DISS. ETH No. 20166

Flexible Models for Programming the Web

A B H A N D L U N G
zur Erlangung des Titels

DOKTOR DER WISSENSCHAFTEN
der
ETH ZÜRICH

vorgelegt von
GHISLAIN FOURNY
Dipl. Informatik-Ing. ETH

geboren am 16. November 1981
von Frankreich

Angenommen auf Antrag von
Prof. Dr. Donald Kossmann, Referent
Prof. Dr. Torsten Grust, Korreferent
Prof. Dr. Peter Müller, Korreferent
Prof. Dr. Timothy Roscoe, Korreferent

2011

In loving memory of my late grand-parents:

Denise, Simon and Marc

Acknowledgements

My first indebted thanks go to Donald Kossmann for his invaluable support during the past four years. I tend to think of doctoral studies as knowledge transfer from a mentor to an apprentice, and this transfer did take place: I left ETH with a lot of new insights and techniques, thought processes and ways to look at things inherited from my advisor. Donald, thank you for entrusting me with your mentorship and for giving me so much leeway in carrying out this research.

This Wissenstransfer took place in a very fertile research ecosystem officially known as the Systems Group, locally fostered successively by the best office mates I could have wished for (Georges, Marcos, Cristian, Lukas, Irina, Tudor, Simon L, Nihal, Jana) and with prolific and entertaining coffee mates (mostly Tudor, Jens, Louis, Stefan, Ercan, Simon P), with whom discussions about life, the universe, time, quantum theory, and pretty much everything actually, were daily business. I collaborated with Peter Fischer on several projects; I found him a judicious and conscientious researcher, and a very pleasant person to work with. I would like to thank Ueli Maurer as well, with whom I began my PhD. From Ueli, I gratefully took home two important words which emboss his research: simplicity and precision.

In a natural complement to this academic environment, I was thrilled to collaborate with colleagues in various companies such as Oracle (Dana Florescu, Markos Zaharioudakis), 28msec (Matthias Brantner, William Candillon, David Graf, Dennis Knochenwefel, Gabriel Petrovay, Till Westmann), Elsevier (Darin McBeath). I am very thankful for their time and support, especially to Dana who inspired me a lot with her vision and way of thinking, and who gave me a lot of her valuable time and advice throughout my thesis. The W3C XML Query working group was a very stimulating environment as well, which gave me the opportunity to go in depth in XML technologies. Also, many thanks to Jonathan Robie for his insights and impulse on the JSONiq project.

During this time, I had the pleasure to supervise four students who greatly backed me in my research with their developer skills: Markus Pilman (XQuery in the Browser),

Konstantinos Tsoulos (Object Orientation in XQuery), Andreas Morf (Time Machine), Thomas Etter (XQuery in the Browser without setup). Let them be thanked as well.

This exciting territory was the catalyst of the present document, patiently reviewed by the committee members: Timothy Roscoe, Peter Müller, Torsten Grust. I am very grateful to them for their time and for providing me with constructive and very much appreciated remarks on the content and the structure, for raising questions allowing improvements, and for pointing to me errors and typos that escaped my attention. I would like to thank Thomas Gross for chairing the examination committee. Many thanks also to Thomas and Nicola for the translation of the abstract in German and Italian.

Regardful thanks to Jean-Pierre Dupuy, who has also been a mentor, inspirer and supporter to me for the past years, with the wish that our exciting game theory research work will keep going and, I hope, ramifications on the quantum theory side can be investigated further.

Affectionate thanks to my loving parents, Rolande and Francisque, and grand-mother, Hélène, who actively supported me with their kind words and presence in the good as well as more challenging times.

Warm thanks to my caring friends and colleagues Angelo and Bettina, Daniel and Tabitha, Eric, Hafez, Julien and Aurore, Nicola and Raissa, Pascale and Vincent, Pierre, Stephan and Patricia, Steven and Anna, Thomas, André, André-Samuel, Apostolos, Baptiste, Blandine, Carlos, Cécile, Christian D, Christian G, Daniel D, Daniel Z, David, Dejan, Dominic, Dominique, Flavio, Florin, François M, François T, Gianmatteo, Jonas, Lars, Laurence, Leo, Martin, Matus, Nikhil, Olivier, Pascal, Patrice, Philipp, Sami, Sebastian, Steve, Tobias, Thérèse, Yannick, Yoshi, Yves M, Yves W.

