University of Kaiserslautern
Department of Computer Science
Databases and Information Systems Group
Prof. Dr.-Ing. Dr. h.c. Theo Härder

Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems

A Dissertation

by

Dipl.-Inf. Christian Mathis

Supervisor:

Prof. Dr. Theo Härder

April 2009

Abstract

The eXtensible Markup Language was designed as a technique for document representation and data exchange. With the success of this meta language, the volume of data represented in XML grew steadily, resulting in large document collections. Keeping such collections serialized as text in files or as BLOBs in relational database management systems is clearly a bad idea. The process of parsing the relatively verbose XML representation upon access is too expensive. Furthermore, loading large XML instances into main memory is often not viable and multi-user access with updates cannot be efficiently supported without dedicated access mechanisms to document stubstructures. Therefore, tailored XML database management systems are required that can compactly encode XML documents, that enable the transfer of substructures of a document into main memory, and provide for isolation techniques. The XML Transaction Coordinator (XTC) developed at the University of Kaiserslautern is such an XML database management system (XDBMS). XTC is a so-called *native* XDBMS, because all its internal structures are tailored to XML storage and processing (in contrast to systems that map XML to relational tables for storage and processing).

In the past, the development of a declarative, set-based interface to access data stored in a DBMS (e. g., SQL for relational systems) was a key ingredient for the success of database systems in general. For XML, the lingua franca for declarative data access is XQuery. Therefore, the idea of implementing XQuery as the declarative interface of an XDBMSs suggests itself. In essence, this thesis presents concepts and techniques for the implementation of an XQuery engine which is tightly coupled with—and embedded into—a native XDBMS. It describes all stages of the query evaluation process: from parsing over query normalization, type checking, query simplification, query rewriting, and plan generation to evaluation and the final result materialization. Of course, the just sketched "query processing pipeline" strongly reminds us of relational query processors and, in fact, this work borrows quite some number of concepts, as we will see. However, the semantically richer XML data model and the XQuery language pose enough interesting problems to justify a thesis of this extent.

The query processing pipeline can roughly be split up into a logical or system-independent part and into a physical or system-dependent part. At the logical level, parsing, normalization, and type checking are standard problems with standard solutions. Even for XQuery, elaborative specifications already exist. Therefore, we do not discuss these stages in deeper detail. Query simplification aims at the removal of semantically irrelevant subexpressions. Therefore, query simplification can be seen as an "insurance" against badly designed queries (wherever they might come from). Our implementation of query simplification operates at a syntactical level only and is just a proof of concept.

For semantic query rewriting, an efficient and expressive internal query representation is required. For that purpose, we develop the so-called XML Query Graph Model (XQGM), in which a query is represented as a box-and-arrow diagram. Query rewriting is then imple-

mented using a rule engine which can be flexibly parameterized by a set of rewriting rules. The rule engine matches rule pattern definitions against the XQGM and, if a match is found, fires the rule's transformation code. The main objectives during rewriting are *query unnesting* and *twig discovery*: Typically, XQuery expressions are highly nested and may contain many correlated subexpressions. Correlated subexpressions imply nested-loops evaluation semantics, which is disadvantageous in many cases. Therefore, the rewriter tries to get rid of nested subexpressions. Furthermore, efficient bulk-processing algorithms—the so-called holistic twig joins—have been developed in the literature. To allow their usage, the rewriter has to discover twig pattern structures in the XQGM. The rewritten result serves as the starting point for the plan generator.

At the physical level, the database layout (access structures) and the existing query processing algorithms (physical algebra) are of major importance. Both issues will be discussed in this thesis: The database layout defines how documents are encoded on external memory and it provides secondary access structures. The encoding developed in this thesis virtualizes the inner structure of a document to avoid the storage of redundant XML substructures. Furthermore, it collects and provides path information, thus, facilitating the creation and maintenance of path indexes. The indexes developed in this work can answer path queries with an optional content predicate. Similar to relational indexes, they are optional and can be adjusted by the database administrator w. r. t. a query workload.

The physical algebra operates on the database layout. It contains all necessary algorithms to evaluate an XQuery expression. An important algorithm of the physical algebra is the holistic twig join operator. This algorithm is capable of efficiently matching a path pattern against a document. Twig join algorithms have been developed in the literature as early as 2002. In this work, we extend the expressiveness of a well-known algorithm to broaden its potential use for query processing. We furthermore show how the results obtained from path index scans can be processed by the algorithm.

For some logical operators (in the XQGM), multiple physical alternative implementations exist. Among them, the plan generator has to choose the most promising. Because a cost model and statistics on the stored documents are still missing in the XTC system, we content ourselves with a heuristics-based plan generator. Finally, the query is evaluated. Evaluation is purely implemented on logical node references, i. e., actual nodes and subtrees are not read from external memory until the result is to be materialized.

Because XML is a hot research topic, XTC is not alone. Quite a number of other systems with a similar focus have been developed, like Tamino, Natix, Timber, MonetDB/XQuery, IBM DB2 pureXML, or Galax. In the related work sections of this thesis, we will compare the techniques developed in the XTC system with some of these competitors. As we will see, among the research projects, XTC is the system with the most intuitive internal query representation, the richest physical algebra, and the most flexible twig join algorithm.

This thesis concludes with an empirical analysis of the storage and query processing techniques developed in this thesis. All concepts have been implemented in the XTC system in a way, such that the next big step in query processing can be tackled, namely cost-based query optimization.

Zusammenfassung

Anfangs wurde die *eXtensible Markup Language* als Format zur Repräsentation von Dokumenten und als Datenaustauschformat entworfen. Mit der Zeit und mit dem wachsendem Erfolg dieser *Meta*-Sprache wuchs auch das vorhandene Volumen an Daten, die in XML repräsentiert wurden. Als Folge entstanden große Kollektionen von XML-Dokumenten. Solche Kollektionen in einfachen Text-Dateien oder in den von relationalen Systemen angebotenen BLOBs zu verwalten, ist offensichtlich keine gute Idee. Allein das Parsen der relativ aufwändigen XML-Repräsentation bei jedem Zugriff auf ein Dokument ist schon zu teuer. Außerdem ist es oft nicht möglich, sehr große XML-Instanzen in den begrenzten Hauptspeicher zu laden, und ein effizienter Mehrbenutzerbetrieb mit Änderungsoperationen lässt sich, ohne den dedizierten Zugriff auf Teile eines Dokuments, auch nur schwer realisieren. Hier können speziell zugeschnittene XML-Datenbankmanagementsysteme Abhilfe schaffen. Im Allgemeinen erlauben solche Systeme eine kompakte Repräsentation von XML-Daten auf dem Externspeicher. Außerdem gewähren sie Zugriff auf Substrukturen und können Isolationstechniken für den Mehrbenutzerbetrieb anbieten. Der *XML Transaction Coordinator* (XTC) ist solch ein XML-Datenbankmanagemensystem (XDBMS). Er wurde an der Technischen Universität Kaiserslautern entwickelt und fällt in die Klasse der sogenannten *nativen* XDBMSe. Diese zeichnen sich durch speziell zugeschnittene interne Speicher- und Verarbeitungsstrukturen aus (im Gegensatz zu solchen Systemen, die XML-Daten auf relationale Tabellen abbilden, um sie zu speichern und zu verarbeiten).

Der Erfolg von Datenbanksystemen im Allgemeinen lässt sich unter anderem auch auf die Entwicklung einer deklarativen, mengenbasierten Schnittstelle zurückführen (denken wir zum Beispiel an SQL für relationale Systeme). Die allgemein akzeptierte deklarative Anfragesprache für XML ist XQuery. Es ist daher naheliegend, XQuery als deklarative Schnittstelle für ein XML-Datenbanksystem zu implementieren. Im Wesentlichen präsentiert die vorliegende Arbeit Konzepte und Techniken zur Implementierung einer XQuery-Engine, die eng an ein natives XDBMS angeschlossen ist. Die Arbeit beschreibt alle Phasen des Anfrageauswertungsprozesses: Angefangen beim Parsen über die Normalisierung, Typüberprüfung, Anfragevereinfachung, logische Optimierung und Plangenerierung bis hin zur Ausführung und dem abschließenden Materialisieren des Ergebnisses. Natürlich erinnert uns die soeben beschriebene "Auswertungs-Pipeline" stark an relationale Anfrageprozessoren und, wie wir sehen werden, lassen sich einige Konzepte auch wiederverwenden. Auf der anderen Seite sorgen das semantisch reichere XML-Datenmodell und die XQuery-Anfragesprache für genügend interessante Probleme, die eine Ausarbeitung dieser Größe rechtfertigen.

Wie im relationalen Fall kann auch hier die Anfrageverarbeitung in einen logischen oder systemunabhängigen Teil und in einen physischen oder systemabhängigen Teil aufgeteilt werden. Auf der logischen Ebene stellen das Parsen, die Normalisierung, und die Typüberprüfung Standardprobleme dar, für die es – auch für XQuery – Standardlösungen gibt. Da-

her schenken wir diesen Themen im Folgenden weniger Beachtung. Die Anfragevereinfachung zielt darauf ab, semantisch nicht relevante Teilanfragen aus einer Anfrage zu entfernen. Die Vereinfachung kann als "Versicherung" gegen ungünstig gestellte Anfragen gesehen werden (wo immer diese Anfragen auch herkommen mögen). Die in dieser Arbeit beschriebene Vereinfachungskomponente arbeitet lediglich auf einer syntaktischen Ebene und stellt eine Proof-of-Concept-Lösung dar.

Für die anschließende logische oder auch algebraische Anfrageoptimierung (engl. "Query Rewriting") wird eine effiziente und ausdrucksmächtige interne Anfrage-Repräsentation benötigt. Dazu wird in dieser Arbeit das sogenannte XML-Anfragegraphmodell (engl, "XML Query Graph Model" oder kurz XQGM) entwickelt. Eine Anfrage wird in XQGM als ein Diagramm bestehend aus Rechtecken und Pfeilen dargestellt. Die eigentliche Anfrageoptimierung ist dann mit Hilfe einer Regel-Maschine implementiert, welche durch eine Menge von Optimierungsregeln konfiguriert werden kann. Wann immer die Regel-Maschine das in einer Regel definierte Muster in einer XQGM-Instanz findet, führt sie die zur Regel gehörigen Transformationsanweisung aus, welche die XQGM-Instanz entsprechend modifiziert. Bei der Optimierung sind die Entschachtelung von Anfragen und das Auffinden von sogenannten *Twigs* die hauptsächlichen Ziele: Typischerweise sind XQuery-Anfragen stark in sich geschachtelt, wobei viele korrelierte Unteranfragen existieren. Diese korrelierten Unteranfragen implizieren bei der späteren Auswertung eine in vielen Fällen unerwünschte "Nested-Loops"-Semantik. Deshalb versucht der Optimierer, solche Teilanfragen umzuschreiben. Darüber hinaus wurden in der Literatur effiziente Blockauswertungsverfahren, die sogenannten *Holistic Twig Joins*, entwickelt. Um diese Algorithmen einsetzen zu können, müssen gewisse Pfadmuster (*Twigs*) in der Anfrage erkannt werden. Das Ergebnis der Optimierung dient dann als Ausgangspunkt für den Prozess der Plangenerierung.

Auf der physischen Ebene spielen das Layout der Datenbank (Zugriffsstrukturen) und die existierenden Algorithmen zur Anfrageauswertung eine übergeordnete Rolle. Beide Themen werden in dieser Arbeit ausführlich diskutiert: Das Datenbank-Layout definiert, wie Dokumente auf dem Externspeicher abgelegt werden und welche sekundären Zugriffspfade angeboten werden. Das Externspeicherabbildungsverfahren, welches hier vorgestellt wird, virtualisiert die innere Struktur der Dokumente, um so das Abspeichern von redundanten Teilstrukturen zu vermeiden. Desweiteren sammelt das Verfahren Pfadinformationen und bietet diese zum Beispiel der Indexierungskomponente zum Erstellen und zur Wartung von Pfadindexen bereit. Die in dieser Arbeit entwickelten Indexierungsverfahren können einfache Pfadanfragen mit einem optionalen inhaltsbasierten Prädikat beantworten. Ähnlich zu korrespondierenden relationalen Verfahren, sind unsere XML-Indexe optional und können bei Bedarf vom Datenbank-Administrator zur Optimierung einer Anfragemenge definiert werden.

Auf dem Datenbank-Layout arbeitet die physische Algebra. Sie beinhaltet alle notwendigen Algorithmen zur Auswertung von XQuery-Anfragen. Ein wichtiger Vertreter ist hier der *Holistic Twig Join*, welcher es ermöglicht, ein komplettes Pfadmuster effizient auf einem Dokument auszuwerten. Die ersten Twig-Algorithmen wurden schon früh (2002) in der Literatur publiziert. In dieser Arbeit erweitern wir die Ausdrucksmächtigkeit eines bekannten Twig-Algorithmus, um die Menge seiner Einsatzmöglichkeiten bei der Anfrageauswertung zu erweitern. Darüber hinaus zeigen wir, wie Pfadindexe effizient mit Twig-Algorithmen kombiniert werden können.

Für logische Operationen (im XQGM) existieren häufig viele verschiedene physische Implementierungsmöglichkeiten. Aus ihnen muss der Plangenerator diejenigen aussuchen, die

die niedrigsten Auswertungskosten versprechen. Leider fehlen im XTC-System bisher ein Kostenmodell und eine Statistik-Komponente, sodass wir uns hier mit einer heuristischen Operatorauswahl begnügen müssen. Zum Schluss wird der generierte Anfrageplan ausgeführt. Diese Ausführung ist ausschließlich mit Hilfe von Knotenreferenzen implementiert, das heißt, XML-Knoten und Teilbäume werden erst dann vom Externspeicher gelesen, wenn das Ergebnis materialisiert wird.

XML ist ein sehr aktives Forschungsthema, weshalb XTC nicht alleine dasteht. Parallel zu XTC haben sich einige weitere Systeme entwickelt, wie zum Beispiel Tamino, Natix, Timber, MonetDB/XQuery, IBM DB2 pureXML oder auch Galax. In den Teilkapiteln zu verwandten Arbeiten werden diese Ansätze mit den in XTC implementierten verglichen. Wie wir sehen werden, ist XTC – unter den wissenschaftlich entstandenen Prototypen – dasjenige System mit der intuitivsten internen Anfragerepräsentation, mit der reichhaltigsten physischen Algebra und mit dem flexibelsten Twig-Algorithmus. Die vorliegende Arbeit schließt mit einer empirischen Analyse der vorstellten Speicherungs- und Verarbeitungsverfahren. Alle Konzepte wurden so im XTC-System implementiert, dass der nächste große Schritt in der XML-Anfrageverarbeitung angegangen werden kann, nämlich die kostenbasierte Optimierung.

Contents

Figures

Listings

Part I

Introduction

Chapter 1 Motivation

We will either find a way, or make one!

Hannibal

At the time of writing, the eXtensible Markup Language (XML) [Bray 06] celebrates its 11th anniversary. In 1998, XML was designed as a format for large-scale electronic publishing, where it played its role as a mechanism to structure the content of text documents in a meaningful way and, therefore, served to separate presentation from content. However, because of the flexibility of its underlying semi-structured data model, XML was soon discovered as a general format for electronic data interchange. With its roots, both in the document community *and* in the data processing community, XML rapidly conquered many applications areas in the last decade. Perhaps XML can nowadays be characterized best by enumerating the operations that can possibly be applied to XML and by the systems that are built on top of the markup language. For example, XML is stored, shredded, reconstructed, transformed, queried, retrieved, validated, sent over the wire, navigated, scanned, modified, standardized, imported, exported, cleansed, summarized, indexed, compressed, shared, sampled, archived, encrypted, linked, etc. Furthermore, XML serves as the basis for quite diverse software systems like, for example, in the area of web content management, service-oriented integration, metadata handling, and healthcare, and has lead to a plethora of government, education, and industry standards [Cover 05].

The success and attention XML has gained over the last ten years has various reasons. The following often stated XML characteristics are the most crucial key factors:

- *XML is human readable.* XML documents are text documents. Everybody can open an XML document in his favorite editor and read or modify the document's contents. This is not the case in many other proprietary formats (for example in electronic data interchange). Therefore, XML keeps a certain proximity to the user.

- *XML separates content from presentation.* In contrast to other markup languages (such as HTML or Latex), the XML technology allows to model the data first and to transform this data later into an appropriate representation. Therefore, XML provides for a certain kind of data independence.

- *XML is open, standardized, platform independent, and supported by a large variety of tools.* The combination of these four factors lead to a high acceptance of XML. Often, it is very simple to build an XML application, because for all imaginable

tasks, appropriate tools exist, for example: to define the structure of an XML document (XML Schema), to navigate (DOM), scan (SAX), and query XML (XQuery), to store XML (XML-enabled database systems), etc. Because XML is open, platform independent, and standardized, everybody can "speak" and "understand" XML, which is quite important for *Business-to-Business* (B2B) applications.

- *XML is hierarchical, flexible, and extensible.* XML defines a semi-structured data model, representing data as trees. In contrast to previous data models, where records had to conform to a specific schema, in XML, a schema is optional, thereby allowing the complete variety from completely unstructured to highly structured data. For example, in XML it is not necessary to explicitly state that a particular information is not available (as required in the relational model with NULL values) the particular subtree is simply missing in the XML tree. Furthermore, because XML is actually a meta-language allowing the specification of all forms of XML dialects, XML is extensible.
- *XML contains data and metadata.* Again, in contrast to previous data models where data and meta data (i. e., data about data) are separated, XML unites them into a single document, thus facilitating data interchange[1].

Despite these positive points, XML has also a downside: Often, the technology has been criticized as a bandwidth and CPU-time squanderer, because the representation containing a lot of redundancy is very verbose. Furthermore, there is no comparable technique as the relational schema normalization theory for XML documents, there are at most some best practices. Additionally, because schemas are optional and because the semi-structured data model is complex, data management (especially data access) becomes a lot more complicated. However, XML is still young and industries as well as researchers from all over the world are constantly improving the technology.

1.1 Objectives

All in all, the heavy use of XML in the past decade led to a large volume of XML data that requires efficient management. At this point, database management systems (DBMSs) step into the picture. Database management systems arose in the early 1960s to face the problems that file-based data management was suffering from, for example redundancy and inconsistency. DBMSs were introduced as a software component between the application and the operating system (file system) to control data access. Today, after 40 years of research, it is impossible to imagine information technology without DBMSs, which provide a long list of appealing features, in particular: logically centralized data management, data independence, simple application programming interfaces (APIs), centralized integrity control, transactional data processing, efficient and parallel processing of large data volumes, high availability and dependability, etc. Furthermore, because DBMSs are generic software components, they also conquered a large area of applications.

When large volumes of XML data need to be managed, it is quite natural to marry XML with the DBMS technology to bring an XML-enabled database management

[1]Note, this feature is often stated by the phrase "XML is self-descriptive". In our opinion, this statement leads into the wrong direction, because it implies that computers can automatically "understand" XML data, which is obviously not the case.

system (XDBMS) into being that combines the benefits from both worlds. Such a system can, for example, store and retrieve XML documents, allow users to collaboratively share XML documents, control integrity and schema constraints upon modification, and provide efficient means to access, transform, and query XML. As for any other DBMS, the following statement also holds from XDBMSs: "Performance is not everything, but without performance everything is worth nothing" [Härder 05a]. One of the most performance-critical component of a DBMS is the query engine. Often, the performance of the query engine accounts for the performance of the DBMS itself. The problem is that bad query evaluation plans, i. e., bad access strategies, can lead to a disastrous performance behavior of the complete system. Therefore, the query engine is a key factor for the overall quality of a DBMS.

Query processing—and especially query processing over XML data—is a complex task. In computer science, the answer to complexity is abstraction.[2] Therefore, it is quite natural to split up query processing into system-independent problems (logical abstraction level), and into problems that are system-dependent (physical abstraction level). Considering XML query processing at the physical level, a plethora of evaluation algorithms have been proposed by industry and research groups over the recent years, such as navigational primitives, bulk processing algorithms, and index structures. However, often, these approaches content themselves with solving subproblems (e. g., finding matches of a specific subtree structure in the document) that are important to XML query processing, but they do not reason about the integration of their proposals into a full-fledged query engine. On the other side, at the logical level, there is still no commonly agreed logical treatment of XML queries, in particular, a common logical XML algebra is still missing.

The vision of this thesis is to bring the valuable concepts from both communities, i. e., from physical and logical parties, together and to integrate them in an XML query evaluation engine. Therefore, the following fundamental aspects of query processing shall be developed here and are, therefore, subject of this work:

- *Space-efficient storage and indexing for dynamic documents.* XML documents may occur in all imaginable size and complexity ranges, from small document collections of some kilobytes size, e. g., a bunch of configuration files, to complex structured multi-gigabyte documents, for example in bioinformatics [UniProt 08]. While small documents are probably best processed by simple scans or navigations in main memory, large instances forbid such a strategy and require *indexed access* to keep query performance in tolerable ranges. However, if a document is structured, it has to be stored in a space-efficient manner, because query processing frequently requires substructure reconstruction for result generation. Therefore, succinctness is an important requirement leading to reduced external I/O and logging costs, too. Finally, an XDBMS is not a read-only system, but supports dynamic XML documents, requiring efficient means to update documents and to propagate modifications to secondary index structures.
- *Development and integration of efficient XML evaluation algorithms in a physical XML algebra.* The physical algebra of an XML query engine consists of the low-level evaluation algorithms (operators) that run on top of XML indexes and the document store. Often, for a single task there is no one-fits-all solution, but rather, depending on the queried data, one algorithm performs better than another one.

[2] "Eine Hauptaufgabe der Informatik ist systematische Abstraktion" ("A principal task of computer science is systematic abstraction", H. Wedekind).

Therefore, a physical algebra containing XML-specific operators shall be developed to support a large variety of different data distributions. Research has already developed a large number of such algorithms. However, quite often these approaches are isolated suggestions and do not consider their integration into an XML algebra. In this work, these proposals shall be re-considered, both on a theoretical and an empiric level, and, where appropriate, shall be integrated into the physical algebra such that all operators work hand-in-hand.

- *Design of a query rewriting rule set and an internal query representation.* Often, queries can be pre-optimized at a logical level without knowledge of system-specific details. For example, in most cases it is better to evaluate a predicate as early as possible to reduce the number of items processed. Furthermore, the physical XML algebra will contain some very efficient evaluation algorithms that can, however, only be applied in certain circumstances. In this work, a set of query rewriting rules will be developed that copes with the discovery of these opportunities and with pre-optimization. For the implementation of query rewriting, a suitable (logical) query representation has to be found that efficiently supports the restructurings defined by the rule set.

- *Definition of the necessary query transformations and their integration into a complete query evaluation process.* To deal with the complexity of query evaluation, a query typically passes various stages inside a query engine, from its external string representation over the logical and the physical representations to its execution and result construction. Basically, the transitions between these stages are language transformations which need to be defined in a sound manner and which have to be integrated resulting in an overall query evaluation process.

Unfortunately, a dissertation can never cover a topic completely. This is especially true, when dealing with such a complex task as query processing. Therefore, already at this point, the following important topics left open to future work shall be stated: 1) the development of a document statistics component to estimate result cardinalities of query expressions, 2) the development of a cost model, and 3) the design of a cost-based query optimizer for XML queries. A cost-based query optimizer selects the most promising algorithms from the physical algebra to implement a query. For this selection, the optimizer needs statistical information about the documents (e. g., how are elements distributed?) and ways to name a price for an algorithm (depending on the estimated input size). The first information comes from the statistics component, the second one from the cost model. The development of these components would reveal enough material for another thesis of this extent. Therefore, they are left open to future research. However, this work serves as a foundation for the development of an XML query engine with a cost-based optimizer, because it delivers the basic ingredients: XML storage and index structures, evaluation algorithms, an internal XML query representation, XML query rewriting, and a complete query evaluation process.

To empirically assess the solutions developed for each of the presented aspects, they were implemented in the XML Transaction Coordinator (XTC) [Haustein 06a], a prototype native XDBMS.

1.2 Outline

This dissertation is organized into the five parts 1) *Introduction*, 2) *Logical Aspects of XML Query Processing*, 3) *Physical Aspects of XML Query Processing*, 4) *Experimental Evaluation and Future Research*, and 5) the *Appendix*. After the motivation and the outline section you are currently reading, the introduction part presents necessary preliminaries facilitating the comprehension of the following work. Chapter 2 provides an overview over the concepts developed. It summarizes the query evaluation process and shows by examples how a query is translated from an external string representation into an executable program. Because some parts of the query processor are system-dependent, Chapter 3 presents the the XTC system.

In the second part of this thesis, the logical (system-independent) aspects of XML query processing are introduced. Chapter 4 presents the syntax and semantics of the *XML query graph model* (XQGM)—an internal query representation. Furthermore, it shows how queries can be transformed from their external string representation into XQGM. Then, in Chapter 5, the rewriting rules transforming an XQGM instance into a pre-optimized alternative are discussed. The focus here lies on *query unnesting* and the discovery of the so-called *holistic twig joins* (for which efficient algorithms exist).

The third part of this work deals with physical aspects in XML query processing. A query processor is built upon basic access primitives, which are either implemented directly on the persistent data store (XML document store) or which are implemented over (XML) indexes. Therefore, Chapter 6 (*Document Storage*) presents how XML documents are stored and accessed, whereas Chapter 7 (*XML Indexing*) introduces XML index structures. Chapter 8 (*The Physical Algebra*) then presents a set of physical XML operators (evaluation algorithms) for XML processing and a glimpse on how to map logical XQGM instances to this physical algebra.

In the fourth part, Chapter 9 presents experimental results. This dissertation concludes with Chapter 10 summarizing the main contribution and providing directions future work. Related work will be discussed in each chapter separately.

1.3 Conventions

In this work, XML queries are expressed in the XQuery language. These queries are, whenever they occur, set in a typewriter font, for example: `doc("sample.xml")/bib/book/author`. XQuery functions, when used inside a query, are also set in a typewriter font. However, when they occur in the main text, they are type set like all other functions, by using an italic font style and by omitting braces and arguments, for example: *doc*. Result values of XQuery expressions are represented in a typewriter font, e. g., `<author>Michael Ende</author>`. On the other hand, when we refer to some XML node, we use an italic font style, e. g., "the *author* node". The introduction of new term and concepts is also highlighted in an italic font.

Interface and class definitions are represented inside listings with some Java-like syntax highlighting. For an example, see Listing 6.1 on Page 171. Algorithms are either represented in a Java-like pseudocode or in an algorithmic pseudocode (e. g., Listing 4.1 on Page 59). Note, in the algorithmic pseudocode, the assignment opera-

tor is the left arrow ($\leftarrow$), because the equality operator ($=$) is used for comparisons. All further notations will be introduced, when they are required.

Chapter 2 XML Query Processing on XTC—An Overview

> If you can't describe what you are doing as a process, you don't know what you're doing.
>
> W. Edwards Deming

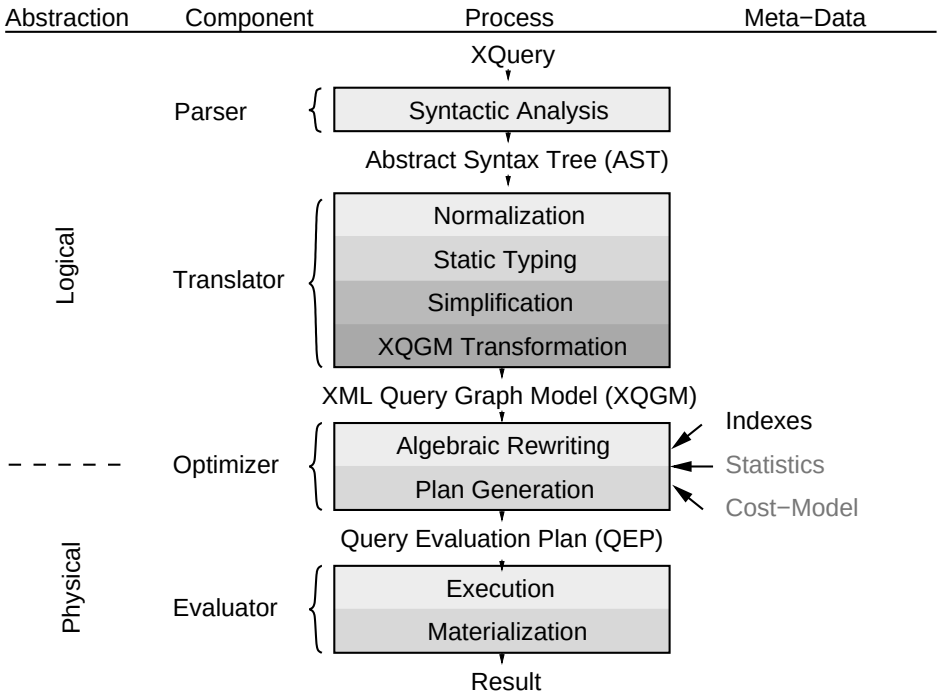
The overall goal in query processing is to translate a declarative query from its *external representation* as a string into a procedural *executable program* that can be evaluated on the data store to deliver the answer of the query. Obviously, the semantics of the query in its external representation and the semantics of the executable program have to be the same, because otherwise the program would not return a correct result. The major distinction between these two query representations is the level of *abstraction*. The external query representation has a high abstraction level, because it only takes the logical structure of the database (in our case the logical structure of one or more XML documents) into account. The executable program however has a low abstraction level, because it is composed of operators that work on the physically stored objects of the database. Therefore, the program has to deal with storage structures, secondary indexes, data distributions, etc. Given these low-level objects, there is often a plethora of different evaluation alternatives for one and the same query. Each of these alternatives has its own access characteristics and costs. To refine the overall goal in query processing from above, for a given external declarative query, query processing has to find the best (or at least *a* good) procedural executable program in terms of processing costs.

The intention of this chapter is to introduce the overall process of XML query evaluation from the external representation of a query in the XQuery language to the execution on the data store. The chapter sketches the concepts developed in this thesis, it provides pointers to the following chapters where specific concepts are discussed in detail, and it generally facilitates the comprehension of this work.

2.1 The Query Evaluation Process

In the late 1980s and in the 1990s, the DB research community spent substantial efforts on the development of *extensible* query processors for database systems. The essential idea behind this movement was to provide for a framework in which new concepts, such as new language constructs, new data models, or new processing al-

Figure 2.1 Query evaluation in XTC



gorithms could easily be integrated without the cumbersome task to completely reimplement large portions of a query processor [Mitschang 95, Kabra 99]. Systems like EXODUS [Graefe 87], VOLCANO [Graefe 93, Graefe 94], and Starburst [Mavis K. Lee 88, Haas 89, Pirahesh 92] are some well-known examples from that time, which all encompass the framework idea. The query processor developed in this thesis stands in the tradition of these systems. Therefore, many concepts and terms could be borrowed, and, although the XTC query processor was built from scratch, it can be seen as a true extension in the sense of the framework idea.

To cope with complexity, query processing is generally split up into a number of stages. Each stage receives a query representation generated by some preceding stage (or given as input) and produces a further representation with a lower level of abstraction but enriched with more specific information on how the query has to be evaluated over the database. Figure 2.1 depicts all the stages of the query evaluation process of the XTC query processor.

2.1.1 Logical and Physical Abstraction

The process can roughly be divided into a *logical abstraction layer* and into a *physical abstraction layer*. The logical layer is completely system independent, in the sense that the query representations and actions at this level could be reused as they are, to implement a query processor for another XML data source. The aim at this layer is 1) to find a procedural internal representation such that semantically equivalent (but syntactically different) queries are mapped onto the same representation (if possible), and 2) to rewrite the query in a way such that intermediate results are minimized. Such a representation is a good starting point for the actions at

the system-dependent physical abstraction layer below, because, in contrast to the declarative external query representation, a procedural internal representation contains more information about how the query can be evaluated. Furthermore, mapping semantically equivalent queries to the same internal representation makes the query processor robust. At the physical layer, the query processor considers low-level issues such as document storage layout, possibly available index structures, or processing algorithms to generate a program that operates on the database and efficiently computes the query result.

In total, the query processor consists of the six components, of which five are depicted in Figure 2.1: the *parser*, the *translator*, the *optimizer*, the *evaluator*, and the *metadata component* of the XTC system. As the illustration indicates, parser, translator, optimizer, and evaluator are connected in a series, each component implementing one or more processing stages as introduced above. As we will see, some functionality can be reused for the implementation of one or the other stage. We combine this functionality in a sixth *infrastructure component*, which is not depicted in Figure 2.1. The logical part of the process starts by a syntactical analysis of the given query and is finshed with the algebraic rewriting stage. The output of this stage serves as the input for the physical query evaluation part, which is completed when the final result is computed.

2.1.2 Parsing and Translation

In the first stage, the parser reads the XQuery expression received as a string and executes a syntactic analysis. If a syntactic error is detected, e. g., a missing closing brace, a message containing some information about the error is reported to the caller of the query processor and the evaluation stops. Otherwise, the parser constructs the so-called *abstract syntax tree* (AST) of the query, which is basically a grammar-oriented tree representation of the query expression. The XQuery grammar is specified by the W3C Recommendation [Boag 04] in the form of EBNF productions. Instead of writing the parser by hand, a parser generator [Parr 07] reads the XQuery grammar and automatically generates the necessary parser code. Because parsing is a standard technique in compiler construction, this issue will not be further discussed in the following.

The next four stages are implemented by the query translator, whose task it is to transform the given AST into an internal representation for the query optimizer. During this transformation, the query is revised from syntactic sugar and from unnecessary subexpressions. This is the first step towards a robust query processor.

Normalization is a process defined by the XQuery Formal Semantics Recommendation[1] [Choi 07]. It transforms an XQuery expression to an equivalent expression in the so-called *XQuery Core Language*, which is a subset of the original XQuery language. XQuery Core has the same expressive power as XQuery, but contains only a minimal number of language constructs, i. e., normalization basically removes all syntactic sugar from a query. Therefore, it facilitates the following translation steps, because a smaller fraction of the XQuery language has to be considered. The normalization process is implemented on the AST by a recursive function that replaces all subexpressions of an expression with their normalized version. Because this implementation slightly differs from the recommendation, normalization will be

[1] Note, in the following, we will refer to this document as the "Formal Semantics".

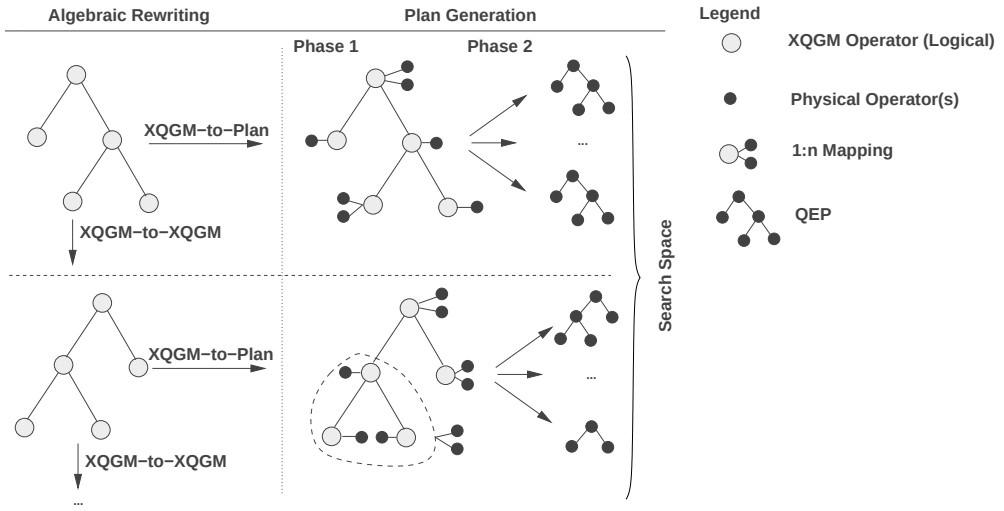
further discussed in Section 4.3.1.

In the next stage, the static type of the query is inferred. XQuery is a strongly typed language, therefore, every XQuery expression returns a typed value from the XQuery Data Model [Fernández 04]. During static typing, all subexpressions are annotated with the static type they return. Certain expressions expect a specific input type. For example, the *plus* operator (+) expects numeric arguments. If, however, the static type of the input expression does not match the expected type, a type error is raised and the evaluation stops. Note, because static typing takes place at compile time, it solely relies on the query but not on the data that is queried. Therefore, it is often only possible to infer some supertype of an expression, but not the specific subtype which will occur later at run time during query evaluation. In summary, static typing can detect "coarse" typing errors, whereas dynamic typing errors have to be detected during evaluation. The rules to infer the static type from a normalized XQuery expression are also defined in the Formal Semantics. Because the query translator implements these rules without substantial changes, static typing is not discussed in detail in this work.

Simplification aims at the removal of subexpressions with no effect. Such redundant constructs are sometimes introduced by programs that automatically generate queries, by users who do so accidentally, and by normalization. Furthermore, for systems allowing to define XML views, the so-called *view expansion*, which replaces a view with its definition in the query, might also introduce redundant subexpressions. Simplification is implemented using the *infrastructure component* of the query processor. Basically, this component interprets a query representation (in this case the AST) as a tree and employs a rule-inference engine to apply tree transformations, such as expression substitutions, defined by *restructuring rules*. A rule is specified by a *pattern* and a *transformation instruction*. When a rule matches the tree representation, the transformation instruction can be applied to the match, typically rewriting the tree at that position. Because the infrastructure component is just an implementational detail, it will not be introduced in detail. Simplification, however, will be discussed in Section 4.3.2.

The last translation stage is the XQGM transformation. So far, we followed the Formal Semantics. In our approach, the Formal Semantics is implemented directly on the AST. However, now that we leave the path of the Formal Semantics, the AST is not an appropriate format anymore, because in an AST, subexpressions are only loosely coupled. This is problematic for the optimizations introduced in this work. As an example for loose coupling consider resolving the connection between a variable declaration (for example in an *in*-binding expression) and its references. This is often complex, because references can occur at any location in the subexpression of the declaration. Furthermore, an AST is grammar-oriented, meaning that only XQuery language constructs can be represented. However, often the query processor considers evaluation strategies beyond of what can be expressed in language constructs. All in all, the AST representation lacks two of three properties postulated in [Mitschang 95] for internal representations, namely *efficiency* and *procedurality*. It can be argued that the third property, *flexibility*, is present, because the AST representation could be easily extended. This does, however, not compensate the other two disadvantages. An internal representation meeting all these requirements for *relational* queries is the so-called *query graph model* (QGM) introduced in the Starburst system [Haas 89, Pirahesh 92]. The QGM was designed for flexibility

Figure 2.2 The query optimization strategy in the XTC XQuery processor (sketched)



such that new language constructs like, for example, SQL recursion, could easily be integrated into the query processor. However, as we will see in this work, the QGM can be extended to support XML query processing. The resulting internal representation is called XQGM for *XML query graph model*. The complete XQGM and the transformation from the AST representation will be discussed in Chapter 4.

2.1.3 Optimization

As we have seen, the Formal Semantics defines three XQuery evaluation stages: *normalization*, *static typing*, and *dynamic evaluation* (the first two were already discussed above). The semantics of the initial XQGM instance generated by the transformation stage is very close to the semantics of the dynamic evaluation, i. e., the same evaluation model is implemented. However, because this evaluation model heavily relies on nested subexpressions and *node-at-a-time* navigational methods, it is often far from being optimal.

Therefore, besides classical algebraic optimizations such as *selection push-down* and *select fusion* to minimize intermediate results and the number of operators required, the *algebraic rewriting* stage tries to unnest queries as far as possible to enable bulk (i. e., *set-at-a-time*) processing. Unnesting substitutes correlated subexpressions by joins, i. e., by bulk operators. Like simplification, algebraic rewriting is also implemented using the infrastructure component. The XQGM instance, although being a graph, is interpreted as a tree structure on which the generic rule engine executes rule-based transformations. Because a rule transforms an XQGM instance into another XQGM instance, such transformations are called *XQGM-to-XQGM transformations* in the following (cf. [Kabra 99]). The left-hand side of Figure 2.2 illustrates the algebraic rewriting stage. The result of the algebraic rewriting stage is an unnested and pre-optimized XQGM instance. At this point, the physical optimization of the query begins and system-specific issues come into play. The algorithmic rewriting stage will be discussed in Chapter 5.

During *plan generation*, the optimizer has to find a program, whose evaluation on the database computes the final result. In the following, we will refer to the component of the optimizer responsible for plan generation as the *plan generator*. The plan generator has a large set of basic XML processing algorithms at his disposal, which are called *physical operators* or just *operators*. The plan generator can stitch different (descriptions of) physical operators together to create a *query evaluation plan* (QEP). A QEP is then a description of the desired program.

Physical operators can roughly be grouped into 1. navigational, join-based, and index-based methods for path matching, and 2. into all remaining operators that are necessary to evaluate selections, projections, grouping, value-based joins, etc. The operators of the first group, which are also called *path processing operators* (PPOs), play a major role in this work, because PPOs access the document (in contrast to the operators in the second group, which merely operate on the intermediate results delivered by path operators). Document access can be expensive, therefore these operators need special attention. The set of all physical operators is called *physical algebra* (PAL) throughout this work. This term was introduced in [Graefe 93] and shall help to distinguish operators from the physical level (algorithms) from operators on the logical level (XQGM).

Figure 2.2 presents how logical XQGM instances are mapped onto physical operators, i. e., how plans are generated. Let us consider the upper part of the picture first: Given an XQGM instance, plan generation is implemented in two stages, the first one of which also relies on the rule engine of the infrastructure component. Here, the rules describe logical-to-physical mappings and are called *XQGM-to-Plan transformations* (similar to [Kabra 99]). Whenever a rule matches, a description of the physical operators implementing the matched XQGM operator is created (represented by the black dots) and attached to the matched logical operator. Considering the relationship between a logical XQGM operator and operators from the physical algebra, the well known 1:1, 1:n, n:1, and n:m cardinalities apply: As illustrated in the overview, sometimes there is only one physical alternative for a logical operator (1:1), sometimes there are more than one alternatives (1:n), and sometimes a group of logical operators is implemented by a (group of) physical operator(s) (n:1 or n:m). In the second stage, the plan generator iterates over the XQGM instance and builds different QEPs from the physical alternatives it finds. Often, the optimizer can create a large variety of structurally different but logically equivalent QEPs for a single XQGM instance. The completeness of all these alternatives is the *search space*.

From all the different QEPs, the query processor now has to decide, which of them is the cheapest in terms of processing costs. The answer to this question depends on a large variety of parameters, such as the optimization goal (e. g., response time, throughput, main-memory usage, etc.), the structural layout of the document, value distributions in the document, the current system state (I/O-bound or CPU-bound), and so on. The applicability of certain physical operators depends on the physical layout of the database, i. e., on document storage and indexing. To introduce the physical aspects of XML query processing, storage and indexing are discussed in Chapters 6 and 7. Then, the physical algebra (with the PPOs) and the plan generation are shown in Chapter 8.

On the Role of the Optimizer in this Thesis

As already stated in the introduction, *cost-based optimization is not considered in this work*, because the necessary statistical component and an operator cost model are still missing (indicated by the light grey font color in Figure 2.1). Therefore, the role of the optimizer in this work is defined as follows:

1. *It has to provide means to generate a large variety of possible plans in the search space.* Actually, at the time of writing, the research community does not exactly know in general, what the best evaluation strategy for a given query over one or more documents is. Therefore, the query optimizer has to deliver a large variety of QEPs, to make them accessible for assessment and the development of a cost model in the future.

2. *It has to provide means to find at least one good plan using heuristics.* Independent of the state of current research, the optimizer should at least find one good evaluation alternative to provide for an embedding into a productive system.

3. *It has to be extensible to seamlessly integrate cost-based query optimization.* A cost-based optimization is future work. Therefore, the query processor should facilitate its integration into the existing heuristics-based optimizer.

To address the first point, the right part of Figure 2.2 comes into play. As the illustration suggests, the plan generator is able to generate plans for every XQGM instance possible, i. e., XQGM-to-Plan transformations can be executed after each XQGM-to-XQGM transformation. In this way, all the physical alternatives for all logical XQGM instances can be generated, thus completely spanning the search space. In Chapter 9, where an experimental analysis of different evaluation strategies is described, this generation technique is applied.

In general, a true cost-based query optimizer requires some kind of book-keeping facility to manage alternative XQGMs and QEPs in main memory. In [Graefe 87], the so-called *mesh* data structure was proposed for this task. A mesh can efficiently manage multiple similar graphs (generated during optimization), by storing equal subgraphs only once. However, because cost-based query processing is out of the scope of this thesis, there is also no need for such a kind of in-memory XQGM or QEP management. Both, for heuristics-based optimization as proposed in the second point and for the generation of all alternatives, only one XQGM representation (on which transformations are executed) is sufficient. Nevertheless, to address the third point, the data structures of XQGM and the physical algebra are designed in a way to facilitate cost-based query optimization without major changes.

In contrast to the discussed optimization process, which separates the algebraic rewriting from plan generation, some contributions [Graefe 94, Graefe 93] suggest to treat logical rewritings (XQGM-to-XQGM) and plan generation rules (XQGM-to-Plan) equally. This means that the optimizer applies both types of rules in one unified rewriting process. As a result, the algebraic rewriting stage is not separated from plan generation anymore. In this way, the cost-based optimization process is extended to algebraic rewriting and is more directed towards the optimization goal, because expensive alternatives can be pruned from the search space much earlier. With XQGM and the physical algebra developed in this work, such an "interleaved" optimization strategy is also realizable. However, because cost-based optimization is not considered, this work simplifies the process and keeps the separation between algebraic rewriting and plan generation intact.

2.1.4 Execution and Materialization

A QEP is a description of a program whose evaluation on the database returns the query result. The question is now how such a QEP is actually evaluated. The literature suggests two possibilities: interpretation vs. code generation [Mitschang 95]. In the first approach, a program called *Interpreter* reads the program description given by a QEP and issues the necessary algorithms that operate on the access system of the database to compute the result. In the second approach, the QEP description is used to generate a code module, which is compiled, loaded, and executed. Interpretation was suggested as the ideal method for ad-hoc user queries, because no compilation time is required. However, compiled programs as generated in the second approach often perform better than interpreted QEPs. To keep query processing simple, most systems only support one strategy.

The XTC query processor does not strictly follow either approach, however, it is logically close to compilation: In contrast to what was introduced above, a QEP is not a description of a program, but it is the program itself. All necessary algorithms (operators) from the physical algebra are simply stitched together during plan generation. The result is an operator tree called *PAL tree* (for *physical algebra tree*) in the following, which is directly executable. Note, because XTC does not distinguish QEPs from PAL trees, we use these terms as synonyms in the following.

The operators of a PAL tree implement the *Open-Next-Close* protocol, resulting in an iterator-like query-execution supporting pipelining. In a first step, the evaluator calls the *open* method on the root of the PAL tree. Recursively, the *open* call is issued on the child and descendant operators. During *open*, all operators have the chance to prepare for evaluation. Some operators can even start to read input data (after calling the *open* method on their children). To actually retrieve the result, the evaluator repeatedly calls the *next* method on the root of the operator tree. In each call, some portion of the result data is computed and returned. When a call to the *next* method returns a NULL value, the result is completely computed. Then the evaluator calls the *close* method, which is also propagated through the tree and signals the operators to clean their internal data structures. Internally, like *open* and *close*, the *next* method recursively calls *next* on child operators to retrieve data, which is then processed. This evaluation model has several advantages: 1) the evaluator does not need to explicitely manage data flow and control flow, 2) when no operator in the tree is a so-called *pipeline breaker*, the first elements of the result can be already returned when the rest is still in computation. A pipeline breaker is an operator that needs to read the complete input of at least one of its child operators, like for example a *sort*.

What happens inside the operators during evaluation heavily relies on the salient features of DeweyIDs. As will be shown in Section 3.1.3, a DeweyID is a node identification mechanism with embedded “structural knowledge”. This knowledge is exploited during query processing, thus reducing expensive document access. In the end, when the execution finishes, the result is a sequence of items that reference resulting document subtrees via DeweyIDs (similar to the Tuple ID concept in relational systems). For external representations, these references have to be dereferenced, which is the task of the final *materialization* stage. Basically, materialization can be implemented either by a scan over the document or by per-DeweyID index lookups that retrieve the resulting subtrees.

After this rather abstract description of the query processor internals, the next section illustrates the sketched techniques on a sufficiently complex example.

2.2 Query Processing by Example

Consider the following query that emanates from the XMark benchmark [Schmidt 02] (Query 5) and returns the number of *price* elements that have a content larger than or equal to "40":

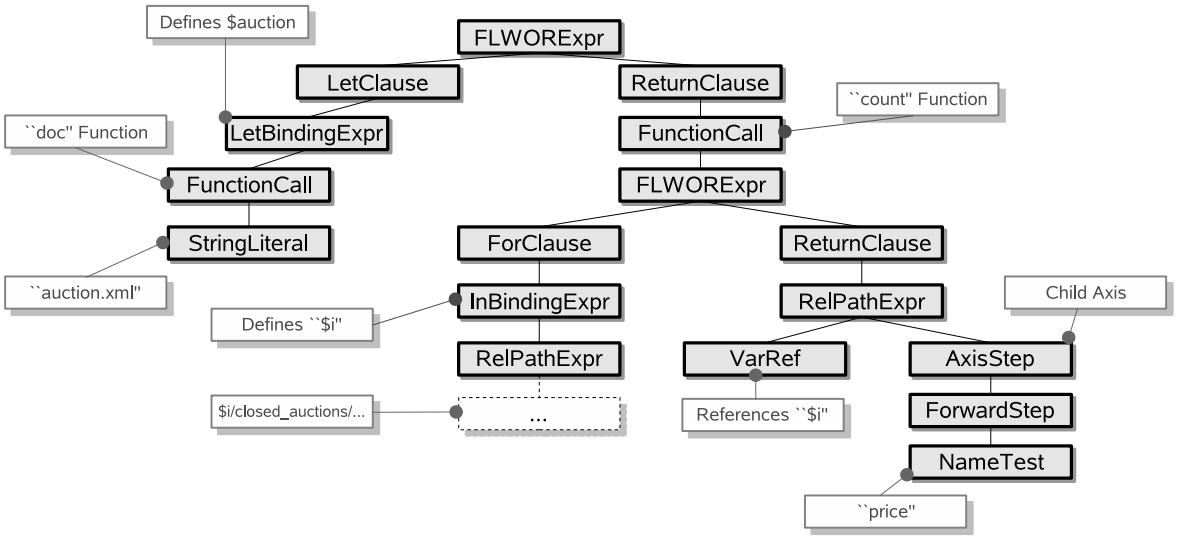
```
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

In the following, the transition of this query through the various query processing stages will be shown.

2.2.1 Syntactic Analysis

The abstract syntax tree produced by the parser for this query consists of roughly 40 nodes. For the sake of brevity, Figure 2.3 does not contain all these nodes, but only a fragment of the complete AST. As you can see, the representation is quite straightforward. Every particle from the XQuery grammar corresponds to a node in the AST. You can also observe the points discussed in Section 2.1.2: the AST is not very useful as an internal representation for query optimization, because it does not easily reveal data flow and control flow (and thus lacks procedurality) and the contained information is only loosely coupled (for example, the variable definitions and their references are not directly interconnected). The latter point would make

Figure 2.3 Abstract syntax tree for XMark query Q5



restructurings in query optimization unnecessarily complex.

2.2.2 Normalization

As introduced, normalization translates the AST produced by the parser into a rewritten AST with the same semantics, but with a reduced set of language constructs. Thereby, normalization removes syntactic sugar. The normalized version of the above query has the following form[2]:

```
let $auction := doc(auction.xml)
 return count(
     for $i in ddo(
       for $fs:dot in $auction
       return ddo(
           for $fs:dot in child::site
           return
             ddo(
               for $fs:dot in child::closed_auctions
               return child::closed_auction)))
     where fn:data(ddo(
           for $fs:dot in $i
           return
             ddo(for $fs:dot in child::price
                 return child::text()))) >= fn:data(40)
     return
       ddo(for $fs:dot in $i
           return child::price))
```

You can easily observe that the normalized variant of the query does not contain any path expressions, only axis steps (like `child::site`). Path expressions are rewritten to *for* clauses. Furthermore, the normalization process injects the *ddo* and *fn:data* function ensuring correct duplicate-free intermediate results (in case of *ddo*) and atomic values for comparisons (in case of *fn:data*).

2.2.3 Static Typing

Static typing infers the type of all subexpressions in a normalized query. For example, in the query above, the static type of the integer literal "40" is trivially *integer*. The surrounding *fn:data* function also delivers type *integer*, which is then used in the comparison. The comparison, in turn, is of type *boolean*, and so on.

Static typing can help to find type errors in XML queries at an early stage. However, as explained above, some type errors only occur during processing, when the accessed data does not deliver the expected type. Besides error detection, the type information generated by the static typing stage can also be used in query simplification, as we will see in the next stage.

2.2.4 Simplification

Even in our small example, you can observe that the normalization process is defined in a rather defensive manner, i. e., it injects certain functions blindly, even when they are not necessarily required. For example, the injected *fn:data* function around the integer literal "40" does not have an effect and can be safely omitted. A further example is the *ddo* function that is always injected, even when the interme-

[2] Note, this representation is simplified to facilitate comprehension. Function *ddo* stands for *fn:distinct-doc-order*, and—against the W3C recommendation—the constructs to produce positional information are omitted.

diate result will always be in distinct document order. Besides normalization, users might write XQuery expressions with redundant or unnecessary subexpressions. Simplification aims at removing this kind of redundancy. An equivalent query for the above one might look like the following:

