Proc. PODS’05*, pages 84–97, 2005.

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

[69] C. Koch and S. Scherzinger. “Attribute Grammars for Scalable Query Processing on XML Streams”. *VLDB Journal*, 16(3):317–342, 2007.

[70] 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.

[71] 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 *CoRR Technical Report cs.DB/0406016*, 2004.

[72] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah. “TelegraphCQ: An Architectural Status Report”. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

[73] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. “StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures”. In *Proc. VLDB’05*, pages 1259–1262, 2005.

[74] J. R. Levine, T. Mason, and D. Brown. *lex & yacc* . O’Reilly & Associates, Inc., 1992. Second Edition.

[75] X. Li and G. Agrawal. “Efficient evaluation of XQuery over streaming data”. In *Proc. VLDB’05*, pages 265–276, 2005.

[76] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. “A Transducer-Based XML Query Processor”. In *Proc. VLDB’02*, pages 227–238, 2002.

[77] A. Marian and J. Siméon. “Projecting XML Documents”. In *Proc. VLDB’03*, pages 213–224, 2003.

[78] N. May, S. Helmer, and G. Moerkotte. “Strategies for query unnesting in XML databases”. *TODS*, 31(3):968–1013, 2006.

[79] D. Megginson and D. Brownell. “Simple API for XML (SAX)”. http://www.saxproject.org/.

[80] “MemBeR: XQuery Micro-Benchmark Repository”. http://monetdb.cwi.nl/xml/.

[81] “MonetDB/XQuery”. http://monetdb.cwi.nl/XQuery/.

[82] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. “Query Processing, Approximation, and Resource Management in a Data Stream Management System”. In *Proc. CIDR’03*, 2003.

[83] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. “The Niagara Internet Query System”. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[84] D. Olteanu. “Forward Node-Selecting Queries Over Trees”. *ACM TODS*, 32(1):3, 2007.

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

[86] P. E. O’Neil, E. J. O’Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. “ORDPATHs: Insert-Friendly XML Node Labels”. In *Proc. SIGMOD’04*, pages 903–908, 2004.

[87] F. Peng and S. S. Chawathe. “XSQ: A streaming XPath engine”. *ACM TODS*, 30(2):577–623, 2005.

[88] C. Pitcher. “Visibly Pushdown Expression Effects for XML Stream Processing”. In *Proc. PLANX’05*, 2005.

[89] “XMLData Repository, University of Washington”, 2002. http://www.cs.washington.edu/research/xmldatasets/.

[90] “Qizx/open”. http://www.axyana.com/qizxopen/.

[91] P. Rao and B. Moon. “PRIX: Indexing And Querying XML Using Prüfer Sequences”. In *Proc. ICDE’04*, pages 288–300, 2004.

[92] C. Re, J. Simeon, and M. F. Fernandez. “A Complete and Efficient Algebraic Compiler for XQuery”. In *Proc. ICDE’06*, page 14, 2006.

[93] “Saxon”. http://saxon.sourceforge.net/.

[94] S. Scherzinger. “Scalable Query Processing on XML Streams”. Diploma thesis, University of Passau, Germany, 2004.

[95] S. Scherzinger. “Bulk Data in Main Memory-based XQuery Evaluation”. In *Proc. XIME-P’07*, 2007.

[96] S. Scherzinger and A. Kemper. “Syntax-directed Transformations of XML Streams”. In *Proc. PLAN-X’05*, 2005.

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

[98] A. C. Snoeren, K. Conley, and D. K. Gifford. “Mesh Based Content Routing using XML”. In *Proc. SOSP’01*, pages 160–173, 2001.

[99] B. Stegmaier. *“Query Processing on Data Streams”*. PhD thesis, Technische Universität München, 2006.

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

[101] H. Su, E. A. Rundensteiner, and M. Mani. "Automaton meets Algebra: A Hybrid Paradigm for XML Stream Processing". *DKDE*, 59(3):576–602, 2006.

[102] M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. "Processing Text Files as Is: Pattern Matching over Compressed Texts, Multi-byte Character Texts, and Semi-structured Texts". In *Proc. SPIRE'02*, 2002.

[103] M. Theobald, R. Schenkel, and G. Weikum. "TopX - Efficient and Versatile Top-k Query Processing for Text, Semistructured, and Structured Data". In *Proc. BTW'07*, pages 475–485, 2007.

[104] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. "Exploiting Punctuation Semantics in Continuous Data Streams". *TKDE*, 15(3):555–568, 2003.

[105] United States National Library of Medicine. "MEDLINE Sample NLM Data", 2006. http://www.nlm.nih.gov/bsd/sample_records_avail.html.

[106] L. Villard and N. Layada. "iXSLT: An Incremental XSLT Transformation Processor for XML Document Manipulation". In *WWW'02*, pages 213–224, 2002.

[107] A. Vyas, M. F. Fernández, and J. Siméon. "The Simplest XML Storage Manager Ever". In *Proc. XIME-P'04*, pages 37–42, 2004.

[108] H. Wang, S. Park, W. Fan, and P. S. Yu. "ViST: a dynamic index method for querying XML data by tree structures". In *Proc. SIGMOD'03*, pages 110–121, 2003.

[109] B. W. Watson and G. Zwaan. "A taxonomy of sublinear multiple keyword pattern matching algorithms". *Sci. Comput. Program.*, 27(2):85–118, 1996.

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

[111] World Wide Web Consortium. "Document Object Model (DOM)". http://www.w3.org/DOM/.

[112] World Wide Web Consortium. "SPARQL Query Language for RDF (SPARQL)". http://www.w3.org/TR/rdf-sparql-query/.

[113] World Wide Web Consortium. "XML Path Language (XPath)". http://www.w3c.org/TR/xpath/.

[114] World Wide Web Consortium. "XML Query (XQuery)". http://www.w3c.org/XML/query/.

[115] World Wide Web Consortium. "XSL Transformations (XSLT)". http://www.w3.org/TR/xslt/.

[116] World Wide Web Consortium. "XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Candidate Recommendation 8 June 2006, 2002. http://www.w3.org/TR/query-algebra/.

[117] World Wide Web Consortium. "Extensible Markup Language (XML)", 2006. http://www.w3.org/XML/.

[118] E. Wu, Y. Diao, and S. Rizvi. "High-performance Complex Event Processing over Streams". In *Proc. SIGMOD'06*, pages 407–418, 2006.

[119] Y. Wu, J. M. Patel, and H. V. Jagadish. "Structural Join Order Selection for XML Query Optimization". In *Proc. ICDE'03*, pages 443–454, 2003.

[120] "XBench – A Family of Benchmarks for XML DBMSs", 2002. http://se.uwaterloo.ca/~ddbms/projects/xbench/.

[121] “The Xerces Dom Parser”, 2006. http://xml.apache.org/.

[122] “XMark”. http://monetdb.cwi.nl/xml/.

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