Many thanks to God for making 3^{12} so close to 2^{19}.

Abstract

The Web has evolved considerably over the past twenty years. Browsers now support full-fledged Web applications, and the latter are slowly replacing local desktop applications, since, combined with cloud storage, they allow a device-independent access.

The most widespread Web architecture is probably the three-tier architecture. Back-end storage is often supported by a relational database. Business logics is implemented on a Web server using powerful imperative, object-oriented programming languages. Client-side code is written in a scripting language, some properties of which might differ between the browsers. The front-end user interface is decoupled between a tree-based page structure, and its cosmetics which are controlled separately with stylesheets.

The discrepancy within this technology stack leads to the need to convert data between different formats between the layers: for example, relational tuples can be converted to objects for the middleware, to a lightweight exchange format for transport over the Web to the client, and then to a page structure for display in the navigator. It also leads to higher costs as programmers need to be trained or hired for each language, and to the impossibility of reusing code between the layers.

This thesis suggests a simplification of this technology stack with the use, on all layers, of semi-structured data (such as XML) with a programming language handling this data natively (such as XQuery) for more flexibility in the architectural space. It provides version control capabilities in order to support query-based time travel in semi-structured data. Finally, it investigates how XQuery can be extended and improved towards more intuition (new scripting grammar) and more code reuse (object-orientational features), to allow for scalable Web applications and gain acceptance among Web developers.

Zusammenfassung

Das Web hat sich in den letzten zwanzig Jahren deutlich weiter entwickelt. Browser unterstützen nun den Gebrauch von vollwertigen Webanwendungen. Diese Anwendungen verdrängen schrittweise die lokalen Desktop-Anwendungen, da durch diese, kombiniert mit Cloud Storage, geräteunabhängiger Zugriff möglich wird. Die am weitesten verbreitete Webapplikationarchitektur ist warscheinlich die dreiteilige Architektur: Backend-Storage wird meistens von relationalen Datenbanken unterstützt; Business-Logic läuft auf einem Webserver, der mächtige, objektorientierte Programmiersprachen ausführt, wärend auf Benutzerseite verschiedene Skriptingsprachen zum Einsatz kommen. Das Frontend-User-Interface ist entkoppelt zwischen einer baumbasierten Struktur und seiner Darstellung, welche separat durch Stylesheets kontrolliert wird.

Die Diskrepanz innerhalb dieser Infrastruktur verlangt das Konvertieren von Daten zwischen den verschiedenen Schichten: zum Beispiel können relationale Tupel in Objekte für die Middleware, dann in ein effizientes Austauschformat für den Transport über das Internet zum Benutzer, und schliesslich in eine Struktur für den Bildschirm im Navigator konvertiert werden. Zudem führt dies auch zu höheren Kosten, da einerseits mehr Programmierer geschult oder Spezialisten eingestellt werden müssen, und da die Wiederverwendung von Code zwischen den Schichten unmöglich ist.

In dieser Dissertation wird eine Vereinfachung dieser Infrastruktur vorgeschlagen, indem in allen Schichten semi-strukturierte Daten (wie etwa XML) verwendet werden, und eine Programmiersprache welche diese Daten nativ handhabt (wie etwa XQuery) um mehr Flexibilität in der Architektur zu erreichen. Ausserdem wird auch Versionskontrolle vorgestellt, um anfragenbasierte Zeitreisen in semi-strukturierten Daten zu ermöglichen. Schliesslich untersucht diese Abhandlung, wie XQuery erweitert und verbessert werden kann, um die Intuitivität (neue Skripting-Grammatik) und Wiederverwendbarkeit im Code (objekt-orientierte Eigenschaften) zu erhöhen, damit Webapplikationen besser skalieren, und damit eine grössere Akzeptanz unter Web-Programmierern erreicht wird.