```
let $auction := doc(auction.xml)
 return count(
     for $i in
       for $fs:dot in $auction
       return
           for $fs:dot in child::site
           return
               for $fs:dot in child::closed_auctions
               return child::closed_auction
     where fn:data(
           for $fs:dot in $i
           return
             for $fs:dot in child::price
             return child::text()) >= 40
     return
       for $fs:dot in $i
       return child::price)
```

Essentially, the *ddo* functions are not necessary and the *fn:data* function around the integer literal can be removed[3]. Currently, the XQuery processor can detect simplification opportunities in various situations (see Section 4.3.2). Note, however, that the simplification logic aiming at removing *ddo* functions is not yet integrated (although this topic has already been discussed in the literature [Fernández 05]). Since XQuery is a quite flexible and freely composable language, many more situations than those handled in this work allowing for simplifications might exist. This work does however not dwell further.

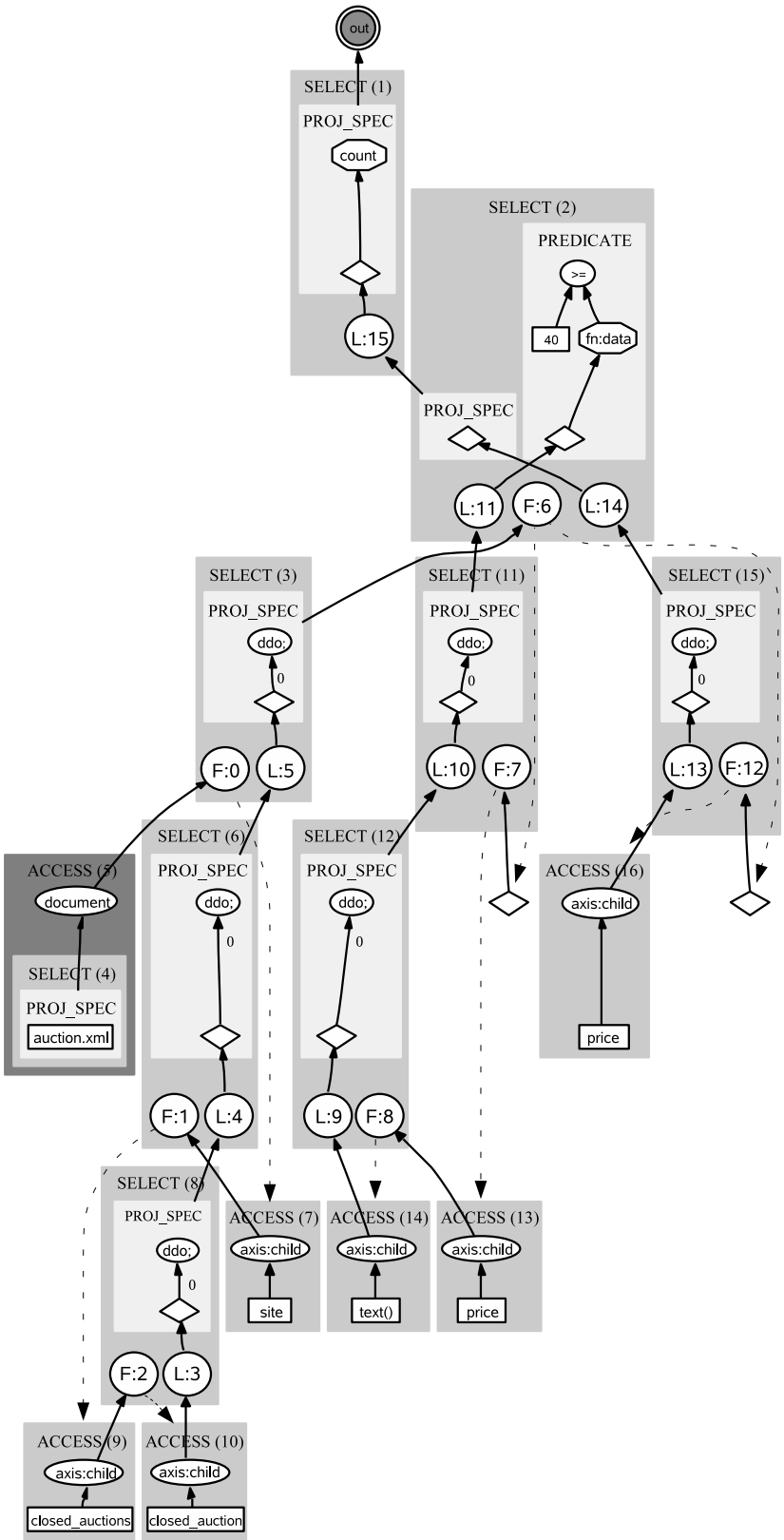
2.2.5 XQGM Transformation

After simplification, the query is transformed into a graph representation that can express the query's semantics more naturally than the AST. The graph representation is called *XML Query Graph Model* (XQGM) and can capture a large fraction of the XQuery language. The initial XQGM instance for our sample query is depicted in Figure 2.4. Note, all logical and physical plans presented in this and the following chapters are generated by a plan visualization tool developed in [Mathis 08].

An XQGM instance is an operator graph or a box-and-arrow diagram. Every box is a logical operator which produces data (most operators also consume data). The produced data flows along the arrows. All operators have a *name* describing the functionality of the operator and a unique *identifier* that follows the name in braces, e. g., "`SELECT(3)`". In the following, we refer to an operator by its name and its ID in braces. To highlight an operator, we use a typewriter font and write the operator name in lower-case letters for better readability, as for example "`select(3)`". The graphical elements inside an operator specify how the operator processes input data and how it computes results. For example, `select(2)` consists of three so-called *tuple variables* (depicted as circles) controlling the input data flow, a *predicate* describing the selection expression, and a *projection specification* defining how the output shall be computed. Tuple variables carry a quantifier (e. g., "F" or "L") and a unique identifier to facilitate their distinction. The quantifier and the identifier are

[3]This is actually possible, because static typing revealed that the argument of the *fn:data* function is already an atomic value and therefore does not need atomization.

Figure 2.4 XMark query Q5 represented in XQGM



separated by a colon. For tuple variables, we use the sans-serif font in the text as well as in the figures, e. g., we write "F:1", "L:15", and so on.

XQGM is only a logical query representation. It cannot be executed directly. However, to illustrate the semantics of the XQGM instance shown, we step through a "virtual" query execution:

- The query processor calls the topmost `select(1)` operator, which, in turn, calls the next `select(2)` operator below to produce some output. `Select(2)` has three tuple variables, one of which carries an "F" specifying *for*-quantification semantics, and two of which carry an "L" for *let*-quantification semantics. Basically, the tuple variables receive the output generated by their subgraphs below and define how this output is assembled into a stream of tuples. How this actually works will be sketched below. For now, we just proceed with the subexpression under tuple variable F:6. `Select(3)` is called and, in turn, `access(5)`.

- So far, we only considered control flow. Every operator calls its dependent suboperators and awaits data for further processing. Now `access(5)` is the first operator that actually produces data. In our example, `access(5)` is a document access operator which delivers the *virtual root node* [Fernández 04] of the "auction.xml" document. This node is passed to the `select(3)` operator which binds it onto tuple variable F:0 and calls `select(6)` to produce a result for tuple variable L:5.

- `Select(6)` in turn calls `access(7)`, which is a navigational access operator. This type of access operator needs a *context node* as input from which the navigation starts. The context node is delivered by a *correlated input edge*, depicted as a dotted arrow. Tuple variable F:0 provides this input by passing the currently bound virtual root node to `access(7)`. The result of the navigation on the child axis and the subsequent name test is a single *site* node. This node is passed to `select(6)` which binds it on tuple variable F:1 and calls `select(8)` to produce results for tuple variable L:4.

- `Select(8)` calls `access(9)` which delivers the *closed_auctions* element (exactly one in every XMark document) using the current node at tuple variable F:1 as correlated input. The *closed_auctions* element serves as correlated input for `access(10)` which returns all *closed_auction* elements below. These elements are passed to tuple variable L:3 which collects them all, puts them into a sequence, and binds this sequence as the current value (which is actually the semantics of the *let* quantification).

- The sequence is then passed to the projection specification, which (unnecessarily) applies the *ddo* function (note, our simplification process cannot yet remove these unnecessary calls). A tuple variable may either be referenced via a correlated edge (dotted arrow) or by a so-called *tuple variable reference* depicted as a rhomb. The *ddo* function is also applied in `select(6)` and `select(3)` passing the sequence of *closed_auction* elements to tuple variable F:6.

- So far, every *for*-quantified tuple variable received only a single node as input. For single nodes, the semantics of *for* and *let* are the same. This time, however, tuple variable F:6 receives a sequence of possibly more than one node. While *let* passes these nodes as a whole as described above, *for* iterates over the sequence items, just like the corresponding constructs in the XQuery language. You can further notice that the subtrees below tuple vari-

ables L:11 and L:14 depend on the current node at tuple variable F:6, because these subgraphs have a correlated input edge starting at F:6. This means that for every node at F:6, the dependent subtrees are evaluated and their result sequences are bound to the corresponding tuple variables. In the following, we will call L:11 and L:14 *dependent tuple variables*, whereas F:6 is called *independent*. Obviously, the subtrees below independent tuple variables have to be evaluated first, because they provide the input for the subtrees below dependent tuple variables. Essentially, the subtree below F:6 evaluated `doc("auction.xml")/site/open_auctions/open_auction`. For every *open_auction*, the expression below L:11 evaluates the relative path `price/text()` and L:14 the relative path `price`.

- Inside `select(3)`, the predicate is evaluated for every *open_auction* element. If the predicate evaluates to *true*, the current value at tuple variable L:14 is read by the projection specification and passed as an intermediate result to `select(1)`. In turn, `select(1)` collects all these intermediate sequences in another sequence on which the *count* function is evaluated to obtain the final result.

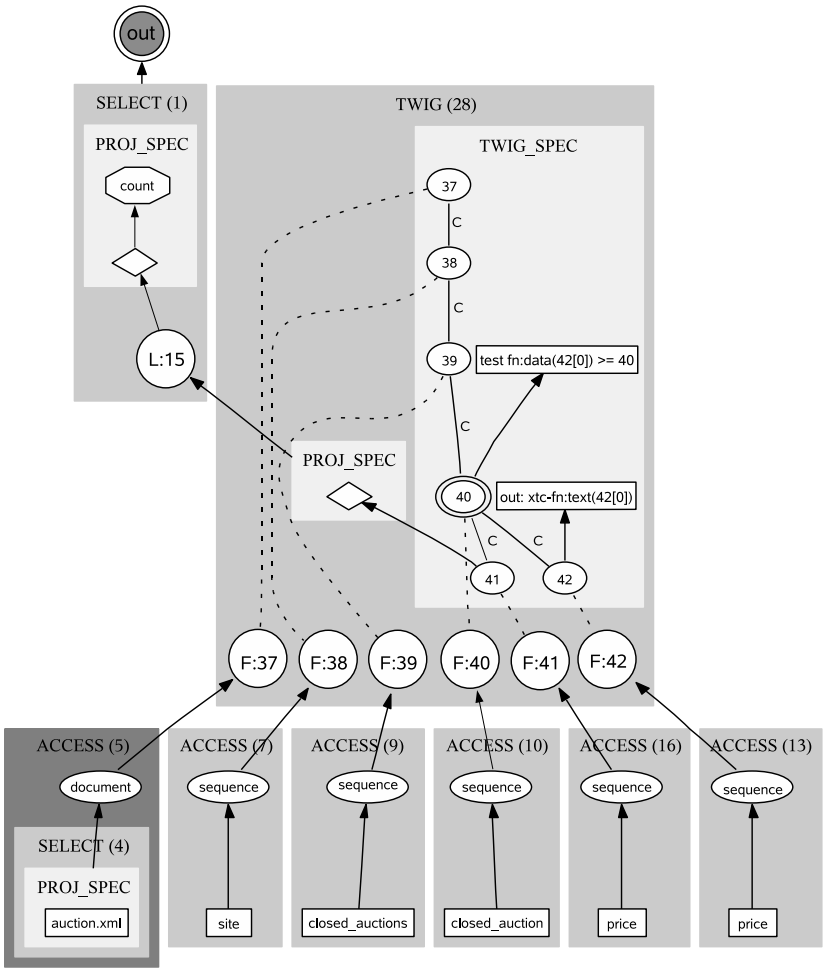
The reader familiar with the *dynamic evaluation* phase specified in the Formal Semantics has noticed that the evaluation model defined there and the XQGM is essentially the same, i. e., an initial XQGM instance acts as specified in the Formal Semantics. This is meaningful, because it ensures correctness. In a way, you can regard XQGM as a graphical representation for normalized XQuery expressions. Interestingly, a large fraction of XQuery can be captured solely by XQGM's select and access operators (as we will see in Chapter 4).

2.2.6 Algebraic Rewriting

The proximity of the initial XQGM translation with the Formal Semantics ensures correctness. However, evaluating the query as depicted in Figure 2.4 and as described in the previous section often results in a bad performance, as we will see in Chapter 9. The problem originates from the large number of nested subexpressions (easily recognizable by the dotted arrows). Every nested subexpression has to be repeatedly evaluated for every new correlated input, thus imposing a nested-loop-style query evaluation. The situation is similar to relational queries containing nested sub-selections. A standard approach for those types of queries is to rewrite the nested subexpression using a join operator [Mitschang 95]. This is also possible for XML queries in XQGM: a nested subexpression with a navigating access operator can be rewritten into a so-called *structural join* operator. The process for this kind of restructuring is called *unnesting* and will be shown in full detail in Chapter 5. Structural joins are binary operators. However, often they can be grouped together into more complex multiway join operators called *twig joins*. The completely rewritten version of our sample query is depicted in Figure 2.5 and contains such a twig join operator.

At a first glimpse, the rewritten query has not much in common with the original one. However, its semantics is still the same. First you can observe that the query does not contain any nested subexpressions, it has been completely unnested (note, the dotted lines inside `twig(28)` have a different semantics). As a result, all access operators are not based on navigation anymore, but access *all* nodes that match a certain node test (e. g., all *price* nodes). The twig operator is a multiway join operator semantically capturing the structural pattern defined in the query. The nodes in

Figure 2.5 Rewritten XMark query Q5

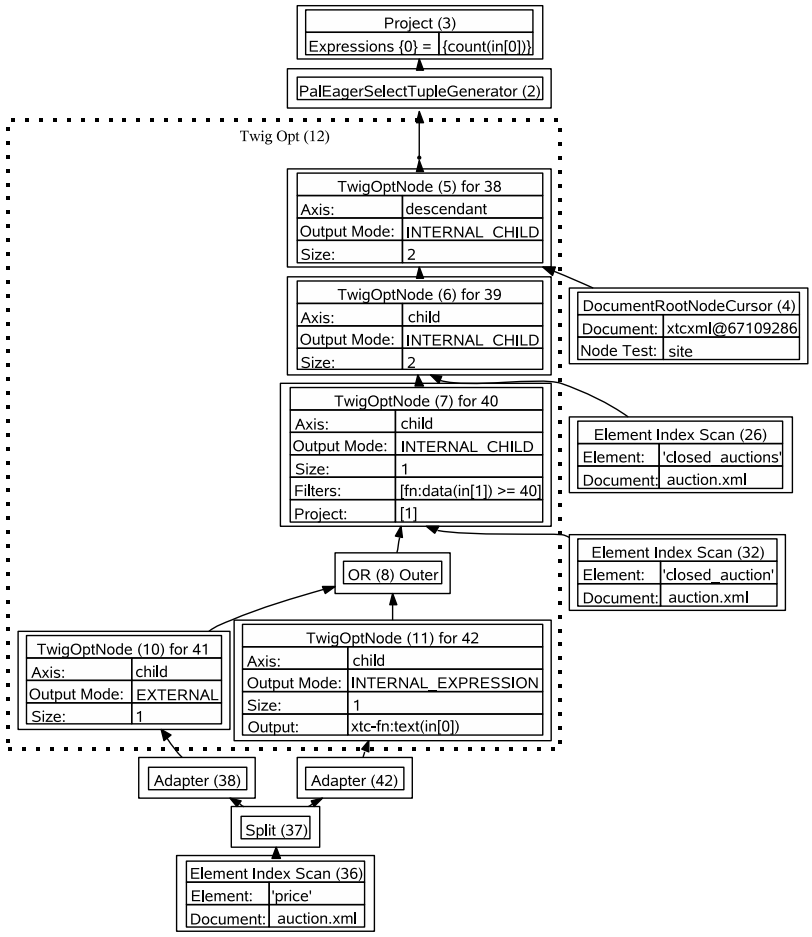


the structural pattern are connected to the corresponding input tuple variables by dotted lines, whereas the structure of the twig itself is represented by solid lines. The `C` stands for a child relationship (`D` for descendant, and `@` for attribute). A twig operator is capable of finding all structural matches in the given input sequences. Furthermore, it can directly evaluate

- *predicates*, e. g., the `test` expression at node 40;
- *projection*, e. g., only *price* nodes delivered by tuple variable `F:41` shall be returned;
- *output expressions*, e. g., the `fn:xtc-text` function is applied to nodes delivered for input tuple variable `F:42`;
- *nesting*, e. g., the double circle at `twig node (40)` specifies that the input of its children shall be nested; and
- some more functionality, as we will see later on.

Note, in XQGM, these capabilities of the twig operator are defined at a logical level only. This does not necessarily mean that a single algorithm can directly evaluate them. Rather, it would be possible to implement this algorithm by structural joins, group-by operators, path indexes, etc. However, as we will see in Chapter 8, a twig

Figure 2.6 QEP for XMark query Q5



join algorithm that can directly evaluate output expressions, predicates, projections, and nestings in a single sweep is being developed in this thesis.

The `select(1)` operator is the same as before. It collects the *price* sequences generated by the twig operator, adds them to a sequence and applies the *count* function.

2.2.7 Plan Generation

Given the result of the rewriting stage, the query processor now has to assemble a query execution plan (QEP), i. e., it has to map the logical operators onto physical ones. Section 2.1.3 already sketched how this process is implemented. Here, we only show the resulting generated plan (see Figure 2.6).

You can observe that the resulting QEP is quite similar to the original XQGM instance. Also here, every operator has a unique ID. However, the QEP contains more low-level details (represented using the tableau technique, similar to [Mitschang 95]). In this overview, only some highlights of the representation shall be discussed: In the QEP, almost all access operators are implemented by scans over

the so-called element index. Only one input operator (4) is different. It directly retrieves the root element from the document and applies the *site* name test. You can also observe that, because *price* elements are required twice, the result of the scan operator is shared using a *split* operator. The twig operator itself is represented inside the area surrounded by a dotted line. We will not delve into the semantics of every twig node, but only state that it is possible to infer for which logical twig node the physical one was generated (for example, in "TwigOptNode (5) for 38", the physical twig node 5 was generated for the logical twig node 38 in Figure 2.5). In the physical projection operator (3), the *count* function is finally applied. The complete physical algebra of the XQuery processor implemented in XTC has more than 60 operators.

2.2.8 Execution and Materialization

Finally, the generated QEP can be evaluated on the database. Therefore, every operator implements an iterator-like protocol, which defines the following three methods: *open*, *next*, and *close*. In the opening phase, every operator has the chance to initialize its internal data structure, while the closing phase is used for clean-up. The actual result is computed by the *next* method, which is, as *open* and *close*, recursively carried out over the operator tree. Physically, an operator returning an XML element does not return the element with its complete subtree (materialized), but rather a node reference to the element (containing a DeweyID). For example, the twig operator in Figure 2.6 would deliver the following sequence as intermediate result to project (13): `<[price, 1.3.3], [price, 1.7.3], [price, 1.9.5], ...>`. In our example, the *count* function simply returns the size of this intermediate result. If we assume that this function would be absent, the result would have to be converted into an output string or into DOM nodes (depending on the requirements of the application program). Therefore, materialization resolves node references by replacing them with the subtrees from the document. Resolving references at the end of the query process ensures that only those subtrees are materialized that actually belong to the query result.

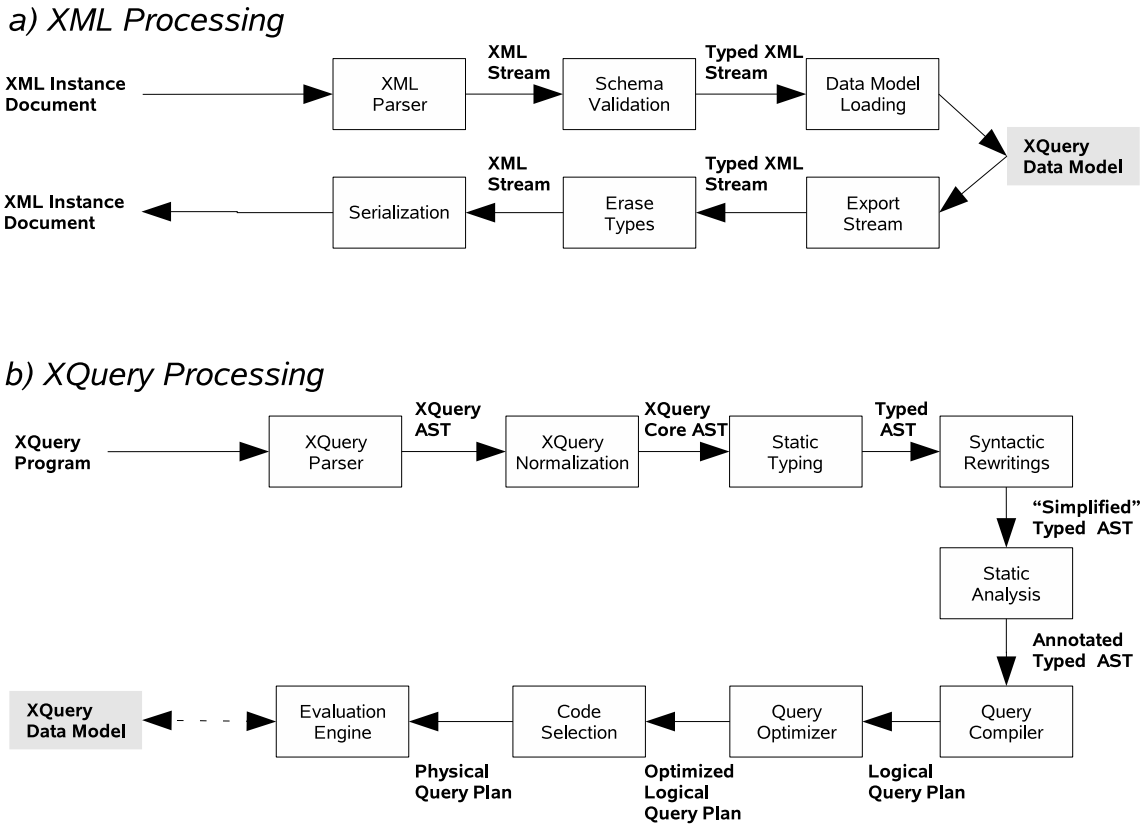
2.3 Related Work

Already at this point, we want to start with the discussion of related work. Our intention is to give an overview over other existing systems, such as Galax, IBM DB2 pureXML, Timber, Natix, and MonetDB/XQuery. The focus of this thesis lies on query processing in native XDBMS. However, XML query processing techniques have also been developed in non-native systems and even "outside" DBMSs. One example is the Galax system, which shall be discussed first.

2.3.1 Galax

Galax [Fernández 03] is an open source implementation of the XQuery language written in the functional OCaml language [Leroy 08]. Galax was developed as a reference implementation of XQuery (in fact, many authors whose names appear on XQuery-related W3C documents contributed to Galax). The system is close to the Formal Semantics and its conformance with the standard is very high. In summary,

Figure 2.7 The Galax document processing and query evaluation pipeline [Siméon 04]

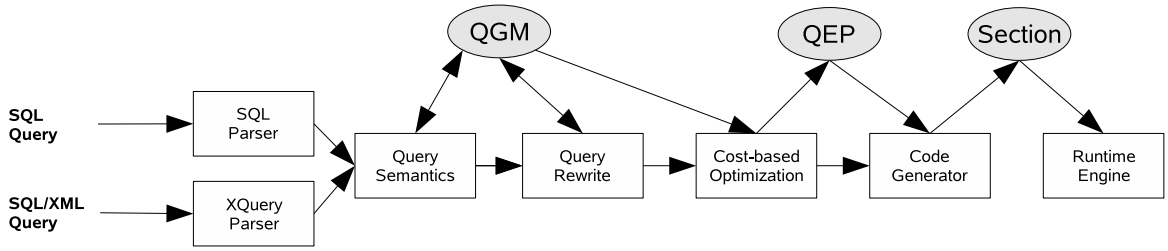


Galax supports all features of the XQuery 1.0 Recommendation [Boag 04], XML Schema support and validation, static type checking, as well as the new XQuery Update Facility [Chamberlin 07a] and the XQuery scripting extension (XQueryP [Chamberlin 06]). Figure 2.7 gives an overview over the document processing and the query processing pipeline of the system according to [Siméon 04].

To query a document, the document has to be fetched into main memory first. The upper branch in Figure 2.7a describes the necessary actions: The document is parsed resulting in an XML stream. This stream can be validated against a possibly existing XML schema resulting in a stream with type-annotated XML nodes. The stream is then loaded into a main-memory representation ("XQuery Data Model"). Queries are evaluated on this representation. The lower branch in Figure 2.7a describes how a result document (represented in the XQuery Data Model) can be serialized to an XML instance again.

Galax query processing is shown in Figure 2.7b. The processing stages are quite similar to our approach: A query is first parsed into an abstract syntax tree (AST), normalized, typed, and simplified. Then, the query is compiled into a logical query plan. Here, however, Galax does not rely on a query graph model, but on an algebraic representation. This representation is then optimized and a physical plan is generated. The plan can be executed against the main-memory representation of the queried document. A problem resulting from the sketched document processing

Figure 2.8 The DB2 hybrid XML/XQuery processor [Balmin 06]



approach is the vast main-memory usage. Representing a document of size 30 MB easily reqires 250 MB in main memory. As a solution for this problem, [Marian 03] suggest to analyze the query and the document first, to prune unnecessary parts before evaluation.

Being a native XML database system, XTC naturally does not suffer from these problems. As we will see, in XTC, every node in an XML document can be fetched from external storage by its ID. Therefore, only the necessary parts of a documents are touched (without a prior projection stage). Because in XTC the first four stages, i. e., parsing, normalization, static typing, and simplification are quite similar to Galax, we do not elaborate on these stages (much). The differences between Galax and XTC begin at the internal query representation and the rewriting stage.

2.3.2 IBM DB2 Pure XML

IBM DB2 Pure XML [IBM 09] is a commercial database system that follows the side-by-side architecture [Halverson 04] to manage relational data and XML data in a single system. The product was influenced by various predecessors, i. e., DB2 Viper [Päßler 06] and SystemRX [Beyer 05]. Essentially, XML documents are stored in a special native store. Documents can appear as a "value" of a tuple field in a relational table. Consequently, a single query can be posed simultaneously against the relational data and against the XML data. The language to express such queries is an SQL extension called SQL/XML [ISO/IEC 03]. Internally, SQL/XML queries are handled by a hybrid query processor, which is able to optimize relational queries and XML together.

Figure 2.8 gives an overview over the hybrid query processor [Balmin 06]. Both languages—SQL, XQuery, and mixes thereof—are translated into an internal representation similar to the one developed in this thesis, namely the *query graph model* (QGM). As we have seen, the QGM originally originates from the Starburst System [Haas 89, Pirahesh 92] and is also utilized by DB2. Clearly, QGM (and the complete relational evaluation process) existed before XML and XQuery came up. However, because QGM was designed for extensibility, the integration of the XQuery language into this model was possible.

QGM is the basis for the algebraic query rewrite phase and for the generation of a so-called *query evaluation plan* (QEP). Algebraic rewriting restructures the query by removing unnecessary operators and operator merging. The QEP is then assembled by the optimizer, i. e., the optimizer tries to find the cheapest plan w. r. t. to some

cost model. From a QEP, the code generator compiles a so-called *section*. Sections are stored in the database and can be executed to retrieve the query result.

Obviously, there is a certain proximity between the DB2 approach and this work and you might wonder, where the differences are. In particular, the integration of XQuery into the query graph model is also a main goal of this thesis. The general problem with the commercial DB2 system is that research papers are fairly high-level and not much information on "the real details" officially reach the research community. Although some overlap on the general approach to query processing in DB2 and XTC exists, our work is a first in-detail discussion on how Starburst's query graph model can be extended to support XQuery.

2.3.3 Timber

Timber [Jagadish 02a] was one of the first scientific projects on *native* XML data management. The authors argue that XML trees should be processed as trees. Therefore, Timber consequently follows the "tree approach": The core contribution of the project is the TAX algebra [Jagadish 02b]—a tree-manipulating algebra for XML—and the holistic twig join [Bruno 02]—a bulk tree-pattern matching algorithm. Let us take a look at how queries are processed in Timber. As always, the parser is the first component in the query processing pipeline. It analyzes an XQuery expression and generates an equivalent expression in the logical TAX algebra. This expression is then optimized and mapped onto a physical algebra expression, which can be executed with the help of the data manager and the index manager.

In Timber, the data manager is responsible for XML document storage and retrieval. To avoid re-coding of the storage layer, Timber relies on the Shore [Carey 94] storage manager. To store a document, every XML node is mapped to a record, which is, in turn, passed to Shore for storage. The index manager also exploits Shore for index storage. It provides access to quite simple indexes: a value index for attribute values, an index on numeric element content, a term-based index for text, and an index providing posting lists of all elements with a certain name. Path indexes are not available [Jagadish 02a].

A specialty of the Timber approach is the tree-based TAX algebra. Every operator in this algebra can receive a set of trees as input and can generate a set of trees as output. In TAX, operators are parameterized by so-called pattern trees. These pattern trees can be matched against XML documents/fragments and, thereby, influence the result delivered by an operator. The tree-based algebra approach stands in contrast to other systems, like Natix, MonetDB (see below), or XTC, which represent (intermediate) results as streams of (nested) tuples. With Timber, XTC has some communalities at the physical level. As we will see, XTC extends the tree-matching operator (i. e., the holistic twig join) idea.

2.3.4 Natix

Natix [Fiebig 02] is also an early scientific project with a focus on *native* XML data management (started in 1998). In Natix, XML queries are compiled into the so-called Natix algebra [May 04], which is an extended relational algebra. As a consequence, the algebra operates on tuples. However, in contrast to relational query processing, tuples can have complex values, i. e., they can be nested. Similar to

TAX in the Timber project, the Natix algebra operates at a logical level. Therefore, it serves as a basis for algebraic query restructuring. For query evaluation, Natix maps algebra expressions to a physical algebra containing evaluation algorithms. A specialty in Natix is that these algorithms can be parameterized by programs. For example, the physical *selection* operator can be parameterized by a program that computes the selection predicate. In Natix, these programs are executed on the so-called *Natix virtual machine* (NVM). The NVM has some internal state kept in a set of registers and a fixed set of built-in commands that can modify register values. Besides the ability to evaluate selection predicates, the instruction set also provides for access to XML documents, navigational operators, further XML processing functionality, etc. For storage, Natix fragments a document into page-sized subtrees. As in traditional systems, these pages are handled in a system buffer.

The research around query processing in Natix mainly focuses on algebraic XQuery treatment [May 04, Brantner 05, May 06b]. Physical operators and index structures (with the exception of some very special operators to treat existential quantification [Brantner 06a] and group by operations [May 05]) are not in the center of the research. XTC has some touching points with Natix regarding the internal algebra. XTC's algebra is also tuple-based, providing for nested tuples. In [Mathis 07b, Mathis 07a], we have extended the Natix algebra by a structural join operator and presented how XPath queries can be unnested utilizing this new operator. This technique will also be applied in Chapter 5, however, then in the context of XQGM and not in the Natix algebra. A small shortcoming of Natix is that, although always XQuery is the focus of Natix research papers, the system only provides an XPath front-end [Moerkotte 09].

2.3.5 MonetDB/XQuery

Natix, Timber, (and, as we will see, XTC) are all *native* XML database systems.[4] Their storage and query processing system is tailor-made for tree-based XML data. MonetDB/XQuery [Boncz 06a] goes a different way. Essentially, MonetDB/XQuery is—at its core—a *relational* database system with an XML front-end called Pathfinder [Boncz 05b]. Therefore, it follows the XOR ("XML over relational") approach introduced by [Halverson 04]. In Pathfinder, XML documents are decomposed and stored in relational tables. A query is parsed and translated into relational algebra. In MonetDB, this algebra is called *MonetDB interpreter language* [Boncz 99] (or MIL for short). However, Pathfiner does not depend on MonetDB, any other relational system can also serve as a back end [Grust 07]. Nevertheless, MIL incorporates the *staircase join* operator—a special operator for XML path matching [Grust 03b], which is similar to a structural join operator and which can speed up this frequent XML query processing operation. Of course, in a standard relational algebra, this operator is not available.

Mapping XML documents to relational tables and XML queries to relational algebra is an appealing technique, because large parts of existing infrastructure can be reused. However, the resulting system is not tailored to XML. The first XML-enabled systems produced by the commercial database vendors (e. g., IBM, Oracle [Banerjee 00], and Microsoft [Rys 05]) have also followed the XOR approach. However, they all left this path and nowadays provide a native XML storage engine.

[4] For one part of DB2 pureXML, this statement is also true.

2.3.6 Other Systems

The five systems introduced so far are not the end of the rope. Many more scientific and commercial XML query processors and XML database systems exist, for example, Tamino [Schöning 00], as one of the first commercial native systems, or Saxon [Kay 09], an open-source stand-alone query processor. Further approaches are Niagara [Naughton 01], eXist [Meier 02], LegoDB [Bohannon 02], OrientX [Mang 03], and Sedna [Grinev 06]. XML in general, XML database systems, and XML query processing have been hot topics in the scientific community over the last decade. Furthermore, database vendors have integrated XML support into their commercial systems or even built completely new ones to address the customer need for DBMS-supported XML management. As reference [XQuery 09] impressively illustrates, this development led to a plethora of systems, theories, and approaches to the topic. The related work section of a dissertation like this can never be "complete" w. r. t. to all these concepts. We chose to introduce the above five approaches in greater detail, because they are well-known in the community and there are some touching points between these systems and the XTC query processor. Because we all are database researchers with the same background, it is also not astonishing that the efforts and general solutions all point in the same direction. A nice example is the similarity among the above introduced approaches. You might say that XTC is yet another solution. However, as you will discover numerous differences during the related work sections embedded into the following chapters, making this work worth reading.

2.4 Summary

This chapter sketched the query evaluation process of the query engine developed in this thesis, with its nine stages of 1) syntactic analysis, 2) normalization, 3) static typing, 4) simplification, 5) XQGM transformation, 6) algebraic rewriting, 7) plan generation, 8) execution, and 9) materialization. During processing, a query can be represented in four different ways: as a string (external representation), as an abstract syntax tree (AST), in XQGM (logical representation), and as a PAL tree (physical representation). From a scientific point of view, some of these stages and representations are more interesting than others, which is also reflected in this work. Therefore, the explanations mainly focus on XQGM, algebraic rewriting, the physical algebra, and the various transformations from one representation into the other (i. e., XQGM transformation and plan generation). The related work section gave an overview over Galax, DB2 pureXML, Timber, Natix, and MonetDB/XQuery, which we consider close to the XTC approach. After outlining important preliminaries of this work in the next chapter, the main contribution starts in Chapter 4 introducing the XML query graph model.

Chapter 3 The XML Transaction Coordinator

> In the beginning there was nothing, and it exploded.
>
> Terry Pratchett

The query processing concepts developed here are implemented in a *native* XML database system: the XML Transaction Coordinator (XTC). Generally, query processing deals with both, system-*in*dependent problems (logical abstraction) and with system-dependent problems (physical abstraction). This is also the case for the present work. To address the system-dependent part of the query processor, selected internals of the XML Transaction Coordinator shall be introduced.

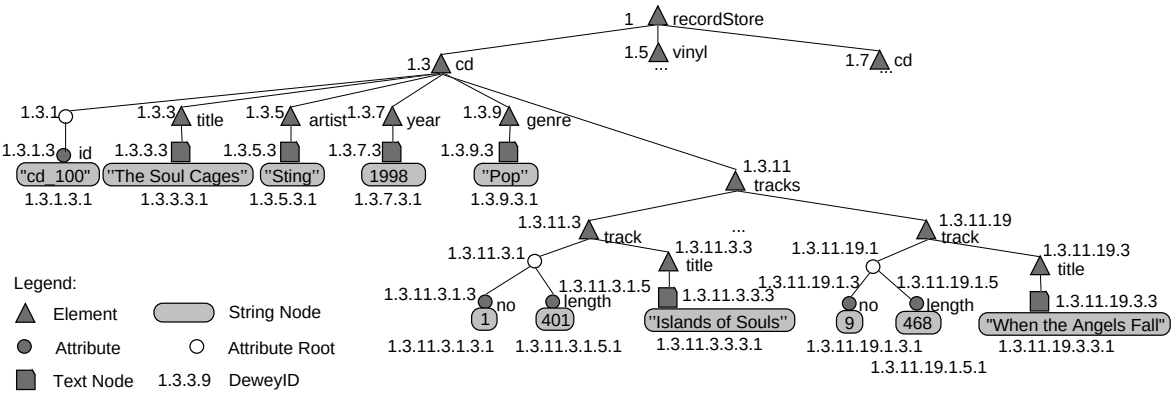
XTC is a native XDBMS. Here, the term "native" denotes that the system is specially optimized for XML data management, from storage layout over synchronization concepts to external interfaces. In summary, XTC allows to store, process, and retrieve XML data in a multi-user environment supporting collaborative XML sharing via ACID transactions and fine-grained node locking. At the external client interface, the database server provides for: storage, reconstruction, and deletion of single documents and whole document collections managed in a virtual directory structure; primitives for ACID transactions, such as "begin transaction", "commit", and "abort"; primitives to influence multi-user synchronization (for example, to control the isolation level); DOM and SAX interfaces; access to the XML repository via HTTP, FTP, and a driver package; and, finally, system monitoring services.

The following introduction is split into a conceptual part, where the internal XML data model and the locking protocol are introduced, and into an architectural part, where implementation-specific details are discussed.

3.1 The taDOM Data Model

The taDOM data model [Haustein 03] serves as a logical internal representation for XML documents in XTC. Basically, taDOM is an extension of the DOM data model [DOM 04] and was developed to support synchronization of concurrent DOM-based multi-user access over XML documents, therefore the name *taDOM* for *transactional DOM*. Synchronization is implemented by a pessimistic hierarchical locking protocol which is introduced in the next section.

Figure 3.1 Snippet from a sample *recordStore.xml* document as taDOM tree (the original XML document can be found in the Appendix)



3.1.1 The taDOM Tree

Figure 3.1 depicts a snippet from a sample *recordStore.xml* document modeled in taDOM. The data model is a tree consisting of nodes of the following five types: *element*, *attribute*, and *text* (as in the DOM), and additionally *string* and *attribute root*. Further node types (such as *processing instructions*, *comments*, etc.) are not supported in the data model and, therefore, XTC cannot store them[1].

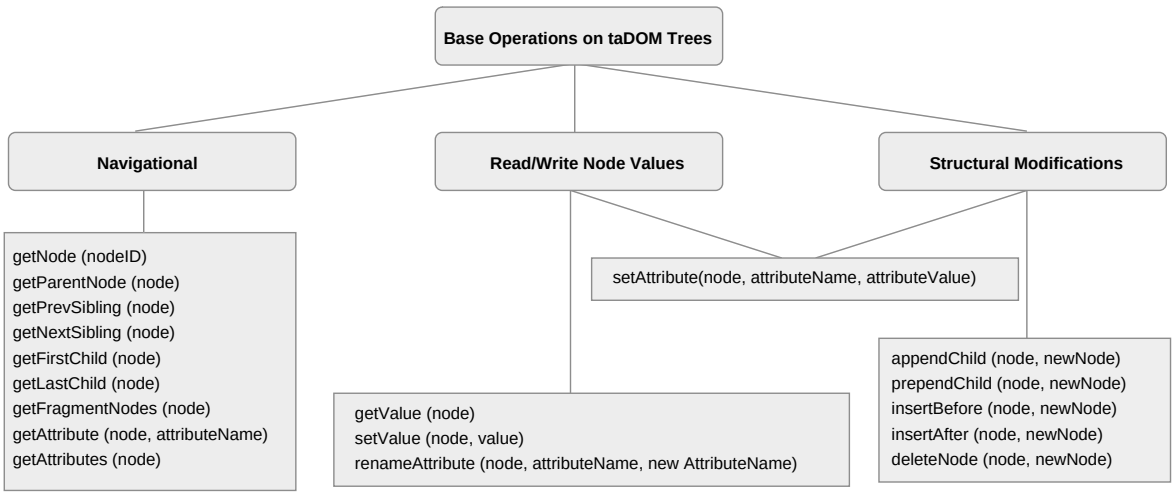
As indicated in the illustration, attributes are not directly attached to their containing element nodes, but dangle below an attribute root. Similarly, text nodes do not contain their content directly, but have a string node below that wraps the content. These structural features facilitate transaction parallelism in the hierarchical lock protocol: If a transaction only needs access to particular attributes, they can be protected by individual node locks, thereby allowing concurrent access to the remaining (unlocked) attributes. However, when the transaction needs access to *all* attributes, a single lock on the attribute root is sufficient. The optimization for text nodes addresses navigations over elements with mixed content. Suppose a transaction navigates over the children of such an element to find a particular child element. The content of a passed text node (stored in string node) is not relevant for this transaction, therefore it is sufficient to only lock the text node. Another transaction can then even modify the string node below such a locked text node.

3.1.2 Operations on the taDOM Tree

As an internal data structure, the taDOM tree supports various kinds of navigation and modification primitives that resemble the methods of the external DOM interface and that, in fact, serve for their implementation. However, in contrast to classical DOM, XTC protects taDOM access primitives by ACID transactions. Figure 3.2 depicts possible taDOM operations. All operations, except *getNode*, require at least a (context) node as parameter. Navigational primitives allow to 1) jump into the tree by providing a node ID (*getNode*, see below), 2) to traverse the taDOM tree,

[1] However, these node types are straightforward extensions to taDOM and, because XTC is a research project, this limitation is not too critical.

Figure 3.2 Operations supported by taDOM (according to [Haustein 06a])



or 3) to access all descendants of a particular node (*getFragmentNodes*). The second group provides primitives to alter node values and to rename attributes. A node value is defined as the element name for elements and the content for string, attribute, and text nodes. Methods on the node value are undefined for the attribute root. The last group of operators allows to insert new nodes and attributes to the document.

Section 3.3 presents how documents are stored on external memory. In fact, only the nodes of the taDOM tree are stored, and all edges, i. e., *parent*, *first-child*, *last-child*, *next-sibling*, and *previous-sibling* are virtual. Crucial to this storage mapping is a suitable node identification mechanism that encodes structural relationships among nodes and that allows to address single nodes in the document directly. XTC implements a variant of the *DeweyID* node labeling scheme [Haustein 05b, Härder 05b] for that purpose. This variant is an adaption of the OrdPath numbering scheme introduced by [O'Neil 04].

3.1.3 DeweyIDs for Node Identification

A *DeweyID* is a sequence of dot-separated integers also called *divisions*. Initially, DeweyIDs are assigned during document storage and only odd division values are used as divisions. Even division values are required as a kind of overflow mechanism when new nodes are inserted (see discussion below). The DeweyID of the root of the document is set to "1". All other DeweyIDs are computed as follows: The prefix of the DeweyID of a particular node n is the DeweyID of its parent node. As a suffix, n attaches one further division value depending on the position of n among the children under the n's parent: The first child attaches 3 as a suffix division, the next child $3 + 2 = 5$, the next one $5 + 2 = 7$, and so on, i. e., the suffix division of a node is obtained by adding 2 to the suffix division of its preceding sibling. For example, the DeweyID of the *year* element is composed of the DeweyID of the *record* element (1.3.3) as the prefix and the suffix division of the preceding *artist* element (5) plus 2, i. e., 1.3.3.7. Attributes, attribute root nodes, and string nodes get a special

treatment. For the attribute root node and for string nodes, always "1" is attached as last division, because these node types cannot have any siblings. DeweyIDs have many salient features that not only facilitate persistent XML storage but that are also central to XML query processing, as we will see throughout this work:

1. A DeweyID contains the DeweyID of its parent node. Transitively, given a DeweyID, all its ancestor DeweyIDs can be computed without access to the document. For example, the ancestors of 1.3.3.9.3.1 are 1.3.3.9.3, 1.3.3.9, 1.3.3, 1.3, and 1.

2. The number of divisions in a DeweyID is equal to its level in the tree plus one (by definition, the level of the root node is 0).

3. Given two DeweyIDs, the structural relationships of their corresponding nodes can be calculated without access to the document. In particular, all XPath axis relationships [Berglund 04] can be inferred. For example, DeweyIDs 1.3.3.3 and 1.3.3.7 are siblings, because they have the same parent (1.3.3). Furthermore, given two DeweyIDs, their least common ancestor (LCA) can be computed. The LCA is simply the DeweyID resulting from the longest common prefix of the two nodes.

4. Sorting nodes by their DeweyID in lexicographical order results in the document order.

5. DeweyIDs support document modifications without violating the 4 points above and without reassigning DeweyIDs to any other nodes. This point requires a little bit more explanation: Deletions and updates to node names or content do not affect the above features. Insertions of new nodes require the creation of new DeweyIDs. Depending of the position, where the new node has to be inserted, this process might become quite complicated (but not inefficient) [Haustein 05b, Härder 05b]. Therefore, we only sketch the simple cases. Attributes are not ordered. Therefore, a new attribute is simply appended to the appropriate attribute root and its DeweyID is computed as sketched above. Appending a text/string node to an empty element is also trivial. Inserting a node a between two adjacent existing nodes h and l results in a DeweyID computed as follows: The prefix of the new DeweyID is the prefix of h's parent. The suffix division is h's suffix division incremented by 1, thus an even division value is created. Because in this state, between h and a there is no free suffix division left for further insertions, the even division value is only used as an overflow mechanism and the actual new DeweyID is obtained by adding 3 as a *further* suffix division. For example, insertion of a node between DeweyIDs 1.3.3.5 and 1.3.3.7 results in the new DeweyID 1.3.3.6.3 (note, 1.3.3.6 would not leave "space" for further insertions). Reconsidering the above features 1 to 4, you recognize that the first two are violated. There is however a simple workaround: just skip odd division values. For example, the parent of 1.3.3.6.3 is 1.3.3 and its level is 3 (simply don't count even division values).

A conclusion you can even draw from the small example in Figure 3.1 is that DeweyIDs tend to get long. However, there is quite a large body of work on how to efficiently encode/compress DeweyIDs for small memory consumption [Haustein 05b, Härder 05b]. Furthermore, as we will see in the following discussion, DeweyIDs lend themselves for prefix compression. Empirical evaluations in [Härder 05b] have shown that, for a wide range of real-world documents, a DeweyID only requires around 3 to 5 bytes in average on external memory.

In the literature, quite a large number of alternative node identification schemes have been proposed (see [Haustein 06a] for an overview). An often occurring alternative solution to DeweyIDs is the so-called *range-based* node labeling scheme, which shall briefly be introduced. In the range-based scheme, the XML tree is traversed in pre-order. Every time a node is visited, a label consisting of a pair of integers (r, l) is generated. The first component r is the relative position of the node in document order. The second component can be calculated by $r+s+1$, where s is the number of nodes in the subtree of r. Thus, r and s span a range. Given the ranges of two nodes u and v, it is possible to decide the *descendant* (*ancestor*) relationship between them: if the range of v is contained in the range of u, v is a descendant of u. Similarly, the *preceding* and *following* relationships can be decided. However, *parent*, *child*, and the sibling relationships cannot be inferred, because level information is missing. Therefore, the range-based scheme is extended to $(r, l, level)$. Compared to the DeweyID scheme, range-based node identification has two substantial drawbacks: 1) it is not insert-friendly, because even when gaps are left, insertions might provoke relabeling; and 2) it does not reveal the IDs of ancestor nodes (thus, expensive document access operations are required, when the ancestor ID need to be computed).

3.2 The taDOM Lock Protocol

Multi-user access to a taDOM tree is isolated by a family of pessimistic hierarchical lock protocols called *taDOM2*, *taDOM2+*, *taDOM3*, and *taDOM3+* [Haustein 06a]. Each lock protocol is tailored to the operations supported by a particular DOM version. For example, taDOM2 provides only appropriate locking mechanisms to synchronize access via DOM Version 2 operations. Therefore, taDOM protocols are only sufficient to protect multi-user access in the case of navigational DOM operations. Indexed access, i. e., jumps into the document based on some kind of index structure (e. g., as introduced in Chapter 7), cannot be synchronized, due to the phantom problem. To alleviate this situation, [Haustein 05a] introduces the concept of value-based axis locks, which is only sufficient for a certain type of index structure (i. e., element indexes). In some cases, the taDOM2 or taDOM3 lock protocols fail to satisfy certain lock requests without access to the document (i. e., when lock conversion is required). Due to high I/O costs, however, access to the document is prohibitively expensive, as various performance tests revealed [Haustein 06b]. Therefore, the "+" variants of the taDOM protocols introduced new lock types to alleviate this situation. For brevity, only the basic taDOM2 protocol is introduced in the following. Please refer to [Haustein 06a] for a full description of all protocols.

In taDOM2, access to a particular node has to be protected by a *node lock*. To acquire a node lock, a lock request carrying a transaction identifier, the DeweyID of the node to be locked, and a lock mode, is issued to the lock manager component of the XDBMS. The lock manager then decides whether the lock request is in conflict with existing locks of other transactions or not. If the lock conflicts, the requesting transaction is blocked until the conflict is resolved. Otherwise, the lock is granted and the requesting transaction can proceed. Because taDOM is a hierarchical lock protocol, the lock manager does not only lock the requested node, but also has to protect its ancestor path by placing appropriate locks on each ancestor node starting from the root. Because DeweyIDs easily allow to compute the ancestor path, this

operation is possible without document access.

In summary, taDOM2 supports eight lock modes:

- *Intention Read* (IR) signals a read lock (NR, LR, or SR) in the locked node's subtree. Furthermore, an IR lock requires the parent node also to be locked in IR mode.
- *Node Read* (NR) locks the given context node in read mode. Concurrent read access to the node's subtree is allowed. A NR lock requires an IR lock on its ancestor nodes.
- *Level Read* (LR) locks the given context node and all its children. As for NR, ancestors have to be protected by IR.
- *Subtree Read* (SR) protects the context node and all its descendants. Again, IR is required on ancestors.
- *Intention Exclusive* (IX) signals an SX lock on at least one of the node's grand children or in at least one subtree below a grand child, i. e., the IX lock does not protect direct children (this is actually the task of the CX lock). An IX lock requires IX locks on all ancestors.
- *Child Exclusive* (CX) signals at least one SX lock on a child. The distinction into IX and CX is necessary to decide, if an LR lock can be allowed on a node (i. e., LR and IX are compatible, because the SX lock protects grand children and their subtrees, while LR and CX are not compatible, because at least one child is locked in SX).
- *Subtree Update* (SU) signals lock conversion at a later point in time into an SR lock (downgrade) or into an SX lock (upgrade). SU requires IR on all ancestors. These locks are upgraded into IX and CX, when the SU shall be upgraded to SX. In case of a downgrade, the IR locks are sufficient.
- *Subtree Exclusive* (SX) locks the context node and the complete subtree below for modification. An SX lock requires CX on its parent and IX on the remaining ancestors.

Figure 3.3a shows the *lock compatability matrix* of the taDOM2 lock protocol, based on which the lock manager decides whether a lock request (column header) is in conflict with an existing lock from another transaction (row header). For example, if transaction T_1 holds an SR lock on a particular node, for which another transaction T_2 requests an LR lock, this request is granted by the lock manager. However, if T_2 requests an SX lock on the same node, the lock manager blocks transaction T_2 until T_1 releases the SR lock, because subtree read and subtree modification are in conflict.

Generally, the lock manager only keeps exactly one lock per node and transaction. Therefore, if a transaction requests a new lock on a node already locked by the same transaction, the former lock has to be converted to isolate the new node access. Therefore, taDOM2 defines the *lock conversion matrix* depicted in Figure 3.3b. The column header identifies the already existing lock, whereas the row header identifies the new lock request from the same transaction on the same node. A specialty are those combinations leading to locks with a lock mode occurring as subscript, e. g., the combination LR-CX, which results in a CX_{NR} lock. In these cases, the resulting lock distribution actually consists of multiple locks, where the main lock mode is applied to the locked node, and the lock mode written in the subscript is applied to all its children. In the example, a CX lock is registered for the requested

Figure 3.3 a) Compatability matrix and b) conversion matrix of the taDOM2 lock protocol

	–	IR	NR	LR	SR	IX	CX	SU	SX
IR	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	-	-	-
SR	+	+	+	+	+	-	-	-	-
IX	+	+	+	+	-	+	+	-	-
CX	+	+	+	-	-	+	+	-	-
SU	+	+	+	+	+	-	-	-	-
SX	+	-	-	-	-	-	-	-	-

	–	IR	NR	LR	SR	IX	CX	SU	SX
IR	IR	IR	NR	LR	SR	IX	CX	SU	SX
NR	NR	NR	NR	LR	SR	IX	CX	SU	SX
LR	LR	LR	LR	LR	SR	IX_{NR}	CX_{NR}	SU	SX
SR	SR	SR	SR	SR	SR	IX_{SR}	CX_{SR}	SR	SX
IX	IX	IX	IX	IX_{NR}	IX_{SR}	IX	CX	SX	SX
CX	CX	CX	CX	CX_{NR}	CX_{SR}	CX	CX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

node, and a set of NR locks is put on its children. To satisfy this lock request, all the children of the particular node have to be retrieved from the document—a very costly operation. To handle this situation efficiently, the taDOM2+ protocol introduces new lock modes avoiding document access [Haustein 06a].

Node locks protect the direct access to a node, its subtree, and its ancestor path. However, they do not protect sibling relationships and the first/last child property. For example, if a transaction T_1 navigates from a node to its sibling, another transaction T_2 is not allowed to insert another node between the two navigated nodes. Furthermore, if a transaction T_1 accessed the first child of a particular node, another transaction T_2 is not allowed to delete this first child. If these situations would not be synchronized correctly, the actions of transaction T_1 would not be repeatable. To synchronize these access patterns, the taDOM protocol introduces the so-called *edge locks* that protect the edges during navigation. The edge lock protocol is implemented by a classical RUX lock protocol, where the locks are called *edge read* (ER), *edge update* (EU), and *edge exclusive* (EX). The compatability and conversion matrices for this protocol are omitted here for brevity.

In summary, taDOM provides a logical data model and a tailored family of lock protocols that allow to synchronize multi-user DOM access to XML documents. To keep this overview short, only the basic principles of the taDOM lock protocols were introduced here. For further concepts like, for example, *consistency levels*, *lock depth*, and the *completeness* and *correctness* of the taDOM protocols, please refer to [Haustein 06a].

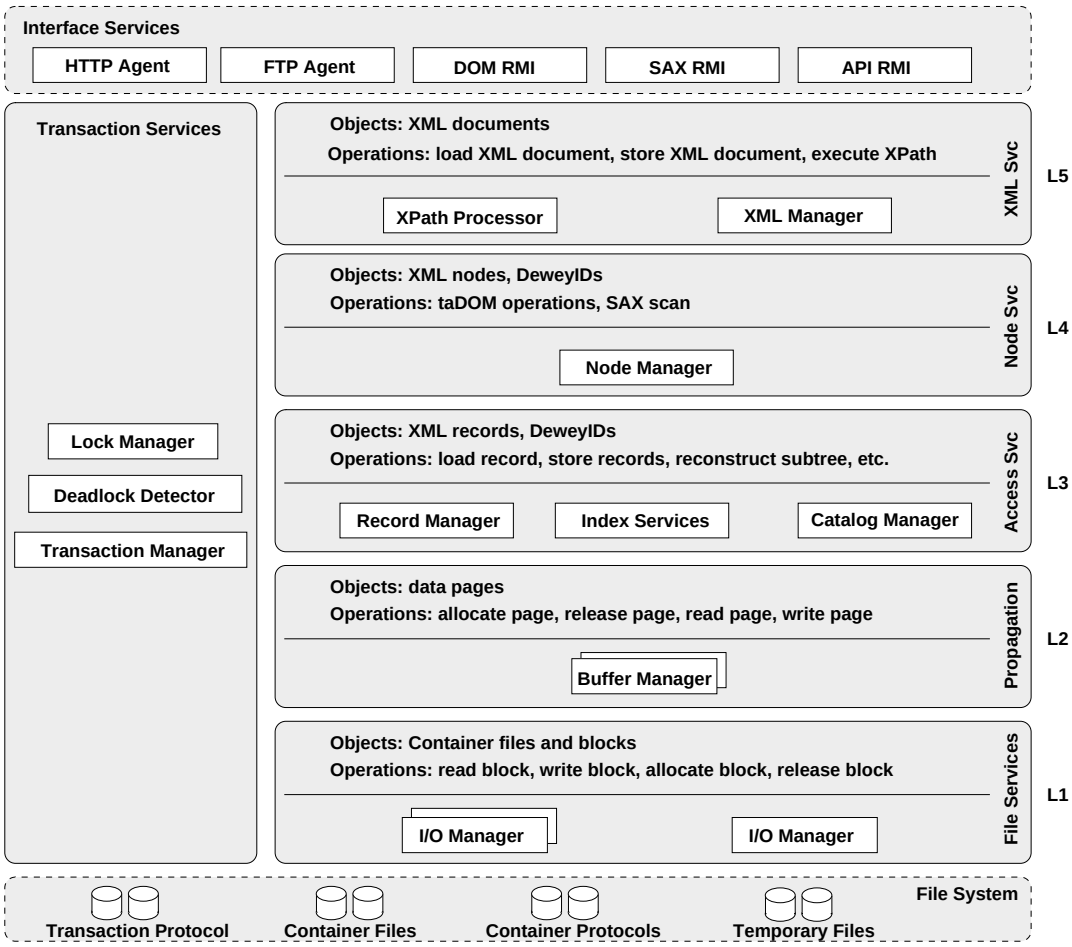
3.3 XTC's Architecture

Internally, the architecture of XTC follows the classical five-layer approach introduced by [Härder 83] (see Figure 3.4). In the following discussion, we will strive through the system in a bottom-up fashion, thereby sketching the major concepts at each of the five layers (L1 to L5). In particular, we will introduce how XML documents as taDOM trees are stored on external memory.

3.3.1 File Services and Propagation

The two bottom-most layers L1 and L2 are standard DBMS components. They control the buffered flow of fixed-size blocks from external memory to main memory. All data, (e. g., documents, metadata, etc.) are stored in a set of so-called *container*

Figure 3.4 Architecture of the XML Transaction Coordinator (according to [Haustein 06a])



files, which are plain random access files managed by the file system. A container consists of a sequence of sequentially arranged fixed size *blocks*. The first block in a container is called *index block*. It stores the two parameters *size* and *extend*. The size parameter denotes the block size which is assigned for each container separately when the database system is installed. Hence, XTC can support multiple containers with different block sizes. For access, *size* allows to identify the byte position of block borders in the container. The second parameter (*extend*) denotes the number of new blocks to be appended, when the container is full and needs to be extended. The remaining blocks in a container store the actual data. Inside each block, the first four bytes store the so-called *block number*, which uniquely identifies a block in the XTC system. A block number consists of a one byte *container identifier*, and a three byte *block identifier*, which is simply the sequential block number relative in the container. The remaining portion of a block is playload data. All in all, XTC supports up to 255 containers and (assuming 32 KB block size) can address a maximum of 512 GB per container. At the file-services layer (L1), a designated *I/O manager* controls access to container files and is responsible to transfer data stored in its container file into main memory. Basic operations are *readBlock*, *writeBlock*, *allocateBlock*, *releaseBlock*, and so on. To write a block, the I/O manager first stores the current block under the writing position as a *before-image* in the specially reserved container block

(blockNo 1). Then, the actual data block is written. The before image ensures block consistency, when the server crashes during a write operation.

Propagation layer (L2) implements the system buffers. Each *buffer manager* in this layer has a single designated I/O manager to access the blocks from a specific container. A buffer manager controls a fixed-size array of so-called *buffer frames*, each of which can hold a *page*. The buffer frame size is equal to the block size and, therefore, each buffer frame can provide exactly one block as a data page. To identify pages, the block number is reused as a *page number*. The page number therefore allows to uniquely identify the responsible buffer manager and, in turn, the I/O manager. If the next higher system layer requests access to a page (via the page number), the buffer manager either returns the page from the buffer or instructs the I/O manager to load the corresponding block, depending on whether the page is already in the buffer or not. If a new page is transferred to the buffer, but the buffer is full, a victim has to be chosen for displacement. If this victim was modified, the I/O manager writes it back to external memory. Currently, XTC supports LRD-V2 and LRU [Effelsberg 84] as displacement strategies. A strategy can be configured for each buffer separately during system startup. To synchronize multi-user access to the buffer, a fix/unfix mechanism is implemented on the buffer pages.

3.3.2 Access Services

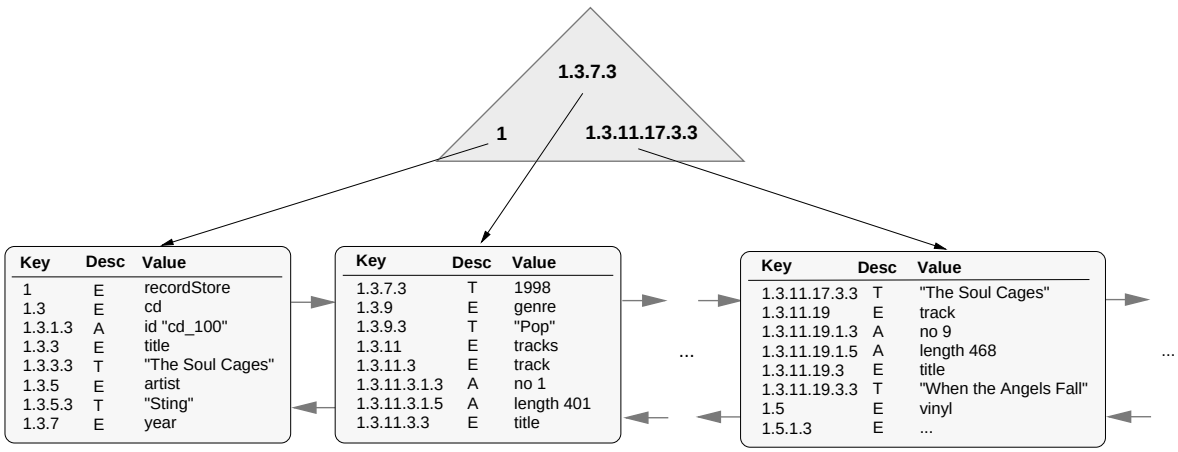
The access services layer (L3) provides the internal record interface and actually implements 1) the mapping from XML documents and collections as taDOM trees to the pages provided by L2; 2) the taDOM access methods from the previous section; and 3) the document scan methods for SAX scans. Furthermore, L3 manages metadata on the stored documents. The three components responsible for these tasks are the *index devices*, the *catalog manager*, and the *record manager*.

Like layers L1 and L2, the functionality of the index services can also be considered standard in many DBMS. To store records as key-value pairs, the component implements a *doubly-linked list*, a *B-tree* [Bayer 72], and a *B*-tree* [Wedekind 74] on the buffered pages of layer L2 using page numbers (i. e., block numbers) as page links. All indexes support prefix compression [Wagner 73] and duplicates on the keys as well as variable key and variable value lengths. Algorithms to access and maintain these index structures are well-known [Härder 01] and, therefore, not further discussed here.

The record manager stores a document (taDOM tree) in a B*-tree index, which is called *document index* or alternatively *document store* in the following. Every B*-tree record corresponds to a taDOM tree node, where the node's DeweyID serves as the record key. Therefore, this storage strategy is called *node-oriented storage* in the following. Obviously, indexed access to nodes via their DeweyIDs is possible. The record value consists of a one-byte descriptor containing information about the stored node (e. g., its type) followed by the actual payload data which, in turn, depends on the node type:

- Attribute root nodes and string nodes are *virtual*, i. e., they are not stored at all, because they were only introduced to facilitate transaction parallelism and do not further carry any semantics.
- For text nodes, the payload contains the content of the string node below (encoded in byte representation). If the content of a text node is larger than a page,

Figure 3.5 Storage of sample document *recordStore.xml* in a B*-tree



the content is distributed over a set of chained pages and referenced from the corresponding record.

- For elements, the payload is a so-called *vocabulary ID* or *vocID* for short. To compress element (and also attribute) names, the record manager administrates a vocabulary containing all the names occurring in one or more documents. A vocID then is an integer uniquely identifying an element or attribute name.
- For attributes, the payload contains both, a vocID and the content of the attribute (encoded in byte representation).

Figure 3.5 illustrates the storage layout for our sample document (vocIDs and descriptors omitted). As you can easily observe, the records in the leaf pages of the B*-tree are stored in document order (due to the implied ordering of DeweyIDs), and, because the DeweyID of two consecutive records often only vary in a few suffix divisions, prefix compression can heavily reduce storage space. Therefore, all documents are stored in a B*-tree index with prefix compression [Bayer 77].

As part of the internal record interface, the record manager has to provide all access methods defined for the taDOM tree as well as the necessary infrastructure for SAX processing. Because this tree is internally stored in a B*-tree, the record manager translates all access methods to appropriate index access methods. To support various other characteristic access patterns, XTC additionally allows to create secondary index structures on the stored document, namely the so-called *element index* and the *ID index*. Because these issues will be discussed in Chapter 6 and Chapter 7, we omit their discussion here.

The third component in the access services layer is the catalog manager, which keeps track of per-document or per-collection metadata. Therefore, the manager stores information about the location of a particular document (referenced to by a unique *document ID*) and its secondary indexes on external memory.

To summarize, the record manager, the catalog manager, and the index services together implement the internal record interface of the XDBMS. Basically, records are byte-encoded taDOM tree nodes. In the next higher level (L4), which implements the external record interface, these records are transformed to XML nodes.

3.3.3 Node Services and XML Services

The interface provided by the *node manager* on Level L4 is very similar to the interface of access services interface on level L3. Therefore, both record manager and node manager support the same operations on the taDOM tree. In fact, for most operations, the node manager just calls the corresponding implementation methods in the record manager. Additionally, however, the node manager has two tasks: First, it has to decode internal records received from the internal record interface and create external XML nodes with readable node names and content. And second, it has to request locks from the lock manager in the transaction services package to protect access to the taDOM tree for multi-user processing. Which locks are actually requested is beyond the scope of this introduction. However, because the taDOM lock protocols are tailored to the specified taDOM access methods (Figure 3.2), it should be intuitively clear that for every operation there is a suitable lock request.

At the last Level (L5), the *XML manager* and the *XPath processor* implement the set-based interface to the XTC system. The XML manager allows to store and reconstruct documents and to evaluate XPath queries using the XPath processor. Note, the XPath processor is a quite simple implementation of a small subset of the XPath 1.0 Recommendation [Clark 99]. Throughout this work, it is replaced by an XQuery processor. Therefore, the XPath processor will not be introduced in more detail.

Besides document storage and reconstruction, the XML manager provides a virtual directory structure for the management of all XML documents and collections in the database. The virtual directory structure has the same functionality as a file system directory structure. It is possible to create and remove directories, store documents in directories, and change/rename directories and documents. Technically, the virtual directory structure is a view on the so-called *master document*. This XML document keeps track of the per-database metadata information, such as, which containers exist, where documents are stored (external memory address), and how they are organized in the virtual directory structure.

3.3.4 Transaction Services

As indicated in Figure 3.4, the *transaction services* are connected to all five system layers. In summary, it is their responsibility to synchronize transactions (i. e., *lock manager* and *deadlock detector*) and to keep log information for recovery in case of system crashes and rollback operations (*transaction manager*). The lock manager receives lock requests from the node manager and checks, whether the current lock request is in conflict with existing locks from other transactions. If so, the requesting transaction is blocked. Otherwise, the lock is granted and the transaction can proceed. To assess the possible concurrency provided by different lock strategies (e. g., taDOM2 vs. taDOM2+ vs. other strategies proposed in the literature), [Haustein 06b] introduced the concept of *meta-synchronization*. For a lock protocol L, the lock manager "translates" the lock requests from the node manager to the appropriate set of lock supported by protocol L. Currently, the lock manager of the system supports 12 different lock protocols.

When objects are locked for multi-user access, deadlocks may occur. Therefore, the *deadlock detector* records transaction dependabilities in a wait-for graph. When a deadlock is detected, the detector chooses a transaction for rollback and restart.

As the last component in the transaction services package, the *transaction manager* (together with the buffer manager), implements the Non-Atomic/Steal/No-Force [Härder 01] variant for page propagation (non-atomic), page replacement (steal), and end-of-transaction handling (no-force). During normal run time, the buffer manager and the transaction manager write the necessary log information into the *container protocols* and in the *transaction log*. Based on this information, the necessary undo and redo recovery can be implemented. A detailed description of the components in this package is beyond the scope of this introduction.

3.3.5 Interface Services

The *interface services* provide the necessary mechanisms to connect to the XTC server from the outside world. Because XDBMS are frequently used in web applications, XTC directly provides access via the HTTP and the FTP protocol, thereby exposing the virtual directory structure. To support DOM and SAX access, as well as calls to the XPath processor, XTC contains a driver package (similar to the Java Database Connectivity; JDBC) allowing to connect to the system via Remote Method Invocation (RMI). In contrast to many other XDBMS, XTC provides a "native" DOM interface, in which all DOM calls are executed inside the system (as opposed to shipping the complete document to the client). A similar functionality is not possible for the SAX parser, because the application logic of the client program cannot run inside the server. Therefore, XTC splits the document into fixed-size chunks of nodes, which are sent to the client to implement the SAX parser. The last component, API RMI is responsible for all other calls to the system, e. g., to store or reconstruct documents or to show monitoring information.

3.4 Summary

This chapter introduced the necessary system-specific preliminaries for this thesis from the XTC system. XTC is almost a full-fledged native XML database system. It provides the necessary interfaces for a meaningful embedding into Java applications and it is able to synchronize DOM and SAX access. Obviously, XTC is still not complete, because a declarative interface to the XML data is missing. The design and implementation of such an interface in the form of an XQuery evaluation engine will now be presented. We take a top-down approach and start with the logical aspects in the next part.

Part II

Logical Aspects of XML Query Processing

Chapter 4 The XML Query Graph Model

A smart model is a good model.

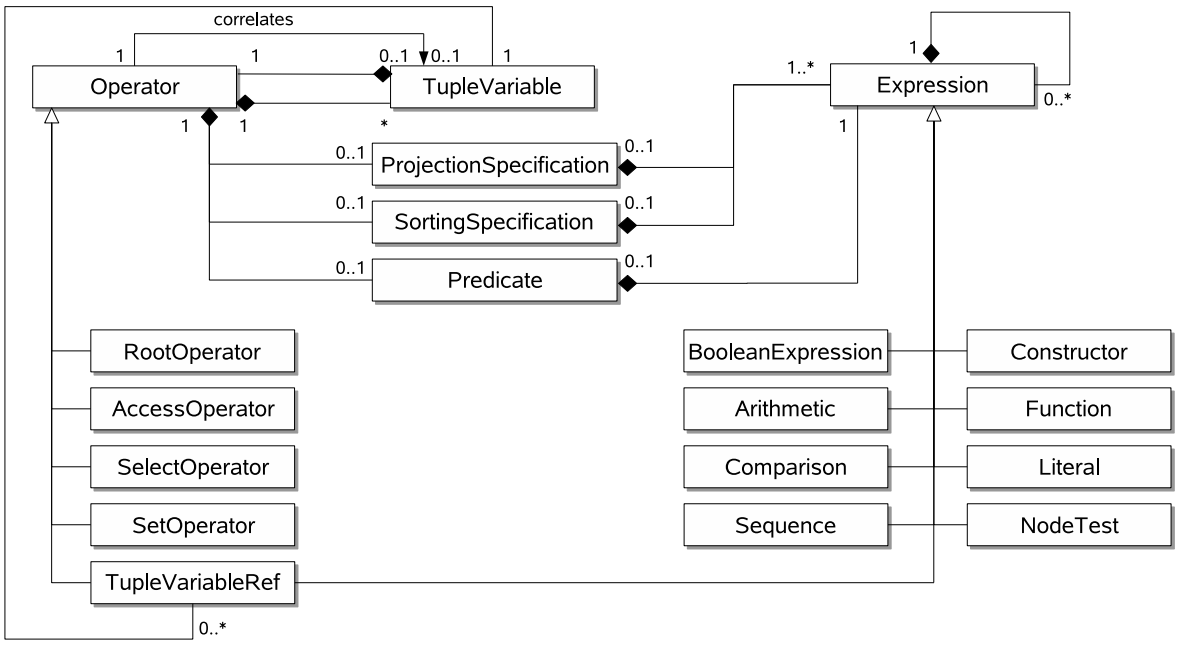
Tyra Banks

In general, queries can be represented in various ways, for example as strings, as abstract syntax trees, or as algebraic expressions. Every form has its right of existence, i. e., the string representation exists for application programs and ad-hoc queries, syntax trees for parsing, and algebraic expressions for formal treatment. Basically, all these forms are simply different syntactical shapes of one and the same semantic thing—namely the query. And all these shapes serve a special purpose. Another syntactical shape introduced in this chapter is the *XML query graph model* (XQGM). The purpose of the XQGM is to serve as a logical internal representation for XML queries. The XQGM is *logical*, because it hides low-level details about how a query is actually evaluated. Nevertheless, it has precise semantics and can capture a substantial fragment of the XQuery language. The separation between a logical and physical view of a query makes sense, because it introduces another level of abstraction and therefore reduces complexity at each level. The XQGM is furthermore *internal*, because it is designed for the needs of the XML query processor frequently requiring functionality for *searching*, *restructuring*, and *mapping* queries or subqueries.

The idea of a query graph model for internal query representation is not new. First publications arose from the Starburst system as early as 1988 [Mavis K. Lee 88, Haas 89, Pirahesh 92]. In this context, the QGM there was developed to capture relational queries. XQGM proposed in this work can be seen as an extension to the original model to integrate support for XML queries. Although this point is not exploited here, XQGM can conceptually capture relational and XML at the same time, i. e., its expressive power is not restricted to only one query language. Additionally, further XML query languages might also be translated to XQGM. In practice, mixes of relational and XML queries might occur, e. g., expressed in the SQL/XML query language. These queries could then be compiled into one XQGM representation and optimized "together". This idea is also followed by major database vendors [Beyer 06, Liu 08]. Although these systems follow a similar approach as this work, their publications remain quite shallow. This work presents the first in-detail presentation on how XML queries can be represented in a Query Graph Model.

The rest of this chapter is organized as follows: First, a slim version of the XQGM syntax and its semantics is introduced. Albeit being slim, this version contains

Figure 4.1 The slim version of XQGM in UML notation



enough operators to capture all supported constructs of the XQuery language. In the second part, the necessary transformation steps, starting from the AST representation generated by the parser and resulting in the XQGM representation, are presented. In the following chapter, the slim version is then extended to full size, i. e., new operators are introduced to capture the semantics generated by various query rewritings.

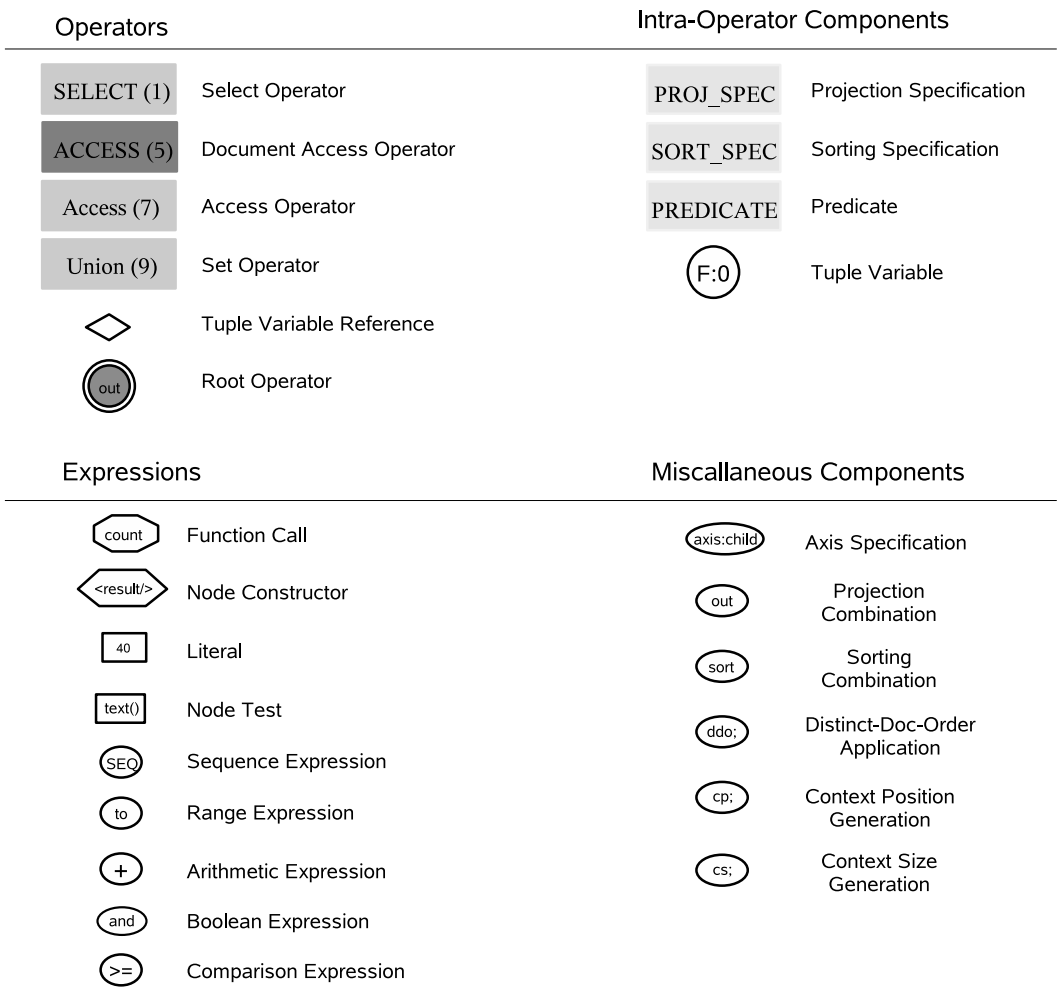
4.1 The XQGM Syntax

Purely introducing syntax without at least some glimpse at the semantics is a little bit cumbersome. To facilitate comprehension, we will, therefore, give a short informal description about the meaning of the syntactic components. The formal description of the semantics will then be presented in the next section.

4.1.1 XQGM Components

An XQGM instance is a box-and-arrow diagram, where boxes are operators for data manipulation and arrows indicate the data flow. Inside each operator box, the operator semantics is also represented by a box-and-arrow diagram. In the following, we will use the term *component* for any box. Sample instances of XQGM have been shown in Figure 2.4 and 2.5 on Pages 20 and 23 (in this chapter, only the components of the first example are introduced; others follow in Chapter 5). Figure 4.1 clarifies the relationship between the various graphical components. They are summarized in Figure 4.2 and introduced in the following (note, a complete example illustrating

Figure 4.2 Components of the slim XQGM version



the components can be found in Figure 4.3):

- **Operators**: An operator is either an *access operator*, a *select operator*, a *set operator*, a *root operator*, or a *tuple variable reference*. The first three components are represented by boxes in XQGM and carry the operator's type (e. g., `ACCESS`, `SELECT`, or `UNION`, `INTERSECTION`, `DIFFERENCE`) and a unique identifier enclosed in braces in the upper part. A special type of access operator is the *document access*, which is depicted in a darker shade (not represented in Figure 4.1 for simplicity). A tuple variable reference is a rhomb that can optionally contain an integer number, the string "cp", or the string "cs". A tuple variable reference has no type information and no identifier. The root operator is represented as a double circle with the word "out" inside.

  Informal semantics: Generally, operators produce streams of tuples and most operators also consume streams of tuples. Thus, as we will also show in the following, the query processor is *tuple-based*. But first, let us discuss the various operator types. The semantics of the *set-based operators* directly emerges from the corresponding constructs in the XQuery language and requires no further explanation. *Access* operators retrieve XML nodes from documents: a *document access*

returns the virtual root node of a document referenced by its name; all other *access operators* return elements, attributes, and so on. The *select operator* is responsible for combining the input of several input operators, and, for that purpose, applies predicates, sorting, and projection. Despite its name, a select operator can also perform joins and sorting[1]. The *root operator* "loops" its input "through". It just serves as a hitch to clarify where the output of the query is delivered. Finally, a *tuple variable reference* allows to peek into the internal state of some select operator and to fetch a field of the operator's current tuple. Thereby, a tuple variable reference can provide a so-called *correlated input* to a subexpression. This subexpression is then called a *correlated subexpression*. For an example, see Figure 2.4 on Page 20. `Select(15)` has a tuple variable reference as input, which peeks into the current input tuple of `select(2)`. The notation inside a tuple variable reference (i. e., *none*, an integer, "cp", or "cs") specifies which field of the tuple will be accessed. *None* implies the first field (at position 0), an integer i implies the ith field, and "cp" or "cs" imply some special fields containing information about the current *context position* or the *context size*.

- **Specifications and Predicates**: Internally, an operator can consist of 1) an optional *projection specification*, 2) an optional *sorting specification*, 3) an optional *predicate*, and 4) a list of zero or more *tuple variables*. Graphically, these components are placed inside the operator box, where tuple variables are represented as circles and the other components are represented as boxes. A tuple variable circle contains a quantifier followed by a colon followed by a unique identifier. Quantifiers are F for *for*, L for *let*, E for *exists*, and A for *for all*. The specification and predicate boxes contain a name indicating the component, i. e., `PROJ_SPEC`, `SORT_SPEC`, or `PREDICATE`. Access operators furthermore contain a graphical description about what they deliver. This description is either depicted by an oval (see first component in Figure 4.2 inside the "Miscellaneous Components" column), in case of an ordinary access operator or by a contained select operator (with a literal for the document name), in case of a document access (see Figure 2.4 on Page 20).

  Projection specifications, sorting specifications, and predicates all contain *expressions*—either one or more, in case of projection and sorting specification, or exactly one, in case of a predicate. An expression cannot exist alone in XQGM; likewise predicates and specifications cannot exist alone. Whenever multiple expressions inside a projection specification occur, their is output combined, which indicated by an oval containing the string "out" (cf. the second component inside "Misc."). Similarly, in projection specifications, expressions are combined by an oval containing the string "sort". The input edges to this component carry an ordering number and a sorting direction (e. g., "descendant"). Finally, the functionality to apply a *ddo* function or to generate positional information is built into the projection specification. The presence of these modifiers is signaled by oval compontents containing the strings "ddo;", "cp;", or "cs;" inside the projection specification.

  Informal semantics: Inside an operator, the tuple variables are responsible to retrieve input tuples and to produce an operator-internal tuple stream (by combining the input). The quantification mode defines how the internal stream is assembled: F leads to an iteration over the input, L groups the input, and E and A are required to express *existential* and *all* quantification. Access operators have no

[1] We chose the term "select" to conform to the terminology in the original Starburst work [Haas 89]

tuple variables, because they receive no input but directly access the document (for example, by evaluating an axis step). However the internal tuple stream is generated, it can be filtered by a predicate, sorted by the sorting specification, and projected by the projection specification. The sorting specification and the projection specification can contain multiple expressions that are evaluated over the internal tuple stream. For sorting, the multiple expressions correspond to multiple sorting dimensions. In a projection specification, every expression generates a field in the output tuple. A special kind of output expression is signaled by an output modifier containing any combination of "ddo", "cp", and "cs". If either *cp* or *cs* is defined there, the projected output tuple stream delivered by the operator is extended with a special field containing 1) the current context position (i. e., the position index of the tuple in the stream), or 2) the current context size (i. e., the size of the tuple stream), or both. If *ddo* is defined, the tuple stream is freed from duplicates and sorted in document order (note, this is only possible, if all tuples in the stream contain exactly one node).

- **Expressions**: The set of expressions emerges from the expressions defined by the XQuery language, i. e., there are *Boolean expressions*, *arithmetic expressions*, *constructors*, and so on. An expression can be composed of other expressions. Whenever this happens, an *expression edge* is drawn as an arrow from the child expression representation to its containing parent expression representation. Note, tuple variable references are also expressions. The following graphical elements are used to display expressions: the *octagon* for a function call, the *hexagon* for a constructor, the *box* for a literal or a node test, an *oval* containing the string "SEQ" for a sequence, an *oval* containing the string "to" for a range expression, or an *oval* containing an arithmetic/Boolean/comparison operator in case of an arithmetic/Boolean/comparison expression.

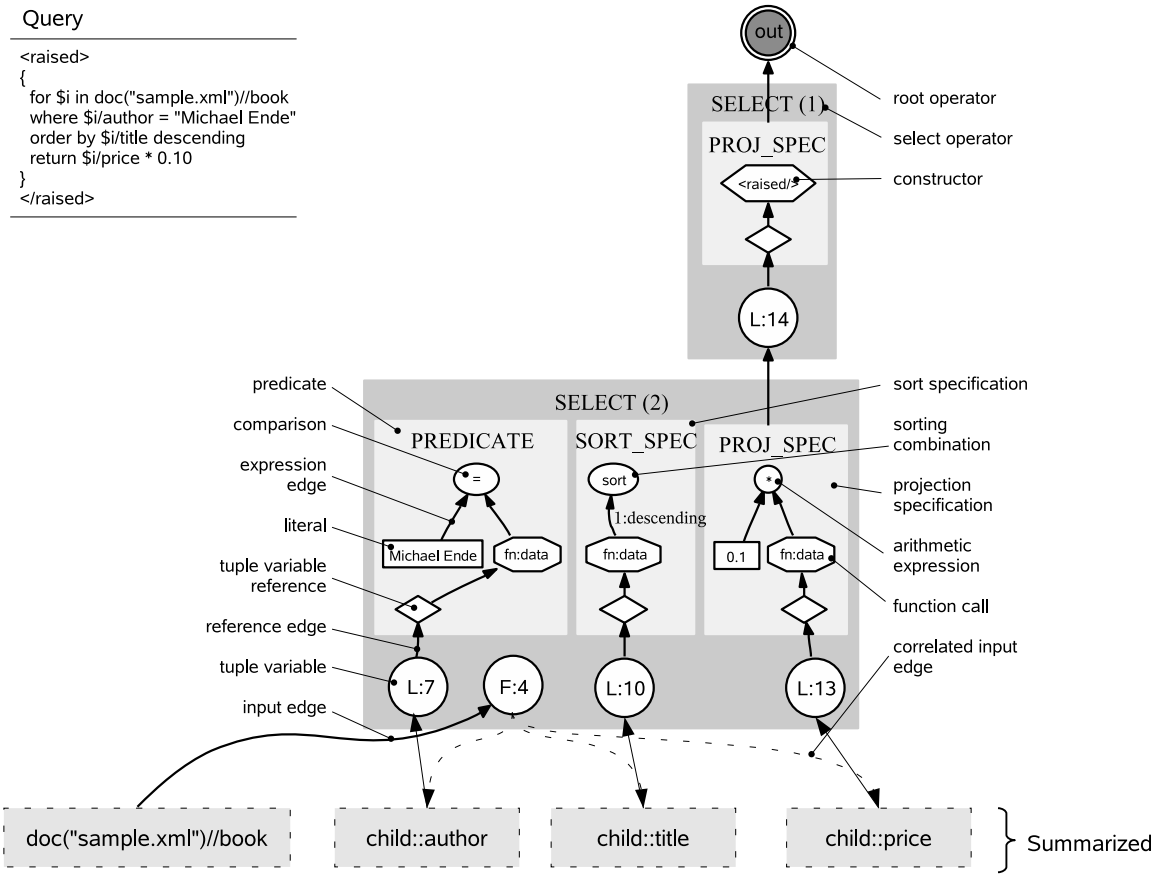
  Informal semantics: The semantics of expressions is imported for the Formal Semantics without change. The only problem is that XQuery expressions operate on items (and item sequences), whereas XQGM operators operate on tuples (and tuple sequences). To translate from one model into the other, tuple variable references (which select exactly one item from a tuple as explained above) are also expressions (see Figure 4.1).

- **Tuple Variables**: The skeletal structure of an XQGM is build upon operators and tuple variables. Between operators and tuple variables, three relations exist: 1) operators contain zero or more tuple variables (represented as circles inside the operator boxes); 2) a tuple variable contains exactly one (input) operator; this relationship is expressed by an *input edge* represented as an arrow pointing from the operator box to the tuple variable circle; and 3) a tuple variable can serve as correlated input to an operator (modeled by the "correlates" association in Figure 4.1); a correlation is represented by a *correlated input edge* which is an arrow with a dotted line from the tuple variable circle to the operator box. The last edge type is a *reference edge* which is an arrow pointing from a tuple variable to the corresponding tuple variable reference (no matter whether the reference occurs as an operator or whether it is contained in an expression).

  Informal semantics: Tuple variables have already been discussed above. They receive and combine the operator's input and can deliver tuples to correlated tuple variable references.

Figure 4.3 contains another example to clarify the syntax of the various components.

Figure 4.3 An XQGM sample instance



4.1.2 Identifying Components

During rewriting and for the definition of the XQGM semantics, a notation to identify the components inside an XQGM instance is required. To save space, this notation shall not be introduced formally, but only by example. A formal description should not be necessary, because the notation is quite straightforward and easy to understand:

- **Literals**: The types in Figure 4.1 are the literals of the notation and refer to the component types of an XQGM instance. We can therefore write *operator*, *constructor*, or *predicate*. Note, the notation is case-insensitive, i. e., "PREDICATE" and "predicate" would refer to the same type of component.
- **Dots**: Let S be an operator. Then we can reference components inside S using the "dot" notation. For example, when S is a select operator, $S.predicate$ references the predicate. The notation also works the other way around. Suppose P is the predicate of a select operator. Then $P.select$ (or, more generic, $P.operator$) reveals the containing operator.
- **Arrows**: If two components are connected by an edge, we use the "arrow" notation. Let T be a tuple variable and O be an operator. Then we write:
  - $T \rightarrow O$, when O is the input operator of T (following the input edge).

  - $O \xrightarrow{p} T$, when T is the containing tuple variable of O (following the input edge in reverse direction).
  - $T \rightharpoonup O$, when O is a correlated operator (following the correlation edge).
  - $O \xrightarrow{p}_{\triangle} T$, when T is the correlated input (following the correlation edge in reverse direction).

  Furthermore, let E be an expression, then we can write $E \rightarrow expression$ to navigate the contained expressions along the expression edge, and $E \xrightarrow{p} expression$ to navigate from an expression to its parent. Finally, let R be a tuple variable reference and let V be a tuple variable, we can write $R \rightarrow V$ and $V \xrightarrow{p} R$.

- **Brackets**: If the dot or the arrow notation is ambiguous, because multiple components of the same type exist, we use the bracket notation to identify the desired component. For example, if a projection specification P contains multiple expressions, P.*expression*$[3]$ references the third one. Note, where necessary, XQGM reveals the order of the graphical components using an integer ordinal number[2]. Furthermore, tuple variables are ordered by their IDs. For example, in Figure 4.3, F:4 precedes L:7 .

- **Question Marks**: We can express an existential condition over the structure of an instance using the "?" notation. For example, S.*predicate*? yields true, if operator S has a predicate, and false otherwise.

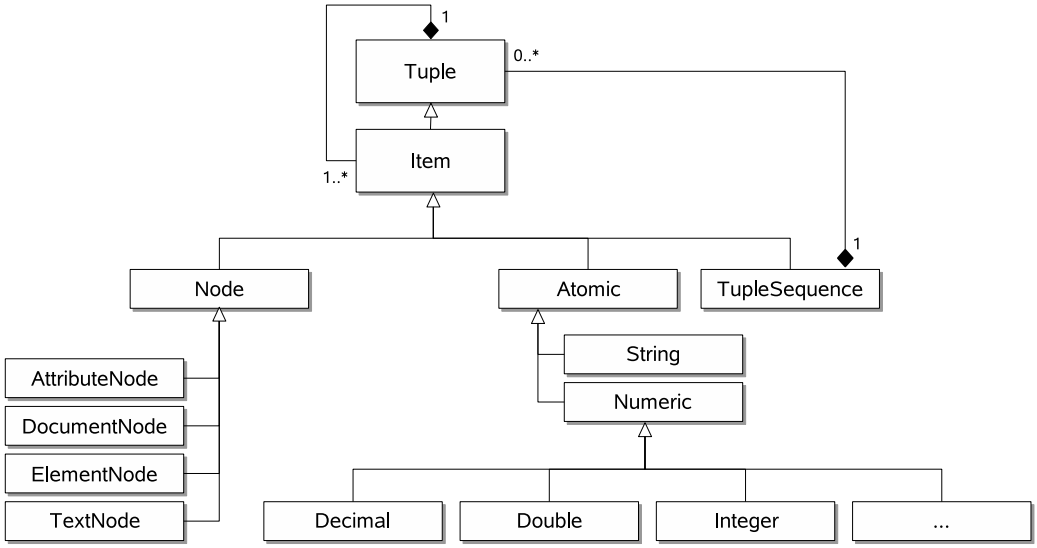
- **Patterns**: We allow to specify patterns on the XQGM operators using braces and the well-known pattern cardinalities zero-or-one (?), one-or-more (+), and zero-or-more (*). For example, given operator O_1, we can specify the pattern $O_1(\xrightarrow{p} T.O)*$ (where T are tuple variables and O are operators) to retrieve all operators up to the XQGM root operator.

4.2 The XQGM Semantics

The semantics of XQGM has to be defined in a precise way, because otherwise we will not be able to capture the semantics of an XML query expression without loss. For that purpose, although XQGM is a logical query representation, an evaluation model has to be developed to clarify how each of the operators in XQGM can be evaluated. In the following, we will first define the data model on which the evaluation model can operate. A data model is essential, because without it, it is impossible to define the behavior of the XQGM operators. The data model will, for example, formally introduce the above mentioned tuples. Then, as another preparation step, we consider the dynamic evaluation environment of the evaluation model. This environment is required to capture a global evaluation context. Finally, the actual XQGM operators are then considered. The problem here is that XQGM is a graphical representation and we need to "attach" semantics to each of the graphical components. We will not do so directly, because this would actually not be possible in a proper way. Rather, an XQGM instance is mapped onto a logical algebra expression (consisting of lower level operators expressing the XQGM semantics).

[2]This is necessary, because the layout algorithm of the program used to render the XQGM instances ignores the correct component order.

Figure 4.4 The data model of the XTC XML query processor



4.2.1 The Data Model

The data model introduced in this section defines the objects on which the logical evaluation model operates. Later on, this data model is then reused to implement the operators in the physical algebra of the query processor (see Chapter 8). We already know an XML query data model, namely the *XQuery Data Model* [Fernández 04]. To summarize, this model defines: *atomic values*, *nodes*, *items* (the union of atomic values and nodes), and *sequences* (ordered lists of items). Unfortunately, this data model is too restrictive for the logical and physical XML operators developed in this work: Intrinsically, XML data is complex and the operators should be able to consume and produce complex data. However, sequences as defined in the XQuery data model, are "flat" structures. In a sense, support for "multi-dimensional" data is missing, i. e., the data model lacks the concept of *tuples* to incorporate complex structures.

Therefore, the definition of the XTC query processing data model subsumes the XQuery data model and adds the *tuple* concept. The structure of the resulting data model is depicted in Figure 4.4. Tuples are the most generic type in the data model. Every data model component is a tuple. Note, in contrast to the relational algebra, the fields of a tuple are not named and can therefore not be referenced by name. Rather, the relative position of a field is used, as we will see in the following. As in the XQuery data model, items are either nodes or atomic values with the corresponding subtypes to represent elements, attributes, and so on. Tuple sequences are also modeled as items and, in turn, contain an ordered list of tuples. This means that sequences can contain tuples containing sequences, and so on. Thus, complex nestings/values are supported. A tuple sequence containing exactly one tuple is said to be *singleton*. A singleton tuple sequence is equal to the contained tuple. Likewise a tuple with exactly one item is called *singleton* and a singleton tuple is equal to the contained item. As a result, a tuple sequence can naturally act as an XQuery sequence, namely when the tuple sequence contains singleton tuples only or when

it is empty (as *empty sequence*). The overview in Figure 4.4 does not contain all types defined by the XQuery data model and, in fact, XTC does not implement all these types. However, this type hierarchy can easily be extended to do so.

Besides the base relationship between the introduced data types, the operations defined on their instances have to be shown, too. Because the data model is derived from the XQuery data model, we naturally also borrow the definition of the operator semantics from there. For brevity, an in-detail definition of the complete semantics is omitted here. Only the most relevant operations and those for which a different notation was chosen shall be highlighted.

Operations on Tuple Sequences

- *Creation*: The creation of an empty tuple sequence is indicated by braces: ()
- *Iteration*: Let S be a tuple sequence, then we can write: "*for Tuple* $t \in S$ *do* ... *end*" to iterate over the tuples in S.
- *Append*: Let S be a tuple sequence and t be a tuple. Then we write $S + t$ to add t after the last tuple in S. Furthermore, let S_1 and S_2 be two tuple sequences, then we write $S_1 + S_2$ to express the concatenation of S_2 after the last tuple in S_1. Note, in both cases, the order of the tuples is maintained inside the sequences.
- *Boolean Value*: The Boolean value of a tuple sequence is *false*, if the tuple sequence is empty. If the tuple sequence is singleton, the Boolean value is the Boolean value of the contained tuple. Otherwise, the Boolean value is *true*. Let S be a tuple sequence. Then we write $S?$ in the following to indicate the Boolean value of S.
- *Tuple Access*: Let S be a tuple sequence and let i be an integer, then we can access the tuple at position i using the bracket notation $S[i]$.
- *Size*: Let S be a tuple sequence, then $|S|$ is the notation to deliver the number of tuples in S.

Operations on Tuples

- *Creation*: The creation of an empty tuple of size x is indicated using the bracket notation $[x]$. The fields of the resulting tuples are undefined. Tuples with undefined fields are never passed between XQGM operators. Rather, they appear "inside" XQGM operators during tuple construction. If we want to specify a default value v, we use the notation $[x, v]$ resulting in a tuple with x fields of value v.
- *Append*: Let t be a tuple with the first undefined field at position x and let i be an item. Then $t + i$ results in a tuple, where item i is placed at field x. Furthermore, let t_1 and t_2 be two tuples where the first undefined field in tuple t_1 is at field x. Then $t_1 + t_2$ is the concatenation of both tuples, where the first field of t_2 is placed at field x. As with sequences, the relative order of tuple fields is maintained.
- *Boolean Value*: The Boolean value of a tuple t is expressed by $t?$, where the result of this operation is the Boolean value of the item at the first field.
- *Field Access*: Let t be a tuple and x be an integer, then we can access the item at field x using the bracket notation: $t[x]$.
- *Size*: The size of a tuple (i. e., the number of fields) can be calculated by $|t|$.

- *Copy*: Function $copy$ clones a tuple t (undefined fields are cloned to undefined fields).

Operations on Nodes and Atomic Values

The operations on nodes directly emerge from the XQuery semantics. Therefore, they are not further introduced. Similarly, we do not explicitly state all operations on atomic values (such as comparisons, arithmetic operations, etc.). However, in the following, we use the generic *navigate* function as a notation for axis evaluations. The function takes a node n and an axis a as parameters and returns all nodes in document order that are reachable from n on a. Furthermore, we assume a generic function *op* that takes an expression and a list of items and evaluates this expression. For example, $op(+, 1, 2)$ yields the value 3, because the operation is "+" and the arguments are the atomic integer values 1 and 2.

4.2.2 Map, Set, Eval and the Logical Algebra

To define the semantics of the XQGM operators, they are translated into an algebraic notation called LAL for *Logical ALgebra*. The translation is defined by a pair of functions called *map* and *set*. Translation is necessary because an XQGM operator is a graphical notation which can become quite complex and, therefore, hard to define properly. For an example, take a look at the select operator in Figure 4.3. In a way, the translation distributes the semantics of an XQGM operator over a set of much simpler algebraic ones. LAL operators will be introduced as required during the discussion of the map and set functions for every XQGM operator.

In general, LAL operators carry a *name* in capital letters, a list of *parameters* in square brackets, and a list of *arguments* in braces, e. g. $SELECT_{[predicate]}(input)$. A parameter can be considered as a part of the operator's definition that influences evaluation. Often, it will be a simple expression, for example, a selection predicate as in the previous example. Arguments, on the other hand, are used to declare input operators. Thus, a LAL expression itself forms a tree of operators. Naturally, LAL operators work on the same data model as the XQGM.

For an XQGM operator X, a LAL operator L is produced by the *map* function:

$$map : X \rightarrow L$$

Besides operators, the logical algebra needs expressions, for example, to handle an XQGM selection predicate or an XML constructor (see Figure 4.1). For this purpose, we introduce LAL expressions, i. e., for each XQGM expression there is a corresponding LAL expression. An XQGM expression E can be transformed into a LAL expression X using the *set* function:

$$set : E \rightarrow X$$

Functions *map* and *set* are responsible to create LAL operators and expressions. The semantics of a LAL operator is defined by the *eval* function, which will be presented as an algorithm in the following. The result of the *eval* function is a tuple sequence. Applying the *eval* function to the root operator of the algebra tree delivers the final result. Therefore, to define the semantics of the XQGM, we only need to define the

eval function on every algebraic operator. Formally, on a LAL operator L, the *eval* function returns a tuple sequence S:

$$eval : L \rightarrow S$$

4.2.3 The Dynamic Evaluation Environment

Similar to the Formal Semantics, an evaluation environment keeping track of context information is defined. This information is required for the evaluation of correlated edges. For XQGM, the dynamic evaluation environment is kept rather simplistic: In the following, C will refer to the *XQGM evaluation context*. C manages a set of mappings from tuple variables T to values of the data model v. Let T be a tuple variable and let v be a value. Then $C(T) \leftarrow v$ assigns value v to the tuple variable stored in context C. A value can be read from the context using notation $C(T)$, where T is a tuple variable.

4.2.4 XQGM Select

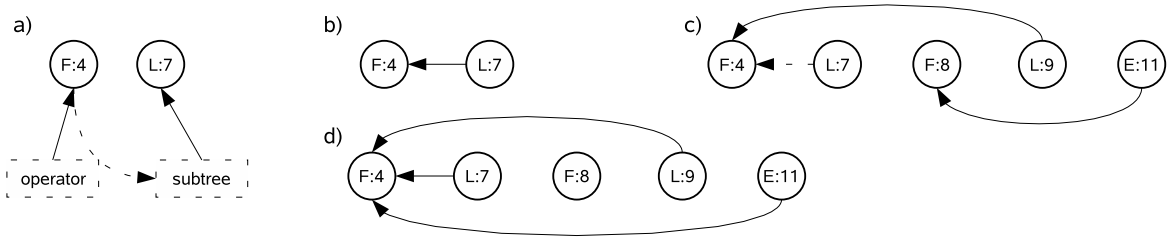
The select operator occurs frequently in XQGM instances. Therefore, it is the first operator to be discussed: A select operator S with a projection specification $X = S.projectionSpecification$, a sorting specification $U = S.sortingSpecification$, a predicate $F = S.predicate$, and n tuple variables $T_1 = S.tupvar[1], \ldots, T_n = S.tupvar[n]$ is mapped to LAL operators as follows:

$$\begin{aligned}
&map(S) = \\
&\quad DDOCPCS_{[X.ddo?,X.cp?,X.cs?]}(\\
&\quad\quad PROJECT_{[set(X.expression[1]),\ldots,set(X.expression[n])]}(\\
&\quad\quad\quad SORT_{[U.modifiers,set(U.expression[1]),\ldots,set(U.expression[n])]}(\\
&\quad\quad\quad\quad SELECT_{[set(F.expression)]}(\\
&\quad\quad\quad\quad\quad TUPGEN_{[T_1,\ldots,T_n]}(\\
&\quad\quad\quad\quad\quad\quad map(T_1 \rightarrow operator_1), \\
&\quad\quad\quad\quad\quad\quad \ldots, \\
&\quad\quad\quad\quad\quad\quad map(T_n \rightarrow operator_n))))))
\end{aligned}$$

In case, any of the set components (X, U, F, and T_i) is not defined in the select operator, the corresponding LAL operator is simply not generated, e. g., if an XQGM select does not have a predicate F, the algebraic select operator would be missing. The given mapping essentially means that from the tuple variables, a tuple sequence is generated by *TUPGEN* which is filtered by a *SELECT* operator, then sorted by the *SORT* operator, projected by *PROJECT*, and finally modified by the *DDOCPCS* operator. Note, the *DDOCPCS* operator is responsible to establish the distinct document order (*ddo*), the context position (*cp*), or the context size (*cs*). All these operators will be introduced in detail below.

Now, the *set* and *eval* functions of the five operators required to map an XQGM select operator shall be discussed. We will start with the innermost *TUPGEN* operator.

Figure 4.5 Examples of dependent tuple variables



TUPGEN

TUPGEN is the tuple generator. Basically, it reads tuples from the input operators and generates a sequence of output tuples (this sequence was referred to as "operator-internal tuple stream" in Section 4.1.1). An analogy to this operator in the relational algebra is the *Cartesian product*. You can observe this correspondence in the above semantics of the XQGM select operator (essentially an XQGM select is a *join*, and a join is a Cartesian product followed by a selection). However, here the "Cartesian product" is ordered. As stated above, if an XQGM select operator does not have any tuple variables, the tuple generator is not required and, therefore, not generated by the map function. Furthermore, the arguments of the *TUPGEN* operator are the mapped input operators of the various tuple variables ($map(T_i \rightarrow operator_i)$ for all T_i). Therefore, a tuple variable has no influence on the input of a *TUPGEN* operator (because the argument's results are simply passed to the tuple generator as input). However, tuple variables have influence on the way how the output of TUPGEN is generated. Therefore, they are passed as parameters.

Tuple variables define how input data is processed and combined. In summary, we said that the set of tuple variables is ordered by their IDs, that every tuple variable has some input and a quantifier, and that it can serve as the starting point for references and correlated subexpressions (via the correlation edge). However, we did not yet pose any further restrictions on tuple variables, thus, allowing to define very complex interdependencies between them. Some examples are depicted in Figure 4.5 and will be discussed below. The problem with complex interdependencies is that they make the evaluation process (i. e., the *eval* method) unnecessary hard to specify. For XQuery, rather simplistic dependencies are quite sufficient. In the following, we will define what kind of dependencies are supported by the evaluation model (note, of course, we may only rely on the supported settings, when XQuery expressions are translated into the XQGM):

Let T_1 and T_2 be two tuple variables of the same select operator S. Then, we say that T_2 *depends on* T_1, if T_1 has a correlated edge into the subtree of T_2 (i. e., if the path $T_1 \rightharpoonup O(\xrightarrow{p} T.O)* \xrightarrow{p} T_2$ exists, where T are tuple variables and O are operators). We furthermore define that a tuple variable T_2 is *dependent*, if it depends on a tuple variable T_1 of the same operator S. If no such T_1 exists, T_2 is *independent*.

The above definition states that a dependent tuple variable contains a correlated subexpression. Figure 4.5a presents an abstract XQGM with such a situation. Dependent tuple variable L:7 depends on the independent tuple variable F:4. In the

remaining examples depicted in Figure 4.5, the fact that a tuple variable depends on another tuple variable is represented by a solid arrow, like in Figure 4.5b.

So far, tuple variables can carry arbitrary quantifiers and can depend on each other in plentiful ways, thereby inducing complexity into the tuple generation process. As mentioned above, this complexity is not required. We therefore restrict the way how tuple variables can depend on each other by the following conditions:

1. Let $T_1, \ldots, T_n$ be the dependent tuple variables of one and the same select operator S. Then they all have to depend on one and the same independent tuple variable T (of S).

2. Let S be an operator and let T be the independent tuple variable (as in 1). Then T is the first tuple variable among all tuple variables in S, and T is either *for*-quantified or *let*-quantified.

A counter example for the first condition is shown in Figure 4.5c, where essentially two independent tuple variables exist, on which dependent tuple variables depend. A correct example is shown in Figure 4.5d, where all dependent tuple variables depend on the first one, which is *for*-quantified. The two requirements above ensure that tuple variables can be evaluated in a simple and meaningful way (from left to right). The proposed restrictions immediately become clear, when the semantics of correlated subexpressions are introduced. Until then, however, we simply take them as syntactical constraints.

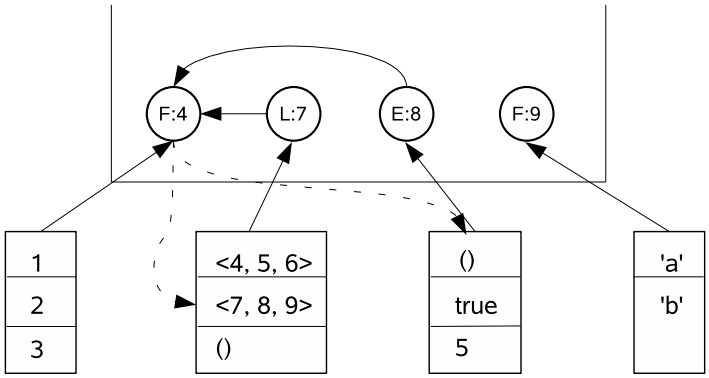
Just like the relational Cartesian product, the *TUPGEN* operator combines several input streams to one output stream by creating tuples. The data model (cf. Section 4.2.1) defines tuples with positional access (i. e., tuple fields carry no name). Hence, for stream combination, the *TUPGEN* operator has to be aware of positional tuple information and, in particular, the cardinality (number of fields) of the input tuples is required. Therefore, we design XQGM operators in a way such that they always deliver a *homogeneous* output stream, where all tuples have the same number of fields (i. e., the same cardinality). As a consequence, the size of a tuple generated by any XQGM operator can be calculated from an XQGM instance (how this actually is accomplished will be shown, when projection specifications are discussed on Page 65). To express cardinality, we extend the notation introduced in 4.1:

Let O be an XQGM operator. Then the cardinality (number of fields) of the tuples generated by O is written as $|O|$. Likewise, let T be the tuple variable containing O, then the cardinality of T is $|T| = |O|$. Finally, let $M = T_1, \ldots, T_n$ be the set of tuple variables of one and the same select operator S, then $|M| = |T_1| + \cdots + |T_n|$ is the cardinality of the set of tuple variables.

TUPGEN can be seen as the "heart" of XQGM, because it defines the tuple flow at the very bottom. Depending on the tuple variable configuration, the algorithm can get quite complex. Therefore, before the algorithm itself is shown in pseudocode, we look at a small synthetic example: Figure 4.6 shows an operator with four tuple variables, of which the second and the third depend on tuple variable one. As input, we assume the values in the boxes below each operator. Among the first three input boxes, the values partly depend on each other via a correlation. We assume that the positions of the values inside the boxes express the correlation, e. g., the value `1` corresponds to sequence `<3, 4, 5>` and to the empty sequence `()`. Note, the last box is not correlated. Integer `2` corresponds to sequence `<7, 8, 9>`, and so on.

During evaluation, the *TUPGEN* operator first evaluates the tuple variable, on

Figure 4.6 A tuple generation example



which the other ones depend, i. e., F:4. Because this tuple variable is *for*-quantified, it starts an iteration over the received result sequence (`<1, 2, 3>`). In each step, the current value is written into an *evaluation context* C (defined previously). This context keeps track of correlated values (and is, therefore, similar to the dynamic evaluation context in XQuery). With the written value (e. g., `1`), the other tuple variables are evaluated, each returning a result sequence: `<4, 5, 6>` in case of L:7, `()` in case of E:8, and sequence `<'a', 'b'>` in case of F:9[3]. These intermediate results are then treated according to the quantifier of their corresponding tuple variables. For *for* and *let*, essentially nothing happens. In case of the existential quantification at tuple variable E:8, the returned value is interpreted. In the example, we have an empty tuple sequence, which is interpreted as *false* (as in the Formal Semantics for empty item sequences). Now, the ordered product of all these intermediate values is computed by combining the input tuples into a stream of output tuples. In our example, we get for the first iteration the two tuples `[1, <4,5,6>, false, 'a']` and `[1, <4,5,6>, false, 'b']`. In the next iteration, value `2` is written into the context at tuple variable F:4, and the tuples `[2, <7,8,9>, true, 'a']` and `[2, <7,8,9>, true, 'b']` are produced, etc. Note, in this example, the cardinality of each tuple variable is one, and the cardinality of all tuple variables is four, which is why the *TUPGEN* algorithm will always return tuples of that size (even, if any input tuple variable does not contribute any value).

With the previous definitions and the example in mind, we can now look at the algorithm. Listing 4.1 presents the evaluation of the *TUPGEN* operator. It consists of the top level *eval* function which, in turn, rests on the three helpers *results*, *enqueue*, and *product* shown in Listing 4.2. Before going into details, we can sketch these three functions as follows: *results* is responsible to evaluate exactly one input operator of *TUPGEN* to a tuple sequence; the produced tuple sequences are then *enqueue*d into a set S of tuple sequences, before the ordered Cartesian product is computed by the *product* function.

The *eval* function first constructs three arrays T, O, and S. T contains all tuple variables, O all operators, and S is a list of tuple sequences that will carry intermediate results. At the beginning, S is initialized with an array of empty tuple sequences (lines 4–6). The main algorithm itself distinguishes two cases: T contains depen-

[3] In case of F:9, the complete input sequence is delivered, because the tuple variable is independent of F:4.

Listing 4.1 $eval(TUPGEN_{[T_1,\ldots,T_n]}(O_1,\ldots,O_n))$

```
   Input: Parameters: Tuple Variables T_1,...,T_n; Arguments: Operators O_1,...,O_n
   Output: Tuple Sequence R
 1 begin
 2     TupleVariables T ← T_1,...,T_n; // the tuple variable array
 3     Operators O ← O_1,...,O_n; // the operator array
 4     for T_i ∈ T do
 5         S_i ← (); // temporary tuple sequence
 6     end
 7     TupleSequences S ← S_1,...,S_n; // array for temporary tuple sequences
 8     if a tuple variable depends on T_1 then
 9         S_1 ← eval(O_1); //evaluate operator O_1 to temporary sequence S_1
10         if S_1 is empty then
11             return ();
12         else
13             if T_1 is 'for' then
14                 for Tuple t ∈ S_1 do
15                     S ← enqueue(t, S, T); // receive input from other operators and write into S
16                     R ← R + product(t, S, T); // create the "product" over the intermediate results in S
17                     clear(S);
18                 end
19             end
20             if T_1 is 'let' then
21                 S ← enqueue(S_1, S, T);
22                 R ← R + product(S_1, S, T);
23                 clear(S);
24             end
25             if T_1 is 'exists' or T_1 is 'all' then
26                 raise error("Unsupported quantification");
27             end
28         end
29     else
30         S ← enqueue(NULL, O, T);
31         R ← R + product(NULL, S, T);
32     end
33     return R;
34 end
```

dent tuple variables or not. The first case in treated in lines 8–29; the second case is treated in the rest of the code. We start the discussion with the first problem. If T contains dependent tuple variables, the first tuple variable T_1 defines the correlated input (according to the requirement posed above). In a first step, the input operator O_1 of T_1 is evaluated (via a recursive call of the *eval* function on the input operator), returning a sequence of tuples. In lines 10–12, if this tuple sequence is empty, the algorithm also returns an empty sequence (because no input to evaluate a dependent tuple variable has been found).

If the tuple sequence is not empty, the type of T_1 determines the further actions. If T_1 is *for*-quantified, the evaluation algorithm needs to iterate over the input sequence of T_1. This happens in lines 14–18. For every tuple, the *enqueue* function is called, which evaluates the remaining input operators and puts their results into the intermediate result sequences $S_2,\ldots,S_n$ (inside S). In the next step, the tuples in the intermediate result are combined by the *product* function and appended to the result sequence R. After all combinations out of the current sequences in S are computed, their values are not required anymore and can be deleted (by the *clear* function). A *let*-bound tuple variable can also provide correlated input. However, to evaluate *let*, no iteration is necessary (see lines 20–24). Tuple sequence S_1 is used

Listing 4.2 Functions *enqueue*, *results*, and *product*

a) **Function:** *enqueue*

```
    Input: Tuple t, TupleSequences S, TupleVariables T
    Output: TupleSequences S
 1  begin
 2      if t is not NULL then // context tuple given
 3          C(T_1) ← t; // store context
 4      end
 5      for O_i ∈ O_2, ..., O_n do // remaining operators
 6          S_i ← results(O_i, T_i);
 7          if S_i is empty then // pad empty tuple
 8              S_i ← S_i + [|T_i|, ()];
 9          end
10      end
11      return S;
12  end
```

b) **Function:** *results*

```
    Input: Operator O, TupleVariable T
    Output: TupleSequence S
 1  begin
 2      if T is 'for' or T is 'let' then
 3          S ← eval(O);
 4      end
 5      if T is 'exists' then // check one
 6          TupleSequence K ← eval(O);
 7          S = S + K?; // Boolean value
 8      end
 9      if T is 'all' then // check all
10          TupleSequence K ← eval(O);
11          for Tuple t in K do
12              if not t? then
13                  S ← S+FALSE;
14                  return S;
15              end
16          end
17          S ← S+TRUE;
18      end
19      return S;
20  end
```

c) **Function:** *product*

```
    Input: Tuple t, TupleSequences S,
           TupleVariables T
    Output: TupleSequence R
 1  begin
 2      if |S| = 1 then // → done
 3          R ← S_1;
 4          return R;
 5      end
 6      TupleSequence I ← (); // intermed. seq.
 7      TupleSequence R ← (); // result seq.
 8      if T_1 is 'for' then
 9          if a tuple variable depends on T_1 then
10              I ← I + t;
11          else
12              I ← I + S_1;
13          end
14          for Tuple t ∈ I do
15              Tuple n ← [|T|];
16              n ← n + t;
17              R ← R + n;
18          end
19      else if T_1 is 'let' then
20          Tuple n ← [|T|];
21          n ← n + S_1;
22          R ← R + n;
23      end
24      // now R contains 'unfinished tuples'
25      for S_i ∈ S_2, ..., S_n do
26          I ← ();
27          if T_i is 'for' or T_i is 'exists' or T_i is 'all'
            then
28              for Tuple t ∈ R do
29                  for Tuple n ∈ S_i do
30                      Tuple c ← copy(t);
31                      c ← c + n;
32                      I ← I + c;
33                  end
34              end
35          end
36          if T_i is 'let' then
37              for Tuple t ∈ R do
38                  t ← t + S_i;
39                  I ← I + t;
40              end
41          end
42          R = I;
43      end
44      return R;
45  end
```

"as a whole" by passing it to the *enqueue* function. The rest remains the same as before. Finally, *exists* and *all* cannot serve correlated inputs. Therefore, they result in an exception.

In the second case, when all tuple variables are independent, the evaluation becomes very simple (lines 29–32): All remaining intermediate results are computed by the *enqueue* function. Then, these intermediate results are combined by the *product* function to obtain the final result in R.

Functions *enqueue* and *product* are depicted in Listings 4.2a and 4.2c. The first func-

tion fills up the intermediate tuple sequences for the input operators that have not been evaluated in the *eval* function (i. e., $S_2, \ldots, S_n$). If a non-NULL tuple is passed, *enqueue* is called for a tuple variable set with dependent variables. Then it is necessary to put the passed tuple into the dynamic evaluation context C for later reference by correlated operators (lines 2–4). After that, each input operator O_i is evaluated using the *results* function and the resulting sequence is stored in the corresponding S_i. It could happen that such an intermediate result is empty. In this situation, we have to remember that the tuple generator has to produce a correct result cardinality. Assume that O_i delivers an empty tuple, but the cardinality $|T_i|$ on that input is larger than 1. Then it is not sufficient to just keep the empty sequence. Rather, a tuple with $|T_i|$ empty sequences needs to be generated. This happens in line 8, where the bracket notation is used to create the tuple.

Function *enqueue* does not call *eval* directly on the input operators. Rather, the *results* function is used. This function is a small wrapper that ensures the correct semantics for tuple variables with *all* and *exists* quantifiers. On simple *for* and *let* quantification, *results* calls the *eval* function directly on the input operator (cf. Listing 4.2b, lines 2–4). When the quantifier is an *exists*, the Boolean value (using the "?" notation) is computed on the resulting intermediate sequence (lines 5–8). As a result, the Boolean value is kept (and the intermediate sequence is discarded). Similarly, on an *all* quantification, the Boolean value for every tuple in the intermediate sequence is evaluated. If at least one of them evaluates to false, the FALSE literal is used. Otherwise, the TRUE literal is written into the intermediate result sequence. This behavior is also described in the example shown in Figure 4.6.

After the intermediate result sequences have been evaluated, they can be combined into the final result. This is the task of *product*. Obeying the tuple variable's quantifiers, this methods computes an *ordered Cartesian product* on the intermediate result in a nested loops fashion. In the trivial case that only one tuple variable is defined, the algorithm presented in Listing 4.2c directly delivers its corresponding intermediate tuple sequence (lines 2–5). Otherwise, two tuple sequence are initialized (I and R), the last one of which will carry the final result. If the first tuple variable T_1 is *for* quantified, I is either initialized with the context tuple t, or with the first intermediate result sequence S_1 depending on the existence of dependent tuple variables. Then, a list of "unfinished tuples" is stored in R (lines 14–18). The same is done in case of a *let* quantification. Here, however, the complete intermediate sequence S_1 is written into the "unfinished tuple".

After initialization, the unfinished tuples are filled up with the values generated for the remaining tuple variables in S_2 to S_n. Tuple variables with *for*, *all*, and *exists* quantification result in an iteration and a production of new tuples (by a call to the *copy* function) using the unfinished tuples in R as a template (lines 27–35). Note, in case of *all* and *exists*, the corresponding intermediate tuple sequence contains only one Boolean value (generated in the *results* function). If the tuple variable is *let*-quantified (lines 36–41), the current sequence S_i is appended to the current unfinished tuple t.

SELECT

The LAL select operator is used to "implement" the predicate testing semantics in the XQGM select operator. As presented in the mapping on Page 55, a select

Listing 4.3 The *set* function

```
   Input: XQGM Expression E
   Output: LAL Expression R
 1 begin
 2     if E is tuple variable reference then
 3         TupleVariable T_k ← (E → tupvar); // dereference tuple variable reference
 4         // calculate relative access position for tuple generated by TUPGEN
 5         Integer j ← 0; // index for tuple access position
 6         for T_i ∈ T_1, ..., T_k do
 7             j ← j + |T_i|; // add up tuple sizes
 8         end
 9         if C.cp? then // tuple variable reference for context position access
10             j ← j + 1; // use relative index 1
11         if C.cs? then // tuple variable reference for context size access
12             j ← j + 2; // use relative index 2
13         if not C.pos? and not C.cs? then // tuple variable reference for normal access
14             j ← j + C.pos; // use position directly
15         end
16         return LalTupleAccess(j); // new tuple access expression
17     end
18     LALExpressions L ← (); // initialize array of child LAL expressions
19     for Expression e ∈ (E → expression) do
20         L ← L + set(e); // recursively convert child expression and append to LAL children
21     end
22     return expr(E, L); // create the resulting LAL expression
23 end
```

operator is parameterized with the converted XQGM predicate F:

$$SELECT_{[set(F.expression)]}(\dots)$$

If no such predicate exists, the LAL select operator is simply not generated. Before the *eval* function of the select operator is presented, we have to take a look at the parameter conversion defined by the *set* function.

Let F be an XQGM select predicate. Then the *set* function is called with $F.expression$ as parameter. A predicate is a graphical representation of a simple XQGM *expression* (see Figure 4.1), e. g., a Boolean expression, an arithmetic expression, a comparison expression, and so on. As we will see, the semantics of expressions is directly "imported" from the Formal Semantics without change into XQGM. This means that XQGM expressions operate on plain items (rather than on tuples). However, as introduced with the *TUPGEN* operator, items do not flow between operators, but tuples (containing items). As explained before, to translate between the tuple stream of LAL operators and the item stream of expressions, we need to project the relevant information from input tuple streams. The link here are *tuple variable references*, which are also modeled as expressions (again, see Figure 4.1). Tuple variable references project exactly one field of a tuple, the position of which has to be calculated on the set T of tuple variables. This calculation is one task of the *set* function. The other one is to translate XQGM expressions into LAL expressions (which can be set as parameters to LAL operators).

Figure 4.1 suggests that expressions contain other expressions. For example, the comparison predicate expression in `select(2)` of Figure 4.3 contains a literal ("Michael Ende") and a function call (*fn:data*), which in turn contains a tuple variable reference. For the following discussion, we disregard specific types and take a generic view: Let E be an XQGM expression, then $E \rightarrow expression$ returns all its

Listing 4.4 $eval(SELECT_{[E]}(O))$

```
   Input: Parameter: Expression E, Argument: Operator O
   Output: Tuple Sequence R
1  begin
2  |  TupleSequence S ← eval(O);
3  |  for Tuple t ∈ S do
4  |  |  if E(t)? then
5  |  |  |  R ← R + t;
6  |  |  end
7  |  end
8  |  return R;
9  end
```

child expressions by navigating the *expression edges* (as introduced in the notation on Page 46). For example, if E is a comparison, $E \rightarrow expression$ returns the literal and the function call. Furthermore, we introduce a notation to construct a LAL expression R: Let $L = L_1, \ldots, L_n$ be LAL expressions and let E be an XQGM expression. Then $expr(E, L)$ constructs a LAL expression for E on $L_1, \ldots, L_n$ (as children).

With these preliminaries, the *set* function has the form depicted in Listing 4.3. As stated above, it translates tuple variable references to tuple access positions (lines 2–17) and it converts XQGM expressions to LAL expressions (lines 19–39). A tuple variable reference has three different functionalities: It can access context position information (depicted by the string "cp" inside a rhomb), context size information ("cs" inside a rhomb), or a relative position (depicted as an integer inside a rhomb or a blank rhomb, if that integer has the value 0). Remember that the input is generated by *TUPGEN*. We can therefore compute the relative position j of the referenced tuple variable T_k (line 9–12) as a basis to compute the access position. Positional information is generated by the *DDOCPCS* operator, which will be introduced below. Here, we only state that, if *DDOCPCS* receives an input tuple t, it can attach the tuple's context position as a new field and its context size as another new field. Therefore, to reference *cp*, we have to add 1 to j (line 10) and, otherwise, if *cs* is referenced (line 12), 2. If no context information is referenced, the integer inside the reference is simply added (line 14). With the correct access position, the LAL tuple access expression can then be instantiated and added to the list of child expressions (line 16).

If the passed expression is not a tuple variable reference, an empty array of LAL expressions is initialized (line 18). It will carry the translated child expressions. For each child expression of E, *set* recursively calls itself on the child and the result is written to the array of converted children. Finally, the current expression is converted and returned (line 39). Note, the method does nothing else than keeping the structure of an expression and converting tuple variable references to access positions.

With the converted parameter, we can now specify the semantics of the select operator. The pseudocode is shown in Listing 4.4. For each returned tuple generated by the *eval* function on the input operator, predicate expression E is evaluated (notation $E(t)$, see below). If the Boolean value ("?") of the returned result is true, the tuple belongs to the final result, otherwise it is discarded. Note that the select operator does not alter the order of the input sequence.

Listing 4.5 The evaluation of expression E on tuple t: $E(t)$

```
   Input: Parameter: Expression E, Tuple t
   Output: Item i
 1 begin
 2     Items I ← (); // initialize array of items for intermediate results
 3     for LalExpression C ∈ E.expression do // for each child expression E consists of
 4         if not C is LalTupleAccess then
 5             I ← I + C(t); // recursive evaluation on child expression
 6         else
 7             I ← I + t[C.j]; // access tuple at position j defined by C
 8         end
 9     end
10     return op(E, I[1], ..., I[n]);
11 end
```

The last piece missing now is the evaluation of the predicate expression E with a tuple t. As before, we abstract from specific expression types. We assume that the semantics of an expression E is captured by a generic *op* function that takes an expression E and items $v_1, \ldots, v_n$ for evaluation. For example, for the arithmetic expression "1 + 3" we can write $op(+, 1, 3)$ and can rely on the Formal Semantics for the definition of the semantics of *op*. Furthermore, we assume that $E.expression$ provides access to the LAL child expressions of E and that, for a LAL tuple access operator C (as introduced in the *set* function), notation $C.j$ returns the access position. The algorithm presented in Listing 4.5 then defines how expressions are evaluated. Note, the algorithm is structurally similar to the *set* function, i. e., it recursively descends down the expression and evaluates the tuple access operator at the very bottom.

In the following, the *set* function and the evaluation of expressions on tuples $E(t)$ will be reused to evaluate the *SORT* and the *PROJECT* operator.

SORT

The LAL sort operator defines the semantics of an XQGM sort specification. As presented, if U is a sort specification, it is mapped as

$$SORT_{[U.modifiers, set(U.expression[1]), \ldots, set(U.expression[n])]}(\cdots)$$

Interestingly, XQuery ordering semantics is defined on tuples. As already mentioned, the XQuery data model does not support tuples. Therefore, the XQuery recommendation does not specify the semantics of the *order by* clause formally. In XQGM however tuples are supported. Therefore, we can define the XQGM ordering semantics properly.

The sort operator is parameterized with the ordering modifiers from the sorting specification ($U.modifiers$) and a set of sorting expressions that are immediately translated by the *set* function from above. Ordering modifiers directly emerge from the XQuery language, and for every translated expression, there is exactly one of these modifiers. Sample modifiers are *ascending*, *descending empty greatest*, and so on. Furthermore, the order among the sort expressions is significant, i. e., the first expression acts as the primary sort criterion and the last expression acts as the last sort criterion. In the following, we only show how tuples are "prepared" for sorting and not how they are actually sorted (w. r. t. the sorting specification). This means,

Listing 4.6 $eval(SORT_{[M,E_1,\ldots,E_n)]}(O))$

```
   Input: Parameter: Modifiers M, Expressions E_1, ..., E_n, Arguments: Operator O
   Output: TupleSequence R
 1 begin
 2     TupleSequence S ← eval(O);
 3     Integer i ← |S[1]|; // retrieve tuple cardinality
 4     for Tuple t ∈ S do
 5         for Expression E ∈ E_1, ..., E_n do
 6             Item v ← E(t); // evaluate sorting expression on t
 7             t ← t + v; // append to tuple
 8         end
 9         R ← R + t;
10     end
11     return sort(M, R, i + 1); // sort the prepared tuple sequence, where relevant fields start at i + 1
12 end
```

we hide the actual sorting behind the *sort* function, which takes the sorting modifiers M, the prepared tuple stream R, and the start index of the tuple fields i, on which sorting is based as input. The resulting algorithm is presented in Listing 4.6.

PROJECT and DDOCPCS

In contrast to a relational projection, which simply retains certain fields from input tuples, the XQGM projection specification allows to compute expressions over input tuples. This is necessary, for example, to construct new elements, as shown in operator `select(1)` (Figure 4.3 on Page 50). As the previous ones, this operator also relies on the *set* function to convert input expressions:

$$PROJECT_{[set(X.expression[1]),\ldots,set(X.expression[n])]}(\ldots)$$

The straightforward evaluation of a projection is shown in Listing 4.7a. For every input tuple t, all expressions are evaluated. Each evaluation returns an item which is written into a new tuple n. The cardinality of the resulting tuples therefore corresponds to the number of expressions.

Another operator that depends on the XQGM projection specification is *DDOCPCS*. The *DDOCPCS* operator can: 1) reorder a sequence of nodes into distinct document order; 2) attach a context position (*cp*) information to a tuple; and 3) attach a context size (*cs*) information to a tuple. The first property is required to implement the *ddo* function, the latter two to evaluate positional predicates. Projection specification X signals the presence of any of these functions. During the creation of a *DDOCPCS* operator, they are passed as parameters

$$DDOCPCS_{[X.ddo?,X.cp?,X.cs?]}(\ldots)$$

Figure 4.7b presents the straightforward implementation of the *DDOCPCS* operator. When a reordering in document order is required, the *ddo* function is called. For brevity, its semantics is not formally introduced here. We simply state that this function interprets input sequence S as an item sequence (i. e., a tuple sequence with singleton tuples), where each item is a node. If a non-node item is found, *ddo* raises an error. The context position and the context size are added to each tuple of the sequence by counting (the current position is kept in i and the size derived from $|S|$).

Listing 4.7 $eval(PROJECT_{[E_1,\ldots,E_n]}(O))$ and $eval(DDOCPCS_{ddo,cp,cs}(O))$

a) $eval(PROJECT_{[E_1,\ldots,E_n]}(O))$

```
   Input: Parameter: Expressions E_1, ..., E_n, Arguments: Operator O
   Output: TupleSequence R
 1 begin
 2     Expressions E ← E_1, ..., E_n;
 3     TupleSequence S ← eval(O);
 4     for Tuple t ∈ S do
 5         Tuple n ← [|E|]; // create empty tuples of size |E|
 6         for Expression C ∈ E do
 7             n ← n + C(t); // evaluate expression and append to tuple
 8         end
 9         R ← R + t;
10     end
11     return R;
12 end
```

b) $eval(DDOCPCS_{[ddo,cp,cs]}(O))$

```
   Input: Parameter: Boolean ddo, Boolean cp, Boolean cs, Arguments: Operator O
   Output: TupleSequence R
 1 begin
 2     TupleSequence S ← eval(O);
 3     if ddo then
 4         S ← ddo(S); // apply distinct document order to sequence S
 5     end
 6     if cp or cs then
 7         Integer i ← 0;
 8         for Tuple t ∈ S do
 9             i ← i + 1;
10             if cp then
11                 t ← t + i; // append current position to tuple t
12             end
13             if cs then
14                 t ← t + |S|; // append size of S to tuple t
15             end
16             R ← R + t;
17         end
18     else
19         R ← S;
20     end
21     return R;
22 end
```

With the discussion of the *TUPGEN*, *SELECT*, *SORT*, *PROJECT*, and *DDOCPCS* operators, we are nearly finished with the definition of the semantics of the XQGM select operator. However, one piece is still missing: the calculation of $|T|$, i. e., the calculation of the cardinality of a tuple variable. This information is for example required in the *set* function (Listing 4.3 on Page 62) or in the *enqueue* function of the tuple generator (Listing 4.1a on Page 60). Listing 4.8 presents the algorithm to calculate the tuple variable cardinality. Basically, the cardinality is the number of expressions in the projection specification of the input operator plus one additional tuple per positional context information.

Final Remarks on XQGM Select

You may have noticed that it took quite some effort to define the semantics of the XQGM select operator, which probably surprised you. However, when we take a look at the expressiveness of the operator, we see that it can already capture a large

Listing 4.8 The Cardinality of a tuple variable: $|T|$

```
   Input: Tuple Variable T
   Output: Integer c
 1 begin
 2     // get projection specification of input operator
 3     ProjectionSpecification P ← (T → operator.projectionSpecification);
 4     c ← |P.expression|; // add the number of expressions
 5     if P.cp? then
 6         c ← c + 1; // add one in case of context position
 7     end
 8     if P.cs? then
 9         c ← c + 1; // add one in case of context size
10     end
11     return c;
12 end
```

fraction of the XQuery language. We can translate *for* and *let* clauses, *existential* and *all* quantification, predicates, all kinds of simple expressions, sorting, and the provision of context information for correlated subqueries into this one operator.

4.2.5 XQGM Access

XQGM access operators retrieve XML nodes from a document. These nodes then serve as input for further XQGM operators. Therefore, access operators are always located at the bottom of an XQGM instance and are analogous to access operators in the relational QGM. We distinguish two types of access operators: 1) operators to retrieve the document node(s) (i. e., the virtual root node(s)) of a document or a document collection, in the following called *document access operators*, and 2) operators to evaluate a step expression, in the following called *node access operators*.

Document/Collection Access

Document/collection access operators are represented in XQGM by a box with a darker shaded color. The box contains an oval with the string "document" in case of a single document being accessed, or the string "collection" when multiple documents have to be delivered. For an example, see `access(5)` in Figure 2.4 on Page 20. As you can see, the operator contains a select operator with a projection specification and a simple string literal. The literal is a URI addressing the document(s) to be accessed. The operator does not contain any other XQGM components.

A document access operator A is mapped onto LAL operators as follows:

$$map(A) = \begin{cases} DOC_{[set(A.select.projectionSpecification.literal)]}() & : \quad A.document? \\ COLL_{[set(A.select.projectionSpecificaition.literal)]}() & : \quad A.collection? \end{cases}$$

This means that a document access operator is either translated to a LAL *DOC* operator or to a LAL *COLL* operator, depending on the type of access. As parameter, the literal contained in the select operator is passed (after being converted to a LAL expression by the *set* function).

Obviously, only allowing a select operator to define the URI literal restricts XQGM w. r. t. XQuery. In XQuery, the URI can be computed by a full featured nested ex-

Listing 4.9 $eval(DOC_{[U]}())$ and $eval(COLL_{[U]}())$

a) $eval(DOC_{[U]}())$

```
    Input: Parameter: String U, Arguments: none
    Output: TupleSequence R
 1  begin
 2  |  R ← R + DocumentNode(resolve(U)); // create new document node for resolved document
 3  |  return R;
 4  end
```

b) $eval(COLL_{[U]}())$

```
    Input: Parameter: String U, Arguments: none
    Output: TupleSequence R
 1  begin
 2  |  for Document D ∈ resolve(U) do
 3  |  |  DocumentNode n ← DocumentNode(D);
 4  |  |  R ← R + n;
 5  |  end
 6  |  return R;
 7  end
```

pression. The URI could even be computed from the contents of another XML document. XQGM does not support this feature, however, the modification to do so is straightforward: document and collection access methods could be handled like any other XQuery function, returning the requested document node(s). Therefore, handling document access operators as presented is not a "real" restriction. The decision for the current implementation emerged from the need to know the documents being accessed by a query *before* the query is evaluated. This is essential for plan generation.

As before, we present the evaluation function for the *DOC* and the *COLL* operators (see Listing 4.9). The algorithms depend on the *resolve* function which returns the (or all) document(s) for a given URI. From these documents, the required document nodes can then be constructed.

Node Access

Node access operators evaluate step expressions, i. e., an axis step without a predicate. Therefore, they specify an axis and a node test. In our example in Figure 2.4 on Page 20, you can find various access operators as leaves of the XQGM instance. The axis is specified in an oval component, whereas the node test is represented in a box attached with an arrow. Despite the arrow, we write $A.axis$ and $A.test$ in an access operator A to refer to these components. As you can also observe, every axis step needs a correlated input edge. This edge defines the context under which the access is evaluated, i. e., "where the navigation starts".

An access operator A is mapped to the LAL *STEP* operator as follows:

$$map(A) = SELECT_{[set(A.test)]}(STEP_{[A.axis, A \xrightarrow{p} tupvar]}())$$

This means that a *STEP* operator is parameterized by an axis and the tuple variable providing the correlated input. The operator simply returns the nodes reachable on the axis for a given correlated input node. The node test is then applied in the following *SELECT* operator. Essentially, the *STEP* operator is a tuple generator.

Listing 4.10 $eval(STEP_{[a,T]}())$

```
   Input: Parameters: Axis a, Tuple Variable T, Arguments: none
   Output: TupleSequence R
1  begin
2      Tuple t ← C(T); // read correlated input from context
3      if |t| ≠ 1 or t[1] is no node then
4          raise error("Singleton tuple containing node expected");
5      end
6      Node n ← t[1]; // get correlated node
7      return navigate(a, n); // navigate on axis a and return result nodes
8  end
```

In the following, we will not formally define how all nodes for a given axis are found or how a node test is evaluated. These semantics are sufficiently specified in the Formal Semantics and they can be "imported" here, similar to the semantics of simple expressions. However, we specify the algorithm for the *STEP* operator (see Listing 4.10).

As already mentioned, a node access operator is always a dependent subexpression with a correlated input edge. In the presented algorithm, the first action is to retrieve the current correlated input value from the context (remember, the *TUPGEN* operator analyzes its tuple variables to call them in the right order and to place a tuple into the dynamic evaluation context C). If the retrieved tuple actually contains a single node (checked in lines 3–6), the algorithm issues the *navigate* function with the desired axis and returns the resulting tuple sequence of nodes. In the following *SELECT* operator, the node test is then applied to these generated nodes.

4.2.6 XQGM Set Operators

XQuery knows three types of set operators: *union*, *intersect*, and *difference* (*except*). Likewise, XQGM contains a corresponding *SET* operator. In XQGM, this operator is a box that carries the type of set operation as name and only contains tuple variables (without further specification or predicate components). A set operator S is mapped as follows:

$$map(S) = \begin{cases} UNION_{[]}(map(T_1 \rightarrow operator_1), \ldots, map(T_n \rightarrow operator_n)) & : & S.union? \\ INTERSECT_{[]}(map(T_1 \rightarrow operator_1), map(T_2 \rightarrow operator_2)) & : & S.intersect? \\ EXCEPT[](map(T_1 \rightarrow operator_1), map(T_2 \rightarrow operator_2)) & : & S.except? \end{cases}$$

Depending on the type, a *UNION*, an *INTERSECT*, or an *EXCEPT* operator is generated. All these operators do not carry any parameters, and only the first one (union) can operate on more than two input arguments. For their definition, we again rely on the Formal Semantics. There, a set-based operator works on node sequences only (i. e., no sequences with atomic values are allowed). As a consequence, the results produced by input arguments in our logical algebra have to be tuple sequences, where each tuple is singleton and carries a node. In the following, we assume the functions *union*, *intersect*, and *except* to capture the XQuery semantics.

The UNION operator is evaluated as follows:

$$eval(UNION_{[]}(O_1, \ldots, O_n)) = union(union(\ldots(union(eval(O_1), eval(O_2)), \ldots, eval(O_n))))$$

Listing 4.11 $eval(TUPACCESS_{[T,E]}())$

```
   Input: Parameters: Tuple Variable T, LalTupleAccess E, Arguments: none
   Output: TupleSequence R
1  begin
2  |  Tuple t ← C(T); // read correlated input from context
3  |  Item i ← t[E.j]; // access tuple at position j
4  |  R ← R + i;
5  |  return R;
6  end
```

The n-way union is broken down to binary union operators, which are effectively evaluated using the above introduced *union* function. Similarly, we can define the evaluation of the other two operators

$$eval(INTERSECT_{[]}(O_1, \ldots, O_n)) = intersect(eval(O_1), eval(O_2))$$
$$eval(EXCEPT_{[]}(O_1, \ldots, O_n)) = except(eval(O_1), eval(O_2))$$

4.2.7 Tuple Variable References

As we have already seen during the discussion of the *SELECT* logical operator, tuple variable references (represented as rhombs) are used in expressions to project items out of tuples. During mapping they are translated to positional access operators (see Page 61) by the *map* function. This means, tuple variable references "act as expressions".

However, as the diagram in Figure 4.1 on Page 46 suggests, tuple variable references can also act as operators. Therefore, we have to define the *map* and *eval* functions on them. Let R be a tuple variable reference, then R can be mapped as:

$$map(R) = TUPACCESS_{[R \rightarrow T, set(R)]}()$$

A tuple variable reference is mapped onto a so-called *TUPACCESS* operator and two parameters, of which the first one is the tuple variable referenced and the second one is computed by the *set* function on the tuple variable reference. As shown in Listing 4.3, *set* returns a *LalTupleAccess* expression for a simple tuple variable reference. The information, which tuple variable is referenced, is essential to retrieve the current value of this tuple variable from the dynamic evaluation context, as shown in the *eval* function for the *TUPACCESS* operator in Listing 4.11. After the correlated value is fetched (line 2), the tuple access position encoded in the *LalTupleAccess* expression is used to retrieve the referenced item. This item is then written into a result sequence and returned.

4.2.8 The XQGM Root Operator

The root operator has a no further semantics in XQGM. It merely acts as a marker for the root of the XQGM graph and contains exactly one tuple variable, which in turn contains the XQGM instance. Therefore, during mapping, the call to the *map*

function is just passed to the contained XQGM operator: Let R be the root operator and let T be the contained tuple variable, then

$$map(R) = map(R.T \rightarrow operator)$$

A definition of the *eval* function is not required.

4.2.9 Final Remarks on the XQGM Semantics

XQuery is a Turing complete [Kepser 04] (and functional) language. What makes it Turing complete is the existence of (user-defined) recursive functions. Furthermore, the language also supports some features emerging rather from programming languages than from query languages, for example, the *if-then-else* statement, type casts and switch statements, recursive functions, and the like. The crucial question is now, which of the language constructs introduced by XQuery are expressible by XQGM and which are not. In this work, the set of these language constructs can be divided into three partitions. The first partition is completely supported by the XQGM semantics introduced above. These are, for example, FLWOR expressions, document and collection access, path expressions with predicates, filter expressions, range and sequence expressions, quantified expressions, function calls, node constructors, variable references, and Boolean/arithmetic/comparison expressions. The second partition of language constructs is yet not supported by the introduced XQGM semantics (and the support for them is not aspired in this work for reasons of brevity). However, these constructs are considered to be easily integrable as for example, type-based and schema-based expressions. The third group of language constructs does not easily fit with XQGM, because the underlying processing model is intrinsically different. These are, for example, user-defined recursive functions and the if-then-else statement. Because this work is primarily concerned with query processing aspects of XML management and not with programming language aspects, the omission of the third group of language constructs in XQGM can be tolerated. Its integration is left open for future work.

4.3 Query Translation

After having defined an internal representation for XML queries, the question now arises how queries in external representation can be translated into XQGM. In this section, an answer to this question is given. The responsible component of the query processor is the *translator*, which was introduced in the overview of Section 2.1.2 and in Figure 2.1 on Page 10. As presented, query translation is implemented in four stages: 1) syntactic sugar is removed from a query during *normalization* (resulting in a XQuery Core Language representation), 2) the static type is inferred during *static typing*, 3) subexpressions without effect are removed in *simplification*, and 4) an XQGM instance is created during *XQGM transformation*. All these stages operate on the abstract syntax tree generated by the parser (the first three stages rewriting the AST, while the last one transforms it). Note, because parsing and the creation of an AST are standard problems for which appropriate tools exist, we do not delve further into this topic. In the following, we will only briefly consider the first three stages, before the AST-to-XQGM translation is discussed in full detail.

4.3.1 Normalization and Static Typing

For XQuery, normalization and static typing are standardized and rigorously described in the Formal Semantics. Both stages have their right of existence: by removing syntactic sugar, normalization deflates the grammar, i. e., the number of language constructs required while still preserving the same expressive power. Operating on a reduced grammar obviously simplifies the following three stages. Static typing is beneficial, because 1) type errors can be detected at an early stage, and 2) the type information generated can be exploited in the simplification stage. With the Formal Semantics, it is simple to implement normalization and static typing, because the contained inference rules can be realized as a recursive function over the abstract syntax tree. Therefore, we do not discuss normalization and static typing in detail, but only summarize the major points:

1. Expressions that belong to the core language remain untouched. For example, these are literals, parenthesized expressions, function calls, etc.
2. Path expressions are normalized to cascades of *for/let* expressions. A path predicate (in square brackets) is normalized into an *if-then-else* expression. The resulting normalized expression makes the intuitive evaluation model for path expressions [Berglund 04] explicit.
3. Normalization ensures the correct binding of dynamic context variables *fs:dot* (context item), *fs:position* (context position), and *fs:last* (context size) in every *context switch*. Because a context switch takes place when a path step, a filter expression, or a predicate have to be evaluated, normalization hooks in at exactly these expressions.
4. When a step expression has a positional predicate (such as `//book/author[2]`) or an *fn:last()* function call, normalization makes the selection of the correct element in the filtered subsequence explicit. This happens by introducing an appropriately parameterized *fn:subsequence* function. Furthermore, if the step expression has a reverse axis, the reordering of the subsequence is also made explicit.
5. Some axes are expressed by means of other axes.
6. Abbreviated syntax is made explicit, i. e., "@", "..", and the omitted "child::" before axis steps are resolved/inserted.
7. Arithmetic/Boolean operations are mapped onto special internal functions, for example, a "+" is normalized to function *fs:plus*. Comparison operators with existential semantics are made explicit by using quantifiers and value comparisons. The arguments of these operators are wrapped into special conversion functions to ensure correct input types.
8. Direct node constructors are replaced with computed constructors.
9. FLWOR expressions with multiple variable bindings are normalized into cascades of *for/let* clauses (each *for/let* clause having only one variable binding). A *where* clause is replaced by an *if-then-else* expression. Likewise, quantified expressions with multiple bindings are normalized to cascades.

The Formal Semantics sometimes "over-normalizes" certain language constructs, for example, a simple *where* clause is normalized to an *if-then-else* construct. This is justified to simplify the definition of the XQuery semantics (which is the actual task of the document). For query evaluation, however, this "over-normalization" often inflates the query disproportionately by making many things explicit that

also could be handled implicitly with a much smaller query representation. For example, during the normalization of Boolean operators, the *fn:boolean* function is wrapped around the operator's arguments to ensure that they return a Boolean value. The interpretation of the argument's result as a Boolean value, however, can also be implicitly handled by the Boolean operator itself (thereby removing the need for *fn:boolean*). In this style, the normalization process deviates from the suggestions of the Formal Semantics. The following gives a brief overview over the changes to [Choi 07]:

- Functions *fn:convert-operand*, *fn:convert-simple-operand*, *fn:boolean*, and *fs:apply-ordering-mode* are not generated during normalization. The semantics of the first three functions is embedded into the appropriate operators, e. g., into Boolean/arithmetic/comparison operators. The fourth function is not generated, because we always assume an ordered query evaluation in this work.

- Function calls to *fn:position* and *fn:last* are translated to variable references *fs:position* and *fs:last*.

- No positional information (i. e., bindings to the XQuery internal context variables *fs:position* and *fs:last*) is generated for relative path expressions of the form `RelativePathExpr/StepExpr` and for axis-step or filter expressions of the forms `Axis::NodeTest[Predicates]` and `Expr[Predicates]`. Positional information is necessary to evaluate positional predicates, for example, as in `//book/author[2]`. Normally, the normalization process makes sure that the necessary positional information (in this case on the sequence of *book* nodes) is available. We drop this requirement and the necessary normalization overhead and implicitly assume that the query processor delivers positional information whenever required. Furthermore, we omit the explicit generation of reverse positional information and the selection of the item using *fn:subsequence* in case a numeric predicate or the *fn:last* positional predicate is applied to an axis step with reverse order.

- Path predicates are translated to *where* clauses instead of *if-then-else* expressions.

- *Where* clauses in FLWOR expressions are not normalized to *if-then-else* expressions.

- No axes are normalized into other axes.

- Function *fs:item-sequence-to-node-sequence* is omitted during the normalization of constructors. This functionality is embedded into the result generation process. Namespace attributes are not supported in constructors.

- Existential comparisons are not made explicit, i. e., no quantified expressions are generated during normalization.

- The usage of the *fs:** methods for comparison/arithmetics/Boolean expressions is omitted. Rather, these expressions remain untouched.

The described omission of certain normalization particles requires an adjustment in the static typing implementation, which is however straightforward and not shown here.

4.3.2 Simplification

A normalized query might still contain certain subexpressions that have no effect on the query result. For example, often the *ddo* function is applied to node sequences that are guaranteed to be duplicate-free and in the correct order. Such subexpressions might exist, because sometimes users introduce them, because queries are automatically composed by programs or because the normalization process injected them. Although not being harmful to the semantics of a query, they may have an impact on the evaluation performance. In the above example, sorting an already ordered list still has a best-case cost of $O(n)$, where n is the size of that list. Therefore, these redundant subexpressions should be removed, if possible.

Simplification is implemented by means of AST-based restructuring rules. Every such rule carries a *pattern* and a *transformation instruction*. A pattern can express structural and content-based predicates. Whenever a pattern matches the AST instance, the transformation instruction (which is a series of rewriting commands) is applied at the position of the match, thereby transforming the AST. In practical systems, the simplification component of a query processor can be seen as an "insurance" against badly designed (yet semantically correct) queries. The simplification component has to be able to detect a large range of simplification opportunities. In this work, the simplification component is merely a proof-of-concept implementation that can detect the following simplification opportunities:

- *fn:data*: This rule removes unnecessary calls to the function *fn:data*. This function implements the so-called *atomization*, where XML nodes are converted into atomic values. The normalization process frequently injects *fn:data* functions around arguments for comparison/arithmetic operations to ensure atomic values. When the input (to the function) is, however, already given as atomic values, atomization has no effect and is, therefore, not necessary. This circumstance is checked in the pattern of the rule, where the static type (previously deduced during the static typing stage) is checked for any atomic type. When the pattern matches, the transformation instruction then exchanges the *fn:data* function with its argument expression (note, *fn:data* has always exactly one argument).

- *for-at*: XQuery allows to specify a loop counter in *for* loops. The loop counter is defined with the *at* keyword and, during evaluation, binds the current recurrence value to a variable. The variable can be referenced in the return expression of the *for* statement, for example: `for $i at $p in (1, 2, 3) return $i * $p`. If the variable for the loop counter is not referenced in the return expression, it can be removed.

- *let*: A *let* clause binds the sequence-valued result of an expression to a variable. If this variable is referenced once or not at all and if the *let* clause has no other tasks (like a *where* clause or an *order by* clause), the *let* clause is not necessary. In case the variable is referenced once, the variable reference can be replaced by the *let* expression, e. g., in `let $p := (1, 2, 3) return min($p)` is equal to `min((1, 2, 3))`. If the defined variable is not referenced at all, the complete *let* clause can be replaced by the *return* expression.

- *order by*: An *order by* clause sorts a tuple stream generated by a cascade of *for*/*let* clauses. It can consist of one or more order specifications which, in turn, contain single XQuery expressions. These expressions are evaluated over the input tuple stream, where the link between the expression and the tuple stream is established

by variable references. Thus, if an order specification does not contain a variable reference, it can safely be removed without altering query semantics.

- *some/every*: Similar to *for* and *let*, XQuery quantifications specify a variable binding. If the defined variable is not contained in the *satisfies* expression of the quantification, then the variable binding can be removed.

- *typeswitch*: A typeswitch expression has a *switch* expression, multiple *case* clauses, and a *default* clause. *Case* clauses and the *default* clause contain a variable binding and *switch* expressions. Due to static typing, the type of the *switch* expression is known in advance. If it is possible to uniquely identify the clause that will always be chosen by the *typeswitch* during evaluation, there is an opportunity to remove the *typeswitch*. This is exactly the case, when either the *switch* type is a subtype of (is equal to) exactly one of the types defined among the clauses, or if none of the *case* clauses matches (then, the *default* clause is activated). As a second restriction, the *case* expression may only have exactly one reference to the variable defined by the *case* clause. If a clause has been identified as the winner, the variable reference in the case expression is substituted with the *switch* expression, resulting in an intermediate expression E. Then the complete *typeswitch* is substituted with E.

The normalized, typed, and simplified query in AST representation is now ready to be transformed into an XQGM instance.

4.3.3 XQGM Transformation

XQGM transformation is the process of translating the AST representation generated by the parser to the XQGM representation, whose syntax and semantics was introduced in the previous sections. After a query is transformed, the query processor discards the AST representation and, further on, solely works on the generated XQGM instance. XQGM transformation is defined by means of a recursive function over the AST. Every time the function is applied, a specific part of XQGM is constructed. To define the process, we follow the (syntactic) style of the normalization process defined in the Formal Semantics. Specifically, we will show the transformation rules for *for*/*let* clauses, step expressions, variable references, plain expressions and function calls as well as quantified expressions and *set* expressions. However, before, we have to clarify the meaning of the *translation environment* and of some *notations*. The chapter will conclude with two examples illustrating the transformation process.

The XQGM Transformation Environment

The *XQGM transformation environment* is a context under which the AST-to-XQGM mapping takes place. In the following, we refer to this context as *transEnv* (similar to the *statEnv* and *dynEnv* in the Formal Semantics). The *transEnv* context has various context components that can be addressed using a *dot* notation, e. g., the *transEnv.posVars* component keeps track of the positional variables defined in a query. Components can be read and modified. Internally, they organize the data they store in different ways: 1) as plain atomic values, 2) as lists, or 3) as a map. The notations to read and write data from/to context components will be introduced below. The following components exist:

- Component *transEnv.cp()* stores a Boolean value that signals the requirement to generate the *context position* (cp) information within the projection specifi-

cation. The notation to assign a value to this component looks as follows: *transEnv.cs()←true*.

- Component *transEnv.cs()* stores a Boolean value that signals the requirement to generate the *context size* (cs) information. The notation is analogous to the one above.
- Component *transEnv.ddo()* is also a Boolean and signals the generation of a *ddo* output modifier within the projection specification.
- Component *transEnv.posVars()* is a list of strings that keeps track of positional variables found in a query. A new list entry can be appended with the + notation, as in *transEnv.posVars() + "VarName"*. Notation *transEnv.posVars().contains("VarName")* returns true, if the list contains the given string, and false otherwise.
- Component *transEnv.vars()* maps from a string to a tuple variable. The mapping keeps track of XQuery variable names and the tuple variables they are logically assigned to during the translation. To add a mapping, the notation *transEnv.posVars("VarName") + T* can be used, where T is a tuple variable.

At the beginning of the transformation process, the mapping is empty having undefined components. The first assignment to a component also defines it. Sometimes it is necessary to memorize and re-establish the state of the transformation environment. Therefore, we define a function *memento* which copies the complete environment (with all components).

Transformation Notations

Besides the above introduced notations to modify the transformation environment, some means to define the recursive transformation function are required. As already stated, the function operates on the AST representation. To present the transformation here, we consider the AST as a string. Over this string, we try to match certain patterns consisting of terminals and non-terminals from the XQuery grammar. Thereby, a non-terminal is used as a placeholder to denote any subexpression of a certain type (i. e., an AST subtree), which we do not want to qualify in more detail. Non-terminals are written using a slanted font shape. For example, the following object may occur in one of the following mapping rules: $Expr_1$ + $Expr_2$. Here, the subscripts are used to distinguish two non-terminals of the same type. The pattern essentially refers to an additive expression (terminal), whose arguments (non-terminal) are two general XQuery expressions. We thereby adopt the definition of *patterns* from the Formal Semantics.

Similar to normalization rules, a transformation rule has the following form:

$$transEnv \vdash [\text{Object}]^{Role}_{Arguments}$$

$$==$$

$$\text{Mapped Object}$$

This rule reads as follows: Given the current transformation environment *transEnv*, an *Object* (for example, a *for* clause) is transformed to a *Mapped Object*, where the transformation is executed under a *Role* and is parameterized by the given *Arguments*. The application of the rule, given a certain environment, is notated using the *turnstile* (⊢). Because this environment is always assumed, the prefix "*transEnv*⊢"

will be omitted in the following. The object to be mapped is described by the patterns from above. Whenever a pattern matches a query (AST), the rule is applied. Sometimes, transformations require arguments. These are passed as subscripts to the square brackets. Furthermore, one and the same pattern may be transformed with different intentions. These intentions are signaled by the *Role* superscript, which may also be empty (for the default transformation). Finally, the *Mapped Object* is an XQGM construction specification. The construction specification is given by means of an algorithm in pseudocode similar to the specifications in the XQGM semantics section.

In the construction specification, the XQGM types introduced in Figure 4.1 on Page 46 can be instantiated in two ways: either by a constructor-like notation, such as *SelectOperator()*, or by using the dot/arrow notation together with a pair of braces where the arguments go. For example, let S be a select operator, then $S.ProjectionSpecification()$ creates (and directly adds) its projection specification, if it not already exists. Note, we do not formally define, which component receives which arguments during instantiation. The construction specification should be sufficiently descriptive in this point. A component can be added to another component using a $+$. For example, let S be a select operator and let T be a tuple variable, then $S + T$ adds the tuple variable to the select operator. With these preliminaries, the XQGM transformation rules can now be specified.

For/Let Clauses

A *for/let* clause is translated into a XQGM select operator with at least two tuple variables and a projection specification. The first tuple variable contains the binding expression, the second one contains the return expression. The projection specification, in turn, references one of the two tuple variables to generate some output. Further tuple variables can be necessary for the transformation of the *where* and the *order by* clauses. The following rule defines the construction process:

$$\left[\begin{array}{l}\textbf{for } \$\mathrm{VarName}_1 \textbf{ at } \$\mathrm{VarName}_2 \textbf{ in } \mathrm{Expr}_1\\ \textbf{where } \mathrm{Expr}_2\\ \textbf{order by } \mathrm{Spec}_1, \ldots, \mathrm{Spec}_n\\ \textbf{return } \mathrm{Expr}_3\end{array}\right] \qquad \text{(T1)}$$

$$==$$

```
1  begin
2      Environment C ← memento(transEnv); // copy the context
3      Operator S ← SelectOperator(); // create a select operator
4      Operator B ← [$VarName1 at $VarName2 in Expr1]_S^binding; // translate binding expression
5      TupleVariable T ← S.TupleVariable("F", B); // create for-quantified tuple variable with input B
6      transEnv.vars(VarName1) ← T; // register tuple variable in context
7      if defined(VarName2) then
8          transEnv.vars(VarName2) ← T; // register tuple variable in context
9          transEnv.posVars() + VarName2; // store positional variable
10     end
11     if defined(Expr2) then
12         S ← [Expr2]_S^where; // transform where clause
13     end
14     if defined(Spec1) then
15         S ← [Spec1, ..., Specn]_S^order by; // transform order by
16     end
17     S ← [Expr3]_S^return; // transform return clause
18     transEnv ← memento(C); // re-establish context
19     return S;
20 end
```

Let us take a look at the construction specification: During transformation, the environment may be altered while the subexpression of the *for/let* expression is transformed. Afterwards, however, the transformation environment has to yield the same state as before. Therefore, the current state of the environment is memorized in variable C. Then, for each *for/let* expression, a select operator is created. At this state, the select operator is nothing but an empty box that will be filled during the transformation of the subexpressions. To do so, the "box" is passed as an argument to these transformation calls. The first subexpression is the *binding* expression, which returns an operator upon transformation. Note, the *role* of this transformation is set to "*binding*", and the argument is the select operator S. The returned operator is "wrapped" inside a tuple variable T having a *for* quantification. The mapping between the binding variable name (VarName_1) and the tuple variable is kept in the environment. If a positional variable is defined, the mapping for this variable to the same tuple variable (which produces the context information) is kept in the environment. The occurrence of a positional variable is recorded in *transEnv.posVars*. These mappings are required for the transformation of correlated subexpressions (referencing tuple variables and access operators). After the context is written, the subexpression can been transformed. Note, the *where* clause and the *order by* clause are not necessarily specified. As mentioned above, the select operator is passed as an argument. The transformation of the subexpressions triggers the insertion of more tuple variables and specifications to S, if necessary. After the *return* clause has been transformed, the environment is re-established and the select operator is returned.

Above, we stated that after creation, the select operator is "nothing but an empty box". This is not exactly true, because during the instantiation of *any* operator some environment-dependent initializations take place. For the sake of simplicity however, these issues are discussed initially here. The following construction algorithm instantiates an operator S. The algorithm is executed every time an operator (except a tuple variable reference) is created, for example, via *SelectOperator()*, *AccessOperator()*, and the like.