Résumé

Pendant les vingt dernières années, le Web a considérablement évolué. Les navigateurs supportent des applications Web complètes, lesquelles remplacent peu à peu les applications de bureau. Associées au stockage dans le Cloud, elles permettent un accès indépendant de l'appareil utilisé.

L'architecture la plus répandue pour les sites Web est probablement l'architecture à trois niveaux. Souvent, le niveau de stockage est réalisé avec une base de données relationelle. La logique du site est implémentée sur un serveur Web grâce à des langages de programmation impératifs et/ou orientés object puissants. Le code exécuté du côté du client est écrit dans un langage de script dont les caractéristiques peuvent différer entre les navigateurs. L'interface utilisateur est découplée entre une page à structure arborescente, et la cosmétique qui est contrôlée séparément avec des feuilles de style.

Le décalage dans ces technologies conduit au besoin de convertir les données dans plusieurs formats pour chacun des niveaux : par exemple, les n-uplets relationels peuvent être convertis en objets pour le niveau logique, en un format d'échange léger pour être transportés sur le Web au client, puis en une structure de page pour être affiché dans le navigateur. Cela conduit aussi à des coûts plus élevés puisque les programmeurs doivent apprendre ou être recrutés pour chacun de ces languages, ainsi qu'à l'impossibilité de réutiliser le code entre les niveaux.

Cette thèse suggère une simplification de cette architecture avec l'utilisation, sur tous les niveaux, de données semi-structurées (comme XML) avec un langage de programmation traitant ces données nativement (comme XQuery) pour plus de flexibilité dans l'espace architectural. Elle fournit une capacité de versionage pour supporter des requêtes de voyage dans le temps dans les données semi-structurées. Finalement, elle étudie comment XQuery peut être étendu et amélioré pour plus d'intuition (nouvelle grammaire de script) et plus de réutilisation du code (orientation objet), afin de permettre le dévelopement d'applications Web qui peuvent monter en charge, et afin d'être potentiellement accepté par les développeurs Web.

Riassunto

Durante gli ultimi vent'anni il Web si è evoluto considerevolmente. Al giorno d'oggi i browser supportano delle applicazioni web complete, che stanno rimpiazzando poco a poco le applicazioni desktop tradizionali: combinate con il cloud-storage, esse permettono un accesso indipendente dall'apparecchio utilizzato.

L'architettura più diffusa per i moderni siti internet è probabilmente quella a tre livelli: il livello per il salvataggio dei dati è solitamente affidato a una banca dati relazionale; la logica del programma è implementata su un server web tramite performanti linguaggi di programmazione imperativi e orientati agli oggetti; il codice lato client è invece scritto in un linguaggio di scripting, che può variare a dipendenza del browser.

L'interfaccia per gli utenti finali è divisa in una parte con un struttura gerarchica ad albero e una parte grafica, controllata separatamente con dei fogli di stile.

Tutte queste differenze fra le tecnologie usate implicano la necessità di convertire le informazioni in diversi formati tra i vari livelli: per esempio, le tuple relazionali devono essere convertite in oggetti per il livello logico, in un altro formato di scambio leggero per essere trasportate attraverso il web all'utente, e infine ancora in una struttura di pagina web per poter essere visualizzate sul browser. Questo comporta l'impossibilità di riutilizzare il codice fra i vari livelli, nonché dei costi più elevati, poiché diversi programmatori devono essere formati o assunti per ogni linguaggio di programmazione.

Questa tesi suggerisce una semplificazione dell'architettura classica grazie all'uso, su tutti i livelli, di dati semi strutturati (come XML) e di un linguaggio di programmazione che possa gestire questi dati in modo nativo (come XQuery) per una maggiore flessibilità architetturale. Essa presenta inoltre una capacità di versioning per supportare query temporali nei dati semi-strutturati.

Infine, analizza come XQuery può essere esteso e migliorato per una maggiore intuitività (nuova grammatica di scripting) e un maggior riutilizzo del codice (orientazione agli oggetti), al fine di permettere lo sviluppo di applicazioni web scalabili e guadagnare così l'approvazione degli sviluppatori web.

Contents

Chapter 1

Introduction and Vision

When the World Wide Web was invented twenty years ago [47], its goal was to provide a platform facilitating the exchange of information thanks to a network of documents hyper-linked to each other.

During the last two decades, many noticeable evolutions took place:

Web of data The Web no longer only contains human-readable documents, it also contains huge amounts of raw data which can be directly processed by computers. The Web, in addition to the originally intended human-computer communication, allows computer-computer communication. One of Tim Berners-Lee's top priorities in the future of the Web is linked data (what he originally called semantic Web), i.e., leverage the Web to link data to make it available to everyone, query it and build new applications on top of this data [46].

Main platform In the 90s, the Web was a technology available on the Internet among many others (E-mail, Usenet, IRC, FTP, telnet, instant messaging...). Today, it has become the main communication platform: forums are available as Web sites, files are commonly downloaded over the Web, users can interact and discuss using Web-based fora and social networks.

Web applications HTML pages are no longer static documents. They contain code which is executed on the user's side. The browser has become a real programming environment. Many client-side applications (for example text processors, spreadsheets, presentations) are now available in the browser, potentially replacing their desktop application ancestors [1] [15] [26].

Mobile devices The Web can be accessed on hand-held devices such as smart-phones or tablets, which means that the number of nodes in the network is increasing.

The IPv4 address space is exhausted. This trend is very likely going to gain in momentum, moving towards what is already called the Web of things [34].

Online time With DSL and 3G subscriptions prominently replacing access to the Internet through regular phone lines in the last decade, most of the devices are now online most of the time. This enables use of cloud storage [25][5]: the data is stored in remote data centers, can be shared between devices and accessed through Web interfaces. Also, Google's Chrome OS [14] provides a Web-applications-only environment (with offline capability though).

These evolutions suggest that programming the Web [31], i.e., having servers and client devices create, exchange and process data and code through the HTTP protocol [18], is becoming increasingly crucial.

1.1 State of the Art

This section presents the state of the art in technologies used for Web applications. XML technologies are presented further down, in a section of their own.

1.1.1 Data formats

Web pages to be displayed in the browser are writing using HTML (HyperText Markup Language) [17] and formatted using CSS (Cascading StyleSheets) [8]. Clients can also receive and process other formats such as XML [38], JSON [21], raw text. Persistent data can be stored using these formats as well, but traditionally, it is stored in a relational database [58].

1.1.2 Programming languages

The most widely used architecture to build Web applications is based on three tiers: database, business logic and client.

The database layer stores and manages persistent data. The business logic layer (also called middleware) performs server-side computations on top of this data. The client (mostly a browser) communicates with the server and might perform computations on its side as well.

The technologies used on these three tiers are very different:

- On the database tier, the most popular language is SQL [42], although database companies (Oracle, Microsoft...) are increasingly including XML support with the XQuery [48] language.

- On the business tier, Java EE [19] and ASP.NET [24] are the main proprietary alternatives, whereas the most popular open-source language is PHP [30].

- The client also uses very specific languages. Dynamic Web pages execute code, often written in JavaScript [80], a dialect of the ECMAScript standard [11]. A popular alternative is Flash/Flex [2] which is used for animations, requiring the installation of a plugin, although the upcoming HTML5 might supplant it.

The Google Web Toolkit [16] allows cross-compiling Java code to JavaScript code. The main advantages is that it leverages Java features such as strong typing, allowing early error detection, and it allows Java experts without knowledge in JavaScript to program the client. Alternatively, there are attempts to use JavaScript on the server.

1.1.3 Communication protocols

The layers in the three-tier architecture communication using protocols on top of TCP/IP.

The communication protocol between the database and the business logic layer is technology-dependent. In some cases (especially smaller applications), these two layers are even on the same machine.