```
1  begin
2      ProjectionSpecification P ← S.ProjectionSpecification();
3      if transEnv.cp() then
4          P.cp ← true;
5          transEnv.cp() ← false;
6      end
7      if transEnv.cs() then
8          P.cs ← true;
9          transEnv.cs() ← false;
10     end
11     if transEnv.ddo() then
12         P.ddo ← true;
13         transEnv.ddo() ← false;
14     end
15 end
```

In the first step, a projection specification is created (if it not already exists). Because tuple variable references do not carry a projection specification, this algorithm is not executed for them. In the remaining lines, the context position ("cp"), context size ("cs"), and distinct document order ("ddo") information is read from the environment and, if set to true, passed on as an output modifier to the projection specification. Note, this information may have been set during previous translation

rules (as we will see in the following).

After the translation of the *for* clause, we proceed with the subexpressions. A *binding* expression is transformed as follows:

$$[\$\mathrm{VarName}_1 \textbf{ at } \$\mathrm{VarName}_2 \textbf{ in } \mathrm{Expr}_1]^{binding}_{SelectOperator\ S} \quad (T2)$$

$$==$$

```
1  begin
2  |  if defined(VarName2) then
3  |  |  statEnv.cp() ← true; // make context position available
4  |  end
5  |  Operator B ← [Expr1]; // transform the binding expression
6  |  if defined(VarName2) then
7  |  |  statEnv.cp() ← false; // restore previous state
8  |  end
9  |  return B;
10 end
```

If the binding expression contains a positional variable ($\mathrm{VarName}_2$), this circumstance is recorded in the *cp* component of the environment. Then the subexpression itself is transformed into an XQGM operator (embedding the results in the passed select operator S). As before, the state of the environment has to be re-established. We do so by assigning the value *false* to the *cp* component. Finally, the operator is returned. Remember that this operator will be wrapped inside a *for*-quantified tuple variable.

A *where* clause results in an XQGM predicate. This is shown by the following transformation rule:

$$[\mathrm{Expr}]^{where}_{SelectOperator\ S} \quad (T3)$$

$$==$$

```
1  begin
2  |  Operator P ← [Expr]S; // transform where expression
3  |  S ← S.Predicate(P); // set P as predicate into select
4  |  return S;
5  end
```

The transformed expression is set as a predicate into operator S.

An *order by* clause results in a new sorting specification inside select operator S and is transformed as follows:

$$[\mathrm{Spec}_1, \ldots, \mathrm{Spec}_n]^{order\ by}_{SelectOperator\ S} \quad (T4)$$

$$==$$

```
1  begin
2  |  SortingSpecification X ← S.SortingSpecification(); // create the sorting specification
3  |  foreach OrderSpec Spec_i in Spec_1, ..., Spec_n do
4  |  |  Operator O ← [Spec_i]S;
5  |  |  X + O; // add the transformed operator to the sorting specification
6  |  |  X.modifier_i ← Spec_i.modifier // set the order modifier
7  |  end
8  |  return S;
9  end
```

First, a sorting specification is created inside the select operator. Then, all the order specs are transformed and added step-by-step to the specification. Modifiers, such as "ascendent", "descendant", "empty greatest", etc., are also written into the sorting specification.

Finally, a *return* clause creates the projection specification inside the select operator:

$$[\text{Expr}]^{return}_{SelectOperator\ S} \quad (T5)$$
$$==$$

```
1 begin
2 |   ProjectionSpecification P ← S.ProjectionSpecification();
3 |   Operator R ← [Expr]_S;
4 |   P + R; // add the return operator to the projection specification
5 |   return S;
6 end
```

As you may have observed, the translation of an expression given a role, as for example $[Expr]^{return}_{SelectOperator\ S}$ contains a recursive call to the transformation function without a role, but passing parameter S: $[Expr]_S$. What happens now, if the expression to be transformed is a *for* clause? Then, the above rule (without argument S) does not apply. We have to define the correct rule explicitly. Let Expr_0 be a *for* clause as shown in rule T1, then the following transformation applies:

$$[\text{Expr}_0]_{SelectOperator\ S} \quad (T6)$$
$$==$$

```
1 begin
2 |   Operator O ← [Expr_0];
3 |   TupleVariable T ← S.TupleVariable("L", O);
4 |   return TupleVariableRef(T);
5 end
```

The *for* clause is translated as before, but the result is wrapped inside a tuple variable which, in turn, is embedded into operator S. Then, a reference to this tuple variable is returned to make the result of the translated *for* expression available. Note, the quantifier of the tuple variable is set to L, despite of the *for* semantics of the expression. The rationale is that the *for* semantics is assured by the transformation during $[Expr_0]$ (i. e., in the select operator "below").

The *let* clause is treated exactly like to *for* clause with two exceptions: 1. instead of the F-quantification, an L-quantification is used, and 2. because a *let* clause cannot define a positional variable (using *at*), the corresponding part is not required. We do not separately enlist the transformation rules for *let* here.

Step Expressions

For step expressions, we need the following two transformation rules:

$$[\text{Axis :: NodeTest}] \quad == \tag{T7}$$

```
1 begin
2 |   TupleVariable T ← transEnv.vars("fs:dot");
3 |   AccessOperator O ← AccessOperator(Axis, NodeTest, T);
4 |   return O;
5 end
```

$$[\text{Axis :: NodeTest}]_{SelectOperator\ S} \quad == \tag{T8}$$

```
1 begin
2 |   AccessOperator O ← [Axis :: NodeTest];
3 |   TupleVariable T ← S.TupleVariable("L", O);
4 |   return TupleVariableRef(T);
5 end
```

If the expression is translated without an argument, the tuple variable delivering the input for *fs:dot* is first retrieved from the environment. Note, the normalization process makes sure that this variable is always correctly bound and the translation process makes sure that there is a tuple variable already defined for this variable name (see Rule T1, Line 6). The returned tuple variable belongs to some select operator previously created. It serves as the correlated input for the access operator, which is newly created here.

If the expression is translated "under" some given select operator S (second case), the transformation actions from the first rule are first executed. Then a *let*-quantified tuple variable is added to S receiving the input of the step expression. The result is then a tuple variable reference to actually access this input.

Variable References

Variable references can occur almost everywhere in an XQuery expression. During translation, they are mapped onto tuple variable references by the two rules in the following:

$$[\$\text{VarName}] \quad == \tag{T9}$$

```
1 begin
2 |   TupleVariable T ← transEnv.vars(VarName);
3 |   if defined(transEnv.posVars(VarName)) then
4 |   |   return TupleVariableRef(T, "cp");
5 |   end
6 |   return TupleVariableRef(T);
7 end
```

In the first case, a variable reference is translated independently, i. e., without an operator as an argument. First, the translation environment is queried to return the tuple variable, whose input returns the referenced values. This tuple variable has to

be defined by rule T1 before. There are two possibilities: the variable is a positional variable (generated for some *at* construct in a *for* expression) or not. In the first case (lines 3 to 5), a special tuple variable reference addressing the context position ("cp") has to be instantiated. In the latter case, an ordinary reference is sufficient. Note, addressing the context position is only possible, because of the interplay of rules T1 (which puts the variable name into *transEnv.posVars()*), T2 (which manifests the need to produce positional information in the environment), the initialization algorithm for operators (which actually configures the projection specification to produce positional information), and rule T9.

If a variable reference is translated under an operator S, the first action is also the retrieval of the responsible tuple variable from the environment. However, in this case, the situation could occur that the variable is a positional one, i. e., *fs:position* or *fs:last*. In the XQuery normalization process, these variables are properly defined. However, because we chose to handle them implicitly by the omission to generate them (as described in Section 4.3.1), they are not bound and the "responsible" tuple variable, therefore, is *undefined*.

$$[\$\text{VarName}]_{SelectOperator\ S} \quad (\text{T10})$$
$$==$$

```
1  begin
2      TupleVariable T ← transEnv.vars(VarName);
3      if ¬ defined(T) then
4          TupleVariable F ← S.TupleVariable[1];
5          ProjectionSpecification O ← (F → Operator.ProjectionSpecification());
6          if VarName = "fs:position" then
7              P.cp ← true;
8              return TupleVariableRef(F, "cp");
9          end
10         if VarName = "fs:last" then
11             P.cs ← true;
12             return TupleVariableRef(F, "cs");
13         end
14     end
15     if T.Operator ≠ S then
16         TupleVariableRef R ← TupleVariableRef(T);
17         TupleVariable N ← S.TupleVariable("L", R);
18         return TupleVariableRef(N);
19     end
20     return TupleVariableRef(T);
21 end
```

Undefined tuple variables are no problem. We simply have to make sure that the required positional context information is actually generated. This happens in lines 3 to 14. In line 4, the first[4] tuple variable of the passed select operator is retrieved and then (in line 5) the projection specification P of its input operator. If this specification does not exist, it is created. Depending on the variable name (only *fs:position* and *fs:last* are possible), the *cp* or *cs* output modifier is set in P. Then, a tuple variable reference, either addressing "cp" or "cs", is returned. A further case arises, when the referenced tuple variable is not contained in operator S passed as the argument (i. e., when $T.Operator \neq S$, as checked in line 15). Then, the tuple variable reference has to be embedded (using tuple variable) into S, i. e., the tuple variable reference is treated as an ordinary XQGM operator, similar to the access operator in

[4]Note, here the brackets are used to choose the first component, and not as an indicator for transformation.

rule T8. Finally, in all other cases (i. e., when the tuple variable referenced belongs to S), a simple tuple variable reference can be returned.

Simple Expressions

Under the name "simple expression", all XQuery expressions are subsumed that do not produce any variable bindings and that are no set-based expressions, such as Boolean/arithmetic/comparison expressions, constructors, literals, functions, and sequence expressions. For them, a corresponding concept exists in XQGM, as you can observe in Figure 4.1 on Page 46. Technically, we do not transform these expressions into "something else", but keep them as they are: as XQuery expressions. This means that they are just imported into XQGM. The following two rules implement these considerations:

$$[\text{SExpr}] \quad (\text{T11})$$
$$==$$

```
1 begin
2 |   Operator S ← SelectOperator();
3 |   foreach Operand O in operands(SExpr) do
4 |   |   O ← [O]_S;
5 |   end
6 |   ProjectionSpecification P ← S.ProjectionSpecification();
7 |   P ← P + SExpr; // add SExpr as output expression
8 |   return S;
9 end
```

We do not want to provide rules for every type of expression. Instead, we take a generic view, where a simple expression (referenced as *SExpr*) has a number of operands that are, in turn, simple expressions or variable references. These operands are revealed by the *operands* function. With these preliminaries, the above rule reads as follows: first, a select operator is created. This is necessary, because, obviously, the object to be transformed is a standalone expression, such as 1 + 3. As stated above, an expression cannot stand alone in XQGM, which is why the select operatore is required. In the second step, the operands of the simple expression are transformed (passing the new select operator as an argument). Finally, the transformed expression is written into the newly created projection specification of the select operator, which is then returned.

In case an operator S is already given (e. g., as above, or when an expression is translated for example to an XQGM predicate), the transformation is even simpler. You can see that the expression itself is not transformed:

$$[\text{SExpr}]_{SelectOperator\ S} \quad (\text{T12})$$
$$==$$

```
1 begin
2 |   foreach Operand O in operands(SExpr) do
3 |   |   O ← [O]_S;
4 |   end
5 |   return SExpr;
6 end
```

Function Calls

So far, we classified function calls as simple expressions. However, the transformation of functions *doc*, *collection*, and *ddo* is specialized, which is why a separate rule is required. For the following rule, we do not distinguish between the version with and without the S subscript; if the subscript is missing, it is simply not passed in subsequent recursive calls. The rule has the following form:

$$[\text{FunctionCall}]_{SelectOperatorS} \quad (\text{T13})$$
$$==$$

```
1  begin
2      if FunctionCall is fn:doc then
3          Argument X ← first(arguments(FunctionCall)); // get the argument
4          DocumentAccessOperator A ← DocumentAccessOperator([X]);
5          return A;
6      end
7      if FunctionCall is fn:collection then
8          Argument X ← first(arguments(FunctionCall)); // get the argument
9          CollectionAccessOperator A ← CollectionAccessOperator([X]);
10         return A;
11     end
12     if FunctionCall is fn:ddo then
13         transEnv.ddo() ← true;
14         return [first(arguments(FunctionCall))]_S;
15     end
16     // translate as SExpr
17 end
```

If the function call is a *doc* or *collection* function, the first argument is retrieved (which is a literal containing the URI of the document or collection), transformed, and passed to the constructor of a document or collection access operator. This operator is then returned. In case, the function call is a *ddo* function, the corresponding environment information *transEnv.ddo()* is set and the first argument is transformed and returned. In all other cases, the function call is transformed as explained above in rule T11 and T12.

Set Expressions

Set expression combine several sequences produced by their subexpressions into one. XQuery set expressions are transformed into corresponding XQGM set expressions. The transformation process is quite straightforward:

$$[\text{SetExpr}] \quad (\text{T14})$$
$$==$$

```
1  begin
2      SetOperator S ← SetOperator(typeOf(SetExpr));
3      foreach Operand X in operands(SExpr) do
4          TupleVariable T ← S.TupleVariable([X], "L");
5      end
6      return S;
7  end
```

We assume a generic set expression *SetExpr*, where the type of the expression (i. e., *union*, *intersect*, *except*) can be deduced using the *typeOf* function. The subexpres-

sions are separately transformed and attached to the set expressions using *let*-quantified tuple variables. In case, an operator is passed, the transformation takes the same form as rule T6 for *for* expressions. Therefore, we do not repeat this transformation rule.

Quantified Expressions

The last expression type we need to transform are quantified expressions. In a sense, quantified expressions are similar to *for*/*let* expressions, because they also bind variables. Therefore, their transformation looks alike.

$$[\text{some } \$\text{VarName in } \text{Expr}_1 \text{ satisfies } \text{Expr}_2] \quad \text{(T15)}$$
$$==$$

```
1  begin
2      Environment C ← memento(transEnv);
3      Operator S1 ← SelectOperator();
4      Operator B ← [$VarName in Expr1]_{S1}^{binding};
5      TupleVariable T1 ← S1.TupleVariable("F", B);
6      transEnv(VarName) ← T1;
7      Operator X ← [Expr2];
8      TupleVariable T2 ← S1.TupleVariable("E", X);
9      S ← S1.Predicate(TupleVariableRef(T2));
10     ProjectionSpecification P1 ← S1.ProjectionSpecification();
11     P1 ← P1 + BooleanLiteral(true);
12     Operator S2 ← SelectOperator();
13     TupleVariable T3 ← S2.TupleVariable("L", S1);
14     ProjectionSpecification P2 ← S2.ProjectionSpecification();
15     P2 ← P2 + fn : boolean(TupleVariableRef(T3));
16     transEnv ← memento(C);
17     return S2;
18 end
```

The steps in lines 1 to 6 are very similar to the corresponding lines in rule T1 (only the variable names are somewhat different): The rule saves the environment, creates a select operator, transforms the binding expression, generates a tuple variable for the result, embeds the tuple variable into the select operator, and registers the variable name with the tuple variable in the environment. Then the *satisfies* expression is transformed and a second tuple variable T_2 is embedded into S_1. The quantification mode of T_2 depends on the type of quantification. Rule T15 is responsible for the *exists* quantification, therefore the mode is "E". We do not explicitly show the algorithm for the *all* quantification, because its structure is the same, with the only difference that T_2's mode is "A" for *all*. A predicate, which is inserted into S_1 at line 9 is responsible to check the Boolean value generated by T_2 during execution. This predicate is a simple tuple variable reference.

A *satisfies* expression returns a Boolean value, therefore, the corresponding XQGM expression also has to return a Boolean value. To meet this requirement, a simple "static" Boolean literal with the value *true* is given to select operator S_1 as projection specification (lines 10 and 11). Now, with this translation, the *satisfies* expression either returns the value *true* or the empty sequence. The latter is the case when the predicate set in line 9 never evaluates to *true*. Therefore, a second select operator S_2 is necessary, which is simply wrapped around S_1 using a *let*-bound tuple variable and an *fn:boolean* function as projection specification. This is accomplished in lines 12 to 15. Finally, the context is restored and the second (outer) select operator is

returned. The rule for quantified expressions "under" a select operator, i. e., when S is passed has the same form as T6. Therefore, we do not repeat it here.

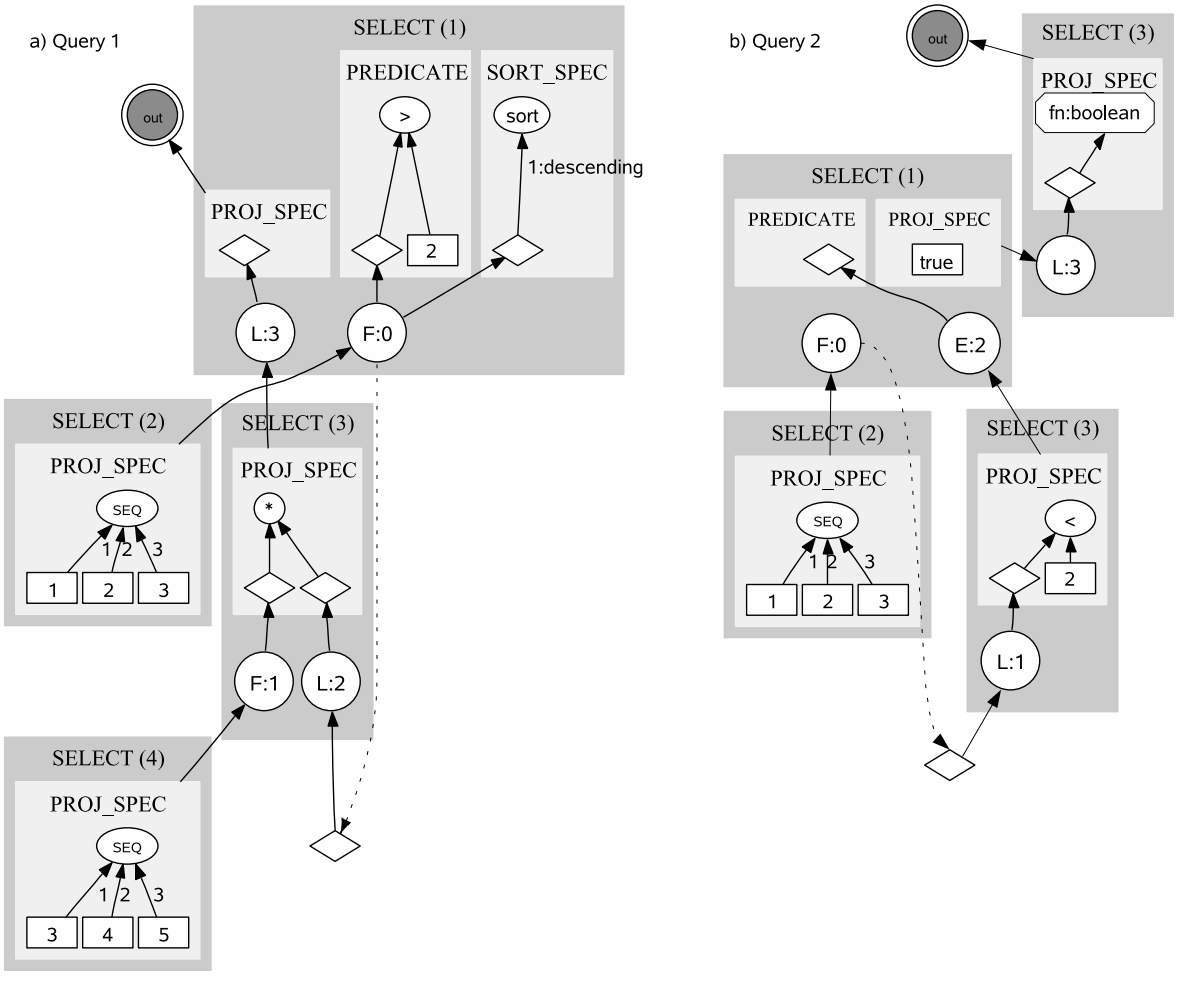
Examples

To facilitate the comprehension of the transformation process, we look at two simple examples. The first one is dedicated to the *for* clause; the second one is a quantified expression to illustrate rule T15. For each example, we provide the query, the XQGM result, and a short description of the rules that have been applied. Consider query

```
for $i in (1, 2, 3)
where $i > 2
order by $i descending
return
  for $j in (4, 5, 6)
  return $i * $j
```

The result of this rather synthetic query is `<12, 15, 18>`. The corresponding XQGM instance is shown in Figure 4.7a. The overview in Listing 4.12 at the end

Figure 4.7 AST-to-XQGM transformation of two sample queries



of this chapter "expands" the call structure of the recursive transformation process, where each call to a transformation rule is represented by a *begin*/*end* block, thus forming a tree structure. Behind each `begin` keyword, the called rule and a short version of the pattern on which the transformation takes place is indicated. Some simple transformations are not shown, such as the transformation of literals. You can trace the creation process by identifying the corresponding actions in the above introduced transformation rules.

For query `some $i in (1, 2, 3) satisfies ($i < 2)`, we do not present a detailed overview, because mainly rule T15 is involved. The correspondences between the query, the XQGM in Figure 4.7b, and the rule should be obvious.

4.4 Related Work

For relational query processing, a commonly agreed-on algebra exists, i. e., the relational algebra. This is, even after a decade of research on the topic, not true for XML query processing. One of the first attempts to develop an XML algebra was to position the Formal Semantics as one [Fernández 00]. Because the Formal Semantics rather describes a functional language and not an algebra, the proposal was not accepted *as an algebra* by the research community (however, it was accepted as the formal semantics of the language). After this first attempt, two major approaches emerged: *tuple-based* algebras and *tree-based* algebras. Tuple-based algebras are superior in number. The main reason is probably that they are (extensions of) the relational algebra and, therefore, easily integrate with abundant relational query processors. On the other side, the tree-based approach provides operators that manipulate trees (instead of tuples). Therefore, the tree-based approach follows more the style of XML.

In the related work section of the next chapter, we will consider the logical query representations of the five systems introduced in Section 2.3 in more detail. We will see that DB2, Natix, and MonetDB/XQuery employ tuple-based algebras (as XTC does). Timber has a tree-based algebra, and Galax contains operators of both types, i. e., tuple operators and tree operators. We will discuss these approaches in the next chapter, because there we can also directly summarize the various rewriting strategies suggested.

4.5 Summary

In this section, we have seen how XQuery expression can be transformed into an internal XML query representation called XML Query Graph Model. First, we introduced the XQGM syntax. The semantics was defined indirectly by a mapping to a logical algebra (LAL). For every XQGM operator, we have seen the transformation to its corresponding LAL expression and how the semantics of this expression was defined by several algorithms. Then we described the normalization, static typing, and simplification stages, followed by the necessary transformation rules to map an XQuery (represented as an abstract syntax tree) into XQGM. With the 15 rules shown, it is possible to transform a large part of the XQuery language into XQGM, including FLWOR expressions, path expressions, functions, literals, quanti-

fied expressions, etc. Missing features are user-defined functions, type expressions, the if-then-else statement, and everything concerning the so-called prologue, e. g., module definitions, schema imports, etc. These missing features are more or less programming language constructs, which is why we leave their integration into XQGM to future work.

Regarding the XQGM semantics introduced in Section 4.2, one issue is still open: on Page 57, we demanded that 1) dependent tuple variables of one and the same select operator S shall depend on one and the same tuple variable T, which 2) has to be the first tuple variable in S, and which is *for*-quantified or *let*-quantified. This restriction simplified the definition of the *TUPGEN* operator which is responsible to generate a stream of tuples out of the input operators of a selection. By an examination of all 15 transformation rules, you can observe that 1) a correlated operator has to access *transEnv.vars()* to find the tuple variable providing the correlated input; 2) the tuple variables written into *transEnv.vars()* are either *for*-quantified or *let*-quantified; and, 3) the tuple variables written into *transEnv.vars()* are the first tuple variables generated for an operator. These four points guarantee that the first tuple variable T in an operator S is referenced and that T is *for*-quantified or *let*-quantified (i. e., the second requirement from above is fulfilled). The first requirement follows because, according to these observations, only one tuple variable per operator can have correlated edges.

In the next chapter, we will see how queries can be rewritten from their internal representation into a pre-optimized alternative. This alternative will be the starting point for the plan generator to assemble a query evaluation plan. During rewriting, query unnesting and twig discovery are of major concern.

Listing 4.12 Call-structure expansion of an AST-to-XQGM transformation run

```
begin T1: [for . . . ]
    1. store environment (currently empty)
    2. create SELECT 1 (named S1 in the following) and translate binding expression → T2:
    begin T2: [$i in . . . ]^binding_S1
        1. transform sequence expression as simple expression → T11
        begin T11: [1, 2, 3]
            1. create SELECT 2 (named S2 in the following)
            2. transform each operand (not shown, because operands are simple literals)
            3. create the projection specification of SELECT 2
            4. add transformed sequence expression to projection specification and return S2
        end
        2. return S2
    end
    4. create for-quantified tuple variable F:0 with S2 as input and register F:0 for $i in environment
    7. translate where expression → T3
    begin T3: [where . . . ]^where_S1
        1. transform comparison expression as simple expression → T12
        begin T12: [$i > 2]_S1
            1. transform each operand; transformation of variable reference $i → T10
            begin T10: [$i]_S1
                1. get tuple variable for $i from environment: F:0
                2. tuple variable is defined and operator of tuple variable (SELECT 1) equals S1
                3. therefore, just return tuple variable reference to F:0
            end
            2. return comparison expression
        end
        2. set comparison expression as predicate into S1
        3. because the returned expression is no tuple variable reference: return S1
    end
    8. transform order by → T4
    begin T4: [order by . . . ]_S1
        1. create the sorting specification in S1
        2. transform the order specification, which is just a variable reference: [$i]_S1 (not shown)
        3. add returned tuple variable reference to sorting specification
        4. set the 'descendant' modifier and return S1
    end
    9. transform return → T5
    begin T5: [for . . . ]^return_S1
        1. create projection specification in S1 and transform return expression "under" S1
        begin T6: [for . . . ]_S1
            1. transform for clause without S1 as argument → T1
            begin T1: [for . . . ]^return_S1
                1. store environment (currently maps $i to tuple variable F:0)
                2. create SELECT 3 (named S3 in the following)
                3. transform the binding expression to SELECT 4 as above (the result is named S4)
                4. create for-quantified tuple variable F:1 with S4 as input
                5. register F:1 for variable name $j in environment
                9. transform return as simple expression → T5
                begin T5: [$i * $j]^return_S3
                    1. create projection specification in S3
                    2. transform return expression "under" S3
                    begin T12: [$i * $j]_S3
                        1. transform $i → T10
                        begin T10: [$i]_S2
                            1. get tuple variable for $i from environment: F:0 (which is defined)
                            3. operator of tuple variable (S1) does not equal S2
                            4. create tuple variable reference R to F:0
                            5. create let-quantified tuple variable L:2, add R, and return R
                        end
                        2. transform $j → T10 (not shown) and return multiplication expression
                    end
                    3. add returned expression to projection specification of S3 and return S3
                end
                10. re-establish environment (only maps $i to tuple variable F:0 again) and return S3
            end
            2. add transformed for clause with new let-quantified tuple variable (L:3) to S1
            3. return tuple variable reference to new tuple variable
        end
        3. add returned tuple variable reference to projection specification of S1 and return S1
    end
    10. re-establish environment (now empty again) and return S1
end
```

Chapter 5 Query Unnesting and Twig Discovery

> Never send a human to do a machine's job.
>
> Agent Smith

Let us take the time and look back at the previous chapter. What was achieved? Basically, 1) XQGM—an internal query representation—has been developed, together with a mapping from XQuery into XQGM, and 2) the semantics of XQGM has been defined by a logical algebra (LAL), and 3) a mapping from XQGM into LAL was given. In a way, the previous chapter has already presented a complete[1] XQuery processor: We could implement the LAL operators as they are and call them "physical algebra". We could implement the XQGM-to-LAL mapping and call it "plan generation". Thereby, we would have defined the complete query evaluation process, from a query in its external string representation to its evaluation based on the physical algebra. This thesis could be done at this point. From the number of following pages however, you might guess that this is not the case. The sketched approach to a "complete" XQuery processor has several drawbacks:

1. Often, the generated XQGM instances contain many correlated subexpressions (i. e., many dotted edges). Consider the XQGM instance in Figure 2.4 on Page 20. This simple query already contains eight nested subexpressions. Generally, a nested subexpression is produced for every variable reference in the normalized XQuery statement. Because normalization introduces a variable reference to `fs:dot` for every axis step and, because axis steps occur frequently, the number of correlated subexpressions is naturally high. At a logical level, this is no problem. But if we take a glimpse at the way how nested subexpressions can be (and will be) physically evaluated in the above sketched query processor (see also Chapter 8), the problem becomes clear: For example, consider `select(15)` in Figure 2.4. For every incoming *closed_auction* node, `access(16)` calculating *child::price* is evaluated. This *item-at-a-time* evaluation mode implies a certain amount of overhead, because the document (or an index) has to be opened (closed) in every step, before (after) the required nodes can be read. If many nodes are processed, this overhead adds up to costs that cannot be neglected and that influence the evaluation performance.

2. Operators may generate unnecessary large intermediate results. For an example, consider again the query in Figure 2.4. Operator `select(2)` contains a predi-

[1] ... to the extent XQuery is supported.

cate on the input generated by `access(14)`. All operators "in between" have to pass the generated text nodes. If the predicate would be pushed down from `select(2)` to `access(14)`, only text nodes fulfilling the predicate would be passed. Note, during plan generation, this predicate push-down would also facilitate the application of a content index to implement `access(14)`, because the necessary information to "detect" content-index applicability is now available in a single operator (and not distributed over multiple operators).

Besides *late* predicate evaluation, a suboptimal sequential operator arrangement also leads to unnecessary large intermediate results. In Figure 2.4, all paths are evaluated top-down, e. g., *site* → *closed_auctions* → *closed_auction* (in the subtree below L:5). Depending on the document, a "reverse" evaluation (*closed_auction* → *closed_auctions* → *site*) could lead to a much smaller size of intermediate results (if we assume that the number of *site* elements is high and the number of *closed_auction* elements is low). Although the decision about operator ordering is a physical issue (and has to be discussed during plan generation), a suitable treatment at the logical level would be beneficial to clarify the rewriting semantics.

3. The higher the number of operators in an XQGM instance, the higher the number of intermediate results that have to be passed. Therefore, for a given query, the number of operators should be reduced to a minimum to avoid the overhead resulting from passing intermediate results. In the initial XQGM instance, there are already opportunities to remove operators. For example, in Figure 2.4, `select(15)` receives one input from a tuple variable reference, which is then passed on via a correlated edge to `access(16)`. This is not really necessary, because the input from the tuple variable reference could be directly given to the access operator, thereby making `select(15)` unnecessary (assuming that the access operators returns nodes in distinct document order). But also during restructuring, opportunities to remove operators arise.

4. In Chapter 8, a set of XML evaluation algorithms will be introduced that provide for efficient bulk path matching. Besides these algorithms, the research community has also developed many techniques to find paths in XML documents. A query processor should be able to make use of these algorithms and techniques. As a prerequisite it needs to "discover" opportunities to do so. As an example, consider again Figure 2.4: the information that path `site/closed_auctions/closed_auction` has to be matched, is "encoded" into six operators. Investigating that large number of operators, for example, to test the applicability of a path index, is obviously cumbersome and leads to a high complexity in plan generation.

With the exception of *operator reordering*, the sketched problems will be tackled in this chapter by the following means: 1) *query unnesting* will remove correlated subexpressions, 2) intermediate results will be reduced by *predicate push-down*, 3) *operator fusion* will get rid of unnecessary operators, and 4) *twig discovery* will reveal opportunities to apply bulk path matching operators. The first three techniques have also been developed in the relational context [Mitschang 95]. However, for their applicability in XML, they have to be adapted. Operator reordering is an important ingredient for cost-based query optimizers, where the cost for different join orders can be assessed. Because this work does not consider a cost-based optimization, operator reordering is left open to future research. The fourth point, twig discovery, accounts for the integration of a certain class of physical operators. You

might argue that the assumption of certain physical operators at the present logical level is a design violation, because the logical layer should be oblivious of physical issues (the same also holds for operator reordering). Technically, this is true. The issue should be handled at the physical layer during plan generation. However, we chose to discuss the issue at the logical level, because it is much simpler. Before we can introduce how XQGM instances are restructured, a methodology to describe these restructurings is required.

5.1 Rewriting Methodology

Query rewriting is expressed by *rewriting rules*. A rule consists of a pattern, which is matched against an XQGM instance, and a *transformation instruction* (or a restructuring program), which rewrites the XQGM instance at the matched component. Technically, restructuring is implemented using the *infrastructure component* of the query processor. This component basically consists of a rule engine, which can traverse tree structures to match rule patterns. For this purpose, an XQGM instance is interpreted as a tree (although being a general graph) by ignoring correlated edges. Whenever a match occurs in the tree, the infrastructure component executes the rule's transformation instruction. The transformation instruction leaves the XQGM in a consistent and correct state, i. e., the result is still an XQGM instance and the semantics of the query before and after restructuring is the same. Query rewriting is a deterministic process, i. e., for every input XQGM instance, there is exactly *one* rewritten XQGM instance (which is passed to the plan generator). Note, if we would support operator reordering at this level, this statement would not hold anymore because, in general, a query can be reordered in many ways. To ensure determinism, only one rule is allowed to match at any time. As a design consequence, the patterns of the complete rule set have to be pairwise disjoint. Rules can be *chained*, i. e., the resulting subexpressions generated by the restructuring program of a particular rule can serve as input for another rule. Rule chaining requires that the infrastructure component matches rules bottom to top or subtree first: before a rule can match at a certain XQGM component C, all rules must have been matched in the subexpression below C. This guarantees that all rewritings have been applied to the subexpression.

The rule approach has some advantages over the "hard wired" (search and replace) alternative: Rules are self-enclosed and more descriptive, i. e., they only define what to match and how the matched component has to be modified; no further logic, for example to actually find a match, is required. Furthermore, the rule set can be easily extended when new optimization opportunities are found. Finally, a rule set can be seen as a query engine parameter. By altering the rule set (switching rules on and off), the behavior of the query engine can altered without changing the query engine itself. This is quite important for scientific projects such as XTC, because then various rewriting strategies can be tested quite easily.

In the following, a rule set containing nine disjoint rewriting rules will be introduced. According to the four problems sketched at the beginning of this chapter, these various rules can be categorized as:

1. rules for query unnesting,
2. rules for intermediate result size reduction,

3. rules to minimize the number of operators, and
4. rules to facilitate the mapping onto bulk operators.

Some rules might produce XQGM operators that have not been introduced in this work so far. New XQGM operators will, therefore, be defined "on the way". The rule set shown can be seen as a best-effort approach, i. e., many standard cases can be handled. However, the rule set will not be complete and further optimization opportunities might exist, which are left open for future work.

Most of the nine rules to be introduced will be shown by example. This decision was made to keep the size of this chapter reasonable. The first two rules show how unnecessary tuple variable references and unnecessary *descendant-or-self::node()* steps can be removed.

5.2 External Tuple Variable Reference Removal

As we have seen, during the transformation from XQuery AST into XQGM, all variable references are mapped onto tuple variable references. Some of these tuple variable references occur *inside* the same operator also containing the tuple variable referenced. For an example, consider the following query and its XQGM representation in Figure 5.1a:

```
for $i in doc("auction.xml")//item
where $i/location="United States"
return $i/name
```

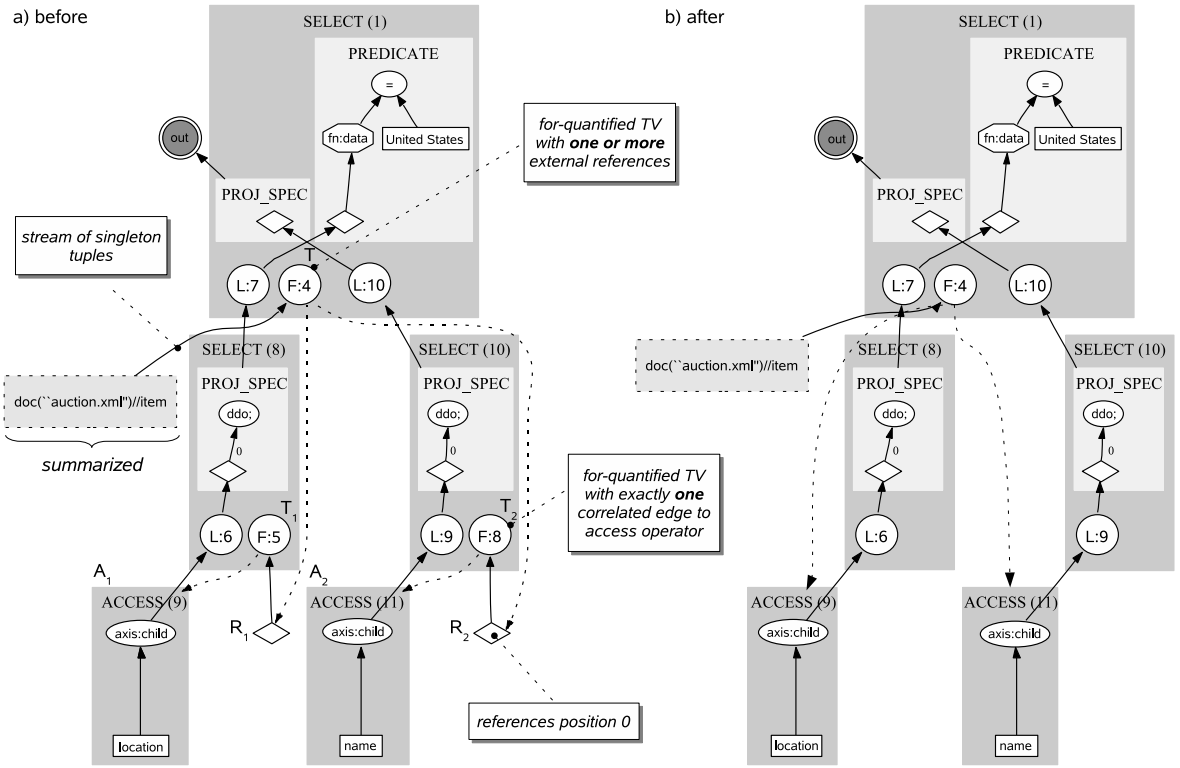
The tuple variable reference in the predicate of `select(1)` points to tuple variable L:7. In the following, we will call this type of reference *local*. On the other hand, there are also references to tuple variables from "outside". In Figure 5.1a, the input of tuple variable F:8 in `select(10)` is a reference to variable F:4 of `select(1)`. In the following, we will call this type of reference *external*.

Often, external tuple variables are not required, because their correlated input can be passed on directly to some access operator. The removal of these references is subject to the first restructuring rule, which is called *external tuple variable reference removal*. Figure 5.1 shows an application of this rule to the XQGM of the above query.

In Figure 5.1a (before the rule application), tuple variable F:4 has two external references. Each of these references is input to a *for*-quantified tuple variable (F:5 and F:8) which, in turn, serves a correlated input to some access operator (9 and 11). In this case, both correlated edges can be directly drawn from F:4 to the access operators, as shown in Figure 5.1b (after the rule application). The two tuple variable references and the two tuple variables they delivered their input to (F:5 and F:8) are not required anymore and are removed.

Of course, this rewriting is only possible, when the XQGM instance has a certain structure. The necessary restrictions are defined in the rule pattern. In Figure 5.1a, these restrictions are attached to the matched XQGM components for visualization. In summary, the rule pattern "searches for" a *for*-quantified tuple variable T with a stream of singleton tuples as input and one or more external references R_i. The external references deliver the item at position 0 to a *for*-quantified tuple variable T_i, which, in turn, has exactly one correlated edge leading to an access operator

Figure 5.1 Tuple variable reference removal



A_i. The rule's transformation instruction then rewrites every external reference to T fulfilling the requirements defined by the pattern. It removes the correlated edge to reference R_i and inserts an edge to access operator A_i. Then it removes reference R_i and the containing tuple variable T_i.

With the described rewriting in mind, the restrictions defined by the rule pattern become clear:

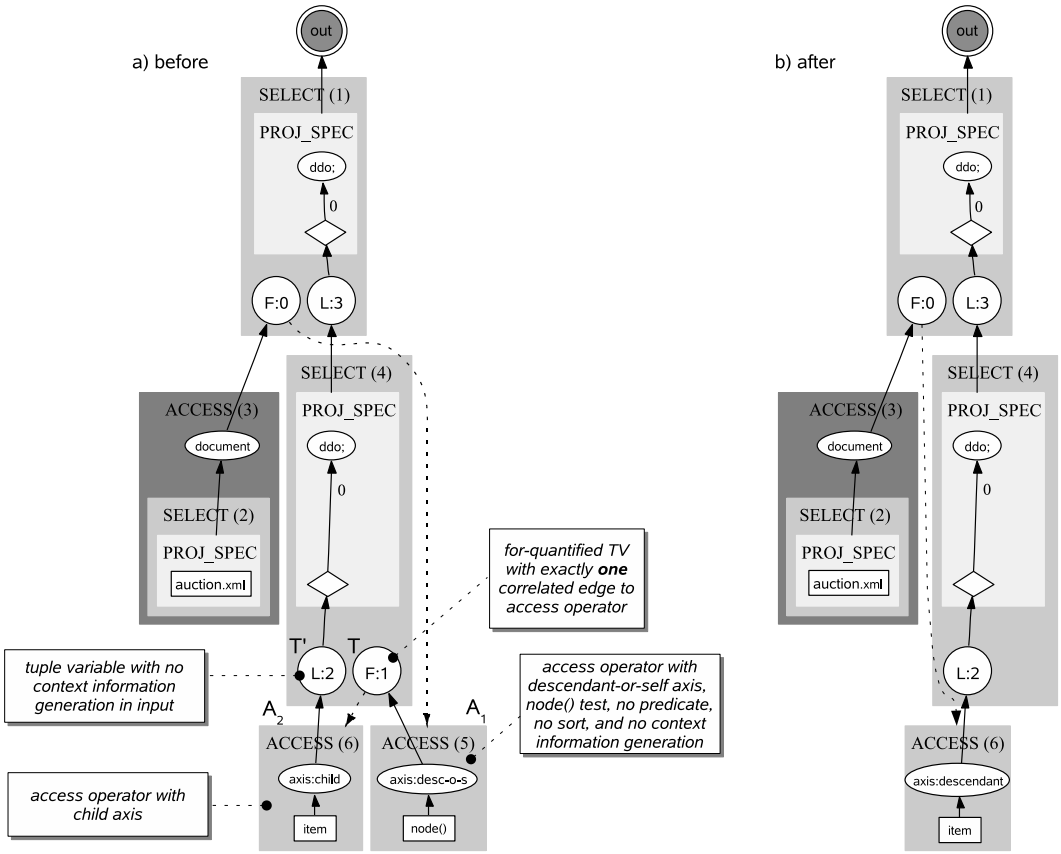
1. T has to be *for*-quantified because, in case of a *let* quantification, a complete sequence of nodes would be delivered to variable references R_i. Because access operators require a single node as correlated input (and not a sequence), the direct connection via a correlated edge between T and the access operators would not be possible. All further quantifications do not allow correlated edges.

2. A *let*-quantified tuple variable T_i with a correlated access operator cannot occur (due to the AST-to-XQGM transformation process and the rewriting rules introduced in this chapter). If T_i would have more than one reference (for example, also a local reference), its removal during rewriting would not be possible.

3. If R_i does not reference position 0 and the input below T would not deliver singleton tuples, the removal of R_i would not be allowed. Note, this situation cannot occur after the AST-to-XQGM transformation. However, as a result of other rewriting rules, this setting might be generated, which is why this restriction is required.

The rule removes unnecessary operators (i. e., references R_i). Therefore, it belongs

Figure 5.2 Removal of `descendant-or-self`



to the third category defined on Page 93. However, because it does not alter the structure of the XQGM instance substantially, it can be classified as a *helper rule*. Helper rules simplify the graph. Another approach to get rid of the type of correlated edges removed by this rule would be to avoid them during AST-to-XQGM mapping. This would require a more complex—however possible—mapping process.

5.3 Removal of `descendant-or-self`

The semantics of the double slash operator in XQuery sometimes causes confusion. As an example, consider query `doc("auction.xml")//item`, which is normalized to `doc("auction.xml")/descendant-or-self::node()/child::item`, before it is transformed into the XQGM instance shown in Figure 5.2a. Generally, the query yields the same result as `doc("auction.xml")/descendant::item`, which is why many XQuery beginners think that "`//`" equals "`descendant`". This is not true, when context information comes into play, i. e., `doc("auction.xml")//item[3]` is generally not equal to `doc("auction.xml")/descendant::item[3]`. The first query returns the third *item* below any node in the document, while the second query returns the third *item* in the whole document. Nevertheless, it is beneficial to avoid the

`descendant-or-self::node()` step expression whenever possible, because it is quite expensive to evaluate. The reason for that is the combination of the axis and the node test, which is very general and can affect the complete document. The *descendant-or-self removal rule* replaces this step expression with a descendant step.

Figure 5.2a shows the initial situation for sample query `doc("auction.xml")//item` from above. Similarly to the external tuple variable reference removal, the pattern searches for a *for*-quantified tuple variable T with exactly one correlated edge to an access operator A_2. The input of T has to be an access operator A_1 with a *descendant-or-self* axis, a *node()* test, no predicate, no sorting specification, and no context information generation (i. e., no "cp" or "cs" in the output specification). The access operator at the end of the correlated edge (A_2) has to have a *child* axis, no predicate, and no sorting specification. The node test, however, can be arbitrary. Finally, tuple variable T' which resides in the same operator as T and receives the input of A_2 (L:2 in our example), may not receive any input tuple with context information (i. e., with "cp" or "cs" set). Note, in our example, A_2 is the input of T'. These two components are, therefore, directly connected. This is, however, not necessarily the case, which is why the check for positional information is not issued on A_2, but on the input of T'. Whether context information is generated or not can be inferred from the projection specifications in the subtree below T'. The transformation instruction replaces the *child* axis in A_2 with the *descendant* axis and directly sets the correlated input of A_1 to A_2. Then, tuple variable T and access operator A_1 can be removed.

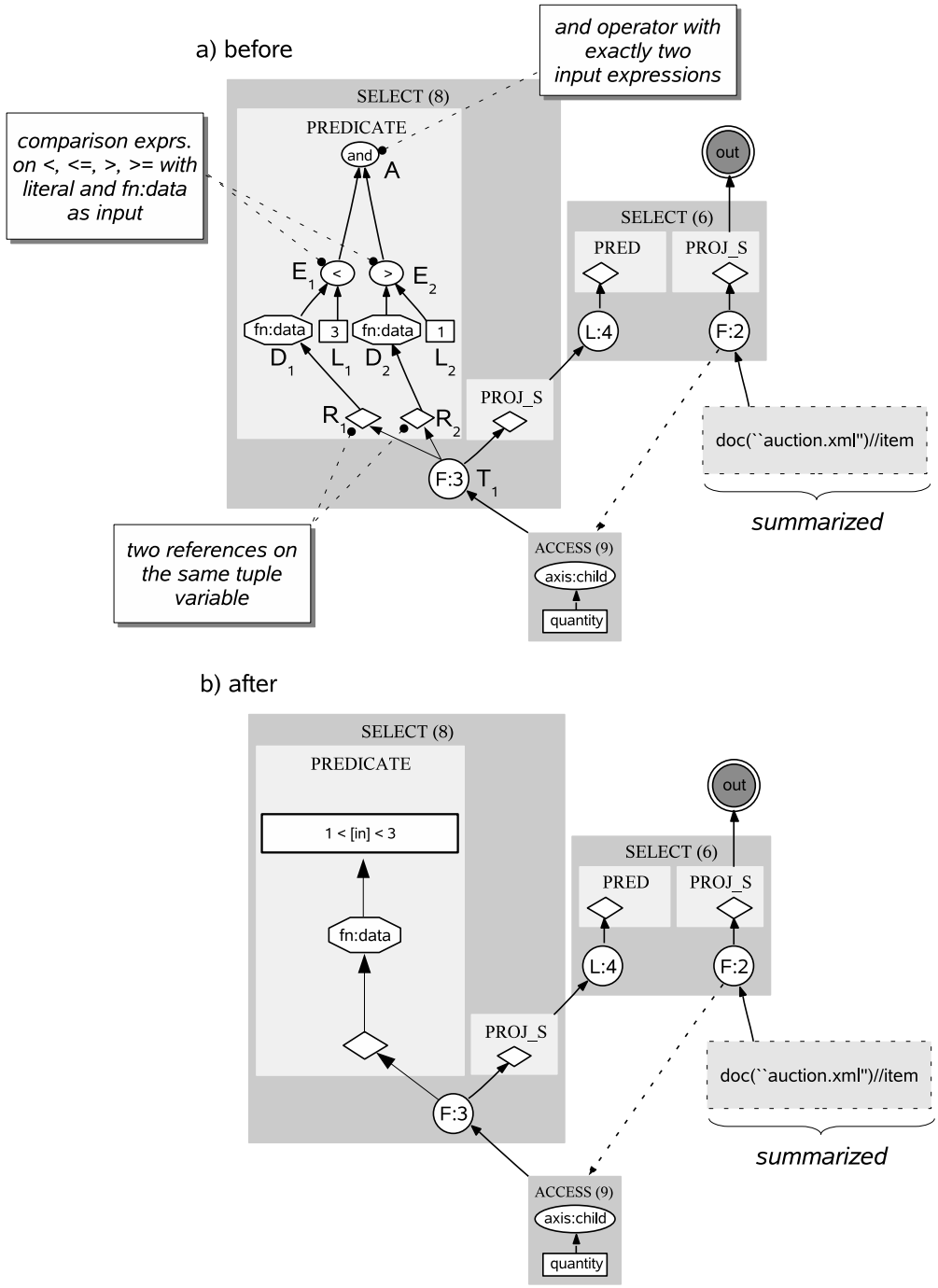
As in the previous rule, T has to be *for*-quantified with exactly one correlated edge to an access operator. Further references to T are not allowed, because then T could not be removed during rewriting. The input of T is the XQGM-transformed `descendant-or-self::node()` expression (A_1). The input may not have any predicate, sorting, or projection specification generating context information, because the input would carry additional semantics and would not be removable. Operator A_2 simply has to carry a *child* axis to capture the second part of the normalized "//" query. Tuple variable T' may not receive any positional context information. This requirement explains itself with the discussion at the beginning of this section.

As with the previous rule, the effect of this rewriting rule is to remove unnecessary operators, in this case, an access operator. Therefore, it also belongs to the third category on Page 93. Because it also does not alter the structure of the XQGM instance substantially, it can be classified as a *helper rule*. Compared to the previous rule, however, it gets rid of a more heavy-weight access operator, thus reducing I/O during evaluation (if we assume a physical 1:1 implementation of the XQGM instances).

5.4 Range Query Detection

Chapter 7 will introduce two kinds of content indexes that allow to efficiently evaluate content-based predicates. Content-based predicates can be *point queries* or *range queries*. A query with a point predicate contains a comparison for equality with some literal as, for example, in `doc("auction.xml")//item[location="United States"]`. A range pred-

Figure 5.3 Range query detection



icate defines one or two boundaries, between which the desired values have to reside, for example `doc("auction.xml")//item[quantity < 5]` or `doc("auction.xml")//item[quantity[. > 1 and . < 3]]` (see Figure 5.3a for the XQGM instance of this query[2]). The first predicate has an *open boundary* (the range is not limited on one side), while the second one has a *closed*

[2]Note, query `doc("auction.xml")//item[quantity > 1 and quantity < 3]` is not an equivalent version of the query shown in Figure 5.3a, because an *item* could have multiple *quantity* children.

boundary (the range is limited on both sides). Detecting range queries with open boundaries is simple, because they contain just one comparison with a literal (similar to the point query). Detecting range queries with a closed boundary is more complicated: A range is defined using the *and* Boolean XQuery operator and two comparisons. The problem is to analyze, whether the boundaries span a range. For example, predicate `quantity[. < 5 and . < 6]` does not span a range although the *and* keyword and two comparisons were used.

The *range query detection* rule can detect ranges as follows: First, it searches for an XQuery *and* expression A and examines its input expressions E_i. If there are not exactly two input expressions, the rule pattern does not match. Furthermore, the input expressions have to be comparison expressions based on one of the following comparison operators: $<$, $<=$, $>$, or $>=$. Each comparison has to have an *fn:data* function D_i and a literal L_i as input. Based on the literals, the comparison operators, and the position of the *fn:data* function, the rule then computes whether a range is spanned. If not, the rule does not match. The functions D_i have to have a tuple variable reference R_i as input, which either references the same tuple variable T_i or not. In the first case, the rule matches successfully, because the same input is given to both data functions D_i. In the second case, the rule does not match.

During rewriting, the complete subtree below A inside the predicate is replaced by a *between expression*, as shown in Figure 5.3b. A *between* expression is a new type of XQGM expression (for which no counterpart in XQuery exists). Thus, it is a subtype of the Expression type, defined in Figure 4.1 on Page 46. The *between* expression receives five parameters:

- two literals, one for the lower boundary and one for the upper boundary,
- one input, referenced as `[in]` in the graphical representation, and
- two Boolean values *minInclusive* and *maxInclusive* specifying whether the lower/upper boundary is included in the range or not.

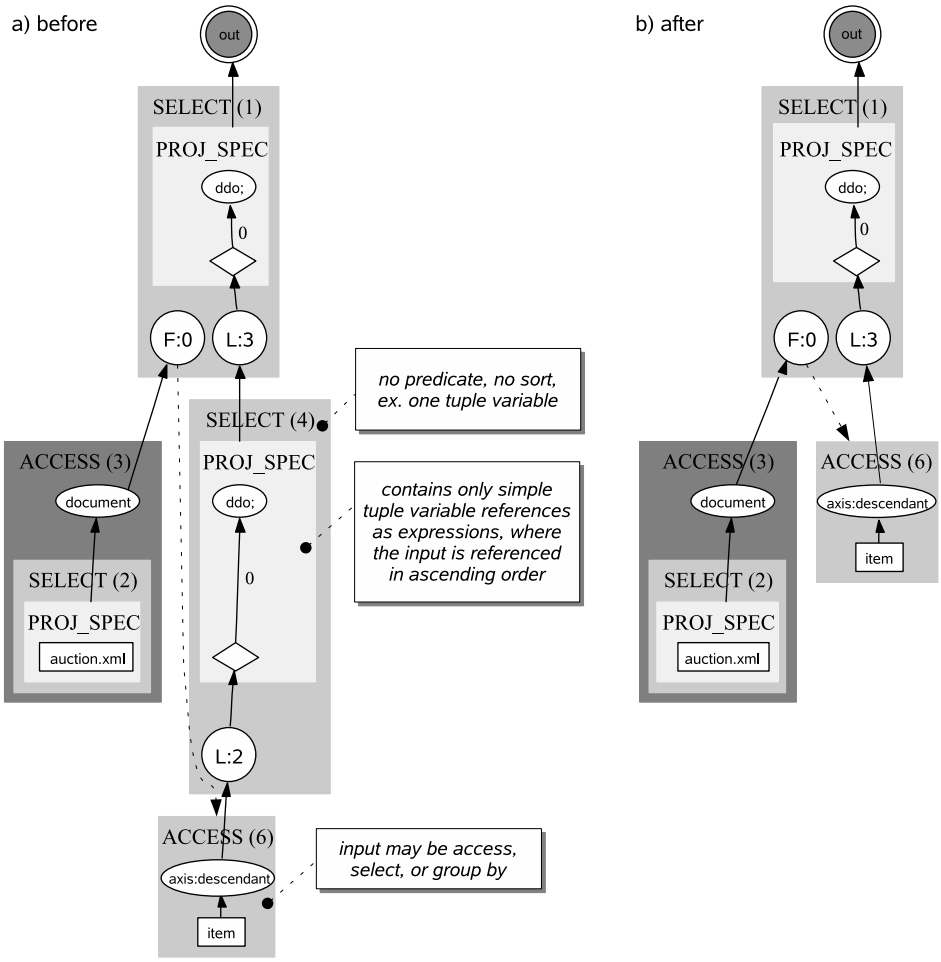
The *between* expression is a Boolean expression. It checks whether the input item lies inside the fixed range. The range query detection rule facilitates the mapping onto physical operators. It, therefore, belongs to category four on Page 93.

5.5 Select Fusion

All of the rewriting rules introduced so far modify select operators. Sometimes, a select operator is left in a state, where it just has one tuple variable and a projection specification. For an example, consider `select(4)` in Figure 5.2b, which is again depicted in Figure 5.4a for convenience. The only task, this select operator still has, is to apply the *ddo* function to the result of the access operator below. In this case, the projection specification of the select operator can be merged into the access operator below and the complete select operator can be removed. Note, because the access operator returns the nodes already in distinct document order, the projection specification vanishes completely in this example.

The *select fusion rule* is responsible for this kind of rewriting. It merges projection specifications and removes select operators. The select operator matched by the rule pattern must not have any predicate or sorting specification (because, otherwise, the select operator cannot be removed). Furthermore, the select operator must contain

Figure 5.4 Select fusion



exactly one tuple variable. The projection specification may define the *ddo* function and the generation of context information ("cp" and "cs"). However, it may not have any "complex" expressions. The only expressions allowed are tuple variable references that simply pass items from the input tuples (i. e., the projection specification defines a projection in the classical sense). In the following, we will refer to a projection specification of that shape as a *simple projection specification*. On the other hand, if the projection specification contains also non-reference expressions, we will call it *complex*. In our example, there is only one tuple variable reference. However, in more complex queries, there might be more. Because they all refer to one and the same tuple variable, they differ in the position they access on an input tuple. As a further restriction, a position may not occur twice among the tuple variable references. Finally, the input operator may be a select operator, an access operator, or a *group by* operator. The *group by* operator will be introduced in Section 5.7, where the unnesting rule is discussed. For now, we only need to know that the *group by* operator also has a projection specification, into which the projection specification of the select operator can be merged.

The transformation instruction distinguishes between access operators and *select*/*group by* as input operators. In case of an access operator, an existing *ddo* output

modifier is ignored, while context information generators are simply copied into the projection specification of the access operator (which is created, if not already there). Ignoring *ddo* is correct, because all access operators deliver their result in distinct document order. If the input operator is a select or a *group by*, the "upper" projection specification P_1 (of the select operator to be removed) is merged into the "lower" projection specification P_2 (of the input operator). First, *ddo*, "cp", and "cs" are copied to P_2. Then, output expressions are removed from P_2. During removal, only those output expressions are retained, which are referenced by some tuple variable reference contained in P_1. Note, the reason why the access positions of the tuple variable references of P_1 have to be disjoint, is to avoid copying output expressions in P_2. Every output expression is referenced exactly once. However, it may happen that the access positions in P_1 imply a reordering of output expressions in P_2. Then, the output expressions in P_2 are reordered accordingly. Finally, the upper select operator is replaced by its input operator and, thus, removed.

The select fusion rule will never match on an initial XQGM instance. However, as stated at the beginning of this section, some rules might leave a select operator in a state allowing the removal of the select operator. Therefore, the rule can also be classified as a *helper rule*. It, furthermore, belongs to category three on Page 93.

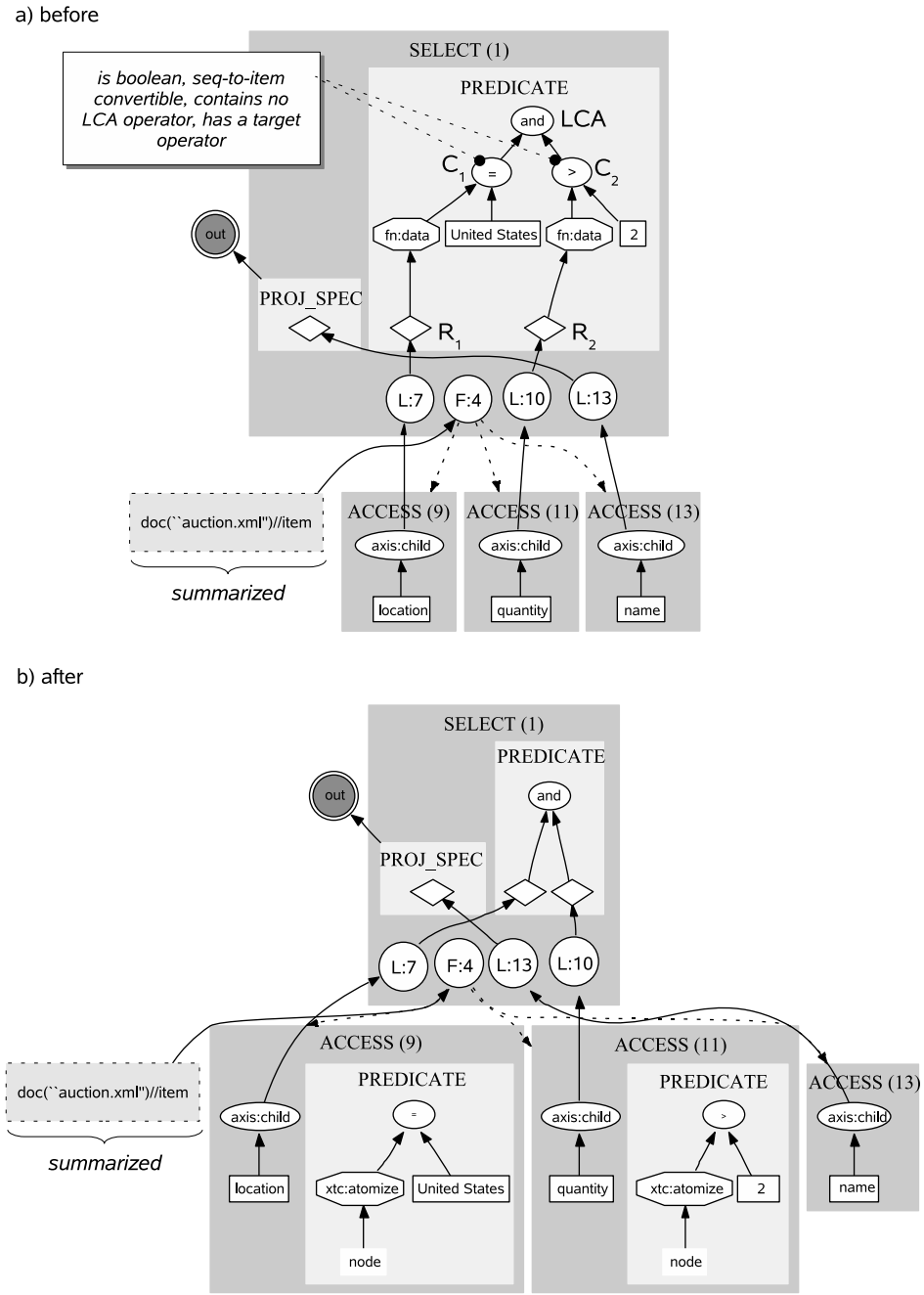
5.6 Predicate Push-Down

Pushing down predicates is a standard technique in relational query processors. Also for XML query processing, the concept can be applied. Consider the following query:

```
for $i in doc("auction.xml")//item
where $i/location = "United States" and $i/quantity > 2
return $i/name
```

Figure 5.5a presents the XQGM instance for this query. You can observe that the predicate is evaluated inside the select operator on the output generated by the two access operators delivering all *location* and all *quantity* nodes below a qualified *item* node. In Figure 5.5b, the select predicate is split up and pushed down into the access operators. In this case, the access operators do not deliver all nodes, but only those that fulfill the content predicate. The remaining predicate in the original select operator (*and*) then only has to check, whether at least one node is returned by an access operator. During rewriting, a new syntactical construct is introduced into XQGM, namely, the *accessed node reference*, depicted as a borderless box containing the string "node" inside the access predicate (see Figure 5.5b). The semantics of this box is to refer to the current node accessed by the access operator. Note, because an access operator does not carry a tuple variable, a tuple variable reference could not be used to express this kind of node reference. Therefore, the accessed node referenced is used to refer to the current node. Besides this component, a new function was introduced: *xtc-fn:atomize*. The prefix *xtc-fn* signals that this is not a standard XQuery function, but a function specific to the XTC system. The *xtc-fn:atomize* function has almost the same semantics as the *fn:data* function: it returns the *string value* of a node (i. e., the concatenation of all text nodes in the subtree below a specific node). The only difference is that *fn:data* receives and returns sequences of nodes/strings, while *xtc-fn:atomize* operates on single items (i. e., node-by-node). The conversion from *fn:data* to *xtc-fn:atomize* was necessary because, before rewriting, the first func-

Figure 5.5 Predicate push-down



tion was called after a *let*-quantified tuple variable collected all intermediate results returned by the access operator. During rewriting, the *fn:data* function is "pushed over" the *let*-quantified tuple variable, thus no node collection is done on the input of the function. Therefore, it operates directly on each item. To make this transition explicit, a new function, the *xtc-fn:atomize* function, was introduced. This conversion process is called *sequence-to-item conversion*. Note, not all XQuery functions can be converted. For example, an aggregation function, like *count*, requires a *let*-quantified tuple variable to collect the intermediate result. Therefore, a predicate

with a *count* function cannot be pushed down.

In general, the predicate push-down rule can handle more complicated situations than in the example above. The rule works as follows: It analyzes expressions with one or more tuple variable references contained in selection predicates. The references may not be positional (i. e., no references to "cp" or "cs"). Furthermore, they all have to reference *let*-quantified tuple variables. For every pair of references, their least common ancestor (LCA) is computed. In our example, the LCA of references R_1 and R_2 is the *and* expression. Then every branch below an LCA operator (in our example, the two comparison expressions C_1 and C_2) is matched as follows:

1. The branch is not allowed to contain any other LCA operator. This situation might occur in case of multiple Boolean operators. Consider the following counter example: A branch may contain another LCA operator, if the predicate is nested, as for example *green and yellow or orange* (where *green*, *yellow*, and *orange* are tuple variable references). Then the LCA of *green* and *orange* is 'or', and one branch rooted at 'and' contains another LCA, namely the one between *green* and *yellow*. As a result, every branch has exactly one tuple variable reference.

2. The topmost operator of the branch has to be a Boolean operator. This ensures that it can be embedded into an existing predicate (somewhere in the subtree where the branch is pushed down to) using an *and* operator.

3. All functions in the branch have to be sequence-to-item convertible. This is necessary, because the branch will be "pushed over" a *let*-quantified tuple variable.

4. A *target operator* for the branch has to exist. The target operator is an operator in a subtree below the predicate's containing select operator, which finally receives the branch after rewriting.

If all of these criteria are true for at least one branch, the rewriting rule matches. Otherwise, the predicate cannot be rewritten.

In our example, the two branches did not contain any other LCA, the topmost operators C_1 and C_2 are Boolean, and the *fn:data* function is sequence-to-item convertible. The target operators are of course `access(9)` and `access(11)`. In general, finding the target operator might be slightly more complex. For example, consider the query depicted in Figure 2.4 on Page 20. Here, `select(2)` contains a predicate with a tuple variable reference on `L:11`. This tuple variable, in turn, receives its input from operator `select(11)`; `select(11)` receives input from `select(12)`; and, finally, `select(12)` receives input from `access(14)`. Basically, the input for the predicate is passed across a chain of operators that consists of tuple variables, operators, projection specifications, and tuple variable references. This chain is named *input chain* in the following. The target operator may be any of the operators in the input chain; the deeper in the subtree, the better. The rationale is to restrict the intermediate result as early as possible. During rewriting, the predicate push-down pattern examines the input chain from top to bottom to find the target operator. Let us assume, the current operator in this descent is O. The following criteria on O influence the selection: 1) If O is not a select operator or not an access operator, the previous operator in the chain is returned as target operator. 2) If O has a predicate, the operator is returned as target operator. 3) If O has a complex projection specification (i. e., a projection specification that does not solely consist of tuple variable references), the previous operator is returned. 4) If O generates context information ('cp' or 'cs'), the previous operator is returned. If the chain does

not contain a target operator (because already the first operator O had to "return the previous operator" which does not exist), the fourth criterion from above is not fulfilled and the branch cannot be rewritten.

If the branches and the target operators are found, the rewriting process can start. Every branch is detached from its LCA and replaced by a tuple variable reference corresponding to the one contained in the branch. Depending on the type of target operator, the following actions are executed: If the target operator already has a predicate, the branch is attached to that predicate using a Boolean *and* operator. Otherwise, a new predicate is created and the branch is added. If the target operator is a select operator, the tuple variable of the branch is rewritten to the tuple variable, which belongs to the input chain for that branch. If the target operator is an access operator, the tuple variable reference is replaced by the above introduced accessed node reference ("node").

The predicate push-down rule minimizes intermediate results assuring that predicates are evaluated as early as possible. Therefore, it belongs to the second category on Page 93. Besides intermediate result size reduction, the rule also facilitates the mapping onto the physical algebra. As you can see in Figure 5.5, the content predicates now reside inside the access operators. Chapter 7 will introduce efficient content indexes that can be used for the implementation of these access operators. Therefore, the rule also belongs to category four. Finally, we want to state that the application of the predicate push-down rule is not always beneficial, as the following example shows: Assume that every *item* node has exactly one *location* node and exactly one *quantity* node. Then it does not make much difference whether the predicate is pushed down or not (if we further assume that the access operators cannot be answered by appropriate indexes directly).

5.7 Query Unnesting

We have already discussed that, due to variable references, an XQGM instance can contain many correlated subexpressions. These subexpressions imply an item-at-a-time evaluation style, i. e., for every value at a tuple variable with a correlated edge, the subexpression has to be evaluated. This results in a certain amount of overhead (for example, to open and close indexes) influencing evaluation performance. To avoid this overhead, we could try to get rid of the correlated edges. A possible solution would rewrite correlated subexpressions into joins. Joins are bulk operators that can process larger chunks of data at a time. This strategy is also practiced in relational systems. Consider the following SQL query (which is a modified version of Q9 from [Mitschang 95]):

```
SELECT DISTINCT Q1.name, Q1.profession
FROM EMP Q1
WHERE EXISTS                          // existential subquery
        (SELECT *
         FROM DEPT Q2
         WHERE Q1.dno = Q2.dno        // correlated predicate
         AND Q2.location = 'KL')
```

The query returns the *name* and *profession* of all *employees* whose *department* is located in 'KL'. The query contains an existential subquery receiving data via a correlated predicate. Essentially, the subquery is executed for every employee. An alternative

version of this query, which is based on a join, looks like follows:

```
SELECT DISTINCT Q1.name, Q1.profession
FROM EMP Q1, DEPT Q2
WHERE Q1.dno = Q2.dno
AND Q2.location = 'KL')
```

This query has no subquery; instead, a join is used and the correlated predicate became the join predicate. For the implementation of the relational join, many efficient algorithms have been developed. The query optimizer can now pick one to execute this query. Because this query does not contain any nested subexpression anymore, we call it *unnested*. The process of rewriting a query into its unnested form is called *unnesting*.

Unnesting is also possible for XML queries, but instead of value-based correlated subexpressions, we unnest *structural* correlated subexpressions (or just structural subexpressions for short). A structural subexpression starts at a tuple variable having a correlated edge to an access operator. The access operator receives the current node at the tuple variable and executes some navigation. Examples for structural subexpressions can be found quite frequently: in Figure 5.5, where tuple variable F:4 provides the correlated input for three structural subexpressions (to access operators 9, 11, and 13) or in Figure 2.4 on Page 20, where tuple variable F:8 provides the input for exactly one structural subexpression. In the following, we consider tuple variables with multiple structural subexpressions first. Queries with this kind of tuple variables are rewritten to alternative representations, where every tuple variable has at most one correlated edge. Tuple variables with a single correlated edge will then be discussed at the end of this section.

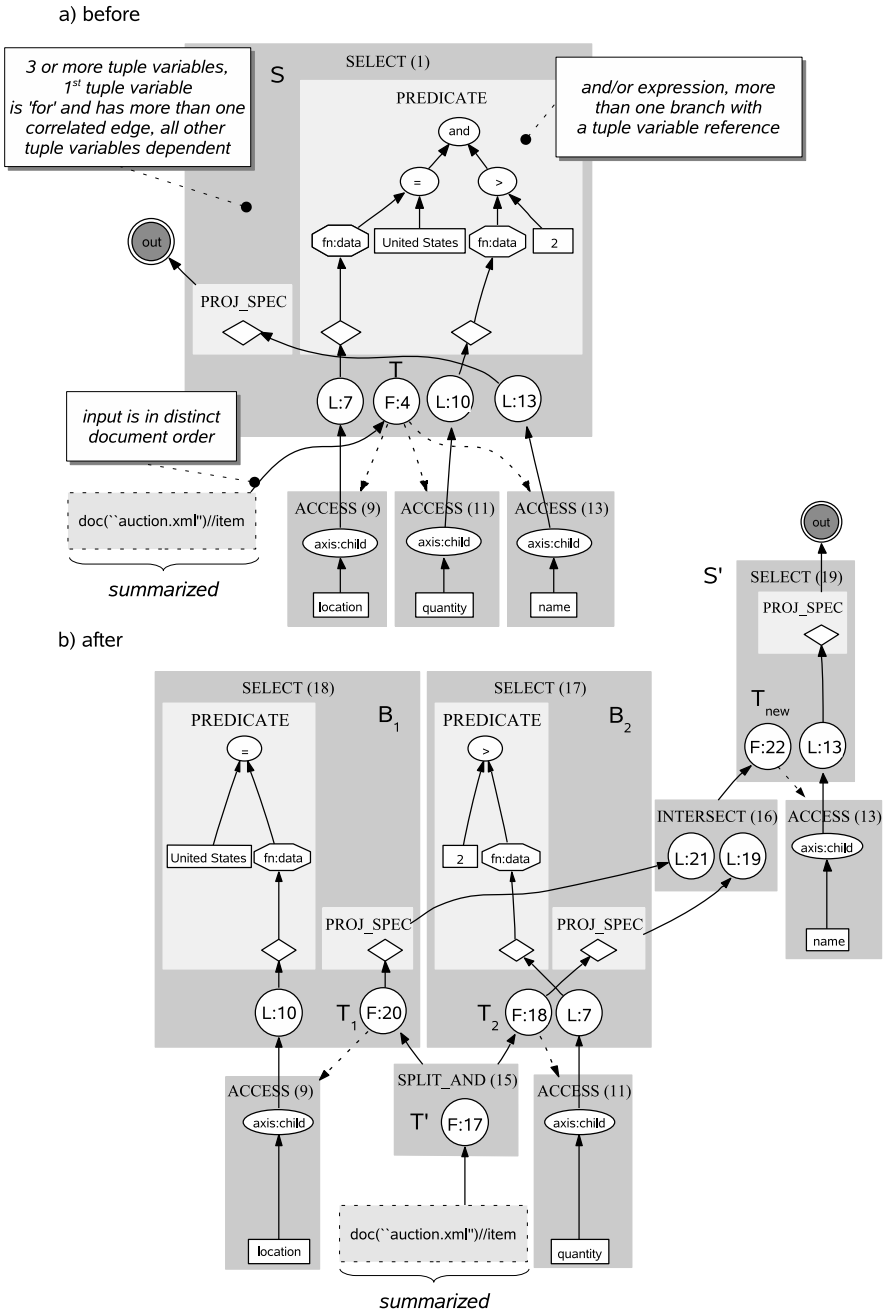
5.7.1 Boolean Split

The first situation we consider occurs, when Boolean *and* and *or* operators are used on the output of dependent tuple variables. As an example, consider again the following query:

```
doc("auction.xml")//item[quantity > 2 and location = "United States"]
```

We have already seen the initial XQGM for this query in Figure 5.5a, when we considered predicate push-down. However, now we want to get rid of the multiple correlated edges at tuple variable F:4. This is the task of the *Boolean split rule*. An example is presented in Figure 5.6. The initial XQGM instance is on the left hand side. Tuple variable F:4 has three correlated edges, and F:4's select operator has a Boolean *and* predicate which is defined over the dependent tuple variables of F:4. In this situation, the select operator can be rewritten by splitting the select operator and the Boolean predicate and by using an XQuery intersect operator to merge the split results. The rationale behind this rewriting is the equation $\sigma_{a \wedge b}(v) = \sigma_a(v) \cap \sigma_b(v)$: A selection ($\sigma$) with a conjunctive predicate ($\wedge$) and the conjunctors a and b over input v is the same as the intersection of the selection of each conjunctor on input v. Similarly, we can rewrite *or*-based predicates as $\sigma_{a \vee b}(v) = \sigma_a(v) \cup \sigma_b(v)$ using a union operator. On the right-hand side of these equations, input v occurs twice. In XQGM, however, the input operator graph should not be "copied", because this would imply a repeated evaluation. Therefore, a new operator is added to the XQGM operator set, which allows to split an intermediate result and to send it to multiple following operators. This operator is called *split* operator. The only component, a

Figure 5.6 Boolean split



split operator contains, is a *for*-quantified tuple variable, which signals that each input is directly forwarded to the following operators. As all other XQGM operators, a split has a unique ID (shown in braces after the operator name). To distinguish split operators which have been instantiated for *and* and *or* predicates, the operator name furthermore carries the split type (`AND` or `OR`). Due to the simplicity of the operator's semantics, we do not give a more detailed definition.

With the split operator, Figure 5.5b reads as follows. The input of the split operator is evaluated to a sequence of *item* nodes. This sequence is copied and each copy is

passed to a select operator. Each select operator filters its input sequence using the given predicate. The filtered sequences are passed to the intersect operator, which returns only those *item* nodes occurring in both input sequences. On these items, the final `child::name` step is evaluated.

The pattern of the Boolean split rule searches for a select operator S with a predicate. The select operator must have three or more tuple variables, of which the first one (T) is *for*-quantified and the remaining ones are dependent on that first tuple variable. Furthermore, T must have an input in distinct document order and more than one correlated edge. The predicate must be an *and* or an *or* expression, where more than one branch of the expression has a tuple variable reference (i. e., a non-constant input).

During rewriting, the rule first creates a split operator with a *for*-quantified tuple variable T'. The input of T is removed and attached to T'. Then, a union or intersect operator is generated, depending on the type of Boolean expression. The original select operator S is then decomposed as follows: For every branch, the rule

- creates a new select operator B_i with a *for*-quantified tuple variable T_i receiving the split operator as input;
- generates a projection specification in each B_i referencing tuple variable T_i;
- generates a predicate in each B_i and injects the predicate; during injection, the necessary tuple variable is moved from the original select S into the branch B_i;[3]
- adds the created select branch B_i as input to the set operator.

In the last step, the projection specification or a possibly existing sorting specification of the original select operator S have to be handled. Therefore, if 1) S contains a complex projection specification (i. e., if it contains an expression that does not simply consist of a reference), or 2) if positional information is generated, or 3) if a tuple variable other than T is referenced, a new select operator S' is created with a *for*-quantified tuple variable. The *set* operator is passed as input to this new tuple variable. Furthermore, S' receives the projection specification and all necessary tuple variables from S. The same is done for a sorting specification. Note, if dependent tuple variables are transferred, the new tuple variable T_{new} is used as the source for the correlated edge into the subtree of the dependent variable. In our example, F:22 serves as input to `access(13)`, which is the input of dependent tuple variable L:13. Finally, S is replaced by S'.

In the rewritten XQGM instance, the number of correlated edges is the same as before. However, now every tuple variable created for a branch (in B_i) has at most one correlated edge. This was the intention behind the Boolean split rule. Tuple variable T_{new} may still have multiple correlated edges. These are handled in the next subsection. Note, the input of the split up tuple variable needs to be in distinct document order, because, in XQuery, the set-based operators are defined to return their result in distinct document order. If the input would contain duplicates, they would be falsely removed by the set operators.

Although the rule does not unnest the query, we put it into category one on Page 93. We do so, because the rule substantially simplifies unnesting. In this section, we used the same example and the same initial XQGM instance as for the discussion

[3] If the predicate contained a reference to T, no compenents need to be relocated; a reference to T_i is then sufficient.

of the predicate push-down rule. Essentially, both rules match this instance and, therefore, could rewrite the query, resulting in two different outcomes. If both rules are contained in a rule set, they are nevertheless disjoint. We just have to make sure that one rule matches, before the other one does. In fact, the order among the rules would even be arbitrary.

5.7.2 Multiple Correlated Expression pull-out

In case of Boolean *and* and *or* operators, the set-based rewriting provided an elegant opportunity to get rid of multiple correlated edges. Unfortunately, multiple correlated edges can also occur in other situations, where they have to be handled as well. Otherwise, the unnesting rule introduced at the end of this section would not be applicable. As an example, consider the following simple query:

```
for $i in doc("auction.xml")//item
return <qloc>{$i/quantity}{$i/location}</qloc>
```

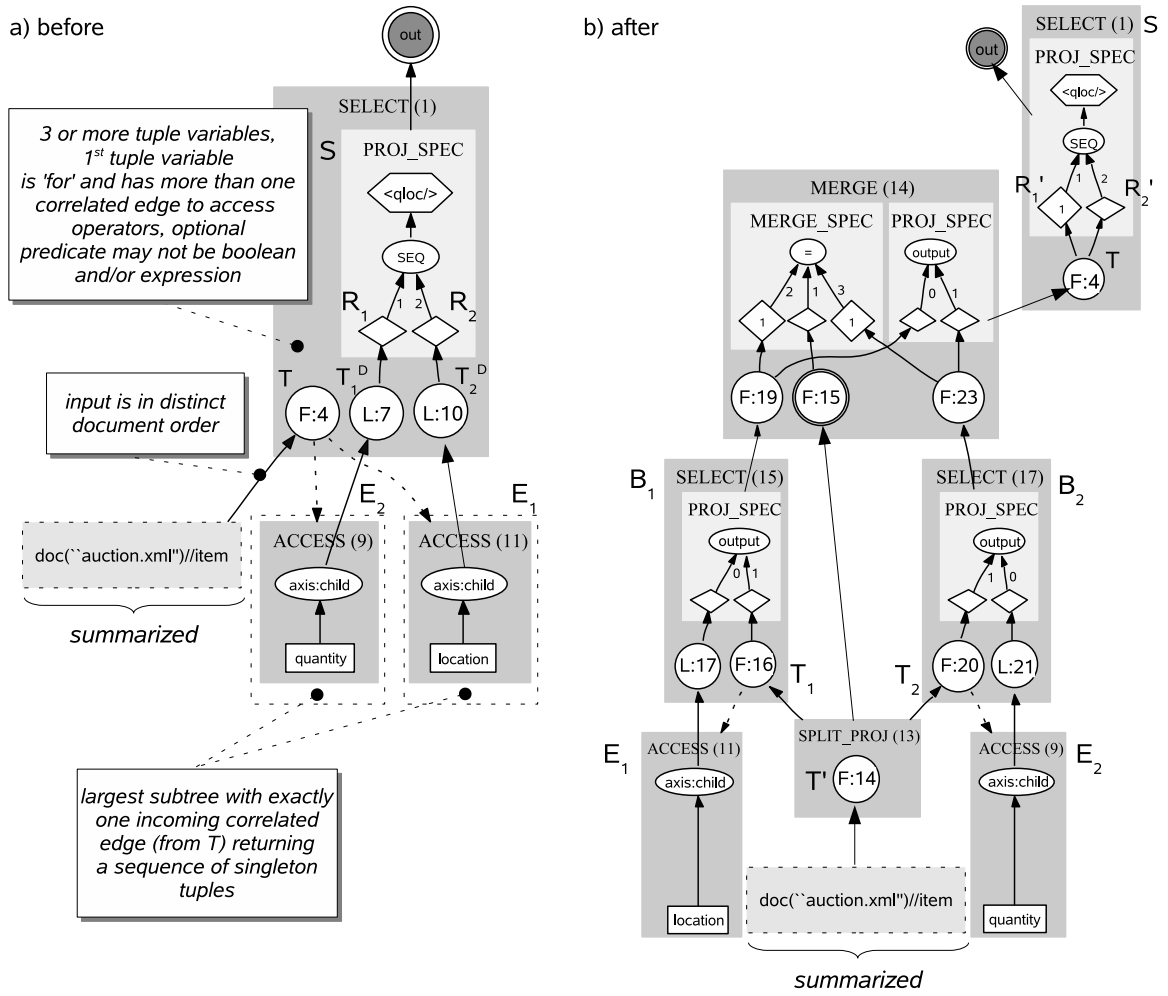
For every *item* element, this rule creates a new *qloc* element containing the *quantity* and the *location* of the item. The XQGM of this query is presented in Figure 5.7a. The tuple variables in the select operator have quite a similar structure to the example in Figure 5.6a. Tuple variable F:4 has two correlated output expressions into access operators and two dependent tuple variables. However, this time, not a predicate is evaluated over the dependent tuple variables, but a complex projection specification. It is, therefore, not possible to apply the Boolean split rule, which is why we need another technique to split up the correlated edges.

The *multiple correlated expression pull-out rule* (or *pull-out rule* for short) is a generalization of the Boolean split rule. It can split up the correlated edges of a *for*-quantified tuple variable in a select operator S independent of further components inside S (i. e., independent of predicates, sorting, and projection specifications). Compared to the Boolean split rule, the logic behind this one is more complicated. The rule requires the introduction of a new operator to XQGM: the *merge* operator. Before we introduce this new operator, let us go back to the example and consider the rewriting.

On the left hand side, the select operator receives a sequence of *item* elements. For every *item*, the two correlated access operators are evaluated and their results are collected in two intermediate sequences. These are fed into the 'SEQ' operator, which combines them. The combined result then is wrapped inside a new *qloc* node, which is returned. Note, the number of returned *qloc* nodes is the same as the number of *item* elements. Even if neither of the two access operators produces a result, the projection specification generates an (empty) output element. We have to keep this behavior in mind during the discussion of the rewriting rule.

The right hand side has a similar structure as the rewriting generated by the Boolean split rule: the input is given to a split operator whose output is processed in two branches. The result of the branches is then merged together again. Let us take a closer look to the branches: In the first branch, an axis step is executed on the given *items*. For each *item*, this step returns a sequence of *location* nodes (let us assume that an *item* has more than one *location* node). The select operator passes a complex tuple consisting of the *item* node and the sequence of *location* nodes to the *merge* operator. The same happens in the second branch with *quantity* nodes. Finally, the merge operator has a third input directly originating from the *split* operator. This input

Figure 5.7 Multiple correlated expression pull-out



simply passes all *item* nodes. The *merge* operator then has the task to merge the three input sequences into one, where the *merge specification* defines, which tuples belong together. In our example, all tuples with the same *item* (value) belong together and are merged into one. A merged tuple then has a field containing the *item* node and two other fields containing the sequence of *location* and the sequence of *quantity* nodes. From this merged tuple, only the sequences are projected and passed on to the final select operator. This operator builds the new *qloc* elements. Note, at this point, the 'SEQ' operator has to receive the same input as before. However, one issue is still missing: What happens, if an item has neither a *location* nor a *quantity* child? For this case, we define the tuple variable of the merge operator receiving the input directly from the *split* operator (i. e., F:15), as *outer*. This is indicated by the double circle around F:15. If a tuple variable is outer, a tuple is generated, even if the other tuple variables do not generate any result. Their values are simply set to "empty sequence".

The pattern of the pull-out rule is quite similar to the one of the Boolean split rule. It searches for a select operator S with three or more tuple variables, of which the first

one (T) is *for*-quantified and has a duplicate-free input in document order. Again T must have more than one outgoing correlated edge to access operators, but, in contrast to Boolean split, not all remaining tuple variables need to depend on T. If S has a predicate, it may not consist of a Boolean *and* or *or* expression to avoid a collision with the Boolean split operator.

The transformation instruction also resembles the actions of the Boolean split rule: First, a split operator with a *for*-quantified tuple variable T' is created. The name of this split operator carries the suffix `PROJ` to signal that this operator was generated during a multiple correlated expression pull-out (in the following, we will call this kind of operator a *projection split*). The input of T is removed and attached to T'. Then, a *merge* operator is instantiated and a *for*-quantified *outer* tuple variable is added, which receives an input from the split operator. In contrast to Boolean split, the original select operator S is not completely decomposed. Only the tuple variables are modified. For each dependent tuple variable T_i^D, the input expression E_i is analyzed bottom up, starting at the correlated access operator. Essentially, we are searching for a tuple variable that roots an XQGM-transformed, already rewritten relative path expression starting at the correlated access operator. Navigating up the subtree (at most until T_i^D is reached), the rule searches for the largest subexpression with the following properties: 1) the current subtree has exactly one incoming edge (and this edge starts at T), 2) no tuple variable in the subtree has a correlated edge (i. e., the subtree is completely unnested), and 3) the current subtree produces a sequence of singleton tuples as result. The tuple variable rooting this subtree T_i^x might be equal to T_i^D (as in our example), but it does not necessarily have to be equal. An example containing both "types" of tuple variables is the following query of the XMark [Schmidt 02] query set (because the corresponding XQGM instance is quite large, we omit its display here):

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

In XQGM, variable `$p` is represented by a *for*-quantified tuple variable (corresponding to T) inside a select operator S, which provides the correlated input for the two paths `$p/name/text()` and `$p/@id` (which we assume are already rewritten, i. e., unnested, and which, therefore, do not contain a tuple variable with a correlated edge). The first path is rooted by a tuple variable T_1^x, which is equal to some T_i^D, i. e., it belongs to the same operator S as T. This is because, in the XQuery representation, path `$p/name/text()` is evaluated in the return expression of the *for* clause, which defined `$p`. On the other hand, the rooting tuple variable T_2^x for path `$p/@id` is not equal to any T_i^D in S, because the path is evaluated inside the *where* expression of different *for* clause (namely, of the one defining `$t`). Obviously, the pull-out rule can only pull out the two rewritten paths, but not any further subexpressions (such as the value-join predicate in the *where* clause).

With this example in mind, let us come back to the rewriting process. Having found T_i^x, the rule

1. creates a select operator B_i containing a *for*-quantified tuple variable T_i having the split operator as input;

2. detaches the input E_i of tuple variable T_x^D and adds a *let*-quantified tuple variable to B_i with E_i as input;
3. adjusts the correlated input edge of E_i to T_i;
4. creates a projection specification inside B_i and adds an output expression containing a tuple variable reference to the *let*-quantified tuple variable;
5. adds a tuple variable reference to T_i to the projection specification;
6. adds another *for*-quantified tuple variable to the merge operator having B_i as input;
7. creates a *merge specification* inside the merge operator (if it not already exists) and adds a *merge group* (see below) containing tuple variable references to all positions providing information generated by the split operator (via tuple variable T_i in B_i);
8. creates a projection specification inside the merge operator (if it not already exists) and adds tuple variable references to the positions of information *not* generated by the split operator (via the other tuple variable in B_i);

When all correlated expressions are rewritten, the rule checks whether T was referenced in S before the rewriting. If so, it adds another tuple variable reference to the projection specification of the merge operator referencing the *outer* tuple variable. Then, it replaces the input of T with the merge operator and finally, it rewrites select operator S. For every dependent tuple variable: if $T_i^x = T_i^D$, the complete subexpression of dependent tuple variable T_i^D was pulled out and T_i^D can be removed. Tuple variable references R_i to T_i^D (R_1 and R_2 in our example) have to be rewritten to T, resulting in references R_i'. Note, the positions of these tuple variable references have to be adjusted to the right values (therefore, the position in R_1' in our example was modified to 1). If $T_i^x \neq T_i^D$ for any T_i^D in S, only a part of the subexpression was pulled out. Tuple variable T_i^D cannot be removed. However, because the input of T_i^x was removed, the resulting information generated by select B_i (containing the input of T_i^x) is provided using an (external) tuple variable reference to T. This external tuple variable reference also has to access the "right" position (similar to the R_i').

As you can observe, the first part of the rewriting is quite similar to the Boolean split rule. In the second part, however, S is not completely removed as in the Boolean split rule, but its input at tuple variable T is redefined. Thus, tuple variables are removed from S and all tuple variable references inside S have to be adjusted to the new input positions.

An issue so far not discussed is the concept of a *merge group*. Generally, the merge specification defines which items in the input tuples have to have the same value. If the input generated by the split operator consists of singleton tuples (as in our example), there is exactly one field defining the comparison value in each input stream. However, if the split input consists of n-ary tuples (as it happens in the general case), n fields define n comparison values in each input stream. A merge group allows to define which fields have to be compared in each stream. Therefore, a merge group is a list of tuple variable references. This list specifies for each input stream (i. e., for each output generated by a B_i), which position has to be compared. Our example defines only one merge group with tuple variable references to positions 0, 1, and 1 in tuple variables F:15, F:19, and F:23.

After rewriting, the rewritten tuple variables in select operators B_i have at most one correlated edge as desired. Tuple variable T in the rewritten version may, however, still have correlated edges. These edges cannot be rewritten (and thus, cannot be unnested) in the current version of the query processor. Their treatment is left open for future work. Again, although the rule does not unnest the query, it facilitates unnesting and is therefore added to the first category on Page 93. At the end of this section, the merge operator shall be introduced the same way the other XQGM operators were introduced in Chapter 4.

The Merge Operator

The syntax of the merge operator extends the existing XQGM syntax by the introduction of a merge specification (`MERGE_SPEC`). A merge specification displays one or more merge groups. A merge group is a list of tuple variable references. If n merge groups are defined, then the ith entry of each merge group is attached to an oval component carrying an equal sign. This component is called *group root* in the following. The connections between the group root and the tuple variable references declares, which values have to be equal during the merge. The projection specification, the optional sorting specification, and the optional predicate of a merge operator have the same syntax as in select operators.

To define the semantics of the merge operator, we again need to specify the *set*, *map*, and *eval* functions. We start with the *map* function: A merge operator M with a projection specification $X = M.proj\_spec$, a sorting specification $U = M.sort\_spec$, a predicate $F = M.predicate$, a merge specification $G = M.merge\_spec$ and n tuple variables $T_1, \ldots, T_n$ is mapped to a logical algebra (LAL) expression as follows:

$$
\begin{aligned}
&map(M) = \\
&\quad DDOCPCS_{[X.ddo?,X.cp?,X.cs?]}(\\
&\qquad PROJECT_{[set(X.expression[1]),\ldots,set(X.expression[n])]}(\\
&\qquad\quad SORT_{[U.modifiers,set(U.expression[1]),\ldots,set(U.expression[n])]}(\\
&\qquad\qquad SELECT_{[set(F.expression)]}(\\
&\qquad\qquad\quad SELECT_{[set(G.group\_root[1],\ldots,G.group\_root[n]))]}(\\
&\qquad\qquad\qquad TUPGEN_{[T_1,\ldots,T_n]}(\\
&\qquad\qquad\qquad\quad map(T_1 \rightarrow operator_1), \\
&\qquad\qquad\qquad\quad \ldots, \\
&\qquad\qquad\qquad\quad map(T_n \rightarrow operator_n)))))))
\end{aligned}
$$

The structure of this mapping is almost the same as for the select operator (presented on Page 55 in Section 4.2.4). The only difference is the extra select operator handling the group roots. The rationale behind this mapping is the following: Because all tuple variables are *for*-quantified, the tuple generator creates the Cartesian product of all input tuples. The merging specification is a special selection predicate over this Cartesian product. It selects all tuples, whose values on the positions defined by the merging groups are equal. On the merged stream, the further components of the merge operator can be evaluated as in an ordinary select operator. Note, of course, the Cartesian product can become very large, if the merge operator would be implemented as suggested by this mapping. However, because we are at the logical level in this discussion, we do not care about this issue. A physical implementation of the merge operator would exploit the fact that the input of the split operator is in distinct document order, i. e., every input tuple is unique. Therefore, it is possible to attach a dynamic tuple identifier TID to each split input tuple and implement the tuple comparison based on the TID.

Listing 5.1 The a) *set* function and the b) *setGroup* function for group roots

a) *set*

```
    Input: Array of XQGM Group Roots E
    Output: LAL Expression R
 1  begin
 2  |   R ← setGroup(E[1]);
 3  |   for i in 2 to |E| do
 4  |   |   R ← R ∧ setGroup(E[i]);
 5  |   end
 6  |   return R;
 7  end
```

b) *setGroup*

```
    Input: Group Root G
    Output: LAL Expression R
 1  begin
 2  |   LAL Expression R ← set(G.tupvarref[1]) is set(G.tupvarref[2]);
 3  |   for i in 3 to |G| do
 4  |   |   R ← R ∧ set(G.tupvarref[i]) is set(G.tupvarref[i − 1]);
 5  |   end
 6  |   return R;
 7  end
```

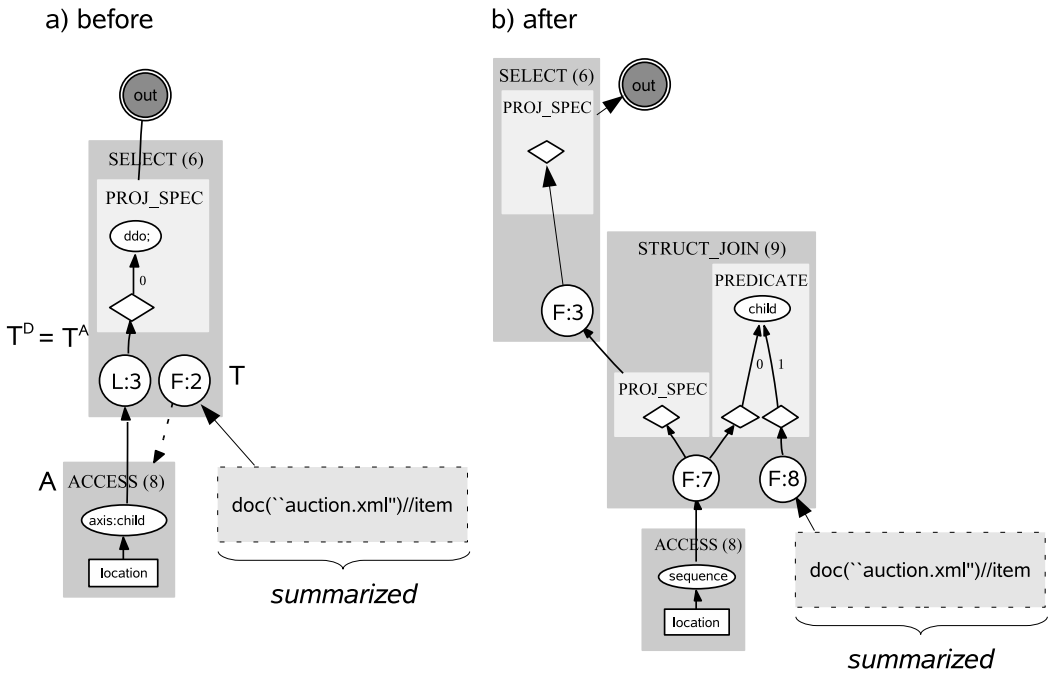
Because all the logical operators in this mapping are already defined, we do not need to provide an *eval* function. Nevertheless, note that the concept of outer tuple variables has been introduced above. Tuple variables fall into the responsibility of the *TUPGEN* operator. Therefore, the *eval* method of this operator has to be adjusted to support outer semantics. The necessary modifications to the operator's code are straightforward and not explicitly shown here.

The final missing piece is the *set* function. So far, it is only defined on XQGM expressions (see Listing 4.3 on Page 62). In the above mapping, however, *set* was applied to the group roots of the merge specifications. These group roots are no XQGM expressions. Therefore, to define the input of the inner LAL select operator, the *set* function has to be overloaded to generate a LAL expression. Listing 5.1a shows the overloaded *set* function. The *set* function creates a chain of conjunctions based on the results returned by the *setGroup* helper function (shown in Figure 5.1b) on each group root. The *setGroup* function is based on the XQuery node comparison operator '*is*' to express equality.

5.7.3 The Unnesting Rule

The *unnesting rule* introduced in this section can transform structural subexpressions into structural joins. A structural join is a special operator, for which—similar to the relational join operator—many efficient evaluation algorithms have been proposed in the literature, e. g., [Al-khalifa 02, Chien 02, Mathis 06a]. Because large performance differences can be expected between the node-at-a-time processing style implied by structural subexpressions and the bulk evaluation style of unnested queries, the *unnesting rule* is quite important. As stated in the introduction of this section, the rule can operate on tuple variables having *exactly one* correlated edge leading to an access operator. The previous two rules were introduced to dissolve tuple variables with multiple correlated edges into this simpler form.

Figure 5.8 The simplest unnesting scenario



Examples

To illustrate the features of the unnesting rule, we consider more than one example. Let us start with a simple one. On the left hand side of Figure 5.8a, you can see a simple select operator with an input of *item* tuples, for each of which an access operator is evaluated. The access operator returns the child *location* elements. The input to the access operator is given via a correlated edge from tuple variable F:2. On the right hand side, the correlated edge as well as the tuple variable have been removed. Because the correlated edge is missing, the access operator does now not have any "context" anymore: there is no current *item* node providing the starting point for the navigation calculated by the access operator. The only thing the access operator can do, is to return all *location* elements. To indicate that an access operator returns all elements fulfilling a certain node test, the oval component carries the keyword "sequence". The operator is then called a *sequence access operator*. The output of the sequence access operator is the sequence of all *location* nodes in document order. The sequence is given to a new type of operator, namely the *structural join* operator. This operator receives two input node sequences and joins them on a structural predicate, the input nodes have to fulfill. In our example, the structural predicate defines that the location nodes have to be children of the item nodes. The projection specification of the structural join specifies that only the location elements have to be returned. Thus, the operator is actually a *structural semi-join operator*. The location elements are given to the original select operator. Note, this operator now has no "ddo" output modifier anymore. On the left hand side, its task was to ensure that the result is in document order and duplicate-free. Because we expect this property directly from the output of the structural join operator, the "ddo" output modifier could be removed.

As a second example, consider the following query, whose initial XQGM instance is shown in Figure 5.9a:

```
doc("auction.xml")//item/location[2]
```

For each *item*, the subtree below L:4 is evaluated. The access operator does not only produce all child *location* element of an item, but also the location's context position ("cp"). In `select(8)`, a predicate is evaluated on the generated position. Only tuples with position 2 qualify. The *location* elements of the qualified tuples are passed on to the final select operator, which simply applies output modifier "ddo". As an example run, let us assume that item i_1 has two location children l_1 and l_2 and that item i_2 has children l_3 and l_4. Then, for i_1, the access operator returns the two tuples [l_1, 1] and [l_2, 2]. Only the second tuple fulfills the positional predicate and l_2 is returned. Similarly, for i_2, location l_4 is returned.

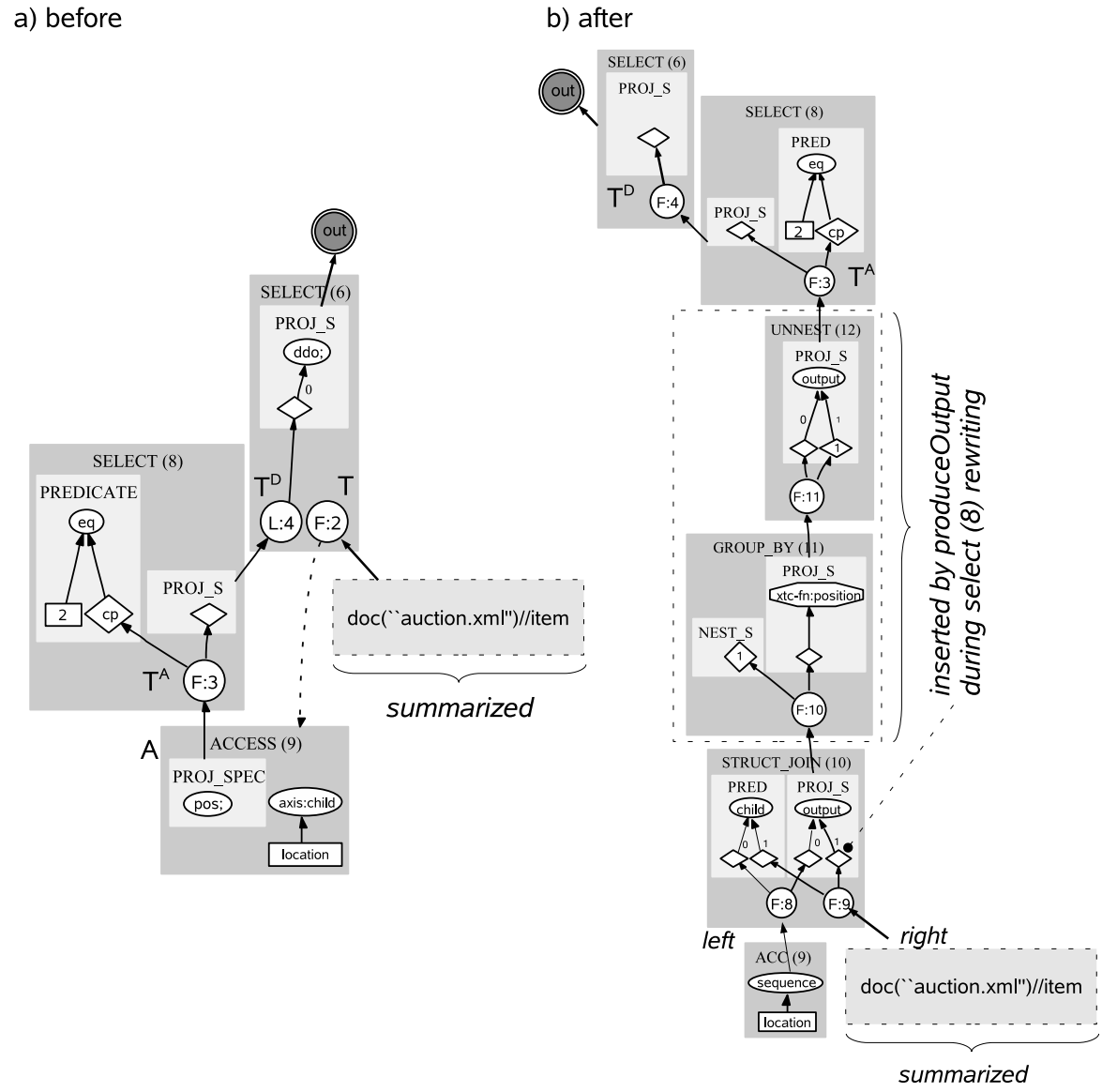
Unnesting a query with a positional predicate is slightly more complex than the previous example. Below L:4, some actions (adding the "cp" information and filtering) are carried out for the *group* of all the *location* elements which are the children of a certain *item*. The group is induced by the correlated edge, which provides each *item* as a kind of group "context". When the correlated edge is removed, the group information is lost, because the join replacing the edge simply delivers pairs of matches, but no groups. Therefore, to correctly attach the correct context position information to each location, grouping is required. Grouping is expressed by the *group by* operator in XQGM.

To understand the rewriting in Figure 5.9b, let us consider the above example with i_1 and i_2 and their children l_1 to l_4 again: The structural join operator, which was inserted to remove the correlated edge, returns the following tuple stream: [l_1, i_1], [l_2, i_1], [l_3, i_2], [l_4, i_2]. In contrast to the structural join operator inserted during the previous rewriting, this is not a semi-join, because in the projection specification, both tuple variables are referenced.

On the intermediate result, the task is now to attach the "cp" information. However, because unnesting destroyed the "for each item" groups, we have to group the stream again. This happens inside the *group by* operator. The nesting specification NEST_SPEC of the *group by* operator defines, which tuple field(s) *induce* the group, i. e., which fields have to have the same value. In our example, this is at position 1. Thus, all tuples having the same item on the second field form a group: [<[l_1], [l_2]>, i_1], [<[l_3], [l_4]>, i_2]. The projection specification of the *group by* operator contains a function, called *xtc-fn:position*. The prefix indicates that this function is an XTC-internal function. It receives a sequence of nodes and attaches the context position to each node. In our example, the projection specification returns the following sequence of tuples: [<[l_1, 1], [l_2, 2]>], [<[l_3, 1], [l_4, 2]>]. Note, because the *item* field was not referenced in the projection specification, no items are contained in this intermediate result anymore.

The next task is to filter all tuples with context position 2. On the previous intermediate result, this is, however, not directly possible, because the tuples are still grouped. Therefore, they have to be "ungrouped". This is the task of the *unnesting* operator. Similar to the *group by* operator, the unnest operator contains a nesting specification, defining the fields that need to be "copied". Unnesting is the reverse operator of *group by*. As an example, consider unnesting intermediate result [<[l_1], [l_2]>, i_1], [<[l_3], [l_4]>, i_2]. The nesting specification

Figure 5.9 Unnesting over positional predicate

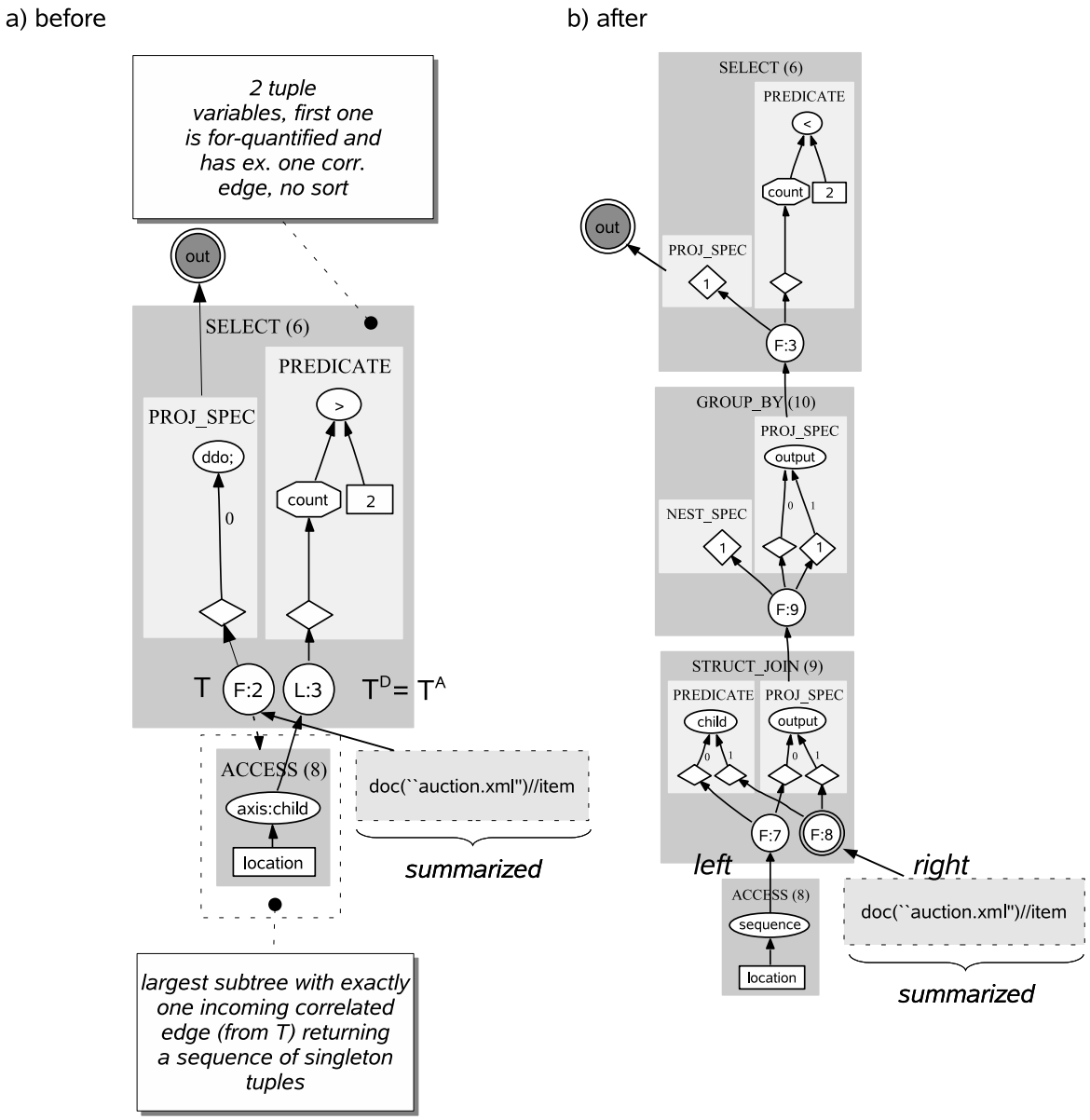


of the unnest operator would contain a reference to position 1 (i. e., to items i_1 and i_2), resulting in the unnested sequence $[l_1,\ i_1],\ [l_2,\ i_1],\ [l_3,\ i_2],\ [l_4,\ i_2]$. The intermediate result generated by the *group by* operator, however, only contains groups of *location* elements, but no *items* anymore. Therefore, the nesting specification of the unnest operator is empty. This leads to the following result, where the groups are simply "flattened": $[l_1,\ 1],\ [l_2,\ 2],\ [l_3,\ 1],\ [l_4,\ 2]$. The remainder of the query is evaluated as before. The sequence is filtered on the position and locations l_2 and l_4 are returned.

As a third and last example, consider the following query, whose initial XQGM instance is shown in Figure 5.10a:

```
doc("auction.xml")//item[count(location) < 2]
```

Figure 5.10 Unnesting with group semantics



On the left hand side, the group of *location* elements is calculated for each input item. These groups are then passed to the predicate in the select operator. If the number of *location* elements in a particular group is smaller than 2, the *item* element is returned. Otherwise, the *item* element is skipped. Obviously, the *location* elements are grouped by the *item* elements. If we want to get rid of the correlated edge, we have to consider this group semantics again. A similar situation already occurred in one of our examples earlier: The multiple correlated expression pull-out pattern leaves queries in a state, where group semantics have to be regarded. In Figure 5.7b on Page 109, operators `select(15)` and `select(17)` have a tuple variable with a correlated edge. Their projection specification returns information from both tuple variables, where the intermediate results delivered by one subtree (e. g., the

location elements from `access(11)`) are grouped by the *item* input.

After the correlated edge has been unnested to a structural join, the grouping operator can reconstruct the necessary groups again. Let us consider the rewriting in Figure 5.10b and our example scenario with i_1, i_2, and their children l_1 to l_4 again. As before, the structural join returns sequence [l_1, i_1], [l_2, i_1], [l_3, i_2], [l_4, i_2]. Note, an *item* has to be passed on, even if it does not have a *location* child, because otherwise the *count*-based predicate would not be evaluated on that item and the result would be not correct. Therefore, the structural join has to be *outer*. Again, this fact is indicated by a double circle around the *outer* tuple variable (as in the merge operator).

In the next step, the *group by* operator nests the *location* elements, resulting in sequence [<[l_1], [l_2]>, i_1], [<[l_3], [l_4]>, i_2]. Note, in contrast to the previous example, no grouping function is applied and both tuple fields are returned (*items* are required as final output). Then, operator `select(6)` can test the predicate against the first field of each tuple (consisting of a sequence of location elements). If the sequence size is smaller than 2, the first field containing the *item* element is returned. In our example, this is never the case.

Finally, let us consider what happens during the application of the unnesting rule on the example presented in Figure 5.7b on Page 109: In each branch, the unnesting pattern is applied, which introduces a structural join operator. Because of the group semantics, a *group by* follows the structural join. Operators `select(15)` and `select(17)` are rewritten to simple projections which are merged into the *group by* operator by the select fusion rule.

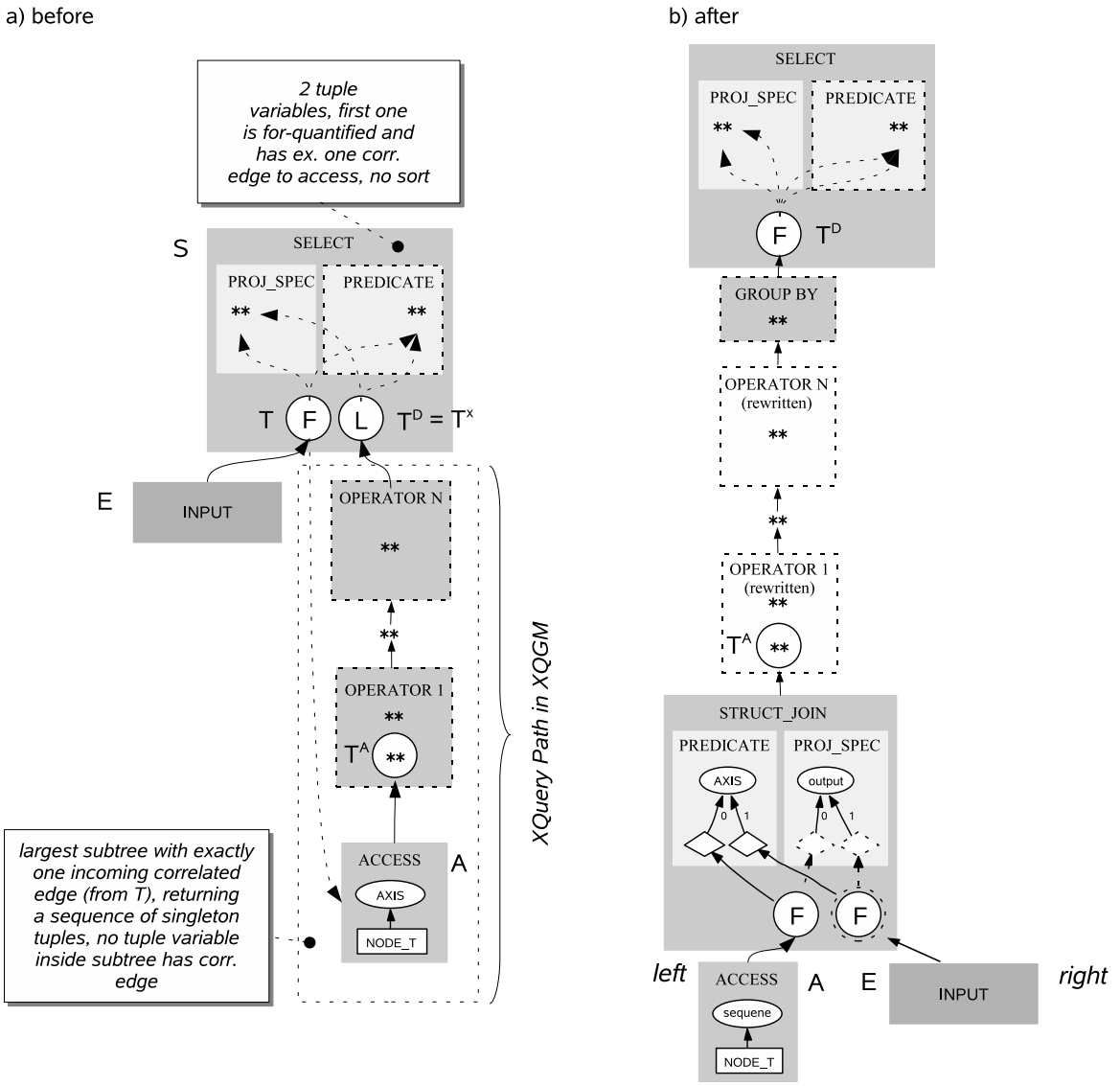
The Pattern

The pattern of the unnesting rule is sketched in Figure 5.11a (note, dashed components and arrows are optional; the double star (**) stands for any set of components). It searches for a select operator S with no sorting specification and two tuple variables, of which the first one (T) is *for*-quantified and has exactly one correlated edge to an access operator A. The access operator resides inside a subtree below dependent tuple variable T_D, which also belongs to S. Similar to the pattern of the multiple correlated expression pull-out rule, this rule examines the subtree below T^D. As above, we are searching for a tuple variable rooting an XQGM-transformed relative path expression starting at the correlated access operator. Again, navigating up the subtree (at most until T^D is reached), the rule searches for the largest subexpression with the following properties: the current subtree has exactly one incoming correlated edge (and this edge starts at T), no tuple variable in the subtree has a correlated edge (i. e., the subtree is already unnested), and the subtree produces a sequence of singleton tuples as result. The tuple variable rooting this subtree is called T^x. In contrast to the pull-out rule, we demand that T^x is equal to T^D. This means that unnesting is only possible over XQuery path expressions.

The Transformation Instruction

The result generated by the transformation instruction is sketched in Figure 5.11b. As you can see, the correlated edge is rewritten to a structural join, with the original input E of T and the rewritten access operator A as input. In contrast to the nested version, all operators between A and T^D now "see" the information generated by E, because they possibly receive this information now as input. To keep the

Figure 5.11 The unnesting pattern and the result of the transformation instruction



query semantics correct, these operators have to be rewritten. Sometimes more than one operator is generated for the one to be rewritten (therefore, the dashed boxes are represented in white color). Finally, also the original select operator S needs to be rewritten, because now, it only contains a single tuple variable and also the input generated by E flows over this tuple variable. Sometimes this input has to be "adjusted" by a grouping operator. After this introduction, we now delve into the details. The actions necessary are grouped into analysis, structural join creation, operator path rewriting, and select rewriting.

Analysis

The rule first analyzes the select operator S matched. The result is captured in the following Boolean flags, which influence the following rewriting process:

- `forGeneratesOutput` is true, when T is referenced in the projection specification, i. e., when the left-most arrow in Figure 5.11a starting at T exists. For an example, see `select(15)` in Figure 5.7b on Page 109.
- `hasExistentialPredicate` is true, when the predicate exists and if it contains a simple tuple variable reference, for example, as `select(6)` in Figure 5.3 on Page 98.
- `hasPredicate` is true, when S contains a predicate, as `select(18)` in Figure 5.6 on Page 106 .
- `forReferencedInPredicate` is true, when T is referenced in the predicate, as in `select(8)` of Figure 5.3 on Page 98[4]. This flag corresponds to the second arrow starting at T leading into the predicate.
- `hasComplexProjSpec` is true, when the projection specification does not solely consist of one simple tuple variable references, as in `select(1)` of Figure 5.7a on Page 109.

Structural Join Creation

Based on these settings, let us consider the creation of the structural join operator. The first input of this operator will be access operator A; the second input will be the input E of T. We will refer to the first input as *left* and of the second input as *right* in the following. The default type of the structural join operator to be created is *full join*. In a full join, both tuple variables are referenced in the projection specification. Other join types depend on some of the five flags above, as follows:

- If `forGeneratesOutput` or `forReferencedInPredicate` is true, the join type is *right outer*. In a right outer join, the right input (which originally was the input of T) will be passed on, even if the left input did not return any results. The value of the left input is then the empty sequence. This situation occurs in our third example, shown in Figure 5.10.

  This semantics may not lead to a correct result, when `forReferencedInPredicate` is false and `hasExistentialPredicate` is true. In case of an existential predicate on the left input, no tuples may be passed on from the right input, when the left input does not generate any results. Therefore, in this case, the join is not right outer. This situation occurs in Figure 5.3b on Page 98. Note, in this example, a right outer join would not yield a false result, because there is still the content-based predicate that would filter tuples with empty *quantity* elements. Without the content-based predicate, however, the rewriting would not be correct.
- Else, if `forGeneratesOutput` and `hasExistentialPredicate` are true, `forReferencedInPredicate` is false, and T^A is T^D, the join type is *right semi*. In this case, the left input is not required for output, because it just serves as an existential predicate. For this situation, no previously shown example query exists.
- Else if `forGeneratesOutput`, `forReferencedInPredicate`, and `hasPredicate` are false, the join type is *left semi*. In this case, only the left input is required as output. This is the case in our first example, shown in Figure 5.8.

[4]Note, this particular select operator cannot be unnested, because it has no correlated edge. It was referenced here only to make the meaning of the Boolean flag clear.

Having defined the join type, the structural join can be instantiated. The left input is access operator A. Therefore, A is detached from its containing tuple variable T^A and is transformed to a sequence access operator. The right input of the structural join operator is the subtree below T. The structural join operator is set as the input of tuple variable T^A.

Operator Path Rewriting

In the next step, all operators "on the operator path between" tuple variable T^A (which originally contained access operator A) and T^D have to be rewritten. As we will see, the projection specification of A also plays a role in this rewriting. In our first example (Figure 5.8), there are no operators to rewrite, because T^A and T^D are the same variable. Nothing has to be done here. In our second example (Figure 5.9), all operators between `F:3` and `L:4` have to be rewritten, i. e., `select(8)`. In the third example (Figure 5.10), again T^A is equal to T^D. However, rewriting is necessary here, because the access operator generated positional information, which needs to be specially handled. In general, the operator chain between T^A and T^D describes an unnested, relative path expression (as defined by the pattern above). Therefore, we have to be able to rewrite any operator that can occur in this already unnested path expression (e. g., *group by*, unnesting, select, structural joins, split, merge, and set-based operators). Note, a repeated application of the unnesting rule frequently leads to the situation that unnesting is executed over an operator path, over which unnesting has been applied before. As a result, it also has to be possible to apply unnesting over our sample results in Figures 5.8b, 5.9b, and 5.10b (because these are already unnested path expressions). We have to keep this important consideration in mind to correctly define the rewriting semantics.

Rewriting the operator path is executed by an iteration over all operators in a bottom-to-top manner. Every operator is rewritten separately, resulting in one or more modified/new operators. The following two Boolean flags are initialized in each iteration step. Besides the five flags from above, they also influence the operator's rewriting:

- If `posInfo` is true, the previously rewritten operator generates the context position information. The flag is initialized with the information from the projection specification of access operator A. For example, in Figure 5.9a, the flag is set to true.

- If `lastInfo` is true, the previously rewritten operator generates the context size information. Again, projection specification A provides this information initially. After both positional flags have been initialized, the corresponding information is deleted from A's projection specification.

Finally, if flag `forwardAxis` is true, access operator A evaluated a forward axis. We now discuss the various rewritings depending on the setting of the introduced flags.

Case 1: The most simple situation occurs, when the information generated by T is neither required in the projection specification nor in the predicate, and when no positional information is required, i. e., `forGeneratesOutput`, `forReferencedInPredicate`, `posInfo`, and `lastInfo` are all false. Then, the operators do not need to be rewritten at all, because the information generated by the right input does not influence the operators: they still only "see" the left input.

Case 2: A more complex situation arises, when either `forGeneratesOutput` or `forReferencedInPredicate` is true, i. e., when the right output is required in select S (and still `posInfo` and `lastInfo` are false). In this case, the structural join operator was instantiated as full join or as a right outer join before. The remaining operators have to be rewritten in a way such that they "loop the right input through". Basically, this is achieved by adding tuple variable references to the operators. Furthermore, in the case when the structural join operator was instantiated as right outer, we have to assure that the right input is delivered to T^D. If the operator path contains further structural joins, they are also set to right outer. For the sake of brevity, we do not explicitly provide the semantics of this rewriting here, because adding tuple variables without altering further operator semantics is quite straightforward. In the following, we assume a rewriting function *produceOutput*. This function is applied in every iteration to an operator on the operator path, resulting in a rewritten representation that delivers the requested output.

Case 3: Finally, we consider the situation, when during the iteration `posInfo` is true or `lastInfo` is true. This means that the current operator to be rewritten is a select operator and the projection specification of the previous operator generated positional information (which will be checked in this select operator). An example for this scenario is shown in Figure 5.9, where `select(8)` has to be rewritten. In the example, the unnesting rule created a left-semi structural join with A and T's subtree as input so far (shown in Figure 5.9b). The structural join is directly attached to tuple variable F:3 (T^A), i. e., the operators in the dashed box are not yet there.

The first action rewrites the subtree below T^A in a way such that T^A receives the results generated by the right input (if not already the case, due to previous applications of the unnesting rule). This is again achieved by the *produceOutput* function, which is applied to the structural join operator, thereby adding a tuple variable reference (as indicated in Figure 5.9b).

In the next step, the input (structural join operator) of T^A is detached, because further operators will be inserted at this position. First, a *group by* operator is created. It has exactly one *for*-quantified tuple variable and receives the structural join as input. Because we group by the right input, the nesting specification of the *group by* operator receives a tuple variable reference for each field generated for the right input in the structural join. In our example, this is just one reference to position 1. The actual task of the grouping operator here is to reconstruct the context and to generate the necessary positional information, as explained during the discussion of the second example. As we have seen, generating positional information is done by special XTC-specific functions. These are: *xtc-fn:position*, *xtc-fn:reverse-position*, and *xtc-fn:last*. The input to all these functions is a tuple variable reference to the field generated by the left input of the structural join. In case `posInfo` is true, either *xtc-fn:position* or *xtc-fn:reverse-position* is generated, depending on the value of `forwardAxis`. The second function has similar semantics to the first one. The only difference is that *xtc-fn:reverse-position* attaches the positional information in reverse order to the tuples of an input sequence. Thus, the function makes the special "reverse semantics" of positional predicates on reverse axes possible. If both, `posInfo` and `lastInfo` are true, the functions are simply chained. The grouping operator is now complete and the required unnest operator can be instantiated.

The unnest operator does not require a nesting specification, because only nested results (groups) are passed on from the *group by* operator. The task of the unnest

operator is then to get rid of the groups, which automatically happens, when no nesting specification is given. The projection specification has to reference all necessary fields in the unnested tuple sequence. The number of required tuple variable references again depends on `posInfo` and `lastInfo`. If both are true, three references are required, otherwise only two. The first reference refers to the input generated by the left expression; the remaining ones reference the positional information generated. The unnest operator is set as input to T^A. The rest of the rewriting is executed in analogy to Case 2, i. e., depending on `forGeneratesOutput` and `forReferencedInPredicate`.

Select Rewriting

Finally, select operator S itself has to be modified to correctly process the input returned from the rewritten subtree. If `hasComplexProjSpec` is true or `hasPredicate` is true and `hasExistentialPredicate` is false, a grouping operator is required. The rationale lies in the semantics before the rewriting: for every input delivered by E, the predicate and the projection specification have been evaluated. This has also to be true after the rewriting. We have already discussed this requirement in our third example (see Figure 5.10). To assure group semantics, the right input is delivered to T^D and a *group by* operator has to be inserted. The rewriting for input delivery is either already completed (due to the operator path rewriting) or it has to be achieved. In the latter case, the *produceOutput* function is applied to the operator path as before. The new *group by* operator receives the operator path as input. It nests by the tuple fields generated for the right input, and also returns these fields, if `forGeneratesOutput` or `forReferencedInPredicate` is true. The *group by* operator, in turn, is attached to T^D.

In Figure 5.11b, you can observe that now, the information generated by both subtrees is delivered by one tuple variable (instead of two as before). Therefore, if `forGeneratesOutput` or `forReferencedInPredicate` is true, all tuple variable references to T have to be rewritten to reference T^D. Finally, if an existential predicate has been rewritten to a semi-join before, the predicate can be deleted. To finish the transformation instruction of the unnesting rule, tuple variable T is removed.

This unnesting rule belongs to the first and the fourth category of the ones shown on Page 93: it unnests and, thereby, facilitates the mapping onto physical structural join operators. To complete this section about unnesting, the three newly introduced operators have to be embedded into XQGM. These operators are the structural join, the *group by* operator, and the unnest operator. We start with the first one.

The Structural Join Operator

Conceptually, the structural join operator is a special type of select operator. The only extension is the new predicate type, because a structural join has a structural predicate. Of course, the items referenced by these predicates have to be nodes. On these nodes, the predicate has to decide the structural relationship of one of the 13 XPath axes. Because this is straightforward, we do not specify structural predicates in more detail. The *map*, *set*, and *eval* functions do not need to be modified for the structural join operator, because they are the same as for the select operator.

The Group By Operator

Syntactically, a *group by* operator has exactly one *for*-quantified tuple variable, a nesting specification (`NEST_SPEC`), and a projection specification. The nesting specification contains a list of tuple variable references. To define the semantics of the operator, we again need to define the *set*, *map*, and *eval* functions: A *group by* operator G with a projection specification $X = G.projectionSpecification$, a nesting specification $N = G.nestingSpecification$, and one tuple variables T is mapped to a logical algebra (LAL) expression as follows:

$$
\begin{aligned}
&map(G) = \\
&\quad DDOCPCS_{[X.ddo?,X.cp?,X.cs?]}(\\
&\quad\quad PROJECT_{[set(X.expression[1]),\ldots,set(X.expression[n])]}(\\
&\quad\quad\quad GROUP\_BY_{[set(N.ref_1),\ldots,set(N.ref_n])}(\\
&\quad\quad\quad\quad SORT_{[M,set(N.ref_1),\ldots,set(N.ref_n])}(\\
&\quad\quad\quad\quad\quad map(T \rightarrow operator)))))
\end{aligned}
$$

Note, we used "*ref*" to abbreviate "tuplevariable_reference". The projection operator and the DDOCPCS operator are already introduced. They handle the projection specification. Their input is provided by a LAL GROUP_BY operator, whose semantics will be defined below. The tuple variable references are simply translated by the *set* function to LAL tuple access operators, which are given to the grouping operator as parameters. Therefore, the *set* function has the standard semantics (defined in Listing 4.3 on Page 62).

The semantics of the LAL GROUP_BY operator is given by the algorithm in Listing 5.2. The operator only works on inputs sorted on the fields referenced by the nesting specification (sort-based group-by implementation). This behavior is sufficient in most cases, because, due to the `ddo` function and the generation of ordered streams by structural join operators, the input streams are usually ordered. If this is not the case, a special SORT operator is inserted. The sort operator reorders the tuples such that the values of the tuple variable references given (i. e., $ref_1, \ldots, ref_n$) occur in ascending order. The sort modifier M, therefore, contains n "*ascending*" modifiers. Because the *group by* operator is sort-based, it simply consumes the input tuple stream and collects all tuples with the same value on the fields referenced by the nesting specification. As soon as this values changes, the *group boundary* has been reached and a new group starts.

For the discussion of the algorithm in Listing 5.2, let us assume an example input: operator O in line 2 of the main algorithm (Listing 5.2) returns the following tuples in a sequence S: $[a, a, b]$, $[a, a, c]$, $[a, b, c]$, $[a, b, d]$, and $[a, c, x]$. We furthermore assume the nesting specification to refer to positions 0 and 1, i. e., the first two tuple fields are checked for equality. We also call these fields *group definition fields*. Obviously, S is not empty, therefore, *initGroup* is called on the first tuple of sequence S in line 8. This function is a helper function with a multi-valued return value (returning TupleSequence I and Tuple r). The function is shown in Listing 5.3. Its task is to initialize a tuple r containing the group definition fields (which always have the same value for each group) and a sequence I containing the first tuple of the new group. In our example, *initGroup* is called with tuple $[a, a, b]$. At first, local variables for I and r are defined. Then, another variable q for the first group tuple is instantiated. Then, the algorithm iterates over all fields of the given tuple. If the field is referenced in the set of access operators (i. e., in the nesting specification), it is appended to r (containing the group definition fields), otherwise, it is appended

Listing 5.2 The main algorithm of the GROUP_BY operator

$eval(GROUP\_BY_{[A_1,...,A_n]}(O))$

```
Input: Parameters: TupleAccessOperators A ← A_1, ..., A_n, Arguments: Operator O
Output: TupleSequence R
 1 begin
 2     TupleSequence S ← eval(O);
 3     if S is empty then
 4         return ();
 5     end
 6     Tuple r; // intermediate result tuple
 7     TupleSequence I; // intermediate group tuple sequence
 8     (I, r) ← initGroup(A, S[1]);
 9     for int i ← 2 to |S| + 1 do // the remaining tuples in the sequence
10         if i = |S| + 1 or newGroup(S[i], S[i − 1]) then
11             // finalize current group and init new one
12             Tuple x ← [|r| + 1];
13             x ← x + I; // set group sequence as first tuple field
14             x ← x + r; // set group definition fields
15             R ← R + x; // add to result sequence
16             if i ≠ |S| + 1 then
17                 (I, r) ← initGroup(A, S[i]);
18             end
19         else
20             Tuple t ← S[i];
21             Tuple q ← [|t| − |A|]; // intermediate group tuple
22             for int j ← 1 to |t| do
23                 if A not contains j then
24                     q ← q + t[j];
25                 end
26             end
27             I ← I + q;
28         end
29     end
30     return R;
31 end
```

to q (the first tuple of the new group). When the iteration is complete, q is added to the intermediate group sequence I, which, in turn, is returned together with r. The result of *initGroup* in our example is $r = [a, a]$ and $I =$<[b]>.

Listing 5.3 The *initGroup* algorithm

```
Input: TupleAccessOperators A, Tuple t
Output: TupleSequence I, Tuple r
 1 begin
 2     I ← (); // intermediate group tuple sequence
 3     r ← [|A|]; // intermediate result tuple
 4     Tuple q ← [|t| − |A|]; // intermediate group tuple
 5     for int j ← 1 to |t| do // for each index to a tuple field
 6         if A contains j then
 7             r ← r + t[j]; // append to intermediate group definition tuple
 8         else
 9             q ← q + t[j]; // append to intermediate group tuple
10         end
11     end
12     I ← I + q; // add to intermediate result sequence
13     return (I, r); // returned combined result
14 end
```

The second helper function in Listing 5.4 can detect new groups in the sorted input

streams. It simply compares the values of the group definition fields of the current tuple with the corresponding fields from the previous tuples. If at least one field does not match, a new group is detected.

Back in the main algorithm, the remaining tuples from S are consumed. If the end of the tuples is reached ($i = |S| + 1$) or if a new group is detected by function *newGroup*, the tuple for the current group has to be finalized and, in case of the existence of subsequent tuples, a new group has to be started. This happens in lines 11 to 17. Note that the grouped field is written to the first position in the resulting tuple. In our example, the second tuple, $[a, a, c]$, does not trigger the creation of a new group, therefore $[c]$ is added to I, resulting in $I =$ <[b], [c]>. Then, the third input tuple, $[a, b, c]$ triggers a new group, because the second position of this tuple is different to the previous one. Therefore, combined tuple [<[b], [c]>, a, a] is written to the result sequence R, I is initialized to <[c]>, and r is [a, b]. After the fourth tuple has been consumed, the last group is detected, and tuple [<[c], [d]>, a, b] is written to R. The last group then has the following value: [<[x]>, a, c].

Listing 5.4 The *newGroup* algorithm

```
   Input: TupleAccessOperators A, Tuple t1, Tuple t2
   Output: boolean R
 1 begin
 2     for int j ← 1 to |t| do
 3         if A contains j and t1[j] ≠ t2[j] then
 4             R ← false;
 5             return R;
 6         end
 7     end
 8     R ← true;
 9     return R;
10 end
```

The Unnest Operator

Syntactically, the XQGM unnest operator has a similar structure as the grouping operator: Unnest operator U consists of a single tuple variable T, a nesting specification[5] $N = U.nestingSpecification$, and a projection specification $X = U.projectionSpecification$. No other components are allowed. The nesting specification can contain at most one tuple variable reference. This reference points to the field containing the nested value to be unnested. The unnest operator is mapped to a LAL operator as follows:

$$
\begin{aligned}
&map(U) = \\
&\quad DDOCPCS_{[X.ddo?,X.cp?,X.cs?]}(\\
&\quad\quad PROJECT_{[set(X.expression[1]),\ldots,set(X.expression[n])]}(\\
&\quad\quad\quad UNNEST_{[set(N.ref)]}(\\
&\quad\quad\quad\quad map(T \rightarrow operator))))
\end{aligned}
$$

The mapping is quite similar to the GROUP_BY mapping and, therefore, does not need further explanation. In Listing 5.5, the unnesting algorithm is shown. To facil-

[5] ... although this "nesting specification" belongs to an unnest operator, we used the same name as for the grouping operator here.

itate comprehension, we resume the previous example: let O deliver the following tuple sequence: [<[b], [c]>, a, a], [<[c], [d]>, a, b], and [<[x]>, a, c] to variable S. Furthermore, let the nesting specification point to the first field in the tuple. The given tuple sequence is not empty. Therefore, the return statement at line 4 is skipped. If A is undefined and the size of the input tuples is exactly one, no "real" unnesting is required. We only have to dissolve the groups, as motivated during the discussion of the second example in Figure 5.9 on Page 116. This happens in lines 7 to 10. Every sequence in S is simply appended to the result sequence, which is then returned.

In case, the unnesting position referenced by A is defined, we have to generate unnested tuples. Before, however, we have to calculate the size of the tuples to be instantiated. Therefore, we examine the first tuple t and the first sequence I to be unnested. Variable s_t stores the input tuple size (in our example 3); variable s_g stores the group tuple size (in our example, this is the size of [b], i. e., 1); and variable s_o stores the output tuple size (in our example, 3). Then, an iteration over all tuples in sequence S is executed (line 20 to 32). For each tuple, the nested sequence I is extracted. Then, for each tuple q in this sequence, an output tuple r (of size s_o) is generated as follows:

1. all fields before the unnesting position are copied from t to r;
2. all fields of the current group tuple q are copied to r;
3. all fields after the unnesting position are copied from t to r;

If I is an empty sequence, we nevertheless want to generate an unnested tuple. Therefore, in this case, the positions where the unnested group tuple is stored, is padded with empty sequences (line 33 to 45).

In our example, the first value of t is [<[b], [c]>, a, a] and I is <[b], [c]>. For the first q =[b] in I, tuple r =[b, a, a] is generated; for the second q =[c], tuple r =[c, a, a] is generated, and so on. Finally, we receive the following unnested tuple sequence: [b, a, a], [c, a, a], [c, a, b], [d, a, b], and [x, a, c].

With the discussion of the unnest operator, we now finish this section. The next section shows how opportunities to apply twig pattern matching can be found in a rewritten and unnested XQGM instance.

5.8 Twig Query Detection

So far, the query unnesting rule(s) introduced before are capable of revealing opportunities to apply structural join operators for path evaluation. But what does a structural join operator actually do? Basically, it finds node matches fulfilling a certain structural relationship. Because a structural join operates on two input sequences at a time, it can only find *binary* matches. Therefore, to evaluate a path expression, with, let's say, four steps, three structural joins are required (and possibly some more operators for grouping, unnesting, merging, etc.). The approach to evaluate path queries with structural joins can have some disadvantages: 1) Because path expression occur frequently in XQuery, an unnested XQGM instance often has quite many structural join operators and, thus, many intermediate results are passed; and 2) the intermediate result produced by a structural join may contain tuples that do not contribute to the final result, because some tuples will probably

Listing 5.5 LAL UNNEST evaluation

$eval(UNNEST_{[A]}(O))$

```
    Input: Parameters: TupleAccessOperator A, Arguments: Operator O
    Output: TupleSequence R
 1  begin
 2      TupleSequence S ← eval(O);
 3      if S is empty then
 4          return ();
 5      end
 6      if A is undefined and |S[1]| = 1 then
 7          for int i ← 1 to |S| do
 8              R ← R + S[i];
 9          end
10          return R;
11      end
12      // analyze tuple sizes
13      Tuple t ← S[1]; // fetch the first tuple
14      TupleSequence I ← A(t); // fetch first tuple sequence to be unnested
15      int s_t ← |t|; // input tuple size
16      int s_g ← |I[1]|; // group tuple size
17      int s_o ← s_t − 1 + s_g; // output tuple size
18      for t in S do
19          I ← A(t);
20          for Tuple q in I do
21              Tuple r ← [s_o]; // create new unnested tuple of size s_o
22              for int i ← 1 to (position of A) -1 do
23                  r ← r + t[i]; // append original tuple fields residing before unnest field
24              end
25              for int i ← 1 to s_g do
26                  r ← r + q[i]; // append fields of unnested tuple
27              end
28              for int i ← s_g + 1 to |t| do
29                  r ← r + t[i]; // append original tuple fields residing after unnest field
30              end
31              R ← R + r;
32          end
33          if I is empty then // pad group tuple with empty sequences
34              Tuple r ← [s_o]; // create new unnested tuple of size s_o
35              for int i ← 1 to (position of A) -1 do
36                  r ← r + t[i]; // insert original tuple fields residing before unnest field
37              end
38              for int i ← 1 to s_g do
39                  r ← (); // append empty sequence(s)
40              end
41              for int i ← s_g + 1 to |t| do
42                  r ← r + t[i]; // append original tuple fields residing after unnest field
43              end
44              R ← R + r;
45          end
46      end
47      return R;
48  end
```

be filtered out by following structural joins. [Bruno 02] were the first ones to notice this problem and to provide a solution for it, namely the *holistic twig join* (HTJ) operator. A holistic twig join is an extension of the structural join. It can be parameterized with a tree-shaped pattern—the *twig*—and is able to find all occurrences of this pattern in the document. Thus, it is, in contrast to the structural join, an n-ary operator working on multiple input streams.

In contrast to evaluating a query with structural joins, the holistic twig join operator does not produce an intermediate result tuple until it is clear that this tuple belongs

to the final twig result. This is achieved by checking *all* necessary twig conditions before the tuple is generated for the output, thus the term *holistic*. As we will see in Chapter 8, the twig join operator internally keeps the tuples contributing to a match compactly encoded in a set of stacks. With these characteristics, [Bruno 02] solved the two problems for structural join operators sketched above. After their first publication in the literature, a plethora of further holistic twig join algorithms have been proposed by the research community (see Chapter 8). Some of them extended the "expressiveness" of the twig algorithm, e. g., by adding the capability to match a *not* predicate on a path. Some others improved the matching performance of the algorithm. In this work, a further variant of this operator will be developed. All in all, the holistic twig join is an important operator class and it is worthwile to think about how it can be integrated into our query processing framework.

Revealing opportunities to apply holistic twig join operators is the task of the *HTJ discovery rule* introduced in this section. In contrast to the previously introduced rules, we will state the notion of the XQGM holistic twig join operator before we actually show how it can be used. The twig join operator is quite complex. Therefore, we do not introduce it formally, i. e., by defining the *map*, *set*, and *eval* method. Introducing the operator this way would fill many pages and it would forestall the introduction of the physical counterpart for this logical operator in Chapter 8. Rather, to define the semantics, we rely on a little trick: In Section 5.8.3, we will show how an XQGM subtree can be transformed into a twig operator. The semantics of the twig operator can then be defined by: 1) reversing the actions of transforming an XQGM subtree into a twig operator, i. e., transforming the twig operator back into the originating operators; and 2) relying on the semantics defined for the originating (non-twig) operators. This means, for the definition of the twig operator M, we regard the originating XQGM subtree that generated M as the implementation of M.

5.8.1 The XQGM Twig Join Operator

In our overview (Section 2.2.6 on Page 23), we have already seen a twig structure for the following sample query (which we reuse for the discussion of the twig join operator here):

```
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

An XQGM twig operator (`TWIG`) consists of a set of *for*-quantified tuple variables, a *twig specification*, and of further components of a select operator, i. e., a projection specification, an optional predicate, and an optional sorting specification. The twig operator is, therefore, a special XQGM select operator. The interesting thing is the twig specification. As you can see in Figure 2.5, a twig specification contains a tree composed of several *twig nodes* (represented by oval components) connected via some edges. Generally, i. e., independent of the representation in XQGM, a twig node stands for a *node predicate*, e. g., a name test like *site*, *price*, etc. All XML nodes that fulfill such a node predicate are matched by the particular twig node. The structural relationships between the twig nodes poses another predicate (i. e., a *structural predicate*) defining the relationships between the XML nodes to be matched. In the

literature, algorithms for axes *child* (C in our representation), *descendant* (D), and the *attribute* axis (@) have been proposed, e. g., [Bruno 02]. Due to the internal structure of the twig join algorithm, further axes, like the sibling axes or reverse axes, cannot be evaluated. In XQGM, all XML node fulfilling a certain predicate (e. g., a name test) are delivered by access operators reachable via the *for*-quantified tuple variables. Therefore, the input of a twig node is given by a tuple variable. The link between these two components is represented by a dotted line. From the point of the twig join operator, the node predicate is already checked and only the structural predicate needs to be matched. For example, node 42 is connected to tuple variable F:42 which solely delivers *price* nodes.

In the following, we want to introduce the semantics of the twig nodes and how twig nodes influence twig matching and result generation. To facilitate the representation, we slightly abstract from the twig specification in XQGM and only show the twig itself (with embedded node predicates). For example, in the "Twig" column of Figure 5.12a, a twig with four nodes having node tests a, b, c, and d is presented. We will introduce the twig node semantics with the help of the eight examples in Figures 5.12 and 5.13. The first column in these figures contains the name of the example and a short description. The second column presents the twig to be matched against the document in the third column. Column number four presents the result graphically, whereas the last column shows how the result looks like in tuple representation. This representation is generated by the twig join operator during evaluation. Lets step through the examples:

a) The first example shows a plain twig consisting of four nodes connected via three *child* edges. Each node produces output (indicated by the white color of the node). The twig join algorithm matches this twig against the document in the third column. The two occurrences of this twig are shown in the fourth column. As you can see, element a_1 occurs twice in the result, because it is the root for both matches. This is also true for the tuple result. Further, note that the order among sibling twig nodes is not important for matching: although node c occurs before node d in the twig, both subtrees in the document are matched. However, the order of the nodes in the twig plays a role for the generation of the tuple result. Here, the order of the tuple fields correspond to the twig nodes in pre-order.

b) The second example underlines the output ordering semantics again. Here, the twig consists of three nodes connected by *descendant* edges. The pattern can be matched five times and the tuple result consists of three fields per tuple. The result is sorted by the first field, then by the second field, and then by the last field. For output generation, we could decide to only return nodes for a certain subset of twig nodes. In the following, we call nodes producing a field in the result tuple *output nodes*. Non–output-nodes have a grey color in the our examples. If some nodes do not produce output, the result is projected on their corresponding fields and duplicates are removed. In our example, if a and c were the only output nodes, the result would be: <[a_1, c_1], [a_1, c_2], [a_1, c_3], [a_2, c_1], [a_2, c_2]>

c) The third example introduces two new Boolean node types, namely the *and node* and the *or node* (other nodes will be called *path nodes* in the following). An *and node* can have two or more branches. All these branches have to be present in the document for the *and* node to match. Note, as a convetion we allow a certain shortcut notation for and nodes: if a path node has multiple branches, we assume

Figure 5.12 Twig matching examples

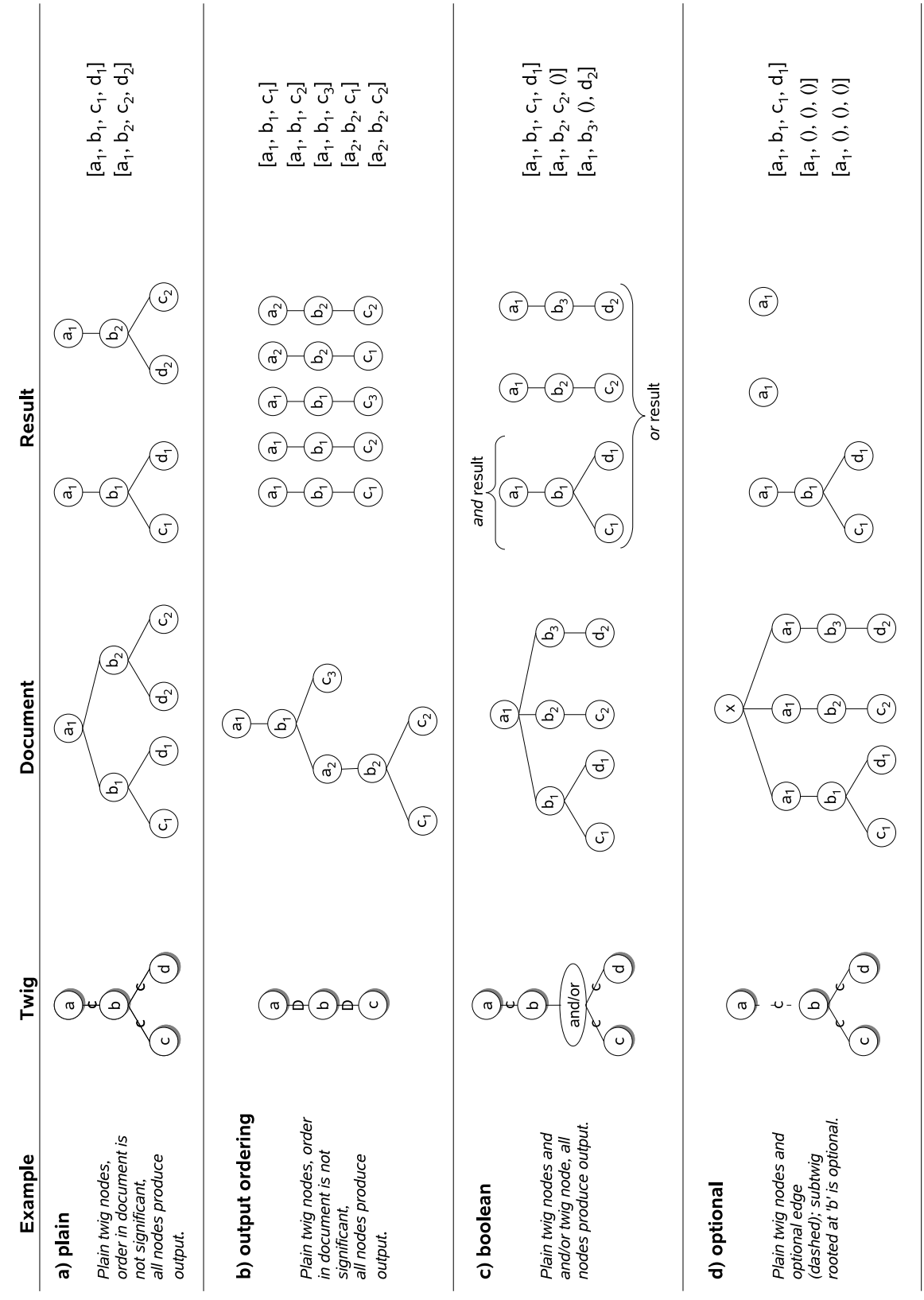


Figure 5.13 Twig matching examples continued

an *implicit and* node. In case of an *or node*, only one of the branches has to exist. Therefore, if we assume an *and node* in our example, only the first subtree is a match. Otherwise, in case of an *or node*, all three subtrees are returned. Because the *or* node also matches, when some information is not available, we have to pad the result tuple with an empty sequence. This ensures a homogeneous result, i. e., each tuple fields corresponds to a particular output node.

d) Our fourth example introduces a new edge type, namely the *optional subtree edge*. It is represented by a dashed line in the twig. A node with an incoming optional subtree edge roots a subtree which may or may not exist in the document. In our example, the subtree exists only in the first branch of the document. In the second and in the third branch, the subtree exists only partially. Therefore, only the non-optional node (matching a) is returned. In the tuple result, all fields that belong to a non-existent optional subtree are filled with empty sequences. Please note the similarity between *or nodes* and *optional subtree edges*. We will exploit this similarity for the implementation of the twig operator.

e) The fifth example introduces yet another type of node, namely the *grouping node*. A grouping node is a special path node that groups the matched subtrees below in the tuple result. In the example, you can see that the complete document is matched. The tuple result reveals that the result delivered by the subtrees below the grouping node is (separately) nested w. r. t. the result delivered for the grouping node. If b would not group, the result would consist of four tuples: <[a_1, b_1, c_1, d_1], [a_1, b_1, c_1, d_2], [a_1, b_1, c_2, d_1], [a_1, b_1, c_2, d_2]>.

f) The next example shows how an *embedded output expression* works. An output expression is a simple XQGM expression with one or more tuple variable references. An output expression is indicated by the prefix "`out`". It can be attached to a grouping node in the twig, where the output expression is applied to the nested subtrees for every match of the grouping node. In our example, the grouping node creates an element x and adds all d elements matched for a b element inside this x. Note that the tuple variable references twig nodes. In the graphical representation of the output expression, the reference to twig node d is presented as `[d]`. Furthermore, note that the subtree below each d has to be materialized in the tuple result, because the ds are embedded in a new element. This fact is indicated by the curly braces.

g) In the last but one example, we show the application of an *output filter* on some intermediate twig result. An output filter is indicated by prefix "`test`" and works similar to an output expression. It can be attached to a grouping node and it can reference one or more tuple variable references (represented by the bracket notation in the graphical representation). If the predicate in the output filter is true, the subtree rooted at the grouping node matches. Otherwise, it does not match. In our example, only the left subtree of the sample document matches. Because the output node is set to d, only element d_1 is in the output.

h) The last example shows how positional predicates can be embedded into the twig specification. To make sure that the necessary context information is generated, a path node can be adorned with two flags. These flags signal the need for context size and context position information. In our representation, these flags are visualized by appending a string ("`cp`" or "`cs`") after the name test in the twig node. A *positional filter* (prefixed "`ppred`") can query the positional information

generated for a path node. In our example, the left subtree in the document fulfills the positional predicates, while the right subtree does not. Because b is the output node, only b_1 is returned.

Obviously, the twig pattern is quite expressive. To summarize, it supports output nodes (i. e., projection), grouping, output ordering, Boolean *and* and *or* predicates, output expressions, output filters, and positional predicates. Currently, there is no algorithm in the literature able to evaluate a twig pattern of this expressiveness. In Chapter 8, we will design such an algorithm. You might ask the question, why it is meaningful to extend the expressiveness of a twig in the way shown at all. The simple answer is: with these extensions, the twig algorithm can be applied in many more situations than without them. Now, lets see how we can map XQGM substructures to twig operators, i. e., how twigs are discovered.

5.8.2 The HTJ Discovery Rule Pattern

In this work, discovering twigs is a best effort approach. This means that the pattern of the twig discovery rule searches for a "twig starting point" and the transformation instruction tries to integrate the semantics of as many (following) operators as possible into the twig. Therefore, the higher the expressiveness of the twig pattern, the more operators can be transformed into it. In the current implementation, the twig discovery rule is applied in its own rewriting stage, i. e., insulated from the other rewriting rules. The necessity for this desicion becomes clear during the discussion of the transformation instruction. Basically, the separation simplifies the rewriting process. Integrating twig discovery with the other rule set would, nevertheless, be possible. However, this task is left open to future work. With respect to this task, the rule proposed in this section can, at least, define the twig discovery semantics.

As stated above, the pattern searches for a starting point. This starting point is given by a structural join followed by 1) another structural join, 2) a split operator (followed by structural joins), or 3) a select operator. Figure 5.14 presents examples for the first two cases. The matches of the pattern are shown in dashed boxes. Note, we have already seen these two queries:

- The XQGM instance in Figure 5.14a is the third example from the unnesting section (Figure 5.10b on Page 117). The query was:

```
doc("auction.xml")//item[count(location) < 2]
```

- The XQGM instance in Figure 5.14b is the unnested version of the example shown for the multiple correlated expression pull-out rule (Figure 5.7b on Page 109). Here, the query was:

```
for $i in doc("auction.xml")//item
return <qloc>{$i/quantity}{$i/location}</qloc>
```

Besides the structural conditions, the operators participating in a pattern match have to fulfill certain requirements: A structural join operator in the match has to have a *twig axis*, i. e., one of the axes supported by the twig pattern as introduced above (*child*, *descendant*, and *attribute*). Furthermore, the first access operator has to have two access operators as input. A select operator in the match has to have exactly one tuple variable T without a correlated edge and no sorting specification. Furthermore, all expressions in the select operator have to reference T. In case of a

Figure 5.14 Twig discovery

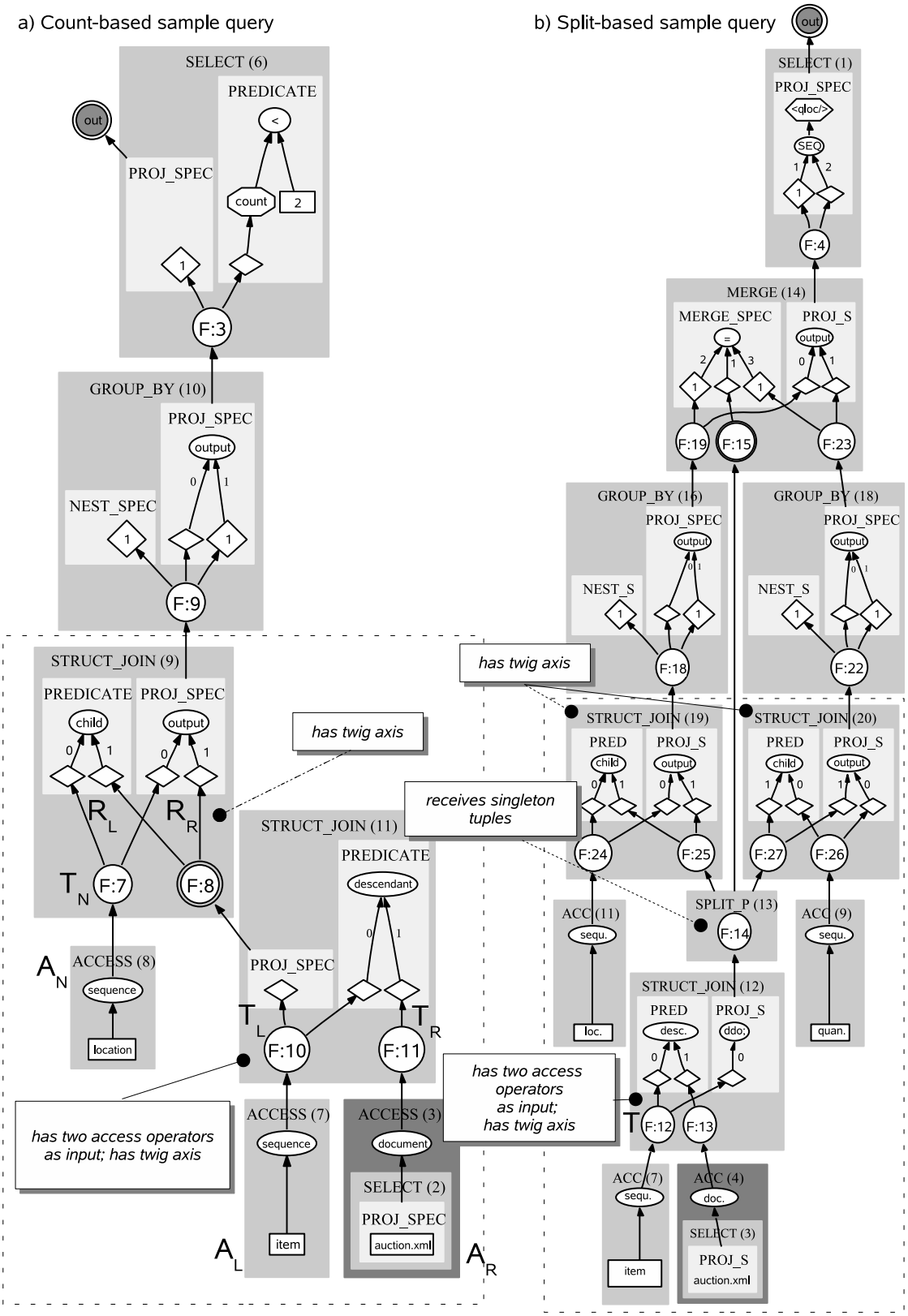
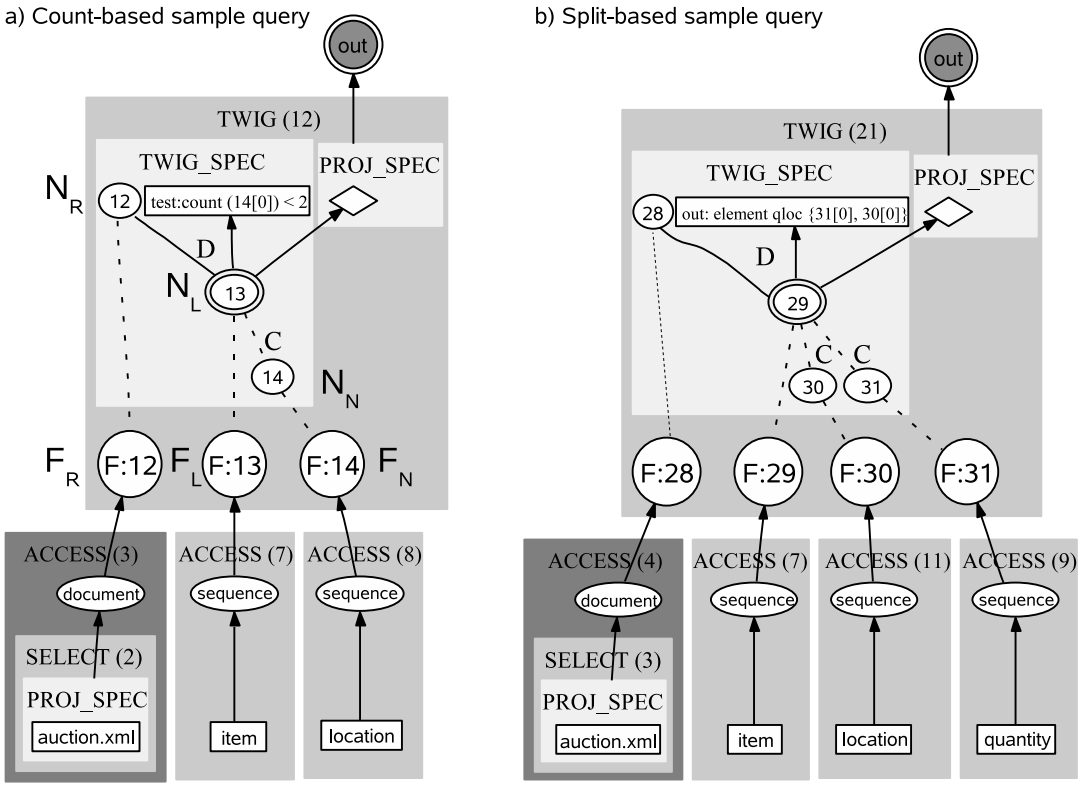


Figure 5.15 Discovered twigs of two sample queries



split operator, these restrictions also apply to the structural join and select operators inside the split branches. The branches have to be completely unnested, i. e., they do not contain any operator with a correlated edge. Finally, the split operator has to receive a sequence of singleton tuples (recognizable by the projection specification of their input operator).

5.8.3 The HTJ Discovery Transformation Instruction

Starting from the match, the transformation instruction first creates a twig pattern and then tries to embed as many operators as possible into this twig. In our sample queries, all operators will be completely integrated. The result is shown in Figure 5.15. The twig representation in this figure slightly differs from the representation in the overview of Figures 5.12 and 5.13 on pages 131 and 132. Here, not the node tests themselves are printed inside the twig nodes. Rather, tuple variables are referenced (by their ID). Furthermore, output nodes are defined by an output edge into a tuple variable reference of the projection specification (all twig nodes not referenced do not produce a direct output). Finally, the tuple variable (and not the node test) is referenced in the output expression and in the filter expression . The number in brackets denotes the access position in the input tuple delivered by the tuple variable (which is "0" in most cases). Let us now take a look at the actions that transformed the XQGM instances in Figure 5.14 to twig operators.

First, a holistic twig join operator H is created. At the beginning, this operator is

empty. Then, the transformation instruction iterates over the XQGM operators as follows: Starting from the matched structural join operator, every operator O (up to the XQGM root operator) is probed. When the operator can be integrated into a twig, the necessary actions take place and the next operator is examined. Otherwise, the transformation instruction stops at O, finalizes the twig construction, and replaces the input of O with the twig. In case of a split operator, all split branches are checked first. Only when every operator in every branch can possibly be integrated into the twig, the transformation instruction executes the integration. Otherwise, the twig discovery stops at the split operator. To pass information from one iteration step to another, a *discovery context* C is required.

The first operator to be integrated into a twig operator is a structural join (as defined by the pattern). Therefore, we start our discussion with this operator.

Structural Join Integration

As we have seen in the pattern, a structural join operator can be integrated into the twig, when its predicate defines a twig axis and when the first structural join operator to be integrated has two access operators as input. If a structural join operator does not fulfill these requirements, twig discovery stops (as explained above). Otherwise, the operator has to be integrated. In case of an initial integration (first structural join), the following actions are executed (as an example, consider `structuraljoin(11)` in Figure 5.14a):

- The axis is extracted from the structural join.
- Access operators A_L and A_R delivering the input to the two tuple variables T_L (left) and T_R (right) are detached.
- Two *for*-quantified F_L and F_R tuple variables receiving A_L and A_R as input are instantiated inside twig operator H (see Figure 5.15a).
- Two twig nodes N_L and N_R are created. Technically, twig nodes are modeled as tuple variables. Therefore, they can be referenced by tuple variable references. However, twig nodes, in turn, also reference tuple variables. The ID of a twig node (N_L or N_R) is the ID of a tuple variable (F_L or F_R) in H. For later use, the correspondance between the generated twig node and the tuple variable (T_L or T_R) of the original structural join operator is written into context C, i. e., $C(T_L) \leftarrow N_L$ and $C(T_R) \leftarrow N_R$.
- An edge between the two twig nodes is created. The edge carries the axis inferred above. The parent twig node references the tuple variable with A_R as input (i. e., N_R); the child node references tuple variable A_L as input (i. e., N_L).

The result of these operations is a twig operator H with two twig nodes. Essentially, this twig is a structural join. At this point, the projection specification of the twig is still missing. It will be instantiated at the very last.

In case, the structural join is integrated into an existing twig, the following actions are necessary (as an example, consider `structuraljoin(9)` in Figure 5.14a):

- The axis is again extracted.
- The input A_N of the tuple variable T_N whose input is not yet integrated is detached.
- A *for*-quantified tuple variable F_N is inserted into H receiving A_N as input.

- A twig node N_N is created which references the new tuple variable. As before, the context is populated with this information: $C(T_N) \leftarrow N_N$.

- To insert an edge, the twig parent of the newly instantiated twig node has to be found and the edge type (optional or not) has to be inferred. The first task is achieved by analyzing the original XQGM instance: The structural predicate contains two tuple variable references R_L and R_R. R_L references T_N. R_R references the subtree already integrated into the twig. Following the second reference down the tree (i. e., over a chain of components consisting of tuple variables, projection specifications, and tuple variable references), a tuple variable inside a structural join operator can be found, which receives the output of an access operator. In the following, we will call this tuple variable *source tuple variable* for reference R_R. In our example (Figure 5.14a), the source tuple variable is T_L. The access operator provided one join partner for the currently processed structural join. The tuple variable generated a twig node during a previous operator integration. This twig node is the searched twig parent (N_L in our example). It can be found in the context (i. e., $C(T_L)$).

  The second task (inferring the edge type) is also achieved by analyzing the original XQGM instance. If the right tuple variable (i. e., not T_N) is *outer*, the edge is optional. As we will see below, a previously integrated split operator might also require an edge to be set as outer. This requirement is stated by a Boolean flag ("*nextTwigNodeOptional*") in context C and is obeyed here. When all structural join operators following the split operator are integrated, the Boolean flag is set to *false*.

In our example of Figure 5.14a, after the integration of the first two structural join operators, the twig specification in H consists of the three twig nodes (see Figure 5.14a). Because in the original XQGM instance, F:8 was defined as outer, the edge between node 13 and node 14 is an optional subtree edge. In this state, the projection specification, the grouping semantics, and the content test are still missing. These components will be produced during the integration of the following operators.

Split Integration

The integration of a split operator is quite straightforward. If the split operator does not receive a sequence of singleton tuples, twig discovery stops here. The reason for this circumstance is given below. As another prerequisite, the operators in all branches of the split operator are checked to ensure that they can be integrated into the twig (note, the conditions for each operator can be found at the beginning of the integration discussion). If at least one operator cannot be integrated, the twig discovery stops here. The reason is that, essentially, a branch stands for a part of a relative path expression. If one of these parts cannot be integrated into the twig, the discovery pattern is not able to integrate the other ones, because following operators might depend on the integrated parts. In some cases, the integration of only a subset of branches might be possible. However, we leave this case open for future work.

If all branches can be integrated, the rewriting process begins. In case of a projection split operator (i. e., when the multiple correlated expression pull-out rule matched), the edges leading to the next twig nodes generated have to be optional subtree edges. Therefore, the *nextTwigNodeOptional* flag in context C is set to *true*. The reason is that in the merge operator receiving the result from the split branches has an outer tuple variable with the projection split as input. In case of an *and* split

operator or an *or* split operator (i. e., when the Boolean rule matched), an *and/or* twig node N is created. Because the split operator receives a sequence of singleton tuples (as required above), we can retrieve source tuple variable T in the original XQGM instance (that generated this input) by following the component chain of the input down to the access operator. Let us pretend that in our example in Figure 5.14b, the split operator is an *and split*. Then, the source tuple variable would be F:12. The twig node generated for this tuple variable (currently stored in $C(T)$) becomes the parent of the new *and/or* twig node. All following integration actions have to respect the new twig node. Therefore, we have to update the context by setting $C(T) \leftarrow N$.

With the discussion so far, we can understand how the integration of the operators in the dashed boxes of Figure 5.14 works. The result in this state for the second query in the figure is shown in Figure 5.15b. The twig discovery rule created a twig operator with four tuple variables and four twig nodes, of which 30 and 31 have an optional incoming edge. Still missing are the grouping information, the output expression, and the projection specification. The integration of the *group by* operator will be discussed next.

Grouping Integration and Unnesting Integration

Before a *group by* operator can be integrated, its structure has to be checked. An integration is not possible, when the grouping operator contains a complex projection specification not solely consisting of tuple variable references and the special positional functions (*xtc-fn:position* and *xtc-fn:last*) introduced in Section 5.7. Note, the rule set introduced in this work will not generate such a case. However, for completeness, we nevertheless pose this requirement here. If the projection specification contains a positional function, the tranformation instruction searches for the consuming select operator among the operator's ancestors. The consuming operator and all operators "in between" have to be integrable. Otherwise, twig discovery stops at the grouping operator. The rationale behind this condition is to ensure that also the consumer can be integrated into the twig.

If all preconditions are met, the twig is integrated. First, all tuple variable references in the nesting specification are examined. For each of these references, the *source tuple variable* is retrieved. From the context, we get the twig nodes generated for these tuple variables. These twig nodes from a path, of which the leaf node is converted into a *grouping twig node*. In our first example (Figure 5.14a), tuple variable F:10 is the source tuple variable for the reference in the nesting specification of `group by (10)`. The twig node generated for this tuple variable was number 13 (in Figure 5.15a).

In case, a positional function exists in the projection specification, we have to make sure that the "cp" and "cs" flags are set in some twig node accordingly (as shown in Figure 5.13h). The twig node in question can be inferred by 1) calculating the *source tuple variable* of the tuple variable reference, that is, the argument of the position function, and by 2) looking up the generated twig node for the source tuple variable in context C. In case of an *xtc-fn:position* function, the "cp" flag is set; in case of an *xtc-fn:last* function, the "cs" flag is set. Note, the positional predicate (`ppred`) is created, when the consuming select operator (matched by the transformation instruction above) is integrated.

Unnesting is the reverse operator of grouping. The effect of integrating an unnest operator into a twig operator is the conversion from a grouping path node to a

normal path node. This is done by analyzing the projection specification and the nesting specification of the unnest operator. We skip the description of the necessary actions here, because they are straightforward.

Merge Integration and Set Integration

In an XQGM instance, a merge operator only exists, when the multiple correlated expression pull-out pattern matched. In a way, the merge operator *closes* the split by uniting the output of the split branches. Therefore, we call the merge operator a *closing operator*. Set operators can also be closing operators, when they unite the results generated by *and/or* split branches. In both cases, all necessary integration activities have already been executed by the integration of the split operator. Therefore, nothing has to be done.

Select Integration

We have already discussed the conditions for the integration of a select operator in the pattern above: A select operator has to have exactly one tuple variable T without a correlated edge and no sorting specification. Furthermore, all expressions in the select operator have to reference T. For integration, therefore, only the projection specification and an optional predicate are relevant. We start with the second component.

The integration of a select predicate leads to an output filter, as shown in Figure 5.13g. Embedding a predicate like this is quite straightforward. For every tuple variable reference, the source tuple variable T_S is inferred. Then the reference is rewritten such that it points to the twig node N, which was generated for that source variable (i. e., by a lookup in the context: $C(T_S)$). After this rewriting, the predicate is removed from the select operator and attached to a twig node N_P. This twig node is inferred as follows: If the predicate contains exactly one tuple variable reference, we navigate from N up in the twig until a grouping twig node is found. This twig node is then the host for the predicate. If the original predicate has multiple tuple variable references, multiple twig nodes are referenced in the rewritten predicate. Then the least common ancestor of these twig nodes is then the host.

As an example for an output filter, consider the predicate in `select(6)` (Figure 5.14a). The source tuple variable for the reference in this predicate is F:7, for which twig node 14 was generated. You can see that this twig node is referenced in the rewritten predicate shown in Figure 5.15a. The host twig node for the predicate is twig node 13, because the expression contained only one reference and twig node 13 is the first grouping ancestor.

Positional predicates and existential predicates receive a special treatment. Positional predicates are not transformed to output filters but to positional filters (see Figure 5.13h). Because an unnest operator removes the nesting node, the positional filters are directly attached to the twig node generating the positional information (i. e., the node with an "cp" flag or "cs" flag set to *true*). An existential predicate (i. e., an XQGM predicate containing a simple tuple variable reference) does not result in an output filter. Rather, the corresponding incoming edge of the referenced twig node is set from optional to mandatory (because the nodes produced by this twig node have to exist).

The integration of the projection specification is executed quite similar to predicate

integration. The differences are: 1) a projection specification can contain multiple output expressions, and 2) output expressions are generated instead of filter expressions. As a result, a twig node can host multiple output expressions. Furthermore, if an output expression consists of a simple tuple variable reference (i. e., the nodes delivered by the twig have to be returned), nothing needs to be done. This case is handled when the twig is finalized (see below).

As an example for the integration of an output expression, consider `select(1)` in Figure 5.14b. The source operator of the left reference in the output expression is F:26. Therefore, this reference is rewritten to twig node 31. The source operator for the right reference is F:24; the reference is rewritten to twig node 30. The least common ancestor of twig nodes 30 and 31 is 29, which is, therefore, the host for the output expression, as you can observe in Figure 5.15b.

Twig Finalization

When all operators are integrated, the instantiated twig operator has to be finalized. Especially, the projection specification of the twig operator is still missing and has to be created. For this task, the projection specification of the most recently integrated operator is examined. We have to distinguish three cases:

1) The projection specification contains a single tuple variable reference. In this case, the source tuple variable of the reference is retrieved, the reference is rewritten to the twig node created for the source variable, and, finally, the reference is integrated into the projection specification of the twig operator.

2) The projection specification contains a single non–tuple-variable reference (i. e., a complex expression). Then this complex expression has been integrated as an output expression into the twig as described above. As we have seen, the output expression is "attached" to a twig node N. To deliver the output generated by this expression, a tuple variable reference to N is added to the projection specification.

3) The projection specification contains multiple expressions. In this case, each of the expressions is rewritten and integrated as described in the previous two points.

An example for the first alternative is shown in Figure 5.14a: the tuple variable referenece of the projection specification in `select(6)` points as tuple variable F:10 as source. For this tuple variable, twig node 13 was generated. This twig node is simply referenced in Figure 5.15a. An example for the second alternative can be found in Figure 5.14b: The complex expression in the projection specification of `select(1)` has been integrated as an output expression into the twig shown in Figure 5.15b. It simply has to be referenced.

As a final action, the newly created twig operator has to be created with the subtree it "consumed". The holistic twig join operator is a bulk processing operator. Therefore, the twig discovery rule belongs to the fourth category on Page 93.

5.8.4 Summary

At this point, we finish the discussion about the query processing at the logical level. We have seen how queries are parsed, normalized, typed, simplified, transformed into XQGM, and how they are rewritten. During rewriting, especially query unnesting and the discovery of opportunities for the twig join algorithm were of ma-

jor concern. We will now consider related work on XML query algebras and XML query rewriting, before we proceed with the physical aspects of query processing.

5.9 Related Work

In the following, we will consider the internal query representation and algebraic rewriting phase of the five systems introduced in Section 2.3. For each system, we will provide an example. Furthermore, we outline what kind of rewritings have been published so far.

5.9.1 Galax

The algebraic teatment of queries inside Galax mainly relies on [Ré 06] and [Michiels 07]. The first reference presents how queries can be transformed into a tuple algebra (called RSF in the following after the names of the authors). The second reference then shows how opportunities can be discovered to employ a holistic twig join operator. The first reference is related to Chapter 4 (*The XML Query Graph Model*), while the second paper discusses strategies similar to Sections 5.7 (*Query Unnesting*) and 5.8 (*Twig Query Detection*). Additionally, the authors of Galax have published how unnecessary calls to the *ddo* function can be eliminated [Fernández 05] and how XML data can be projected before a query is processed [Marian 03] to reduce the size of the DOM tree Galax operates on. Let us take a closer look at the algebra and the proposed rewriting techniques.

The Algebra

In RSF, a query is translated into an algebraic expression. The algebra contains operators for tuple manipulation, for XML manipulation, and for tuple-to-XML (XML-to-tuple) conversion. XML operators are constructors, XML navigation and projection (i. e., subtree selection by path expression), type operators, functional operators (e. g., a call to an XQuery function), and I/O operators (to retrieve or serialize XML items). Operators for tuple manipulation are borrowed from the relational algebra, i. e., tuple construction, select, project, join, map (i. e., the application of a function on a tuple), grouping, and sorting. Conversion operators are map-from-item and map-to-item. The first one converts a sequence of items into a sequence of tuples. The second one operates vice versa.

In general, an operator has the form: $Op[p_1, \ldots, p_i]\{DOp_1, \ldots, DOp_h\}(Op_1, \ldots, Op_k)$, where the p_i's are static parameters, DOp_i are dependent parameters, and Op_i are independent parameters. A static parameter "configures" the operator, for example, an element constructor operator receives the name of the element to be constructed as a static parameter. The meaning of independent and dependent operators is similar to our notion of dependent and independent tuple variables. A dependent operator is a subexpression that can receive some value from outside. This value is generated by an indepdent operator and can be referenced using the IN keyword. Let us consider one of the examples from [Ré 06] to get an impression of the algebra:

```
for $p in $auction//person
let $a as element(*,Auction) :=
  for $t in $auction//closed_auction
```

```
  where $t/buyer/@person = $p/@id
  return validate {$t}
return <item person="{ $p/name }">
         { count($a/element(*,USSeller)) }
       </item>
```

The query searches for all *person* elements that appear as *buyer* in some *closed_auction*. For each of these *person* elements, the *name* and the number of *closed_auctions* with children of type *USSeller* are returned. Additionally, the type of the *closed_auction*s is checked and validated. The RSF expression of this query has the following form:

```
1.MapToItem
2.  {Element[item]
3.    (Sequence
4.      (Attribute[person]((IN#p)/name/text()),
5.      count(IN#a/element(*, USSeller)))))}
6.  (MapConcat
7.    {[a : TypeAssert[element(*, Auction)*]
8.      (MapToItem{Validate(IN#t)}
9.        (Select
10.           {IN#t/buyer/@person = IN#p/@id}
11.           (MapConcat{MapFromItem{[t : IN]}
12.             ($auction//closed_auction)}(IN))))]}
13.   (MapConcat{MapFromItem{[p : IN]}($auction//person)}([])))
```

The algebra expression is an operator tree and is read from bottom to top, starting with line 13. Note, normally, the path expressions would not literally appear in the query. They would also be normalized. However, for brevity, they are kept as paths. In line 13 of the algebra expression, all *person* elements are retrieved from the document, mapped to tuples (MapFromItem) and concatenated to a sequence (MapConcat). The dependent/independent expression evaluation can be explained by the MapFromItem expression in line 13: independent expression *$auction//person* is evaluated, and for every *person* element, dependent expression $[p : \text{IN}]$ is evaluated. This dependent expression is a tuple constructor, referencing the person element by IN and constructing a tuple with a field named p. With the generated tuple stream, the dependent expression of MapConcat (line 6) can be evaluated. This subexpression contains the join on the *buyer* of some *closed_auction* element (and a validation followed by a type check). The final result is assembled in the dependent subexpression of line 2 to 5.

Both being tuple algebras, LAL (the logical algebra introduced in this work) is quite similar to RSF. In fact, instead of LAL, we could have mapped XQGM to RSF. RSF would then have been applied as background for the definition of the XQGM semantics: Essentially, the IN technique is similar to our evaluation context C, which keeps track of variable bindings. MapConcat is similar to the tuple generator. However, the latter one is more expressive, because MapConcat can only evaluate one correlated subexpression, while TUPGEN can handle more than one. Therefore, the AST-to-XQGM translation process would have to be modified to generate select operators with at most one dependent tuple variable. This would be possible. However, the resulting structure would contain more operators, more complex projection specifications, and would be harder to read. In the end, we decided against

the application of RSF to define the XQGM semantics, because the operators in the literature [Ré 06] are not thoroughly defined (only their signature is given, but not their implementation). In this work, however, we wanted to introduce the XQGM semantics completely.

Algebraic Optimization

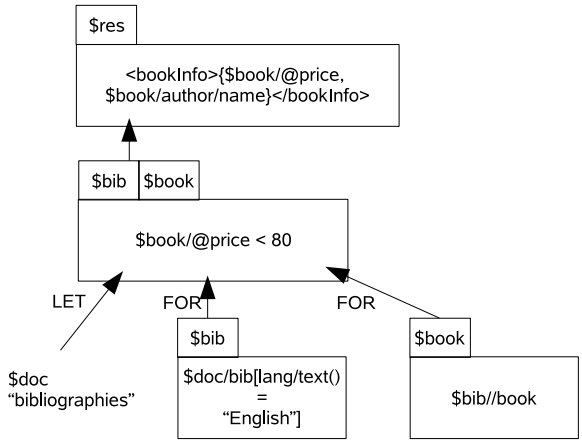
Let us take a look at the query rewriting strategies presented in the Galax context [Michiels 07]: The first bulk of rewriting rules aims at query simplification and is comparable to the actions presented in Section 4.3.2. The second set of rules aims at twig discovery. Therefore, a twig operator is introduced into the RSF algebra. This operator has the form TupleTreePattern[*TreePattern*$\{q_1, \ldots, q_n\}$]($S(t)$) and takes a tree pattern *TreePattern*$\{q_1, \ldots, q_n\}$ as static parameter and an operator generating a sequence of tuples $S(t)$ as independent parameter. The tree pattern consists of tree nodes q_i and allows the *child*, *descendant*, or *descendant-or-self* axis. Output generation is restricted to exactly one tree node. The twig is matched against the items provided by the independent operator. Compared to the definition of our twig operator in Section 5.8.1, the TupleTreePattern has several drawbacks. It cannot capture: 1) multiple output nodes, 2) boolean *or* predicates, 3) optional subtrees, 4) grouping, 5) output expressions, 6) output filters, and 7) positional predicates. Therefore, the TupleTreePattern operator can only be applied in very simple queries not requiring one of the previously enlisted functionalities. Reference [Hidders 07] also presents how these kinds of tree patterns can be discovered in an XQuery expression. However, the expressiveness of the assumed twig join operator is likewise restricted as in [Michiels 07].

Obviously, Galax and XTC have several techniques in common. Both rely on the Formal Semantics for normalization and static typing. Furthermore, their logical algebra is quite similar and they both try to discover opportunities for the application of twig operators. However, the XTC approach reaches further: With XQGM, we chose a non-algebraic internal representation, which provides a more intuitive basis for implementation and explanation (than algebraic expressions). In situations, when the twig join operator cannot be applied, we can, nevertheless, generate plans with structural joins, because we *unnest* all queries. Galax has no notion of a structural join (only of a twig join) and, therefore, relies on a node-at-a-time evaluation, when the twig operator cannot be applied. Note, the query unnesting capabilities of XTC are a prerequisite for twig discovery. We have developed this technique indepenently from Galax in [Mathis 07a, Mathis 07b]. Finally, our notion of a twig and our twig discovery rules are much more expressive than the ones presented in [Michiels 07, Hidders 07].

5.9.2 IBM DB2 Pure XML

As stated in Section 2.3.2, the internal query representation of DB2 is the Query Graph Model (QGM), originally developed for the Starburst System [Haas 89, Pirahesh 92]. This internal representation is also used to process XML queries. The following description on how the QGM is drawn from [Beyer 05], [Balmin 06], and [Beyer 06]. Unfortunately, these papers are fairly sketchy only giving some hints as a short example on how the QGM solution might look like. Rewriting strategies were only recently published in [Özcan 08]—developed in parallel to the work presented in this dissertation.

Figure 5.16 A DB2 QGM example



QGM

Starburst's QGM was designed for extensibility. Prior to the DB2 pureXML extension, QGM was employed to represent relational queries internally. XQuery integration required two types of extensions: the underlying data model had to be enriched with sequences, and special operators for sequence handling and path evaluation had to be inserted into the operator set. Figure 5.16 shows an example QGM instance taken from [Beyer 05]. You can observe that path expressions are not normalized but kept as they are. For their evaluation, DB2 uses a bulk operator (as XTC does), which is embedded into QGM by a table function. Furthermore, we can observe that QGM is also extended with the *let* and *for* semantics of *table* variables (which is the corresponding concept of a tuple variable). In case of a *for*, an iteration over the input is executed, in case of a *let*, the inputs are aggregated into a single sequence. How XML queries are exactly represented in the QGM is not publicly available. [Özcan 08] only reveal that FLWOR expressions are translated to two select operators. The first select operator generates tuple sequences for the *for*/*let* bindings, while the second select operator calculates the predicate of the *where* expression and the *order by*/*return* expression. The reference further states that, in some cases, these two select operators are merged into one.

Query Rewriting

In DB2, the goal of the algebraic query rewriting is to pre-optimize the query and to bring it into a normal form. [Balmin 06, Beyer 06] state that they eliminate redundant operators, merge nested query blocks, remove unused variables, and push down navigation steps and content predicate. We can infer that the tree patterns supported can have at most one output generating node and that they support optional edges. In [Özcan 08], this functionality is extended by grouping. Furthermore, the latter reference adds rewriting strategies to transform *let* and *for* quantifications to simpler expressions. Certain types of XPath expressions can be merged into one operator, if one expression is used as input for the other. Finally, they claim to have implemented various join/query decorrelation methods. All these rewritings, however, are only sketched and not fully traceable.

5.9.3 Timber

The algebra developed for the native Timber XML DBMS is called TAX [Jagadish 02b], which stands for "tree algebra for XML". For the design of TAX, the authors drew an analogy from relational query processing and relational algebra to XML query processing to an XML algebra. In relational algebra, the operators consume one or more relations and produce an output relation. Consequently, in TAX, an operator consumes one or more collections of labeled ordered trees (documents or fractions) and produces a collection of labeled ordered trees. The structure of XML data is heterogeneous (in contrast to relational data). To operate on this heterogeneous data, XQuery binds XML nodes to variables. The set of bound variables is then "homogeneous" and further processing operations can be applied. The same strategy is followed in TAX. Here, every operator is parameterized by a pattern tree (i. e., a twig; similar to XTC). TAX pattern trees support the *child* and *descendant* relationship as well as content predicates on twig nodes. Pattern trees can be matched against the document resulting in an ordered forest of so-called *witness* trees. A selection operator, for example, can match a pattern tree against the document and returns a set of witness trees. From the data trees in the original document, only the matched XML nodes are returned. However, if a selection result has to be processed by further operators, the subtrees of the matched nodes might also be of interest. Therefore, an operator can additionally be parameterized by an *adornment list*. This list contains a set of twig nodes, for which not only the node, but also its subtree appears in the output.

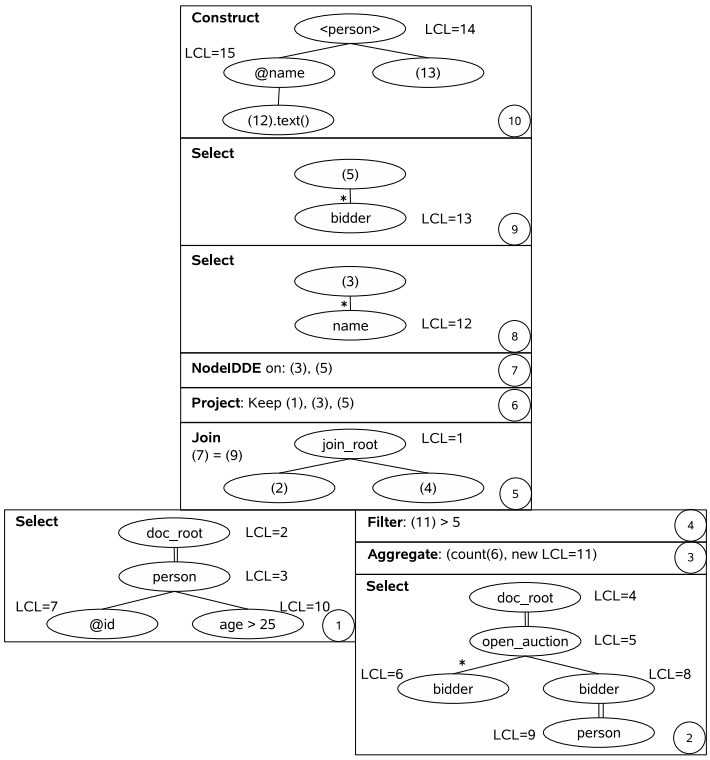
All in all, the TAX algebra defines the following operators: selection, projection, product (and, thus, joins), grouping, aggregation, order by, renaming, reordering, copy-and-paste, value updates, node deletion, and node insertion. Note that TAX is one of the few algebras containing built-in operators for document modification. The authors of TAX recognized that their simple notion of a pattern tree leads to a lot of overhead, because in queries with nested FLWOR blocks, a twig could only range over a single block. Therefore, they generalized their notion of a pattern tree in [Chen 03c] and [Paparizos 04]. In [Chen 03c], they added support for optional edges (as in our understanding of a twig). In [Paparizos 04], they added grouping. Because these generalizations also influenced the algebra operators, the authors chose another name for TAX, namely tree logical classes (TLC).

Figure 5.17 contains a graphical representation of a TLC expression for the following query:

```
for $p in doc("auction.xml")//person
for $o in doc("auction.xml")//open_auction
where count($o/bidder) > 5 and $p/age > 25
          and $p/@id = $o/bidder/@person
return <person name={$p/name}>{$o/bidder}</person>
```

The data flows from bottom to top. The first selection operator (1) matches a pattern against the document returning a *person* subtree with *@id* and *age* children (where *age* is larger than 25). Each of these subtrees is rooted by an artificial *doc_root* node. In the pattern tree specification, you can see the definition of so-called logical classes (LCLs). These logical classes can be referenced by subsequent operators. In operator 2, another the *open_auction* subtree is matched. The "*" defines that all *bidder* nodes shall be grouped below the *open_auction* node. In operator 3, the size of this group is counted, resulting in a new class (LCL = 11), which is filtered in operator 4. Then, the two matched trees are joined, where the join condition is that logical class 7 has

Figure 5.17 A TAX example from [Paparizos 04]



to have the same value as logical class 9. The result is hinged below an artificial *join_root* element with LCL 1. Operators 6 and 7 project the intermediate result and remove duplicates, whereas operators 8 and 9 evaluate a navigation to calculate *bidder* and *name* nodes (grouped below LCL 3 and 5). Finally, the *construct* operator assembles the result. You can see that this query specifies four pattern trees for node selection and two trees for node construction. Due to the expressive power of the holistic twig join in XTC, we only require two pattern trees to answer this query.

The main problem with the TAX approach is that the authors sweep queries with non-child/descendant axes under the rug. The XQuery-to-TAX/TLC translation algorithms directly "parses" XQuery strings into TAX (without normalization, typing, etc.). Because the TAX algebra operates on pattern trees and cannot express non-tree relationships, the approach is quite restricted. Only a small deviation from the supported XQuery templates renders the query not expressible in TAX/TLC. Reference [Paparizos 04] presents some optimizations which require a schema and aim at the reduction of pattern tree nodes. We omit their discussion here. TAX/TLC is a logical algebra. Therefore, for evaluation, a physical algebra is employed. We do, however, not know whether this algebra contains a twig operator (as the XTC algebra does; see Chapter 8). [Paparizos 04] decompose pattern trees into structural joins. In Timber, the actual optimization takes place during this mapping [Wu 03]. The optimization is mainly concerned with finding an optimal structural join order.

5.9.4 Natix

NAL (for Natix algebra) [May 04] is a tuple-based algebra and is an extension of the order-preserving SAL algebra [Beeri 99]. NAL also influenced both, the development of XTC's logical algebra and of the RSF algebra introduced in Section 5.9.1 for the Galax system. NAL operates on sequences of (homogeneous) tuples, each tuple consisting of a set of attribute-to-value mappings. As in XTC, these tuples can be arbitrarily nested. Similar to the notion of the *evaluation context* defined in the Formal Semantics, these mappings keep track of the dynamic variable bindings during query processing.

The Algebra

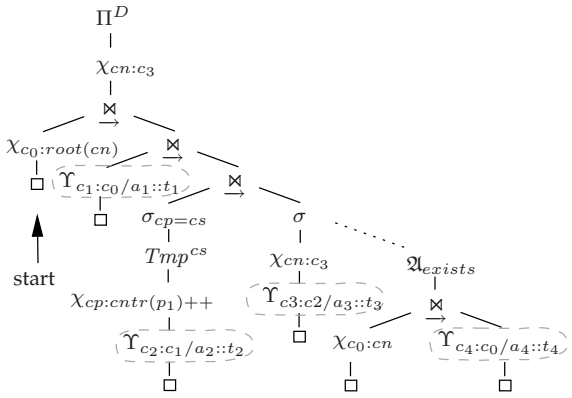
To illustrate how the algebra works, we give a brief example. Let us consider the expression `/a`$_1$`::t`$_1$`/a`$_2$`::t`$_2$`[position()=last()]/a`$_3$`::t`$_3$`[a`$_4$`::t`$_4$`]` depicted in Figure 5.18. The evaluation starts with the singleton scan operator ($\Box$) which creates a singleton sequence containing an empty tuple. It triggers the map operator (χ) to bind the root node of the queried tree to the c_0 attribute of a new tuple. This tuple, in turn, is consumed by the first D-join operator. The D-Join ($\overset{\rightarrow}{\bowtie}$—or $\langle\rangle$ in the textual representation) is similar to XQuery's *for* construct: for each tuple t in the left input sequence, the dependent right expression is evaluated, binding t's attributes to free variables in the right expression (here c_0). Then, the intermediate result calculated for the dependent subexpression is joined with t. In our example, the dependent expression is again a D-Join operator whose left subexpression is an unnest map operator (Υ). This operator is a shortcut for a map operator (χ) followed by an unnest operator (μ). In NAL, Υ is mainly used for the calculation of path axes. Starting from c_0 the path expression $a_1 :: t_1$ is evaluated to a single sequence (using χ) which is immediately unnested (by μ). Together with the D-Join, this results "flat" tuples (although NAL supports complex tuple values).

A predicate is translated into a selection operator (σ), where the predicate's subexpression is compiled into σ's subscript. NAL operators may be arbitrarily nested in this fashion. For each input tuple, the subscript is evaluated. For almost all predicates, certain measures have to be taken to ensure the evaluability of σ's subscript: In case of a relative path expression, the current context variable cn has to be provided explicitly. This is accomplished by the two map operators $\chi_{cn:c_3}$ and $\chi_{c_0:cn}$, the first one binding c_3 to the context variable and the second one "transferring" cn into the variable c_0 of the local context. For positional predicates, the current context position and the context size have to be calculated. This is the task of the special operators $\chi_{cp:cntr(p_1)++}$ and Tmp^{cs}. The first operator simply counts the tuples in its input and attaches a new attribute cp, containing the current position, to them. Tmp^{cs} buffers its input to calculate the total number of tuples in the context, before it attaches attribute cs, containing this number, to each tuple. The aggregation operator $\mathfrak{A}$ evaluates aggregate functions, e. g., $min()$, $max()$, etc. More sophisticated predicates, for example existential comparisons, are possible, too. Finally, the resulting context node is produced by a map operator and duplicate elimination (Π^D) is applied to comply with the XPath semantics.

Algebraic Rewriting

NAL provides an algebraic basement for XPath 1.0 [Clark 99] evaluation. The XPath-to-NAL translation process is described in [Brantner 05]. Furthermore, NAL

Figure 5.18 NAL example



supports FLWOR expressions and quantified XQuery expressions. The translation process for these query types is shown in [May 04]. Regarding rewriting, [Brantner 05] provided some optimization techniques like stacked translation for outer paths, duplicate-elimination push down, and memoization[6]. In [Brantner 06b], certain algebraic equivalences were shown, which enable unnesting of queries with semi-correlated XPath predicates. Queries with semi-correlated predicates have the form $p = e_1[e_2\theta e_3]$, where either e_2 or e_3 is a path expression depending on p's outer—or global—context. Finally, some unnesting strategies for nested queries with aggregation, grouping, and quantification have been proposed in [May 06b].

The major problem with NAL is that it does not encorporate bulk operators (as XTC and the so-far introduced other algebras do). This means that NAL has no notion of a structural join or a twig join. The algebraic rewritings presented in [May 04, Brantner 05, May 06b] are based on value joins (and not on structural joins). To remedy this situation, [Mathis 07b] introduced the structural join operator, resulting in the so-called NAL^{STJ} algebra. The structural–join-based rewriting strategies, for which the correctness proofs are given in [Mathis 07a], build the foundation for query unnesting (i. e., for the *unnesting rule*) introduced in Section 5.7.3.

5.9.5 MonetDB/XQuery

MonetDB/XQuery is an XQuery engine implemented on top of a relational DBMS. The Pathfinder frontend of the MonetDB/XQuery system is a compiler, which is responsible for the translation of XQuery expressions into an extended version of the relational algebra [Grust 04]. The relational algebra contains the following operators: projection/renaming (π), row selection (σ), disjoint union ($\cup$), disjoint intersection ($\backslash$), duplicate elimination (δ), equi-join ($\bowtie$), Cartesian product ($\times$), row numbering (ρ), element construction (ε), text node construction (τ), and arithmetic and comparison operators. Additionally, the algebra contains the so-called staircase join operator [Grust 03a]. The staircase join injects tree awareness into the relational algebra by providing algorithms for the evaluation of XPath axis steps. A key concept for translating XQuery expressions into relational algebra is the way how iter-

[6] These optimizations have not been executed on our example, which is presented in the canonical translation.

ations (i. e., *for* clauses) are translated into joins. The observation is expressed by the following semantic equality [Grust 04]:

$$\text{for } \$v \text{ in } (x_1, \ldots, x_n) \text{ return } e \equiv (e[x_1/\$v], \ldots, e[x_n/\$v])$$

where $e[x/\$v]$ denotes the replacement of the free occurrences of $\$v$ in e by x. This essentially means that expression e can be evaluated "in parallel" for every value in $(x_1, \ldots, x_n)$. Because the underlying data model is relational, all sequences are represented as tables (requiring an additional *pos* column to keep the order among the items in the sequence). In the above expression, if we assume that $(x_1, \ldots, x_n)$ is given as relation R and e is a constant expression given in relation V, a projection over $R \times V$ computes the result of the query. This concept is called loop-lifiting in [Grust 04]. If we assume that $(x_1, \ldots, x_n)$ is a set of context (XML) nodes and that e is an axis step, the "navigations" from each context node can also be computed by a join between the context table and the table containing the document. This join is evaluated by the so-called staircase join operator. Depending on the axis and the given context nodes, this special join operator optimizes axis evaluation by

1) pruning certain nodes from $(x_1, \ldots, x_n)$ (for example, in case of the descendant axis, a descendant x_i of a node x_j in the context sequence will not produce new nodes);
2) partitioning the document such that no duplicates are generated (i. e., when the target sets of each evaluation of e overlap); and
3) skipping such that only those parts of the document are touched containing possible target nodes of the axis step.

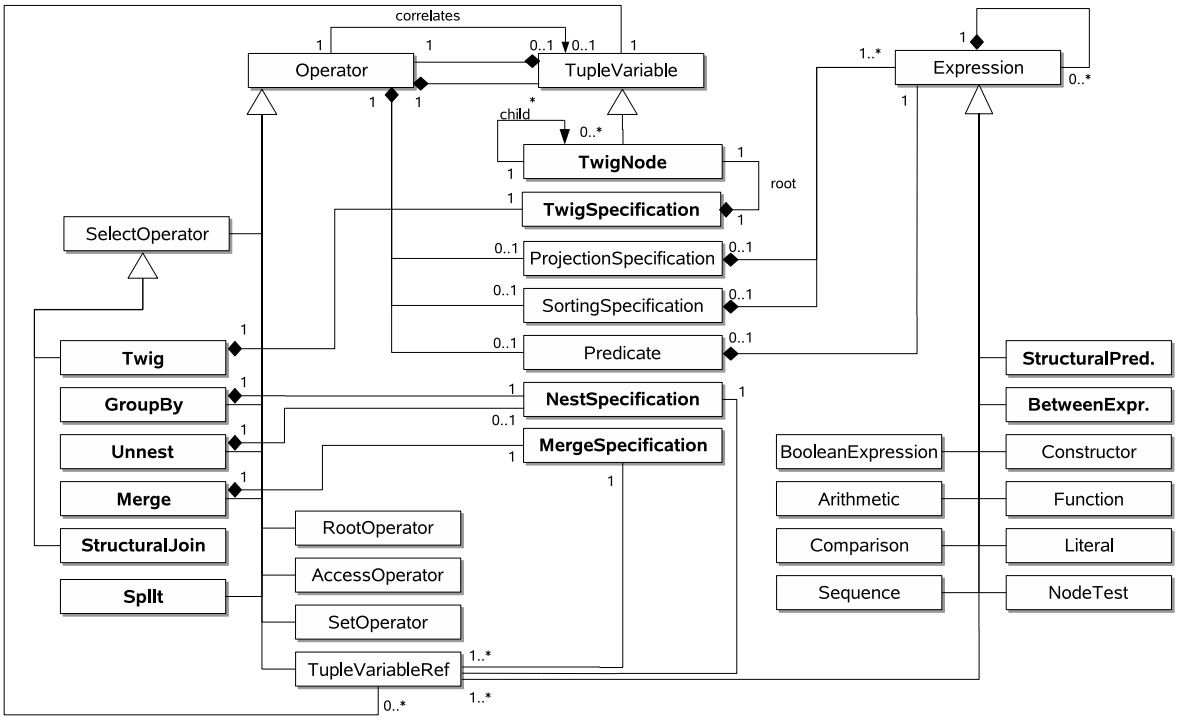
Based on loop-lifting and the staircase join operator, the translator generates quite complex algebraic expressions (often containing a large number of joins). For example, the plan for query *for $v in (10, 20) return $v + 100* contains three input relations and twelve operators [Boncz 05b]. To efficiently evaluate the generated plans, Pathfinder relies on the salient features of the main-memory-based MonetDB backend, which can execute joins quite efficiently. In standard relational systems, further data structures (e. g., partitioned B-trees) are required to guarantee good performance [Grust 07]. In MonetDB/XQuery, optimization is more relational-style [Boncz 06a]. The algebra does not contain any counterpart of a twig join algorithm and path indexes are not available. Therefore, we skip the rewriting discussion here.

5.10 Summary

In this chapter, we have seen how XML queries represented in the XML Query Graph Model can be algebraically rewritten. The chapter introduced nine rewriting rules, which primarily aimed at query unnesting, intermediate result size reduction, operator reduction, and twig discovery. Some rules required new operators and components to be introduced to XQGM, for example, the *structural join*, the *grouping* operator, or the *merge* operator. Besides new operators, also new expression types, like the *between* expression or the *structural predicate* have been introduced. Figures 5.19 and 5.20 provide an overview over the complete XQGM.

Considering related work, our approach shares the first three query processing stages with Galax (i. e., normalization, static typing, and simplification). Then, how-

Figure 5.19 The complete version of XQGM in UML



ever, in contrast to an algebraic representation, we chose to extend Starburst's query graph model. This approach is also implemented in DB2. Nevertheless, because the authors of that system do not publish any concrete internals on how XQueries are represented in the QGM, we are the first ones doing so. With Timber and with Galax, we share the notion of a tree pattern (or twig). However, our definition of a tree pattern is more expressive than those the two other systems. Thus, we are capable of compiling a larger XQuery fragment into tree pattern matching operators (supporting aggregate functions, optional subtrees, positional predicates, element constructors, etc.). Natix influenced the logical algebra developed for XTC. The problem with Natix was that the NAL algebra does not contain any bulk processing operators. With the introduction of the structural join and the unnesting techniques presented in [Mathis 07a, Mathis 07b], we lay the foundation for the rewriting rules shown in this chapter. MonetDB/XQuery and XTC do not have much in common, because the former system is purely relational.

To conclude this chapter, we want to show that the our rewriting rule set can operate on more complex queries (than our example queries). Therefore, consider the following query (which emerges from the XMark benchmark [Schmidt 02]):

```
for $p in doc("auction.xml")/site/people/person
let $l :=
  for $i in doc("auction.xml")/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)}</items>
```

Figure 5.21 shows this query in the final, rewritten stage. Because the query is more

complex and because XQGM is quite verbose in its graphical representation, the example does not fit on the page in a readable way. This is, however, not our intention. We just want to give a proof of concept, showing that our rule set can handle queries like the one above. For this, the display of the query's general structure in its various stages is sufficient. To the initial representation, all unnesting rules are applied. Only *one* correlated edge remains. Essentially, this edge stands for the value-based join in the above query. Since we can only unnest structurally nested subexpressions, this edge cannot be removed (with the provided rule set). Tackling these kinds of nested subexpressions is left open for future work. After twig discovery, the query has the shape presented in Figure 5.21. As you can see, the two structure-related subtrees of the query could be completely integrated into two twig operators. Note, the presented rule set is also able to treat all other queries in the XMark set this way, i. e., all structural predicates can be integrated into twig operators. This was only possible, because we extended the notion of twig operators in the literature with grouping, output expressions, output filters, etc.

Figure 5.20 Components of the complete XQGM version

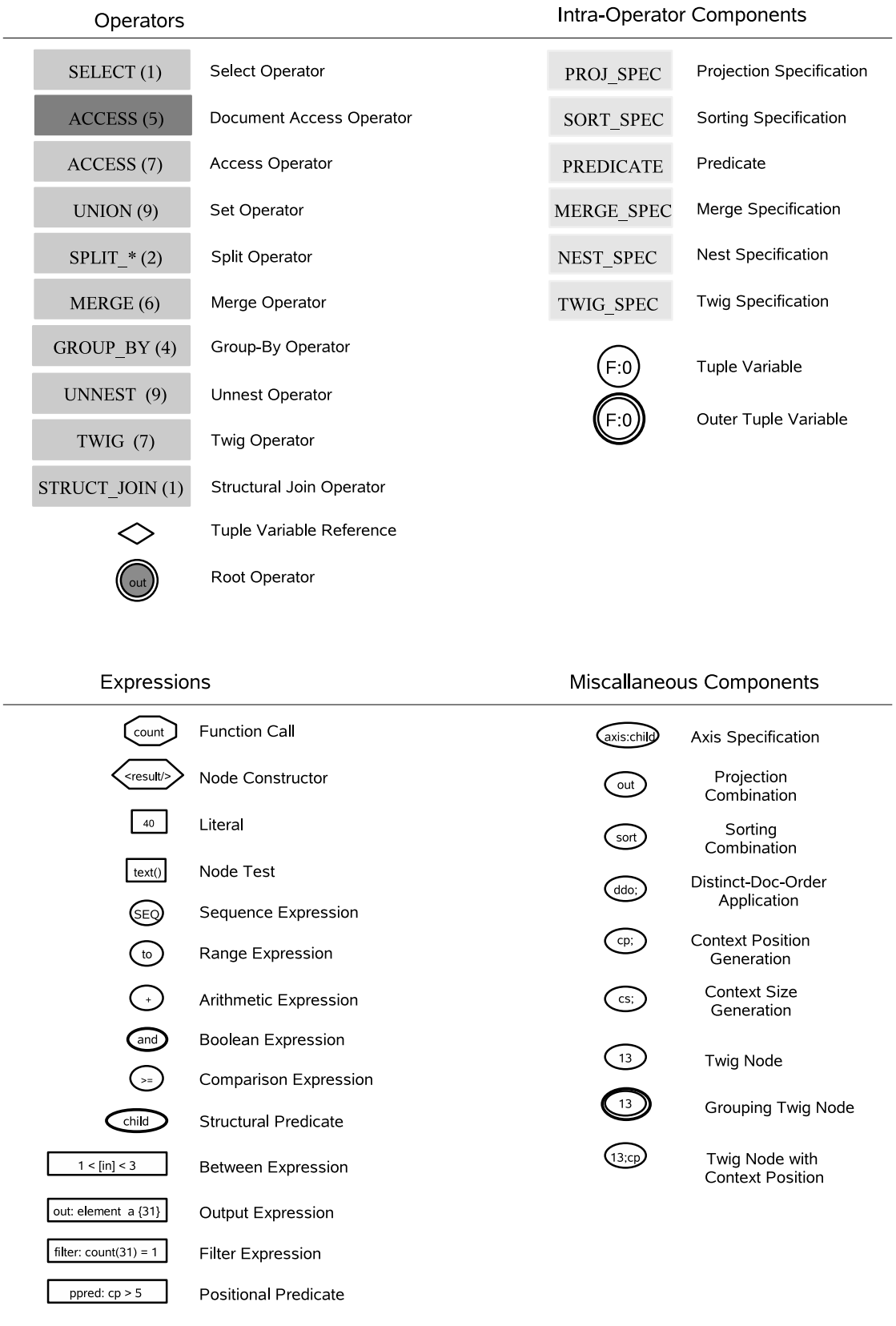
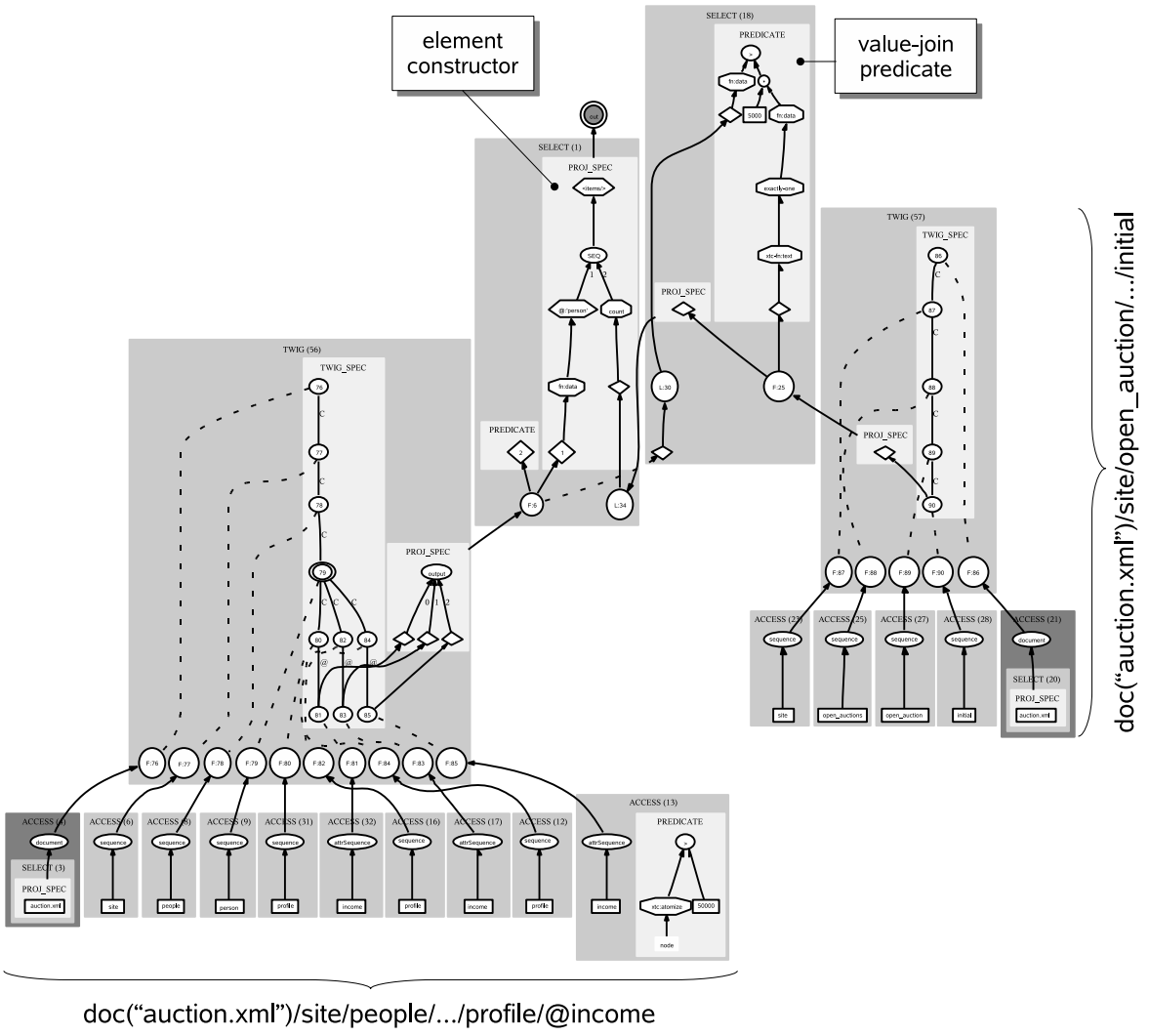


Figure 5.21 A complete rewriting example



Part III

Physical Aspects of XML Query Processing

Chapter 6 Document Storage

> Interestingly, according to modern astronomers, space is finite. This is a very comforting thought—particularly for people who can never remember where they have left things.
>
> Woody Allen

Two substantial XML-specific characteristics collide at the document store of an XDBMS: Structural XML complexity and different XML document processing (XDP) models. Because XML conquered so many different application areas, the structural complexity of XML instances heavily varies. For example, in some applications, XML replaces configuration files, resulting typically in only a few simply structured documents. In other applications, XML is used for electronic data interchange (e. g., SOAP [Mitra 07]), resulting in hundreds of thousands of small XML documents. In bioinformatics, protein sequences can be described using XML. The resulting documents often are multiple gigabytes large [UniProt 08]. Other applications use XML to encode tree-based data like, for example, the Treebank document, which contains information about the syntactical structure of a text [Miklau 09]. Many more examples exist, of which all somehow exploit the flexibility of XML, thus resulting in documents with very different structural complexities. For various reasons given in Chapter 1, it is meaningful to keep all these documents under the centralized control of an XDBMS, where the document store has to cope with structural complexity.

Over the time, various XDP interfaces have been standardized, e. g., DOM, SAX, and XQuery. And still new ones are developed like, for example, the *XQuery Update Facility* [Chamberlin 07a]. All these interfaces assume a different model of how XML documents are processed: DOM navigates, SAX scans, XQuery is declarative and therefore internally does both, and XQuery Update modifies. Clearly, an XDBMS has to provide all these standard interfaces to serve as a powerful backend for XML applications. But then the question arises how all these processing models can be supported efficiently. The document store is critical to the answer of this question because, in the end, all requests have to be evaluated over this component. This is in particular true for XML query performance, which heavily depends on how the document is organized on external memory.

XTC's node-oriented document store introduced in Section 3.3.2 can already handle different XDP models and structural complexities quite efficiently. However, optimizations—especially w. r. t. succinctness and XML indexing—are still possible,

making the following work worthwile. This chapter identifies critical aspects of native XML document storage by postulating a list of desiderata. The current document store is reconsidered and, based on the identified weak spots, solutions are presented. Furthermore, to address succinctness and indexing, the concept of *path-oriented XML storage* is developed.

6.1 Desiderata

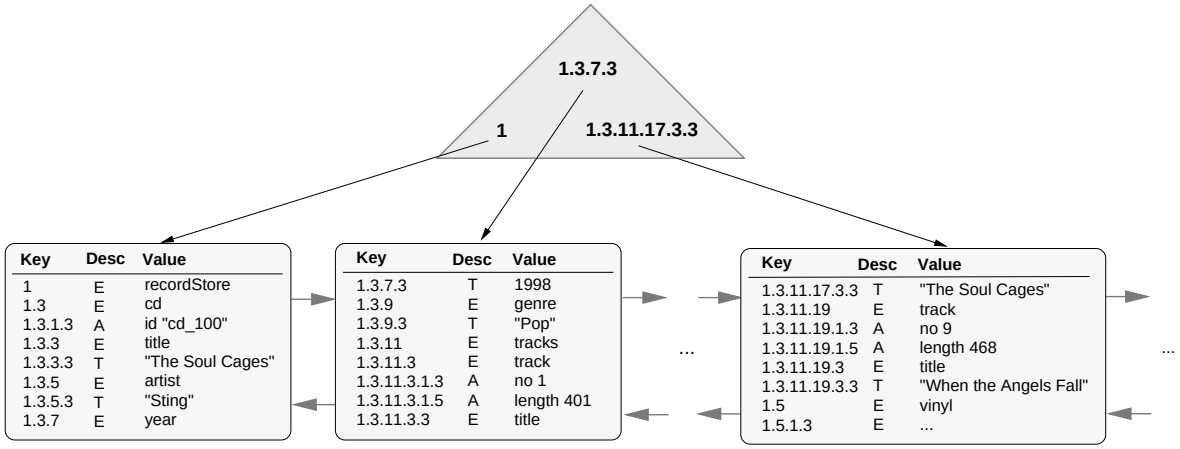
XML database systems are generic. Therefore, they should provide equally good performance and functionality for all shapes of XML documents and for all kinds of XDP models. This section presents a list of desiderata reflecting this consideration. As the name "desiderata" implies, we do not consider this list normative but, nevertheless, suggest that it is meaningful for many XML applications.

An XML document store should provide for the following eight characteristics:

1. *Efficient storage and reconstruction*. Because XML is a format for data interchange, XDBMSs frequently need to receive and emit XML data. Therefore, the document store has to provide fast storage and reconstruction facilities. Otherwise, the XDBMS would become a bottleneck.
2. *Navigational operations*. These operations are required not only to implement the DOM interface, but also to provide low-level operators for XQuery processing.
3. *Scan and subtree reconstruction*. A SAX parse is typically implemented by a document scan. But also for XQuery evaluation, scans are very important, e. g., for subtree reconstruction during result materialization.
4. *Modifications*. Of course, applications need to update XML data stored in an XDBMS. Therefore, the document store should provide means to modify single nodes, content, and subtrees.
5. *Round-trip property*. The round-trip property guarantees that a document can be reconstructed from the document store without any loss. This is in particular of importance for document-centric XML, for example, when legal contracts need to be stored.
6. *Document and collection support*. Documents might come in single instances of large documents or in large collections of small documents. No matter how, the document store should be able to efficiently manage the XML data.
7. *Succinctness*. A space-efficient document store not only saves external memory costs, but also leads to reduced I/O and logging and, therefore, better XDP performance.
8. *Indexing support*. For query processing, secondary path indexes are of particular importance, because documents and collections often are too large to completely load them into main memory for processing. Therefore, the document store should provide mechanisms that allow for cheap path index construction and maintenance.

Some readers now might miss two points that are often associated with XML storage, namely *XML schema validation* and *versioning* (i. e., keeping a history of modified subtrees in the store). We do not consider these points as physical properties

Figure 6.1 Storage of sample document *recordStore.xml* in a B*-tree



of an XML document store, but rather as logical ones that can and should be implemented in a layer on top of the physical representation. Note, in XTC, schema handling and versioning are two open issues that have not been explored yet.