The business logic layer and the client communicate using the HTTP protocol [18]. Other layers of protocols can be used on top of HTTP. REST (REpresentational State Transfer) is an example of very popular, resource-oriented protocol [62]. The main idea of REST is very close to the spirit of HTTP, i.e., resources are identified by URLs and clients use HTTP methods (GET, PUT, DELETE, ...) on top of the resources, retaining their original semantics, to interact with them. An alternate protocol (Web services) uses exclusively POST requests on top of a single URL, transmitting the message in an HTTP body envelope.

1.1.4 Version Control

Version control has two aspects, depending whether it supports data (which can be queried) or code.

The first generation of code versioning systems consisted of SCCS [10] and RCS [32], CVS [9] and SVN [4] then followed. The newest generations are distributed and allow, e.g., local commits (git, bazaar, mercurial). These technologies are all text-based, and model the differences between versions as inserted or deleted lines of text.

Structured data (queryable) is commonly stored in temporal relational databases (Oracle Flashback [28], Microsoft ImmortalDB [70]).

1.2 Challenges

Each language used in the Web technology stack has (inherently) missing features. For example:

- JavaScript, which was originally designed to validate forms on the client using events to avoid unnecessary communication with the server, is very good for small computations, as well as for designing a user interface with events. However, it has neither been designed to navigate through the HTML code of Web pages, nor to process large amounts of data. JavaScript code for HTML navigation and data processing is thus relatively complex.
- Java does not support relational databases. When used in conjunction with SQL, an RDBMS-to-object mapping as well as an interface is required.

The JavaScript limitations are addressed with jQuery [20], which makes navigation in the HTML document easier. The Java limitations are addressed by extending SQL with more imperative abilities (e.g., PL/SQL [29]).

The limitations of each language is compensated by the high number of languages on the technology stack. However, this becomes an issue as well (*technology jungle*):

- A single Web application can contain, sometimes even in the same file, more that five or six different languages.
- There is a fundamental impedance mismatch between the various data models, requiring conversions when sending data around between tiers

- Experts have to be trained in each of the languages in use. This induces training costs.

This technology jungle prevents code mobility: the code is also very hard to move around, as it needs to be translated or compiled back and forth from a language to another language. The same is true for the data, for example, data in relational tables needs to be mapped to XML or objects, and conversely.

This second issue is addressed, for example, by compiling Java to JavaScript (GWT) or by using JavaScript on the server - but this brings back the first issue (missing features).

1.3 XML Technologies

The Web community is aware of these shortcomings and there already are workarounds. First, JavaScript now often embeds XPath expressions. XPath [36] is a W3C standard that has been designed to navigate XML trees, hence also the DOM tree, which is similar in spirit. This addresses the missing language features, but this makes one more language in the technology stack and it does not make the code more mobile. However, the idea of leveraging XML technologies is very appealing, for at least two reasons:

- XML is a cousin of HTML. Both represent data as a tree using markup. Furthermore, there is an alternate XML syntax for HTML known as XHTML.
- XML-based databases (e.g., eXist [12], Sedna [33]) have gained a lot of maturity over the last decade. It is possible to completely replace the relational database with an XML database (at least at the logical level: an XML database can use an RDMS, or a key-value-pair store, for its storage).

There are two main XML-processing languages among W3C recommendations, both based on XPath (and both Turing-complete): XSLT [41] and XQuery [48].

XSLT provides a means of transforming an XML document into another XML document. Its syntax is itself XML.

XQuery is more of a general-purpose language. It started on the database tier and was originally intended for querying XML in the same way SQL is used for querying relational databases. It consists of

- the core language (1.0) [48] as well as a newer version (3.0) on its way [40]

- an Update Facility [54], currently defined for XQuery 1.0.
- a Full-Text Extension [44], currently defined for XQuery 1.0.
- a working draft for a Scripting Extension [53] following the XQueryP proposal [52].

XQuery relies on the XPath 2.0 and XQuery 1.0 Data Model (XDM), the instances of which are sequences of items [61]. It comes with a rich set of functions and operators [72].

For a couple of years, with the growing interest in XML and the related technologies (XML Schema) as well as in the REST architecture, XQuery has also become popular on the middle-tier, as an alternative to PHP and Java EE. Implementations are MarkLogic [22], as well as Sausalito (28msec) [49] [69] [35]. It also runs on mobile devices [27].