6.2 Node-Oriented Storage Reconsidered

As introduced in Section 3.3.2, XTC implements a node-oriented storage model. Every node in the XML hierarchy is mapped onto a record (using the node's DeweyID as the record key) for external memory storage in a B*-tree. For convenience, the document store of our sample document from Figure 3.5 on Page 40 is repeated in Figure 6.1. In the following, this storage layout is reconsidered w. r. t. to the above requirements.

6.2.1 Storage and Reconstruction

Document storage is implemented using a third-party SAX parser. The parser does a single sweep through the document. For each node delivered, the necessary DeweyID, vocID or encoded content, and descriptor information is computed, mapped onto a record, and appended to a B*-tree with enabled prefix compression. Because the DeweyIDs are generated in document order for each node (and the records occur in correct key ordering), B*-tree insertion and prefix compression can be implemented very efficiently. In fact, because the records to be inserted are already sorted, the B*-tree can be built bottom up [Srinivasan 92]. As context information for prefix compression, only the last record stored before the current one needs to be memorized to derive the new suffix.

Document reconstruction can also be implemented very efficiently. Basically, only one scan over the doubly-linked list of leaf-pages of the document store is necessary. During the scan, each record can be decoded to a node using the previous record to reconstruct the node's DeweyID, and the record manager to decode the vocID.

On external memory, both operations, document storage and reconstruction, are sequential operations. Furthermore, they only require a small amount of main memory for the SAX parser and/or the opened set of (leaf) B*-tree pages. In case of storage, the maximum number of pages opened at a time is equal to the height h_t of the resulting B*-tree (bottom-up storage). For reconstruction, the number of pages pages opened at a time is 1.

Storage and reconstruction cannot be further optimized for node-oriented storage, therefore, no modifications to the current implementation of the document store are necessary.

6.2.2 Navigational Operations

As already sketched in Section 3.3.2, navigational operations are "translated" by the record manager to B*-tree index access methods. For a context node n and a B*-tree of height h_t, these translations can be sketched as follows (note, the implementation of the following algorithms is straightforward, which is why their pseudocode description is omitted here):

- *Parent*: The record manager calculates the parent DeweyID from n and calls the *getNode* method, which, in turn, gets the corresponding record from the index by a key lookup. Obviously, because DeweyIDs are prefix-based, no document access is necessary for the parent-ID calculation. To actually retrive the parent, a key-based B*-tree index lookup reading at most h_t pages is required. As an example, consider the calculation of the parent of the *title* node with ID 1.3.3. The parent ID is 1.3, for which a key-based lookup retrieves the *cd* node (see Figure 6.1).

- *First Child*: The record manager opens the index on n. Starting at the opened record, the index is scanned forward. If attributes exist, they are stored before the first child, therefore they have to be skipped. The scan stops on the first non-attribute record it finds. If the level of this record's DeweyID is of one larger than the level of n, the first child is found. Otherwise, the first child does not exist. The number of pages accessed to retrieve the first child in that way is $h_t + x$, where x is the number of pages containing the attributes of n. In most XML documents, the number of attributes for an element is typically small, which is why the first child resides on the same page as its parent most of the time. As an example, consider the calculation of the first child for the *cd* element with DeweyID 1.3. The index is opened at position 1.3, attribute 1.3.1.3 is skipped and element *title* with DeweyID 1.3.3 is returned as the first child.

- *Last Child*: As the previous method, the record manager opens the index on n. Then the index is scanned forward, buffering the last but one accessed non-attribute *child* record r_{l-1}. During the scan all records are skipped until an element record is found, whose DeweyID is not a descendant of n. If r_{l-1} has been assigned at least once, it contains the last child. Otherwise, the last child does not exist. Basically, the scan continues as long as it does not leave the subtree below n. When the scan stops, the last child is buffered in r_{l-1}. Similar to getting the first child, the number of pages accessed is calculated by $h_t + x$, where this time, x is the number of pages containing the subtree of n (and one additional node). As an example for this operation, consider the *cd* node again. The index is opened on the corresponding record. The scan basically skips all nodes starting with 1.3,

keeping the current child of 1.3 in r_{l-1}, i. e., 1.3.5, 1.3.7, 1.3.9, and 1.3.11. At node 1.5, the scan stops, because the subtree is consumed. The last child is then the *tracks* node at 1.3.11.

- *Next Sibling*: This method is similar to *getFirstChild*. Again the index is opened on n, and a forward scan is issued. This time, the scan stops on the first non-attribute record, whose level is smaller or equal to the level of n. If the level is equal, the next sibling is found. Otherwise, the next sibling does not exist. The cost of this operations is again h_t+x, where x is the number of pages that contain the subtree of n (plus one additional node). As an example, consider the *next sibling* method on the *cd* element at ID 1.3. The index is opened on that record and the scan skips all nodes with prefix 1.3 until node 1.5 (*vinyl*) is found.

- *Previous Sibling*: This method is implemented like the previous one, except that a backward scan is used. The cost is $h_t + x$, where x is the number of pages that contain the subtree of the previous sibling. Again, the number of pages may be prohibitively high and an alternative implementation will be shown.

Due to the scan-based implementation, the number of page access operations to find the *last child*, the *previous sibling*, and the *next sibling* is only bound by the document size in the current implementation. Of course, even though sequential scans are fast, this kind of access behavior has to be avoided. Therefore, we develop alternative solutions in the following.

Common to all three methods is the first step to locate the B*-tree record where the scan originates, i. e., the record of the context node n. This *open* operation already requires access to h_t pages. The idea is now to directly locate the page where the last child, previous sibling, or next sibling actually resides, instead of opening the index at the context node. To do so, special *open/get* methods over the B*-tree index have to be developed that do not operate on a direct key comparison, but on a *prefix comparison*. In particular, the following three methods are required:

- *Largest Prefix* (LP): Given a DeweyID d, this method opens the index on the last record having d as prefix. For example, on the document index depicted in Figure 6.1, LP(1.3) opens the index on record 1.3.11.19.3.3. This method can be implemented by a single top-down B*-tree traversal, just as an ordinary comparison-based open command, thus, requiring access to h_t pages.

- *Largest Prefix Right* (LPR): Given a DeweyID d, this method opens the index on the neighbor record at the right of last record having d as prefix. For example, on the sample document, LPR(1.3) opens record 1.5. The implementation of this method is almost equal to the implementation of LP. The search algorithm descends the B*-tree by following records with the largest prefix. If, however, at the leaf-page level, the record with the largest prefix is the last record in the page, the LPR resides on the next page. Thus, this method requires access to $h_t + 1$ pages in the worst case.

- *Smallest Prefix Left* (SPL): Given a DeweyID d, this method opens the index on the neighbor record at the left of the first record having d as prefix. For example, SPL(1.3.7) is 1.3.5.3. Similar to LP, this method requires to access h_t pages, because the leaf-page, where the SPL is stored, can be directly found.

Using these three methods, we can now sketch the index-based evaluation of the *last child*, *previous sibling*, and *next sibling* navigation steps. Note, for brevity, handling attribute nodes is omitted here. However, such an extension is straightforward:

- *Last Child*: For the DeweyID of context node n, LP delivers the last record r_l in the subtree of n. If no such record is found, the last child does not exist. Otherwise, record r_l is either the last child or r_l is a descendant of the last child. In the latter case, the DeweyID d of the last child can be inferred by truncating suffix divisions until the expected child level is reached. Now, the corresponding record for d can be loaded via the *getNode* method.

  As an example, consider the calculation of last child on the *cd* element with DeweyID 1.3. LP(1.3) delivers 1.3.11.19.3.3. With the level information, we can compute the DeweyID of the last child as 1.3.11. The record can be retrieved by a key-based record lookup. The number of pages accessed ranges between h_t and $2 * h_t$.

- *Next Sibling*: For the DeweyID of context node n, LRP delivers the neighbor record at the right of the last record having d as prefix. If such a record cannot be found or if the level of the record's DeweyID is not the same as n's level, the context node has no next sibling. Otherwise, the sibling is found.

  As an example, consider the calculation of the next sibling on *cd* with DeweyID 1.3. LPR(1.3) delivers 1.5, which is the next sibling. As a counter example, consider the next sibling of the *tracks* element with DeweyID 1.3.11. LPR(1.3.11) also returns 1.5, which is no sibling, because the level is not correct. The number of pages accessed ranges between h_t and $h_t + 1$, depending on LPR.

- *Previous Sibling*: This method is similar to the last child evaluation. SPL returns the left neighbor of the first record having the context node's DeweyID as prefix. This record is either the previous sibling or it is a descendant. In the second case, the previous sibling can be retrieved using the *getNode* method.

  As an example, consider the operation on the *vinyl* element with DeweyID 1.5. SPL (1.5) returns 1.3.11.19.3.3, which leads to 1.3 using the level information. The costs for this method range between h_t and $2 * h_t$ page references.

All in all, the costs to locate any node from a given context node ranges between h_t and $2 * h_t$, thus, in all cases, the costs are bounded. In practice, the buffer manager will keep many pages in main-memory anyway such that the number of physical page references is at most 1 for any navigational operation.

6.2.3 Scan/Reconstruction, Modifications, and the Round-Trip Property

Scan and subtree reconstruction are quite similar to document reconstruction sketched in the first point. To reconstruct a subtree for a given DeweyID, the entry page has to be found first, which is achieved by a key lookup in the B*-tree. Then a scan over leaf pages in the B*-tree is issued. In total, the number of accessed pages depends on $h_t + x$, where x is the number of pages that hold the subtree. For an ordinary document scan, the entry page is simply the left-most page among the leafs and the complete linked list of leaf pages is accessed. Because on external memory, these methods result in sequential operations, their implementation is very efficient.

Like navigations, document modifications are translated to B*-tree updates. Renaming a node/attribute or some content leads to a localization of the respective record in the B*-tree and a re-assignment of the vocID or the content. Thus, the operation requires h_t page access operations and one page write. Deleting a node n (and its subtree) results in deleting all records from the B*-tree, whose DeweyIDs

start with n's DeweyID. The well-known B*-tree balancing and merging algorithms [Comer 79] apply. To insert a node, its DeweyID has to be determined first. The insert position of that node depends on this DeweyID. The necessary rules to create the appropriate ID can be found in [Haustein 06a]. Given the ID, a record is created and inserted into the B*-tree, where, again, the well-known balancing and split mechanisms [Comer 79] apply. Because frequent insertions and deletions of whole subtrees can be anticipated, more efficient bulk maintenance methods can be implemented.

Of course, it should be possible to reconstruct a document stored in an XDBMS to its original form, i. e., the XDBMS should support the round-trip property. There are, however, several definitions of what "round-trip" actually means (e. g., [Cokus 05]). For example, round-trip could mean that a reconstructed document has to be byte-wise equal to its original form. Such a definition could be necessary, for example, when digital signatures over documents are created. Another definition requires the original and the reconstructed document to deliver the same data model instances, e. g., both documents have to have the same XML Infoset [Cowan 04]. The data store introduced above cannot provide for byte-wise equal documents, because it does not provide any way to encode intra-markup whitespace. The data-model definition however is supported. Because currently, XTC is a research prototype, only element, attribute, and text nodes are supported for storage (i. e., processing instructions, comments, namespace nodes, etc. are omitted). This is, however, only an implementational restriction and not a conceptual one.

6.2.4 Document and Collection Support

In this work, *collections* are defined as (unordered) sets of documents. Often, but not necessarily, collection documents are rather small and have a similar structure. The question, whether an XML application operates on one large document or one many small documents (in a collection), depends on the personal taste of the application developer. Nevertheless, the XDBMS has to support both designs. However, when large collections of small documents are stored in XTC's document store, a B*-tree is created for each of those documents leading to high memory fragmentation (because much space in nearly empty B*-tree pages is wasted). The situation gets even worse, when additional indexes are created on collections.

As a simple but efficient solution to this problem, XTC is enhanced to store collections in a single document, by adding a virtual root node under which all documents are appended. An additional entry in the (metadata) master document keeps track of the (DeweyID) position for a particular collection document in the document store. All access methods are rewritten to detect document boundaries (e. g., the document scan method). Furthermore, this collection store can be indexed (Chapter 7) just like any other document. Thus, the documents in a collection share indexing and storage space.

6.2.5 Succinctness

A space-efficient document store is mandatory because of storage and I/O costs. The XTC document store tries to minimize required storage space by various compression techniques (two of which will be briefly repeated for convenience): Ele-

Table 6.1 Characteristics of XML documents considered

Document Name	Description	Size [MB]	Elem./Attr. Nodes	Content Nodes	Vocabulary Size	DRTL Paths	RTL Path Instances
lineitem	LinteItems (TPC-H Bench)	32.3	1,022,977	962,801	19	17	962.801
uniprot	Universal Protein Resource	1,821.0	81,983,492	53,502,972	89	121	53,502,972
dblp	Computer Science Index	330.0	9,070558	8,345,289	41	153	8,345,289
nasa	Astronomical Data	25.8	532,967	359,993	70	73	371,593
treebank	English Records	89.5	2,437,667	1,391,845	251	220,894	1,391,845
xmark100	Auctions (Synth.)	100.0	1,776,202	1,060,215	77	451	1,412,762

ment and attribute names are not stored in their string representation. Instead a vocabulary is used that maps a name to a unique integer, i. e., the already introduced *vocabulary ID* (vocID), and vice-versa. Because the vocabulary of XML documents is typically small, only one to two bytes are required to encode element and attribute names. Another technique is prefix compression. Because two consecutive DeweyIDs in document order often have a large common prefix (see Figure 6.1), it is sufficient to store some delta information for following ID, instead of the full representation. As a result, in each page, only the DeweyID in the first record is stored completely. The IDs of the following records are encoded using delta information. A third technique aims at the efficient encoding of DeweyIDs. DeweyIDs should not be stored as strings or integer arrays, because these encodings cannot exploit the observation that DeweyIDs tend to contain small integer (division) values much more often than larger ones. Again, consider Figure 6.1 for an example. In such cases, Huffman encodings that represent often occurring values with short codes and seldom occurring values with longer codes are very efficient. An empiric evaluation on Huffman encoding for DeweyIDs can be found in [Haustein 05b, Härder 05b]. As a further technique (orthogonal to the already introduced), word-based and character-based text compression could be integrated in the document store. However, at the time of writing, this feature is not implemented yet. First considerations can be found in [Schmidt 07].

In general, redundancy is bad for storage, because of higher space consumption and the need to maintain redundant data. In XML, especially data-centric documents often contain a lot of redundant information. Consider, for example, the *recordStore.xml* document schematically presented as a tree in Figure 3.1 on Page 32. The first type of redundancy you can observe in XML is that every opening tag has a closing tag. This is however not a problem, because the node-oriented document store interpretes XML data as a tree and only stores nodes (and no opening and closing tags).

The second type of redundancy evolves from the mixture of metadata (i. e., structure) and content. As an example, consider all subtrees below a *track* element. They have the same structure (but different content). Furthermore, all *cd* subtrees have the same structure (but different content and cardinalities), and the structure of a *cd* element and a *vinyl* element is quite similar.

To give an impression on how much redundancy XML documents can contain, Table 6.1 shows some metrics of seven different documents. The five six documents are real-world examples [Miklau 09], whereas the last one (*xmark*) is a frequently used synthetic benchmark document [Schmidt 02]. All documents are single document instances of several megabytes up to nearly two gigabytes of size. The interesting columns are the three right-most ones. As you can see, the number of different element/attribute names (vocabulary size) is typically small (below 100 for all except *treebank*). Therefore, for most documents, a one-byte field storing the vocabulary ID (vocID) would be sufficient. What is even more interesting, is the ratio between distinct root-to-leaf (DRTL) paths and the number of root-to-leaf (RTL) path instances in the last two columns. This ratio is an indicator for the structural complexity of a document. Most shown documents have less than 200 distinct paths but several orders of magnitude more path instances. Those documents are typically data-centric. The only document with a higher ratio is *treebank*, which contains a syntactically annotated text and is document-centric.

As Table 6.1 illustrates, in data-centric documents, the arrangement of nodes on a path (over all path instances) is very similar, only quite a few combinations (i. e., distinct paths) exist. However, a node-oriented document store does not exploit this feature. Instead, path information is neglected and all nodes are simply mapped to records and stored in the B*-tree. This kind of redundancy can be reduced by a new storing concept called *path-oriented storage*. The idea behind path-oriented storage is to store paths instead of nodes, thereby *virtualizing* the inner document structure. The concept will be introduced in Section 6.3.

6.2.6 Indexing Support

Just like relational data, XML needs to be indexed for fast query evaluation, because scanning a complete document or relying on navigational primitives is often too expensive. While XML indexing will first be discussed in Chapter 7, we nevertheless try to anticipate how the document store can support indexing. In particular, the questions how secondary indexes can be built and maintained are of major importance. Note, in this section, the issues regarding indexing are only presented on a high level to motivate the requirements for the document store.

XML indexes can come in various forms, probably the most natural of which is a *path index*. Basically, a path index consists of an *index definition* which is stored in the metadata of the XDBMS, and a *physical store* that contains the indexed values—in this case, some path instances from the document. Typically, not all path instances of a document should be collected in the physical store, but only the ones of interest, e. g., to serve a particular query workload. Otherwise, maintenance costs would be intolerably high. To restrict the number of path instances, the index definition specifies some kind of path pattern. All paths matching this pattern are part of the index. Regarding index creation and maintenance, the question now arises 1) how the index builder can efficiently decide, which paths are contained in the index, and 2) how the document store can efficiently recognize which index requires maintenance after a document modification.

Let us consider index creation first. There are two possibilities how the necessary paths for an index definition can be found: by *query* or by *scan*. The first method interprets the index definition as a query which is evaluated over the document. The

results belong to the index. The second method scans the document (using a SAX scan) and keeps a *current path* information. Whenever, the current path matches the definition, it belongs to the index. Obviously, the node-based document store supports both creational methods. However, as will be shown in the next section, the path-based store already "knows" paths. Therefore, there is no need to keep the *current path* information anymore and a matching against the index definition can be omitted. Thus, the scan-based construction process is simplified.

For index maintenance, the advantages of a path-oriented store are even more obvious. Suppose an application deletes a subtree in a document for which a set of path indexes exists. Of course, the indexes need to be maintained. In a node-oriented document store, it is quite costly to identify the indexes requiring maintenance, because the *current path* information is not available. However, without the current path, it is not possible to decide which index is affected by an update. The only chance to do so is to reconstruct this information by navigating to all ancestors. Because a path-oriented document store is aware of paths, such a reconstruction is not necessary.

A prerequisite for all indexing methods is the possibility to uniquely identify nodes in a document, because an index inverts document nodes with certain properties into lists of node references. If these references were *physical*, each document modification would lead to an extensive index update, because inverted nodes located after the modified one(s) need to be updated. This, however, is prohibitively expensive and, therefore, not feasible. Actually, the point "Indexing Support" implies a logical node identification mechanism, such as the DeweyID concept introduced earlier.

6.2.7 Summary

In the previous Sections, the functionality of XTC's document store was reviewed w. r. t. to the list of desiderata given in Section 6.1. Table 6.2 summarizes the conclusions. The points *storage and reconstruction*, *scan and subtree reconstruction*, *updates*, and *round-trip* are sufficiently supported by the document store. For *navigational operations* and *collections*, some improvements were necessary to restrict/minimize the number of accessed/required pages. *Succinctness* and *indexing support* were identified as weak spots that will be addressed in the next Section. As we will see, the reduced storage overhead of the path-oriented scheme also improves the first four points in Table 6.2.

6.3 Path-Oriented Document Storage

XML has a tree-based data model and, intrinsically, paths play an important role in trees: In an XML document, the payload data is stored "at the end of a path", and also paths encode information; furthermore, queries over XML naturally follow paths. For example, most of the queries in the *XQuery Use Cases* [Chamberlin 07b] contain only child and descendant axes, i. e., they contain plain path queries.

As shown, the node-oriented storage approach encodes each node and maps it onto external memory. Because a node does not "know", on which path it resides, paths are only second-class citizens in the node-oriented model. To obtain path informa-

Table 6.2 Node-Oriented document storage summary

Requirement	Comment
Storage and Reconstruction	Documents stored and reconstructed node-at-a-time using single-sweep sequential operations. Document store (B*-tree) can be built bottom-up. Low main-memory consumption. No direct improvements possible.
Navigational Operations	Scan-based implementations of navigational operations require access to the complete list of linked B*-tree leaf pages in the worst case. Index-based implementations require $2 * h_t + 1$ page access operations in the worst case; therefore more robust.
Scan and Subtree Reconstruction	Scan and subtree reconstruction single-sweep sequential operations along leaf pages of B*-tree. Subtree reconstruction bounded by $h_t + x$ page access operations, where x is the number of consecutive pages the subtree is stored in. No direct improvements possible.
Updates	Subtree deletions/insertions result in (bulk) B*-tree deletions and insertions, where well-known tree balancing algorithms apply. Renaming a node or content modification requires record fetch and update to record. No direct improvemens possible.
Round-Trip Property	Byte-wise round trip is not possible, however, data-model-based round-trip is implemented.
Documents and Collections	Collections are stored as virtual document, thus collection documents share storage and indexing space.
Succinctness	Compression opportunities based on structural similarities are *not* exploited.
Indexing	Index creation with current path information possible. Index maintenance expensive due to ancestor path reconstruction.

tion, the document needs to be accessed (even though the DeweyIDs of all nodes on the paths are known).

Because paths occur very frequently in query processing, we will see that their optimized management even (or already) in document storage is of major importance. To address this consideration, the *path-oriented storage model* promotes paths to first class citizens. This promotion is done in a careful way such that the value of nodes is not degraded.

The basic idea behind path-oriented storage is simple: Assume a data structure that structurally summarizes all paths in a document. In the following, this data structure is called *path synopsis*. For a large group of XML documents, such a path synopsis would be typically small: Table 6.1 reveals that the number of distinct paths, i. e., the number of paths in such a synopsis is less than 500 for all considered documents except one. The key ideas is to establish a link between a node on one side and the node's path on the other side. This can be achieved by the use of a path synopsis.

A node (and its DeweyID) together with its path can be used as a kind of coordinate in the document. Not only the DeweyIDs of the node's ancestors can then be inferred, but also the ancestor's names. This mechanism allows a kind of "upside-down" storage, in which the inner structure of the document is not explicitely stored, but virtualized.

In the following, the path synopsis and the path-oriented storage model are introduced, for which then the list of desiderata posed in Section 6.1 is considered.

6.3.1 The Path Synopsis

In the following, we content ourselves with only an informal description of the path-synopsis concept, given in Definition 1 below. Actually, a path synopsis is the same as a DataGuide [McHugh 98] or a 1-Index [Milo 99]. However, to introduce the necessary concepts and terms, we nevertheless provide our own definition. The first criterion states that a path synopsis only contains information about element and attribute nodes (and not about text nodes). Therefore, only structural information is stored in a path synopsis. Criterion number 2 actually means that, for every path in the synopsis, there is at least on instance in the document, i. e., no paths are contained that do not occur in the document. The last criterion states that, for each path in the document, there is exactly one path in the synopsis, i. e., the paths in the synopsis are distinct.

Definition 1 (Path Synopsis (PS)) *A path synopsis is a tree structure containing all distinct paths of a document. Let p_D be the path synopsis for document D, then the following criteria hold:*

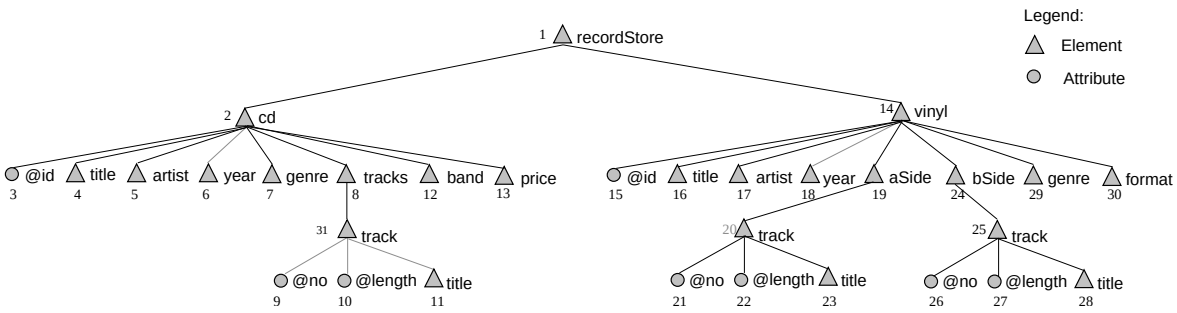
1. *The nodes n of a path synopsis carry a node name l, where such a node name is either an element name from D or an attribute name. In the latter case, the name is prefixed by an "@". A path synopsis does not contain text nodes.*
2. *Let n_p be a node in p_D and let $l_1, l_2, \ldots, l_n$ be the node names on the path from the root of p_D to n_p. Then there is at least one element (or attribute) node n_D in D having the same names $l_1, l_2, \ldots, l_n$ (or $l_1, l_2, \ldots, @l_n$) from the root of the document D to n_D.*
3. *Let n_D be an element (or attribute) in the document and let $l_1, l_2, \ldots, l_n$ (or $l_1, l_2, \ldots, @l_n$) be the node names on the path from the root of D to n_D. Then there is exactly one node n_p in p_D having the same names $l_1, l_2, \ldots, l_n$ (or $l_1, l_2, \ldots, @l_n$) from the root of p_D to n_p.*

As an example, consider the path synopsis for the *recordStore.xml* sample document depicted in Figure 6.2. You can verify the above criteria by comparing the path synopsis with the complete document depicted in the Appendix. Further on, note that the order of the paths in the path synopsis is not significant. For example, in the synopsis, the *tracks* node appears before the *band* node (in document order), however, in the depicted document, the order is exactly the other way around. Furthermore, note the names of siblings in the path synopsis is usually a superset of the names of siblings in the document. For example, a *cd* element usually has either a *band* or an *artist* child element, but not both. In general, the definition of the path synopsis does not make any assumptions on the order of the nodes or on the existence of siblings in a particular document. Assumptions are only made on the correspondence between paths in the synopsis and in the document.

A path synopsis can be seen as a kind of up-to-date structural schema description of the document. The use of this concept is manifold. In this work, the path synopsis will be used to store documents and to create indexes over them. Additionally, a path synopsis can serve for example as a container for statistical data, as a means to define data consistency, or it can be used to prune path expressions from queries, for which the synopsis reveals that they will produce an empty result.

In XTC, the path synopsis is a main-memory data structure with an external memory encoding for persistence. For easy reference, every node in the synopsis has

Figure 6.2 Path synopsis of the *recordStore.xml* document



a node label. In contrast to documents, where order plays a role and where the DeweyID mechanism is required, the path synopsis is order oblivious and, therefore, a simple integer value to reference a node is sufficient. Because every path p in the path synopsis subsumes several document paths p_{D_i} (Criterion 2 of Definition 1), we call p a *path class* and the the documents paths its *path instances*. The node labels in the path synopsis are called *path class references* or PCRs, for short. Note, also PCRs have no particular order. For example, the PCR of the left-most *track* node in the path synopsis is larger than its child nodes.

The size of the path synopsis obviously depends on the structural complexity of the document. In the worst case, the path synopsis is as large as the document. However, as the structural analysis in Table 6.1 revealed, the ratio between path classes and path instances is often very low and path synopses are typically small.

6.3.2 The Store

To promote path information to first-class citizens in our storage model, all that needs to be done is to connect path information to nodes such that every node "knows" on which path it resides. This can be accomplished by storing the node's corresponding PCR together with the node information in the document store. Then, for every node in the document, it is possible to calculate the DeweyIDs and the names of its complete ancestor path. As an example, consider our sample document depicted in Figure 6.3 on. Assume, we regard the *title* node with DeweyID

Figure 6.3 Path-oriented document storage

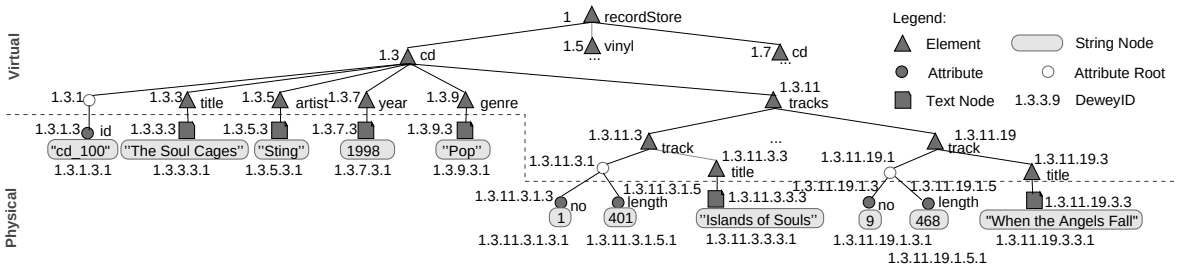
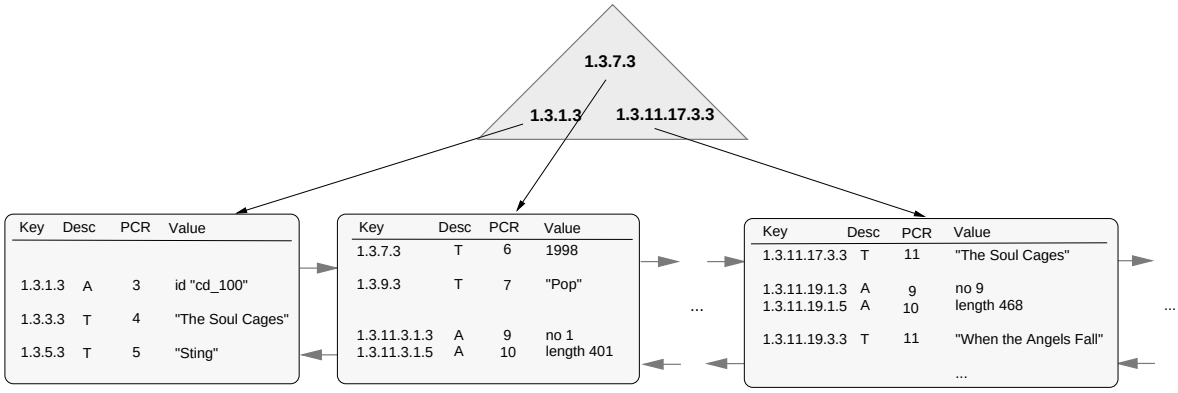


Figure 6.4 The path-oriented document store for document *recordStore.xml*



1.3.11.3.3. In the path synopsis, this node has PCR 11. Together with the path synopsis, we can then compute the ancestor path of this node as the following list of name-DeweyID pairs: (*track*, 1.3.11.3), (*tracks*, 1.3.11), (*cd*, 1.3), (*recordStore*, 1).

Storing the path information together with every node results in higher memory consumption by the document store. Although a PCR size of two bytes (to encode up to 65,536 paths) is sufficient for many documents, this size has to be multiplied by possibly millions of nodes in the document. To remedy this situation and to even *save* storage space, we can simply skip the storage of inner nodes and only store leaves. This is possible, because, as shown above, the nodes on an ancestor path can be recomputed using DeweyIDs and the path synopsis. This is the basic idea behind the path-oriented storage scheme. For example, consider Figure 6.3 again. Everything that needs to be stored in the depicted subtree are the nodes below the dashed line, their PCRs, and the path synopsis of the document. All inner nodes can be recomputed, when required. As shown, only string nodes need to be stored. Because a string node always ends on division "1", this last division can be skipped to save space.

The resulting document store which is also based on a B*-tree is depicted in Figure 6.4. To facilitate the comparison of the path-oriented document store with the node-oriented document store for the same document (shown in Figure 6.1 on Page 159), the path-oriented store is illustrated using gaps where previously records were stored. For each text (T) or attribute (A) node in the document, a record is contained in the store. In the new record format, besides the *key*, *desc*, and *values* components, the *PCR* is also stored. For attributes, the PCR from the path synopsis can be directly used (because attributes are directly represented there). For text nodes, the PCR of their containing element is used, because text nodes are not directly represented in the path synopsis. Note, this storage mode also works for mixed content and empty elements. In the latter case, an empty (dummy) text string is generated to encode the element.

All in all, less records are required, because inner elements are omitted in path-oriented storage. Thus, path-oriented storage provides an opportunity to reduce the external memory consumption. This space reduction will be empirically measured in Chapter 9. In the following, we consider the given list of desiderata and show

Listing 6.1 The storage content handler

```
class StorageContentHandler extends DefaultHandler implements ContentHandler
{
  Array<int>     divisions = {-1, ..., -1}; // for DeweyID creation
  int            level = 0;                 //      -- "" --

  Record         dummyRecord;       // to buffer the dummy node
  SynopsisNode   psNode;            // to create the synopsis
  BSTreeBuilder  bsTreeBuilder;     // to create the document store

  // SAX methods ...
}
```

that all operations on the node-oriented document store are also supported by the path-oriented version.

6.3.3 Storage and (Subtree) Reconstruction/Scan

As in the node-oriented approach, we will see that the document can be stored and reconstructed in a single sweep through the document or over the leaf pages of the document index. However, because inner nodes are virtualized, less data needs to be read or written. In contrast to the previous approach, the algorithms for storage and reconstruction require a little more explanation. Therefore, they are presented in the following.

Storage

For storage, a third-party SAX parser is used, which reads the document and issues the call-back methods of the storage content handler. The storage content handler, in turn, simultaneously computes the necessary DeweyIDs, builds the path synopsis, and constructs the document store. To facilitate the presentation, only the treatment of element and text nodes is discussed. Storing attributes only requires some simple extensions to the presented algorithm.

Before the algorithm is presented in detail, consider the following: In the node-oriented storage model, a record (DeweyID-value pair) is created and immediately written to the document store for every node reported by the SAX parser. Because only leaf nodes and no inner elements are stored in the path-oriented storage mode, this process should intuitively become simpler. However, the occurrence of empty elements slightly complicates the process: As introduced above, an empty element requires an empty (dummy) text element to be written into the document store. However, at the time an element is reported by the SAX parser, it is not known, whether the element is empty or not. This question can only be answered by the following SAX call, which signals whether a child element or text node was found or not. Therefore, a one-node lookahead is required. With that, the member variables keeping the state of the storage content handler in Listing 6.1 become clear.

The `divisions` array and the `level` variable keep track of the value for the current DeweyID. The array is initialized with a set of *-1* values. Both, `divisions` and `level` are modified when the start or the end of an element is reported. The `dummyRecord` variable is the one-node–sized buffer introduced above. It can con-

Listing 6.2 The *startElement* method for path-oriented document storage

```
public void startElement (String namespaceURI, String localName, String qName)
{
  // compose element name
  String elementName = createElementName(namespaceURI, localName, qName);

  // calculate the divisions for the current DeweyID
  divisions[level] += 2;
  DeweyID id = createDeweyID(divisions);

  // maintain the path synopsis
  if(psNode.hasChild(elementName))
    psNode = psNode.getChild(elementName);
  else
    psNode = psNode.append(createSynopsisNode(elementName));

  // check and optionally store the dummy record
  if(defined(dummyRecord))
    storeDummyRecord(id);

  // assign current element to dummy record
  dummyRecord = createRecord(id, pcr, "");

  // increase level
  level++;
}
```

tain a DeweyID, a PCR, and a value. To build the path synopsis and to generate PCRs, the `psNode` variable keeps track of the current position in the synopsis. Finally, the `bsTreeBuilder` buffers the records and materializes the document store as a B*-tree.

The following code contains some auxiliary methods that are more or less self explaining. Because two particular methods will also be used later on, we will explain them in more detail: the *defined* method checks whether the passed object has a value and the *undef* method sets the value of a passed object to *undefined*[1]. The pseudocode in Listing 6.2 shows the actions executed upon element start.

As a first action, the new element name is created out of the three name particles passed to the start method. Then the method computes the division of the current level. If the current level is visited the first time, the division value becomes 1, otherwise, the previous division value at the current level is increased by two. With the `divisions` array, the new DeweyID can be initialized. In the third step, the path synopsis is constructed. If the current synopsis node has already a child with the element name (from previous elements), this child is assigned to the current `psNode`. Otherwise, the new child node has to be created (which internally generates a new PCR).

The next two blocks handle empty elements. The latter block simply stores the DeweyID and PCR information of the current element (together with an empty text string) in the dummy record. Assume that prior to this *startElement* call, an element occurred. Then this element might be empty and had to be stored first, before a new one can be assigned. This is the task of the first block. The check for the condition, if the dummy record actually needs to be stored, depends on the current DeweyID

[1] In Java, *defined* would check for NULL, while *undef* would assign NULL.

Listing 6.3 The *endElement* method for path-oriented document storage

```
public void endElement(String namespaceURI, String localName, String qName)
{
  // reset the division value for this level
  divisions[level] = -1;

  // navigate path synopsis up
  psNode = psNode.getParentNode();

  // check and optionally store previous record
  if(defined(dummyRecord))
    storeDummyRecord(unset(createDeweyID(divisions)));

  // decrease level
  level--;
}
```

and is hidden inside the *storeDummyRecord* method, which will be presented below. As a last action, the current level has to be increased.

The code snippet in Listing 6.3 shows the corresponding code for the *end element* event, which basically mirrors the actions from the *start* method: First, the current division is set such that the next time this level is reached, value 1 will be assigned. Then, the parent of the current path synopsis node is obtained. In the third block, again empty elements need to be handled, because an empty element might have occurred before the current call. Note, this time, the current DeweyID does not play a role, which is why an undefined value is passed. The reason is that, whenever an empty element occurred before an *endElement* call, this element needs to be stored. As a last action, the current level is decreased.

Because only leaf nodes produce a record in path-oriented storage, both so-far discussed methods only keep track of the current state, but do not write to the document store. The *characters* method shown in Listing 6.4, however, does.

As before, the DeweyID is computed first. Then the check and storage of empty elements takes place again. Finally, the current PCR is obtained and a record is

Listing 6.4 The *characters* method for path-oriented document storage

```
public void characters(String characters)
{
  // calculate the divisions for the current DeweyID
  divisions[level] += 2;
  DeweyID id = createDeweyID(divisions);

  // check and optionally store previous record
  if(defined(dummyRecord))
    storeDummyRecord(id);

  // get current PCR and store record
  int pcr = psNode.getPcr();
  bsTreeBuilder.appendRecord(createRecord(id, pcr, characters));
}
```

Listing 6.5 The *storeDummyRecord* method for path-oriented document storage

```
public void storeDummyRecord(DeweyID id)
{
  if(!defined(id) || !dummyRecord.getDeweyID().isParentOf(id))
  {
    int pcr = dummyRecord.getPcr();
    int id  = dummyRecord.getID() + ".3";
    bsTreeBuilder.appendRecord(id, pcr, "");
  }
  undef(dummyRecord);
}
```

passed to the B*-tree builder.

Because empty elements can occur before text nodes, before other elements, and inside other elements, their handling was an issue for all three methods above. The condition, when an element is actually empty is coded into the following method shown in Listing 6.5. Only when the `dummyRecord` is not a parent of the passed DeweyID (for another element or text node), the buffered record actually belongs to an empty element. As explained above, the passed DeweyID is undefined in case of the *endElement* method and, thus, also requires the storage of an empty element. The PCR of the written dummy record is the PCR of the empty element. Its DeweyID however belongs to the (contained empty) text node (therefore, a `".3"` division has been appended). In the final method, the actual B*-tree/synopsis is built and the necessary metadata is written during the *endDocument* method shown in Listing 6.6.

Obviously, the space complexity of the algorithm is bounded by the number n of nodes in the document: Let e be the number of elements (of which x are empty), let a be the number of attributes and let t be the number of text nodes (i. e., $n = e+a+t$). Furthermore, let d be the maximum length of a path in the document. Then the space complexity of the divisions array is $O(d) = O(n)$, the space complexity of the path synopsis is $O(e) = O(n)$, and the space complexity of the document store (B*-tree) is $O(a + t + x) = O(n)$.

The time complexity of the algorithm is bounded by $O(n \log n)$: the time complexity to handle the divisions array and empty elements is $O(n)$; also the time complexity to build the B*-tree is $O(n)$ (note, no record sorting is required, because the records occur in the right order). However, the worst-case complexity to build the path-

Listing 6.6 The *endDocument* method for path-oriented document storage

```
public void endDocument() throws SAXException
{
  bsTreeBuilder.materialize();
  // register document (B*-tree) in metadata

  Synopsis synopsis = createSynopsis(psNode);
  // register path synopsis in metadata
}
```

synopsis is $O(n \log n)$, because of the *hasChild* and *getChild* method calls contained in method *startElement*. Each of these methods needs to search for an already existing path synopsis node with the given name. This search requires $O(log\, n)$ operations and is issued for each element. Therefore, the overall complexity is $O(n \log n)$. However, the number of children will be bounded in many documents and efficient hashing schemes can be used to implement the above two methods. This is why also the path synopsis construction will take place in near linear time in most cases.

(Subtree) Reconstruction/Scan

For document reconstruction, the XDBMS has to fetch the necessary nodes from the document store (in document order) and convert them into the external representation as a character string. Given a node list in document order, this conversion is straightforward, which is why it is not presented in this chapter. Here, only the generation of the node list from the document store is discussed.

In node-oriented storage, retrieving this node list is simple: The leaf pages of the document index are read sequentially and, for each record found, a node is produced. In the path-oriented storage mode, only records for leaf nodes and attributes are stored. Therefore, the remaining virtualized elements have to be reconstructed with the help of the path synopsis and the stored PCRs. The pseudocode if Listing 6.7 presents an algorithm that reconstructs all element and text nodes (again, to facilitate the code, attribute nodes are omitted).

For the iteration over the B*-tree records during reconstruction, a set of context variables is necessary. Variable `record` represents the current record delivered by the B*-tree. Remember that this record is always a leaf node and contains information about its DeweyID, its PCR, and its value (which are immediately assigned to the corresponding variables). During reconstruction, a particular DeweyID that belongs to the *least common ancestor* (LCA) is required. Initially, this LCA is set to the DeweyID of the root node. The root ID is computed by the *getAncestor* method, which returns the ancestor of a DeweyID at a particular level (here, at level 0).

As a first result node, the root node is added to the `result` list. To compute this node, its name has to be resolved in the path synopsis by specifying a path class (via a PCR), and the level of the node name to be returned.

The while loop iterates over all records. As a first action, all elements below the current least common ancestor (`lcaLevel + 1`) and above the current leaf node (`level < id.getLevel()`) are reconstructed and added to the result. If the current record is not a dummy record (i. e., if it does not encode an empty element), the record value (content) needs to be added as a text node.

Finally, the iteration context needs to be re-assigned, during which the new record, its PCR, its DeweyID, and its value are again decoded. Then, the new least common ancestor of the old record and the new record has to be calculated. Like calculating ancestors at a particular level, the calculation of the LCA between two DeweyIDs is a straightforward operation that operates on the ID's prefixes. You may have noticed that no PCRs are assigned to the generated nodes. The necessary code for that was omitted to keep the listing simple.

Assuming that a name can be resolved by the synopsis in constant time and that ancestor computations on DeweyIDs are also constant (which both are in practical implementations), the space and time complexity of the algorithm is bounded by

Listing 6.7 The node reconstruction algorithm

```
List<Node> reconstructNodes(BSTree bsTree, Synopsis synopsis)
{
  List<Node> result;

  // reconstruction context variables
  Record record = bsTree.getNextRecord();
  DeweyID id    = record.getDeweyID();
  int pcr       = record.getPcr();
  String value  = record.getValue();
  int lcaLevel  = 0;
  DeweyID lca   = id.getAncestor(lcaLevel);

  // add the root node
  String nodeName = synopsis.getName(pcr, lcaLevel);
  result.add(createNode(lca, nodeName));

  // iterate over all records
  while(defined(record))
  {
    // reconstruct and add inner elements from lca on
    for(int level = lcaLevel + 1; level < id.getLevel(); level++)
    {
      DeweyID nodeID = id.getAncestor(level);
      nodeName = synopsis.getName(pcr, level);
      result.add(createNode(lca, nodeName));
    }

    // add the content
    if(!isDummyRecord(record))
      result.add(createNode(id, value));

    // re-assign reconstruction context
    record = bsTree.getNextRecord();
    if(defined(record))
    {
      pcr           = record.getPCR();
      String value  = record.getValue();
      DeweyID tmpID = record.getDeweyID();
      lca           = id.calculateLCA(id, tmpID);
      lcaLevel      = lca.getLevel();

      id = tmpID;
    }
  }
  return result;
}
```

$O(n)$ (because each node is exactly decoded once).

Embedding the Reconstruction Algorithm in XTC

The presented algorithm can not only be used to reconstruct the document, but also to reconstruct fragments (for example, during result materialization) and to implement the SAX parser interface of the XDBMS, i. e., it can be used in general, when a (partial) document scan is required. In practice, it is necessary to restrict the size of the resulting node list, for example, when nodes need to be transferred to the client for a SAX scan. Therefore the scan is split up in several partitions. Because document scans are part of the physical XML algebra, the interface to such a partitioned scan is introduced here. The *getScanPartition* method has the following parameters:

- `DeweyID root`: Only elements in subtree below the root are returned.
- `DeweyID start`: The scan starts at this DeweyID.
- `DeweyID end`: The scan ends with this DeweyID.
- `NodeTest filter`: Only elements that fulfill the node test are part of the result.
- `Integer resultSize`: Restricts the number of nodes returned.
- `Boolean self`: If true, the first node (root or start) is part of the result.

At least one of the first three parameters has to be specified to get a starting point for the scan. The filter and the result size may remain unspecified. Internally, the algorithm operates like the one presented above and a similar version for node-oriented storage also exists.

6.3.4 Navigational Operations

Navigational operations are necessary to implement the DOM interface, but, because physical XML operators can be implemented using navigations, they also play a role for the query processor. As we will see, the evaluation of the five navigational primitives *parent*, *first child*, *last child*, *previous sibling*, and *next sibling* requires only a single top-down traversal of the document index and a single access to the path synopsis. In the following, we only consider element nodes. The extensions of the presented base algorithms to support further node types is trivial.

Listing 6.8 The *parent* method

```
Node getParent(Node contextNode, Synopsis synopsis)
{
  Node    resultNode;

  DeweyID contextID = contextNode.getDeweyID();
  int     pcr       = contextID.getPcr();

  DeweyID parentID = contextID.getParent();
  if(defined(parentID))
  {
    int     parentPCR = synopsis.getParentPcr(pcr);
    String  name      = synopsis.getName(parentPCR);

    resultNode = createNode(parentID, parentPCR, name);
  }
  return resultNode;
}
```

Given a context node, the algorithm in Listing 6.8 computes its parent node. Because a node contains its path information, this code simply calculates the parent node with the help of this path information and the synopsis. Because no document access is required, this method is very cheap.

For the implementation of the remaining operators, the prefix-based B*-tree access methods *Largest Prefix* (LP), *Largest Prefix Right* (LPR), and *Smallest Prefix Left* (SPL), introduced in Section 6.2.2 on Page 161, are also required here. Additionally, a similar method named *Smallest Prefix* (SP), which returns the left-most record with the prefix of the passed ID is implemented over the B*-tree. As LPR, this method requires at most $h_t + 1$ page access operations to find the record.

The *first-child* algorithm depicted in Listing 6.9 makes use of SP. Using the DeweyID

of the context node, the record with the smallest prefix of this ID is obtained from the B*-tree using SP. If this record's DeweyID is the same as the context ID, no first child exists and the method returns an undefined value. Otherwise, the first child must reside at the level below the current context node. Using the path synopsis, the child can then be reconstructed. As an example, consider the *first-child* method

Listing 6.9 The *first-child* (*last-child*) method

```
Node getFirstChild(Node contextNode, Synopsis synopsis)
{
  Node resultNode;

  DeweyID contextID = contextNode.getDeweyID();
  Record record = bsTree.smallestPrefix(contextID);

  DeweyID smallestID  = record.getDeweyID();
  int     smallestPCR = record.getPcr();

  if(smallestID != contextID)
  {
    int     childLevel   = contextID.getLevel() + 1;
    DeweyID childID      = smallestID.getAncestor(childLevel);
    int     childPCR     = synopsis.getPcrAtLevel(smallestPCR, childLevel);
    String  name         = synopsis.getName(childPCR);

    resultNode = createNode(childID, childPCR, name);
   }
   return resultNode;
}
```

on context node 1.3.11 in Figure 6.3 on Page 169 and the document store illustrated in Figure 6.4 on Page 170. The SP method returns record 1.3.11.3.1.3. Because this ID is different from the context ID, a first child exists, which has to reside one level below 1.3.11. Therefore, the ID of the first child is 1.3.11.3 and together with the PCR of the opened record and the synopsis, the *track* element can be reconstructed.

The method to retrieve the last child works analogous. The only difference is that the index is opened on the record with the largest prefix (LP). All remaining conditions and actions remain the same. Therefore, the pseudocode for this method is not presented separately. As an example, consider the computation of the last child on context ID 1.3.11. The LP method returns record 1.3.11.19.3.3 as the largest prefix from the document store, which is not equal to the context ID. Therefore, the last child exists and has ID 1.3.11.19. Together with the synopsis, the last `track` element can then be recomputed.

The algorithm to compute the previous sibling is illustrated in Listing 6.10. Here, the SPL method is used to retrieve the record left to the one with the smallest prefix in the document store for the given context ID. From this record on, the ancestor at the same level as the context node is computed. If such a node exists, it is checked, whether it is really a sibling (it could also be a node in an unrelated subtree that happens to be at the correct level). If this check is passed, the sibling can be computed using the synopsis.

The algorithm to find the next sibling is again analogous, except that the record right to the one with the largest prefix (LPR) is retrieved from the document store.

If we analyze the access behavior of the presented navigational primitives, we can

Listing 6.10 The *previous-sibling* (*next-sibling*) method

```
Node getPreviousSibling(Node contextNode, Synopsis synopsis)
{
  Node resultNode;

  DeweyID contextID = contextNode.getDeweyID();
  Record record    = bsTree.smallestPrefixLeft(contextID);
  int contextLevel = contextID.getLevel();

  DeweyID siblingID = record.getDeweyID().getAncestor(contextLevel);
  if(defined(siblingID) && siblingID.isSiblingOf(contextID))
  {
    int smallestPCR = record.getPcr();
    int siblingPCR  = synopsis.getPcrAtLevel(smallestPCR, contextLevel);
    String name     = synopsis.getName(siblingPCR);

    resultNode = createNode(siblingID, siblingPCR, name);
  }
  return resultNode;
}
```

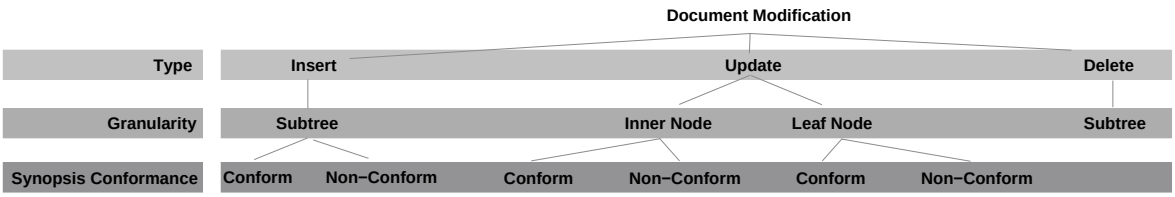
infer that at most $h_t + 1$ access operations to pages are required for any method, which is a improvement to the $2 * h_1 + 1$ operations required in the worst case on the node-oriented document store.

6.3.5 Modifications

Modifications on the node-oriented document store can be implemented fairly straightforward, because every record corresponds to exactly one node. On the path-oriented store, the synopsis captures some kind of document *schema*, which might change if the document is updated and some nodes are not explicitely stored. Therefore, the problem gets slightly more complex (but only conceptually and not computationally). For updates in path-oriented storage mode, the following three "dimensions" of the problems have to be considered:

1. The *type* of the update, which can be an insertion, an update, or a deletion;
2. The *granularity* of the update, which can affect a single inner node, a leaf node, or a whole subtree; and
3. The *schema conformance*, which indicates, whether the path synopsis needs modification or not.

Figure 6.5 Update scenarios for the path-oriented document store



We regard these dimensions as not necessarily orthogonal. Figure 6.5 illustrates the meaningful combinations to be discussed in this section:

- Insertions of single nodes are always insertions of leaf nodes, because it is not allowed to insert a node between a parent and a child (this functionality is neither required in DOM nor in the declarative XQuery Update Facility, which is why we do not provide for it). Furthermore, because a single node can be seen as a subtree, only subtree granularity is discussed here. New subtrees might conform to the existing synopsis or they might not. As we will see, both cases can be treated quite similarly.

- Updates can effect inner nodes or leaf nodes. This is sufficient, because neither DOM nor XQuery Update define a "subtree update" functionality. All updates are translated to single-node modifications. Both granularities, however, can affect the path synopsis.

- Deletions of inner nodes always results in deletions of their complete subtree. Therefore, this is the granularity of a deletion. For deletions, a possible synopsis modification is not considered because, even if due to a deletion, a particular path class has no instances in the document anymore, it is meaningful to keep this class for possible later insertions of corresponding instances.

In the following, the actions to implement these different scenarios are described. In contrast to the previous sections where pseudocode was presented for that purpose, the description here is given in prose and is illustrated by examples, because code fragments tend to be too long for convenient comprehension. Again, only elements are discussed.

Insertion

Let n be the context node to which the new subtree shall be attached. There are two possibilities: 1) n is virtual, or 2) n is an empty element and thus non-virtual. In the second case, the non-virtual element has a corresponding record in the document store. This record has to be deleted because, when the insertion is complete, the empty element is *virtualized*. However, to avoid information loss, the deleted node is (conceptually) attached as a new root node to the subtree to be inserted. From this point on, both cases from above are handled equally.

For the actual subtree insertion, the document storage algorithm presented in Section 6.3 can be reused. For that purpose, the storage content handler (from Listing 6.1 on Page 171) is initialized with information from context node n, i. e., the `divisions` array is initialized with the divisions from the context ID, the `level` is initialized with n's level, the `dummyRecord` remains undefined, and the current `psNode` node is loaded from the synopsis. Instead of the `bsTreeBuilder`, a reference to the B*-tree itself is now required, to directly insert the newly generated records. During the insertion algorithm, the path synopsis is automatically maintained, when a new path synopsis node is created.

The costs for an insertion consist of the cost to locate the insertion position (on B*-tree traversal), optionally the cost to delete a record, and the cost to insert the new subtree, which depends on the subtree size. Compared to subtree insertion in the node-oriented approach, the possibly required deletion operation and the number and size of the new records account for differing costs. In case, the subtree to be inserted is small, e. g., only one node as in node-by-node DOM-style insertions, the

deletion cost can actually decrease the performance. However, in case of larger subtrees, to be expected for example in XQuery Update statements, the fewer number of records to be written in path-oriented storage can amortize the required deletion costs.

Deletion

Intuitively, subtree deletion in the path-oriented storage mode works in the same manner as in the node-oriented case: To delete the subtree rooted at context node n, all records having n's DeweyID as a prefix have to be removed from the document. However, there is a small pitfall: Assume, n has no sibling nodes, then (at least) the parent node p of n is also removed from the document store, because 1) p is virtual, and 2) only the records corresponding to the leaf nodes in the subtree below n "store" the information that p exists (but these records are to be deleted). Therefore, first a sibling has to be located and, if none exists, a record for p has to be prepared. Then, after all records have been deleted, the new record for the parent is inserted.

The deletion cost consists of the cost to do the sibling check, the cost to actually delete the records, and the insertion cost, if no siblings were found. For the comparison with the node-oriented approach, again the trade-off depends on the subtree size. If it is small, the deletion costs are higher, if it is large, the costs are amortized by the smaller number and size of records to be deleted.

Update

Updating leaf nodes conforming to the synopsis is straightforward: the node's record is located in the document store and the value is changed. Under this type of update, also value changes to attributes and text content are subsumed. The cost for such operations consists of a B*-tree traversal to locate the node to be updated and of the cost to actually change the value. Thus, these costs are the same as in the node-oriented case. When the leaf update affects the path synopsis, a new synopsis node with a new PCR has to be created and assigned to the updated XML node. All other actions and costs remain the same.

The update to an inner (virtual) node (i. e., renaming an element or attribute) is the only operation conceptually more expensive in the path-oriented case than in the node-oriented case. Because a virtual node is not stored explicitly, it "exists" implicitly in the PCRs of the records that correspond to its descendant leaf nodes. Therefore, when an inner node is modified, all the PCRs of these "descendant records" have to be updated, because they now belong to different path classes. Furthermore, if the update does not conform to the path synopsis, the complete synopsis subtree below the changed subtree node needs to be copied and new PCRs have to be generated.

As an example, consider a modification on the last *track* element with ID 1.3.11.19 in the document presented in Figures 6.3 on Page 169 and the store in Figure 6.4 on Page 170. Let the new name be *hiddenTrack*. Because the new name has no corresponding class in the path synopsis (in Figure 6.2 on Page 169), the subtree below the synopsis node with name *track* (PCR 31) needs to be copied, having a new synopsis node *hiddenTrack* as root node. This new subtree is now added to the *cd* node (PCR 2) as a child. Then, all records in the document store having 1.3.11.19 as prefix have to be updated with the newly generated PCRs.

Of course, this operation is much more expensive on the node-oriented store than on the path-oriented one, and no amortization can be expected this time. The update costs consist of the cost to locate the node to be updated, the cost to modify the path synopsis, and the cost to adjust the PCRs in the subtree below the updated node. In contrast, in the node-oriented case, the costs only consist of a location and an the subsequent update of a single record.

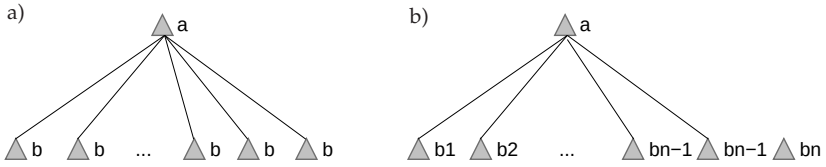
6.3.6 Round-Trip Property and Collection Support

Because the inner structure of an XML document can be reconstructed without loss (as shown above), path-oriented document storage supports the same type of round-trip property as node-oriented storage. This means that the data model (XML Infoset) of a stored document can be reconstructed. However, as before, storing byte-wise equal documents is not possible, because of intra-markup white-space. Note, as in the node-oriented implementation, only element, attribute, and text nodes can currently be stored in XTC.

Collection handling can be implemented similar to the node-oriented approach, i. e., all documents of the same collection share the same document storage and indexing space by adding all documents under a virtual root node. In many XML applications operating on collections, all collection documents have a very similar structure (for example, because they all adhere to the same schema). However, some XML applications may store differently shaped documents in a single collection. In the following, we refer to documents that have the same shape, as *document classes* (note, what is meant by "the same shape" will be clarified below). Whenever a collection contains multiple document classes, we can anticipate that the documents of these classes are often processed in one context (or user request). For example, consider a *music* collection containing document-centric *articles* about music records as the first class and—similar to our running example—a data-centric *description* of music records as the second class. If the user is only interested in articles, then the first class would be queried. In this case, a *physical clustering* of all articles, i. e., of the first class, would be beneficial. Such a physical clustering is simply implemented by creating a distinct document index (and distinct secondary indexes) for each class. Note, if we assume that document classes are sufficiently large, this clustering strategy only requires a small overhead to operate on multiple classes in one user request (namely to open different indexes).

To identify different classes in a collection (to find documents with "the same shape"), the path synopsis as a kind of ad-hoc document schema can be exploited. Identifying similar tree structures is a standard research problem and many publications exist (e. g., [Helmer 07]). For our needs, a rather simple comparison scheme can be used: We compare the path synopses of the documents to be stored on the first k levels, where k is a user-defined storage parameter. If the path synopses of two documents on the first k levels are equal (except permutation), the two documents belong to the same class and are stored together in one document index. Such a simple scheme is sufficient, because it 1) needs to be fast, i. e., more time-consuming complex comparisons on the whole tree structure unnecessarily slow down the storage process, and 2) a certain fuzziness in document classification can be tolerated.

Figure 6.6 Best-case and worst-case space reduction scenario for path-oriented documents



6.3.7 Succinctness

A main motivation behind the design of the path-oriented document store is succinctness, i. e., the reduction of storage cost by virtualizing the inner document structure. The "amount" of reduction depends on the structural complexity of the document, which we measured in Section 6.2.5 as the ratio between distinct root-to-leaf (DRTL) paths and root-to-leaf path (RTL) instances. Figure 6.6 shows the best-case (a) and the worst-case (b) scenarios for this ratio. In the best case, there is only one distinct root-to-leaf path and n path instances, i. e., the ratio is $1/n$. On the other hand, in the worst-case scenario, the number of distinct root-to-leaf paths is equal to the number of path instances, resulting in a ratio of 1. Therefore, in the best case, the path synopsis contains only one path, whereas, in the worst case, the path synopsis is as large as the original document.

Of course, reality lies between these two extremes. For many data-oriented XML documents, such as the ones presented in Table 6.1 (with the exception of *treebank*, which is document-oriented), the DRTL-RTL ratio is a very small number. For them, structure virtualization is quite efficient and also results in reduced processing time, because more data fits into the leaf pages of the document store which therefore leads to less read/write overhead and a higher number of buffer hits. We will underlay these statements with experiments in Chapter 9.

Concepts like prefix compression, Huffman-based DeweyID encoding, the use of a vocabulary (vocID) to represent element and attribute names, and text compression, as already sketched in Section 6.2.5, are orthogonal to both storage approaches. However, we have to state that the overall compression ratio to be expected from prefix compression is smaller in the path-oriented store than in the document-oriented store. The relationale is that in the path-oriented case, the DeweyIDs in the B*-tree leaf pages are not so dense as in the node-oriented case. Therefore, the delta information that has to be stored requires a little bit more storage space for each node. However, in our experiments, this effect did never overshadow storage savings that resulted from smaller absolute number stored nodes.

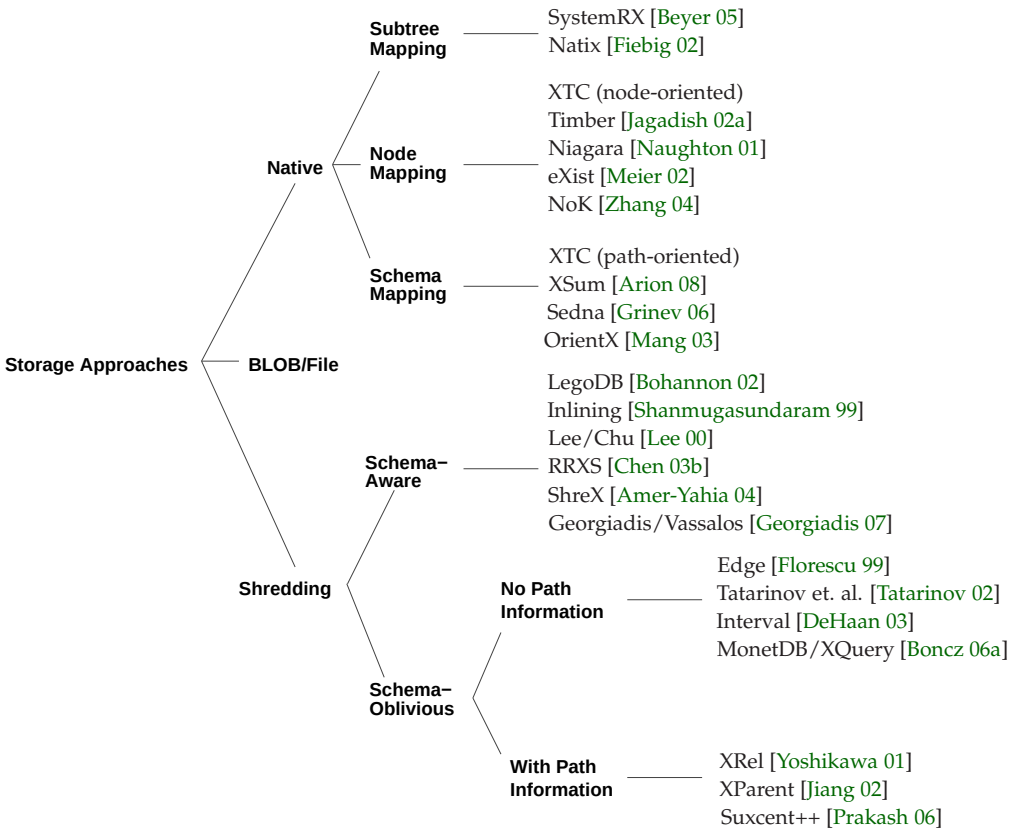
6.3.8 Indexing Support

Again, we want to consider path indexes and the question of how they can be efficiently built and maintained. As sketched in Section 6.2.6, the definition of a path index contains a path pattern to specify which paths of the documents are contained in the index. The pattern is evaluated against the document, either by direct evaluation or by a document scan (with path reconstruction) and an interleaved path pattern matching on the reconstructed path. Furthermore, when the document is modified, the ancestor path of the modified node/subtree has to be reconstructed

and matched against all index definitions to retrieve the indexes to which the modification needs to be propagated.

For both problems, the path-oriented document store provides simple and efficient solutions, which shall only be sketched in this section (the next chapter will provide a detailed discussion): The path-oriented document store already "knows" paths and furthermore shares the path synopsis with the path indexes. This means that a PCR assigned to a record in the document store can also be used in a path index to identify a path class. As a result, index creation can be implemented as follows: The path pattern of the index definition is evaluated over the path synopsis, resulting in a list P of PCRs. Then the document is scanned, and every node with a PCR contained in P is written into the index. Path reconstruction and repetitive path matching as necessary in the node-oriented store are therefore avoided. Likewise, index maintenance is also very simple: To detect, which index requires maintenance, the PCR of a modified node/subtree can directly be compared to the document's index definitions in the metadata. No path reconstruction (and as we will see, no path matching) is necessary.

Figure 6.7 A classification of related work on XML storage



6.4 Related Work

Figure 6.7 classifies recent XML storage techniques. At the first level, we can distinguish three basic approaches: storage in a binary large object (BLOB) or a flat file in the file system, storage in a specially designed XML store (native), or a mapping onto relational tables (shredding). Of course, the first alternative is not very appealing because, on every document access, the complete document has to be parsed and eventually loaded into main memory for processing. Furthermore, in case of document modifications, the complete document has to be written back to disk, too. Although BLOB or flat file storage was one of the first approaches to support XML documents in database systems, we do not consider this alternative as a true competitor to our storage schemes.

The proposed node-oriented and path-oriented storage layouts were developed in the context of a native XML database system. However, because some properties are similar, shredding approaches shall also be discussed in this related work section. In the following, one representative of each class introduced in Figure 6.7 will be presented. Among them, we discuss two of the five systems introduced in the related work section of our overview chapter, namely DB2 pureXML and MonetDB/Query. For Natix, Timber, and Galax will not be further discussed (we just categorize them).

6.4.1 Native XML Storage

As in relational systems, native XDBMSs have to map data onto fix-sized external memory pages. Depending on the granule of XML items that are actually mapped onto a page, we can distinguish *subtree mappings* from *node mappings*. The first strategy partitions an XML tree into different regions (subtrees), which are then mapped onto external memory. In the second strategy, the mapping granule corresponds to a node. Several nodes are written in document order into a page (as in the node-oriented approach). This means that subtrees are not clustered together on pages in node mapping schemes. A third solution exploits the ad-hoc structure (in form of a path synopsis) or schema to store the document (similar to the path-oriented approach). In the following, we will discuss the storage layout of various native XDBMSs.

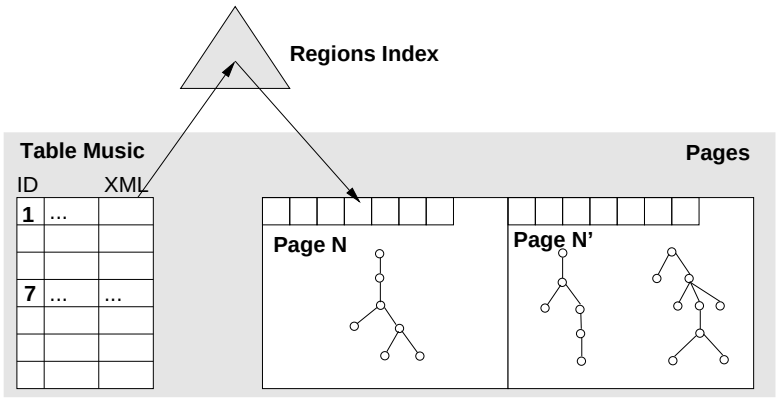
Subtree Mapping: DB2 pureXML

IBM's DB2 pureXML [Beyer 06] (and its predecessor SystemRX [Beyer 05]) are hybrid database systems that can store and process (i. e., query) both, relational and XML data at the same time. For XML storage, DB2 follows a side-by-side architecture [Halverson 04], in which the XML data is placed into a separate native XML store (i. e., no shredding is applied). The developers expect native storage to outperform shredding. An empirical evaluation however is still an open problem. Unfortunately, only high-level literature on the DB2 pureXML system is available, making the following discussion on physical storage layout more or less speculative. However, because DB2 gained much attention, we think a consideration is worthwhile.

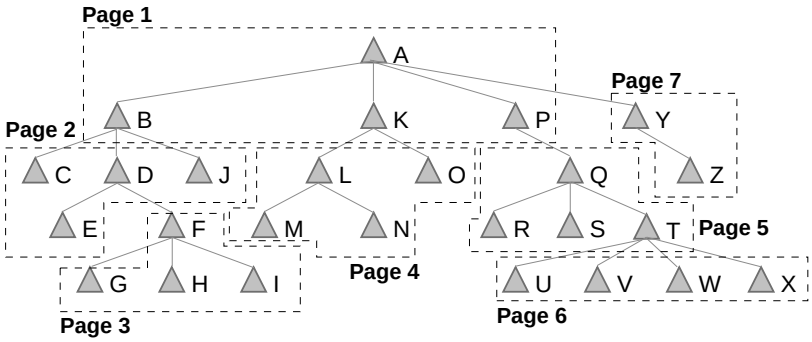
At the logical level, the system assumes that XML documents are stored in columns of relational tables. At the physical level, these documents are mapped onto external memory pages. Figure 6.8a gives a high-level impression over this mapping.

Figure 6.8 DB2 pureXML storage overview

a) Layout (from [Beyer 05])



b) Paged sample tree (subtree clustering)



If possible, a complete document is encoded as a binary representation of a type-annotated tree and stored in a single page. Of course, multiple small documents can also be stored within in a single page. However, if a page is not sufficient, a document is grouped into subtrees which are then stored in multiple pages. The literature does not contain any information about how these subtrees are identified. However, because the authors of [Beyer 05] state that parent child relationships are stored in page slots, we can assume that the storage mapping tries to cluster sibling nodes onto the same page, resulting in a kind of *breadth first* storage. A sample paged tree is illustrated in Figure 6.8b. We assume that a page can hold four nodes. Different subtrees of a single document are "glued together" by the *regions index*. Therefore, when a tree operation navigates over a subtree boundary, the regions index is queried to find the next adjacent subtree. As in our approaches, element and attribute names are compressed using a vocabulary and, to logically and physically reference a node, a suitable node identification mechanism is used (but not published). Furthermore, the authors state that the store supports versioning, i. e., upon document modification, a new version of the updated subtree is stored and presented to the user.

Evaluation: Because detailed information is not published, a fair and objective comparison to our storage proposal is not possible for DB2 pureXML. However, if we assume a mapping as depicted in Figure 6.8b, we can nevertheless draw some conclusions: 1) we can state that pureXML does not provide any technique to virtualize the inner structure of a document and, with the given mapping, virtualization

would also not be possible; 2) we note that the store of pureXML is not aware of paths, thus complicating index construction and maintenance; 3) document storage is more complex than in our approaches, because pages need to be visited/buffered multiple times. As an example for the last point, consider the reconstruction of the document in Figure 6.8b: After Page 1 is opened and nodes A and B are serialized, the page should then remain in the buffer because, after reconstruction the subtree below B, it is referenced again. If the subtree is large, it is likely that the page was already replaced and needs to be refetched again.

An advantage of this storage scheme is that navigational operations along the five primitives *parent*, *first-child*, *last-child*, *previous-sibling*, and *following-sibling* are efficiently supported. Subtree clustering makes sure that related nodes are often placed in the same page, thus minimizing I/O cost for these operations. The rationale behind subtree clustering is to process queries using the above navigational primitives directly on the document store. A similar subtree mapping approach with the same restrictions is implemented in the native Natix XDBMS [Fiebig 02].

Node Mapping: The NoK Approach

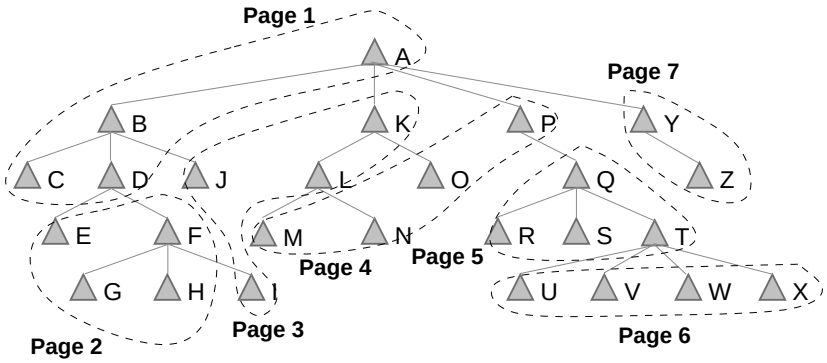
Storage engines based on node mapping store every node in document order on external memory. Because the mapping granule is a node, no subtree clustering is achieved. Rather, neighboring nodes in document order are also written together on disk. Of course, the node-oriented approach proposed in this work is also a node mapping scheme. Further implementations can be found in the following native XDBMSs: Timber [Jagadish 02a], Niagara [Naughton 01], and eXist [Meier 02].

As another interesting representative for node mapping, we consider the NoK ("Next-of-Kin") approach [Zhang 04]. NoK separates the document structure from the content and stores both parts separately. The structure is encoded into a string representation, using the well-known braces notation for trees. For example, the tree in Figure 6.8b can be represented by the string: `(A(B(C)(D(E)(F(G)(H)(I)))(J))...(Z))` or, because the opening braces are dispensable: `ABC)DE)FG)H)I)))J))...Z))`. This string representation is then distributed across several pages in a linked list (if it does not fit into a single page). Note, no labeling mechanism to reference single nodes is implemented in this storage mapping. The omission of a node ID mechanism results in a very succinct structure representation. For example, for the 89 MB Treebank document (Table 6.1 on Page 164), only 5.3 MB are required to encode the structure [Zhang 04].

After the separation of structure and content, a mechanism is required to connect both parts for lossless document reconstruction. Here, the scheme relies on DeweyIDs, which are constructed on the fly, when the document is stored, scanned, or reconstructed. If the storage SAX parser encounters a text node, a record (DeweyID, nodeValue) is written to a value index. If a document is reconstructed, the system generates DeweyIDs and queries the text index for corresponding values. Another index structure inverts text values and stores records of the from (HashedText, DeweyID) for index-based content matching.

The advantage of this approach is its structural succinctness, which makes scan-based query matching over the structural part efficient. However, although [Zhang 04] state that updates are supported, they are likely to be not very efficient, because the assigned DeweyIDs in the value index do not remain stable and have to be adjusted for large parts of the document after every modification. Furthermore,

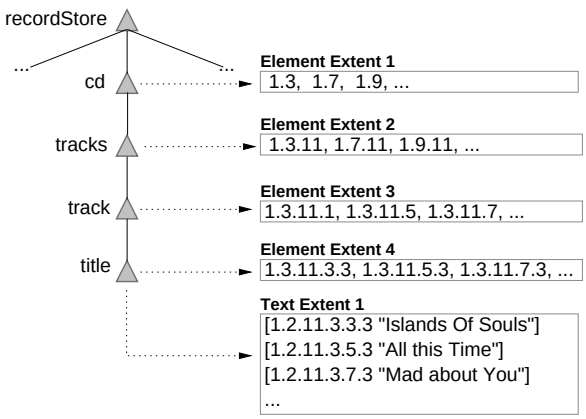
Figure 6.9 The node mapping approach



because content and structure are separated, they need to be joined for document reconstruction, thus requiring an index lookup for every generated DeweyID (or at least some kind of merging algorithm that operates on the structure and content pages). This process can be very time consuming, as similar experiments in Section 9.3.3 will show.

In general, node-mapping approaches, as the one introduced in this work, do not suffer these penalties, because they employ a node identification mechanism and do not separate structure from content. Furthermore, in contrast to subtree mapping, they do not cluster siblings, but neighboring nodes (in document order). Figure 6.9 illustrates a paged sample tree. As an effect, sometimes nodes are stored in the same page that are not anyhow related by the above stated five navigational primitives (for example, in page 3). However, the nodes in a page can also form a subtree (as in page 5). A general comparison between subtree mapping and node mapping is still missing. In the latter case, we suppose that with a large number of nodes stored in a single page, the probability that a subtree is formed is very high. Therefore, access cost for page boundary traversal will be statistically equal to subtree clustering for many operations. The only significant difference will then result from the intra-page layout and the efficiency to operate on the page content.

Figure 6.10 The XSum schema-based mapping [Arion 08]



Schema-Based Mapping: XSum

As our path-oriented approach, XSum [Arion 08] employs a structural summary to store a document. Such a structural summary can be seen as a kind of ad-hoc schema, therefore the name "schema-based mapping". In contrast to separating the document store from secondary path indexes—as in this work—, XSum integrates the document store *into* the path index. A snippet of the resulting structure for our sample document is depicted in Figure 6.10, where the structural summary is on the left side. As in our case, it is a strong DataGuide or a 1-Index. Attached to each node, there are one or more extent lists, depicted on the right side. XSum separates extents containing structural information from extents containing content (text) information. In [Arion 08], the authors use a range-based node labeling scheme, but explicitly mention that any other scheme (which encodes certain structural relationships) can also be used. In our sample, we, therefore, depicted the storage scheme based on DeweyIDs.

The advantage of this mapping is the integration of the document store into a path index. For example, to retrieve all elements on path `//cd/tracks/track`, only the access to one extent is necessary. On the down side, XSum violates several of our desired document store features: Although the concepts in the XSum approach allow for DeweyIDs, they do not make use of them. From Figure 6.10 you can immediately infer that this naive application of DeweyIDs introduces a lot of the redundancy (e. g., the IDs of the *cd* elements is also contained as a prefixes in the extent list of the *tracks* element). Therefore, the store is not really succinct. Note, because our path-oriented store virtualizes inner elements, such as *cd*, *tracks*, ..., this type of redundancy is avoided. A much worse problem is that the document is "shredded" over the extents. Although the document order is maintained *inside* the extents, a merge join is necessary to combine two or more of them, as frequently required during result construction. This process is very expensive and violates our requirement for fast reconstruction and scan operations. Furthermore, no algorithms for document navigation and modifications have been published for the approach. Similar schema-based approaches have been published in the context of the Sedna [Grinev 06] and OrientX [Mang 03] native XDBMSs.

6.4.2 Shredding

With the emerging need to store large volumes of XML in database systems, a quite simple idea was to distribute (*shred*) XML documents over various relational tables which are then stored in well-known, well-tested, and well-accepted RDBMSs. As a consequence, also XML operations, such as declarative queries, have to be mapped onto SQL statements to be executed over the shredded document. While some research systems were quite successful with the shredding technique (e. g., [Boncz 06a]), large database system vendors nowadays more and more prefer native XML storage [Beyer 06]. What, in the end, will be the better approach, is still an open question. Two of our most substantial doubts on shredding are the following: Because relational systems are generally oblivious of the hierarchical XML structure, they cannot provide efficient operators to evaluate XML path queries and they do not provide any optimized mechanisms for fine-grained transaction control (as in [Haustein 06a]). Nevertheless, we want to give an overview over the numerous shredding proposals.

Figure 6.7 groups the approaches into 1. schema-oblivious shredding, which provides a single generic relational schema into which all XML documents are stored, and 2. into schema-aware shredding, where a distinct set of relational tables is generated for each document (based on a DTD or an XML Schema). The schema-aware approaches can further be refined into non-cost-based (e. g., [Shanmugasundaram 99]) and cost-based strategies (e. g., [Bohannon 02]), where the latter ones take the expected query workload on the stored data into account. In the following, we do not consider schema-aware approaches, because they cannot truly support dynamic documents that have a dynamic (i. e., changing) schema or no schema at all and, thus, violate the key requirement *update support*. Except the both already cited approaches, further schema-aware shredding strategies were developed by [Lee 00, Chen 03b, Amer-Yahia 04, Georgiadis 07].

On the side of schema-oblivious shredding techniques, we distinguish those that store only node/edge information and those that (also) store paths. [Florescu 99] proposed one of the first schema-oblivious node mapping approaches, in which each edge of a document is stored in a row of a relational table. An edge is encoded in the so-called *edge table* by the following information: [*source*, *ordinal*, *name*, *flag*, *target*], where *source* is an integer object identifier (node label), *ordinal* encodes the order among edges, *name* contains the node or attribute name, *flag* is a descriptor indicating the type of the referenced node's content (i. e., int, string, ref, ...), and *target* is a foreign key to a table, where all content values are stored (i. e., content and structure information is separated).

Considering this mapping, two substantial problems can be identified that influenced subsequent shredding approaches: 1. the relational query engine is not and cannot be aware of the hierarchical data model. Therefore, to evaluate a path expression over the above relational schema, self joins over the edge table are required; and 2. to re-construct a document (or a result), structure information and value information need to be joined. Both problems severely affect query performance. Of course, since this first proposal, many research projects tackled these two problems, mainly by encoding path information in the relational model and by embedding specialized algorithms (e. g., [Grust 03b]) for path matching and result construction (e. g., [Chebotko 07]) in the relational algebra. In the following, we will introduce the storage layout of MonetDB/XQuery and the Suxcent++ approaches as representatives for the two schema-oblivious shredding classes.

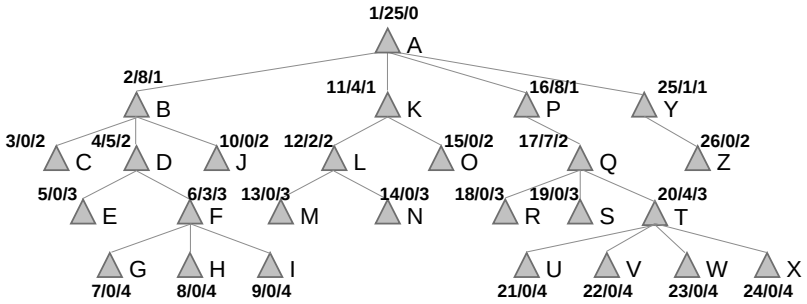
Mapping Without Path Information: MonetDB/XQuery

As we have seen, MonetDB/XQuery [Boncz 06a] consists of two components: the relational MonetDB system and, on top, the Pathfinder XQuery engine [Boncz 05b]. Compared to commercial DBMS, the design of MonetDB has some particularities:

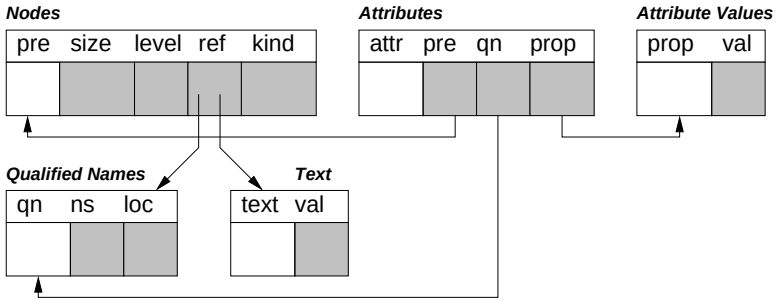
- Physically, relational tables are stored in so-called *binary association tables* (BATs), i. e., a logical relational model is completely decomposed into relations of cardinality 2 for storage. One column of each binary table carries an object identifier (*oid*) of type integer. Actually, *oid*s are not physically stored. They are assigned in ascending order with an increment of 1, therefore, they can be dynamically recalculated from the array position. BATs keep the data in the same order as in the logical relational representation. Therefore, two BATs can be efficiently joined, solely based on the position of their elements. Basically, a BAT is nothing else than a (long) array.

Figure 6.11 MonetDB/XQuery document mapping overview

a) (pre/size/level) Tree Encoding



b) Shredding Schema (primary keys indicated by white box; foreign keys indicated by arrows)



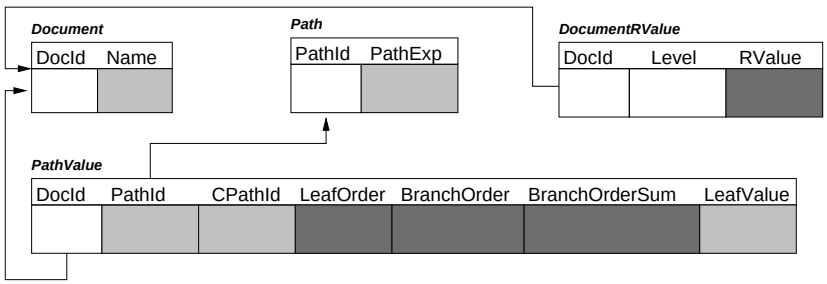
- MonetDB reuses the operating system's virtual memory management as "database buffer". Data is solely addressed in main memory and the operating system takes care of swapping in and swapping out the requested data from/to external storage. As a consequence, relational operators are optimized for main memory usage.

To store a document, Pathfinder first applies a *pre/size/level* labeling scheme on the document (see Figure 6.11a). The *pre* number is an integer assigned to each node in preorder traversal. Attribute *size* contains information about how many nodes are located below each node, and the *level* attribute, designates the node level. Similar to range-based encodings or DeweyIDs, *pre/size/level* numbering can decide all XPath axes and serves as a foundation for the implementation of the staircase join operator [Grust 03b]. For storage, every node in the document is mapped into a relation of the schema presented in Figure 6.11b (physically, this schema is of course further normalized into BATs, however, it would also be possible to use this schema in any other relational system).

In the MonetDB/XQuery shredding schema, all nodes are stored in the *Nodes* table. Besides the introduced *pre*, *size*, and *level* information, this table also has a *kind* column which designates the type of the encoded node and a *ref* column that points to a table containing the node value. Thus, the *kind* column acts as a switch, to select the correct value table (i. e., a qualified name in case of an element, a text value for text nodes, ...).

The MonetDB/XQuery schema is quite similar to the above introduced edge table approach. The only significant difference is the node labeling scheme, which extends the primitive *source* labels with a *size* and *level* attribute. Still, to reconstruct

Figure 6.12 The Sucxent++ storage scheme



a document or a query result, various tables have to be joined. Joins, however, are specially optimized operators in MonetDB (because BATs frequently need to be joined). As a result, although joins are necessary, MonetDB/XQuery delivers acceptable performance for document reconstruction. This may, however, not be true, if the same shredding scheme would be implemented on other relational systems. Although the authors present some work on how XPath axis steps can be evaluated on off-the-shelf relational DBMS [Grust 07], they did not provide any figures on the performance of document reconstruction. A detailed investigation is still an open issue. Updateability poses another problem. Obviously, when a subtree is inserted or deleted, the size information of all ancestor nodes of the subtree have to be re-assigned. Furthermore, because no gaps are left in the *pre* column, all following nodes have to be updated when a new subtree is inserted. As a solution, the authors presented a modified (updateable) shredding schema [Boncz 05a], which rests on the salient features of the MonetDB system and is probably hard to port to off-the-shelf systems. Again, a detailed investigation is an open issue. For path evaluation, MonetDB/XQuery employs the staircase join operator which operates in a scan-and-skip fashion, i. e., the operator scans the nodes table and—based on the pre/size/level information—tries to skip as large parts as possible. Because MonetDB is a main memory database, secondary access structures, such as a path index, are not considered to speed up this path matching process. Therefore, no path information is contained in the storage layout, in contrast to the approach presented in the next section. Further schema-oblivious mapping schemes that do not store path information can be found in [DeHaan 03, Tatarinov 02].

Mapping With Path Information: Sucxent++

Although Sucxent++ (for **S**chema **U**nc**o**nscious **X**ML **En**abled Sys**t**em) [Prakash 06] is a shredding technique, we will see that it is quite close to (but actually not the same as) our path-oriented document store. To overcome that problem of expensive path reconstruction over shredded XML documents, Sucxent++ also stores path information (a technique introduced before in [Yoshikawa 01]). Figure 6.12 illustrates the relational schema into which documents are shredded. As before, white boxes indicate primary keys and arrows stand for foreign keys. The darker shaded boxes have a special meaning which will be discussed below.

Because all documents are stored into the depicted sets of tables, a document identification mechanism is required. This is the task of the *Document* table, which gives each document a unique ID. The *Path* table conceptually resembles our path synop-

sis, because it stores all paths and gives them a unique identifier. In the *PathValue* table, all leaf nodes are stored together with their pathID. To reconstruct a document, it is necessary to keep the leaf nodes in the correct order. Therefore, the *LeafOrder* column simply enumerates them consecutively. Further information required for lossless reconstruction is the information on which level the paths of two consecutive leaf values intersect. This information is stored in the *BranchOrder* column. As an example, consider the document in Figure 6.11a. Nodes *E* and *G* would be stored side by side (i. e., their leaf order values differ by 1). Their branch order would be 2, because their paths intersect at the *D* node.

Besides these tables and columns, Sucxent++ additionally introduces the *DocumentRValue* table and the *BranchOrderSum* column. Without delving into the details (which are quite complex), we simply state that this additional information is required to calculate the least common ancestor between any two leaf nodes in the document (note, the *BranchOrder* column can only deliver the ancestor of neighboring nodes). In summary, the additional table/colum stores some kind of document "geometry".

To the best of our knowledge, Suxcent++ and our independently developed path-oriented document store (formerly named "elementless document store" [Härder 07]) are the only XML storage approaches with structure virtualization. However, it is obvious that the Sucxent++ approach was developed without taking document modifications into account. Upon modification, not only new tuples are inserted into the *PathValue* table, but also large fractions of the darker shaded columns (in Figure 6.12) need maintenance. Thus, Suxcent++ violates *updates* key requirement. Further related work on shredding approaches with explicit path storage can be found in [Yoshikawa 01, Jiang 02].

6.5 Summary

The document store is a central component in any XDBMS. Because XML is a very flexible data format with a large range of possible applications, documents are processed using quite different access models, i. e., scans, navigations and declarative queries. Furthermore, documents themselves can occur in many different shapes and sizes. This chapter started with a list of important characteristics, the document store of a modern XDBMS should provide. During the design of XTC's storage scheme, special attention was paid to fulfill all these desiderata without biasing towards only one or two of the posed requirements. The resulting document container can store documents in a node-oriented manner, which is preferable for document-centric XML data, or in a path-oriented manner with structure virtualization, preferable for data-centric XML. Currently, the user has to fix the storage scheme when the document is imported. In the future, however, the XTC system should internally decide the storage mode, based on some pre-storage analysis phase that samples parts of the document.

Chapter 7 XML Indexing

> I'm so fast that last night I turned off the light switch in my hotel room and was in bed before the room was dark.
>
> Muhammad Ali

Because the XML data model is essentially a tree, path patterns are quite natural and common idiom in XML query languages. Thus, finding path patterns, also known as *path pattern matching*, is a frequent operation in XML query processing. Furthermore in a query evaluation plan, operators that find path patterns are often located at the bottom, i. e., they fetch data from external memory on which subsequent processing algorithms operate. In Chapters 4 and 5, we have seen many examples of this structure. Because external memory access is involved in path pattern matching, they are not only a frequent operation, but also possibly expensive. Of course, path pattern matching can be implemented directly on the document store by relying on the navigational and scan primitives available. However, often navigational and scan performance are not sufficient enough.

A similar situation occurs in relational systems. A B*-tree-indexed base table provides for table scans or ID-based access operations. To speed-up certain queries, secondary access paths (indexes) are specified by the administrator in the physical database layout. In XML data management, we can proceed similarly: The document store provides enough functionality to efficiently store, reconstruct, modify, and navigate a document. It, therefore, can be seen as a plain XML container. However, when query processing over the document store (based on its primitive operations) does not deliver enough performance, the database administrator can decide to create secondary index structures by anticipating the expected query workload. As introduced, declarative XML languages often contain path expressions. Therefore, the ideal candidates for secondary indexes would be path indexes.

Of course, because XML indexing can be considered as a standard problem, many solutions have been presented in the past. Roughly, we can divide them into *value* and *text indexes*, *element/attribute indexes*, *path indexes*, *adaptive path indexes*, and *content-and-structure indexes*. This richness may lead to the conviction that indexing has thoroughly been studied in the literature (and, in fact, we can find many good proposals, some of which we will pick up in the following to design the XTC indexes). However, despite this richness, many proposals lack a "system context" in their solutions, i. e., they completely abstract from the integration of their ideas in a (native) XDBMS and only consider the problem of XML indexing alone. As in the previous chapter, an often forgotten aspect in these approaches is the need for

dynamic (i. e., updateable) documents and, therefore, index maintenance.

To overcome these problems, this chapter takes a global, system-centric view on XML indexing. It illuminates interdependencies and optimization opportunities between document storage, XML indexing, and query processing. As shown in the previous chapter, (path-oriented) document storage could heavily benefit from ideas proposed by the indexing community, e. g., the path synopsis and the B*-tree. Here, we will now see that, because of this kind of proximity, problems like index creation and index maintenance can be implemented in a very simple and efficient way on top of the document store. Furthermore, we will consider how the results of index access operations can be utilized in query processing. The result will be an *integrated* approach to XML indexing *and* storage.

As in the storage chapter, we will start by defining a list of desiderata expressing the requirements kept in mind during the index design. Then, XTC's available index structures will be assessed and, based on our considerations, the new indexing scheme for path and content-and-structure (CAS) queries will be introduced.

7.1 Desiderata

As in the previous chapter, our list of desiderata implies that we do not consider it as normative, but suggest that it is meaningful for many XML applications.

An XML indexing scheme should provide for the following six characteristics:

1. *Optional Use*: As in relational systems, indexes should be secondary access paths that are optional (i. e., not essentially required for document storage). This ensures that indexes can be created on demand to trade query performance with maintenance cost and space consumption.

2. *Expressiveness*: The indexing scheme should be able to answer path queries supporting the child (/) and descendant (//) axes, name tests, wildcards (*), as well as one optional content predicate, e. g., `//record[price="12.99"]`. Queries without content predicates will be called *simple paths* in the following, whereas queries with a content predicate, will be called *content-and-structure* (CAS) *queries*. Both types frequently occur in XQuery expressions.

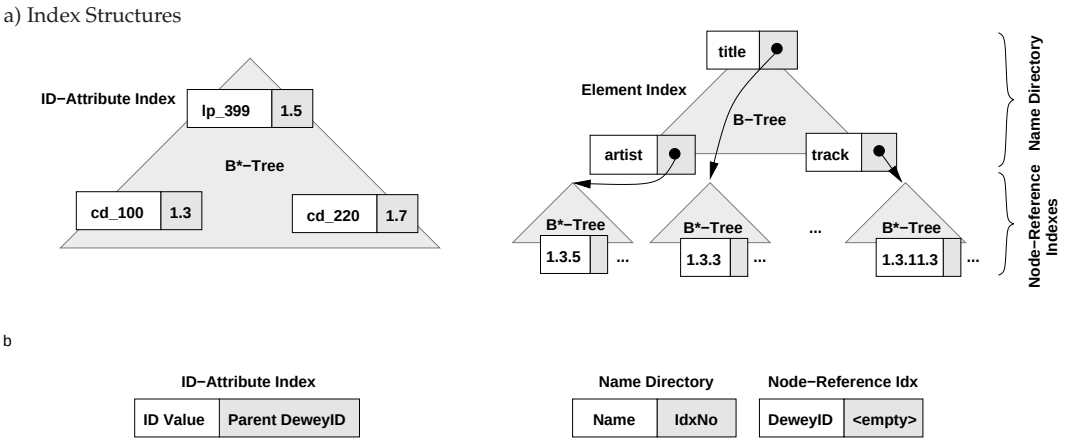
3. *Selectivity*: The selectivity of an index, i. e., which paths are actually contained in an index, should be user-defined. Thereby, a set of indexes can be adjusted to document characteristics and query workload.

4. *Updates*: The index should be updateable. Depending on the index selectivity, not all document updates lead to index updates. However, there should exist efficient mechanisms to discovery, when an index needs maintenance.

5. *Applicability*: The test whether an index can be applied for query evaluation should be simple and cost-efficient.

6. *Result Computation*: The index should be able to retrieve all elements on an indexed path, e. g., if an index can answer the above query `//record[price="12.99"]`, then it should be able to return the matching *record* and *price* nodes. Otherwise, the applicability of the index would be too restricted, w. r. t. further processing algorithms.

Figure 7.1 The ID-attribute index and the element index on sample document *recordStore.xml*



7.2 XTC's Indexing Scheme Reconsidered

Initially, XTC provides two index structures [Haustein 06a]: the so-called *ID-attribute index* and the *element index* (besides the document store, which can also be seen as an index for DeweyID-based access). Figure 7.1a illustrates these two secondary access structures for our sample *recordStore.xml* document.

7.2.1 The ID-Attribute Index

The ID-attribute index is a B*-tree which maps the value of an ID attribute (as for example specified in an XML schema specification) to the element containing the ID attribute. Figure 7.1b presents the record format of this index structure. As a convention, a white box always represents the indexed key and a shaded box the indexed value. The purpose of this kind of index structure is to speed-up the *getElementByID* method, defined in the DOM standard [DOM 04] or to support ID-based XML queries. It can, furthermore, be used to maintain the required uniqueness among all ID attributes in a document.

7.2.2 The Element Index

The element index maintains a posting list of all occurrences for a given element name, where all element names are contained in a B-tree called *name directory*. The record format of the name directory therefore contains the indexed element name as key and a pointer to its posting list as value. The posting lists, in turn, maintain element occurrences by storing DeweyIDs (i. e., by node reference). As depicted, the lists are themselves indexed using B*-trees and are named *node-reference indexes*. The record format of the node-reference index only stores a DeweyID as a key; the value field is empty.

For query processing, the element index provides two basic access primitives:

- *Scan*: Because the DeweyIDs in a node-reference index are stored in document order in the leaf pages of the B*-tree, retrieving all occurrences (DeweyIDs) of

a certain element name in document order can be implemented by a sequential scan. For example, if the query processor needs to access all *artist* elements, the record containing *artist* as a key is retrieved from the name directory, and the corresponding node-reference index (identified by its index number) is scanned. Because this list of *artists* may become very large, the element index provides a similar (partitioning) method as for the document container (see Section 6.3.3 on Page 176): Access method *getScanPartition* can be parameterized with 1) DeweyIDs `root, start,` and `end`, denoting the range to scan in the DeweyID space; 2) a `resultSize` parameter to restrict the number of returned nodes; and 3) a `self` flag indicating whether the start node shall also be returned as an element in the partition. The element index can be opened at any DeweyID for scanning, thus also supporting a "scan-and-skip" access.

- *Axis Evaluation*: Given a context node n, the element index can be used to retrieve all nodes of a certain name having one of 11 XPath axis relations[1] with respect to n. Thus, given n as an XPath variable $\$n$, the element index can evaluate query `$n/<axis>::<nameTest>`. Axes *child*, *descendant*, *descendant-or-self*, *following-sibling*, *preceding-sibling*, *following*, and *preceding* can be evaluated by a suitable range scan. The remaining axes (*ancestor*, *ancestor-or-self*, *parent*, and *self*) are evaluated by single node lookups.

  Because the corresponding methods are more or less straightforward, they shall only be sketched with the help of two examples here. Consider the *track* element with DeweyID 1.3.11.7 in our sample *recordStore.xml* as context node n. Then the query `$n/following-sibling::track` can be evaluated with the help of the element index by: 1) opening the node-reference index for *track* at DeweyID 1.3.11.7, 2) scanning the index, until the first record is found, whose DeweyID is not a descendant of n's parent DeweyID (1.3.11), and 3) (on every returned record) testing whether it actually *is* a following sibling of 1.3.11.7. Points 1 and 2 delimit the range of the scan. Point 3 is necessary, because (in contrast to our sample document) *track* elements may appear as descendants of parent DeweyID 1.3.11 in the scanned range.

  Scan-based evaluation for the second group of axes mentioned above does not make any sense because of their selectivity (i. e., they typically would skip many nodes when implemented by a range scan). Therefore, they are implemented in a node-at-a-time fashion: Consider again our sample document and *track* element 1.3.11.7 as context node n. Then, query `$n/ancestor::cd` can be evaluated, by 1) calculating all ancestor DeweyIDs of 1.3.11.7, and 2) subsequent lookups in the node-reference index of *cd*.

You may have noticed that the *scan* access is only a special case of *axis evaluation* (or the other way around), i. e., a scan is an axis evaluation of the *descendant-or-self* axis on the root DeweyID (1). An axis evaluation could be implemented on top of the *getScanPartition* method. For the sake of simplicity, the above two methods were introduced separately, despite their dependencies.

Note, furthermore, that similar scan and axis evaluation primitives can also be implemented over the document store, i. e., the element index is optional w. r. t. to these operations. However, in general, element-index-based evaluation will perform significantly better, because only elements with the same name are read (and no further nodes, such as for example text nodes in the case of a document scan).

[1] The *namespace* and the *attribute* axis are not supported.

7.2.3 Assessment of XTC's Indexing Scheme

We now consider the introduced indexing scheme w. r. t. our list of desiderata:

- *Optional Use*: Both indexes are optional. In the case of an absent ID-attribute index, the evaluation of the *getElementByID* method would however be extremely expensive, because the complete document would have to be scanned.

- *Expressiveness*: Obviously, not even the combined use of the ID-attribute index with the element index can deliver the required expressiveness for query evaluation. The element index can deliver ordered element lists but, what is actually required are *path matches*. In Chapter 5, we have introduced the twig operator. This logical operator stands for a special pattern matching algorithm that will be introduced in Chapter 8. The algorithm can take ordered element lists (provided, for example, by the element index) as input and can return path pattern matches. However, because the element index does not contain any content information, we cannot evaluate content predicates this way. Therefore, only the structural part of a query like `//record[price="12.99"]` can be evaluated (by the twig operator), but the content part has to be checked against the document. Another problem is the evaluation of wildcards (*). A wildcard matches *any* name. Of course, it would not be viable to read *all* node-reference indexes. Therefore, other techniques to efficiently evaluate wildcards have to be found. Note, by adding the functionality of the ID-attribute index, CAS queries like `//record[@id="d_100"]` can also be evaluated.

- *Selectivity*: The selectivity of both indexes is "hard wired", i. e., no true adjustment to a query workload is possible. For the element index, however, this can be fixed by allowing the user to define the element names that should be contained in the index.

- *Updates*: The update scheme for both indexes is quite straightforward: When a subtree s is deleted from the document, every element and attribute node is visited in document order (using a sequential scan over the document store for the subtree to be deleted). When an ID attribute is encountered, it needs to be deleted from the ID-attribute index. In case of an element node, its name is collected in a *name list*. For each name in the name list, all descendants of the subtree s are removed from the corresponding node-reference index. Similar measures have to be taken, when a subtree is inserted/modified. The maintenance overhead consists of the cost for a subtree scan and the cost for the actual index updates.

- *Applicability*: The question of index applicability has a trivial answer, because the indexes are "hard wired", i. e., the ID-attribute index is applicable, if it exists. The same is true for the element index. If, however, the user has restricted the set of element names indexed (as sketched before), the element index is only applicable for a (sub-)query, when the necessary element names are contained.

- *Result Computation*: Because paths can only be matched algorithmically (and not directly over the indexed data), the question of how inner elements can be recomputed is a mute point. We will see in Chapter 8 how this is actually achieved.

Both index types are independent from the storage mode, i. e., they can be created over a document stored in node-oriented or in path-oriented mode. Furthermore, they also support collections.

In summary, even the combined use of an element and an ID-attribute index lacks expressiveness (no content queries) and user-defined selectivity. Intrinsically, the notion of *paths* is missing in these index structures. Paths have to be reconstructed or recomputed by complex path matching algorithms (see Chapter 8). However, for the following reason, we nevertheless consider the proposed indexes useful: Sometimes documents have a high structural complexity (for example the *treebank.xml* document) and, therefore, indexes encoding precomputed paths would inherit this structural complexity, rendering their management quite expensive. On the other hand, element and ID-attribute indexes do not inherit structural complexity, because they do not encode paths. Therefore, their management on structurally complex documents is cheap.

7.3 Path Indexing

As already motivated, path queries are essential ingredients in XML query languages and their evaluation is critical to query performance. To address this requirement, we propose a path indexing scheme. We want to highlight the salient features of this scheme already at this point: The path indexing scheme is solely based on the structures already proposed for document storage, namely DeweyIDs, the path synopsis, and the B*-tree. This infrastructure "reuse" allows code sharing and a tight coupling between storage and indexing concepts, resulting in an integrated mechanism. In this mechanism, issues like index construction and maintenance can be implemented in a very simple and efficient way, because both, the path-oriented document store and the index manager are aware of paths. Additionally, we have seen that DeweyIDs provide for fast ancestor ID recomputation and that DeweyIDs and PCRs form a kind of coordinate system in the document. This concept will be reused to compute inner elements from path matches, as desired in our indexing "wish list" in Section 7.1.

This section first gives a more formal definition of the query types considered for path indexing, thus, refining Requirement 2 in our list of desiderata. Then, so-called content and structure (CAS) indexes, which are a hybrid form of path and content indexes, are introduced and classified. After discussing how CAS indexes can be queried and maintained, the question of when such an index is applicable for query processing will be answered. Afterwards, we will generalize CAS indexes to content indexes and plain path indexes (to evaluate simple path queries without a content predicate).

7.3.1 Query Types Considered

Definition 2 introduces the concept of a simple XPath query (XPQ). Basically, an XPQ is a path query with at most one predicate, which, in turn, can be a simple path (as in `//cd[genre]`) or a content predicate. Content predicates can be specified as path predicates (as in `//cd[genre="Pop"]`) or as self-predicates, as in `//vinyl/genre[. = "Jazz"]`. Furthermore, content predicates may occur as range predicates (e. g., `//cd[10 < price < 20]`) or as point predicates (e. g., `//vinyl[price="12.99"]`).

In Definition 2, the semantics of an XPQ expression is only informally given by ref-

erencing the XPath semantics. This should however not pose any problems because of the simplicity of XPQ expressions. Nevertheless, some specialties shall be illustrated: The particle "`//`" abbreviates the *descendant* axis[2], while `@` abbreviates the attribute axis. Thus, XPQ $//@*$ refers to all attributes in the document. In contrast to XPath, range predicates are not expressed using the Boolean *and* conjunction, but directly in a mathematical range notation. The dot (.) can be used to refer the output of the rightmost name test in the preceding simple path expression, as in `//cd/genre[. = "Jazz"]`.

Definition 2 (Simple XPath Query (XPQ)) *The syntax of a simple XPath query Q is defined by the* `path` *grammar production below. The terminals* `<nameTest>` *and* `<value>` *are substituted by qualified names (QNames) and values. The semantics of a simple XPath query expression are derived from the XPath semantics [Berglund 04]. XPQs with a content predicate (i. e., with a predicate based on grammar productions* `pathPred` *or* `selfPred`*) are referred to as content-and-structure (CAS) queries. If a CAS query has an equality predicate (=), we name it point query; otherwise, we name it range query.*

```
path      ::= //relPath | /relPath
relPath   ::= simplePath | simplePath[simplePath] |
                        simplePath[pathPred] | simplePath[selfPred]
simplePath::= step | simplePath/step | simplePath//step
step      ::= <nametTest> | @<nameTest> | *
pathPred  ::= simplePath Cmp <value> | <value> RCmp simplePath RCmp <value>
selfPred  ::= . Cmp <value> | <value> RCmp . RCmp <value>
Cmp       ::= < | <= | = | != | >= | >
RCmp      ::= < | <=
```

In the following, we only consider CAS queries and suitable indexes for their evaluation. The necessary modifications on the introduced index structures to support other types of XPQs will be introduced afterwards.

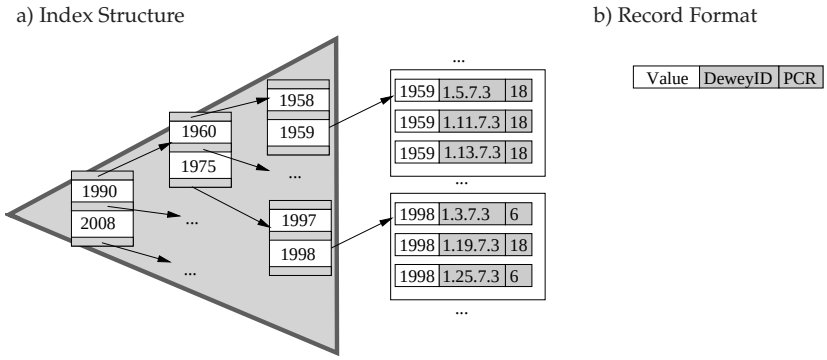
7.3.2 Defining CAS Indexes

To answer CAS queries, we provide a hybrid index structure that captures content and structure. For its definition, we reuse three basic concepts already introduced in the previous chapter, namely: DeweyIDs, the path synopsis (with PCRs), and the B*-tree. The resulting access structure is called *CAS index* and is specified in Definition 3. As a CAS index example, consider definition *I(//recordStore/*/year, Integer)*, for which the resulting access structure is schematically depicted in Figure 7.2. The index contains a record for every *year* element on the specified path[3]. At the level of B*-tree leaf pages, these records are ordered by the content of the *year* element and, inside each group of records with the same key, in ascending document order. Each record carries a PCR denoting the path on which the corresponding *year* element resides, i. e., 6 and 18 (see also Figure 6.2 on Page 169). As we will see, the index can be used to answer queries like `//year[. = 1998]`, `//cd[1960 < year < 1990]`, etc. Of course, the sample index structure is very selective, because only queries related to the *year* element can be answered. Further interesting indexes

[2]Note, here we use "`//`" as an abbreviation for the descendant axis neglecting the previously discussed correct semantics of this operator.

[3]Note, to make the example more illustrative, we added some records representing further *year* elements that are not explicitly shown in the *recordStore.xml* document printed in the appendix.

Figure 7.2 A Sample CAS index on *recordStore.xml* with definition $I(//recordStore/*/year, Integer)$



with a lower selectivity are, for example, the following: *I(//@id, String)*, which provides a similar functionality as the ID-attribute index, and *I(//*, String)*, which is basically an index over all content values in the document.

Definition 3 (Content-and-Structure (CAS) Index) *A CAS index on document D is denoted as $I_D(p, T)$, where the index path predicate p is a simple path query following the syntax of grammar production* `idxPath` *below (and non-terminal* `simplePath` *is borrowed from Definition 2), and T is the indexed content type, e. g., Integer, String, etc. Where non-ambiguous, we omit D and T. A CAS index is implemented using a B*-tree that can handle duplicate keys. For each leaf node n of D, a record is contained in the B*-tree of $I_D(p, T)$, iff C_1) the parent element of n is contained in the result of the evaluation of p against D and if C_2) n matches the content type of the index definition. An index record has the following form: $R = [\underline{C}, D, P]$, where C is the content of the indexed leaf node used as the record key, D is the DeweyID of n, and P its PCR. The keys in I_D are ordered in ascending order w. r. t. T, while the DeweyIDs (occurrences of n in D) are in document order for one and the same key value.*

```
idxPath   ::= //simplePath | /simplePath
```

Obviously, a CAS index is optional and its selectivity can be specified by a user-defined path pattern. Therefore, Requirements 1 and 3 are immediately fulfilled. Because CAS indexes are content-related, pure path queries cannot be answered by them. However, as already stated above, the CAS index concept will be generalized to fully support Requirement 2.

7.3.3 Creating CAS Indexes

An index $I_D(p, T)$ is created on a document stored in path-oriented mode as follows: First, index path p is evaluated[4] against document D's path synopsis, resulting in a list P of PCRs matching p. Then, D is scanned in document order. For each value v_i, we check whether its PCR is contained in P (C_1 in Definition 3) and whether its type matches T (C_2). If so, a record for v_i is inserted into a sort buffer. After the complete document is scanned, a stable sort on the index keys is executed and

[4]The evaluation of a structure query on a path synopsis is not formally defined here. The semantics should, however, be intuitively clear.

I is built bottom-up. The correctness of this creation process—i. e., every indexed content value is on path p and the ordering complies with Definition 3—is assured by the path synopsis' consistency and by visiting the indexed nodes in document order.

As an example for CAS index creation, again consider the index definition *I(//recordStore/*/year, Integer)*. The evaluation of the query *//recordStore/*/year* on the path synopsis in Figure 6.2 on Page 169 returns the two PCRs 6 and 18. During index creation, all leaf nodes having either PCR 6 or 18 are included in I, e.g., nodes 1.3.7.3, 1.5.7.3, etc.

Note, for the implementation of the CAS index creation process, the inner document structure does not need to be re-computed. A simple scan on the leaf pages of the document store is sufficient. As stated, CAS indexes can only be created on documents stored in path-oriented mode. The rationale behind this restriction is that a path synopsis *must* exist. As shown in the previous chapter, some documents are better stored in a node-oriented way, without the need to maintain a complex and possibly large path synopsis. For those documents, it would be a bad idea to built a CAS (or path) index on a path synopsis (because this would mean to maintain this structure, despite its structural complexity). Therefore, another mechanism to define path indexes would be required. A viable solution to create a CAS index over a node-oriented document would be to construct the *current path* during a document scan, which could then be used to match the index definition path p. This solution is not followed in this work, because the resulting index would not be "connected" to the document store in a way such that index updates could easily be detected. Furthermore, most data-oriented documents for which path indexes are actually interesting can be stored in a path-oriented manner anyway.

Depending on the structure of the path synopsis and the given index path predicate p, we can partition the possible indexes into the three classes *unique*, *collective* and *generic*. These classes will be discussed in the next section.

7.3.4 Unique, Collective, and Generic CAS Indexes

In index definition $I_D(p, T)$, parameters p and T determine the index' selectivity (Requirement 3). In Definition 4, we generalize the CAS index definition from above and identify four different index types.

Definition 4 (Unique, Collective, and Generic CAS Indexes) *In a unique CAS index, all entries have the same PCR, while in a homogeneous collective index, the entries may have varying PCRs. For the heterogeneous collective CAS index, we generalize p to $p = p_1 \vee ... \vee p_i \vee ... \vee p_n$ where the p_i are index paths as in Definition 3. A generic CAS index contains all values of a certain type (i. e., p = //*).*

On our sample data, *I(//cd/year)* is a unique index, *I(//year)* is a homogeneous collective index, *I(//@no ∨ //@length)* is a heterogeneous collective index and *I(//*, [Integer])* is a generic index over integers. While unique indexes are specialized and can answer queries on a single path class only, the selectivity "widens" over collective to generic indexes. Because unique indexes contain records with the same PCR, explicit PCR storage could be omitted to save space. Such a design decision, however, should be supported by a fixed schema, because an insertion of a `cd/year` path at any other position in our sample document would turn the unique index *I(//year)*

into a collective one, which requires PCR storage. Homogeneous collective indexes are the standard case. They potentially require some effort to remove false positives (shown below). For example, query `//cd/[year <= 1980]` requires removal of `vinyl/age` entries in *I(//year)*.

Depending on the selectivities of the path classes included in collective indexes and on the overhead to remove false positives, it can be worthwhile to combine as many path classes as possible in CAS indexes. The more frequent an index is accessed, the higher is the locality of reference on the index pages which, in turn, keeps such pages longer periods of time in the DB buffer [Graefe 07]. Therefore, it could be advantageous to broaden the index use and provide heterogeneous collective indexes.

Finally, we can design indexes combining all path classes of a given indexable type, e. g., Integer, String, or Text (where Text implies the use of IR search techniques). Such generic indexes are not tailored anymore to a particular CAS query, but drastically reduce the number of indexes needed. In our running example, *I(//*, [String])* could serve to evaluate such diverse XPath queries as `//cd[artist="Sting"]`, `//vinyl[genre="Jazz"]`, or `//recordStore/*["B" < title < "D"]`.

7.3.5 CAS Index Maintenance

Upon the creation of CAS index I, a record is inserted into the metadata catalog. This record contains I's index definition $I_D(p, T)$ and also the list of PCRs resulting from the evaluation of p on the path synopsis of document D. With this list, we can decide index matching without evaluating path p for each query to be answered (as shown in the Section 7.3.7). Furthermore, we can detect whether the index has to be updated in case of document modifications.

There are two types of modifications: The first type does not alter the path synopsis while the second one does. Subtree insertions/deletions for the first type or plain content modifications can be handled as follows: For each affected content node n (in the modified subtree), its parent's PCR p is inferred from the path synopsis. If any index definition's PCR list in the metadata contains this PCR (hash lookup), the modification is propagated to the corresponding index, because n is contained in that index. Because the index itself is a standard B*-tree, the well-known and efficient record insertion/deletion/modification algorithms apply. Modifications altering the path synopsis trigger a re-computation of the PCR lists in the metadata. Then, the same process described above updates the indexes. This re-computation is implemented by re-evaluating all index paths on the path synopsis of the document. Note, because the path synopsis is typically small to fit into main memory, this re-evaluation can be tolerated.

As an example, consider index *I(//year)* (PCR list {6,18}) on our sample document. If we alter the content of the first cd's *year* element from 1998 to 1999, we can infer PCR 6 from the affected content node and detect that our index has to be updated. If we add a path `biography/born/year` below the *artist* element, we have to alter the path synopsis, resulting in a new PCR (say 17), which is added to I's PCR list, before the index is updated as above. Note, because the path synopsis is unordered, we do not need to reassign PCRs at any time. Thus, PCRs are stable.

Assume, the document store would not encode any path information, as for example in the node-oriented case and as in many other XML storage proposals. Then,

index maintenance would cause substantially more overhead than in the update scheme introduced above. In the path-oriented case, everything happens in main memory, once the path synopsis and the metadata of a document has been loaded (which happens upon the first document access). In the node-oriented case, however, expensive disk access is necessary to reconstruct the current path on which the modification takes place (without this path information it would not be possible to identify the CAS indexes requiring maintenance). Therefore, embedding path information in the document store not only serves for virtualizing the inner document structure, but also to simplify and optimize the interplay with the index manager of an XDBMS.

7.3.6 Answering Point and Range Queries over CAS Indexes

Assume we have an XPQ expression Q, a document D, and a set of indexes J. The two questions arising now are 1) which set of indexes in J can be used to evaluate Q (index applicability), and 2) how is the evaluation of Q using an index I accomplished (search model)? Because the search model is required to clarify how existing indexes are selected, we start the discussion with the second point. Index applicability will be discussed in the next section.

CAS queries, as specified in Definition 2, can be decomposed into a structural part, i. e., a path p, and into a content part, which is referenced to by T in the following. For example, in query `//recordStore/*["B" < title < "D"]`, the structural part is path p =`//recordStore/*/title` and the content part is a range test T with the exclusive range boundaries "B" and "D". Given a suitable index I_D, a CAS query can be evaluated as follows:

1. Path p is matched against the path synopsis of document D, resulting in a set of PCRs P. If P is empty, the result is also empty, because the document does not contain any path matching p.
2. For content predicate T, a point access or a range scan to/over I is issued to deliver all records matching content predicate T.
3. For the PCR of each record R delivered by the index access, the set inclusion in P is checked. If P contains the PCR, record R belongs to the final result, because its ancestor path matches path predicate p. Thus, this step removes false positives.
4. The correct inner path elements (as defined by the brackets of the predicate) are computed.
5. Optionally, if T is a range predicate and further processing operators require an ordered result, the nodes are sorted in document order. Note, for one and the same value, the nodes are already sorted this way.

For an example, assume we have a collective CAS index *I(//recordStore/*/year)* (as depicted in Figure 7.2 on Page 202) and the query Q =`//cd[year=1998]`. Matching path `//cd/year` against the document's path synopsis returns a set P of exactly one PCR: 6. A point access to index I results in a sequence of three records, of which only the PCR of the first and the last one (having DeweyIDs 1.3.7.3 and 1.25.7.3) are contained in P. Therefore, the remaining `vinyl/year` node 1.19.7.3 (false positive) is filtered out. In the last step, the correct inner nodes are computed, as defined by the predicate in the path query: From the path synopsis, we know that *cd* elements reside on the second level of the path with PCR 6. Therefore, we can

simply clip the delivered DeweyIDs at that level, delivering *cd* nodes 1.3 and 1.25 as query result. In general, result computation may be slightly more complex than in this simple example. We dedicate a separate section in Chapter 8 for the discussion of this topic.

Our search model can be implemented very efficiently. Because B*-trees are search trees, they guide the evaluation of the comparison predicate T. Our implementation interleaves steps 2 and 3, such that the PCR is immediately matched for each scanned record. In Requirement 6, we postulated an efficient result computation for inner elements. As sketched above, the powerful DeweyID + PCR construct serving as a coordinate system in the document construct, allows to do so in main memory, i. e., without document access. This mechanism can, therefore, deliver the "right" input nodes to further evaluation operators (e. g., for query $Q =$`//cd[year = 1998]/title`, which calculates a structural join between *cd* and *title* nodes after a CAS index access).

Assume we could not rely on the cheap ancestor-ID reconstruction mechanism provided by the prefix-based DeweyID labeling scheme. Then, our indexing scheme would either loose flexibility w. r. t. its integration into query processing or it would require rather expensive document access. In the first case, the index could only deliver content/leaf nodes or a positive/negative answer to the question whether the query returns a result on a particular document. In the second case, inner elements would have to be reconstructed either by navigating the document or by using structural joins. As we will expose in our experimental results in Chapter 9, this kind of result computation is substantially slower than direct computation as introduced in our search model.

So far, only the PCR-based evaluation of a query using a CAS index has been discussed. However, the relationship between the path predicate of the query and the path predicate of the index is not established, yet. Therefore, the question arises of whether or not the result is complete. This matter is discussed in the next section.

7.3.7 CAS Index Applicability

The initial task of index-based query processing is to find an appropriate set of indexes in J based on which a path expression Q can be evaluated (Requirement 5). Let $E_D(Q)$ be the result of Q's evaluation on document D and let $E_I(Q)$ be the result of its evaluation using index $I \in J$, as outlined in the previous section. Then, there are four possible cases:

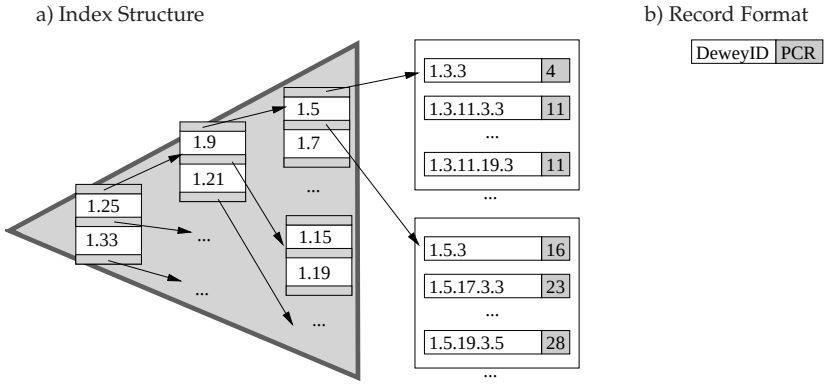
1. $E_D(Q) \cap E_I(Q) = \emptyset$: The evaluation on the document and on the index have no common subset. This either means that the query has no result at all or that the index is not applicable to answer the query.

2. $E_D(Q) = E_I(Q)$: The evaluation on the document and the index returns the same result, i. e., the index is applicable without removal of false positives. In this case, the PCR check (Step 3 in Section 7.3.6) can be omitted.

3. $E_D(Q) \subset E_I(Q)$: In this case, the index contains false positives that make Step 3 above necessary.

4. $E_D(Q) \supset E_I(Q)$: The index does not contain all nodes to answer the query, but only a partial result.

Figure 7.3 A sample plain path index on *recordStore.xml* with definition $I(//title)$



The decision of these four cases based on Q and I's path predicate p alone, i. e., without access to the document, is a difficult problem in the general case (see for example [Hammerschmidt 05, Miklau 04]). Fortunately, the path synopsis and our PCRs provide a basis to solve this problem in a simple way: The above result-set comparison deciding the four cases shown can be replaced by a PCR-set comparison: $E_D(Q)$ is replaced by the evaluation of Q's structure predicate on path synopsis PS, and $E_I(Q)$ is replaced by the evaluation of I's path predicate on PS. Both evaluations return a set of PCRs, based on which the above cases can be decided. In practice, the metadata catalog stores the set of PCRs together with the index definition. Therefore, only the query path has to be evaluated on the path synopsis for the set comparison.

As an example, consider the index *I=(//year)* and the queries Q_1 `=//*[year=1998]` and Q_2 `=//vinyl[year=1998]`. The PCR sets 6,18 for I and Q_1 are equal (case 2) and, therefore, Step 3 can be omitted. For Q_2, the PCR set is {18}. Therefore, Step 3 is required and removes all nodes with PCR 6. Note, in contrast to the use of a plain element index, queries with wildcards (*) can easily be evaluated over CAS indexes. While cases 1 to 3 yield a "positive" result, case 4 signals that the index alone is not sufficient to evaluate the query. However, when multiple (not necessarily unique) CAS indexes qualify, e. g., *I(cd/year)* and *I(//vinyl/year)* for query `//year[. = 1998]`, the qualified node-reference lists of all matching indexes can be merged to derive the result. If the union of all participating PCR sets is a superset of the query's PCR set, the result is complete (but may contain false positives).

7.3.8 Plain Path Indexes and Plain Content Indexes

So far, only indexes that can answer content-and-structure (CAS) queries were introduced. To support plain path queries with indexes, we just have to slightly simplify the proposed indexing scheme. A plain path index definition has the form $I_D(p)$, i. e., no content type information is given. The records stored in a plain path index have the form $R = [\underline{D}, P]$, where D is the DeweyID of the indexed node and P is its PCR (alternatively, we could swap D and P resulting in a different clustering; see next section).

As an example, consider index *I(//title)*, depicted in Figure 7.3. On our sample document, the index collects the following PCR list (see Figure 6.2 on Page 169) upon index definition: {4, 11, 16, 23, 28}. In contrast to CAS indexes, where only leaf nodes are maintained, a plain path index maintains inner document nodes. In path-oriented storage, these nodes are, however, not physically stored, thus, they cannot be retrieved by a simple scan over the leaf pages of the document store. Therefore, to construct a plain path index, the inner document structure has to be recalculated (as described in Section 6.3.3). To do so, the PCRs of the inner nodes have to be generated to check whether a node belongs to the plain path index or not. However, no sorting on the records is required, because the plain path index keeps them in document order. Similar to CAS indexes, plain path indexes can be classified into unique indexes, (homogeneous and heterogeneous) collective indexes, and generic indexes. The generic type (i. e., *I(//*)*) should however be handled with care, because it contains all nodes of the documents and is therefore expensive to maintain. Testing a plain path index for applicability and index maintenance are quite similar to the proposed procedures for CAS indexes. To query a plain path index, the content check is simply omitted. In summary, operations on plain path indexes are quite similar to the ones defined for CAS indexes.

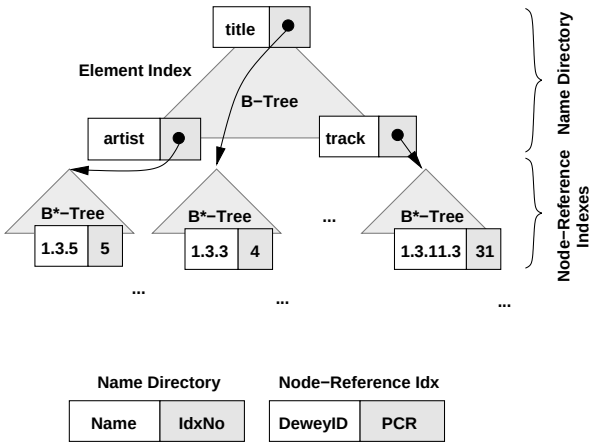
Interestingly, we can also embed path evaluation capabilities into the element index proposed in Section 7.2.2. Logically, the relationship between the element index and path indexes can be regarded as follows: Let $V = \{v_1, \ldots, v_n\}$ be the set of all element names in a document. Then the element index J contains all path indexes $J = \{I(//v_1), \ldots, I(//v_n)\}$. However, as long an element name v_i is not unique within the path synopsis, index $I(//v_i)$ can only answer query $Q(//v_i)$, and nothing else. To alleviate this situation, we provide the possibility to "piggyback" path indexes on top of the element index. This is accomplished by embedding PCRs into the records of each node-reference index. Embedding PCRs is straightforward, because the original record format of node-reference indexes has an empty value field (see Figure 7.1). With a PCR in each record, we can distinguish all paths in $I(//v_i)$ as in an "ordinary" path index. The resulting element index is depicted in Figure 7.4. Regarding maintenance, applicability testing, querying, and so on, the distinct node-reference indexes for element name v_i behave just like a plain path index with definition $I(//v_i)$.

Path indexes were "defined" on the basis of CAS indexes by the omission of content information. Of course, we can also design an index by the omission of structural information. The result is a plain content index that solely maintains value-to-DeweyID mappings. A content index is defined by $I_D(T)$, where D is the document and T is the indexed content type, as before. Content indexes are actually nothing new, as the related work section will reveal. Nevertheless, they are presented here to complete the set of index structures available in XTC. Because content indexes do not encode any path information, they can also be created on a document stored in node-oriented mode.

7.3.9 DeweyID Clustering and PCR Clustering

The records contained in the leaf pages of a CAS index are sorted by the indexed key and, as a second ordering criterion, by the occurrence (i. e., DeweyID) of the indexed node in the document. As an example, consider index *I(//recordStore/*/year)* depicted in Figure 7.2 (with PCRs 6 and 18). Therefore, on the structural part of

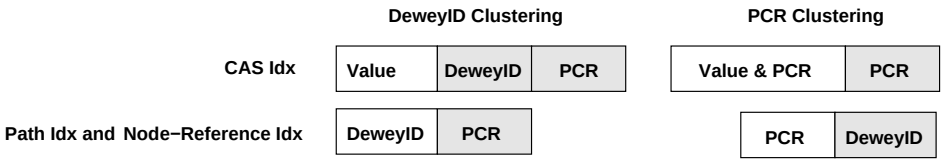
Figure 7.4 The extended element index



a point query like `//vinyl/year[.=1959]`, the DeweyIDs of the resulting *year* nodes are delivered in document order. This is beneficial, if the result of the point query is processed by further XML operators. Actually, the ordering on the key and the DeweyID implies a type of index clustering, which we will refer to as *DeweyID clustering* in the following. Obviously, in DeweyID clustering, the PCRs are distributed randomly over the set of leaf-page records. During query processing over the index, records with PCRs that do not match have to be filtered out. For example, above query only returns records with PCR 18 on the given index. Suppose, the distribution between *vinyl* and *cd* entries is highly skewed, i. e., many more *cd*s are stored. Then, many unnecessary records will be read and many false positives will be removed for this query. In such a case, it would be better to cluster the records by their paths, i. e., by their PCRs. This kind of clustering will be called *PCR clustering* in the following and is achieved by a modified record layout.

Figure 7.5 shows the record layouts that led to the different DeweyID and PCR clusterings on CAS, path, and extended element indexes. Because DeweyID clustering has already been discussed, we only concentrate on the right side. On the extended element index and on a path index, PCR clustering is achieved by swapping the key with the value. The records are then ordered by the PCR (first) and by the DeweyID in document order (second). Of course, the introduced query evaluation over the index does not work anymore and has to be adjusted. The same is true for CAS indexes in PCR clustering, where the key is now a concatenation of the content value and the PCR (forming a combined key) and the value is the DeweyID (sorted in document order for records with one and the same value-PCR combination). Here,

Figure 7.5 Record formats for DeweyID and PCR clustering



we only sketch the modified query evaluation over the CAS index (cf. Section 7.3.6).

Let p be the structural part of a CAS query and let T be the content part. Given a suitable index I_D with PCR clustering, the CAS query can be evaluated as follows:

1. Path p is matched against the path synopsis of document D, resulting in a set of PCRs P. If P is empty, the result is also empty, because the document does not contain any path matching p (this step is the same as for indexes with DeweyID clustering).

2. If T is a point query, for each PCR delivered in the previous step, a combined key is generated that consists of the comparison value and the PCR. For each combined key, an index lookup delivers an intermediate result of records. If T is a range query, the range is scanned using the given range boundaries as prefixes to locate the scan boundaries (note, prefix usage is necessary, because the index contains combined keys). For range queries, the rest of the query evaluation over the index proceeds as sketched in Section 7.3.6 for DeweyID-clustered indexes. The following points only apply for point queries.

3. Optionally, if further processing operators require an ordered result, the nodes of the intermediate results are merged on the basis of the record DeweyID. Note, before merging, each intermediate result is already sorted on the DeweyID field in document order. If no sorting is required, the intermediate results are simply concatenated.

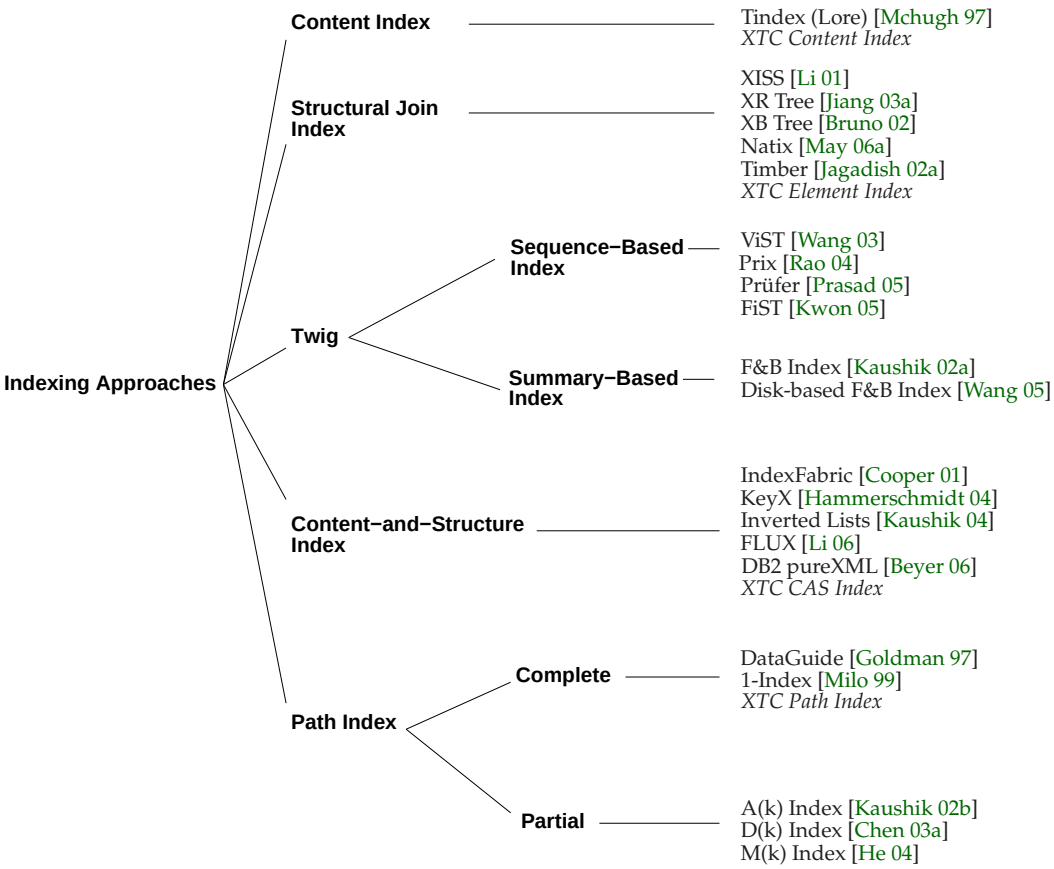
4. The correct inner path elements (as defined by the brackets of the predicate) are computed.

As an example, consider index *I(//recordStore/*/year)* in PCR clustering and query $Q =$`//*[year="1998"]`. Evaluating the structural part of the query (`//*/year`) over the path synopsis delivers PCRs 6 and 18, and thus the combined query keys "1998 & 6" and "1998 & 18". Index lookups for both keys return two result sets, each of which is sorted in document order on the DeweyID field. These result sets are merged and the required inner *cd* and *vinyl* elements are recomputed. For point queries, no removal of false positives is necessary. In the above motivated case, when the distribution of indexes records on different paths is highly skewed, PCR clustering allows to directly "address" the records on the required paths. For example, in query `//vinyl[year="1959"]` only one index lookup an no false positive removal is necessary over the PCR-clustered CAS index. However, if the document is structurally complex, and many intermediate results have to be merged (due to many queried path classes), DeweyID-clustering could be advantageous. As we will see in the experimental section, DeweyID-clustered indexes perform better on explorative queries (where the use of "//" leads to many path classes), whereas PCR clustering should be preferred on selective queries (using "/" and fewer path classes). Finally, we want to state that the evaluation over path indexes is implemented in an analogous way.

7.4 Related Work

Figure 7.6 provides an overview over recent indexing proposals for semistructured data and XML. As you can observe from the number of references, XML indexing a very active field of research. You will also see that many different notions exist on

Figure 7.6 A classification of related work on XML indexing



what "XML indexing" actually means. For brevity, we only sketch some approaches out of every class.

7.4.1 Structural Join Indexes and Content Indexes

The first class of indexes support join-based query evaluation. Therefore, they are called structural join indexes. A structural join [Al-khalifa 02] (or a holistic twig join [Bruno 02], which is actually an extension) can evaluate a path query by joining ordered input lists of element, attribute, and text nodes, in the style of a merge join (see Chapter 8). For example, assume query `//cd//track` which returns all *track*s of *cd* elements. A structural join can read two input lists, one containing *cd* elements, e. g., 1.3, 1.7., etc., and the other one containing track elements, e. g., 1.3.11.3, 1.3.11.5, etc. Then, based on the structural relationships of the elements (in this case, the *descendant* relationship), those *track* elements that actually have a *cd* element as ancestor are returned. To provide the input lists for structural joins, a document scan can be used. However, in most cases, the number of elements read will be much higher than the number of elements required for structural join processing. Therefore, appropriate index structures are required.

A quite simple implementation is the XISS (XML Indexing and Storage) scheme

[Li 01], which maintains a posting list of nodes (identified by a range-based labeling scheme) for each distinct element name. The structure is quite similar to our element index (without embedded PCRs). However, the element index additionally organizes the posting lists in a B*-tree index and therefore can support scan-based XPath axis evaluation, as introduced in Section 7.2.2. As we have seen, for some axes, the scan-based evaluation is not very meaningful, because the selectivity of the axis is too high. The XR-tree (for XML Region Tree) [Jiang 03a] alleviates this situation for the *ancestor* axis by additionally storing ancestor information in an extended form of B*-tree. The authors provide a special structural join operator working on the XR-tree that can exploit ancestor information to speed-up processing time. Finally, the XB-tree [Bruno 02] manages certain node label ranges to enable range skipping. The index can signal, when a certain range of nodes does not contain any match. Obviously, the element index with embedded PCRs makes structural joins for the *child* and *descendant* axis completely unnecessary, because it allows to directly match complete paths (without further algorithmic processing).

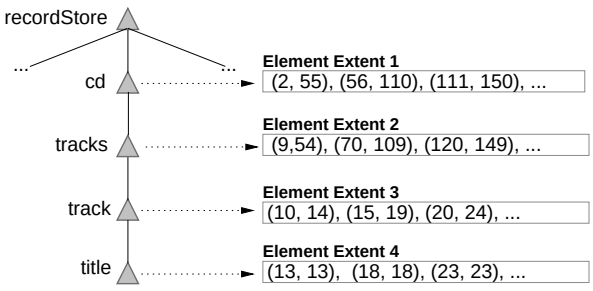
Structural join indexes do not support queries with content predicates, because only elements are indexed, i. e., it is not possible to evaluate query `//cd[year="1998"]`. A content index can remedy this situation. The first content indexes were proposed in the context of the Lore system [Mchugh 97]. The content index introduced in Section 7.3.8 follows the same idea (except that instead of plain integer node labels as in Lore), DeweyIDs are indexed. Because the result of a content index access (point or range) can be used as input to structural join operators and thus help to evaluate content predicates in XPath expressions, content indexes are classified as structural join indexes.

7.4.2 Path Indexes

Indexes to speed up path queries (i. e., structure queries without content predicates) were developed quite early in the XML history, e. g.DataGuides [Goldman 97] and the 1-Index [Milo 99]. Basically, a path index consists of a structural summary (such as the path synopsis) and a collection of *extents*, where each extent is assigned to a node in the structural summary. Each extent contains a collection of XML nodes. These nodes form an equivalence class, i. e., they have a certain property in common. The definition allows a certain kind of freedom in how the equivalence relation inducing these equivalence classes can be defined. This led to the various indexes, such as the DataGuide [Goldman 97], the 1-Index [Milo 99], the A(k)-Index [Kaushik 02b], the D(k)-Index [Chen 03a], the M(k)-Index [He 04], and the F&B-Index [Kaushik 02a].

Common to all these approaches is that they were developed with the semi-structure data model in mind (e. g., OEM [Papakonstantinou 95]) and not particularly for XML. The major difference between these two models is how non-tree edges are handled. In OEM, non-tree edges are first-class citizens, i. e., they are handled as ordinary edges resulting in a graph data model. XML is tree-based and non-tree edges are expressed via id/idref value-based relationships. Therefore, in our understanding of an XML index, non-tree edges do not need to be indexed. Non-tree edges imply additional structural complexity. As a result, true semi-structured indexes may become quite large. For example, the DataGuide can be of exponential size w. r. t. the source [Goldman 97]. Therefore, the research community spent quite some effort to control the index size. In the beginning, there were the DataGuide

Figure 7.7 A sample path index



and the 1-Index. On tree-shaped data, the 1-Index and the DataGuide are the same. In the 1-Index, two nodes u and v are in the same extent (equivalence class), if they have the same incoming label path. The equivalence relation inducing the 1-Index is called *bisimulation*. Figure 7.7 shows an example (note, the Figure is quite similar to the XSum storage structure displayed in Figure 6.10 on Page 188). The IDs in this figure are range labels, which serve as a node identification mechanism in most indexing approaches. On graph-structured data, the 1-Index can still have many extents. Therefore, [Kaushik 02b] relaxed the equivalence relation to k-bisimilarity: not the complete incoming label path of nodes u and v has to be equal, but only the first k ancestors. With this relaxation, the number of extents is also diminished. However, now only queries up to length k can be answered directly by the index. Other queries require expensive post-processing. The D(k)-Index [Chen 03a] and the M(k)-Index [He 04] both refined this idea by allowing different values of k for different "parts" of the index. Another idea is the F&B-Index [Kaushik 02a], which can answer branching path queries. We will discuss this type of index in Section 7.4.4.

The major differences between our approach and the proposed path indexes are:

- We index tree-shaped data, while the above introduced path indexes consider graph-structured data. We think that our restriction is justified, because we apply indexing in an XML database system and not in a system for graph-structured data. However, nevertheless, we exploit the 1-Index as a structural summary. As long as we are dealing with tree-structured data (i. e., as long as we can assign DeweyIDs), our approach can also be integrated with the A(k)-Index, the D(k)-Index, and the M(k)-Index. This could be beneficial for structurally complex documents. However, we did not follow this thread.
- Our approach does not store "inner" extents. Because we rely on DeweyIDs, we can reconstruct inner nodes.
- Our indexes can directly deliver inner elements without further document or index access. For example, consider the evaluation of query `//cd[.//track]` on the sample index in Figure 7.7. A single access to the *cd* extent is not sufficient, because it may contain false positives (in general, a *cd* element might not have a *track* element). Therefore, a structural join between the *cd* extent and the *track* extent is required to remove false positives. In our approach, inner elements can simply be computed, thus, avoiding I/O.
- We provide a flexible mechanism to define index selectivity. Although other approaches (e. g., the T-Index [Milo 99]) allow to define path patterns for index-

ing, our approach reaches further, because we also provide for merged indexes [Graefe 07].

7.4.3 Content-and-Structure Indexes

Indexing content and structure separately results in additional processing cost in case of CAS queries (because content and structure has to be joined again). We assume that CAS queries are quite frequent. Therefore, we developed CAS indexes in this work. The idea of a CAS index has been proposed earlier. IndexFabric [Cooper 01] is one of the first proposals. To store a document in IndexFabric, all tag names are first abbreviated by a unique designator. Then, all root-to-leaf paths are encoded into strings and stored in a paged and layered patricia trie. For example, path `recordStore/cd/artist/"Sting"` is encoded as RCASting, where R stands for *recordStore*, C stands for *cd*, and so on. A content-and-structure query is answered by matching the encoded structure part against the trie and by a subsequence content predicate test. An obvious disadvantage is that the descendant axis can result in substantial trie traversal implying high I/O costs. Furthermore, even if the content predicate is highly selective, the structure part has to be evaluated first. Another disadvantage is that IndexFabric can only return leaf nodes and no inner elements, thus requiring additional post-processing to deliver inner nodes.

An alternative CAS index is KeyX [Hammerschmidt 04]. For the creation of a KeyX index, the database administrator has to define an index path pattern and a result path pattern. The result path pattern has to be a "subpath" of the index path pattern, for example `//cd/year` could be the index path pattern and `//cd` could be the result path pattern. The index path pattern is evaluated against the document returning a list of nodes. For each node, a record is created where the record key is the content of the node. The record value is a reference to the ancestor node in the result path. All records are written into a search tree. With this construction, it is possible to evaluate the content predicate of a CAS query and to return inner elements. However, this structure is quite static compared to our approach, because we can freely compute *all* inner elements and do not require an output path pattern.

Another approach was presented by [Kaushik 04]. Basically, the authors bring inverted lists (on the XML content) and the 1-Index together by embedding a reference into each entry of the inverted list. However, because they do not use a prefix-based labeling scheme, they cannot compute inner elements as we do. Furthermore, they also suggest to answer the structural part of a CAS query first, before the inverted lists are checked.

As a last CAS index, we consider the FLUX approach [Li 06], which is quite similar to ours. Instead of embedding a PCR with each stored content value, the FLUX index embeds a Bloom filter. This filter is generated by applying a hash function to the label path of the content node. The obvious problem with this approach is that the bloom filter generates false positives upon query evaluation. These false positives have to be removed by a subsequent document access. Furthermore, also this approach cannot directly return inner elements.

7.4.4 Twig Indexes

To answer a twig query (with multiple branches) on a set of path indexes, the twig query is first decomposed into single paths. These paths are then matched with the help of the indexes and then joined (or intersected). As a result, some post-processing is necessary to compute the final result (how this actually works in XTC will be shown in Chapter 8). To avoid joins, twig indexes can answer path pattern queries directly. Essentially, two techniques have been proposed as twig index structures: sequence-based indexes and variants of the F&B Index [Kaushik 02a].

The key idea behind *sequence-based indexes* is the following: an XML document can be represented as a sequence of nodes S_D; a tree pattern query (i. e., a branching path query) can be represented as a sequence of nodes S_P. Essentially, S_D and S_P can be represented as strings. Therefore, the problem of tree pattern matching can be transferred to *substring matching*. The first approach exploiting this idea was ViST [Wang 03]. In ViST, a document is represented by a two dimensional sequence $(a_1, p1), (a_2, p_2), \ldots, (a_n, p_n)$, where a_i is a node and p_i is the path of the node. Queries can be represented in a similar fashion. For query processing, the documents are stored in a *suffix tree*, which is indexed by a B*-tree. Subsequence matching to find tree pattern matches is then implemented using this B*-tree. Two major problems with the approach are that false positives may occur and that the space complexity of the B*-tree is quadratic to the size of the document [Rao 04]. Therefore, [Rao 04] proposed PRIX—a similar approach, which relies on so-called Prüfer sequences. Their indexing scheme avoids false positives and guarantees a linear correspondence between document size and index size. False positives are avoided by refining the delivered subsequence matches against the pattern tree structure. A further similar approach was published by [Prasad 05, Kwon 05].

The major advantage of sequence-based indexes is their ability to match complete path patterns with content-based predicates. On the flip side, the technique suffers from the following substantial problems: 1) Pattern trees with descendant axes and wildcards require navigational document access for their refinement (i. e., to assure that the matched sequence is really a tree pattern match). 2) The pattern trees in query processing are order indifferent. Therefore, to find all matches, all sequences generated for pattern trees with permutated branching paths have to be generated, matched, and unified. 3) Sequence-based indexes encode the complete document and are, therefore, not selective (as required in our list of desiderata).

Another approach to twig indexing was presented with the F&B Index by [Kaushik 02a]. The F&B Index assumes a graph data model and defines its own notion of a branching path query containing, for example, also operators to follow non-tree edges and navigations along the parent and ancestor axis. In the F&B Index, all nodes are grouped together into an extent that "cannot be distinguished by a branching path query". This means that if two nodes u and v are in the same extent, *all* branching path queries return either both nodes or none. The problem here is the word "all". Because branching path expressions can get arbitrarily complex, the extents can get quite small. In the worst case, each extent contains only one node. Then, query processing over the F&B Index is as expensive as evaluating the query over the document. Therefore, the authors suggest various methods to reduce complexity by restricting the set of indexed tag names, applying the idea of k-bisimilarity, restricting the complexity of branching path queries, and ignoring non-tree edges. An external memory mapping for the F&B Index was given

by [Wang 05]. The mapping is quite complex and does not rely on standard index structures, such as the B*-tree. In their experimental results, the authors also reported that the index can easily become quite large ranging between 57% and 158% of the original document size. Besides these problems, no index maintenance algorithm has been proposed until the time of writing.

7.4.5 Indexing in Related Query Processors

Let us now take a short look at the indexing capabilities of the five systems introduced in our overview chapter (Section 2.3): We are not aware of any possibility to create and use indexes in Galax, MonetDB/XQuery. Both systems operate on main-memory document representations and, thus, do not require any external memory index structures. [May 06a] briefly describe an indexing scheme for Natix, which is quite similar to our element index (without embedded PCRs). However, how this index is created and whether the query engine makes use of it is not known. Timber also has a similar indexing framework as XTC *before* path indexing was introduced, i. e., they have an equivalent form of the element and the content index [Jagadish 02a]. In contrast, DB2 pureXML provides for path and CAS indexes, as we do [Beyer 06]. However, again, only the external interface is known (i. e., how these indexes are created by the user), but not how they are implemented.

7.5 Summary

This chapter introduced a flexible indexing framework for the XTC native XDBMS. To fulfill our "wish list" posed at the beginning, XTC's basic indexing scheme (consisting of the ID-attribute and the element index) was substantially extended by content, content-and-structure, and plain path indexes. Our index structures are tightly integrated with the document store to foster code reuse (e. g., the reuse of the path synopsis) and to detect and implement index updates. Our indexes can be freely adjusted to the physical DB layout and to the expected query workload. Finally, we developed measures to detect index applicability and we have motivated the importance of returning inner elements to facilitate the integration of the index with the remaining operators of the physical algebra. The latter point will be discussed again in the following chapter, where we show how index scans are actually embedded into a physical XML algebra.

Chapter 8 The Physical Algebra

A goal without a plan is just a wish.

Antoine de Saint-Exupéry

At this point, we have approached XML query processing from two directions: 1) from the logical abstraction level, where we studied the problem in a top-down manner from parsing, over normalization, static typing, simplification, and XQGM translation to rewriting; 2) from the physical abstraction level, where we prepared the foundation in a bottom-up manner by considering the storage and indexing layout of a database. To close the remaining gap, all we have to do is to present the algorithms that run on top of the storage and indexing layout implementing the various XQGM operators. Furthermore, we have to show how these algorithms can be assembled to an executable program.

The latter problem is commonly known as *plan generation*. We have already seen in the overview chapter (Section 2.1.3) how plans are conceptually built: Plan generation is a two-phase process. The first phase is implemented by a rule-based approach, similar to the rewriting stage. When a *plan generation rule* matches an XQGM component, the rule creates the physical implementation of the logical operator. This implementation can be a single algorithm (also called *physical operator* in the following) or a whole group of algorithms (i. e., a tree of physical operators). The implementation is registered with the XQGM component it was created for. When multiple rules match on one and the same XQGM operator, multiple alternative implementations are registered.

After all XQGM operators have been mapped to their implementations, the plan generator walks over the XQGM instance and stitches the physical implementations together. Note, this actually implies that the rule set for plan generation is complete, i. e., that for every logical operator, a matching rule exists. At the time of writing, the plan generation process is implemented in a straightforward manner: If multiple alternatives for an XQGM operator exist, always the first one previously registered is chosen. To make the generation of alternative QEPs for a query possible, plan generation rules can be priorized such that a rule with a higher priority registers its implementation before a rule with a lower priority. In a real cost-based query optimizer, the implementation alternatives will be chosen by the query optimizer based on some cost model, thus, making rule priorities unnecessary. However, because this is future work, we satisfy ourselves with the above sketched simple plan generator.

For the rest of this work, we omit a technical discussion on the plan-generation rule

set and on how plans are actually assembled. Instead, we give an overview over the set of physical operators (called *physical XML algebra* in the following) and over alternative implementation strategies of certain XQGM operators. You will see that most logical-to-physical mappings are quite intuitive (we will give some examples). Note, we have not only developed these concepts on paper, but also implemented them in the XTC system. The query processor was demonstrated by [Mathis 08].

8.1 An Introduction to the Physical Algebra

We can roughly group the operators of the physical algebra into *path processing operators* (PPOs) and into "the rest". The rest consists of operators not developed for document access, such as selection, projection, grouping, unnesting, merging, tuple generation, etc. We omit the discussion of these operators, because, essentially, we have already introduced them in Chapters 4 and 5. With slight deviations, most algorithms of the logical algebra are implemented one-to-one in the physical algebra. Thus, LAL operators are PAL operators. However, the plan generator does not take the "detour" over LAL operators, but directly generates PAL operators out of the XQGM representation. This "mapping" implies that also the data model of the logical algebra is used for the physical algebra. At the end of this chapter (Section 8.6), we will give a summary how the PAL operators differ from the LAL operators. But first, let us consider algorithms for the evaluation of path expressions.

Path expressions occur quite frequently in XML queries. For their evaluation, the document or appropriate index structures have to be accessed, thus often external memory access is implied. From the perspective of a DBMS developer, operations that occur frequently and possibly require expensive external memory access are naturally those algorithms that receive special attention. This is also the case in the work at hand and, therefore, this chapter is mainly dedicated to PPOs. We distinguish *navigational*, *join-based*, and *index-based* PPOs. The first group of operators is also the most expressive one; every path expression in a query can be evaluated by navigations on the document. As we will see in Chapter 9, compared to join-based and index-based methods, they are, however, often enough the group of operators with the slowest performance. Hence, navigational primitives are a "fallback solution", when no operators of the other two groups can be applied to evaluate a certain path expression.

Join-based operators stream through the document or over the element index and evaluate path expressions by matching structural relationships among the streamed nodes. Compared to navigational methods, they often provide for better performance. However, their use is restricted to certain XPath axes. The two most prominent representatives for this group are *structural joins* (STJ) and *holistic twig joins* (HTJ). Especially holistic twig joins have gained much attention in the scientific literature and many variations of the original algorithm [Bruno 02] have been presented. Most of these variations aimed at optimizing the algorithm's structure matching phase and at increasing its performance. Of course, performance will also be an issue for the holistic twig join operator developed in this work. However, we furthermore pay attention to the integration of the HTJ operator into XTC's physical algebra and show how the operator can work hand-in-hand with the rest of the algebra.

The last group of operators provides index access. In the previous chapter, only the structure of our indexes have been presented, while the algorithms working on path and CAS indexes are proposed in this chapter. In particular, we will see, how path queries extracting inner elements (such as `//cd[id="cd_101"]`) can be answered with path/CAS indexes and how index-based operators can be "married" with join-based operators. Note, index-based operators have yet again a reduced expressiveness compared to join-based operators, because join-based operators can match arbitrary branching path patterns and index-based operators can only match linear paths. Below, we will give an overview over the PPOs in the physical algebra, starting with navigational PPOs.

The interface of all physical algebra operators is quite simple. It consists of only three methods named *open*, *next*, and *close* (i. e., the XML algebra implements the ONC protocol). Generally, the operators in the XML algebra from a directed acyclic graph (DAG) structure, i. e., an operator consumes the output generated by one or more input operators and produces output for one or more operators. As an example, consider the operator tree depicted in Figure 2.6 on Page 24. The *open* method of the operator interface causes an operator to initialize its internal data structure. An *open* call is passed to all children of an operator. Likewise, the *close* call is passed recursively over the DAG structure and causes to clean-up internal data structures, when the query is evaluated. The *next* method also recursively descends down the DAG structure and delivers a single result item. If no further results are produced, the *next* method returns a NULL value. For the implementation of the physical XML operators in XTC, ONC was chosen because it hides cumbersome aspects of control flow and data flow and is, therefore, easy to implement (compared to an active scheduler as an alternative solution for these tasks).

8.2 Navigational PPOs

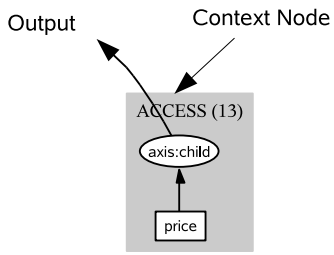
As we have seen in Chapter 6, the document store provides functionality for navigational access and scan access. The operators introduced in this section are built on these navigation and scan capabilities. Because the document store is the place where the document resides, navigation and scan are always "available" (i. e., no secondary index structures are necessary). This is why navigational PPOs provide the default path algorithms in the physical XML algebra.

Navigational PPOs evaluate XPath axis steps, i. e., given one (or more) input node(s) and an XPath axis with a node test, the PPO returns all nodes on the axis that fulfill the name test. Depending on the input cardinality, we can distinguish operators that work on a single node (also called *context node* in the following), and operators that work on a whole sequence of input nodes. The first set of operators shall be discussed first.

8.2.1 A Single-Node Navigational PPO

Often, a situation as depicted in Figure 8.1 occurs in an XQGM: A navigational access operator has to evaluate an axis step starting from a context node delivered by a correlated input edge (dotted line). A similar situation expressed in XQuery would have the form `$d/child::price`, where `$d` is bound to the context node

Figure 8.1 A navigational access operation in the XQGM (for an example, see Figure 2.4 on Page 20)



(i. e., the navigation starts at a single node). In XTC, there are three possibilities for the implementation of this operator:

1. by opening the document store at the given context node and delivering all required nodes by navigations, i. e., by calls to the internal taDOM interface (Figure 3.2 on page 33);
2. by opening the document store at the given context node and delivering all required nodes by a scan, i. e., by calls to the internal node scan interface, see Page 176;
3. by an access to the element index, if it exists (see Section 7.2.2 on Page 197).

You might argue that the name "navigational PPO" might be a bit misleading here, because actually also scans are used. The name was chosen, because logically, the XQGM operator in Figure 8.1 evaluates a navigation. Let us start with the first implementation.

Navigating the Document Store

As a convention throughout this work, physical operators will be described by their constructor method signature. Operators navigating the document store are called *axis-step navigational operators* and have the following signature:

```
AxisStepNavigationalOperator (Axis axis,
                              NodeTest nodeTest,
                              TupleGenerator contextItemProvider);
```

The first parameter is the navigation axis and the second one is the node test that shall be applied to all retrieved XML node. The third parameter is a *tuple generator*, which is also a physical operator. Basically, the context item provider is passed as a parameter to the axis-step navigational operator by the plan generator. The tuple generator provides the context item from which the navigation starts.

The skeleton of the algorithm is the same for all axes: 1. The context item is retrieved from the tuple generator; 2. all nodes on the given axes are retrieved from the document by navigation; and 3. for each delivered node, the node test is evaluated (actually this happens immediately when the node is returned by a navigational operation). If the node test is passed, the node is returned upon a call to the *next* method and belongs to the result, otherwise it is skipped.

Delivering all nodes on a certain axis using the internal taDOM operations from Figure 3.2 is more or less straightforward, which is why we omit a detailed discussion here. As an example, consider the *child* axis which is evaluated by calling

getFirstChild on the context node and repeated calls to *getNextSibling*. Reverse axes need special care because, according to the XPath Recommendation, they also deliver the sequence of result nodes in document order. Therefore, for the implementation of the *preceding-sibling* axis, repeated calls to the *getPreviousSibling* method would not return a correct result. Rather, the first sibling has to be retrieved first (by navigating to the parent of the context node and a call to *getFirstChild*) and then repeated calls to *getNextSibling* have to be issued until the context node is reached. The remaining axes can be evaluated similarly.

Scanning the Document Store

For some axes, retrieving all nodes from the document store by navigation is rather cumbersome. For example, navigating all nodes on the *descendant* axis requires the following actions: 1) navigate along first-child edge until no more nodes found; 2) try navigating next sibling until no more nodes found; 3) try navigating parent and then next sibling, if not successful, try parent twice and then next sibling, and so on; 4) if successful (not all descendants found), proceed with 1), otherwise stop.

As you can observe, this strategy leads to navigations into the void (e. g., "no first child found") and nodes visited twice (because of the parent navigation). Thus, actually more navigation steps are required than nodes returned. Furthermore, in XTC, navigational calls provoke random I/O, because for each step, the document index is traversed. Of course, when the page of the context node resides in the buffer, it is also likely that the neighboring nodes are also buffered. However, still a much better implementation for the *descendant* axis builds on a document scan operation. Remember that the nodes in the document store are stored in document order and are neatly packed side-a-side into the pages. Therefore, the document scan delivers them in the right order, as required by the axis step.

Therefore, we get another operator called *axis-step document scan operator* with the following signature:

```
AxisStepDocScanOperator (Axis axis,
                         NodeTest nodeTest,
                         TupleGenerator contextItemProvider);
```

As before, the skeleton of these operators is the same: 1) the document index is opened (either at the context node or at the root node, depending on the axis); and 2) the document is scanned until some termination condition (which also depends on the axis) is reached. The start and stop conditions for the eight major axes can be summarized as follows:

1. *descendant*: The scan begins at the first child of the context node and ends, when the subtree below the context node is reached.
2. *child*: The scan has the same range boundaries as defined for the *descendant* axis. Besides the plain node test of the axis step a child check is also required (because the scan sweeps over all descendant nodes, of which the child nodes are only a subset).
3. *following-sibling*: The start node is the following sibling of the context node. The scan completes on the last child of context node's parent. As for the child axis, the structural relationship needs to be checked for every delivered node.
4. *following*: Since the descendant nodes of the context node do not belong to the following axis, the next following node of the context node is the start point for

the scan. The end is reached when the complete document has been scanned.

5. *parent*: A document scan to retrieve the parent of the context node is obviously not very meaningful. Therefore, this operator does not exist.
6. *ancestor*: For the same reason, the ancestor axis is not evaluated by a scan.
7. *preceding-sibling*: The scan starts at the first child of the context node's parent and stops on the context node. As in the range scan for the following-sibling axis, the nodes have to be structurally checked.
8. *preceding*: The document root is the starting point for the scan, whereas the context node marks the end point. Ancestors of the context node do not belong to the preceding axis, therefore, they need to be filtered.

You can observe that some axes can retrieve quite a large number of nodes from the document (i. e., *preceding*, *following*, and *descendant*), while others are more selective (e. g., *child*, *following-sibling*, *preceding-sibling*). The latter group possibly discards a large number of nodes returned by a scan that would not have been touched by the navigational implementation. In the end, the structure of the document and the location of the context node decide, which of the two implementations is superior. Thus, finding the right alternative during plan generation depends on document statistics component and a cost model.

Evaluation via Access to the Element Index

The evaluation scheme introduced by the above two operators can not only be applied to the document store, but also to the element index. Section 7.2.2 has already shown by example how scan-based and navigational axis steps can be evaluated. Therefore, we keep this section short.

For scan-based access, the same range boundaries as introduced before can be used. For "navigational" access (i. e., to implement the highly selective axes *parent*, *ancestor*, and *ancestor-or-self*), the necessary node IDs on the axis in question are computed from the DeweyID of the context node and subsequently probed against the element index. Because the element index can only be queried for element names, these operators only support the evaluation of axis steps with *name tests*. Further types of node tests are not possible. Here are the two operator signatures for scan-based an navigational element access:

```
AxisStepElIdxNavOperator (Axis axis,
                          NodeTest nodeTest,
                          TupleGenerator contextItemProvider);

AxisStepElIdxScanOperator (Axis axis,
                           NodeTest nodeTest,
                           TupleGenerator contextItemProvider);
```

8.2.2 A Multi-Node Navigational PPO: NavTree

XPath semantics demands that the result of an axis step is duplicate-free and in document order. Therefore, during normalization, *ddo* function calls are embedded after all axis steps. Throughout this work, we have seen quite some examples for that. Essentially, the implementation of the *ddo* function is based on sorting. In general, sorting is considered a rather expensive operation in query processing because of its non-linear worst-cased complexity and because sorting is a pipeline breaker.

Therefore, the *ddo* function should be avoided whenever possible.

One way to reach this goal is to implement the path processing operators in a way such that they already deliver a sorted and duplicate-free result. With the operators introduced so far, this is impossible because they do not possess enough "context knowledge" to ensure these requirements (their context is only a single input node, from which the navigational operation starts). As an example, consider XQuery expression `$d/descendant::w`, where $\$d$ is bound to sequence S. S is a series of context nodes that serve as starting points for the navigation. Let S contain two nodes u and v, where v is a descendant of u. Suppose that in the document, a node w exists, which is a descendant of both u and v. On the evaluation of axis step `$d/descendant::w`, the above algorithms would return w twice (because they are evaluated both on u and v), thus removing duplicates in the final result would become necessary.

With the "context knowledge" of S containing nodes with the descendant relationship, we can avoid duplicates. The simple idea is to *skip the axis evaluation on certain nodes* (because their "relatives" contained in S already delivered or will deliver all necessary information). While this idea ensures a duplicate-free result, document order is not necessarily established. Further measures need to be taken, as we will see in the following discussion.

The Base Algorithm

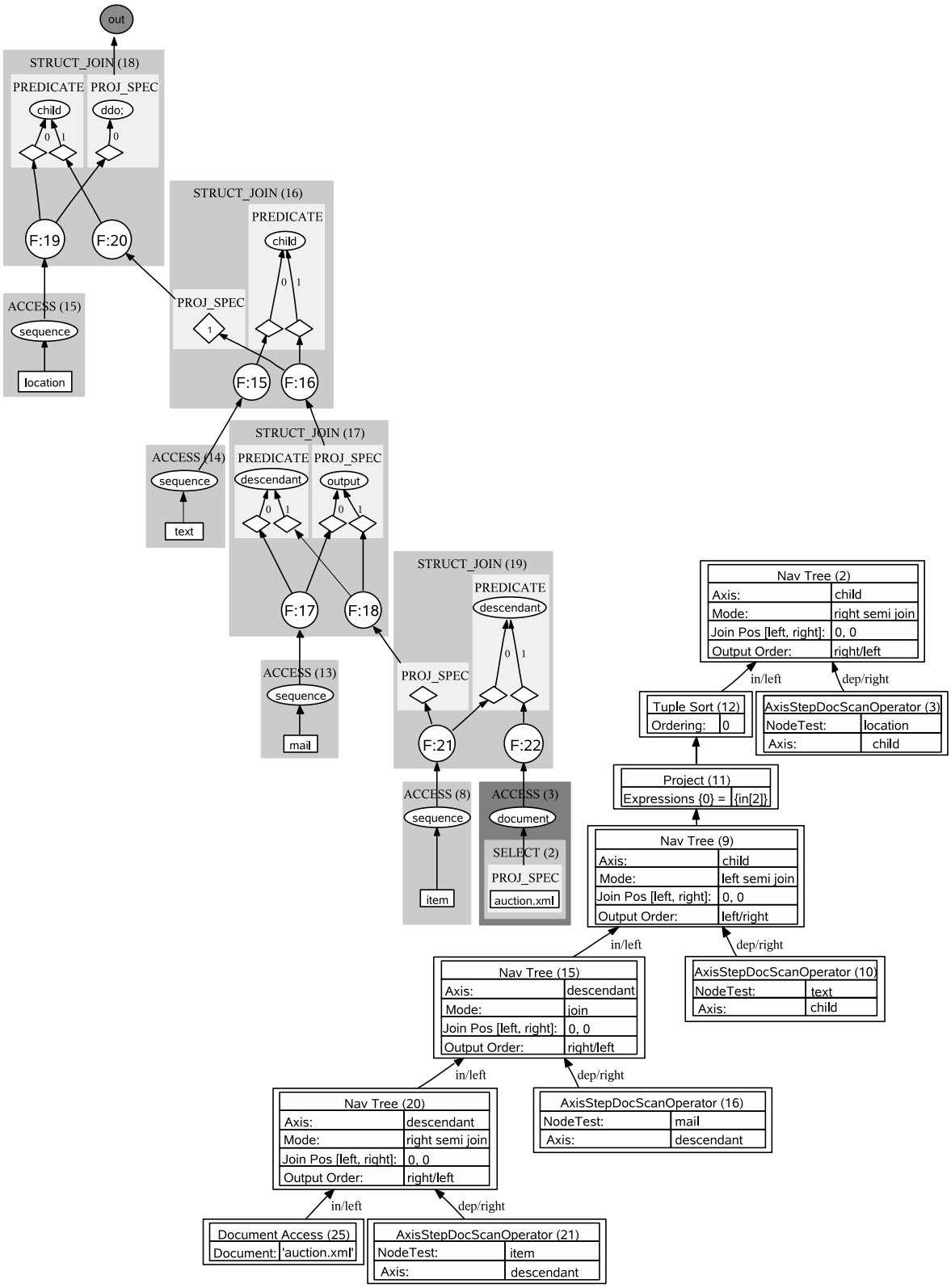
In contrast to the above operators, the *NavTree* operator consumes a complete, sorted and duplicate-free sequence of input nodes (and not only a single node) and returns a sorted and duplicate-free sequence of output nodes. This actually means that the algorithm is closed w. r. t. these two characteristics and, therefore, it can be combined neatly to evaluate arbitrary paths according to the XPath semantics. Because multiple nodes are navigated "simultaneously", we call this kind of algorithm a *multi-node navigational PPO*.

The above discussion motivated solutions to avoid duplicates and to establish the document order. However, it did not reveal the complete truth about all the features of the algorithm. For its integration into XTC's physical XML algebra, the algorithm, furthermore, needs to support different kinds of joins (*semi joins*, *full joins*, and *outer joins*). This becomes immediately clear, when we think about how our context sequence S (containing the starting nodes for the navigations) is provided to NavTree, namely as an input stream of nodes. This can only be achieved, if the input is decorrelated. Therefore, the NavTree operator actually implements a structural join and, thus, all join variants occurring in the XQGM have to be offered[1]. In summary, the plan generation rule for the NavTree operator searches for a structural join, which receives the result of exactly one sequence access operator as input. We will see some examples below. For a full join, the requirement to bring the result into distinct document order makes no sense. Therefore, we establish this order only in semi-joins. However in a full join, the ordering of the result also allows some freedom: the result can be returned in *right/left* order or in *left/right* order.

Let us motivate these considerations by an example: Figure 8.2 shows the XQGM instance and the physical plan for query `doc("auction.xml")//item`

[1]Note, this also means that NavTree cannot be applied to implement the XQGM operator presented in Figure 8.1 on Page 220.

Figure 8.2 An output ordering example: An XQGM instance and its physical plan



`[.//mail/text]/location`. On the left-hand side, we see the query completely unnested in XQGM representation. The XQGM instance consists of four structural joins, of which three are semi joins and one is a normal join. The right-hand side of the figure presents the physical plan. To visualize the physical properties of the operators, we use the tableau technique [Mitschang 95]: Every physical operator carries a name and a unique ID. Each row in a tableau presents one physical property. As you can see, the plan consists of four NavTree operators, each expressing one of the four logical structural join operators. The plan is read bottom-up. The first operator (20) joins the virtual document root with all *item* nodes. The document root is delivered by a document access operator; the *item* nodes are delivered by an `AxisStepDocumentScanOperator`, which we introduced in the previous section. Note, this scan operator is utilized by the NavTree operator, as we will see below. The NavTree operator computes the *descendant* axis and only returns the *item* nodes. Therefore, it is a (right) semi join operator. The property *join position* clarifies on which fields of the input tuples the join is computed. Finally, the *output order* property defines how the result is sorted. Here, only *left/right* or *right/left* are possible. In case of a semi join, the input stream delivered as output is up front (and the second one is not significant). In case of a full join and the combination *left/right*, the left input is the first ordering criterion, while the right input is the second ordering criterion.

In our example, the first `NavTree(20)` operator returns only *item* nodes. The next `NavTree` operator (15) returns pairs of *item/mail* nodes, ordered *right/left*, i. e., by the *mail* nodes first. Here, a full join was generated during XQGM rewriting, because the *item* nodes are required for the final output. `NavTree(9)` joins the *mail* nodes with the *text* nodes. Only the left input is required (and therefore ordered by the *mail* nodes). Because the left input was a pair, we retain only the *item* nodes and discard the *mail* nodes after `NavTree(9)`. This is done by the projection operator. Afterwards, there is a small problem: Because `NavTree(9)` ordered the output by the *mail* nodes, this does not necessarily mean that the *item* nodes are in document order. Therefore, they have to be sorted by a TupleSort operator. The remaining NavTree operator computes the resulting *location* elements by a semi join.

While the physical plan is assembled, the plan generator fixes the output order of the joins. To avoid additional TupleSort operators, the order is set such that the following join operator (if any) receives its input in the correct order. If this is not possible, a TupleSort operator is injected. Due to the lack of space, we do not give an in-detail view on how this actually works.

After this discussion about the features of the NavTree algorithm, let us come back to its implementation. All in all, the algorithm consists of three components: an *input filter* responsible to avoid duplicates, the *base algorithm* controlling the evaluation, and an *output generator* to establish the correct result order. The first and the last component will be discussed in the following sections. Here, the base algorithm is introduced. The NavTree operator has the following signature:

```
NavTreeOperator (Operator input,
                 int leftJoinPos,
                 int rightJoinPos,
                 Axis axis,
                 NodeTest nodeTest,
                 JoinMode joinMode,
                 Sorting sorting,
                 AxisStepOperator baseImplementation);
```

The given operator `input` provides the duplicate-free and sorted input sequence. According to the data model, this input sequence consists of tuples. Therefore, we need to identify on which tuple field the navigation is calculated. The argument `leftJoinPos` points at this tuple field. For completeness, the `rightJoinPos` argument for the input to be navigated is also given. The value of this integer is, however, always "0", because the navigation only produces singleton tuples. Params `axis` and `nodeTest` are the same as before. Arguments `joinMode` and `sorting` capture the discussed features and `baseImplementation` is one of the algorithms introduced above used to implement the navigational evaluation inside the NavTree operator. The algorithm itself is shown in pseudocode representation in Listing 8.1.

For brevity, the parameter variables configuring the algorithm are referenced by letters, such as L, A, M, etc. Besides these variables, we need some further local variables: the input filter F, the output generator G, a buffer queue Q to handle intermediate results, and a tuple generator C for context item provision. These local variables are initialized in the *open* method shown in Listing 8.1a. The context item provider is set as a parameter to the single-node navigational operator (base implementation). The *close* method simply closes the input filter which, in turn, closes the `leftInput`.

The main algorithm is given in the *next* method shown in Listing 8.1c. If buffer queue Q is empty, new tuples have to be generated by the algorithm. Otherwise, the first entry in the buffer queue is removed and returned. To generate new tuples in the buffer queue, the input filter is called to return a new input tuple a (line 3). Depending on the axis A and the join mode M, the input filter guarantees that only tuples are delivered that contribute to the final result. If the input filter returned a NULL value, all input tuples are consumed and the algorithm can terminate. For termination, all remaining tuples in output generator G have to be flushed to the buffer queue (line 5). If the input filter returned a non-NULL tuple, the item at the join position is retrieved (line 8). This item serves as a starting point for the navigation. If this item is an empty sequence and a left outer join has to be computed, the output generator receives an outer join tuple and the rest of the while loop is skipped (lines 10 and 11). Otherwise, the navigation is executed and all retrieved tuples are passed to the output generator (lines 13 to 29). During navigation, when the join mode is *left semi* only one existing node returned by the navigation operator is sufficient (lines 18 to 20). If the navigation returned no tuples and the join mode is left outer, again, the algorithm generates an outer-join tuple (line 26).

Input Filters

The NavTree algorithm assumes to receive a duplicate-free input sequence of XML nodes in document order. If either of these two criteria is not met, the algorithm cannot guarantee them on the output generated. Therefore, the first task of an input filter is to realize these requirements. Note, if the input consists of a non-singleton tuple stream and the join mode is not *right semi*, the requirement of a duplicate-free input stream has to be dropped obviously. Besides these tasks, the input filter can also provide for some optimization, because for certain axes some nodes in the input sequence S may not result in any new information. For an example, consider Figure 8.3 and query `//a/b` (right semi join). In the first round the NavTree algorithm is called for a_1 and returns all descendant b_i elements (i. e., $b_1, \ldots, b_5$). For the second round, elements b_2 to b_4 would be returned by NavTree. However, because the result shall be duplicate-free, these elements are discarded. To avoid producing

Listing 8.1 The NavTree operator methods a) *open*, b) *close*, and c) *next*

Parameter variables:
`Operator leftInput` as L;
`int leftJoinPos` as l;
`int rightJoinPos` as r; // constant "0", because sequence access returns singleton tuples
`Axis axis` as A;
`NodeTest nodeTest` as N;
`JoinMode joinMode` as M; // possible values: *left semi*, *right semi*, *full*, *left outer*
`Sorting sorting` as S; // possible values: left/right or right/left
`AxisStepOperator baseImplementation` as B; // a navigational operator from Section 8.2.1

Local variables:
InputFilter F; // removes redundant input nodes
OutputGenerator G; // generates duplicate-free output in document order
BufferQueue Q; // buffers intermediate results
TupleGenerator C; // provides the context item for the navigation

a) *NavTree.open*:

```
1  begin
2  |  F ← InputFilter(A, L, M, l); // receives the axis, the input, the join mode, and the join position
3  |  G ← OutputGenerator(A, Q, M, S, l); // axis, buffer queue, join mode, sorting, and join position
4  |  Q ← BufferQueue();
5  |  C ← ContextItemProvider(l); // receives join position to project correct item
6  |  B.setContextItemProvider(C); // set context item provider to single node navigation operator
7  end
```

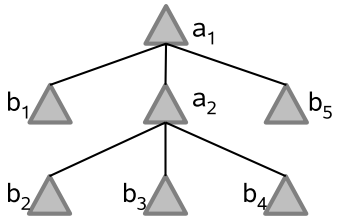
b) *NavTree.close*:

```
1  begin
2  |  F.close();
3  end
```

c) *NavTree.next*:

```
1  begin
2  |  while Q is empty do
3  |  |  Tuple a ← F.next(); // get next filtered tuple from input
4  |  |  if a is NULL then // we're done
5  |  |  |  finalize(G); // write remaining tuples from output generator to buffer queue
6  |  |  |  break;
7  |  |  end
8  |  |  Item i ← a[l]; // get item at the join position
9  |  |  if i is empty sequence and M is left outer join then
10 |  |  |  G.addOutputTuple(a, emptyTupleOfSize(1)); // produce an outer-join tuple
11 |  |  |  continue;
12 |  |  end
13 |  |  C.setContextTuple(a); // prepare the context item provider for the navigation evaluation
14 |  |  B.open(); // open single node navigational operator
15 |  |  Tuples T; // tuple list to capture intermediate result
16 |  |  Tuple b ← B.next();
17 |  |  while b is not NULL do
18 |  |  |  if M is left semi then // in case of a left semi join, the existence of one node is sufficient
19 |  |  |  |  break;
20 |  |  |  end
21 |  |  |  T ← T + b;
22 |  |  |  Tuple b ← B.next();
23 |  |  end
24 |  |  B.close();
25 |  |  if T is empty and M is left outer join then
26 |  |  |  G.addOutputTuple(a, emptyTupleOfSize(1)); // produce an outer-join tuple
27 |  |  else
28 |  |  |  G.addAllOutputTuples(T);
29 |  |  end
30 |  end
31 |  if Q is empty then
32 |  |  return NULL; // no more nodes found
33 |  end
34 |  return Q.first();
35 end
```

Figure 8.3 A filter and output generation example



them in the first place, the input filter simply skips input node a_2. For all other axes and join modes, similar techniques can be applied, which we, however, do not present here. We have implemented input filters for all 12 XPath axes. Interestingly, the skipping algorithms for axes *parent*, *ancestor*, and *following-sibling* are the same.

Output Generators

An output generator receives all tuples generated by the base NavTree algorithm (see Figure 8.1c). Its task is to produce a duplicate-free output in document order, or, where an outer join or a full join is required, the correct *left/right* or *right/left* output ordering. Again, we motivate output generators by a simple example. Consider query `//a/b` on the document shown in Figure 8.3. Because the child axis between a and b has to be computed, the input filter delivers a_1 in the first round, and a_2 in the second round. Node a_1 results in the two children b_1 and b_5. However, these nodes may not be placed into buffer queue Q directly, because the nodes produced for a_2 have to lie "in between". Therefore, the output generator buffers (and orders) the received nodes until no more nodes can be generated. Output generation depends on the axis, the join mode, and the desired output order. We have implemented output generators for all 12 XPath axes, where the following three axis groups share one algorithm: [*child*, *preceding-sibling*, *following-sibling*, *attribute*], [*parent*, *ancestor*], and [*following*, *preceding*].

At this point, we finish the discussion on the NavTree algorithm. In summary, the algorithm 1) computes a structural join (left semi, right semi, full, and left outer) on 2) all XPath axes, 3) is able to skip unnecessary input nodes, and 4) can establish a duplicate-free, sorted output stream. The algorithm can operate on the document or on the element index (depending on the base implementation chosen). Therefore, it provides a fallback solution in case one of the algorithms introduced in the following cannot be applied. Note, the predecessor of the NavTree algorithm (with reduced functionality) has been proposed by [Mathis 06b].

8.3 The Structural Join Operator: Extended StackTree

As we have seen, navigational PPOs require some input node(s), which serve(s) as a starting point for a navigation (or scan) operation over the document or the element index. In contrast, join-based PPOs do not directly access the document. They operate on *two or more* input streams and are capable of finding path matches in these input streams. Where the input streams come from does not matter, e. g., they could be the result of some document scan or they could be the result of other

operators.

In this and the next section, we will introduce two join-based PPOs, namely the *physical structural join* operator and the *physical twig join operator*. In the literature, both operators have gained significant attention, resulting in quite some related work (see Section 8.7). The structural join operator implemented in XTC is a merge-join algorithm, which is very close to the original StackTree operator proposed by [Al-khalifa 02]. Therefore, we will not provide you with the details, but just sketch what the operator accomplishes and how it can be employed for plan generation. On the other hand, in case of the holistic twig join algorithm, we will present some more details, because we substantially extended the algorithms expressiveness compared to the standard approaches in the literature. Let us start with the structural join operator.

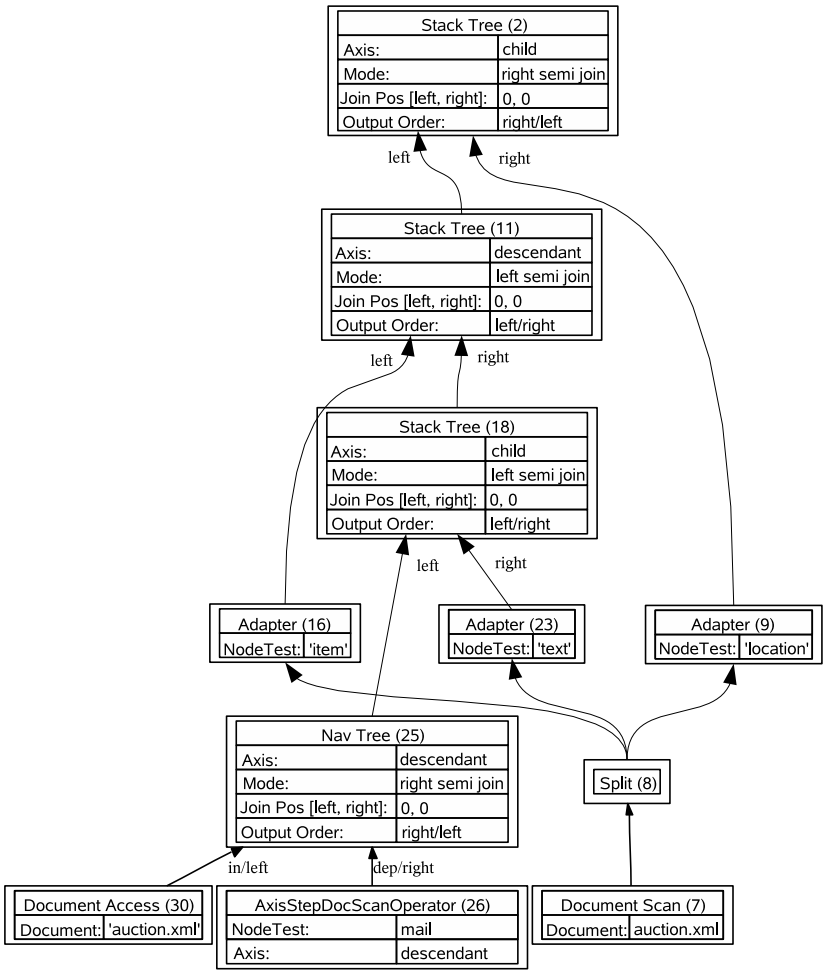
We extended the StackTree operator by some features required for the integration of the algorithm into our algebra. In Section 8.2.2, we have already motivated the need for a structural join implementation supporting semi joins, full joins, and outer joins. The join implementation should, furthermore, return the result in distinct document order (in case of a semi join), or in *left/right* and *right/left* output ordering (in case of a full join or an outer join). The StackTree algorithm also has to provide this functionality. Therefore, we extended the basic algorithm proposed by [Al-khalifa 02]. The resulting extended StackTree operator has the following signature:

```
StackTreeOperator (Operator leftInput,
                   Operator rightInput,
                   int leftJoinPos,
                   int rightJoinPos,
                   Axis axis,
                   JoinMode joinMode,
                   Sorting sorting);
```

In contrast to NavTree, which receives one input operator, StackTree receives two (`leftInput` and `rightInput`). Because these input operators can deliver non-singleton tuple streams, a join position for each input is required (`leftJoinPos` and `rightJoinPos`). The *axis* parameter can have the following values: *child*, *attribute*, *descendant*, *parent*, *ancestor* (and the *-or-self* variations of *descendant* and *ancestor*). Note, the reverse variants of the algorithms are implemented by exploiting join commutativity. For example, matching the *ancestor* axis between input streams S_A and S_B can be implemented by matching the *descendant* axis between S_B and S_A. The input operators are simply swapped (and the other parameters are reconfigured appropriately).

As an example, let us consider query `doc("auction.xml")//mail[text]/ancestor::item/location`. The plan for this query is shown in Figure 8.4. The plan generator mapped the first structural join to a NavTree operator (because a real structural join between the singleton virtual root node and the descendant *mail* does not make much sense). NavTree is then followed by three StackTree operators. A particularity we have not yet introduced, is the way how the input to these StackTree operators is generated, namely by a *split up* document scan. Similar to the logical split operator, the physical implementation splits the incoming input streams. In our example, these streams are processed by three adapters, each of which applies a node test. Note, the query could also have been implemented by three separate document scans (later on, we will see how index structures can deliver this input). The first stack tree algorithm computes the join between *mail*

Figure 8.4 A StackTree example



and *text*. The second join was generated for the *ancestor* axis. As you can observe, the plan generator replaced the axis by *descendant* and swapped the input operators to employ a stack tree operator (note, without this mapping, the plan generator would have to have used a NavTree operator). Finally, the last StackTree operator calculates the join between *item* and *location*.

8.4 The Holistic Twig Join Operator: Extended TwigOpt

The twig join operator is an extension of the structural join. In contrast to structural joins, the algorithm can consume more than two input streams, in which it matches complex branching path patterns (also known as twigs). Our notion of a twig has been introduced at a logical level in Section 5.8. In summary, our logical twig operator defines the following concepts:

1. path patterns supporting axes *child*, *descendant*, and *attribute*,
2. logical *and* and *or* conjunctions,
3. optional subtree patterns (i. e., optional edges),

4. projection,
5. positional predicates,
6. output filters,
7. embedded output expressions, and
8. grouping.

So far, no algorithm has been published in the literature that can capture the expressiveness of our logical twigs. However, such an algorithm is desirable, because the higher its expressiveness, the more operations can be embedded into the twig algorithm, thus, resulting in a smaller number of operators in the final plan. As we will see below, the twig algorithm keeps intermediate results compactly encoded on a set of stacks. Therefore, evaluating as many operations as possible on this encoding can avoid the materialization of intermediate results in some cases. In this section, we will develop an algorithm supporting the above features. Twig algorithms have gained quite a lot of attention in the research community, resulting in a large body of work on the topic (cf. the related-work section). To avoid re-inventing the wheel, we picked a promising approach as the basis for our implementation, namely, the TwigOpt algorithm proposed by [Fontoura 05]. In the following, we will introduce the algorithm and our extension with the help of a brief example. Then we will present implementation details.

8.4.1 Extended TwigOpt by Example

Figure 8.5 presents various stages of a sample extended TwigOpt run for query:

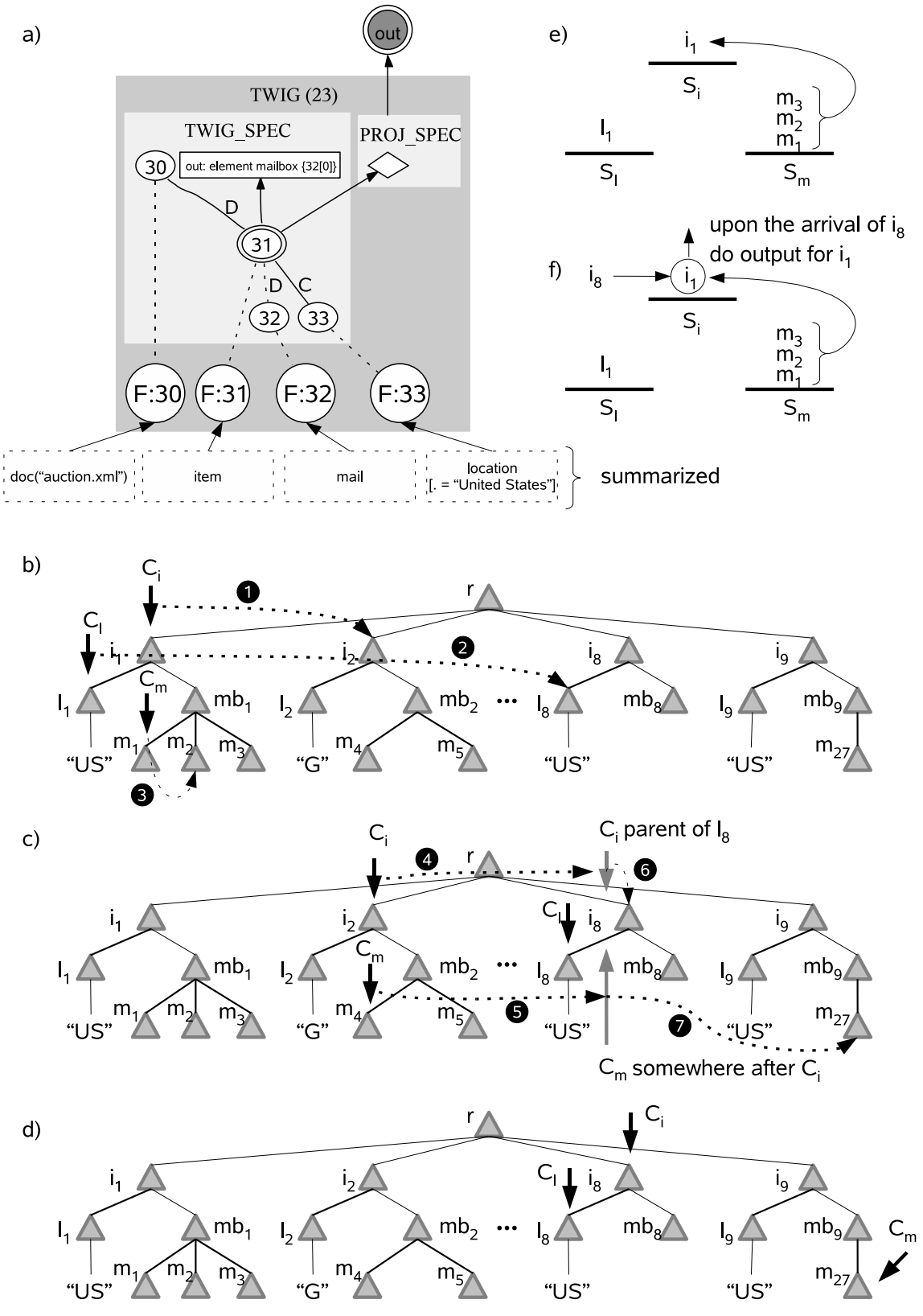
```
for $i in doc("auction.xml")//item
where $i/location = "United States"
return <mailbox>{$i//mail}</mailbox>
```

Unnesting and twig discovery generate the logical XQGM instance shown in Figure 8.5a: The twig has four input sequences (displayed in a summarized way to save space), an optional descendant edge, a grouping node, and an output expression (with a projection). Note, the content predicate has been pushed down to an access operator. Therefore, this operator delivers only *location* nodes with content "United States". Figure 8.5b shows the document, on which our sample query is executed. To keep the presentation simple, we abbreviated *item* by i, *location* by l, "United States" by "US", and so on. Basically, the document consists of nine structurally differing subtrees, of which only four are presented (i_1, i_2, i_8, and i_9). We assume, that only i_1, i_8, and i_9 fulfill the content predicate.

Cursors

The input of the TwigOpt operator is delivered by so-called *cursors* [Fontoura 05]. Essentially, these cursors are iterators (providing some skipping functionality), which deliver XML nodes in document order. In the physical mapping of the logical twig from Figure 8.5a, the plan generator omits the twig root node, if the node delivers the virtual document root. Therefore, we only require three cursors: $C_i = \{\underline{i_1}, i_2, \ldots, i_9\}$, $C_l = \{\underline{l_1}, l_2, \ldots, l_9\}$, and $C_m = \{\underline{m_1}, m_2, \ldots, m_{27}\}$. Initially, the cursor pointers are positioned on the first node of each cursor list (i. e., at the underlined nodes). In our sample figure, the cursors are represented by black arrows. Besides the cursors, the TwigOpt algorithm maintains a set of stacks, one for each twig node. The stacks will be used to encode intermediate results.

Figure 8.5 An example of the extended TwigOpt operator: a) a sample twig query, b) initial cursor position and movements, c) virtual cursor movements, d) real cursor positions, e) stack configuration after containing first match, and f) situation provoking output generation



Solution Extensions

While the algorithm runs, it checks whether the current cursor configuration (i. e., the current cursor pointers) form a match w. r. t. the twig specification. In our example, the initial cursor configuration forms a so-called *solution extension* (see Figure 8.5b). Informally, a solution extension for a twig node n having cursor C_n is found, if

1. the cursor positions of n's child twig nodes are *descendants/children* (depending on the axis) of the cursor position of C_n,
2. cursor position of C_n is a *descendant/child* of the top-most element in the stack of n's parent node, and
3. the children have solution extensions [Fontoura 05].

If any of the information required to check one of these criteria does not exist (e. g., if a twig node does not have any child twig nodes), the criterion is true. In our example, the first criterion is matched for twig node 31, because l_1 (of cursor C_l) is a child of i_1 and m_1 (of cursor C_m) is a descendant of i_1. The second criterion is true, because twig node 31 has no parent twig node. The third criterion results in a recursive check on the two children, which is also evaluated to *true*.

In TwigOpt, the order in which solution extensions are found on a twig depends on the current cursor positions. The twig node, whose current cursor position points to the smallest XML node (in document order), is selected for a solution extension check. This assures that no possible match is omitted. When a solution extension has been found for a twig node n, the XML node at the current cursor position is pushed onto the corresponding stack S_n. Then, cursor C_n can be advanced to the next position. These cursor movements are represented in Figure 8.5b by dotted arrows. First, a solution extension for C_i is found, i_1 is pushed to stack S_i, and C_i is advanced to i_2 (movement 1). Then, l_1 is pushed and C_l is advanced (movement 2), and then m_1/C_m (movement 3). Because in this situation, C_m points to the smallest XML node among all cursors, twig node 32 is checked for a solution extension. This check is evaluated to *true*, because criterion 2 from above is fulfilled: parent stack S_i contains i_1 which is an ancestor of m_2. After m_2 has been pushed to S_m, node m_3 also matches, resulting in the stack configuration represented in Figure 8.5e. To implement grouping, the algorithm has to do some "bookkeeping": because m_1, m_2, and m_3 have been matched as descendants of i_1, they are marked as a *group below* i_1 (note, these *group markers* are implemented by a secondary stack per twig node, which grows and shrinks with the primary stacks).

Cursor Movements (with Skipping)

After the stack configuration in Figure 8.5e has been established, the cursors reside at the positions shown in Figure 8.5c (C_l is located at l_8, because previous subtrees did not contain a *location* element with content "United States"). Obviously, the cursor positions do not match the twig structure and, therefore, twig node 31 has no solution extension. What follows is an adjustment of the cursors such that a solution extension can become possible [Fontoura 05]. For that, TwigOpt tries to skip as many nodes in its cursors as possible. In our example, we can see that C_l resides on l_8. Therefore, because l_8 has to be a *descendant* of an *item* node, the next position of C_i has to be somewhere among the ancestors of l_8. Therefore, the cursor is advanced *virtually* to such a position (represented by the grey arrow in Figure

8.5c; movement 4). Such a virtual cursor movement does not directly lead to a physical (i. e., real) cursor movement. Now, for the next cursor C_m, we know that it has to point to the parent node of C_i. From the virtual cursor position of C_i, we can calculate another virtual cursor position for C_m (movement 5). Now we have two virtual cursor positions and one physical one. In this state, the algorithm has to check the node of the cursor with the smallest position for a solution extension. Because this is not possible for a virtual position, the corresponding cursor is moved physically. In our example, this would be cursor C_i, which is set to i_8 (movement 6). Using this movement, we have skipped all *item* nodes i_3 to i_7 (not shown in our figure). To check a solution extension on twig node 32, its cursor has to be made real, too. The next (real) position after the virtual one is at m_{27} (thus, nodes m_5 to m_{26} have been skipped by movement 7). The resulting situation is depicted in Figure 8.5d.

Output Generation

While the solution extension concept and the cursor movement strategy in our TwigOpt implementation are nearly the same as in the original algorithm, output generation is different (note, [Fontoura 05] did not provide any details on how output is generated). In the situation presented in Figure 8.5d, we can again find a solution extension for twig node 31, although C_m points to a node not contained in the subtree below C_i. The reason why we, nevertheless, find a solution extension is the optional edge between twig nodes 31 and 32. The optional edge states that *mail* elements do not have to exist below *item* elements. Therefore, we find a match here. In our informal solution extension definition above, we have omitted the semantics of optional subtree edges (among other things) to keep the discussion simple. Because a solution extension has been found on i_8, the node has to be pushed onto S_i. Obviously, i_8 is no descendant of the bottom-most node in the stack (i. e., of i_1). Therefore, we know that the subtree below i_1 has been completely processed and that no more matches can be found in that subtree. As a result, we can produce some output for the subtree below i_1, clear all stacks, and push i_8 to S_i (see Figure 8.5f). Note, if i_8 would be a descendant of i_1, it would have been pushed on S_i and no output would have been generated (for now).

Output generation is a recursive process over the twig structure, which is similar to finding a solution extension. If you reconsider all the output options we have introduced for the twig operator in Section 5.8 (i. e., in Figures 5.12 and 5.13 on Pages 131 and 132), you immediately see that all information for this task is there: From the stack configuration we know that i_1 is a parent of l_1 and that i_1 has three descendant *mail* nodes m_1 to m_3. Using this knowledge, we can compute projections, positional predicates, output filters, output expressions, and grouping.

For a twig node n, output is produced as follows:

1. The output process iterates bottom to top over all stack elements of S_n in the group requested from n's parent node. If n is the root node, the output process iterates over *all* stack elements from bottom to top.

2. For a node in such an iteration step, the groups of their child twig nodes are requested (recursive call). Note, for a child c of n this only happens, if c or at least one twig node in the subtree below c produces some output or has further semantics (e. g., an output filter, some positional predicate, etc.). Otherwise, child c is only a structural predicate. Because the matching process has already evalu-

ated all structural predicates, c can be skipped. Each group returned for a child c is processed as follows:

(a) If c contains a positional predicate, this predicate is applied to the group returned for c (thus, removing all group members not fulfilling the positional predicate).

(b) If n contains an output filter for c, this filter is applied to the group returned for c. If the filter evaluates to *false* and the edge to child c is *not* optional, the complete match is discarded. If the filter evaluates to *false* and the edge of the child *is* optional, the group is replaced by an empty sequence. Otherwise, the group is kept as it is.

(c) If n contains an output expression for twig child c, the output expression is applied to the group returned for c. The group value is then replaced by the value returned from the output expression.

3. If n is a grouping node, an intermediate result tuple is generated, that contains the current stack element of S_n and all groups (or values generated by output expressions on the returned groups).

4. If n is *not* a grouping node, the Cartesian product is computed on the current stack element of S_n and all groups (or values generated by output expressions on the returned groups).

5. If n has an output expression or if n does not produce any output (due to projection) and if n is does not have any further semantics (e. g., referenced in an output expression, output filter, etc. in a twig node above), the item generated for n in the intermediate result is removed (projected).

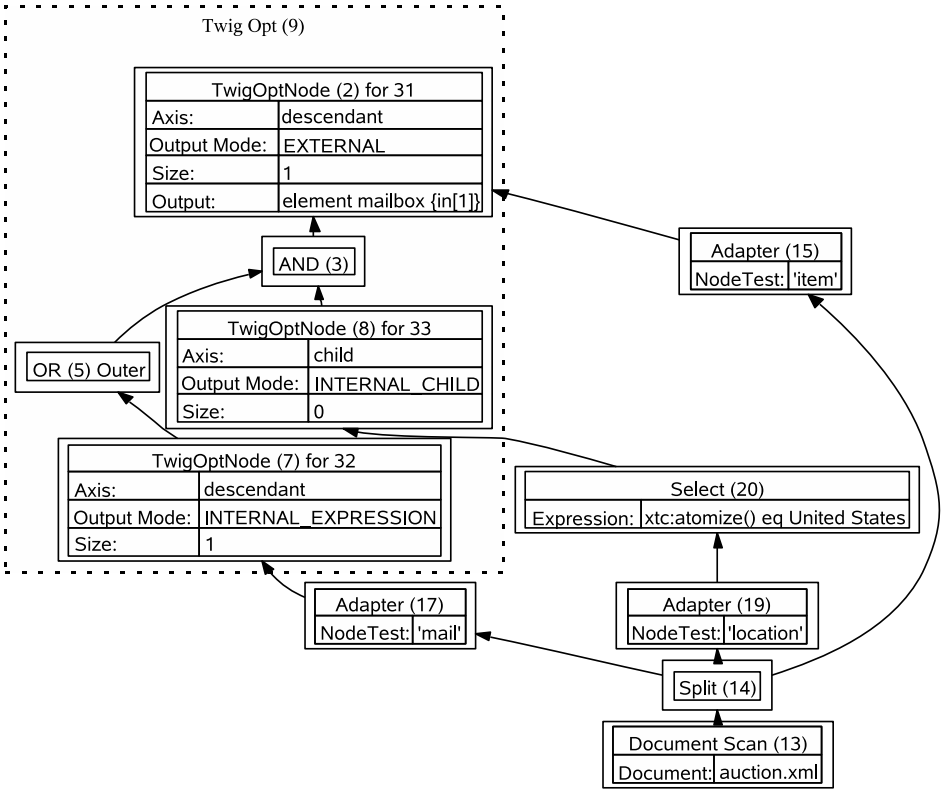
Let us consider the output generation process in our example: The output process iterates over all elements in S_i (point 1 above), which is just one node, namely i_1. For i_1, the groups of the twig children are requested (point 2). Because twig node 33 does not contribute to the output and is, therfore, a pure structure match, this twig node is skipped. The recursive call on S_m returns group <m_1, m_2, m_3> in a sequence. Twig node 31 contains an output expression for twig node 32 (point 2c). Therefore, the group is replaced by the result of the evaluation of the output expression on the delivered group: <mailbox>{m_1, m_2, m_3}</mailbox>. Because twig node 31 is a grouping node, the join with the current value from S_i returns tuple [i_1, <mailbox>{m_1, m_2, m_3}</mailbox>] (point 3). Finally, because node 31 contains an output expression, the information generated for that node is projected out (point 5), resulting in output value <mailbox>{m_1, m_2, m_3}</mailbox>. Note, the implementation of the algorithm does not add i_1 at first, before it directly removes it again (as described here). It can detect such situations and directly delivers the correct result. The deviation from the algorithm here was made to keep the discussion simple.

After this overview over the algorithm, we present some implementation details. For the following discussion, we group the topics around the extended TwigOpt operator into *twig mapping*, *cursors*, the *matching process*, and into *output generation*.

8.4.2 Twig Mapping

During plan generation, a logical XQGM twig is mapped onto the TwigOpt operator. Thereby, also the twig specification is mapped to a physical representation, which is slightly different from the logical one. The TwigOptNode class shown in

Figure 8.6 The physical plan generated for the sample query in Figure 8.5a



Listing 8.2 captures this physical representation. During twig mapping, the plan generator constructs a tree of TwigOptNodes and initializes the member variables of each twig node according to the logical twig representation. Besides member variables, the TwigOptNode class contains some methods. These methods will help to implement the TwigOpt algorithm, as we will see in the next section. However, at first, we want to summarize the twig mapping process and the meaning of the member variables.

Twig Root Handling

As we have seen during the discussion of our example, the twig operator blocks until the bottom-most node at the stack of the twig root node is no ancestor of a node to be pushed onto this stack. To avoid collecting *all* matches on the stacks, before generating output, we do the following: When 1) the twig root node has exactly one twig child, when 2) this child is connected via a *descendant* axis, and when 3) the twig root node is not required for output, we do not map the twig root node in the physical representation. In this case, the twig algorithm will find all occurrences of the twig in the document. If criteria 1) and 3) are true, but a *child* edge connects the twig child, then, the root node is also removed. However, the plan generator presets a special `DocRootNodeCursor` (see below) for the child TwigOptNode. This cursor delivers only the root element of the document. Thus, the restriction to match the *child* axis is assured. Figure 8.6 contains the physical plan for the query represented in Figure 8.5a. Because the three criteria from above were matched, the physical twig consists of only three path nodes.

Listing 8.2 The TwigOptNode class

```
class TwigOptNode {
  // member variables set by the plan generator during twig mapping
  Axis           axis;          // the incoming axis
  Cursor         cursor;        // the input cursor
  Stack          nodeStack;     // stack containing DeweyIDs
  Stack          positionStack; // stack containing IntegerLists
  TwigOptNodes   children;      // a list of child nodes
  PalExpression  expr;          // an output expression
  PalExpressions filters;       // a list of filer expressions
  CmpMethod      cmpMethod;     // allowed: '<', '<=', '=', '!=', etc.
  int            cmpValue;      // for positional predicates
  boolean        isGrouping;    // true, iff this node groups its input lists
  NodeType       type;          // allowed: 'and', 'or', 'outer or', and 'path'
  OutputMode     mode;          // 'none', 'internal child', 'internal filter', etc.

  /* find the node with the smallest/largest cursor position */
  TwigOptNode minCursorPathNode();
  TwigOptNode maxCursorPathNode();

  /* finding a solution extension */
  boolean checkSolutionExtension(); // true, iff this node has an extension
  boolean constraints();            // checks solution extension constraints
  boolean stackContains(Cursor c);  // true, iff nodeStack contains ancestor of c
  boolean allCursorsAreReal();      // true, iff all cursors are real

  /* cursor movement */
  void   moveCursors();             // virtually/physically moves cursors
  void   moveCursorsBottomUp();     // adjusts parent cursors from child positions
  void   moveCursorsTopDown();      // adjusts child cursors from parent positions
  Cursor chooseBestVirtualCursor(); // cursor with most skipping potential

  /* output generation */
  // generates all output and pushes current cursor's node onto the stack
  void outputAndPush(Queue queue);

  // generates output for the root stack
  void outputRootStack(Queue queue);

  // generates output for a stack (starting at startPosition; results below entry)
  Item outputOneStack(int startPosition, DeweyID entry);

  // generates output for one stack element
  Tuple outputOneEntry(int stackPosition);

  // collects the generated results from each twig node child
  Items processChildren(TwigOptNode parent, DeweyID entry, IntegerList positions);

  // applies output filters/expressions, unnesting, and projection
  Tuple processTuple(Tuple tuple);

  // calculates the Cartesian product in the sequences in the tuple
  Tuple unnest(Tuple tuple);

  // removes unnecessary tuple fields
  Tuple project(Tuple tuple);

  // removes all elements from nodeStack and positionStack
  void cleanStacks();

  /* positional predicate handling */
  boolean preCheckPositionalPredicate(int position); // pre-emptive check
  boolean checkPositionalPredicate(int position);    // complete check
}
```

Boolean Twig Node Handling

In our logical twig representation, when a twig node has multiple children, we assume implicit *and* semantics. For the physical twig mapping, we generate an explicit *and* node. Logical *or* nodes are simply mapped onto physical ones. In our example of Figure 8.5a, you can also find the generated *and* node. The type of a twig node is stored in the `type` variable of the TwigOptNode class.

Optional Edge Handling and Grouping Embedding

An optional edge an be expressed by a Boolean *or* node: Let e be an optional subtree, then (informally) "e or *empty sequence*" returns the result of e or, if e does not return a result, the empty sequence. We express these semantics by an *outer or* node, to which the optional subtree is attached. See Figure 8.6 for the mapping of the optional edge of the query in Figure 8.5a. A twig node is declared to be *outer or* by setting the `type` variable appropriately.

If a logical twig node is grouping (i. e., if it has a double circle), then the corresponding physical twig node is also grouping (signaled by the Boolean `isGrouping` flag).

Expression Embedding

During the discussion of the example above, we have seen how output is generated. Essentially, intermediate tuples are generated on which output/filter expressions are evaluated. In the logical representation, these expressions reference tuple variables and tuple positions. For example, in the output expression of the twig in Figure 8.5a, such a reference has the following form: `32[0]`. This notation means that the output expression receives the results generated for twig node 32 at position 0. During physical twig mapping, we remove these tuple variable references (because the physical twig has no tuple variables). To do so, we replace them by pure tuple access positions. In the physical representation of our example, the corresponding access position to `32[0]` is `in[1]`. These access positions are calculated by analyzing the twig structure. The mapping is quite straightforward and not formally introduced here. Output expressions are introduced in the TwigOptNode class by member variable `expr`. Note, similar to the LALExpression concept (see Section 4.2.4), an output expression is a PALExpression, i. e., a physical algebra expression (with the ability to be evaluated on an input tuple). The mapping from LALExpressions to PALExpressions is done by the plan generator and is straightforward, because they essentially translate to XQuery expressions (e. g., algorithmic expressions, comparisons, etc.).

Output Modes

For each physical twig node, the plan generator can define a certain output mode (in the `mode` field). In our example above, we have seen that output generation is a recursive process which, for a certain twig node n, pulls data from n's children and processes on this data. Now, the output mode of a node n specifies for which purpose n generates data. There are five possibilities: *none*, *internal child*, *internal filter*, *internal expression*, and *external*. Output mode *none* is chosen for n, when the cursor of n delivers information that is only necessary for matching the twig structure, but for nothing else. This is the case, when n's incoming edge is *descendant*, when the information generated for n is not used in any predicate/filter, and when n is not

referenced in the projection specification.

Note, despite what was explained in our introductory example, the TwigOpt algorithm implemented here can only match the *descendant* axis. The *child* axis has to be checked, when the output is generated. We swept this problem under the rug, because we wanted to keep the introduction example simple. When the child axis has to be checked, the child XML nodes have to be passed to the parent twig node, where the check is executed. To express this, we set the output mode of the child twig node to *internal child*.

In the following, "internal" means that the output generated for this node will be projected out by some ancestor twig node. In case, a physical twig node is referenced by some embedded expression or filter, its output mode is set to *internal expression* or *internal filter*. Finally, when a physical twig node is referenced in the projection specification, its output mode is set to *external*, because the data it produces will be passed on for further processing.

In our example depicted in Figure 8.6, you can observe three of the above sketched output modes. Twig node 7 is of mode *internal expression*, because its output will be used in the output expression of twig node 2. Twig node 8 has *internal child* as output mode, because the *child* axis has to be verified by the matching algorithm. Twig node 2 has output mode *external*, because this node is referenced in the projection specification.

Note, if a node n has an incoming *child* axis and if this node is also referenced by the projection specification, the *external* output mode will override the *internal child* output mode for n. In general, the output modes have the following priorities (highest to lowest): *external*, *internal expression*, *internal filter*, *internal child*, and *none*. Essentially, the output modes are used to infer the correct tuple access positions during expression/filter embedding (as discussed before).

Positional Predicates

Positional predicates are embedded by specifying a comparison value (`cmpValue`) and a comparison method (`cmpMethod`). For example, for predicate `item[position() < 5]`, the comparison method would be "`<`" and the comparison value would be "`5`". This information is directly attached to the physical twig node on which the predicate shall be applied (in our example, the twig node for the `item` step). The *last* function (e. g., `item[last()]`) receives a special treatment: Essentially, the last function selects the last item from a sequence. We implemented this semantics with the help of the new *xtc-fn:last()* function. For the physical embedding, the *xtc-fn:last()* function is set as an output expression to physical twig node generated for the *parent* of the logical twig node with the `last()` predicate. If no such parent exists, the function is given to a projection operator, which executes it on the complete twig result. Note, positional predicates can be "preemptive". For example, in case of `item[position() < 5]`, we can stop generating the output of *item* nodes, when four items have been returned.

Cursors

Cursors provide the input for physical twig nodes. We will introduce cursors in the following. As we will see, the cursor of a physical twig node can be implemented by an operator of the physical algebra or by special index access cursors. Therefore, the

plan generator can choose from different cursor alternatives. When the plan generator creates the physical twig structure during twig mapping, the cursors remain undefined (with the exception of *virtual* cursors and the *document root node cursor*; see below). In the second plan generation phase, when the plan is assembled based on the generated alternatives, the cursors are set appropriately. However, during twig mapping, we also allow to pre-define a cursor for a twig node by directly setting the `cursor` variable (for example, to prefer an index access cursor). These pre-defined cursors will not be overwritten in the second plan generation phase.

Further Internal Data Structures

Every physical twig node has a `nodeStack` and a `positionStack`. The first stack encodes the intermediate result, while the second stack keeps the group information, as explained in the example above. Furthermore, every twig node contains its children as a list of TwigOptNodes in the `children` variable. Sometimes, access to the parent twig node is necessary. We omitted the necessary variable in Listing 8.2 for simplicity. Besides parents and children, we can also retrieve the set of *path children* which are the nearest descendants of type *path* in each path. Accordingly, the *path parent* is the nearest ancestor of type *path*. For example, in Figure 8.6 on Page 236, the path children of node 2 are nodes 7 and 8 (Boolean nodes have been omitted). Accordingly, the path parent of 7 is 2. All member variables are accessible by appropriate *get* and *set* methods (not shown).

The TwigOptOperator

After the twig has been converted into its physical form, the projection specification of the logical twig operator is mapped onto a physical projection operator, similar to the mapping of an ordinary XQGM select. The signature of the TwigOpt operator is fairly simple:

```
TwigOptOperator (TwigOptNode rootNode);
```

With the physical twig mapping in mind, we can now discuss the extended TwigOpt algorithm itself, starting with the cursor alternatives.

8.4.3 TwigOpt Cursors

Listing 8.3 presents the common interface of all cursors implemented. A cursor has a current position (i. e., `currentID`) and two Boolean flags signaling whether the cursor is consumed or virtual. Important for our cursor discussion are the following four methods (the remaining ones will be discussed in the next section):

- *compareTo* compares the current cursor position `currentID` with the current position of the given cursor. Significant for this comparison is the document order. Note, on DeweyIDs, this comparison can be efficiently computed by a "lexicographical" comparison.
- *contains* checks whether the current cursor position of the given cursor is a descendant of `currentID`. This method is required to check for the above introduced *solution extension*. Note furthermore, *contains* and *compareTo* are generic methods implemented in the abstract class given in Listing 8.3.
- *setToFirst* initializes the cursor by setting its pointer `currentID` to the first cursor position.

Listing 8.3 The abstract Cursor class

```
abstract class Cursor {

  DeweyID currentID;  // the current cursor position
  boolean isConsumed; // true, iff the complete input has been processed
  boolean isVirtual;  // true, iff the cursor points to a computed position

  // compares with current ID of other cursor
  int compareTo(Cursor other){/* ... */};

  // returns true, iff the current ID of the other cursor is descendant of currentID
  boolean contains(Cursor other){/* ... */};

  // initializes the cursor with the first position
  abstract void setToFirst();

  // implements the physical cursor move
  abstract void forwardTo(DeweyID position);

  // for finding a solution extension
  boolean containsAllCursorsOf(TwigOptNodes nodes);
  boolean containsSomeCursorsOf(TwigOptNodes nodes);

  // for virtual cursor position computation
  DeweyID getCurrentPlusOne();
  DeweyID getPositionBelowLCA(Cursor cursor);
}
```

- *forwardTo* implements a physical cursor move. If the cursor is virtual, the cursor "searches" for the next DeweyID in the input, which is *larger or equal* ($\geq$) to the given ID (`position`). If the cursor is not virtual, the next DeweyID has to be really larger ($>$).

For the integration of TwigOpt into XTC and into our physical XML algebra, we have implemented six different types of cursors: two *jump* cursors, that operate on the document (`DocJumpCursor`) or on the element index (`ElIdxJumpCursor`); one virtual cursor (`VirtualCursor`) required for the implementation of *and*/*or* predicates; one cursor for the provision of the document root (`DocumentRootNodeCursor`); one cursor for the integration of path indexes (`ATBCursor`); and one *operator* cursor. The last cursor was implemented by extending the `Operator` interface with the *setBegin* and *forwardTo* methods. The `Operator` interface is, in turn, implemented by all physical algebra operators. Let us briefly discuss these cursors.

Jumping Cursors

A jumping cursor can be parameterized by an optional *selection expression*, by a *node test* (in case of the `DocJumpCursor`), and by a *node name* (in case of the `ElIdxJumpCursor`). The plan generator can use these cursors for the implementation of sequence access operators with optional selection predicates (e. g., for the input operators of twig node 31, 32, and 33 in Figure 8.5a). Because the `DocJumpCursor` receives a node test, it is more general than the `ElIdxJumpCursor`. The latter can only be used by the plan generator, when an element index is available and when the sequence access operator has a simple name test.

Upon *setToFirst*, both cursors open an index—either the document index or the element index. The open position is the first element that fulfills the node test (or name test) and the optional predicate. For the implementation of *forwardTo*, both jumping cursors open the index at the position of the given DeweyID (or before). Starting at this position, they scan until they find an XML node fulfilling the node test (or name test) and the optional predicate.

Jumping cursors were proposed by [Fontoura 05] in the original paper on the TwigOpt algorithm. As we have seen, jumping allows us to skip large parts of the document and only a minimum number of nodes is accessed by the algorithm (thus the suffix "Opt" for *optimal*). However, what [Fontoura 05] conceal is the problem that jumping results in multiple index open/close operations. Each of these operations contains some overhead which cannot be neglected for low selectivity queries. We will consider this problem in Chapter 9, where you can find our experimental results.

The Virtual Cursor

The virtual cursor is required for the implementation of *and/or* twig nodes (see Figure 5.12c on Page 131). During plan generation, the logical twig representation in the XQGM twig operator is mapped onto a physical representation (which is quite similar and which will be introduced in the following). During this mapping, twig nodes that are no *and/or* nodes (i. e., *path nodes*) but that have multiple children are split up into an ordinary twig node and into an *and* twig node. Therefore, the Boolean *and/or* nodes are always explicit in the physical twig representation.

The TwigOpt algorithm requires that every physical twig node has a cursor. However, Boolean twig nodes have no input. Therefore, they receive an artificial cursor in form of the `VirtualCursor`. The cursor has almost no semantics. Upon *setToFirst*, the Boolean `isVirtual` flag is set to *true* (i. e., Boolean nodes are always virtual). In case of *forwardTo*, the given DeweyID is simply stored in the `currentID` variable. Note, this cursor is directly pre-set, when the twig is mapped onto its physical representation.

The Document Root Node Cursor

In Figure 8.5a, you see that the XQGM twig contains a node receiving the virtual document root (namely, node 30). During twig mapping, we try to get rid of this node. In our example, the physical twig for the one presented in the figure only consists of three nodes (as already stated above). This is possible because the logical twig node below the root has an incoming *descendant* axis. If the axis is *child*, we can also remove the root node. However, we have to generate the correct input. We do so by providing a special `DocRootNodeCursor`. Upon *setToFirst*, the cursor loads root node (not the virtual root) and checks a node test (which is given to the `DocRootNodeCursor` as a parameter). When the node test evaluates to *false*, the cursor is immediately consumed (i. e., `isConsumed` is *true*). Otherwise, the *forwardTo* method can be called. Because the cursor contains only one element, the cursor is then also immediately consumed. Note, this cursor is also directly pre-set, when the twig is mapped onto its physical representation.

The ATBCursor

The prefix "ATB" stands for *ancestor tuple builder*. Ancestor tuple builders are required to embed path and CAS indexes (introduced in Chapter 7) into the TwigOpt algorithm. Let us briefly consider how a CAS index on `//item/location` could be utilized for the implementation of the example in Figure 8.5a. The input of twig node 33 has to deliver all *location* elements with content "United States". From the twig specification, we can infer that the *location* elements have to appear as children of *item* nodes. Therefore, we could implement the input of twig node 33 by a scan over the `//item/location` CAS index. However, with the CAS index, we can achieve more than the simple implementation of one access operator. As explained in Section 7.3.6, a CAS index can compute *all* inner elements. In our example, the index can, therefore, also compute *item* nodes. Thus, the input of twig node 31 becomes dispensable. Additionally, twig node 31 "sees" only those *item*s that have a *location* element with content "United States" as a child.

To provide inner nodes (like the *item* nodes from our example), an `ATBCursor` is required. This cursor receives the input of a path or CAS index scan and provides inner elements to the TwigOpt algorithm. In Section 8.5, we will see how this is accomplished.

Operator Cursors

Any operator delivering a sequence of singleton tuples containing XML nodes can be used as input to the TwigOpt operator. Therefore, we implement the *cursor* interface shown in Listing 8.3 for ordinary operators of the physical algebra. This way, we achieve maximum flexibility. The implementation of *setToFirst* and *forwardTo* is quite straightforward. Method *setToFirst* calls *open* on the operator and retrieves the first (singleton) tuple by calling *next*. The XML node of this tuple provides the `currentID`. If the operator did not return any result, the cursor is immediately consumed. Method *forwardTo* is implemented by iterating over the tuples delivered by calls to the *next* method, until a tuple is found, whose node's DeweyID is larger (or equal) to the given one.

8.4.4 TwigOpt Matching

The matching algorithm can now be defined at the cursor interface. Note, [Fontoura 05] sketched the TwigOpt algorithm in a simple manner, thereby neglecting certain corner cases. As a result, the algorithm did not work correctly. In the following, we will not skip these corner cases, but show the complete algorithm (although the matching and cursor movement parts are quite close to the original; cf. [Fontoura 05]). To keep the discussion simple, we skip the *open* and *close* methods. Basically, *open*/*close* is recursively called on all physical operators contained in the twig cursors. Additionally, during *open*, the *setToFirst* cursor method is called for all twig nodes, thus, all cursors are properly initialized. Listing 8.4 presents the main part of the extended TwigOpt algorithm embedded into the *next* function. As you can see, the algorithm is largely built upon the functions, we sketched in Listings 8.2 and 8.3. Other functions will be introduced as required.

We refer to the parameter `twigRootNode` as R. Furthermore, the algorithm buffers intermediate results in a buffer queue Q. If this queue is not empty, its first element

Listing 8.4 The TwigOpt main algorithm embedded in the *next* function (adapted from the original by [Fontoura 05])

Parameter variables:
`TwigOptNode twigRoot` as R;

Local variables:
BufferQueue Q; // buffers intermediate results; initialized in open call

TwigOpt.next

```
1  begin
2      while Q is empty and not done() do
3          TwigOptNode q ← R.minCursorPathNode(); // get twig node having cursor with smallest ID
4          while q is not NULL and q.checkSolutionExtension() is false do
5              q.moveCursors();
6              q ← R.minCursorPathNode();
7          end
8          if q is not NULL then // solution extension found → try to do output and advance cursor
9              q.outputAndPush(Q);
10             q.getCursor().forwardTo(q.getCursor().getCurrentPlusOne());
11         else
12             R.outputAndPush(Q);
13         end
14     end
15     if Q is empty then
16         return NULL; // no more nodes found
17     end
18     return Q.first();
19 end
```

is removed and returned (method *first* in line 18). Otherwise, if Q is empty, results have to be generated by the twig operator. This happens in lines 3 to 13. The algorithm operates, as long as method *done* does not return *true*. Essentially, *done* embodies the termination criteria for the twig algorithm. The method returns *true*, if 1) all cursors are consumed, or 2) in case of a positional predicate on the twig root node, if we know that no more results have to be generated (pre-emption). Let us assume, that we are not done. Then, function *minCursorPathNode* returns the twig node with the smallest cursor position q. This method is implemented using the *compareTo* method defined in the cursor interface (see Listing 8.3). If all cursors are consumed, this method returns NULL. If the result is not NULL, the algorithm tries to find a solution extension on q (method *checkSolutionExtension*, line 4). As long as no solution extension has been found, the algorithm moves its cursors (function *moveCursors*) and retrieves the new twig node with the smallest cursor. If a solution extension has been found and not all cursors are consumed (i. e., if q is not NULL), the algorithm tries to generate output by calling the *outputAndPush* method. This method receives the buffer queue, to which it writes the output tuples. Besides trying to write output, this method also pushes the XML node that has a solution extension to q's stack. We will discuss this method in detail below.

After the output has been generated, q's cursor is physically moved by calling the *forwardTo* cursor method. Because we need to provide a DeweyID for this method, we simply *generate one* out of DeweyID d at the current cursor position. The generated DeweyID is both, lexicographically larger than d and smaller or equal to any existing (following) DeweyID in the document. This ID is calculated by cursor method *getCurrentPlusOne* without document access, by simply appending ".1" to d.

Listing 8.5 The a) *checkSolutionExtension* method and b) the *constraints* helper function

Local variables (for both methods):
`Cursor cursor` as C;
TwigOptNode $p \leftarrow getPathParent()$; // the path parent of this TwigOptNode

a) *TwigOptNode.checkSolutionExtension*

```
1  begin
2  |  if not allCursorsAreReal() or not constraints() then
3  |  |  return false;
4  |  end
5  |  return p = NULL or p.stackContains(C); // parent stack contains ancestor of C
6  end
```

b) *TwigOptNode.constraints*

```
1  begin
2  |  for every TwigOptNode child c among the children do
3  |  |  if c is an "and" node and not C.containsAllCursorsOf(c.getChildren()) then
4  |  |  |  return false;
5  |  |  else if c is an "or" node then
6  |  |  |  if c is "outer" then
7  |  |  |  |  continue;
8  |  |  |  else if not c.containsSomeCursorsOf(c.getChildren()) then
9  |  |  |  |  return false;
10 |  |  |  end
11 |  |  else if not C.contains(c.getCursor()) or not c.constraints() then //here, c is an ordinary path node
12 |  |  |  return false;
13 |  |  end
14 |  end
15 |  return true;
16 end
```

Finally, if no solution extension has been found and if all cursors are consumed, the algorithm tries to generate output for the last time (in line 12). Here, the call to the *outputAndPush* method is necessary to assemble and flush the remaining internally buffered intermediate results.

In the following, we omit further details for the *done*, *first*, *minCursorPathNode*, and *getCurrentPlusOne* methods. Instead, we show the *checkSolutionExtension*, *moveCursors*, and the *outputAndPush* functions.

Checking the Solution Extension

The method *checkSolutionExtension* can be called on a physical twig node q. Its semantics is depicted in Listing 8.5a. If not all cursors in the twig are real, the method immediately returns *false* (in this case, the cursors have to be moved). Otherwise, the method calls the *constraints* helper method. This method actually recursively checks the constraints of the solution extension, as we will see below. If the constraints are not fulfilled, the method returns *false*. Otherwise, it checks, whether q's current cursor position is part of a solution extension. This is the case, if either q is the twig root or if the stack of q's path parent twig node p contains an ancestor of the current position of q's cursor (checked by method *stackContains* in line 5).

In the *constraints* method, we iterate over all child twig nodes of q, where we distinguish the three twig node types: *and* nodes, *or* nodes, and *path* nodes. If the child c is an *and* node, method *containsAllCursorsOf* is called with q's grand children below c. Essentially, it checks, whether the cursors of all these grand children are contained in the cursor of child c and whether these grand children also fulfill

Listing 8.6 The *containsAllCursorsOf* method

Parameter variables:
TwigOptNodes as N_c; // the children this method was called for

Local variables (for both methods):
`DeweyID currentID` as p;

Cursor.containsAllCursorsOf

```
1  begin
2      for TwigOptNode c in N_c do // i. e., for each grand child
3          if c is "and" node and not containsAllCursorsOf(c.getChildren()) then
4              return false;
5          else if c is "or"node then
6              if c is "outer" then
7                  continue;
8              end
9              if not containsSomeCursorsOf(c.getChildren()) then
10                 return false;
11             end
12         else
13             if c.getCursor().isConsumed() then
14                 return false;
15             end
16             if not p.isAncestorOf(c.getCursor().getCurrentID()) then
17                 return false;
18             end
19             if not c.constraints() then
20                 return false;
21             end
22         end
23     end
24     return true;
25 end
```

the constraints. We will discuss this method in detail below. If c is an *or* node, we distinguish ordinary *or* nodes from *outer* nodes. If the node is *outer*, we do not have to check anything at all, because outer semantics always leads to a solution extension (as motivated in our example in the introduction). If the node is an ordinary *or* node, method *containsSomeCursorsOf* is called. This method is very similar to *containsAllCursorsOf*. However, because of the *or* semantics, it is sufficient that the cursor of only *one* grand child is contained in c's cursor (and that this grand child has a solution extension). Finally, in case c is an ordinary path node, its cursor has to be contained in q's cursor and c itself has to fulfill the constraints. Otherwise, the constraints are not fulfilled for q.

To conclude this discussion about checking the solution extension, let us consider the *containsAllCursorsOf* method. We omit the corresponding *containsSomeCursorsOf* method for brevity. Both methods are part of the cursor interface (however, we did not introduce them in Listing 8.3). The structure of the *containsAllCursorsOf* method is the same as the structure of the *constraints* method: It iterates over all passed child nodes. If a child node c is an *and* node or an *or* node, the same semantics is applied, and the *containsAllCursorsOf* (*containsSomeCursorsOf*) method is called recursively. Otherwise, if child c is a path node, 1) its cursor may not be consumed, 2) its current position of c's cursor is a descendant of the current cursor position p, and 3) c itself fulfills the constraints. If these three criteria are matched, the method returns *true*, otherwise, there is no solution extension. Intuitively, the *containsSomeCursorsOf* methods works similarly. The major distinction, however, is

Listing 8.7 The *moveCursors* method

Preconditions:
This algorithm is called for a TwigOptNode q which
- has the smallest cursor position and
- has no solution extension.

Local variables:
`Cursor cursor` as C_q;
TwigOptNode $p \leftarrow getPathParent()$; // the parent path node of this node
Cursor $C_p \leftarrow p.getCursor()$; // the parent's cursor

TwigOptNode.moveCursors

```
1  begin
2      if p is not NULL and p.stackContains(C_q) then
3          if not C_p.isConsumed() then
4              DeweyID b ← C_p.getCurrentPlusOne();
5              if C_q.getCurrentID().compareTo(b) < 0 then // set cursor C_q virtually below C_p
6                  C_q.setCurrent(b);
7                  C_q.setVirtual(true);
8              end
9          else
10             // this cursor is also consumed
11             C_q.setConsumed();
12             C_q.setVirtual(true);
13         end
14     end
15     moveCursorsBottomUp();
16     moveCursorsTopDown();
17     if this node is minCursorPathNode() then
18         // cursor movement had no effect, because this node is still the smallest one → advance
19         Cursor C_c ← chooseBestVirtualCursor();
20         C_c.forwardTo(C_c.getCurrentID());
21         C_c.setVirtual(false);
22     end
23 end
```

that only one child has to fulfill all these criteria (be it an *and* node, an *or* node, or an *path* node).

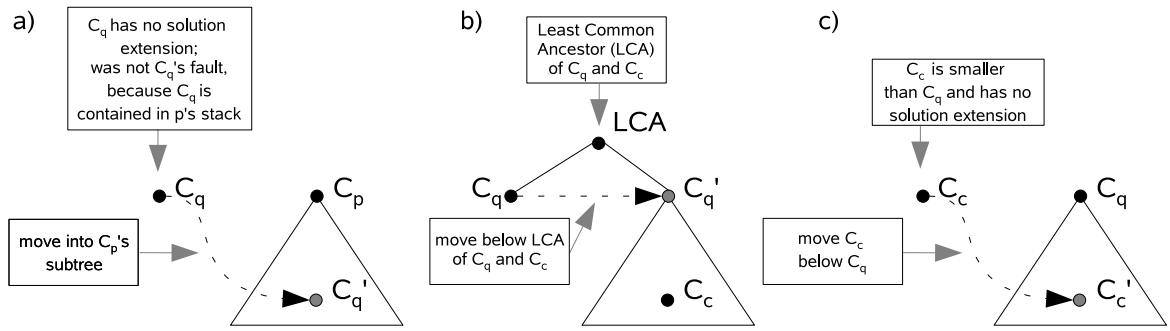
Cursor Movement

In our introductory example, we have already discussed how cursors are moved in the extended TwigOpt operator. A cursor can be *real*, if it points to a node in the document, or it can be *virtual*, if it points to a computed position. Every time the *forwardTo* method is called on a cursor, the resulting position is real, because the document (or an index) is accessed to retrieve an XML node. This XML node then contributes the `currentID` variable (see Listing 8.3) of the cursor.

In the main algorithm, if no solution extension can be found for a twig node q, the TwigOpt operator calls the *moveCursors* method. This method tries to adjust the cursors in the subtree below q such that the probability for a solution extension is maximized. Method *moveCursors* is depicted in Listing 8.7.

Let us start with the first *if*-block in lines 2 to 14: We assume that the *moveCursors* method was called for node q and that node q has a path parent p. The situation checked by the *if*-condition is depicted in Figure 8.7a. If a path parent exists, and if the stack of this path parent contains the current cursor position of q's cursor C_q, we have to move C_q into the subtree below p's current cursor C_p. The rationale is the following: Before the move, C_q had no solution extension (otherwise, the

Figure 8.7 The *moveCursors* function illustrated: a) initial move, b) *moveCursorsBottomUp*, and c) *moveCursorsTopDown*



moveCursors method would not have been called; see preconditions in Listing 8.7), but this was not C_q's fault, because C_q is contained in p's stack (and, thus, fulfills the necessary criterion for a solution extension). Essentially, this means that some twig node below q is responsible for the non-existence of a solution extension. In this situation, we know that q will not contribute to any twig match and that it is safe to move C_q into the subtree below C_p. When cursor C_p is not consumed, this happens in lines 3 to 9: We calculate a virtual position in the subtree below C_p and set this position to the cursor. If C_p is already consumed, C_q is obviously also consumed (because there is no subtree anymore, into which we could move C_q).

After this adjustment, the two helper methods *moveCursorsBottomUp* and *moveCursorsTopDown* are called. Let us again assume that p is the path parent of q and that C_p and C_q are their cursors. Both methods recursively descend down the tree and adjust C_p and C_q (by calculating virtual positions). Method *moveCursorsBottomUp* modifies C_p such that the new position contains C_q (see Figure 8.7b). Method *moveCursorsTopDown* modifies C_q such that it is contained by C_p (see Figure 8.7c). We will discuss these algorithms in detail below.

The main algorithm calls *moveCursors* for TwigOptNode q with the smallest cursor position (see preconditions in Listing 8.7). If, after the so-far described (virtual) cursor movements, q is still the node with the smallest cursor, we would run into an infinite loop (because the *moveCursors* method would again be called by the main algorithm on q). Therefore, we have to physically move one cursor to "move to a new subtree in the document". Of course, we want to skip as many subtrees as possible, to find the "right-most" one in the document. The question is now, which cursor can be physically advanced the farthest. The answer depends on the document characteristics and cannot be given in general. In our implementation, we hide this decision behind the *chooseBestVirtualCursor* method which, essentially, always chooses the C_q cursor as a heuristics. This part of the algorithm is implemented in lines 17 to 22 of Listing 8.7.

To conclude the discussion about cursor movement, let us take a look at the *moveCursorsBottomUp* and the *moveCursorsTopDown* methods. Both are shown in Listing 8.8. The first recursively calls itself for every child node. "On the way back", the cursors are moved. If the node is an *and* node, the cursor of this *and* node can be set to the right-most (maximum) cursor position out of all children's cursors (method

Listing 8.8 The a) *moveCursorsBottomUp* method and the b) *moveCursorsTopDown* method

Local variables:
`Cursor cursor` as C_q;

a) *TwigOptNode.moveCursorsBottomUp*

```
1  begin
2      for every TwigOptNode child c among the children do
3          c.moveCursorsBottomUp();
4      end
5      if this is an "and" node then
6          C_q.setCurrentID(maxCursorChild().getCursor().getCurrentID());
7      else if this is an "or" node then
8          if this is not "outer" then
9              C_q.setCurrentID(minCursorChild().getCursor().getCurrentID());
10         end
11     else if q has a child c then
12         C_c ← c.getCursor();
13         if C_q is in document order before C_c and C_q does not contain C_c then
14             if C_c.isConsumed() then
15                 C_q.setConsumed();
16                 C_q.setVirtual(false);
17             else
18                 C_q.setCurrentID(C.getPositionBelowLCA(C_c));
19                 C_q.setVirtual(true);
20             end
21         end
22     end
23 end
```

b) *TwigOptNode.moveCursorsTopDown*

```
1  begin
2      for every TwigOptNode child c among the children of q do
3          if c is an "and" node or c is an "or" node then
4              c.getCursor().setCurrentID(C_q.getCurrentID());
5          else if c.getCursor().compareTo(C_q) < 0 and not q.stackContains(C_c) then
6              if not C_q.isConsumed() then
7                  C_c.setCurrentID(C_q.getCurrentPlusOne());
8                  C_c.setVirtual(true);
9              end
10         end
11         c.moveCursorsTopDown();
12     end
13 end
```

maxCursorChild; line 6). On the other hand, if the node is an *or* node and if this *or* node is not *outer*, its cursor is set to the smallest cursor position *minCursorChild* (line 9). The rationale is that for an *and* node, we can skip as many subtrees as possible (because we know, that these subtrees do not contribute any result). For an *or* node, we can only skip the subtrees until the first child cursor is met, because after this position, further matches can probably be found. If the node traversed "on the way back" (i. e., node q) is a path node (with one child c), we can adjust q's cursor C_q with the child's cursor C_c. This only happens, if C_q is strictly situated before C_c, i. e., if their subtrees in the document do not overlap and the ID of C_q is smaller than the ID of C_c. This situation is also depicted in Figure 8.7b. If C_c is already consumed, we can also set C_q as consumed (lines 15 and 16). Otherwise, we calculate the first position below the least common ancestor (LCA) of C_q and C_c (which is done by appending ".1" to the DeweyID of the LCA). This position is set as the new virtual position to C_q.

Similarly, *moveCursorsTopDown* adjusts the child cursors w. r. t. their parent cursors.

Listing 8.9 The *outputAndPush* method

Parameters:
BufferQueue Q;

Local variables:
`Stack nodeStack` as S_q^n;
`Stack positionStack` as S_q^p;
`Cursor cursor` as C_q;
int $s \leftarrow$ number of children;

TwigOptNode.outputAndPush

```
 1 begin
 2     if q is the twig root then
 3         while S_q^n is not empty and (C_q is consumed or S_q^n[0] does not contain C_q) do
 4             outputRootStack(Q);
 5             cleanStacks();
 6         end
 7     end
 8     if not C_q is consumed then
 9         S_q^n.push(C_q.getCurrentID());
10         IntegerList m ← [s]; // initialize a list of integers of size s
11         int x ← 0;
12         for each child c among the path children do
13             m[x] ← c.getStack().getSize();
14             x ← x + 1;
15         end
16         S_q^p.push(m);
17     end
18 end
```

TwigOptNode.outputRootStack

```
 1 begin
 2     int p ← 0;
 3     for int i ← 0 to stack size do
 4         if preCheckPositionalPredicate(p) then
 5             Tuple t ← outputOneEntry(i);
 6             if t is not NULL and checkPositionalPredicate(p) then
 7                 Q.add(t);
 8             end
 9             p ← p + 1;
10         else
11             break;
12         end
13     end
14 end
```

The algorithm is depicted in Listing 8.8b. It iterates over all child twig nodes c. If the child is an *and* node or an *or* node, the corresponding cursor is set to the current position of the parent cursor C_q (line 5). Otherwise, if the child cursor is smaller than C_q and if this cursor is not contained in q's stack (i. e., if it has no solution extension), we can move the child's cursor below C_q. This situation is depicted in Figure 8.7c.

After this in-detail discussion on the extended TwigOpt matching and cursor movement phases, we will now take a look at output generation.

8.4.5 TwigOpt Output Generation

While the matching and moving process of our TwigOpt implementation was quite close to the original, output generation is different. The reason for that is naturally

the extended expressiveness of our algorithm. As already explained in the introduction, output is generated, when the bottom-most XML node stored at the root stack is no *ancestor* of the XML node issued for intermediate storage on that stack. At this point, all matches below the bottom-most XML node are encoded on the stacks of the TwigOptNodes. The output generation process reads these stacks and assembles a result according to the configuration of the twig nodes. Because of our extended expressiveness, the strategy to derive the result is quite complex. The algorithm is distributed over six methods, of which some call each other recursively. The entry method is *outputAndPush* (Listing 8.9a). In case, output can be generated, this method calls *outputRootStack* (Listing 8.9b) to actually generate the output. Independent of that, *outputAndPush* maintains the stacks. Therefore, it pushes new XML nodes (represented by their DeweyIDs) onto the stacks or it removes old nodes. Method *outputRootStack* iterates over each entry on the stack of the twig root. For each of these elements, it calls *outputOneEntry* (Listing 8.10) to assemble and fetch the result for this element. The call structure of the next three methods has the following shape: *outputOneEntry* calls *processChildren* (Listing 8.11), when a twig node is an internal node. For each twig child, *processChildren* calls *outputOneStack* (Listing 8.12) to retrieve the result from the child's stack. Method *outputOneStack* is quite similar to *outputRootStack*. It recursively calls *outputOneEntry* for each entry on the stack. Thus, the recursion loop is closed. Naturally, it terminates on the leaf nodes of the twig. Method *outputOneEntry* collects the tuple results generated for each child and calls the *processTuple* method (Listing 8.13) which, in turn, applies filters, output expression, projection, etc.

Let us start our discussion with the *outputAndPush* method. The method receives a buffer queue Q as a parameter, where it will store the results. For brevity, we declare member variable `nodeStack` as S_q^n, `positionStack` as S_q^p, and so on. Local variable s captures the number of twig node children. The algorithm first checks if it is called for the twig root node (only for the root, output can be generated). In this case, we check whether the cursor C_q is consumed or whether the first ID found at the bottom-most node stack position $S_q^n[0]$ is an ancestor of the current cursor position. In either case (and if the stack is not empty), we can produce output. The actual output generation is handled by the *outputRootStack* method. After the output has been generated, all stacks can be cleared (by *cleanStacks*).

Independent of output generation, if cursor C_q is not consumed, we have to push the ID of the current cursor position onto the stack (line 9) and we have to compute the group markers and add them to the position stack. A group marker is calculated for each path child c of node q. It simply captures the group for the currently pushed ID starts where on c's stack. This position can be calculated by retrieving c's stack size. Note, this is only possible, because the *children stacks* for a match are populated after their *parent stacks* have been filled. All group markers are collected in an integer list and pushed onto the position stack (lines 10 to 16).

Method *outputRootStack* initializes a position counter p and iterates over all stack entries. As explained, positional predicates can be pre-emptive. Method *preCheckPositionalPredicate* checks these pre-emptive predicates and stops the output generation, when the predicate will not return *true* for any following call. After that, an intermediate result tuple is generated by calling *outputOneEntry* for the current stack entry. If this tuple is not null and if a possibly existing positional predicate is fulfilled, the tuple is appended to buffer queue Q.

Listing 8.10 The *outputOneEntry* method

Parameters:
int i // the current position in S_q^n to generate output for

Local variables:
TwigOptNode q; // *this* TwigOptNode
`Stack nodeStack` as S_q^n;
`Stack positionStack` as S_q^p;
int $s \leftarrow$ number of path children that produce internal/external output;
Tuple r; // an intermediate result tuple;

TwigOptNode.outputOneEntry

```
1  begin
2  |  DeweyID e ← S_q^n[i];
3  |  if output mode is not ("none" or "internal child") then
4  |  |  r ← [s + 1]; // create empty intermediate tuple of size s + 1
5  |  |  r ← r + e; // append output entry to tuple i
6  |  else
7  |  |  r ← [s]; // create empty intermediate tuple of size s
8  |  end
9  |  if this twig node has no children then
10 |  |  return processTuple(r);
11 |  end
12 |  IntegerList p ← S_q^p[i]; // get the group start positions
13 |  ItemList l ← processChildren(q, e, p);
14 |  if l is not NULL then
15 |  |  r ← r + l; // append complete list to intermediate result tuple
16 |  else
17 |  |  return NULL;
18 |  end
19 |  return processTuple(r);
20 end
```

Method *outputOneEntry* is called on twig node q for a specific stack position i to assemble the output "below" the XML node at that position. First XML node e (represented by its DeweyID) is fetched from the stack. If the twig node, the method was called for, generates output, an intermediate result tuple r of size $s + 1$ is instantiated, where s is the number of children producing output. Furthermore, DeweyID e is set as the first tuple field. If the twig node does not produce any output, the size of the intermediate tuple is simply s.

In case q is a leaf twig node, everything is now ready to produce the result for q. This happens during the *processTuple* method, which we will discuss later. On the other hand, if q has children, we have to process their result, too. To do so, we fetch the group markers from position stack S_q^p a call *processChildren* (additionally passing q and the current output node e). Method *processChildren* returns a list of items, one item for each child. For those children that produce internal/external output (recognizable by their output modifiers), their result is written to intermediate tuple r. In the case, *processChildren* returned NULL, this twig node also returns NULL. This can happen when some internal filter did not match, thus, the subtree does not return a result. Otherwise, if *processChildren* returned a result, it is appended to intermediate tuple r which is, in turn, processed by *processTuple* and returned.

Method *processChildren* is called for every stack entry e of an internal twig node q. The result of this method is a list of items, one for each child producing output. The method iterates over all children c of q. If the child is a path node and if the child or any descendant twig node produces an internal/external output relevant for q,

Listing 8.11 The *processChildren* method

Parameters:
TwigOptNode q; // the parent twig node
DeweyID e; // the entry for which output shall be generated
IntList s; // the group start positions

Local variables:
ItemList r; // the result list of items;

TwigOptNode.processChildren

```
1  begin
2      for each child c of q at position i do
3          if c is path node then
4              if c or any descendant twig node produces internal/external output then
5                  Item p ← c.outputOneStack(s[i], e);
6                  if p is not NULL then
7                      r ← r + p; // append sequence generated from one stack
8                  else
9                      if q is "or" node then
10                         r ← r + (); // append empty sequence
11                     else if q is "and" node then
12                         return NULL;
13                     end
14                 end
15             end
16         else
17             // c is "and" node or "or" node
18             ItemList l ← c.processChildren(q, e, s);
19             if l is NULL then
20                 return NULL;
21             end
22             if c is "or" node then
23                 boolean b ← false;
24                 for each item i in l do
25                     if i is not empty sequence then
26                         b ← true;
27                         break;
28                     end
29                 end
30                 if not b then
31                     if c is "outer" then
32                         return l;
33                     end
34                     return NULL;
35                 end
36             end
37             return l;
38         end
39     end
40     return r;
41 end
```

method *outputOneStack* is called, which computes the result for one child stack. The scanned range on the child stack depends on the starting position $s[i]$ and the given stack element from the parent stack e. We will discuss this method below. Its result is a sequence, which is appended to item list r. If *outputOneStack* returned NULL, and q is an *or* node, we generate an empty sequence as a representative for the missing subtree. Otherwise, in case of an *and* node, the subtree did not match, and we return NULL.

In case, q itself is an *and*/*or* node, we recursively call *processChildren* on q's children. The result may not be NULL, because then no match was found in the subtree below

Listing 8.12 The *outputOneStack* method

Parameters:
int s; // the start position in the stack
DeweyID e; // the entry for which output shall be generated

Local variables:
`Stack nodeStack` as S_q^n;
`Stack positionStack` as S_q^p;
TupleSequence r; // the result tuple sequence

TwigOptNode.outputOneStack

```
 1 begin
 2     int p ← 0;
 3     for each node n at position i in S_q^n starting at s, as long as n is a descendant of e do
 4         if incoming axis is not "child" or n is child of e then
 5             if preCheckPositionalPredicate(p) then
 6                 Tuple t ← outputOneEntry(i); // recursive call
 7                 if checkPositionalPredicate(p) then
 8                     if t is not NULL then
 9                         r ← r + t;
10                     end
11                 end
12             else
13                 break;
14             end
15         end
16         p ← p + 1;
17     end
18     if size of r is 0 and output mode is not "none" then
19         return NULL;
20     end
21     return r;
22 end
```

Listing 8.13 The *processTuple* method

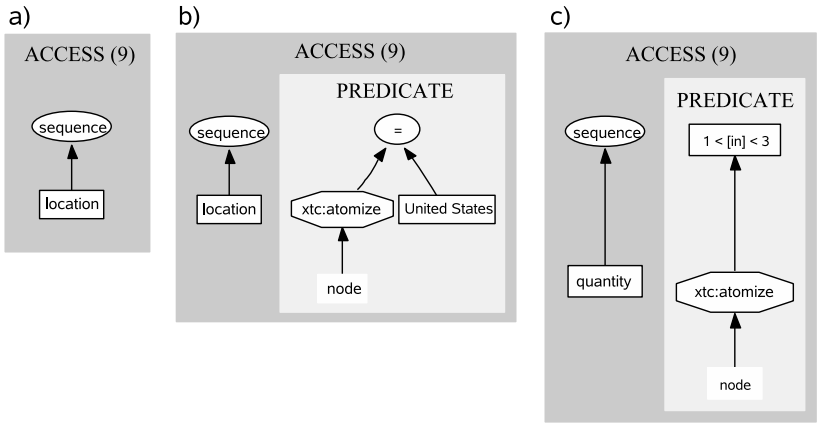
Parameters:
Tuple t; // the tuple to process

Local variables:
`PALExpressions filters` as F_q; // a list of filters
`PALExpression expr` as E_q; // the output expression

TwigOptNode.processTuple

```
 1 begin
 2     if q has one or more output filters F_q then
 3         for each filter expression e in F_q do
 4             if not e(t) then
 5                 return NULL;
 6             end
 7         end
 8     end
 9     if q has output expression E_q then
10         t ← E(t); // evaluate the expression on the tuple and replace value of t
11     end
12     t ← project(t);
13     if q is no "grouping" node then
14         t ← unnest(t);
15     end
16     return t;
17 end
```

Figure 8.8 Various XQGM access operators to be mapped to indexes



q. When c is an *or* node, we demand that at least one child returned a non-empty sequence, otherwise, we return NULL (lines 22 to 36). When c is *outer or*, all children may return empty sequences. If we do not return NULL, the result delivered by the children is simply returned.

Generating output for one stack is quite simple. The structure of the algorithm is similar to *outputRootStack*, however, the range to produce output for is restricted here. The start position i is given by the caller of the *outputOneStack*. The end position is reached, when the element read from the stack is no descendant of the passed entry e from the parent stack. Additionally, the method checks the *child* axis, if necessary, and evaluates positional predicates (again, relying on *preCheckPositionalPredicate* and *checkPositionalPredicate*). Another major difference to the *outputRootStack* method is that NULL is returned (lines 18 to 20), if the intermediate result remains empty, but the node should actually return some result.

To conclude the discussion about the TwigOpt algorithm, let us take a look at how intermediate tuples are processed. Method *processTuple* is called by *outputOneEntry* for the intermediate result generated for a stack entry. Let us assume that, for a particular twig node q, each twig child, as well as q generates output. Then, the intermediate tuple constructed in *outputOneEntry* has the following form: `[e,` s_1`,` s_2`, ...,` s_n`]`, where `e` is the stack entry and the s_i are sequences, one generated for each child. This intermediate tuple is passed to *processTuple*, on which the method applies the output filters (lines 2 to 8), the output expression (lines 9 to 11), projection (line 12), and unnesting (lines 13 to 15). An output filter simply discards a tuple (and returns NULL), when the filter evaluates to *false*. The result generated by an output expression replaces the input tuple. Projection is necessary to remove information generated for twig nodes with an *internal* output mode: When the internal information is not required anymore, we can remove the corresponding tuple fields. Finally, unnesting computes the Cartesian product on the sequences contained in the tuple. The result is again a sequence which is embedded into a result tuple and returned by the *unnest* method. We do not further specify the semantics of these helper methods algorithmically, because they are quite straightforward.

8.5 Index-Based PPOs

In the previous chapter, we have introduced four types of indexes: the content index, the path index, the CAS index, and the element index. Together with the document container, we, therefore, can consider five indexes for query evaluation. In the discussion so far, we only have utilized the document container and the element index. They can be used for navigational access by single-node PPOs or by the NavTree operator, or they can be used in jumping TwigOpt cursors. We now want to enable the plan generator to also exploit the other three indexes.

Basically, an index can be used for the implementation of plain XQGM sequence access operators, such as the ones depicted in Figure 8.8. In the following section, we will show the various alternatives for that. On the other hand, path indexes can also be combined with twig pattern matching: To utilize a path or a CAS index, path information has to be available in the XQGM. Otherwise, index matching (see Section 7.3.7) could not find any applicable indexes. In an XQGM instance, path information is captured by the twig operators. However, not always all branches in a twig may be covered by a path/CAS index. To nevertheless utilize such an index, we have to integrate it into the TwigOpt algorithm. We will see in Section 8.5.2 how this works.

8.5.1 Simple Index Mapping

Throughout this work, we have seen many XQGM sequence access operators. Basically, such an operator returns all nodes that fulfill a certain node test, and sometimes, they additionally match a content predicate. For an example, see Figure 8.8. The first operator is a simple sequence access operator that delivers all *location* nodes of a document. The second access operator delivers only those *location* nodes, whose content is "United States". Finally, the third operator contains a *between expression* and delivers all *quantity* nodes with a content between 1 and 3. The easiest way to implement these three operators is a *document scan*. The physical operator has the following signature:

```
DocumentScanOperator (Document document,
                      NodeTest nodeTest);
```

This operator works on the document container and is, therefore, always available. The operator returns all nodes from the document that fulfill the given *nodeTest*. The *nodeTest* is optional. If left undefined, the operator returns *all* nodes from the document (surely, a quite expensive operation). A possibly available predicate has to be checked by a following select operator. For its evaluation, the `DocumentScanOperator` relies on the document reconstruction capabilities introduced in Section 6.3.3 on Page 176.

Because a document scan is quite expensive, we allow to share its result such that it is possible to pass the result to various other operators. Sharing is implemented by a *split* operator (distributing the input) and by so-called *split adapters* that apply node tests. We omit the details here and refer to Figure 8.6 on Page 236 for an example. As you can see, the output of the document scan is used as an input for a twig operator. Furthermore, the content predicate (i. e., `[. = "United States"]`) is mapped onto a select operator.

A similar operator exists to scan the element index:

```
ElementIndexScanOperator (Index index,
                          QName elementName);
```

Here, the plan generator has to find and provide the index to be queried. Furthermore, the name of the elements to be retrieved has to be supplied. Note, the operator cannot be used for the implementation of a sequence access operator with a node test other than a name test. Therefore, it is sufficient to only pass the *QName* to the `ElementIndexScanOperator`. As the document scan operator, the element index scan can be shared using a split operator.

Access operators with a content predicate can be mapped to a `ContentIndexScanOperator`:

```
ContentIndexScanOperator (Index index,
                          NodeTest nodeTest,
                          Item keyValue,
                          Item lowerKeyValue,
                          Item upperKeyValue,
                          boolean minInclusive,
                          boolean maxInclusive);
```

Again, the plan generator has to provide the index. If the content predicate specifies a point query (such as the example in Figure 8.8b the `keyValue` parameter is set appropriately, e. g., to "United States". In case of a range query (Figure 8.8c), the range boundaries and the inclusion flags have to be provided by the plan generator. In our example, the lower (upper) boundary is 1 (3) and both inclusion flags are set to *false*. The content index contains text nodes. However, the sequence access operator returns elements. Therefore, we have to retrieve the elements (i. e., the parent elements) from the document store (or from an element index). When using the document store, our only option is to fetch each parent separately by a key lookup. On the parent, we then evaluate the given `nodeTest`. When an element index exists, we can read all nodes fulfilling the `nodeTest` and do a structural join with the result from the index.

Furthermore, note that the result of a range query is not correctly sorted in general (because the index inverts the nodes by their content values). In this case, a reordering is necessary to emit the result nodes in document order. Note, the `ContentIndexScanOperator` can only be applied, when we know that the nodes fulfilling the node test only contain a text node (and no other children)! If this were not the case, we would probably miss an internal node, which fulfills the node test, and whose *string value* (result of *fn:atomize*) matches the content predicate. Upon index creation, this information is captured and stored in the metadata catalog of the XTC system.

The above three operators can be exploited regardless in which mode the document is stored (node-oriented or path-oriented). Path indexes are, however, only available in the path-oriented storage mode. The *CasIndexScanOperator* has a quite similar interface to the content index scan operator:

```
CasIndexScanOperator (Index index,
                      IntegerList pcrs,
                      Item keyValue,
                      Item lowerKeyValue,
                      Item upperKeyValue,
                      boolean minInclusive,
                      boolean maxInclusive,
                      boolean splidClustering);
```

The only difference is the integer list containing the PCRs and the clustering flag. For a sequence access operator with a *name test* `n`, we simply evaluate query `//n` on the path synopsis to retrieve the PCR list, e. g., `//location` or `//quantity` in our example from Figure 8.8. If an index for this PCR list exists (see Section 7.3.7), we can employ the `CasIndexScanOperator`. The necessary clustering information comes from the metadata and the remaining parameters are configured as for a content index scan. Furthermore, the same restrictions apply: only nodes with text content can be retrieved and upon a range query, additional sorting is required. However, compared to the content index scan, the CAS index scan does not need to query the document (or do structural join) to retrieve the parent nodes, because they can be directly computed using the stored DeweyIDs.

The usage of the CAS index as sketched does not require any "real" path information. Thus, we can employ a CAS index scan independently of the existence of twigs. However, in combination with twigs, we can further restrict the PCR set given to the scan operator. As an example, consider the XQGM instance from Figure 8.5a. The sequence access operator delivering the *location* elements is depicted in Figure 8.8b. It directly delivers its input to the twig operator. Therefore, we know that the output generated will contribute to path matching and that we can analyze the twig structure to infer the path pattern, e. g., `//item/location`. Instead of retrieving PCRs for `//location` as sketched above, we can now query the path synopsis for the former path, thus restricting the PCR set.

Many XML (sub-)queries do not have a content predicate, but simply have to match plain paths, e. g., path `//item/mail` in our sample query of Figure 8.5a. In the previous chapter, we have introduced path indexes to speed up this kind of queries. The operator to access such an index is the following:

```
PathIndexScanOperator (Index index,
                       IntegerList pcrs,
                       PalExpression predicate,
                       boolean splidClustering);
```

The interface is quite similar to the CAS index scan without the parameters to express the content predicate. PCRs and the clustering flag are derived as explained above. In contrast to the CAS index scan, however, we provide an additional `predicate` parameter to directly evaluate a selection on the fetched path nodes.

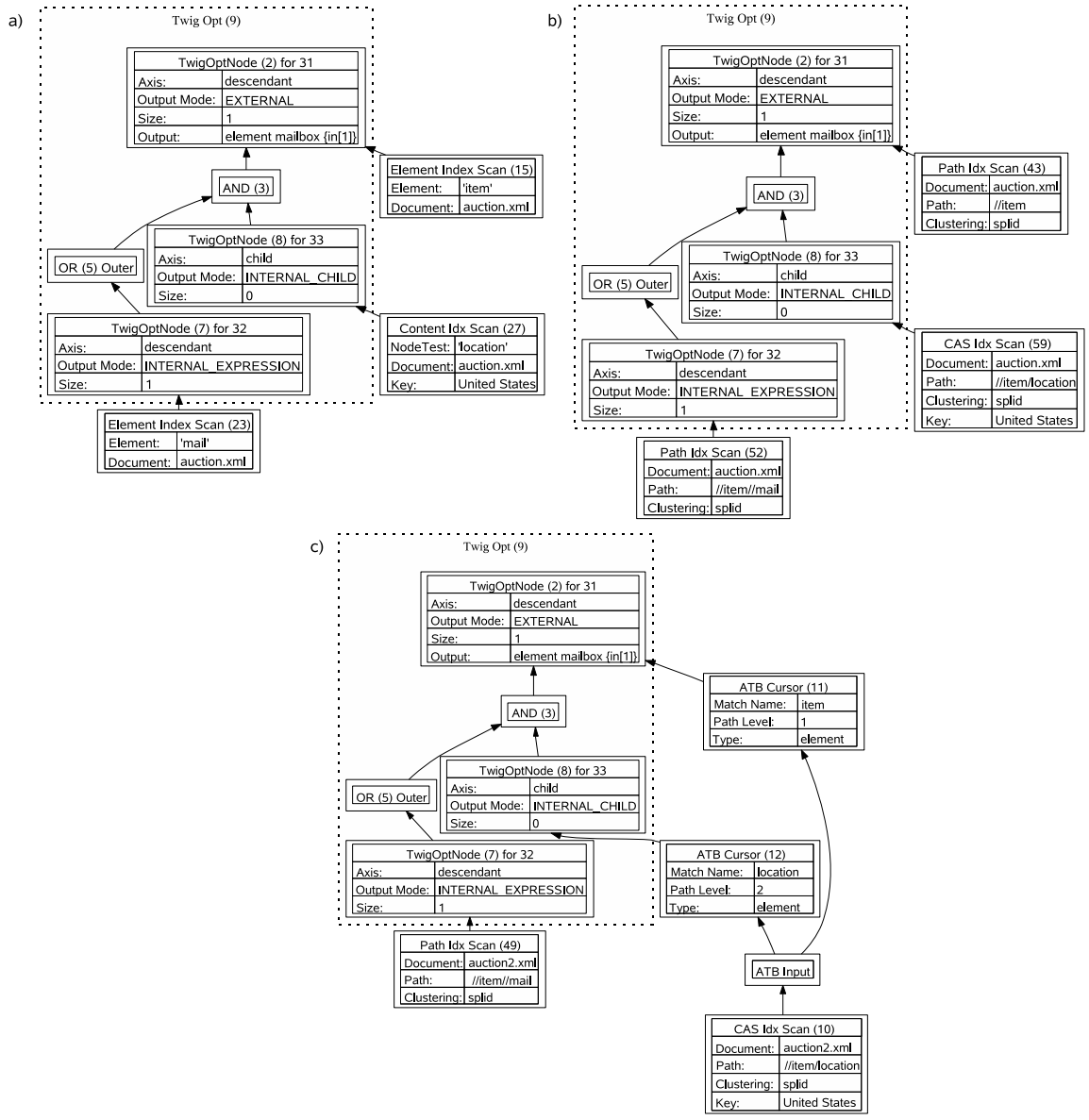
The plan generator can use a path index for the implementation of a sequence access operator as described above, either with or without further path information (inferred from a twig operator). If further path information is not existent, only path indexes of structure $//n$ can be used to implement the sequence access operator.

Finally, in the previous chapter, we have shown how path indexes can be embedded into the element index by storing PCRs. These indexes can be accessed by the following operator:

```
PathOverElementIndexScanOperator (Index index,
                                  IntegerList pcrs,
                                  QName elementName,
                                  PalExpression predicate,
                                  boolean splidClustering);
```

In addition to the path index scan parameters, we have to provide a qualified `elementName` which selects the inverted index (from the element index) on which the scan is executed. This name can be inferred from the sequence access operator

Figure 8.9 Various index-based implementations of the sample query from Figure 8.5a: a) evaluation on the element index and the content index, b) evaluation on a path index and a CAS index, and c) embedded CAS index evaluation



implemented by the scan.

In Figure 8.6 on Page 236, we have already seen an implementation of our sample query from Figure 8.5a, which is based on a shared document scan. Let us now create an element index and a content index and see how the plan generator makes use of them. Figure 8.9a presents the result. Instead of a shared document index, *item* and *mail* nodes are provided by an element index scan. The content predicate is evaluated by a content index scan with a *location* name test. To enable path/CAS indexes, we created path indexes *I(//item//mail)* and *I(//item)* as well as a CAS index *I(//item/location, String)*. In Figure 8.9b, the plan generator exploits these indexes by implementing the sequence access operators on them. The last query in this figure

embeds the CAS index into the twig operator such that the internal twig nodes are delivered directly from the index scan result. We will discuss in the following how this actually works.

8.5.2 Complex Index Mapping

In our simple index mapping scenarios, we implemented sequence access operators with various index scan operators. For the integration with twig operators, therefore, only their "leaf" input can be generated by an index scan. However, as we have sketched in the previous chapter, our indexes can deliver more than just leaf elements. They can deliver inner elements, too. This possibility opens the door to embed path and CAS index operators into the twig operator. Figure 8.9c illustrates how this integration looks like. The CAS index scan delivers the DeweyIDs of *location* elements. Because using a DeweyID, we can compute inner nodes, it is also possible to compute the necessary *item* nodes. Thus, the result of the CAS index scan can be shared and an additional scan to provide *item* nodes can be avoided. Before we discuss the actual embedding, let us take a look at how we can generate inner XML nodes from index scan results.

The Ancestor Tuple Builder

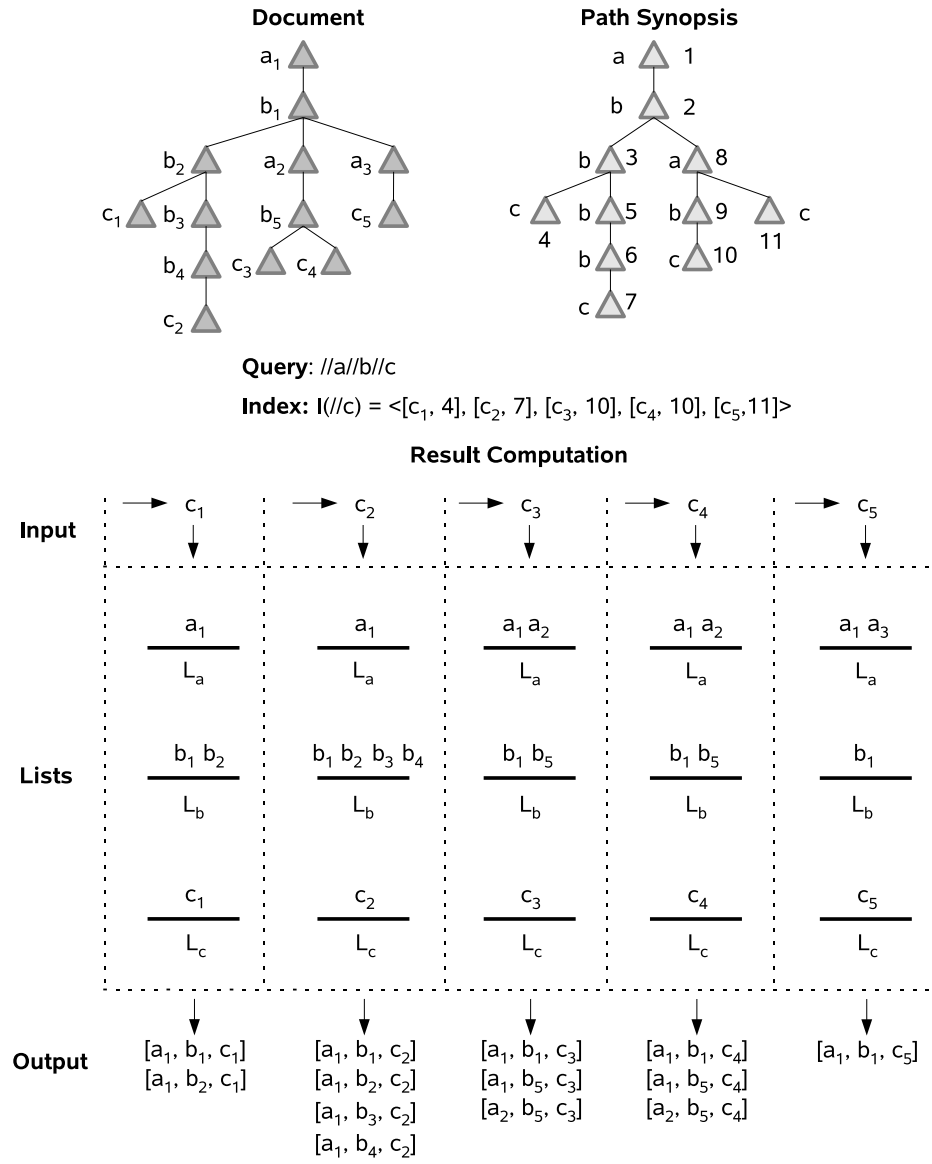
The ancestor tuple builder (ATB) is an algorithm that can receive an ordered stream of XML nodes as input and that can produce an ordered stream of tuples as output, where these result tuples contain the ancestors of the nodes in the input stream. To introduce the algorithm, let us take a look at the example depicted in Figure 8.10. The figure shows a synthetic document and its path synopsis. On this document, query `//a//b//c` shall be evaluated and a path index *I(//c)* exists. With the integration into a twig join operator in mind, we demand that not only the c nodes are returned, but also the ancestors a and b. Our path index contains and delivers c nodes only (together with the node's PCR). However with the DeweyID (not depicted) and the path synopsis, we can compute the inner elements.

For every path step in the query, the ancestor tuple builder maintains a list L. The concept here is similar to the TwigOpt algorithm: A list stores XML nodes of a certain name. However, in contrast to TwigOpt, the nodes in the list have to fulfill the *descendant* relationship[2], i. e., $L[i+1]$ is a *descendant* of $L[i]$. Let us now take a look at how the algorithm produces results. For every node returned by the index scan, method *processOneElement* is called. The pseudocode is shown in Listing 8.14. Besides node N, the method also receives the path synopsis P and the path expression E for which it was called. Based on E, it creates a set of lists L, one list for each path step. In the first processing phase, the algorithm fills its internal lists. Then output is produced.

To fill the lists, the algorithm first initializes an `AncestorBuilder` with the current node N, the path synopsis P, the path expression E, the PCR R, and the set of lists L. The `AncestorBuilder` is a small helper class. Its main functionality is to compute the ancestors of a given node N. For example, if $N = c_1$, it produces c_1, b_2, b_1, and a_1 (in this order). To access the generated ancestor nodes, the `AncestorBuilder` provides iterator functions: *hasNext* returns *true*, if more an-

[2]Note for TwigOpt, this property held only for the root stack. On the other stacks, the elements had only be stored in document order.

Figure 8.10 An ancestor tuple builder example



cestors can be computed using N, and *getCurrentNode* actually returns the ancestor. During processing, we need some more information for each ancestor node produced: method *getCurrentList* returns the list on which the ancestor will be stored and *currentNodeIsFirstOfType* returns *true*, when the first ancestor with a particular name is returned. In our ancestor sequence from above, the method would return *true* for b_2 and *false* for b_1. The necessary information can be derived from the passed parameters.

Let us step through the algorithm. In the first iteration through the *while* loop, the ancestor builder returns node c_1 and list L_c. Because the list is empty, c_1 is simply added. The same happens for the next ancestor b_2 and list L_b. Upon b_1, L_b is not empty, because we just added b_2. Therefore, the algorithm iterates over all entries e in L_b and checks various relationships between b_1 and e. Because b_1 is an ancestor

Listing 8.14 The *processOneElement* method

Parameters:
`DeweyID deweyID` as D; // a node from an index scan
`int pcr` as R;

Local variables:
`PathExpression pathExpr` as E; // the path expression to be evaluated
`PathSynopsis ps` as P; // the structural summary of the document
`Lists lists` as L; // a list of lists

ATBinput.processOneElement

```
1  begin
2      AncestorBuilder b ← AncestorBuilder(D, L, R, P, E); // create a per-node ancestor builder
3      while b.hasNext() do
4          Node n ← b.getCurrentNode(); // get the current ancestor
5          List l ← b.getCurrentList(); // get the list on which we store n
6          if l is empty then
7              l.add(n); // simply add n
8          else
9              for each element e in l do
10                 if n = e then
11                     if b.currentNodeIsFirstOfType() then
12                         l.clearAfter(n); // remove all list entries behind n
13                     end
14                     break; // done pushing onto stacks for the current node N
15                 end
16                 if n is ancestor of e then
17                     l.addBefore(n, e); // add n at the position before e
18                     if b.currentNodeIsFirstOfType() then
19                         l.clearAfter(n); // remove all list entries behind n
20                     end
21                 else if n is no descendant of e then
22                     l.addBefore(n, e);
23                     l.clearAfter(n);
24                 else if e is last element in l and n is descendant of e then
25                     l.add(n);
26                 end
27             end
28         end
29     end
30     // now all elements are in the lists
31     produceOutput();
32 end
```

of b_2, lines 17 to 20 are executed and b_1 is stored before b_2 (see Figure 8.10). Because method *currentNodeIsFirstOfType* returns *false* for b_1, nothing more has to be done. Finally, a_1 is added to L_a. In this state, all ancestors have been computed out of N and output can be generated. Output generation is quite similar to the procudure shown for the TwigOpt algorithm. Therefore, we do not discuss method *produceOutput*. Note, in our example, we have chosen to directly produce the output, thus, generating it in leaf-to-root order (see Figure 8.10).

The next call to *processOneEntry* consumes node c_2 (ancestors c_2, b_4, b_3, b_2, b_1, and a_1). Node c_2 is no ancestor or descendant of c_1. Therefore, lines 22 and 23 are executed. Here, c_2 is stored before c_1 and then everything behind c_2 is removed from the list (i. e., c_1). Ancestor b_4 is appended to L_b, after b_1 and b_2 have been visited. Then, b_3 is stored before b_4. On b_2, lines 10 to 15 are executed (because we already have b_2 on the stack). In this case, we know that no more new information can be generated out of $N = c_2$. Therefore, we can terminate the *while* loop. Element c_3 replaces c_2 on L_c. Then, b_5 triggers the removal of b_2, b_3, and b_4 from L_b (lines 22 and 23). Finally, a_2 is

Listing 8.15 The ATBinput and ATBcursor classes

```
class ATBinput
{
  // member variables set by the plan generator
  Cursor            inputCursor;   // the cursor delivering the index output
  PathExpression    pathExpr;      // the query path

  // internal state
  PathSynopsis      ps;            // the structural summary of the document
  Lists             lists;         // a list of lists for XML nodes

  /* stack maintenance methods */
  void open();
  void processTo(int cursorID, DeweyID position);

  /* helper methods */
  void    processOneElement(DeweyID deweyID, int pcr);
  boolean allListsAreFilled();
}

class ATBcursor extends Cursor
{
  // member variables set by the plan generator
  ATBinput atbInput;  // shared ancestor tuple builder input
  int cursorID;       // the unique identification number

  // internal state
  int startPos;       // list scan start position for output generation

  /* cursor methods */
  void setToFirst();
  void forwardTo(DeweyID position);
}
```

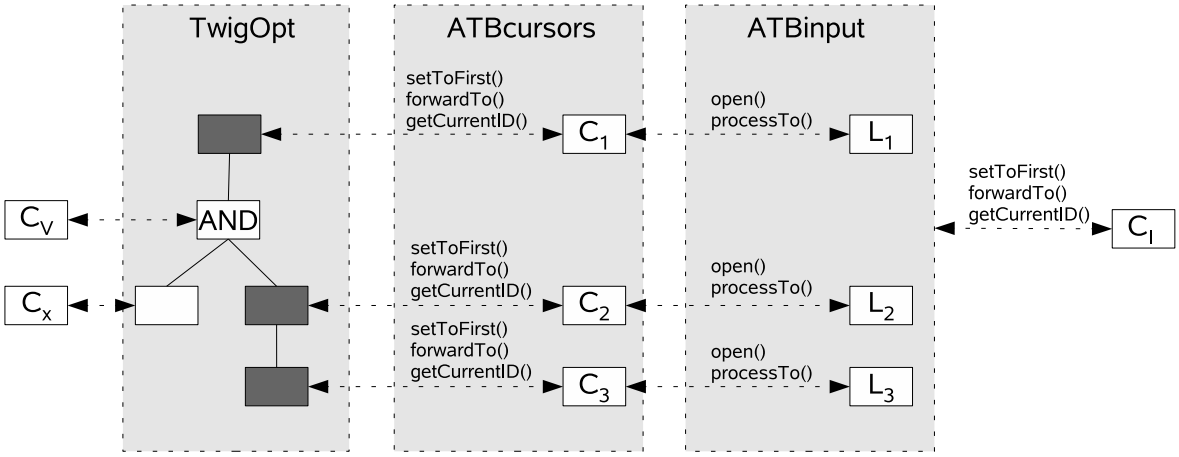
added to L_a. In the following, all code paths in the algorithm are traversed, when the remaining nodes are consumed. Here, we omit the detailed discussion and leave it open for your consideration. Note, the algorithm also works with *child/attribute* axes. Then, the `AncestorBuilder` returns only those ancestors fulfilling the axes in the given path expression.

We could implement this algorithm as a stand-alone operator in the physical algebra (together with the output generation strategies suggested for the TwigOpt operator). In combination with a path/CAS index as input, it would then be able to evaluate *linear* path queries, i. e., path queries with at most one branching path predicate. In the current version of the query engine, we did, however, not follow this approach. In contrast, we decided to integrate the ancestor tuple builder with the TwigOpt algorithm to enable flexible path index usage in complex twig patterns. The essential idea behind this integration is the similarity between the TwigOpt stacks and the lists of the ancestor tuple builder. As we will see, it is possible to directly pump the output of the ATB into the TwigOpt operator.

Path-Index Twig Embedding

For the TwigOpt embedding, we have to provide two classes: one class containing the ancestor tuple builder algorithm (`ATBinput`) and one class providing a cursor (`ATBcursor`). Obviously, the cursor makes the output of the ATBinput accessible to the TwigOpt operator. The definition of these two classes is depicted in Listing

Figure 8.11 Structure of an ancestor tuple builder mapping



8.15 and will be introduced in the following. Note for now, we skip a discussion on how twig operators with embedded indexes are mapped from the logical twig representation by the plan generator. We will return to this point in the next section. Let us assume the situation depicted in Figure 8.11. There, we have twig operator with five nodes, of which one is a Boolean *and* node and of which three nodes (depicted in a darker shaded grey) receive the output generated by an index scan. The fifth node is a path node with another type of cursor.

Each darker shaded node is connected to an `ATBcursor`. Because this class implements the standard cursor interface (from Listing 8.3), the TwigOpt operator can issue methods *setToFirst*, *forwardTo*, and *getCurrentID* (see Listing 8.15). Internally, the `ATBcursor` references the `ATBinput`. Because multiple ATB cursors reference the same ATB input, we need to distinguish them. Therefore, each ATB cursor carries a `cursorID`. Finally, variable `nextPosition` is required for the implementation of the *forwardTo* method.

The `ATBinput` class consists of several lists, one for each cursor, for which the ATB serves as input. Note, in contrast to what we have seen above, for the integration into the TwigOpt operator, not all path steps have to be covered by a list. When a cursor for a twig node on the path is served by some other input operator, we simply omit the list in the ATB input. In this case, the plan generator has to make sure to assign the correct ATB cursor to the correct list.

Of course, the ATB input requires an input itself. We implemented this input as a cursor (i. e., C_I in our example). Thus, the ATB input can call *setToFirst* and *forwardTo* to retrieve the nodes from the input path/CAS index scan. Besides the lists and the input cursor, the class has access to the path synopsis of the queried document and to the path expression derived from the TwigOpt operator. This information is required to reconstruct the ancestor nodes using the index scan result.

In the following, we will discuss the methods of the `ATBcursor` class and the `ATBinput` class. We start with the latter one. Method *processOneElement* has already been introduced in Listing 8.14. The two methods depicted in Listing 8.16, namely *open* and *processTo*, make use of the *processOneElement* method. Method *open*

Listing 8.16 The *open* method and the *processTo* method

Local variables:
`Cursor inputCursor` as C; // the input cursor

a) *ATBinput.open*

```
 1 begin
 2     if not already opened then // only-once semantics, because this method is called by all ATB cursors
 3         C.setToFirst(); // set index scan cursor on first node
 4         if not C is consumed then
 5             processOneElement(C.getCurrentID(), C.getCurrentPCR()); // push ancestors on the lists
 6         end
 7         while not C is consumed and not allListsAreFilled() do
 8             C.forwardTo(C.getCurrentID()); // advance index input
 9             processOneElement(C.getCurrentID(), C.getCurrentPCR()); // push ancestors on the lists
10         end
11     end
12 end
```

Parameters:
`DeweyID position` as p;
`int cursorID` as i; // index pointing to the list to be advanced

Local variables:
`Cursor inputCursor` as C; // the input cursor
`Lists lists` as L;

b) *ATBinput.processTo*

```
 1 begin
 2     List l ← L[i];
 3     while not C is consumed do
 4         C.forwardTo(C.getCurrentID()); // move C to the next position
 5         processOneElement(C.currentID(), C.getCurrentPCR()); // adjusts lists
 6         if l.getLastEntry().compareTo(p) >= 0 then
 7             break;
 8         end
 9     end
10 end
```

is called by each ATB cursor. Essentially, it initializes (*setToFirst*, line 3) and fetches the first element from input cursor C, reconstructs the ancestors, and pushes them into the appropriate lists (line 5). It could happen, that not all lists are filled, e. g., when a *child* axis has to be evaluated. Then, the *open* method simply fetches the next element from the input cursor and tries to fill the lists until all lists are filled (lines 7 to 10). Because opening is required only once for the ATB input, we have to check whether the input was not opened before (line 2). After the *open* call, all lists are initialized with the ancestor set of the first match from the path index.

The *processTo* method is called by an ATB cursor when it has to advance its cursor (i. e., when *forwardTo* is executed). The *processTo* method receives two parameters: the position to advance to and the ID of the cursor calling the method. Basically, the *processTo* method calls *processOneElement* until the list responsible for the cursor contains an element larger or equal to the passed position. Note, of course, all other lists are manipulated, too. At this point, a problem arises: The other cursors "loose their internal state". To alleviate the situation, we alter the "list protocol" by relaxing the *processOneElement* method. In the new protocol, a list is allowed to have non-ancestor nodes, similar to the stacks in the TwigOpt algorithm. With this relaxation, the *processOneElement* method gets even simpler. The new version is depicted in Listing 8.17. Note, cleaning the lists now has to be triggered from outside, because the new version only adds elements. An example with the relaxed list protocol can

be found in Figure 8.12.

Listing 8.17 The relaxed version of the *processOneElement* method

Parameters:
`DeweyID deweyID` as D; // a node from an index scan
`int pcr` as R;

Local variables:
`PathExpression pathExpr` as E; // the path expression to be evaluated
`PathSynopsis ps` as P; // the structural summary of the document
`Lists lists` as L; // a list of lists

ATBinput.processOneElement

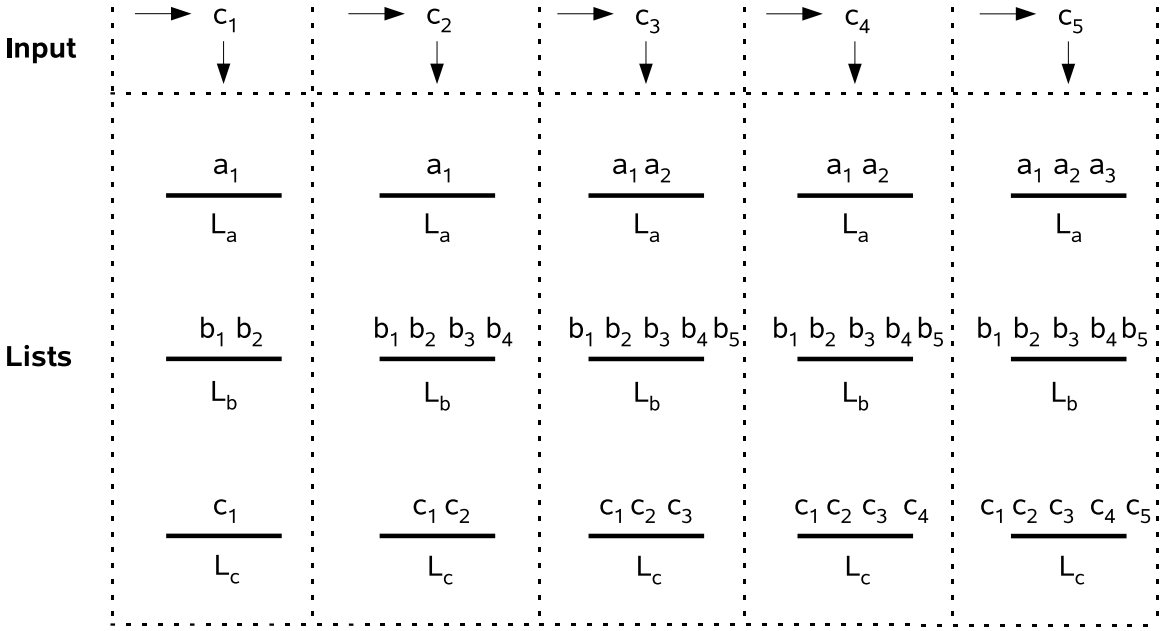
```
1  begin
2      AncestorBuilder b ← AncestorBuilder(D, L, R, P, E); // create a per-node ancestor builder
3      while b.hasNext() do
4          Node n ← b.getCurrentNode(); // get the current ancestor
5          List l ← b.getCurrentList(); // get the list on which we store n
6          if l is empty then
7              l.add(n); // simply add n
8          else
9              for each element e in l do
10                 if n = e then
11                     break; // node and its ancestors already on the stack
12                 end
13                 if n is ancestor of e then
14                     l.addBefore(n, e);
15                 end
16             end
17             if n has not been added so far then
18                 l.add(n); // add at the end of the list
19             end
20         end
21     end
22 end
```

With the relaxed version of the ancestor tuple builder in mind, we can now discuss the implementation of the cursor interface in Listing 8.18. The *setToFirst* method is quite simple: It calls *open* on the ancestor tuple builder. As we have seen, the ATB fetches the first elements from the index cursor and initializes its stacks with the ancestors. When the builder is opened, we fetch the list responsible for this cursor from the ATB. If this list is not empty, the `currentID` member field can be initialized with the first element in the list. Otherwise, the index delivered no results for this cursor and it is set to *consumed*.

The implementation of *forwardTo* in Listing 8.18 also fetches its responsible list l from the ATB. If the position to be forwarded to is larger than the last element on the list, we need to trigger the ATB to produce new list entries. This happens in line 8. If *forwardTo* was called on the bottom-most cursor in the path (e. g., on C_3 in our example), we know from the TwigOpt algorithm that all other cursors above (C_1 and C_2) have already been advanced "over" the position provided. Therefore, it is safe to remove all elements from all lists that are smaller than the last element on the bottom-most list. This happens during the call to *clearAllLists* in line 5.

Every cursor maintains a variable `startPos` that points to a position in the cursor's ATB list, from which the cursor starts to scan upon a call to *forwardTo*. Note, this variable is required, because list maintenance in the ATB has been relaxed. When a the lists are cleared by a call to *clearAllLists*, all cursors need to be set to position 0. This happens by calling *updateCursorPositionsOnAllCursors* in line 6.

Figure 8.12 A sample run on the relaxed ancestor tuple builder (with the input from Figure 8.10)



When the ATB advanced the cursor's list to contain the right range, we can search the list for the first element larger (or equal) to the given `position` DeweyID. If no such position can be found, the cursor is consumed. Otherwise, the `currentID` variable is set and we remember the list position, where we found that element in `startPos` such that the next call to *forwardTo* will avoid scanning all elements before `startPos` again.

8.5.3 Index Embedding Considerations

In Section 8.4.2, we have discussed, how a logical twig is mapped onto a physical one. During this mapping, the twig cursors remain undefined (with the exception of virtual cursors and the document root node cursor). In the second plan generation phase, when the operators are stitched together, also the twig cursors will be defined. However, as explained, we allow to pre-define certain cursors. This possibility is exploited to integrate path indexes into the twig operator. Therefore, the `ATBinput` and the `ATBcursor` have to be instantiated appropriately for each index to be integrated and for each cursor required.

To employ CAS/path index embedding, the plan generator analyzes the twig structure and derives the necessary path information. To do so, it traverses the twig in post-order and, for each node n in the twig, the "incoming" path pattern of n is derived. Note, being a heuristics, the post-order traversal prefers long paths. The derived path pattern is then checked against the metadata to retrieve appropriate CAS or path indexes, as described in Section 7.3.7. As a further heuristics, when a content predicate exists, a CAS index is always preferred to a path index (for the same path), because the CAS index is likely to be more selective.

Listing 8.18 The *setToFirst* method and the *forwardTo* method

Local variables:
`ATBinput atbInput` as I;
`int index` as i;

a) *ATBcursor.setToFirst*

```
1 begin
2 |  I.open(); // open and initialize the ancestor tuple builder
3 |  List l ← I.getLists().get(i); // get the atb list responsible for this cursor
4 |  if l is not empty then
5 |  |  setCurrentID(l[0]); // fetch first element from the list
6 |  else
7 |  |  setIsConsumed(true);
8 |  end
9 end
```

Parameters:
`DeweyID position` as p;

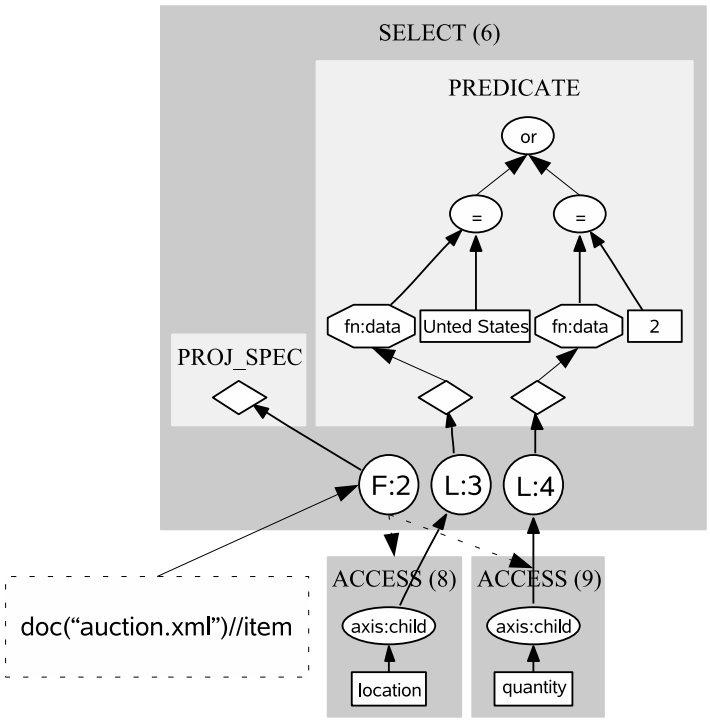
Local variables:
`ATBinput atbInput` as I;
`int cursorID` as i;
`int startPos` as k;

b) *ATBcursor.forwardTo*

```
1 begin
2 |  List l ← I.getLists().get(i);
3 |  if p.compareTo(l.getLast()) > 0 then
4 |  |  if l is last list then //this cursor is the bottom-most cursor
5 |  |  |  I.clearAllLists(l.getLast()); // remove all elements from all lists up to l.getLast()
6 |  |  |  I.updateNextPositionOnAllCursors(0); // set nextPosition = 0 on all cursors
7 |  |  end
8 |  |  I.processTo(i, p); // advance atb on the ith list
9 |  end
10 |  for int j ← k to |l| do // for each element in list l
11 |  |  DeweyID d ← l[j];
12 |  |  boolean f ← false;
13 |  |  if this cursor is virtual then
14 |  |  |  f ← d.compareTo(p) >= 0;
15 |  |  else
16 |  |  |  f ← d.compareTo(p) > 0;
17 |  |  end
18 |  |  if f then
19 |  |  |  setCurrentID(d);
20 |  |  |  k ← j;
21 |  |  |  return;
22 |  |  end
23 |  end
24 |  // no ID found → consumed
25 |  setIsConsumed(true);
26 end
```

A problem occurs when a node in the path pattern has an optional incoming edge or when a node is connected to an *or* node. As an example, again consider the query depicted in Figure 8.5a on Page 232. The twig node contributing *mail* nodes has an optional edge. If we would employ a path index *I(//item//mail)* to retrieve *item* elements, then we would probably skip some, because an *item* element does not necessarily have to contain a *mail* element. To avoid false negatives, we, therefore, do not allow a complete index embedding for these paths. Rather, only subpaths can be answered by indexes. As an example, consider path `//a/b[c/d or e/f]`. In this case, we can neither employ an index I_1*(//a/b/c/d)* nor an index I_2*(//a/b/e/f)* to calculate the inner a and b nodes, because we might skip some of them. However,

Figure 8.13 Motivation for a lazy tuple generator



I_1 could be used to deliver c and d nodes and I_2 could deliver e and f nodes (because all necessary nodes to match the above query are contained in the indexes). Furthermore, if an index I_3*(//a/b)* is available, we can also use it for this query.

This circumstance is also reflected in the plan shown in Figure 8.9c. The input of the twig node contributing *mail* elements is a path index, however, this path index is not integrated into the twig operator (and has been created by a mapping of the *mail* sequence operator to a `PathIndexScanOperator` as described above).

As we will see in Chapter 9, embedding path indexes as described here is very effective, when we can avoid to fetch inner elements from the document or from other indexes (e. g., from the element index). We now finish the discussion on path processing operators and provide a glimpse on how the remaining operators of the physical algebra are implemented.

8.6 LAL Operators as PAL Operators

This chapter mainly introduced the necessary path processing operators of XTC's physical algebra. For brevity, the remaining operators will only be summarized. This is acceptable because their implementation is mostly quite similar to their algorithms given in the logical algebra section. Therefore, we will only highlight the differences.

8.6.1 Lazy Tuple Generation

In the logical algebra, the *TUPGEN* operator is responsible to create the Cartesian product of multiple input streams. Figure 8.13 shows an example for query `doc("auction.xml")//item[location="United States" or quantity=2]`. Let us assume that this nested version of the query shall be evaluated. Then the tuple generator instantiated for the select operator would evaluate `doc("auction.xml")//item` and, for every *item* returned, *both* subqueries returning *location* and *quantity* elements would be executed. Let us assume that for *item* i, *location* elements l_1 and l_2 are returned, and that i has one *quantity* child q. Then, the tuple generator would assemble the following tuple for i: [i, <l_1, l_2>, q]. This tuple is then passed to the physical select operator generated for the predicate. If we assume that q already value 2, then the predicate would evaluate to *true* and, regardless of the values of l_1 and l_2, i will be passed on. Therefore, we have fetched l_1 and l_2 in vain.

To avoid this situation, we implemented a so-called *lazy* tuple generator, which produces information for following selection, projection, and sorting operators only when the information is really required. We omit the details on how this is technically accomplished for brevity.

8.6.2 The Merge Operator

A sample application of the merge operator can be found in Figure 5.7 on Page 109. In this example, the merge operator receives three input streams, one containing *item* nodes, and two containing *item/location* and *item/quantity* tuples. The merge operator finds all tuples with a common *item* node and assembles an intermediate tuple capturing all input information.

In section 5.7.2 on Page 112, we have defined the logical merge operator as a selection over the Cartesian product, where the selection predicate was based on node identity (using XQuery's *is* expression) to find the matching tuples in the input stream. Of course, a physical implementation based on a Cartesian product would not be efficient.

Therefore, for the implementation of the merge operator, we exploit the fact that the input stream S_1 delivered by the *split* operator contains a superset of the nodes to be matched in the other input streams and every node in S_1 is unique. With this knowledge, the merge operator becomes quite simple. For every node in S_1, we have to find the corresponding tuples in the other streams and create an output tuple. We can further simplify the process by passing integer IDs instead of XML nodes to implement the matching. Thereby, the equality check becomes more efficient.

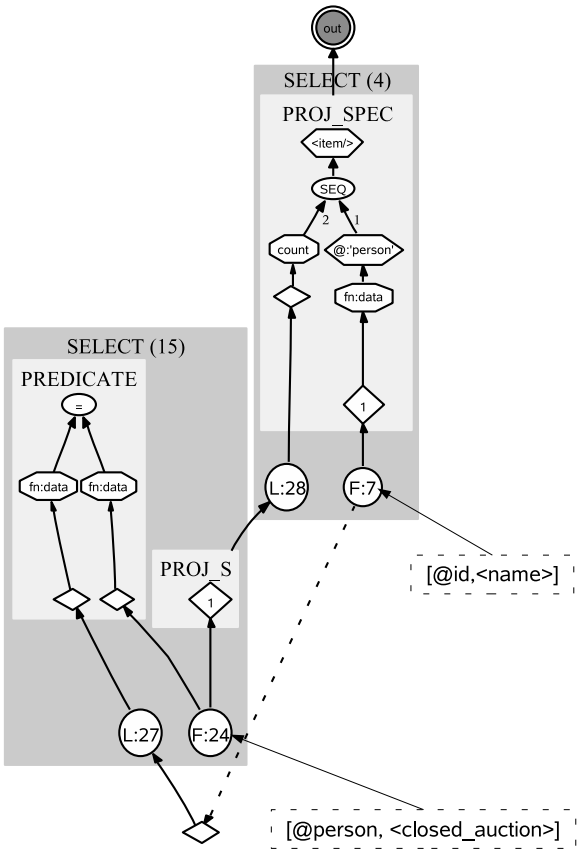
8.6.3 Value-Based Joins in XQuery

As relational algebra, XQuery allows to express value-based joins. However, in contrast to relational algebra, XQuery joins are ordered and have existential semantics. As an example, consider query Q8 from the XMark benchmark [Schmidt 02] shown in Figure 8.14. The query essentially contains two twigs (not shown). One twig returns a sequence of tuples, where the first tuple field contains an *id* attribute and a sequence of *name* nodes. The second twig delivers a tuple sequence,

Figure 8.14 An XQGM instance with a value-based join

Sample XMark Query Q8:

```
let $auction := doc("auction.xml")
return
  for $p in $auction/site/people/person
  let $a :=
    for $t in
      $auction/site/
          closed_auctions/closed_auction
    where $t/buyer/@person = $p/@id
    return $t
  return
    <item person="{$p/name/text()}">
      {count($a)}
    </item>
```



where the first field is a *person* attribute and the second field contains a sequence of *open_auction* nodes. For each *id*/*name* tuple, `select(4)` calls `select(15)`. Operator `select(15)` evaluates the second twig to receive the *person*/*open_auction* sequence. Between *id* and *person*, a value-based join is computed. From every *person* tuple that matches, the *open_auction* sequence is passed to `select(4)`, which constructs a result.

Note, the sketched evaluation mode implies a nested loop join (i. e., for every *id* tuple, the subtree below F:24 is evaluated). However, because the subquery below F:24 is independent of any correlated input, it always returns the same result. Therefore, we can save some work by evaluating the subquery only once. Furthermore, to support the comparison operation, we can maintain all *person* values in a hash table and simply probe the values of the *id* attributes. Essentially, the operator implementing `select(15)` is a hash join. A similar implementation for non-equality predicates is provided by a merge-join. We do not explicitly give the necessary algorithms and content ourselves with the given example.

8.6.4 The Remaining Operators

The following operators are implemented as they have been discussed in the logical algebra: selection, projection, sorting, the attachment of the context position/size, the *ddo* function, external tuple variable references, the root operators well as the set-

based operators, group by, and unnest. The only difference is that all these operators implement the *open-next-close* protocol in the physical algebra (not shown in the logical representation). Of course, node access operators, structural joins, and the twig join operator are implemented as introduced in this chapter.

We now conclude the main contribution of this work and consider related work on path processing operators and the physical algebra. After that, we will assess the concepts developed in our chapter on experimental results.

8.7 Related Work

Path processing operators have been the main contribution of this chapter. Therefore, we will start by discussing various PPO approaches from the literature. In particular, we will look at structural joins and holistic twig joins. After that, we will consider the physical algebras and the plan generation capabilities of the five competitor systems.

8.7.1 Navigational Primitives

Navigation steps are essential to the XQuery language. Therefore, every system that is able to answer XML queries can execute navigation. In native XML DBMSs, these navigations often are mapped to some basic access primitives (similar to our single-node navigational PPOs). In XDBMSs implemented over relational storage engines (e. g., XQuery/MonetDB), navigational have to be calculated by (structural) joins, because the underlying access system is unaware of document hierarchy. However, as we have seen, structural joins can also be applied in native systems. We will discuss them in the next section.

When implementing query processing over navigational primitives, one has to watch out to avoid doing the same work over and over again. We have addressed this issue in Section 8.2.2, where we developed an algorithm based on navigational primitives that filters its input list to avoid redundant navigation steps. Another important requirement is that navigational algorithms deliver a duplicate-free result sequence because, otherwise, intermediate results might grow exponentially. NavTree also incorporates a solution for this problem. You might have noticed that these considerations are quite basic. As a result, similar ideas can be found in quite a number of publications, e. g., [Helmer 02, Grust 03b, Hidders 04, Gottlob 05, Mathis 06b].

8.7.2 Structural Joins

With the StackTree algorithm, [Al-khalifa 02] proposed the one of the first structural join algorithm. It is capable of finding pairs of structural matches in two node input streams. The result is delivered in root-to-leaf order or leaf-to-root order. In this work, we extended this algorithm to support outer joins and semi joins. After the first proposal, quite many other proposals followed:

- [Chien 02] have explored how indexes can be used to avoid scanning. The original StackTree algorithm did not employ any indexes and scanned the complete input sequences. The indexes considered in this work were similar to our ele-

ment indexes. Thus, no path/CAS indexes have been studied. One disadvantage of the proposed algorithm is that it cannot be applied, when at least one input sequence is not indexed. This situation might occur, when the plan generator produces bushy query plans (based on structural joins).

- [Vagena 05] have incorporated the ability to evaluate positional predicates into a structural join algorithm. They furthermore extended their algorithm to evaluate positional queries on the *following-sibling* axis. The authors claim that their approach can easily be extended to all XPath axes. Note, similar techniques to evaluate positional predicates have been embedded into our twig join algorithm.
- [Mathis 06a] have studied hash-based structural join algorithms. The hash-based joins do not expect sorted input streams. Therefore, sorting operations can be avoided. Furthermore, they allow to compute the sibling axes (additional to *ancestor/descendant* and *parent/child*). A disadvantage of the approach is the costly computation of the hash table. Therefore, the approach can only be applied efficiently, when the size of the input streams is substantially differing.
- [Li 03] and [Vagena 04] explored data partitioning to avoid sort operations on both input streams. However, the algorithm of [Li 03] needs to sort at least one input, while the proposal of [Vagena 04] builds an in-memory data structure, whose construction time lies in $O(n \log n)$.
- The staircase join operator [Grust 03b] is also a structural join algorithm, used in the relational XQuery/MonetDB system. It joins a table of context nodes with the document table. The algorithm can "detect" ranges, where no matches will be found. These ranges are then skipped. Furthermore, it produces an output in document order. The skipping feature of the algorithm rests on a range-based labeling scheme. The algorithm is optimized for main-memory database systems (where the costs to retrieve a node are $O(1)$). Note, the staircase join algorithm is conceptually similar to our NavTree algorithm. Like NavTree, it cannot be used to implement a general structural join (required in bushy query plans).

8.7.3 Holistic Twig Joins

Similar to the history of structural join algorithms, after the publication of the first twig join algorithm, many others followed. The first algorithm—TwigStack—was proposed by [Bruno 02]. TwigStack is a two-phase algorithm. In the first phase, all sub-paths are matched over the input streams (only the *descendant* axis is supported). In the second phase, these path matches are merged into the complete result. Note, to implement the *child* axis, a separate *child* check is required during the result construction phase. The result consists of a forest of trees, where each tree node corresponds to a twig node (it was matched for). Internally, the TwigStack algorithm maintains a list of stacks (one for each twig node). The stack protocol requires that every element on the stack is the ancestor of the element above it. Additionally, for every stack position, the StackTree algorithm maintains a pointer to the stack of the parent node, indicating where the descendants of this parent node start. To speed up the matching process, [Bruno 02] suggested the XB-tree, a B-tree-like index structure for the storage of node-label ranges. Let's take a look at other twig join approaches:

- The PathStack algorithm was also proposed by [Bruno 02]. It extends the binary structural join operator to an n-way operator capable of matching linear path

patterns (i. e., path with no branching predicates). Note, PathStack was just a "transitional" algorithm to introduce TwigStack. Its implementation is, therefore, quite similar.

- PathStack¬ by [Jiao 05] extends the original PathStack algorithm to directly evaluate *not*-based path predicates. All other characteristics of the PathStack algorithm are inherited.
- TwigStackList by [Lu 04] was the first algorithm supporting the evaluation of the *child* axis in the matching phase. On structurally complex documents, it can, therefore, reduce the number of intermediate results (generated by the other operators due to a *descendant* match). Its implementation is similar to TwigStack: each twig node has a stack and the stack protocol allows descendants to be stored on a stack. Additionally, each twig node has a list where it can store a certain look-ahead of nodes delivered by the input cursor. This look-ahead can be used to decide the *child* axis[3].
- Twig2Stack by [Chen 06] is one of the first algorithms capable of matching optional axes and projection (i. e., so-called *generalized tree patterns* [Chen 03c]). The algorithm manages the internal result on a set of stacks. However, in contrast to the StackTree algorithm, one twig node can manage *multiple* stacks. The set of stacks captures the hierarchy of the matched elements. The stack management constantly requires to create and delete stacks. The algorithm is, furthermore, the first one-phase twig-matching algorithm. This means that the output is not calculated by merging paths, but directly from the stacks. This is also the case in our algorithm. By relaxing the stack protocol, our algorithm can, furthermore, compute generalized tree patterns (and even more semantics) on a stable set of stacks in a much simpler way.
- TwigList by [Qin 07] is a simplification of the Twig2Stack algorithm. It operates on lists instead of stacks to maintain the internal state. As our approach, it relaxes the stack protocol and is a one-phase algorithm. Furthermore, it supports projection, but no *child* edges.
- TwigStackList¬ by [Yu 06] is a combination of PathStack¬ and the TwigStack algorithm. It can evaluate *not* predicates directly in the matching phase.
- TJFast by [Lu 05] aims at the reduction of elements to be read in the matching phase. For its implementation, the algorithm relies on a so-called *extended dewey labeling* mechanism that encodes node names into DeweyIDs (without destroying the salient features of DeweyIDs). The drawback is that some kind of schema is required to create these extended DeweyIDs and that, upon changes to this schema, the IDs do not remain stable (a major drawback in real XML DBMSs). However, the basic idea behind TJFast is quite similar to our embedded indexes. Only the leaf nodes of a twig have some input cursors. Internal elements are recomputed rather than fetched from the document.
- iTwigJoin [Chen 05] is an extension of the TwigStack algorithm which can operate on streams generated by structural indexes. For its implementation, it assumes a special cursor delivering nodes labeled with their prefix path. The iTwigJoin operator is a two-phase algorithm. Note, the paper makes no assumptions on

[3] Interestingly, the *child* axis is harder to match, because, compared to the *descendant* axis, *child* imposes another requirement: for nodes u and v to match, v has to be a descendant of u *and* its level has to be larger by 1.

where these labels come from. The approach is conceptually similar to our index-based twig evaluation technique. However, because it does not rest on the salient features of DeweyIDs (but on a range-based labeling scheme), it cannot compute the elements for inner twig nodes (as our ATB), but has to scan them from a path index.

- TSGeneric+ [Jiang 03b] assumes the existence of a so-called XR-tree index [Jiang 03a], which encodes the document in a way such that the algorithm can skip regions of the document, where no matches will be found. Its internal *forwardBeyond* and *forwardToAncestorOf* methods implement skipping and can be seen as the counterparts of the *forwardTo* method of the TwigOpt algorithm. Therefore, we consider the TSGeneric+ algorithm as the predecessor of TwigOptimal.
- Finally, this work extended the TwigOptimal algorithm which was originally proposed by [Fontoura 05].

Table 8.1 summarizes the functionality of the various twig operators introduced. If a table cell is left blank, the concept is either not possible to integrate or the authors

Table 8.1 Comparison of various holistic twig join algorithms

Algorithm	descendant	child	and	or	not	optional edges	projection	grouping	expressions	filters	pos. predicates	phases	element indexes	path indexes
PathStack [Bruno 02]	✓											–		
PathStack¬ [Jiao 05]	✓				✓							–		
TwigStack [Bruno 02]	✓		✓									2	✓[1]	
TwigStackList [Lu 04]	✓	✓	✓									2		
TwigStackList¬ [Yu 06]	✓	✓	✓		✓							2		
TJFast [Lu 05]	✓	✓	✓									2		✓[2]
iTwigJoin [Chen 05]	✓	✓	✓									2		✓[3]
TSGeneric+ [Jiang 03b]	✓		✓									2	✓[4]	
Twig2Stack [Chen 06]	✓		✓			✓	✓	✓				1		
TwigList [Qin 07]	✓		✓				✓					1		
TwigOpt. [Fontoura 05]	✓		✓	✓			✓					1	✓	
Ext. TwigOpt (our work)	✓	✓[5]	✓	✓	✓[5]	✓	✓	✓	✓	✓[5]	✓	1	✓	✓[6]

1. Skipping in TwigStack supported only by XB-Tree.
2. TJFast requires special embedding of path information into DeweyIDs.
3. iTwigJoin supports streams generated by path indexes, however, no internal element reconstruction.
4. TSGeneric+ relies on the special XR-tree.
5. Matching *child*/*not*/*filter* integrated in output generation (and not in matching phase).
6. Index embedding with ATB only possible, when DeweyIDs are indexed.

did not publish how an integration would look like. Of course, all operators support the *descendant* axis. The *child* axis is not embedded into the matching process of our extended TwigOpt algorithm. An extension similar to the TwigStackList approach utilizing a look-ahead buffer is, nevertheless, possible. The same is true for the *not* function. In the current implementation, this function is evaluated as a filter (on the already matched intermediate result). However, the function can be embedded into the matching process by redefining the *solution extension* concept.

Essentially, our extended TwigOpt algorithm combines all advantages of the other algorithms into one flexible operator. Extended TwigOpt runs in a scan-and-skip fashion on element indexes (like TwigStack and TSGeneric+), can embed index results like TJFast and iTwigJoin, is able to evaluate optional edges and grouping like Twig2Stack, and, furthermore, provides for positional predicates, filters, and embedded output expressions. The keys to this functionality are 1) the simple cursor interface borrowed from the TwigOpt operator, which flexibly allows to integrate streams from different data sources; and 2) the internal stack protocol, which enables the algorithm to manipulate the intermediate result.

8.7.4 A Glimpse on Physical Algebras in Other XML Query Processors

To conclude this chapter, we take a brief look at the implementation of path processing operators in our five competing XML query systems.

Galax

Galax loads the complete document as a DOM tree into main memory before processing. On this tree, the basic access primitives are navigations. However, for a comparison, [Michiels 07] also implemented the staircase join operator and the TwigStack algorithm. In their experiments, both operators ran on main-memory data structures, thus, making the results of this experimental approach questionable. On the physical algebra of the Galax system, no complete publications exist. [Ré 06], however, report a hash-based algorithm for the calculation of value-based joins (that adheres to the special existential XQuery semantics on general comparisons).

Timber

Timber's physical algebra has been described by [Paparizos 02]. The operators there are more or less a 1-to-1 implementation of the logical operators. Therefore, also Timber's physical algebra does not clarify how non-tree axes (e. g., the *next-sibling* axis) can be evaluated. Furthermore, the authors do not state how their *structural join* (which is actually an n-way join) is implemented. Because the Timber group was involved in the development of the twig join operator, we can assume that behind the physical *structural join* a twig join algorithm does its work. On the other hand, [Chen 03c] show how structural joins are used to implement generalized twig patterns. No further details on alternative algorithms are published in the Timber context. However, a paper on structural join reordering has been published by [Wu 03].

DB2 pureXML

As we have seen in Section 7.4.5, DB2 pureXML allows to define CAS indexes. These indexes can be created over XML-typed relational columns. During query processing, they are used to prune the set of documents to be searched. Index applicability is decided with the help of the framework developed by [Balmin 04]. To find matching indexes, the query processor has to match the query path pattern against all index definition patterns. This process is naturally more complex than our simple PCR hash-lookup for index matching.

When an index has been matched, its integration into a query plan is implemented by a so-called XISCAN operator (i. e., by an index scan algorithm). Several XISCAN operators can be combined by a special XANDOR operator to combine CAS index usage. According to [Balmin 06], the XANDOR operator is conceptually similar to a holistic twig join. The resulting list of XML nodes (retrieved by XISCAN and combined by XANDOR) can then be used as a context set of starting points for the XSCAN operator. Basically, the XSCAN operator streams through each document and evaluates path expressions. Because the DB2 indexes can contain false positives, the path patterns used to access the indexes in the first place are again evaluated by the XSCAN operator to remove these false positives. Note, this is not necessary in our approach, because the results of the indexes can be directly materialized or processed by further operators without a verification against the document. How query plans look like, when no indexes can be applied, is not described by [Balmin 06]. However, a positive point is that DB2 pureXML has a cost model and a cardinality estimation component. It is, therefore, the only system discussed here, which is able to execute cost-based plan generation.

Natix

The basic access primitives in Natix are navigations over the document store [Helmer 02]. [May 06a] have compared navigational access with index-based structural joins. It is, however, not clear whether Natix is able to generate plans with structural joins or whether these operators were just implemented for comparison. Some optimization for in-memory grouping have been published [May 05]. However, sophisticated access operators, like the holistic twig join or path and CAS indexes, are not available in the system's physical algebra.

XQuery/MonetDB

As we have seen, XQuery/MonetDB consists of the Pathfinder XQuery frontend [Boncz 05b] and the MonetDB relational main-memory DBMS as backend. Therefore, Pathfinder compiles XQuery into relational algebra. To inject tree-awareness into the relational algebra, they proposed the staircase join operator [Grust 03b], which was classified in Section 8.7.2 as a structural join operator. Because the backend is main-memory-based, no path/CAS index structures or more complex twig join algorithms were proposed. The authors of the system frequently state that pathfinder can generate efficient query plans for any relational DBMS. However, their optimizations are often tightly coupled to the specifics of the MonetDB system (for an example, consider the work of [Boncz 06b] which discusses updates). [Grust 07] have shown some results using IBM's DB2. However, the queries used in their experimental results are simplest path queries that make no conclusions on more complex queries possible.

In our opinion, the major problem of the Pathfinder approach when implemented on other DBMSs than MonetDB is the scan-and-skip fashion of the staircase join operator. As we will see in our experiments with the similar NavTree operator, scan-and-skip imply frequent index open/close operations.

8.8 Summary

This chapter concludes the main contribution of this thesis. It introduced the basic path processing algorithms in the form of navigational, join-based, and index-based operators to XTC physical XML algebra. The remaining non-PPOs are more or less direct "translations" of the algorithms presented in Chapter 4 and 5 into the ONC protocol. To ensure a meaningful integration of holistic twig joins into the physical algebra, we have extended the TwigOptimal algorithm to support a quite wide range of features such as optional edges, positional predicates, output expressions, filters, grouping etc. Furthermore, we have shown how the result of path/CAS indexes can be embedded via an ancestor tuple builder into the join algorithm, thus, making index intersection possible. To the best of our knowledge, the resulting extended TwigOpt algorithm is the most expressive (w. r. t. functionality) and flexible (w. r. t. input sources) algorithm published so far.

Of the five competitor systems, only DB2 provides for similar indexing techniques, and only DB2 and (possibly) Timber[4] can generate plans based on twig joins. However, in DB2 indexes are not as tightly integrated as in our system, making a re-evaluation of already matched paths on the document necessary. In the next chapter, we will assess XTC's query processor in an experimental evaluation.

[4] ... we are not quite sure on this point.

Part IV

Experimental Evaluation and Future Research

Chapter 9 Experimental Results

> My experiments did not turn out quite like yours Henry. But science, like love, has her little surprises—as you shall see.
>
> Dr. Septimus Pretorius

In this chapter, we empirically assessed and compared the concepts developed in this thesis. In our experiments, we expored the proposed document storage and low-level XML processing techniques (i. e., scan and DOM operations). To analyze our query processing approach, we regarded different types of query plans (i. e., nested vs. unnested plans), physical operators (structural joins vs. twig joins), and physical database layouts (i. e., XML indexing strategies). Before discussing these issues, however, we explain the experimental setup.

9.1 Experimental Setup

The XTC server ran on a host system to which we connected with the help of a benchmark client. The client stored/retrieved documents and issued queries. As we have discussed, the query compiler of the XTC system can be configured by a set of simplification, rewriting, and plan generation rules. These rules are declared in an XML file which is read by the XTC server during system startup.

The host system had an Intel XEON quad core (3350) 2,66GHz CPU with 4096 MB of DDR2 RAM and a 500GB SATA II disk connected via RAID III. On the host was equipped with Ubuntu Linux (kernel "2.6.24-16-server") and a Java Runtime Environment (version "1.6.0_07").

XTC was started with an initial main-memory size of 265 MB and was allowed to consume up to 1200 MB. XTC is a multi-threaded server. However, although the server runs on a quad-core machine, the query processor used only one thread to evaluate a query. No intra-query parallelization is implemented so far. The database buffer was configured with 250 8K buffer pages. Performance timings were recorded on the server side. This means that the resulting numbers do not contain network time. All queries were executed on a cold buffer and have been repeated 35 times. As an exception, we measured extremely long running queries (i. e., queries that ran longer than one minute) only once. When, the compiler configuration or the physical layout of the database changed during a benchmark, the server was restarted. In the following query processing benchmarks (Section

9.3), we configured the query engine with many different compiler configurations. Therefore, many different query plans were generated. To facilitate the understanding of the plans, we executed all queries, collected the XQGM instances and the physical plans, and published them online [Mathis 09].

9.2 Document Processing

In our first set of tests, we assessed document storage and low-level operations, such as scanning and navigation. For the benchmark, we used the documents from Table 6.1 on Page 164, of which the first five are real-world documents taken from [Miklau 09] and of which the last one is a synthetic XMark document [Schmidt 02] with a size of 100 MB.

9.2.1 Space Consumption

First, we measured the space consumption of the XML documents. Figure 9.1a presents the consumption in the external format (i. e., serialized as a text file on disk), in the node-oriented format (i. e., with inner structure), and in the path-oriented format (i. e., with virtualized inner structure). Figure 9.1b shows the relative space consumption of the node-oriented and the path-oriented formats w. r. t. the external format. As you can see, the both internal formats required substantially less space than the external format. Furthermore, in all cases, the path-oriented alternative could further reduce the space consumption, even for the highly irregular Treebank document.

Figure 9.1 Space consumption: external vs. node-oriented vs. path-oriented

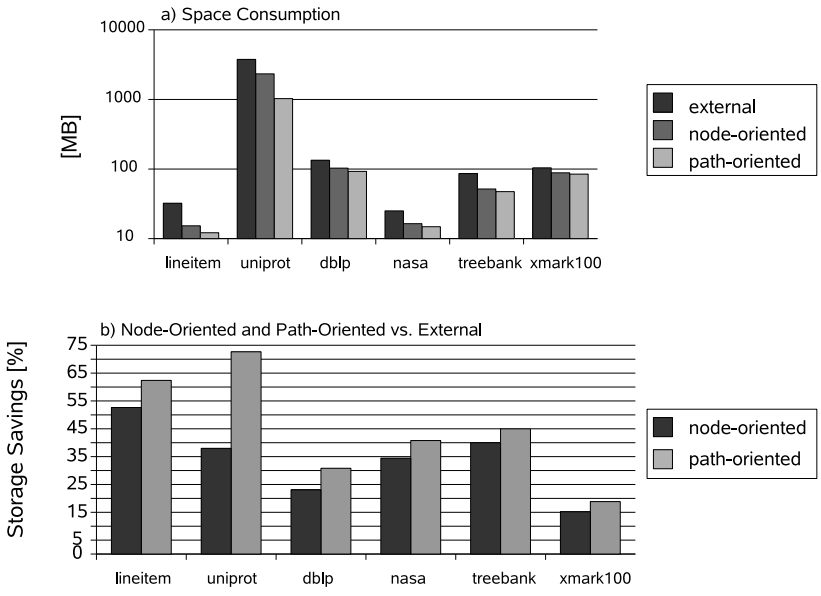
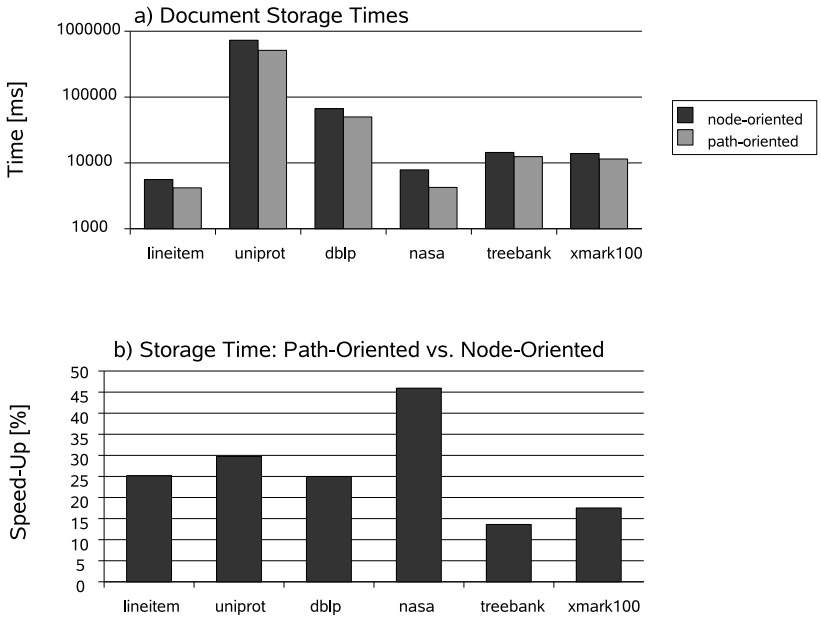


Figure 9.2 Storage time: node-oriented vs. path-oriented



9.2.2 Storage and Reconstruction

In the next test, we measured the storage and reconstruction timings on the tested document collection. In the benchmark, the documents already resided on the host machine such that a transmission over the network could be avoided. For reconstruction, we scanned 1,000,000 nodes of the document and (internally) measured the elapsed time until all nodes were fetched. The results are shown in Figures 9.2 and 9.3 (again with absolute and relative performance figures). With the exception of the XMark reconstruction column in Figure 9.3, the path-oriented approach was always the winner, although we admit that the difference in the reconstruction figure is not that significant. However, we can also state that the path-oriented storage worked well even on the highly irregular Treebank document (which was the only document whose path synopsis did not fit into one page).

9.2.3 Navigation Performance

In the next experiment, we explored the navigation performance resulting from our two storage formats. Therefore, we executed a DOM walk over the document with 100,000 steps and recorded the average time for each of the four navigation directions: first child, last child, previous sibling, and following sibling. The results are shown in Figure 9.4. Again the path-oriented storage variant (PO) was almost always faster than the node-oriented variant (NO). The numbers reflect our worst-case estimations from Section 6.3.4: when h_t is the height of the B*-tree of the document container, then NO-based navigation requires $2 * h_t + 1$ page access operations in the worst-case, whereas the PO-based navigation requires only $h_t + 1$.

To summarize, we can say that the performance and space-consumption differences are document-dependent (e. g., Lineitem vs. XMark) and, furthermore, in most doc-

Figure 9.3 Reconstruction time: node-oriented vs. path-oriented

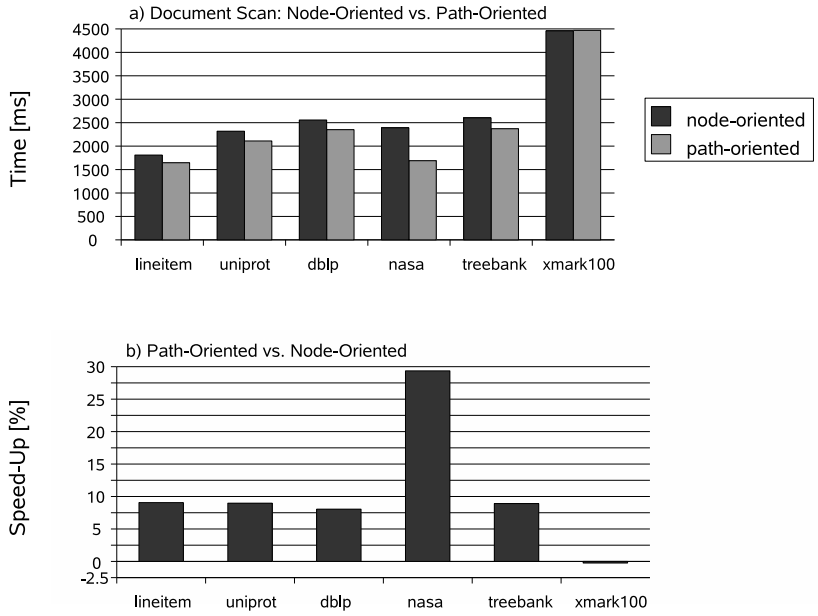
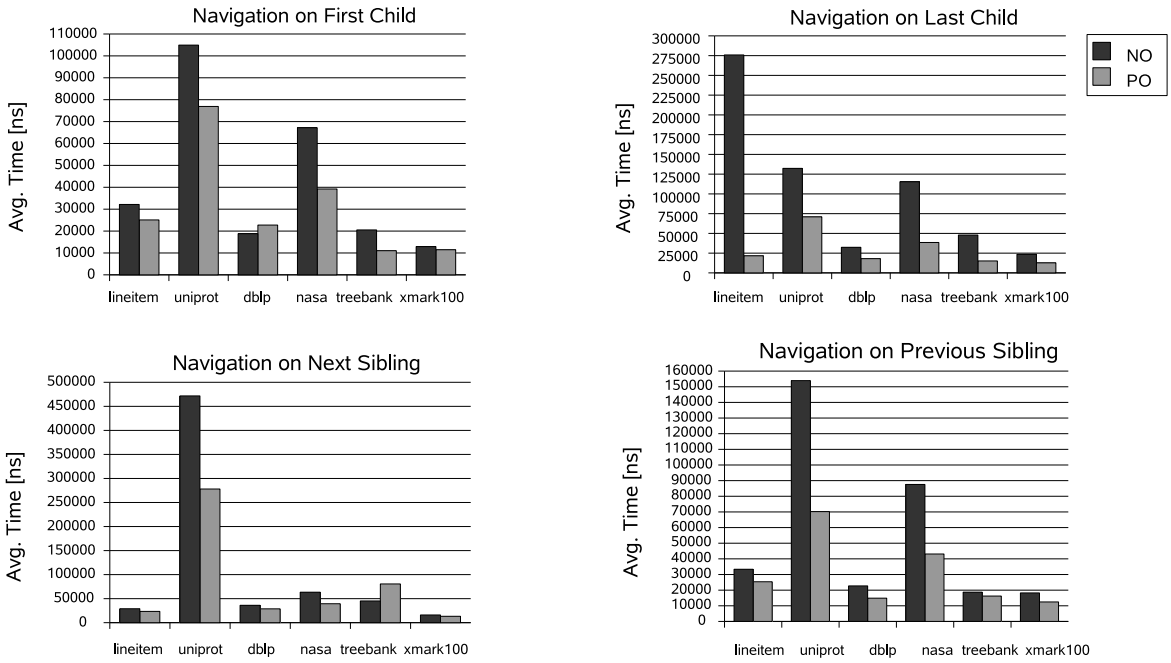


Figure 9.4 Navigation time: node-oriented vs. path-oriented



uments tested, the path-oriented implementation had advantages (even on structurally complex documents). In the next section, we will examine the query processor and the physical operators that work on the database layout.

9.3 Path Processing Operators and Query Plans

Physical operators have been discussed in the previous chapter. In this section, we want to assess the various PPOs introduced, in particular navigational, join-based, and index-based path matching operators. To do so, we configured the query processor to generate plans that make use of a certain PPO as much as possible. We also explored different implementations of query plans (i. e., nested plans vs. unnested plans).

9.3.1 Navigational PPOs

The following benchmark was executed on a 10MB XMark document. We defined two setups: In the basic setup, the document was not indexed at all. Thus, all operators had to navigate or stream through the document. In the *indexed* setup, a plain element index (without further path information) existed. For the evaluation of the queries in the following tests, we configured the query processor to generate plans based on navigational and on join-based primitives. The following scenarios were explored:

- The plan remained nested.

  N1) Node access operators were mapped to single-node *navigational* access algorithms (`AxisStepNavigationalOperator`).

Listing 9.1 Path queries (partly drawn from the XPathMark benchmark)

```
q1:  //site/people/person/name
q2:  //site//people//person[.//name][.//age]//income
q3:  //text[bold]/MPH/keyword
q4:  //listitem[.//bold]/text//emph
q5:  //listitem[.//bold]/text[.//emph]/keyword
q6:  /site/closed_auctions/closed_auction/annotation/description/text/keyword
q7:  //closed_auction//keyword
q8:  /site/closed_auctions/closed_auction//keyword
q9:  /site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
q10: /site/closed_auctions/closed_auction[descendant::keyword]/date
q11: /site/people/person[profile/gender and profile/age]/name
q12: /site/people/person[phone or homepage]/name
q13: /site/people/person
          [address and (phone or homepage) and (creditcard or profile)]/name
q14: //person[profile/@income]/name
q15: /site/people/person[profile/age >= 18 and profile/@income < 10000
          and address/city != 'Dallas']/name
q16: /site/open_auctions/open_auction[bidder/increase = current]/interval
q17: /site/open_auctions/open_auction[(count(bidder) mod 2) = 0]/interval
q18: (count(//text) + count(//bold) + count(//emph) + count(//keyword))
q19: /site/open_auctions/open_auction[sum(bidder/increase) > 10 * initial]/interval
q20: /site/open_auctions/open_auction
          [sum(bidder/increase) != (current - initial)]/interval
q21: /site/regions/europe/item/description/descendant::keyword[last()]
```

N2) Node access operators were mapped to single-node *scan-based* algorithms on the document store (`AxisStepDocScanOperator`).

N3) Node access operators were mapped onto the single-node *scan* algorithms on the element index (`AxisStepElIdxScanOperator`).

- The plan was unnested.

U1) Structural join operators were implemented by the NavTree operator with the *navigational* access algorithm as base implementation (`AxisStepNavigationalOperator`).

U2) Structural join operators were implemented by the NavTree operator with the scan-based access algorithms as base implementation (`AxisStepDocScanOperator`).

U3) Structural join operators were implemented by the NavTree operator with the scan-based access algorithms on the element index as base implementation (`AxisStepElIdxScanOperator`).

U4) Structural join operators were implemented by the StackTree algorithm. The input was delivered by a *shared document scan*.

U5) Structural join operators were implemented by the StackTree algorithm. The input was delivered by (shared) element index scans.

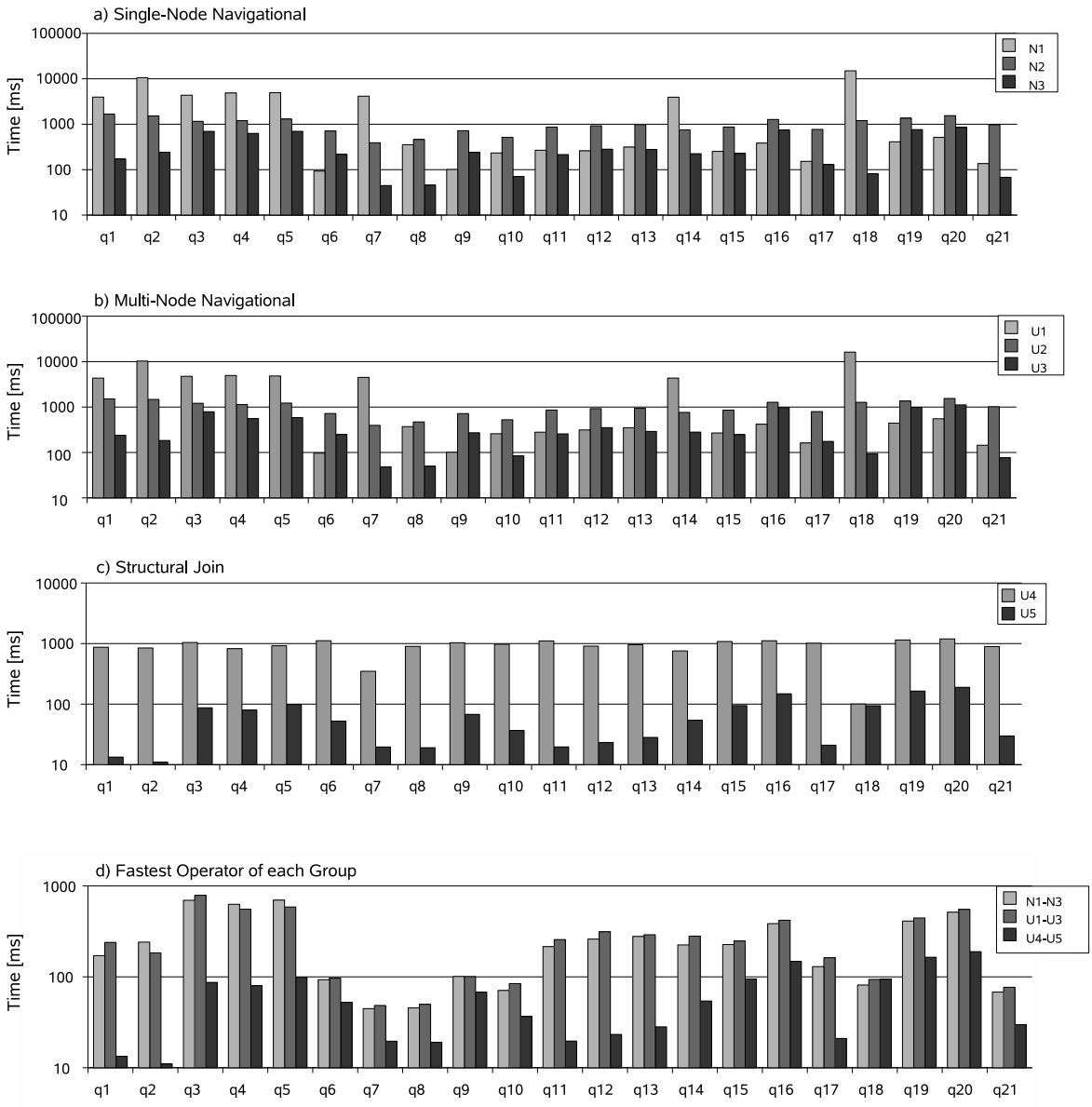
Scenarios N1 to N3 correspond to what the XQuery Formal Semantics specifies as evaluation model. Only the `descendant-or-self` removal rewriting rule (see Section 5.3) was activated. In scenarios U1 to U5, all rewriting rules were applied, with the exception of the twig discovery rule (thus, all structural joins generated during unnesting have to be mapped to physical structural joins). Note, already here, we injected the structural join operator to enable a comparison with join-based query evaluation.

Path Queries

We first consider the query set shown in Listing 9.1. Essentially, it contains a set of path queries over the XMark document. Some of these queries were drawn from the XPathMark benchmark [Franceschet 05]. The result of the benchmark run is shown in Figure 9.5. The first three charts present the run time of groups N1–N3, U1–U3, and U4–U5. The last chart compares the best evaluation strategy of each group. Let us consider Figure 9.5a first. Here, the queries remained nested and we applied a navigational evaluation (N1), a document-scan-based evaluation (N2), and a scan-based evaluation over the element index (N3). The dominating column scheme is that of query `q01`: navigation (N1) is slower than document scan (N2), and this document scan is slower than an element index scan (N3). This scheme holds for queries `q1` to `q5`, `q7`, `q14`, and `q18`. Common to (almost all of) these queries is that they make use of the '`//`' step. In these cases, given a context node, scanning the document or the element index below this context node is more efficient than navigating the subtree. Another colum scheme can be observed for queries `q06`, `q09`, `q19`, and `q20`: Here, the navigational algorithm (N1) is the winner. Common to all these queries is that they define a long path with *child* axes only. Finally, there are situations, when node-wise navigation (N1) is comparable to a node-at-a-time element index scan (N3), e. g., `q11` to `q13`. These queries do also not contain a descendant axis. However, they all have some predicate, which gives the navigational evaluation (N1) a little disadvantage w. r. t. the scan-based mode (N3).

Note, preferring a node-a-a-time document scan (N2) over navigational access (N1),

Figure 9.5 Path evaluation benchmark under scenarios N1–N3 and U1–U5 on a 10MB XMark document



e. g., when an element index is missing, is a good idea in some cases (mainly, when '//' axes are involved; see `q18`). However, the operator should be handled with care, when the axis is selective. Then, it scans the complete subtree, even if only a few nodes will be returned. To decide, which algorithm should be used for a given XQGM node access operator, statistics and cardinality estimations are required.

In Figure 9.5b, the results of the NavTree operator are presented. Here, the queries are unnested (in contrast to Figure 9.5a). The NavTree operator received the previous types of single-node navigational algorithms as base implementation. As you can see, there is no significant difference between the nested (N1–N3) and the unnested (U1–U3) evaluation mode. This essentially means that the NavTree oper-

ator was not able to effectively prune the input set of context nodes during evaluation. Because of the pruning overhead, some queries are even a little bit slower than in the nested version (e. g., U3 vs. N3 on `q18`).

In settings U4 and U5, we used the structural join operator instead of the NavTree operator to evaluate the path queries. Therefore, the queries were unnested by the rewriter. The input to this operator was either given by a document scan (U4) or by an element index scan (U5). As you can see in Figure 9.5c, for all queries, the document-scan-based evaluation required roughly one second, while the element index scan was substantially faster.

The last chart in Figure 9.5c compares the best run times of all three groups with each other. The nested and unnested node-at-a-time evaluation delivers comparable performance results. However, when an element index is available, the (bulk) stream-based evaluation using the StackTree operator beats the node-at-a-time evaluation mode often by at least on order of magnitude (even if the node-at-a-time evaluation also operates on the element index). The rationale is the expensive random access imposed by the node-at-a-time algorithms, where indexes have to be frequently opened/closed to retrieve only a small number of nodes per look-up.

NavTree vs. Nested Evaluation (Reloaded)

Previously, the NavTree operator did not have any advantage over the nested query evaluation. The effect of pruning the sequence of context nodes depends on the query and the shape of the document. To illustrate that the NavTree operator *can* be superior, we constructed queries `nt01` to `nt05` from the following pattern:

```
count(doc("auction.xml")//open_auctions/
    open_auction[position() > $n]/following-sibling::open_auction)
```

Parameter `$n` was set to 0, 250, 500, 750, and 1000 (note, the result returned for `nt01` is 1066). The benchmark was conducted on a 10MB XMark document, where we only considered scenarios N3 and U3 (both operating on the element index). The result is shown in Figure 9.6: The nested version performs significantly worse than the unnested version using the NavTree algorithm. Essentially, the latter one receives an input sequence of *open_auction* elements. The *following-sibling* axis is only evaluated on the first element; the remaining elements are pruned. That the nested version, in fact, evaluates the *following-sibling* axis on all nodes in the context sequence (scenario N3) can easily be recognized by the correlation between the query time and the number of nodes in sequence s (configured by parameter `$n`).

XMark Queries

To assess the performance of the various operators in more complex queries, we also ran the 20 XMark queries (see Appendix B). The plan generator was again configured to deliver the eight scenarios from the beginning of this section. Figure 9.7 shows the result on a 10MB XMark document. Basically, what we have found during our path-only benchmark also holds true for the XMark queries: Comparing N1–N3 in Figure 9.7a, queries `xm01` to `xm05` contain long paths with *child* axes only. As we have seen before, N1 can beat a node-wise element index access (N3). Queries `xm06`, `xm07`, and `xm14` contain at least one '`//`' step but, again, N3 beats N1 and N2. Queries `xm08` to `xm12` are quite interesting. As you can see, their performance is poor on any evaluation scheme. The reason for this effect is that these queries compute at least one value-based join. Due to the nested-loop join

Figure 9.6 Effects of context-sequence pruning

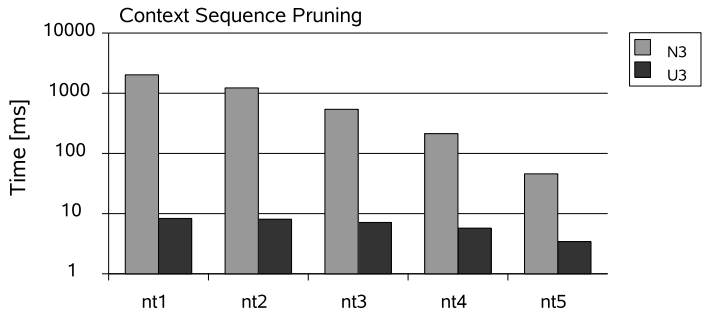
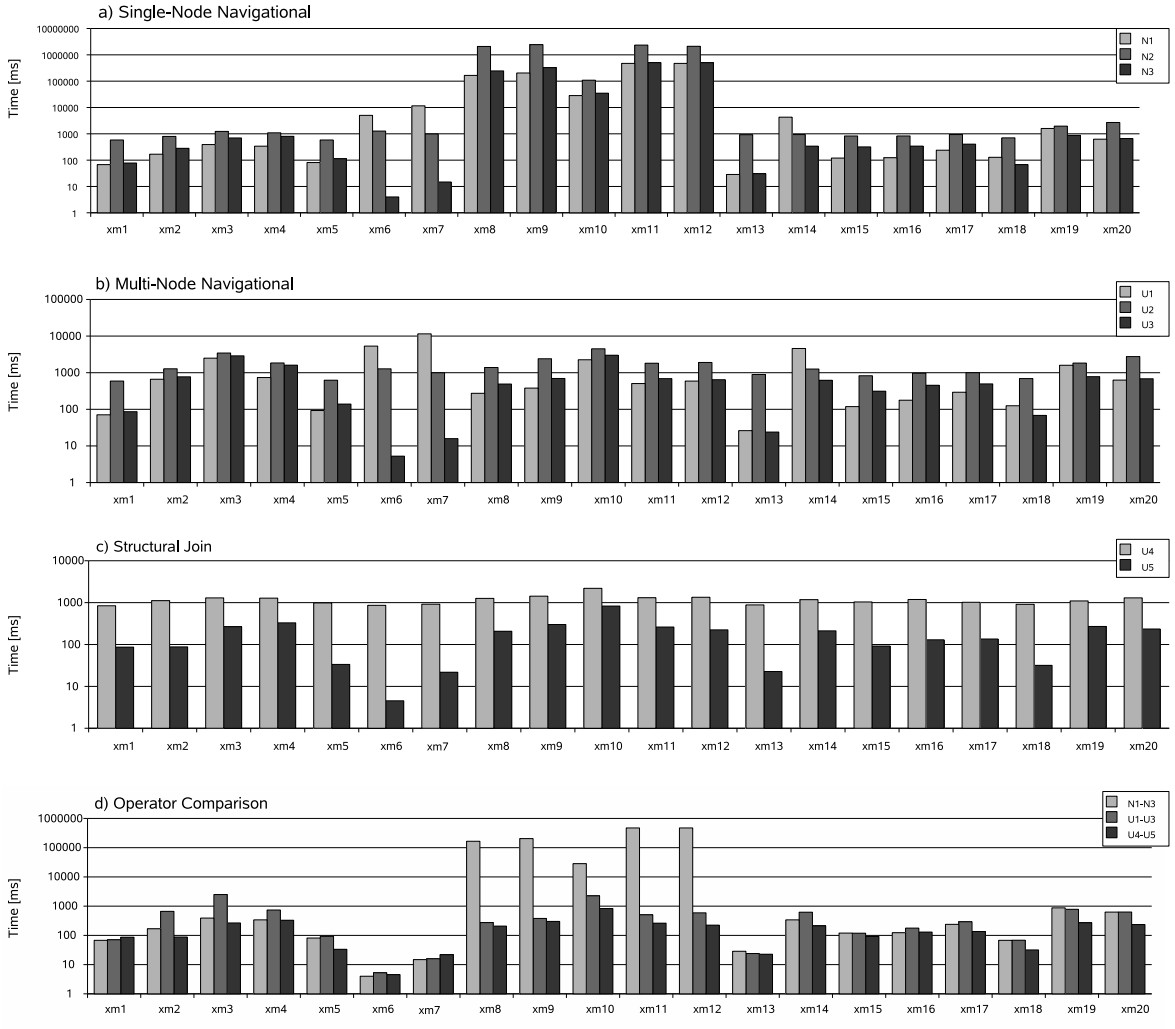


Figure 9.7 XMark evaluation under scenarios N1–N3 and U1–U5 on a 10MB XMark document



evaluation implied by the Formal Semantics (N1–N3), the query plans waste a lot of time for evaluating the inner expressions in such a join over and over again. For

the remaining queries, N1 and N3 show a similar performance, while N2 is the clear looser.

Let us now consider the unnested version which is based on the NavTree operator (Figure 9.7b). For all queries except `xm08` to `xm12`, we received comparable performane numbers w. r. t. the nested version. The rationale is that, again, the NavTree operator can obviously not successfully prune its context sequences. For the join-based queries (`xm8` – `xm12`), however, we can detect a substantial performance boost (one and a half to three orders of magnitude). The reason for that is, however, not the NavTree operator, but the more intelligent evaluation of the join-based queries. As explained in Section 8.6.3, we can avoid evaluating independent subqueries by mapping the XQGM operator with the join predicate to a hash join or a merge join. This happened in U1–U3.

Comparing structural joins with a document scan as input (U4) to structural joins on the element index (U5), Figure 9.7c resembles the corresponding chart from the path-based test. Note here again that the document-scan-based plans require around one second for the evaluation of almost any query. Even on the more complex XMark queries, the timings for the document-scan-based evaluation ranged around one second. Obviously, the processing speed is more influenced by I/O time (i. e., to scan the document) than by the main-memory processing time required by the plan operators.

The group comparison in Figure 9.7d again underlines the large performance gap between the nested and the unnested version of the query plans for the join queries. In summary, if an element index is available, the evaluation based on the structural join delivered the most stable and, in most cases, the fastest processing results. Therefore, we use structural joins as a baseline for the following experiments.

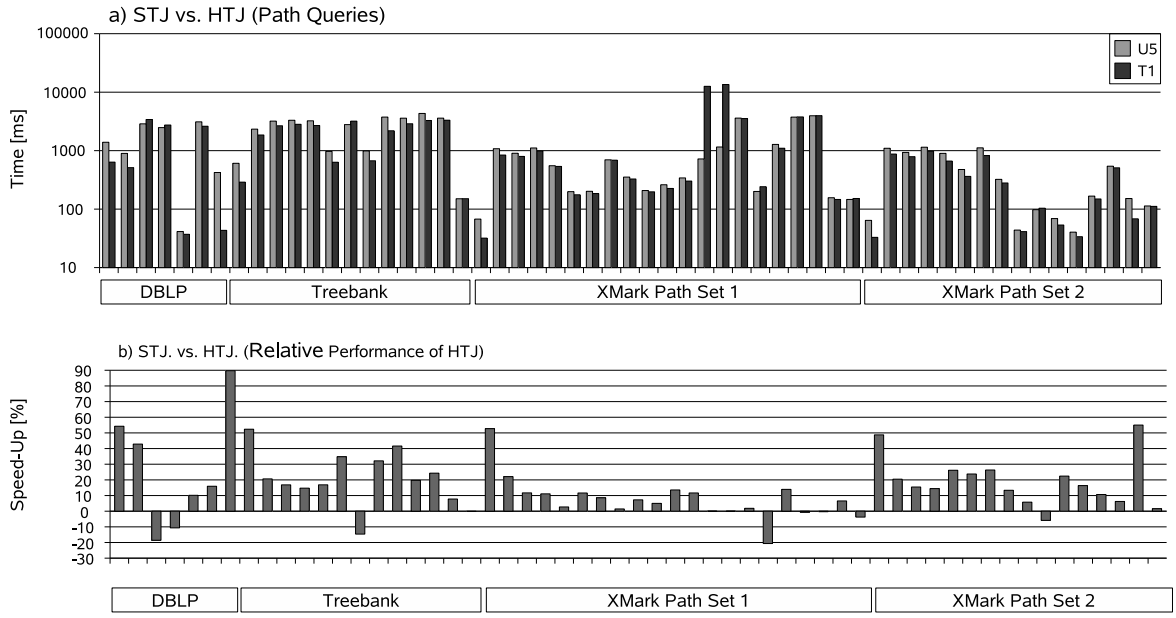
9.3.2 Join-Based PPOs

Join-based PPOs can only support query evaluation for certain axes. In case of the structural join, axes *child*, *descendant*, *descendant-or-self*, *parent*, *ancestor*, *ancestor-or-self*, and *attribute* are supported. Our twig join algorithm can only evaluate *child*, *descendant*, and *attribute* queries. All other axes have to be evaluated by navigations or by document scans, where the performance shown in the previous experiments can be expected. However, because "downward" axes are quite frequent in many XML queries, their optimization makes sense. In this section, we now want to see what further speed-ups can be achieved by using the twig join operator.

STJ vs. HTJ

To compare STJ-based plans with HTJ-based plans, we defined a new query processor configuration named T1: In T1, the query is unnested and twig discovery is enabled. The input of the twig operator is delivered by a (shared) element index scan. With this setting, we can directly compare T1 with U5 from the previous section. To do so, we extended the set of queried documents and the set of queries. In the literature on twig query processing, the three most cited experiment documents are DBLP, Treebank, and the XMark document. For our benchmark, we used a DBLP document of 128MB size. The Treebank document had a size of 83 MB, and the XMark document on of 112 MB. We tested the path queries introduced in the previous chapter (i. e., the queries of Listing 9.1) and, additionally, the path queries

Figure 9.8 STJ. vs. HTJ on the element index (U5 vs. T1)



we collected from the literature (e. g., from [Qin 07]). The additional queries can be found in Appendix B.

The benchmark result is shown in Figure 9.8. The first figure contains the absolute evaluation timings, whereas the second figure presents the speed-up of the HTJ operator w. r. t. the STJ-base evaluation. With the results from the literature in mind (e. g., [Bruno 02, Qin 07]), we expected the twig join to be the clear winner in this benchmark. However, as it turned out, the differences are not as high. Only on `d1`, `d7`, `t1`, `q1`, and `q22`, the twig algorithm is by far the better alternative. In

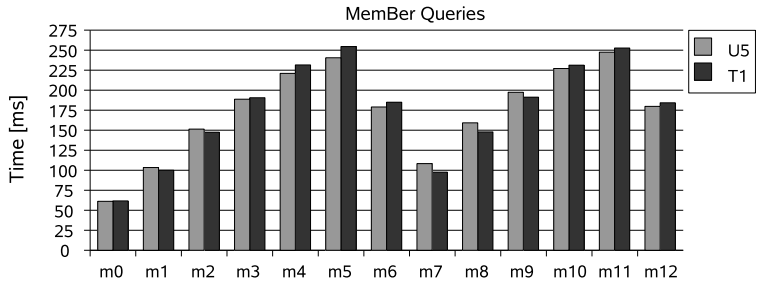
Listing 9.2 Queries on the MemBeR document

```
m0:  count(//t23)

m1:  count(//t23/t24)
m2:  count(//t23[t26]/t24)
m3:  count(//t23[t26]/t24/t31)
m4:  count(//t23[t26/t32]/t24/t31)
m5:  count(//t23[t26/t32 or t24/t31])
m6:  count(//t13[t24][t25][t33][t20])

m7:  count(//t23//t24)
m8:  count(//t23[.//t26]//t24)
m9:  count(//t23[.//t26]//t24//t31)
m10: count(//t23[.//t26//t32]//t24//t31)
m11: count(//t23[.//t26//t32 or .//t24//t31])
m12: count(//t13[.//t24][.//t25][.//t33][.//t20])
```

Figure 9.9 MemBer benchmark results



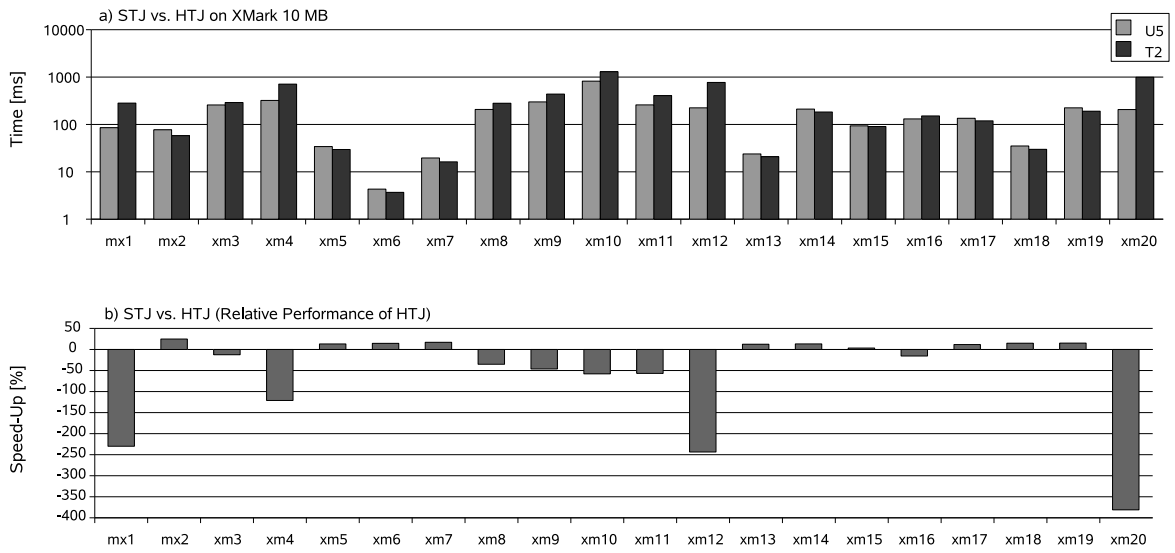
most other queries, the twig algorithm performs better (mind the logarithmic scale), but the speed-up is not breath-taking as expected. To analyze these circumstances and to see whether our twig implementation caused any problems, we profiled our implementation on various queries. For both, the STJ and the HTJ, the determining factor for the processing speed was I/O. In the hotspot analysis, I/O subroutines occurred on the first ranks; method calls of the STJ or HTJ operators occurred way further down the list.

To further check this theory, we conducted another experiment on a synthetic MemBeR document [Afanasiev 05]. The MemBeR data generator produces a more or less balanced tree, where the depth, the fan-out, the number of different tags (and their names), the assignment of tags to levels, and the distribution of tags can be configured. The configuration file for this experiment can be found in Appendix B ("MemBeR File 1"). The resulting document is more or less balanced with a size of 11 MB, has 50 different evenly distributed tag names, and a pre-defined fanout of 6. On this document, we posed the 13 queries of Listing 9.2, which delivered the timings shown in Figure 9.9.

Again, STJ and HTJ show comparable performance. The good thing with evenly distributed tag names is that every tag name has roughly the same cardinality. Therefore, an element index scan over a tag name requires the same time for every name. You can see the raw performance of an element index scan in the column for `m0`. It takes around 60 milliseconds (to retrieve roughly 25000 elements). The query set is constructed such that 1) the structure of `m1` to `m6` resembles the structure of `m7` to `m12` (the only difference is '`//`' usage); and 2) from `m1` to `m4` (`m7` to `m10`) a new path step is introduced in each query. With the introduction of each step, the query processing time raises accordingly (nearly by the time for an element index scan). Query `m6` (`m11`) has as many steps as its predecessor. However, the *or* predicate implies some additional evaluation overhead. On the other hand, query `m6` (`m12`) (also having five steps) is faster. The explanation is that the element index scan is partitioned (not the complete index is read, as in `m0`). Therefore, if the plan detects that no more results can be generated, the scan stops before all elements have been retrieved. Therefore, for most queries, the performance of STJ plans and HTJ plans mainly depends on the number of XML nodes read from external memory.

If a node has been read from external memory, it does not cause substantial main-memory processing costs. Therefore, the essential idea of *holism* behind twig joins—i. e., to discard nodes from which we know that they will not contribute to the final result—is actually not as critical w. r. t. query performance as one might get the im-

Figure 9.10 STJ vs. HTJ on XMark



pression from the literature. Even on the highly irregular and recursive Treebank document, the performance of STJ vs. HTJ was comparable, although the HTJ operator has the opportunity to discard XML nodes early. Therefore, what is more important (w. r. t. our results) is the meaningful restriction of elements read from external memory. This goal can be achieved by more expressive path indexes.

To conclude the discussion on Figure 9.8, consider the following: In queries `q14` and `q15`, the structural join performs significantly better. However, this result is not influenced by the performance of the operators, but by the structure of the plan generated. Query `q14` contains an attribute access and `q15` has content predicates. Both queries cannot be evaluated on the element index alone. They have to access the document. In the join-based implementation, the plan generator uses a NavTree algorithm for that. In the case of the twig join, a document scan is used to provide this input. Therefore, the queries are so relatively slow.

To compare STJ and HTJ on more complex queries, we ran them on the XMark queries. However, because XMark queries frequently need access to attributes and check content predicates, we have to take measures to avoid document scans. Therefore, we allow the HTJ to use a jumping cursor (see Section 8.4.3), whenever the necessary input cannot be read from the element index. The resulting query processor configuration is called T2 in the following.

The result is shown in Figure 9.10. Also this experiment underlines our theory that query processing performance is bound by I/O performance. For purely structural queries, both alternatives perform comparable, again with a slight advantage for the HTJ operator, e. g., queries `xm2`, `xm5` to `xm7`, etc. For queries with a content predicate or for our join queries, the twig alternative is, however, slower, e. g., `xm1`, `xm8` to `xm12`, etc. Here, the bias towards the STJ operator is more distinctive as before. The explanation comes from the fact that the jumping cursors of the HTJ operator need to jump into the document and scan until the first XML node is found that

fulfills the cursor's predicate. The "jump address" is calculated by the TwigOpt operator during a virtual cursor move. There might be quite some distance between the jump target to the first result node. On the other hand, in case of STJ plans, a NavTree operator is responsible for all requests that cannot be evaluated by the element index. Obviously, the input of the NavTree operator is a more reliable source for finding information, because the input nodes have been computed by document access (and not by virtual address computation).

HTJ vs. HTJ

Our HTJ implementation is quite flexible w. r. t. its input cursors. In the next test, we want to see what happens, when we let the HTJ algorithm run on different cursor types. Therefore, we configure the query processor to produce the following four scenarios:

T1) As before, the element index is scanned.

T2) As before, the element index is scanned, but all information not provided by that index is retrieved by jumping cursors.

T3) A shared document scan serves the input.

T4) Jumping cursors are used exclusively (preferring jumps on the element index rather than the document index if possible).

The result is shown in Figure 9.11. T1 and T2 are equal, when all input cursors can be served by the element index. Otherwise, T2 (using a jumping cursor) is the better strategy over T1 (using a document scan). Scenario T3 (shared document scan) produces relatively constant result timings (as before) but is always worse than T1 and T2. T4 (purely jumping) is almost always worse than T1 and T2. Therefore, generally relying on a jumping input cursor, as suggested in the original TwigOpt paper by [Fontoura 05], is not a good idea.

Jumping HTJ Cursors Reloaded

To show that jumping cursors *can* have a positive effect on query performance, we generated a set of documents using the MemBeR benchmark generator. The generator configuration file in shown in Appendix B ("MemBeR File 2"). The resulting document has a structure as depicted in Figure 9.12. The root node is named *x* and has a fan-out of 100000 nodes. At the second level, nodes *a* and *h* can occur. We vary the frequency of *a* nodes from 0.1% in document `member01` to 50% in document `member50`. The corresponding frequency for *h* nodes is set such that the sum of both frequencies is 100% (as required by the MemBeR data generator). The nodes (*b* and *c*) at the third level have a fan-out of 1 (i. e., exactly on child) and are evenly distributed. The fourth level is self-describing.

On these documents, we issued one and the same query: `//a[b/d]/c[e]` and compared scenario T1 (index scan) with T4 (purely jumping). The result is shown in Figure 9.13. On the highly selective documents (only few *a* nodes), the jumping twig implementation is the clear winner. The ratio flips at approximately 1%. Then, the scan-based implementation is faster. Clearly, the cursor configuration to choose for a given twig query and a document requires a cost-based decision.

Figure 9.11 HTJ vs. HTJ on XMark

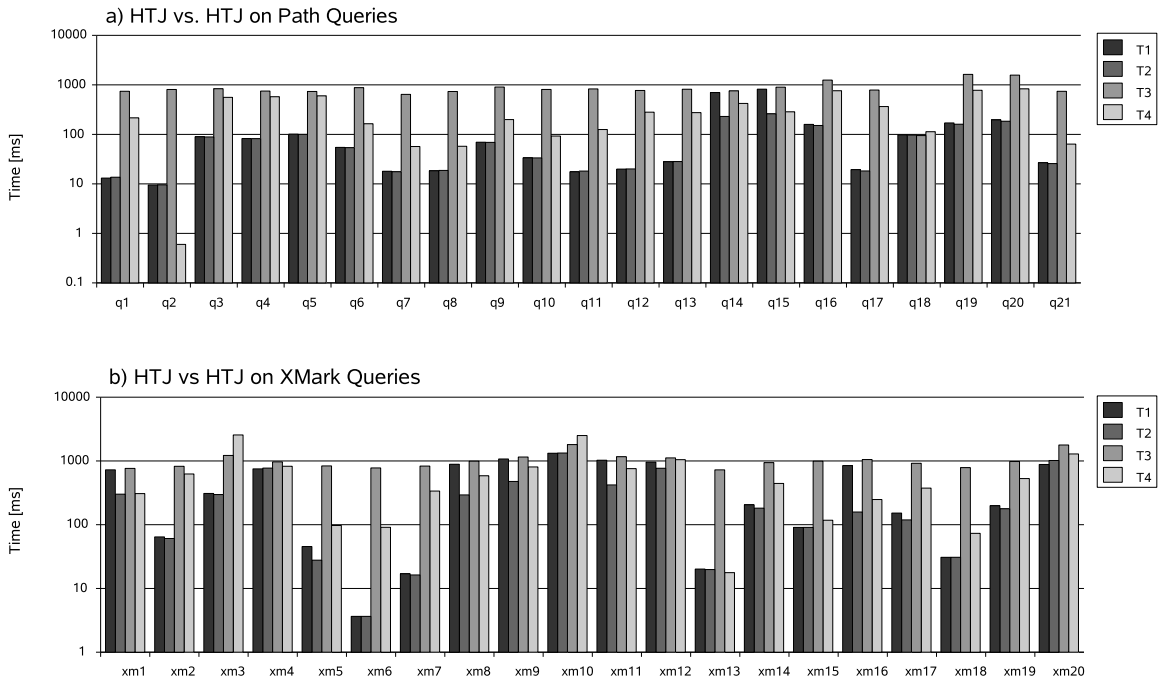


Figure 9.12 Structure of the MemBeR documents

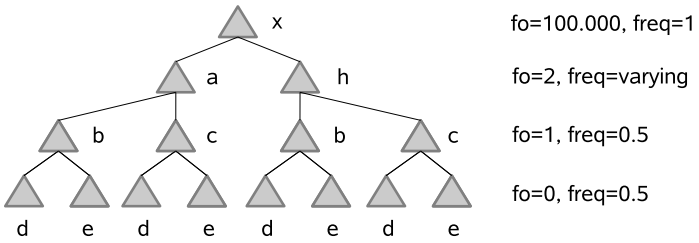


Figure 9.13 HTJ vs. HTJ on the MemBeR documents

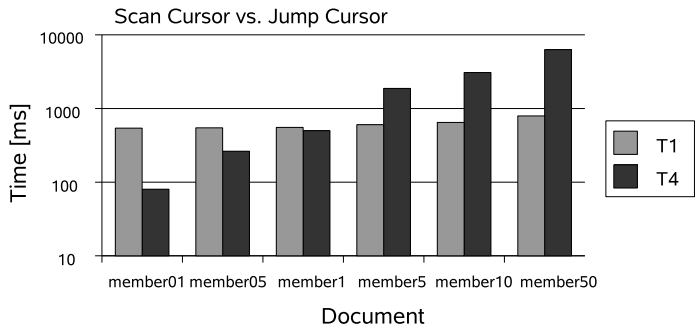
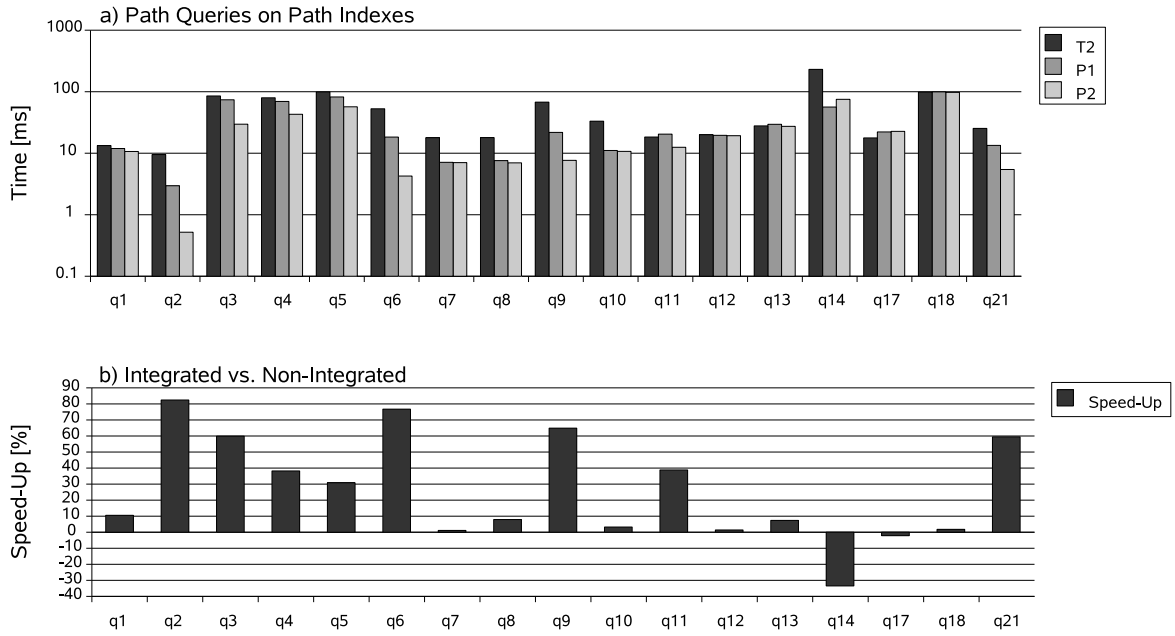


Figure 9.14 Path queries on path indexes



9.3.3 Index-Based PPOs

At this point, we finish the discussion on operators that work on plain indexes (i. e., on the document and the element index). In the next experiments, we added content indexes, path indexes, and CAS indexes as further access structures.

Simple Index Mapping vs. Complex Index Mapping

Let us first consider how we can speed up path queries without content predicates. To do so, we restrict the query set shown in Figure 9.1 by removing the content-based queries `q15`, `q16`, `q19`, and `q20`. Furthermore, we parameterize the plan generator such that plans of the following shape are produced:

P1) Simple index mapping: Path indexes are employed to deliver the input of sequence access operators. However, they are not integrated (i. e., inner elements are not produced; see Section 8.5.1). Inner elements are read from the element index.

P2) Complex index mapping: Path indexes are integrated into the twig, as described in 8.5.2.

As a baseline, we regard setting T2 from the previous section (i. e., a TwigOpt operator over the element index, which jumps when necessary). We create one index for each name drawn from the "ending node test" of all paths in the query set. For example, query `q1` ends on *name*, `q2` ends on *name*, *age*, and *income*, etc. Therefore, the indexes *I(//name)*, *I(//age)*, *I(//income)*, ... are created.

The result of the benchmark is shown in Figure 9.14. Figure 9.14a presents the absolute query timings. Let us consider T2 (evaluation on the element index) vs. P1 (simple path index mapping) first: As you can see, on almost all queries, the path

index reduces the query times. However, some gaps are larger than others. The reason behind this behavior is the way how we defined the path indexes and how the queries make use of them. For example, consider query `q6`. For this query, the element index returns all *keyword* elements for the bottom-most sequence access operator. In contrast, path index *I(//keyword)* returns only those *keyword* nodes that reside on the specified path, thus, effectively pruning the number of nodes processed. In other words: the *keyword* element occurs under 96 different paths (PCRs) in the document, of which only those elements are chosen and delivered by the path index that reside on exactly one path (PCR). In contrast, the element index returns *all keyword* elements.

However, this pruning technique does not always work. Consider, for example, path index *I(//age)*. Because *age* elements occur on only one path in the document, the path index is as good as the element index, i. e., it returns the same number of nodes on queries involving an *age* step. In this case, the path index can even perform worse than a simple element index scan, because the number of bytes read by the path index is larger than of using the element index (in contrast to the element index, the path index also contains PCR information). As a result, the overall query performance might be even worse. This situation occurs in query `q17`. This consideration has to be taken into account during the creation of a set of indexes for a given query workload.

Next, let us compare P1 (simple path mapping) with P2 (complex/integrated path mapping). The speed-up gained by embedding path indexes into the TwigOpt operator is shown in Figure 9.14b. Again, the reduction of the number of XML nodes read for query evaluation is significant for the evaluation time. The more nodes can be saved, the lower the time. Because in P2, inner elements are reconstructed, we do not need to query the element index to return them. Of course, the effect on queries with long paths is most significant, e. g., `q2`, `q6`, `q9`, and `q21`. On short paths, the effect is, however, hardly significant, e. g., `q7`. Furthermore, when inner nodes on the queried path do not occur frequently, the effect is also not significant. Query `q8` is an example. Elements *site* and *closed_auctions* appear only once in the whole document. Their computation using a path index does not deliver any significant improvement over an access to the element index.

Query `q14` seems to be an outlier. Here, both alternatives (P1 and P2) read the same number of XML nodes from path indexes. Therefore, no performance gain from embedding the indexes can be expected. Alternative P2 performs worse because of the integration cost into the TwigOpt operator (via the ancestor tuple builder).

In summary, for most of the queries shown, the use of a path index (and its embedding into the TwigOpt operator) results in a performance gain. However, as we have seen, a path index on all XML nodes with the same name does not guarantee any advantage over the element index per se. Only when the number of processed nodes can be restricted by the path index, an effect can be expected.

Content Index vs. CAS Index

In our next experiment, we considered queries with content predicates. The setting was slightly different than before: We did not alter the compiler configuration, but the physical database layout. Essentially, we modified the set of indexes available and chose the P2 option as compiler configuration. We generated the following physical database layouts (note, an element index existed in all configurations):

C1) Only the element index is available. This is our baseline for the experiment. CAS queries are evaluated by matching the structure first. On the matched XML nodes, the content is then checked via node-at-a-time look-ups in the document.

C2) An additional content index for element and attribute content exists. Queries are evaluated by matching the content predicate first. From the resulting content nodes, the parent element is checked by a node-at-a-time lookup in the document.

C3) A set of path indexes exists. The path indexes are constructed as before, i. e., by deriving the index definition from the last node tests in the query set (introduced below). As in C1, the structural part is evaluated first. However, here, the matching is boosted by path indexes. The content part has to be evaluated by node-at-a-time lookups in the document.

C4) A generic CAS index exists, i. e., *I(//* ∨ //@*)*. The CAS index can match both content and structure. The compiler configuration embeds the index into the TwigOpt algorithm.

C5) A heterogeneous CAS index exists. The indexed paths are derived from the "end steps" of the query as before. The index has the form *I(//name ∨ //age ∨ . . .)*. Again, the index is embedded.

C6) A set of homogeneous CAS indexes exists, i. e., for each "end step" a separate CAS index is created. Also here, indexes are embedded into the TwigOpt operator.

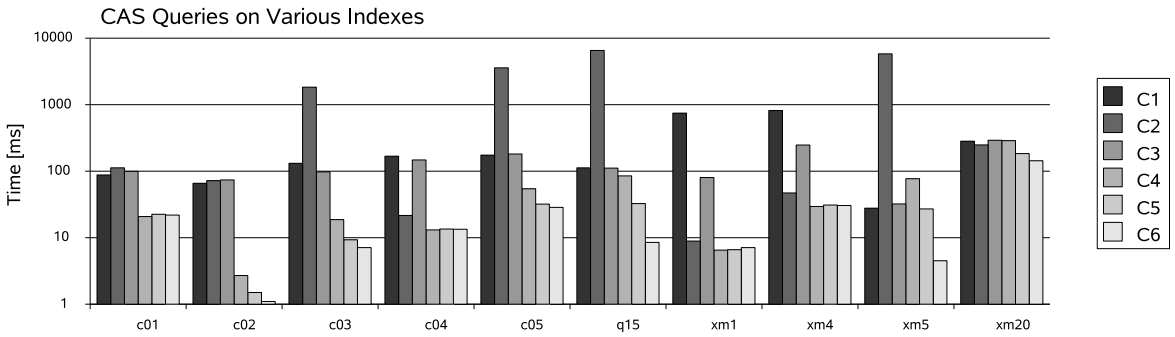
In compiler configuration P2, a precedence on the indexes to be used exists with the following priorities from highest to lowest: CAS, path, content, element index, document. This means, when a CAS index and an element index are available, P2 chooses the CAS index for the evaluation.

The query set for the experiment re-uses some content-based queries from previous experiments and also introduces new ones. The new queries are:

```
c1:  count(doc("auction.xml")//item[location="United States"])
c2:  count(doc("auction.xml")//item[location[. > "C" and . < "F"]])
c3:  doc("auction.xml")//item[description//keyword[. > "e" and . < "l"]]
c4:  doc("auction.xml")//item//mail
       [from="Arunabha Leitman mailto:Leitman@verity.com" and
        to="Tanya Plattner mailto:Plattner@njit.edu"]/date
c5:  count(doc("auction.xml")//item[quantity=1][.//keyword[. < "m" and . > "a"]])
```

From the previous queries, we reused path query `q15` and the XMark queries `xm1`, `xm4`, `xm5`, and `xm20`. With the benchmark setting, we want to explore two issues: 1. differences between indexes approaches, and 2. differences between generic and collective CAS indexes. The result of the benchmark is shown in Figure 9.15. First, we can observe that the evaluation time resulting from CAS index usage (C4 – C6) is in almost all queries lower than the other indexing approaches (C1 – C3). The only exception is setting C4 (a generic CAS index) on queries `xm5` and `xm20`. Comparing CAS indexes (C4 – C6), we can see that either all indexes deliver a similar performance (e. g., on `c1` or `c4`), or that the performance increases from the generic index (C4) over the heterogeneous collective index (C5) to the homogeneous collective indexes (C6) (e. g., `c2`, `c3`, or `xm5`). Again, the performance speed-up directly depends on the number of XML nodes read. For example, in query `c2`, the generic CAS index contains more nodes between "C" and "F" than the heterogeneous collective index, and, in turn, the heterogeneous index contains more nodes than the

Figure 9.15 CAS queries on indexed document



homogeneous index. The higher the selectivity of the index, the better for query performance. For some queries and indexes, no effect can be achieved, because the same amount of nodes has to be read from the index (e. g., `c1`).

Considering settings C1 – C3, we can observe that the pure content index (C2) often suffers a substantial performance penalty. The reason behind this problem is that in range queries, many possible content nodes can contribute to the final result. For each content node, its parent is resolved by a look-up in the document, which implies random I/O and is obviously expensive. However, when the number of nodes to be checked against the document is small, the content index can also deliver good performance (e. g., in query `c1`). Using path indexes to speed-up CAS queries (C3) does not always provide a better performance than just the plain element index (C1). Only for queries `xm1` and `xm4`, a substantial speed-up can be detected.

In summary, the experiment affirmed the rationale behind our CAS indexes. They beat the plain content and element index as well as purely structural path indexes. Furthermore, adjusting index selectivity influences the query performance.

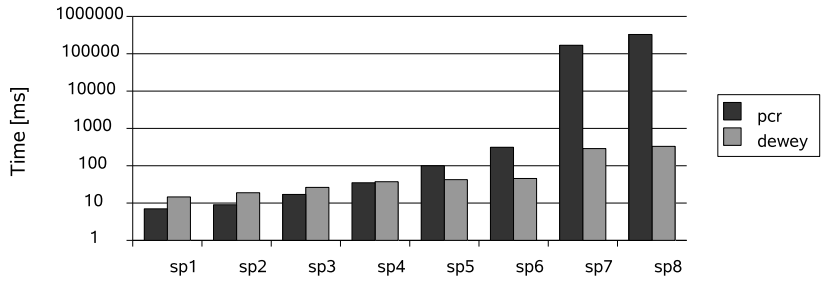
DeweyID Clustering vs. PCR Clustering

In our next experiment, we compare the two clustering strategies for path and CAS indexes, namely DeweyID clustering and PCR clustering. Therefore, we created a path index I_1*(//keyword)* with DeweyID clustering and with PCR clustering on a 100MB XMark document. In total, the index contains *keyword* elements with 99 differing PCRs. For DeweyID clustering, the key-value pairs of the index are ordered by the DeweyIDs. Thus, to retrieve all *keyword* elements with PCR 7 and 13 (for example), the index is scanned and all *keywords* not having PCR 7 or 13 are skipped. In PCR clustering, the pairs are ordered by PCR. Thus, to retrieve *keyword* elements with PCR 7 or 13, the index access operator retrieves all *keyword* elements with PCR 7, then it retrieves all elements with PCR 13, and, finally, the partitions are merged (to produce a result in document order).

To test for varying selectivities, the following query set has been designed:

```
sp1 (1 PCR):
count(doc("auction.xml")//categories/category/description/text/emph/keyword)
sp2 (2 PCRs):
count(doc("auction.xml")//annotation/description/parlist/listitem/text/emph/keyword)
sp3 (9 PCRs):
```

Figure 9.16 DeweyID clustering vs. PCR clustering




```
count(doc("auction.xml")//description//parlist//parlist//text/emph/keyword)
sp4 (18 PCRs):
count(doc("auction.xml")//description//parlist//text/emph/keyword)
sp5 (27 PCRs):
count(doc("auction.xml")//description//text/emph/keyword)
sp6 (33 PCRs):
count(doc("auction.xml")//text/bold/keyword)
sp7 (81 PCRs):
count(doc("auction.xml")//description//keyword)
sp8 (99 PCRs):
count(doc("auction.xml")//keyword)
```

Each query has a different selectivity (starting with the highest). Query `sp1` only returns *keyword* elements of a single path class. Query `sp2` accesses two path classes, `sp3` nine, and so on. Finally, `sp8` accesses all path classes.

The result of the benchmark is shown in Figure 9.16. When the selectivity is lower than 10% (i. e., queries `sp1` to `sp3`), PCR clustering has an advantage. On query `sp4` the ratio is balanced. For the remaining queries, DeweyID clustering wins. Here, it is obviously cheaper to simply scan the path index than to retrieve and merge multiple partitions. These characteristics have to be considered, when indexes are defined for query sets.

Indexing the XMark Query Set

In our last index-related experiment, we try to optimize the performance of all XMark queries by defining appropriate indexes. Therefore, we analyzed the XMark query set and created the indexes in Appendix B ("XMark Indexes") with DeweyID clustering on a 10MB XMark document. We configured the compiler with the following four scenarios:

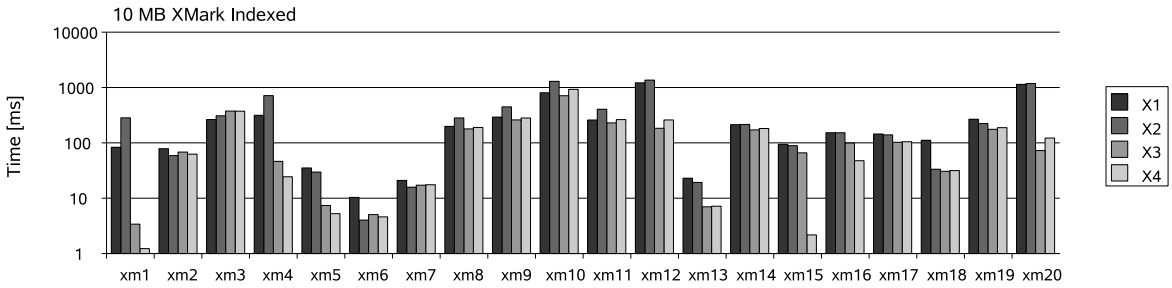
X1) The plan generator does not discover twigs and uses structural joins on the element index and on the document for query evaluation.

X2) Holistic twig joins are generated but, as in X1, only the element index and the document are accessed. Jumping cursors are generated for all input operators that cannot be implemented via the element index.

X3) Holistic twig joins are generated and path indexed are exploited. However, they are not embedded into the TwigOpt operator (simple index mapping).

X4) Holistic twig joins are generated and path indexes are embedded (complex index mapping).

Figure 9.17 Indexing the XMark query set



The result in Figure 9.17 is a little surprising. Only seven queries (`xm1`, `xm4`, `xm5`, `xm12`, `xm13`, `xm15`, and `xm20`) show a substantial performance speed-up. Most of these queries have a content predicate and make use of a CAS index. The corresponding STJ and HTJ alternatives (X1 and X2) do not have any content index and, therefore, have to access the document. On the other queries, either all HTJ implementations (X2 – X4) show a similar performance (e. g., `xm18`) or all implementations reside in similar regions (e. g., `xm14`). The rationale behind these characteristics is threefold:

1. Many path indexes we have created contain the same information as the element index, which is used as a fallback solution in X1 and X2. For example, because the *age* element only occurs in one path class, index *I(//profile/age)* contains as many elements as the corresponding element index for the name *age*. Therefore, no substantial performance speed-up can be gained by the path index. Even the contrary is true: the index contains DeweyIDs and PCRs and is, therefore, larger than the corresponding element index, which only contains the DeweyID (and no PCR information).

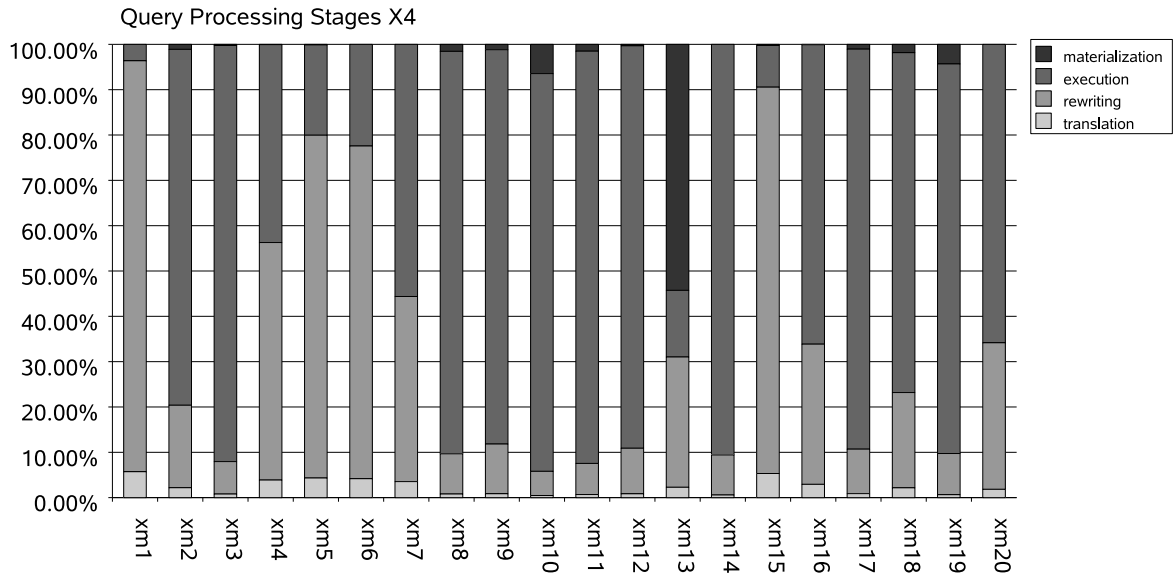
2. Some queries (e. g., the join-based queries `xm8` to `xm12`) have to access the document to evaluate the *data* function. Compared to an index access, this operation is expensive and overshadows the performance gain resulting from path index usage.

3. As we have seen earlier, embedding indexes only works well on longer paths (because then, the access to inner elements can be avoided). Many of the XMark queries have, however, short paths.

9.4 Other Processing Stages

In the previous experiments on query evaluation, we have always only recorded the evaluation time (without translation, optimization, materialization, etc.). However, because the complete response time of query evaluation does also depend on these other stages, we measure them here. In the current implementation, the two most time-consuming operations are the evaluation (shown before) and the materialization step. Because no matter how a particular query is evaluated, the result delivered by all alternatives is always the same. This means that the materialization time is always the same, too. Therefore, we did not include this measure in the

Figure 9.18 Ratio of the various query processing stages



previous experiments.

To assess the various stages, we have measured their run time during the previous experiment in configuration X4. We chose X4, because in this setting, many rewriting rules are fired and many indexes are taken into account during processing. In the overview, parsing, normalization, static typing, and simplification have been added up and represented as *transformation*. The XQGM transformation, query rewriting, twig discovery, and plan generation timings are contained in the *rewriting* figure. Execution and materialization are shown separately.

In all queries, the fraction of the translation time is below 5% and, therefore, quite negligible. Except for query `xm13`, the same holds for materialization. For large results, the materialization time can be a substantial fraction of the overall processing time. However, because the 20 XMark queries do not generate large results, materialization plays a minor role here. Finally, for longer running queries (e. g., `xm8` to `xm12`), the evaluation time is predominating. For short running queries (e. g.`xm1`), also the restructuring time plays a role.

9.5 Summary

In this chapter, we have empirically assessed the storage, indexing, and query processing techniques developed in this thesis. We first analyzed node-oriented (NO) and path-oriented (PO) storage. Both formats allowed to substantially reduce the space consumption for a document compared to the external (text) format. The space savings ranged from around 15% to 65%. Note, these storage savings were achieved without compressing the actual content (text) of the document. With text comression, further space reductions can be achieved [Schmidt 07]. Furthermore,

note that NO has higher information content than the external format and that PO has higher information content than NO: NO contains all XML nodes (as the external format) and, additionally, node IDs (i. e., DeweyIDs). PO, in turn, contains all information NO contains and, additionally, provides path information. In our experiments, we have seen that PO also has advantages over NO w. r. t. space consumption, storage time, scan access, and navigational access.

To assess query processing, we generated various compiler configurations and database layouts. At the beginning, we only allowed access to the document store and the element index. This setting is similar to what Natix and Timber provide as index structures (note, Galax operates in main memory). In this setting, we have tested path queries and the more complex XMark queries under navigational access (single-node navigations vs. NavTree) and under scan-based access (i. e., StackTree). For both query sets, we could not detect a substantial difference between the single-node navigational approach (i. e., XQuery Formal Semantics) and the multi-node navigational approach (i. e., NavTree). To show that the NavTree operator actually can result in a better performance, we had to construct a special experiment. In the end, the bulk-oriented StackTree operator on the element index was superior to the node-at-a-time approaches (although the difference on the XMark set was not as high as on the path set).

In the next experiment, we compared structural joins (StackTree) versus holistic twig joins (TwigOpt) on an extended set of path queries and on the 20 XMark queries. As database layout, we still only allowed the document store and the element index. The TwigOpt algorithm, therefore, had to use jumping cursors, when the element index could not provide the TwigOpt input. In the experiment, we hoped to be able to declare a clear winner. However, this was not possible. For many queries, the TwigOpt algorithm only showed a slight performance advantage (around 10 – 20%). For some queries, the structural join was even the better alternative and for other queries, it was the other way around. Especially, when the TwigOpt operator had to use a jumping cursor, the structural join alternative was better. Based on the experiments, we could infer the simple heuristics that the TwigOpt algorithm should always be preferred, except when it has to use a jumping cursor. In this case, an STJ-based plan should be generated. All in all, we think that the question STJ vs. HTJ is not the central problem. Restricting I/O is much more critical than main-memory processing.

In another experiment, we implemented the cursors of the TwigOpt algorithm in various ways (still on the document store and the element index). The results for the path queries and the XMark queries revealed that document scans can provide relatively stable results (independent of the query complexity). Furthermore, it showed that the original idea of the TwigOpt algorithm, to exclusively use jumping cursors, results in a quite slow execution time (sometimes even slower than the document scan). Here, again, we had to construct a special experiment to verify that jumping *can* have a positive effect. The best TwigOpt result was delivered by the implementation that combined the element index scan with jumping.

In the remaining experiments, we explored the effects of various indexing techniques on query performance. For the set of path queries, we first tested the effects of embedding path indexes into the TwigOpt operator vs. join-based evaluation and non-embedded path indexes. We have seen that embedding works well when the index used is more selective than the element index and when the paths are

not too short. Then we assessed queries with content predicates and content/ CAS indexes. We showed that plain content indexes carry the burden of an expensive post-processing phase to match the structural part of a query. Plain content indexes were always slower than CAS indexes. Furthermore, we have seen that the query selectivity plays a major role in influencing the evaluation time. Testing PCR clustering vs. DeweyID clustering, we revealed that explorative queries with descendant steps are better supported by DeweyID-clustered indexes, while more focuses queries (with only a few path classes) are better supported by PCR-clustered indexes.

Finally, we indexed the XMark document w. r. t. the XMark queries and tested STJ-based vs. HTJ-based evaluation (with and without embedded indexes). The HTJ alternative with embedded indexes was the winner in most queries. However, often enough, the non-embedded alternative showed a similar performance.

In a last test, we revealed the relative time consumption of all query processing stages. We have seen that the translation into the XQGM representation and the materialization took only a small fraction of the overall response time. As expected, the evaluation time required the largest fraction.

Chapter 10 Conclusion and Future Research

> The best thing about the future is that it comes only one day at a time.
>
> Abraham Lincoln

10.1 Conclusion

This work presented XML storage, indexing, and query processing techniques. The central contributions can be summarized as follows:

1) the adaption of the relational query processing pipeline for XML queries;
2) the extension of the query graph model to serve as an internal representation for XML queries (i. e., the XQGM);
3) the specification of the transformation process from an external XQuery expression to XQGM;
4) the algebraic optimization of XQGM instances (mainly query unnesting and twig discovery);
5) path-oriented document storage, path indexes, and their interplay;
6) the implementation of a physical XML algebra containing a rich set of alternative evaluation algorithms; and
7) the development of a plan generator that is able to fully exploit the physical algebra and, therewith, all available access path structures.

Our approach has shown that it is possible to reuse and extend relational techniques for XML processing. For example, we reused the query processing pipeline, the concept of a tuple algebra, and the internal query representation. However, as it turned out, due to the richer XML data model and due to the more complex XQuery language, substantial extensions and inventions were necessary. For example, at the logical level, twig discovery became necessary to address the frequent occurrence of twig patterns in queries. At the physical level, the more complex holistic twig join operator had to be integrated. Furthermore, path indexes provide much more information than simple relational indexes. Their integration into query evaluation plans is, therefore, more complex, too.

However, in the end, all techniques could be implemented in XTC—a native XML database management system. Using XTC as a testbed, we could compare the various approaches and reveal their strengths and weaknesses. The query processor was also demonstrated at the VLDB conference 2008 [Mathis 08]. The demo and the XTC system will/are also be publicly available at the XTC project site (`http://www.xtc-project.de`).

10.2 Future Work

Already in the first chapter, the limitations of this work have been pointed out. What is still missing for a full-fledged query processor is a statistics component and a cost model. The statistics component has to deliver information about the internal structure of the document and about the distribution of content values. The cost model has to provide information about the resource consumption characteristics of all algorithms in the physical algebra. Even with a long research history in the relational context, for XML, these two problems are quite hard to solve. Collecting statistics is complicated due to the rich structure of the XML data model. Developing a cost model is complicated due to the expressiveness of the operators in the physical algebra and the very differing shapes of query plans (nested vs. unnested).

However, we think that, with this work, the foundations for a full-fledged XML query processor have been laid. The integration of a cost model could be achieved in a graceful manner: first by considering more fundamental metrics, such as the number of records in an index, the number of page access operations for certain operators, etc., and, then, by integrating statistics about path and content selectivities. As a first approach, we could also try to "predict" the number of tuples generated by the plan operators and use this number as a coarse measure for the evaluation cost of a subquery. To assess the quality of our predictions, we could simply count the actual number of returned elements during query evaluation and compare them with our prediction.

For the development of a cost model, our first experiments presented in the previous chapter can be used as a starting point. Essentially, we have to "learn" the characteristics of the operators in the physical algebra over a certain statistical distribution of the input data. To achieve this goal, a data generator would be beneficial that can be parameterized to produce documents with different statistical distributions of the XML nodes contained. Then, we could measure the resource consumption during run time and directly link it with the statistical information of the input.

As we have seen, the operators of the physical algebra build upon basic access primitives, such as scan and navigations. Whenever possible, the cost model should be designed such that cost (i. e., the resource consumption) of these primitives could be parameterized, e. g., "fetching 10 nodes in a row from the document store has a cost of X". This facilitates the generalization of the cost model and its integration with other XML database systems and the adaption of new hardware (e. g., flash drives).

Another restriction of this work is the extent where XQuery is supported. Some missing artifacts of this language could be integrated quite easily, such as type expressions, more functions, or updates. Other concepts are probably not possible, i. e., user-defined recursive functions and XQuery scripting [Chamberlin 06]. To enable these concepts, the "programming language part" had to be separated from the

"query processing part" of the language. The first part could then be implemented as an XQuery interpreter, which calls the query processor whenever a supported XQuery expression has to be evaluated. However, separating these parts is not trivial, due to the expressiveness of the XQuery language. Finally, it would be nice to see a large and complex practical application being implemented over the XTC system and its query processor to further approve the value of our proposed techniques.

Part V

Appendix

Bibliography

[Afanasiev 05] L. Afanasiev, I. Manolescu & Ph. Michiels. *MemBeR: A Micro-benchmark Repository for XQuery*. In Proc. XSym, pages 144–161, 2005.

[Al-khalifa 02] Shurug Al-khalifa, Jignesh M. Patel, H. V. Jagadish, Divesh Srivastava, Nick Koudas & Yuqing Wu. *Structural joins: A Primitive for Efficient XML Query Pattern Matching*. In Proc. ICDE, pages 141–152, 2002.

[Amer-Yahia 04] Sihem Amer-Yahia, Fang Du & Juliana Freire. *A Comprehensive Solution to the XML-to-Relational Mapping Problem*. In Proc. WIDM, pages 31–38, 2004.

[Arion 08] Andrei Arion, Angela Bonifati, Ioana Manolescu & Andrea Pugliese. *Path Summaries and Path Partitioning in Modern XML Databases*. World Wide Web, vol. 11, no. 1, pages 117–151, 2008.

[Balmin 04] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta J. Cochrane & Hamid Pirahesh. *A Framework for Using Materialized XPath Views in XML Query Processing*. In Proc. VLDB, pages 60–71, 2004.

[Balmin 06] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. Simmen, M. Wang & C. Zhang. *Cost-based Optimization in DB2 XML*. IBM Systems Journal, vol. 45, no. 2, pages 299–319, 2006.

[Banerjee 00] Sandeepan Banerjee, Vishu Krishnamurty, Muralidhar Krishnaprasad & Ravi Murthy. *Oracle8i: The XML Enabled Data Management System*. In Proc. ICDE, pages 561–571, Washington, DC, USA, 2000. IEEE Computer Society.

[Bayer 72] R. Bayer & E. M. McCreight. *Organization and Maintenance of large Ordered Indexes*. Acta Informatica, vol. 1, no. 3, pages 173–189, 1972.

[Bayer 77] Rudolf Bayer & Karl Unterauer. *Prefix B-Trees*. ACM Transactions on Database Systems, vol. 2, no. 1, pages 11–26, 1977.

[Beeri 99] Catriel Beeri & Yariv Tzaban. *SAL: An algebra for Semistructured Data and XML*. In Informal Proc. of Workshop on The Web and Databases, ACM SIGMOD, pages 37–42, 1999.

[Berglund 04] Andreas Berglund, Scott Boag, Don Chamberlin, Mary Fernández, Michael Kay, Jonathan Robie & Jérôme Siméon. *XML Path Language (XPath) 2.0*. W3C Recommendation, 2004. `http://www.w3.org/TR/xpath20/`.

[Beyer 05] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery & Chun Zhang. *System RX: One Part relational, One Part XML*. In Proc. SIGMOD, pages 347–358, 2005.

[Beyer 06] K. Beyer, R. Cochrane, M. Hvizdos, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, R. Lyle, M. Nicola, F. Özcan, H. Pirahesh, N. Seemann, A. Singh, T. Truong, R. C. Van der Linden, B. Vickery, C. Zhang & G. Zhang. *DB2 Goes Hybrid: Integrating Native XML and XQuery with Relational Data and SQL*. IBM Systems Journal, vol. 45, no. 2, pages 271–298, 2006.

[Boag 04] Scott Boag, Donald Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie & Jérome Siméon. *XQuery 1.0: An XML Query Language*. W3C Recommendation, 2004. `http://www.w3.org/TR/xquery/`.

[Bohannon 02] Philip Bohannon, Juliana Freire, Prasan Roy, & Jérôme Siméon. *From XML Schema to Relations: A Cost-Based Approach to XML Storage*. In Proc. ICDE, pages 64–73, 2002.

[Boncz 99] Peter A. Boncz & Martin L. Kersten. *MIL Primitives for Querying a Fragmented World*. The VLDB Journal, vol. 8, no. 2, pages 101–119, 1999.

[Boncz 05a] Peter Boncz, Stefan Manegold & Jan Rittinger. *Updating the Pre/Post Plan in MonetDB/XQuery*. In Informal Proceedings XIME-P Workshop, pages 1190–1193, 2005.

[Boncz 05b] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger & Jens Teubner. *Pathfinder: XQuery-The Relational Way*. In Proc. VDLB, pages 1322–1325, 2005. (Demo Paper).

[Boncz 06a] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger & Jens Teubner. *MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine*. In Proc. SIGMOD, pages 479–490, 2006.

[Boncz 06b] Peter A. Boncz, Jan Flokstra, Torsten Grust, Maurice van Keulen, Stefan Manegold, K. Sjoerd Mullender, Jan Rittinger & Jens Teubner. *MonetDB/XQuery-Consistent and Efficient Updates on the Pre/Post Plane*. In Proc. EDBT, pages 1190–1193, 2006.

[Brantner 05] Matthias Brantner, Carl-Christian Kanne, Sven Helmer & Guido Moerkotte. *Full-fledged Algebraic XPath Processing in Natix*. In Proc. 21. ICDE Conference, Tokyo, Japan, pages 705–716, 2005.

[Brantner 06a] Matthias Brantner, Sven Helmer, Carl-Christian Kanne & Guide Moerkotte. *Kappa-Join: Efficient Execution of Existential Quantification in XML Query Languages*. Rapport technique, University of Mannheim, 2006.

[Brantner 06b] Matthias Brantner, Carl-Christian Kanne, Guido Moerkotte & Sven Helmer. *Algebraic Optimization of Nested XPath Expressions*. In Proc. ICDE, pages 128–130, 2006. Poster.

[Bray 06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler & François Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C Recommendataon, August 2006. `http://www.w3.org/TR/xml/`.

[Bruno 02] Nicolas Bruno, Nick Koudas & Divesh Srivastava. *Holistic Twig Joins: Optimal XML Pattern Matching*. In Proc. SIGMOD, pages 310–321, 2002.

[Carey 94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White & Michael J. Zwilling. *Shoring up Persistent Applications*. SIGMOD Record, vol. 23, no. 2, pages 383–394, 1994.

[Chamberlin 06] Don Chamberlin, Michael Carey, Daniela Florescu, Donald Kossmann & Jonathan Robie. *XQueryP: Programming with XQuery*. In Proc. XIME-P, pages 801–808, 2006.

[Chamberlin 07a] Don Chamberlin, Daniela Florescu, Jim Melton, Jonathan Robie & Jérôme Siméon. *XQuery Update Facility 1.0*. W3C Working Draft, 2007. `http://www.w3.org/TR/xquery-update-10/`.

[Chamberlin 07b] Don Chamberlin, Peter Frankhauser, Daniela Florescu, Massimo Marchiori & Jonathan Robie. *XML Query Use Cases*, March 2007. `http://www.w3.org/TR/xquery-use-cases/`.

[Chebotko 07] Artem Chebotko, Mustafa Atay, Shiyong Lu & Farshad Fotouhi. *XML Subtree Reconstruction from Relational Storage of XML Documents*. Data and Knowledge Engineering, vol. 62, no. 2, pages 199–218, 2007.

[Chen 03a] Qun Chen, Andrew Lim & Kian Win Ong. *D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data*. In Proc. SIGMOD, pages 134–144, 2003.

[Chen 03b] Yi Chen, Susan Davidson, Carmem Hara & Yifeng Zheng. *RRXS: Redundancy Reducing XML Storage in Relations*. In Proc. VLDB, pages 189–200, 2003.

[Chen 03c] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan & Stelios Paparizos. *From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery*. In Proc. VLDB, pages 237–248, 2003.

[Chen 05] Ting Chen, Jiaheng Lu & Tok Wang Ling. *On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques*. In Proc. SIGMOD, pages 455–466, 2005.

[Chen 06] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal & K. Selçuk Candan. *Twig2Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents*. In Proc. VLDB, pages 283–294, 2006.

[Chien 02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras & Carlo Zaniolo. *Efficient Structural Joins on Indexed XML Documents*. In Proc. VLDB, pages 263–274, 2002.

[Choi 07] B. Choi, M. Fernández & J. Siméon. *The XQuery Formal Semantics: A Foundation for Implementation and Optimization*. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery-semantics/`.

[Clark 99] James Clark & Steve DeRose. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, November 1999. `http://www.w3.org/TR/xpath`.

[Cokus 05] Mike Cokus & Santiago Pericas-Geertsen. *XML Binary Characterization Properties*. W3C Working Group Note, March 2005. `http://www.w3.org/TR/xbc-properties`.

[Comer 79] Douglas Comer. *The Ubiquitous B-Tree*. ACM Computing Surveys, vol. 11, no. 2, pages 121–137, 1979.

[Cooper 01] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason & Moshe Shadmon. *A Fast Index for Semistructured Data*. In Proc. VLDB, pages 341–350, 2001.

[Cover 05] Robin Cover. *XML Applications (XML Coverpages)*, June 2005. `http://xml.coverpages.org/xmlApplications.html`.

[Cowan 04] John Cowan & Richard Tobin. *XML Information Set (Second Edition)*. W3C Recommendation, February 2004. `http://www.w3.org/TR/xml-infoset/`.

[DeHaan 03] David DeHaan, David Toman, Mariano P. Consens, & M. Tamer Özsu. *A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding*. In Proc. SIGMOD, pages 623–634, 2003.

[DOM 04] DOM. *Document Object Model*. W3C Recommendation, April 2004.

[Effelsberg 84] Wolfgang Effelsberg & Theo Härder. *Principles of Database Buffer Management*. ACM Transactions on Database Systems (TODS), vol. 9, no. 4, pages 560–595, 1984.

[Fernández 00] Mary F. Fernández, Jérôme Siméon & Philip Wadler. *An Algebra for XML Query*. In Proc. FSTTCS, pages 11–45, 2000.

[Fernández 03] Mary Fernández, Jérôme Siméon, Byron Choi, Amélie Marian & Gargi Sur. *Implementing XQuery 1.0: The Galax Experience*. In Proc. VLDB, pages 1077–1080, 2003.

[Fernández 04] M. Fernández, A. Malhotra, J. March, M. Nagy & N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Recommendation, 2004. `http://www.w3.org/TR/xpath-datamodel/`.

[Fernández 05] M. Fernández, J. Hidders, Philippe Michiels, Jérôme Siméon & Roel Vercammen. *Optimizing Sorting and Duplicate Elimination*. In Proc. DEXA, pages 554–563, 2005.

[Fiebig 02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele & Till Westmann. *Anatomy of a Native XML Base Management Systemor*. VLDB Journal, vol. 11, no. 4, pages 292–314, 2002.

[Florescu 99] Daniela Florescu & Donald Kossmann. *Storing and Querying XML Data using an RDBMS*. Bulletin of the Technical Committee on Data Engineering, vol. 22, no. 3, pages 27–34, 1999.

[Fontoura 05] Marcus Fontoura, Vanja Josifovski, Eugene J. Shekita & Beverly Yang. *Optimizing Cursor Movement in Holistic Twig Joins*. In Proc. CIKM, pages 784–791, 2005.

[Franceschet 05] Massimo Franceschet. *XPathMark: An XPath Benchmark for the XMark Generated Data*. In XSym, pages 129–143, 2005.

[Georgiadis 07] Haris Georgiadis & Vasilis Vassalos. *XPath on Steroids: Exploiting Relational Engines for XPath Performance*. In Proc. SIGMOD, pages 317–328, 2007.

[Goldman 97] Roy Goldman & Jennifer Widom. *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*. In Proc. VLDB, pages 436–445, 1997.

[Gottlob 05] Georg Gottlob, Christoph Koch & Reinhard Pichler. *Efficient Algorithms for Processing XPath Queries*. ACM Transactions on Database Systems, vol. 30, no. 2, pages 444–491, 2005.

[Graefe 87] Goetz Graefe & David J. DeWitt. *The EXODUS Optimizer Generator*. In Proc. ACM SIGMOD International Conference on Management of Data, pages 160–172, 1987.

[Graefe 93] Goetz Graefe & William J. McKenna. *The Volcano Optimizer Generator: Extensibility and Efficient Search*. In Proc. ICDE, pages 209–218, 1993.

[Graefe 94] Goetz Graefe. *Volcano—An Extensible and Parallel Query Evaluation System*. IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 1, pages 120–135, 1994.

[Graefe 07] Goetz Graefe. *Algorithms for Merged Indexes*. In Proc. BTW, pages 112–131, 2007.

[Grinev 06] Maxim Grinev, Andrey Fomichev & Sergey Kuznetsov. *Sedna: A Native XML DBMS*. In Proc. SOFSEM, pages 272–281, 2006.

[Grust 03a] Torsten Grust & Maurice van Keulen. *Tree Awareness for Relational DBMS Kernels: Staircase Join*. In Intelligent Search on XML Data, pages 231–245, 2003.

[Grust 03b] Torsten Grust, Maurice van Keulen & Jens Teubner. *Staircase Join: Teach a Relational DBMS to Watch Its (Axis) Steps*. In Proc. VLDB, pages 524–535, 2003.

[Grust 04] Torsten Grust & Jens Teubner. *Relational Algebra: Mother Tongue - XQuery: Fluent*. In Proc. TDM, pages 9–16, 2004.

[Grust 07] Torsten Grust, Jan Rittinger & Jens Teubner. *WhyOFff-The-Shelf RDBMSs are Better at XPath Than You Might Expect*. In Proc. SIGMOD, pages 949–958, 2007.

[Haas 89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman & Hamid Pirahesh. *Extensible Query Processing in Starburst*. In Proc. SIGMOD, pages 377–388, 1989.

[Halverson 04] Alan Halverson, Vanja Josifovski, Guy Lohman, Hamid Pirahesh & Mathias Mörschel. *ROX: Relational over XML*. In Proc. VLDB, pages 264–275, 2004.

[Hammerschmidt 04] Beda Christoph Hammerschmidt, Martin Kempa & Volker Linnenmann. *A Selective Key-Oriented XML Index for the Index Selection Problem in XDBMS*. In Proc. DEXA, pages 273–284, 2004.

[Hammerschmidt 05] Beda Christoph Hammerschmidt, Martin Kempa & Volker Linnemann. *On the Intersection of XPath Expressions*. In Proc. IDEAS, pages 49–57, 2005.

[Härder 83] Theo Härder & Andreas Reuter. *Concepts for Implementing a Centralized Database Management System*. In Symposium on Application Systems Development, pages 28–60, 1983.

[Härder 01] Theo Härder & Erhard Rahm. Datenbanksysteme (Konzepte und Techniken der Implementierung). Springer, 2001. German only.

[Härder 05a] Theo Härder. *DBMS Architecture—Still an Open Problem*. In Proc. BTW, pages 2–28, March 2005.

[Härder 05b] Theo Härder, Michael P. Haustein, Christian Mathis & Markus Wagner. *Node Labeling Schemes for Dynamic XML Documents Reconsidered*. Data and Knowledge Engineering, vol. 60, no. 1, pages 126–149, 2005.

[Härder 07] Theo Härder, Christian Mathis & Karsten Schmidt. *Comparison of Complete and Elementless Native Storage of XML Documents*. In Proc. IDEAS, pages 102–113, 2007.

[Haustein 03] Michael P. Haustein & Theo Härder. *taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API*. In Proc. ADBIS, pages 88–102, 2003.

[Haustein 05a] Michael P. Haustein. *Verhinderung von Phantomen in XML-Datenbanksystemen mit wertbasierten Achsensperren*. In Berliner XML Tage, pages 79–92, 2005.

[Haustein 05b] Michael P. Haustein, Theo Härder, Christian Mathis & Markus Wagner. *DeweyIDs—The Key to Fine-Grained Management of XML Documents*. In Proc. SBBD, pages 85–99, 2005.

[Haustein 06a] Michael P. Haustein. *Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen*. PhD thesis, University of Kaiserslautern, 2006.

[Haustein 06b] Michael P. Haustein, Theo Härder & Konstantin Luttenberger. *Contest of XML Lock Protocols*. In Proc. VLDB, pages 1069–1080, 2006.

[He 04] Hao He & Jun Yang. *Multiresolution Indexing of XML for Frequent Queries*. In Proc. ICDE, pages 683–692, 2004.

[Helmer 02] Sven Helmer, Carl-Christian Kanne & Guido Moerkotte. *Optimized Translation of XPath into Algebraic Expressions Parameterized by Programs Containing Navigational Primitives*. In Proc. WISE, pages 215–224, 2002.

[Helmer 07] Sven Helmer. *Measuring the Structural Similarity of Semi-Structured Documents using Entropy*. In Proc. VLDB, pages 1022–1032, 2007.

[Hidders 04] Jan Hidders & Philippe Michiels. *Efficient XPath Axis Evaluation for DOM Data Structures*. In Proc. PLAN-X, 2004.

[Hidders 07] J. Hidders, P. Michiels, J. Siméon & R. Vercammen. *How To Recognize Different Kinds of Tree Patterns from Quite a Long Way Away*. In Proc. Plan-X, pages 14–24, 2007.

[IBM 09] IBM. *IBM DB2 pureXML*. Web Site, 2009. `http://www.ibm.com/db2/xml`.

[ISO/IEC 03] ISO/IEC. *XML-Related Specifications (SQL/XML)*. ISO/IEC 9075-14:2003 (Standard), 2003.

[Jagadish 02a] 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 & C. Yu. *TIMBER: A Mative XML Database*. VLDB Journal, vol. 11, no. 4, pages 274–291, 2002.

[Jagadish 02b] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava & KeithŢhompson. *TAX: A Tree Algebra for XML*. In Proc. DBPL, pages 149–164, 2002.

[Jiang 02] Haifeng Jiang, Hongjun Lu, Wei Wang & Jeffrey Xu Yu. *Path Materialization Revisited: An Efficient Storage Model for XML Data*. Australian Computer Science Communications, vol. 24, no. 2, pages 85–94, 2002.

[Jiang 03a] Haifeng Jiang, Hongjun Lu, Wei Wang & Beng Chin Ooi. *XR-Tree: Indexing XML Data for Efficient Structural Joins*. In Proc. ICDE, pages 253–264, 2003.

[Jiang 03b] Haifeng Jiang, Wei Wang, Hongjun Lu & Jeffrey Xu Yu. *Holistic Twig Joins on Indexed XML Documents*. In Proc. VLDB, pages 273–284, 2003.

[Jiao 05] Enhua Jiao, Tok Wang Ling & Chee Yong Chan. *PathStack¬: A Holistic Path Join Algorithm for Path Query with Not-Predicates on XML Data*. In Proc. DASFAA, pages 113–124, 2005.

[Kabra 99] Navin Kabra & David J. DeWitt. *OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization*. VLDB Journal, vol. 8, no. 1, pages 55–78, 1999.

[Kaushik 02a] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton & Henry F Korth. *Covering Indexes for Branching Path Queries*. In Proc. SIGMOD, pages 133–144, 2002.

[Kaushik 02b] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon & Ehud Gudes. *Exploiting Local Similarity for Indexing Paths in Graph-Structured Data*. In Proc. ICDE, pages 129–138, 2002.

[Kaushik 04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton & Raghu Ramakrishnan. *On the Integration of Structure Indexes and Inverted Lists*. In Proc. SIGMOD, pages 779–790, 2004.

[Kay 09] Michael Kay. *Saxon*. Website, 2009. `http://saxon.sourceforge.net/`.

[Kepser 04] S. Kepser. *A Simple Proof for the Turing-Completeness of XSLT and XQuery*. In Proc. Extreme Markup Languages, 2004.

[Kwon 05] Joonho Kwon, Praveen Rao, Bongki Moon & Sukho Lee. *FiST: Scalable XML Document Filtering by Sequencing Twig Patterns*. In Proc. VLDB, pages 217–228, 2005.

[Lee 00] Dongwon Lee & Wesley W. Chu. *Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema*. In Proc. Conceptual Modeling – ER, pages 641–654, 2000.

[Leroy 08] Xavier Leroy. *The Objective Caml System Release 3.11*. Manual, November 2008. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[Li 01] Quanzhong Li & Bongki Moon. *Indexing and Querying XML Data for Regular Path Expressions*. In Proc. VLDB, pages 361–370, 2001.

[Li 03] Quanzhong Li & Bongki Moon. *Partition-Based Path Join Algorithms for XML Data*. In Proc. DEXA, pages 160–170, 2003.

[Li 06] Hua-Gang Li, S. Alireza Aghili, Divyakant Agrawal & Amr El Abbadi. *FLUX: Content and Structure Matching of XPath Queries with Range Predicates*. In Proc. XSym, pages 61–76, 2006.

[Liu 08] Zhen Hua Liu, Thomas Baby, Sivasankaran Chandrasekar & Hui Chang. *Towards a Physical XML Independent XQuery/SQL/XML Engine*. In Proc. VLDB, pages 1356–1367, 2008.

[Lu 04] Jiaheng Lu, Ting Chen & Tok Wang Ling. *Efficient Processing of XML Twig Patterns with Parent Child Edges: a Look-Ahead Approach*. In Proc. CIKM, pages 533–542, 2004.

[Lu 05] Jiaheng Lu, Ting Chen & Tok Wang Ling. *TJFast: Effective Processing of XML Twig Pattern Matching*. In Proc. WWW, pages 1118–1119, 2005.

[Mang 03] Xiaofeng Mang, Yu Wang, Daofeng Luo, Shichao Lu, Jing An, Yan Chen, Jianbo Ou & Yu Jiang. *OrientX: A Schema-based Native XML Database System*. In Proc. VDLB, pages 1057–1060, 2003.

[Marian 03] Amélie Marian & Jérôme Siméon. *Projecting XML Documents*. In Proc. VLDB, pages 213–224, 2003.

[Mathis 06a] Christian Mathis & Theo Härder. *Hash-Based Structural Join Algorithms*. In Proc. EDBT Workshops, pages 136–149, 2006.

[Mathis 06b] Christian Mathis, Theo Härder & Michael Haustein. *Locking-Aware Structural Join Operators for XML Query Processing*. In Proc. SIGMOD, pages 467–478, 2006.

[Mathis 07a] Christian Mathis. *Extending a Tuple-Based XPath Algebra to Enhance Evaluation Flexibility*. Computer Science – Research and Development, vol. 21, no. 3, pages 147–164, 2007.

[Mathis 07b] Christian Mathis. *Integrating Structural Joins into a Tuple-Based XPath Algebra*. In Proc. BTW, pages 242–261, 2007.

[Mathis 08] Christian Mathis, Andreas Weiner, Theo Härder & Caesar Ralf Franz Hoppen. *XTCcmp: XQuery Compilation on XTC*. In Proc. VLDB, pages 1400–1403, 2008.

[Mathis 09] Christian Mathis. *Query Plans Generated by the XTC Query Processor*. Online, April 2009. `http://wwwlgis.informatik.uni-kl.de/cms/dbis/staff/mathis/`.

[Mavis K. Lee 88] Guy M. Lohman Mavis K. Lee Johann Christoph Freytag. *Implementing an Interpreter for Functional Rules in a Query Optimizer*. In Proc. VLDB, pages 18–229, 1988.

[May 04] Norman May, Sven Helmer & Guido Moerkotte. *Nested Queries and Quantifiers in an Ordered Context*. In Proc. ICDE, pages 239–248, 2004.

[May 05] Norman May & Guido Moerkotte. *Main Memory Implementations for Binary Grouping*. In Proc. XSym, pages 162–176, 2005.

[May 06a] Norman May, Matthias Brantner, Alexander Böhm 0002, Carl-Christian Kanne & Guido Moerkotte. *Index vs. Navigation in XPath Evaluation*. In Proc. XSym, pages 16–30, 2006.

[May 06b] Norman May, Sven Helmer & Guido Moerkotte. *Strategies for Query Unnesting in XML Databases*. ACM TODS, vol. 31, no. 3, pages 968–1013, 2006.

[Mchugh 97] Jason Mchugh & Serge Abiteboul. *Lore: A Database Management System for Semistructured Data*. SIGMOD Record, vol. 26, pages 54–66, 1997.

[McHugh 98] Jason McHugh, Jennifer Widom, Serge Abiteboul, Qingshan Luo & Anand Rajaraman. *Indexing Semistructured Data*. Rapport technique, Stanford University, 1998.

[Meier 02] Wolfgang Meier. *eXist: An Open Source Native XML Database*. LNCS, no. 2593, pages 169–183, 2002.

[Michiels 07] Philippe Michiels, George A. Mihaila & Jerome Simeon. *Put a Tree Pattern in your Algebra*. In Proc. ICDE, pages 246–255, 2007.

[Miklau 04] Gerome Miklau & Dan Suciu. *Containment and Equivalence for a Fragment of XPath*. Journal of the ACM, vol. 51, no. 1, pages 2–45, 2004.

[Miklau 09] Gerome Miklau. *XML Data Repository*. Website, Feburary 2009. `http://www.cs.washington.edu/research/xmldatasets/`.

[Milo 99] Tova Milo & Dan Suciu. *Index Structures for Path Expressions*. In Proc. ICDT, pages 277–295, 1999.

[Mitra 07] Nilo Mitra & Yves Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. W3C Recommendation, April 2007. `http://www.w3.org/TR/soap12-part0/`.

[Mitschang 95] Berhnhard Mitschang. Anfrageverarbeitung in Datenbanksystemen (Entwurfs- und Implementierungskonzepte). Vieweg, 1995. German only.

[Moerkotte 09] Guido Moerkotte. *Natix: Ein natives Datenbanksystem für XML*. Website, 2009. http://pi3.informatik.uni-mannheim.de/~moer/natix.html.

[Naughton 01] Jeffrey Naughton, David DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta & Rushan Chen. *The Niagara Internet Query System*. IEEE Data Engineering Bulletin, vol. 24, no. 2, pages 27–33, 2001.

[O'Neil 04] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller & Nigel Westbury. *ORDPATHs: Insert-Friendly XML Node Labels*. In Proc. SIGMOD, pages 903–908, 2004.

[Özcan 08] Fatma Özcan, Norman Seemann & Ling Wang. *XQuery Rewrite Optimization in IBM DB2 pureXML*. IEEE Data Engineering Bulletin, vol. 31, no. 4, pages 25–32, 2008.

[Papakonstantinou 95] Yannis Papakonstantinou, Hector Garcia-Molina & Jeniffer Widom. *Object Exchange Across Heterogeneous Information Sources*. In Proc. ICDE, pages 251–260, 1995.

[Paparizos 02] Stelios Paparizos, Shurug Al-Khalifa, H.V. Jagadish, Andrew Nierman & Yuqing Wu. *A Physical Algebra for XML*. Rapport technique, University of Michigan, 2002.

[Paparizos 04] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan & H. V. Jagadish. *Tree Logical Classes for Efficient Evaluation of XQuery*. In Proc. SIGMOD, pages 71–82, 2004.

[Parr 07] Terence Parr. The Definite ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers, 2007.

[Päßler 06] Manfred Päßler & Matthias Nicola. *Native XML-Unterstützung in DB2 Viper*. Datenbank-Spektrum, vol. 17, pages 42–47, 2006.

[Pirahesh 92] Hamid Pirahesh, Joseph M. Hellerstein & Waqar Hasan. *Extensible/Rule Based Query Rewrite Optimization in Starburst*. SIGMOD Record, vol. 21, no. 2, pages 39–48, 1992.

[Prakash 06] Sandeep Prakash, Sourav S. Bhowmick & Sanjay Madria. *Efficient Recursive XML Query Processing Using Relational Database Systems*. Data and Knowledge Engineering, vol. 58, no. 3, pages 207–242, 2006.

[Prasad 05] K. Hima Prasad & P. Sreenivasa Kumar. *Efficient Indexing and Querying of XML Data Using Modified Prüfer Sequences*. In Proc. CIKM, pages 397–404, 2005.

[Qin 07] Lu Qin, Jeffrey Xu Yu & Bolin Ding. *TwigList: Make Twig Pattern Matching Fast*. In Proc. DASFAA, pages 850–862, 2007.

[Rao 04] Praveen Rao & Bongki Moon. *PRIX: Indexing And Querying XML Using Prüfer Sequences*. In Proc. ICDE, pages 288–297, 2004.

[Ré 06] Christopher Ré, Jérôme Siméon & Mary F. Fernández. *A Complete and Efficient Algebraic Compiler for XQuery*. In Proc. ICDE, pages 14–23, 2006.

[Rys 05] Michael Rys. *XML and Relational Database Management Systems: Inside Microsoft®SQL Server™2005*. In Proc. SIGMOD, pages 958–962, 2005.

[Schmidt 02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu & Ralph Busse. *XMark: A Benchmark for XML Data Management*. In Proc. VLDB, pages 974–985, 2002.

[Schmidt 07] Karsten Schmidt & Theo Härder. *Tailor-Made Native XML Storage Structures*. In Proc. ADBIS, pages 96–106, 2007.

[Schöning 00] Harald Schöning & Jürgen Wäsch. *Tamino - An Internet Database System*. In Proc. EDBT, pages 383–387, 2000.

[Shanmugasundaram 99] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt & J. Naughton. *Relational Databases for Querying XML Documents: Limitations and Opportunities*. In Proc. VLDB, pages 302–314, 1999.

[Siméon 04] Jérôme Siméon & Mary F. Fernández. *Build Your Own XQuery Processor*. Presentation, September 2004. http://edbtss04.dia.uniroma3.it/.

[Srinivasan 92] V. Srinivasan & Michael J. Carey. *Performance of On-Line Index Construction Algorithms*. In Proc. EDBT, pages 293–309, 1992.

[Tatarinov 02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita & Chun Zhang. *Storing and Querying Ordered XML Using a Relational Database System*. In Proc SIGMOD, pages 204–215, 2002.

[UniProt 08] UniProt. *Protein Knowledgebase*, Feburary 2008. http://beta.uniprot.org/help/.

[Vagena 04] Zografoula Vagena, Mirella M. Moro & Vassilis J. Tsotras. *Efficient Processing of XML Containment Queries Using Partition-Based Schemes*. In Proc. IDEAS, pages 161–170, 2004.

[Vagena 05] Zografoula Vagena, Nick Koudas, Divesh Srivastava & Vassilis J. Tsotras. *Efficient Handling of Positional Predicates Within XML Query Processing*. In XSym, pages 68–83, 2005.

[Wagner 73] R. E. Wagner. *Indexing Design Considerations*. IBM Systems Journal, vol. 12, no. 4, pages 351–367, 1973.

[Wang 03] Haixun Wang, Sanghyun Park, Wei Fan & Philip S. Yu. *ViST: A Dynamic Index Method for Querying XML Data by Tree Structures*. In Proc. SIGMOD, pages 110–121, 2003.

[Wang 05] Wei Wang, Haifeng Jiang, Hongzhi Wang, Xuemin Lin, Hongjun Lu & Jianzhong Li. *Efficient processing of XML Path Queries Using the Disk-Based F&B Index*. In Proc. VLDB, pages 145–156, 2005.

[Wedekind 74] H. Wedekind. *On the Selection of Access Paths in a Data Base System*. Data Base Management, pages 385–397, 1974. J. W. Klimbie and K. L. Koffemann (eds.).

[Wu 03] Yuqing Wu, Jignesh M. Patel & H. V. Jagadish. *Structural Join Order Selection for XML Query Optimization*. In Proc. ICDE, pages 443–454, 2003.

[XQuery 09] XQuery. *The W3C XML Query Site*. Website, 2009. http://www.w3.org/XML/Query/.

[Yoshikawa 01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura & Shunsuke Uemura. *XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases*. ACM Transactions on Internet Technology (TOIT), vol. 1, no. 1, pages 110–141, 2001.

[Yu 06] Tian Yu, Tok Wang Ling & Jiaheng Lu. *TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-Predicates on XML Data*. In Proc. DASFAA, pages 249–263, 2006.

[Zhang 04] Ning Zhang, Varun Kacholia & M. Tamer Özsu. *A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML*. In Proc. ICDE, pages 54–63, 2004.

The Sample Document

```
<?xml version="1.0" encoding="utf-8"?>
<recordStore>
  <cd id="cd_100">
    <title> The Soul Cages </title>
    <artist> Sting </artist>
    <year> 1998 </year>
    <genre> Pop </genre>
    <price> 19.99 </price>
    <tracks>
     <track no="1" length="401">
        <title> Island of Souls </title>
      </track>
      <track no="2" length="294">
        <title> All This Time </title>
      </track>
      <track no="3" length="233">
        <title> Mad About You </title>
      </track>
      <!-- ... -->
      <track no="7" length="402">
        <title> The Wild Wild Sea </title>
      </track>
      <track no="8" length="352">
        <title> The Soul Cages </title>
      </track>
      <track no="9" length="468">
        <title> When the Angels Fall </title>
      </track>
    </tracks>
  </cd>
  <vinyl id="lp_399">
    <title> Kind of Blue </title>
    <artist> Miles Davis </artist>
    <year> 1959 </year>
    <genre> Jazz </genre>
    <format> Long Play </format>
    <speed> 33 1/3 </speed>
    <price> 12.89 </price>
    <aSide>
      <tracks>
        <track no="1" length="562">
          <title> So What </title>
        </track>
        <track no="2" length="586">
          <title> Freddie Freeloader </title>
        </track>
        <track no="3" length="337">
          <title> Blue in Green </title>
        </track>
      <tracks>
    </aSide>
    <bSide>
      <tracks>
        <track no="4" length="693">
           <title> All Blues </title>
        </track>
        <track no="5" length="566">
          <title>
            Flamenco Sketches (take 2)
          </title>
        </track>
      </tracks>
    <bSide>
  </vinyl>
  <cd id="cd_220>
    <title> By the Way </title>
    <band> Red Hot Chili Peppers </band>
    <year> 2002 </year>
    <genre> Rock </genre>
    <price> 7.99 </price>
    <tracks>
      <track no="1" length="217">
        <title> By the Way </title>
      </track>
      <track no="2" length="259">
        <title> Universally Speaking </title>
      </track>
      <track no="3" length="257">
        <title> This Is the Place </title>
      </track>
      <track no="4" length="312">
        <title> Dosed </title>
      </track>
      <track no="5" length="277">
        <title> Don't Forget Me </title>
      </track>
      <!-- ... -->
      <track no="11" length="208">
        <title> Cabron </title>
      </track>
      <track no="12" length="317">
        <title> Tear </title>
      </track>
      <track no="13" length="208">
        <title> On Mercury </title>
      </track>
      <track no="14" length="217">
        <title> Minor Thing </title>
      </track>
      <track no="15" length="256">
        <title> Warm Tape </title>
      </track>
      <track no="16" length="367">
        <title> Venice Queen </title>
      </track>
    </tracks>
  </cd>
  <!-- ... -->
</recordStore>
```

Queries and Settings

XMark

```
xm1:
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()

xm2:
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
return <increase>{$b/bidder[1]/increase/text()}</increase>

xm3:
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2
      <=
      $b/bidder[last()]/increase/text()
return
  <increase
  first="{$b/bidder[1]/increase/text()}"
  last="{$b/bidder[last()]/increase/text()}"/>

xm4:
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where
  some $pr1 in $b/bidder/personref[@person = "person20"],
       $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>

xm5:
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)

xm6:
let $auction := doc("auction.xml") return
for $b in $auction//site/regions return count($b//item)

xm7:
let $auction := doc("auction.xml") return
for $p in $auction/site
return
  count($p//description) + count($p//annotation) + count($p//emailaddress)
```

```
xm8:
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>

xm9:
let $auction := doc("auction.xml") return
let $ca := $auction/site/closed_auctions/closed_auction return
let
    $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a :=
  for $t in $ca
  where $p/@id = $t/buyer/@person
  return
    let $n := for $t2 in $ei where $t/itemref/@item = $t2/@id return $t2
    return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>

xm10:
let $auction := doc("auction.xml") return
for $i in
  distinct-values($auction/site/people/person/profile/interest/@category)
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe>{$t/profile/gender/text()}</sexe>
        <age>{$t/profile/age/text()}</age>
        <education>{$t/profile/education/text()}</education>
        <revenu>{fn:data($t/profile/@income)}</revenu>
      </statistiques>
      <coordonnees>
        <nom>{$t/name/text()}</nom>
        <rue>{$t/address/street/text()}</rue>
        <ville>{$t/address/city/text()}</ville>
        <pays>{$t/address/country/text()}</pays>
        <reseau>
          <courrier>{$t/emailaddress/text()}</courrier>
          <pagePerso>{$t/homepage/text()}</pagePerso>
        </reseau>
      </coordonnees>
      <cartePaiement>{$t/creditcard/text()}</cartePaiement>
    </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>

xm11:
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return <items name="{$p/name/text()}">{count($l)}</items>

xm12:
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)}</items>
```

```
xm13:
let $auction := doc("auction.xml") return
for $i in $auction/site/regions/australia/item
return <item name="{$i/name/text()}">{$i/description}</item>

xm14:
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()

xm15:
let $auction := doc("auction.xml") return
for $a in
  $auction/site/closed_auctions/closed_auction/annotation/description/parlist/
   listitem/
   parlist/
   listitem/
   text/
   emph/
   keyword/
   text()
return <text>{$a}</text>

xm16:
let $auction := doc("auction.xml") return
for $a in $auction/site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/parlist/listitem/text/emph/
       keyword/
       text()
    )
  )
return <person id="{$a/seller/@person}"/>

xm20 (note, the sequence of the following queries has been altered):
let $auction := doc("auction.xml") return
<result>
  <preferred>
    {count($auction/site/people/person/profile[@income >= 100000])}
  </preferred>
  <standard>
    {
      count(
        $auction/site/people/person/
         profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {count($auction/site/people/person/profile[@income < 30000])}
  </challenge>
  <na>
    {
      count(
        for $p in $auction/site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>

xm17:
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
where empty($p/homepage/text())
return <person name="{$p/name/text()}"/>
```

```
xm18 (note, we do not support user defined functions):
let $auction := doc("auction.xml") return
for $i in $auction/site/open_auctions/open_auction
return (zero-or-one($i/reserve) * 2.20371)

xm19:
let $auction := doc("auction.xml") return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending
return <item name="{$k}">{$b/location/text()}</item>
```

Note, we had to alter query `xml18` because in the current version of XTC, we do not support user-defined functions. Therefore, the calculation expressed by such a function in the original query set [Schmidt 02] was "inlined" here. Furthermore, note that we altered the order of queries `xm17` to `xm20` for formatting reasons. In the benchmark results, these queries are displayed in the correct order.

Path Queries on the DBLP Document

```
d1: count(//inproceedings//title[.//i]//sup)
d2: count(//article[.//sup]//title/sub)
d3: count(//dblp/inproceedings[title]/author)
d4: count(//dblp/article[author][.//title]//year)
d5: count(//dblp/inproceedings[.//cite/label][title]/author)
d6: count(//dblp/article[author][.//title][.//url][.//ee]//year)
d7: count(//article[.//mdate][.//volume][.//cite//label]//journal)
```

Path Queries on the Treebank Document

```
t1:  count(//S//ADJ[.//MD])
t2:  count(//S[.//JJ]/NP)
t3:  count(//S/VP/PP[NP/VBN]/IN)
t4:  count(//S//NP[.//PP/TO][VP/_NONE_]/JJ)
t5:  count(//S/VP/PP[IN]/NP/VBN)
t6:  count(//S[DT]//PRP_DOLLAR_)
t7:  count(//S[.//VP/IN]//NP)
t8:  count(//VP[DT]//PRP_DOLLAR_)
t9:  count(//NP[NN]/PP)
t10: count(//S/VP/PP[.//IN][.//NP[.//CD]/VBN]/IN)
t11: count(//S[.//VP][.//NP]/VP/PP[IN]/NP/VBN)
t12: count(//EMPTY[.//VP/PP//NNP][.//S[.//PP//JJ]//VBN]//PP/NP//_NONE_)
```

Additional Path Queries on the XMark Document

```
q22: count(//site/people/person/name)
q23: count(//site//people//person[.//name and .//age]//income)
q24: count(//text[bold]/emph/keyword)
q25: count(//listitem[.//bold]/text//emph)
q26: count(//listitem[.//bold]/text[.//emph]/keyword)
```

```
q27: count(//text[.//bold]//keyword)
q28: count(//description[.//text]//parlist)
q29: count(//text[bold][.//keyword][.//emph])
q30: count(//item[location]/description//keyword)
q32: count(/site/closed_auctions/closed_auction/price)
q32: count(/site/regions//item/location)
q33: count(/site/people/person/gender)
q34: count(/site/open_auctions/open_auction/reserve)
q35: count(//people/person[.//address/zipcode]/profile/education)
q36: count(//item[location][.//mailbox/mail//emph]/description//keyword)
q37: count(//people//person[.//address/zipcode][id]/profile[.//age]/education)
q38: count(//open_auction[.//annotation[.//person]//parlist]//bidder//increase)
```

MemBeR File 1

```
<?xml version="1.0"?>
<Tree
       xmlns="http://microbenchmarks.org/treegen"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://microbenchmarks.org/treegen␣TreeGen.xsd"
       fanout="6" size="50000" distribution="normal" tags="50"/>
```

With this input, the MemBeR generator produces an XML document of 11 MB size.

MemBeR File 2

```
<?xml version="1.0"?>
<Tree
       xmlns="http://microbenchmarks.org/treegen"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://microbenchmarks.org/treegen␣TreeGen.xsd"
       depth="4" fanout="3" size="5000000" distribution="normal">

  <Tag name="x" levels="1" frequency="1"    fanout="100000"/>
  <Tag name="a" levels="2" frequency="$y"  fanout="2"/>
  <Tag name="h" levels="2" frequency="$x"  fanout="2"/>
  <Tag name="b" levels="3" frequency="0.5"  fanout="1"/>
  <Tag name="c" levels="3" frequency="0.5"  fanout="1"/>
  <Tag name="d" levels="4" frequency="0.5"  fanout="0"/>
  <Tag name="e" levels="4" frequency="0.5"  fanout="0"/>
</Tree>
```

This is an input file for the MemBer data generator [Afanasiev 05]. For `member10`, `$x` is set to `0.9` and `$y` is set to `0.1`. Thus, 10% *a* nodes are generated. Other files are generated analogous.

XMark Indexes

```
create element index

create cas index paths //people/person/@id
create cas index paths //bidder/personref/@person
create cas index paths //closed_auction/price
create cas index paths //person/profile/@income

create path index paths //people/person/@id
create path index paths //people/person/name
create path index paths //open_auction/bidder/increase
create path index paths //open_auction/bidder
create path index paths //open_auction/reserve
create path index paths //closed_auction/price
create path index paths //regions//item
create path index paths //closed_auction/buyer/@person
create path index paths //itemref/@item
create path index paths //item/@id
create path index paths //profile/interest/@category
create path index paths //profile/gender
create path index paths //profile/age
create path index paths //profile/education
create path index paths //profile/@income
create path index paths //person/address/street
create path index paths //person/address/city
create path index paths //person/address/country
create path index paths //person/emailaddress
create path index paths //person/homepage
create path index paths //person/creditcard
create path index paths //open_auction/initial
create path index paths //australia/item
create path index paths //australia/item/name
create path index paths //australia/item/description
create path index paths //item/description
create path index paths //closed_auction/seller/@person
create path index paths //item/name
create path index paths //item/location
create path index paths //closed_auction/annotation/description/parlist
  /listitem/parlist/listitem/text/emph/keyword
create path index paths //annotation/description/parlist/listitem/parlist
  /listitem/text/emph/keyword
```

Curriculum Vitae

Christian Mathis

Personal Data:

Business Address: Gottlieb-Daimler-Str. 47
67663 Kaiserslautern
Germany
Phone: + 49 (0) 631 205 3282
E-Mail: mathis@informatik.uni-kl.de

Date of Birth: December 29th 1977

Place of Birth: Bad Kreuznach, Germany

Marital Status: Married

Nationality: German

Education and Work Experience:

10/2004 – 03/2009	Doctoral Candidate at the University of Kaiserslautern (Databases and Information Systems Workgroup)
01/2001 – 06/2003	Student Assistant at the "Databases and Information Systems" Workgroup Programmer at the SENSOR Project ("Supporting Software Engineering Processes by Object-Relational Database Technology")
10/1998 – 09/2004	Academic Studies at the University of Kaiserslautern Academic Degree: "Diplom-Informatiker" Main Focus: Databases and Information Systems Minor Subjects: Physics and Economics Diploma Thesis: "Application Programming APIs for XML Database Systems"
08/1997 – 08/1998	Civil Service at the Red Cross, Bad Kreuznach Qualification as Emergency Medical Technician
08/1988 – 07/1997	Secondary School: "Elisabeth-Langgässer-Gymnasium" Alzey General Qualification for University Entrance