There also exist proposals for XML-based versioning in various ways (an additional layer on top of a text-based versioning system [81] or an XML format for encoding several versions of an XML document in a compact way [78]).

All this indicates that XQuery and XML are very good candidates for an integrated architecture and a unified Web technology stack.

1.4 Contributions

The broader vision of this thesis is to show that it is possible to gain flexibility in three dimensions when programming Web applications:

Time Keep track of data history in a queryable way,

Space Allow the same data and code formats on all layers,

Scale Gain acceptance, and have programs scale, by using widespread Software Engineering paradigms.

without sacrificing performance, cost, and while retaining backward compatibility with existing technologies.

This implies that we should be able to compensate for missing language features and at the same time reduce the number of languages in the technology jungle and the impedance mismatch.

The general method to achieve this goal is to leverage existing W3C standards. XML is already accepted and used as the main means of exchanging data between tiers. XQuery is a very good candidate to program the Web and is already in use in server-side and database applications. It can be leveraged by extending the data model to add what is missing, and by defining new data structures and algorithms for an efficient implementation and its optimization.

The contributions of this thesis are five-fold:

Time - PUL composition Chapter 3 introduce an operator on XQuery Pending Update Lists for computing the difference between any two versions of XML data, given the differences between any two consecutive versions. This chapter is an updated and more complete version of a paper published at XML Prague in 2010 [64].

Time - an XML Time Machine Chapter 4 makes XQuery temporally savvy so that it becomes possible to keep track of and manipulate several versions of XML data. This enables temporal flexibility, as XML data will then be accessible in any past version.

Scale - an intuitive XQuery Scripting Grammar Chapter 5 introduce a new scripting grammar to make XQuery programming more intuitive to users familiar with Java or C++.

Scale - Object Orientation in XQuery Chapter 6 moves XQuery towards more object-orientation by adding behavior to XML nodes, enabling code reuse.

Space - XQuery in the Browser Chapter 7 makes XQuery available for client-side, browser programming. This includes a function library to handle events and styles, an adaptation of the Browser Object Model and an implementation. This enables code mobility, as XQuery then becomes available on all three tiers: database, business logic and client. This chapter is an updated version of a paper published at the World Wide Web conference in 2009 [66]. A demo was also given at SIGMOD 2008 [65].

These contributions are complementary to each other and can be used jointly to build Web applications with a unique declarative and object-oriented language on all layers, relying on persistent, versioned semi-structured data in the back-end.

Chapter 2

Short Introduction to XML technologies

This chapter provides an introduction to XML technologies. The reader familiar with these can skip to the next chapter.

2.1 The XML syntax

XML stands for eXtensible Markup Language. The XML 1.0 specification [38] provides a syntax for describing semi-structured data, and is the basis for all XML-related technologies. The main motivation for the use of this format is that it is widespread if not pervasive. Many parsers are already available. The XML syntax has strict requirements, so that a well-formed XML document is guaranteed to be accepted (and a non-well-formed XML document rejected) by an XML-compliant parser.

2.1.1 Markup

A well-formed XML document is text which matches certain syntactic conditions. Some parts of this text are parsed in a special way and are called markup. We (non-exhaustively) present the most widely used markup forms and are not going in all the subtleties behind them.

Element tags

XML is often referred to as a “tag soup”: Element tags are one of the forms of markup that can appear in an XML document.

There are two forms for tags:

- A pair consisting of an opening and a closing tag. The latter is distinguished with an initial slash.

  ```
  <element> (some XML) </element>
  ```

- Empty elements, distinguished with a final slash.

  ```
  <element/>
  ```

  They are equivalent to a pair with no content inside:

  ```
  <element></element>
  ```

Between these tags, there can be further character data and markup. In particular, Elements can be nested:

```
<element><other-element></other-element></element>
```

However, intertwining of elements is not allowed. This is not well-formed XML:

```
<element><other-element></element></other-element>
```

Opening tags and empty tags can contain attribute specifications. In a nutshell, these are key-value pairs. An element may not have two attributes with the same name.

```
<element attribute1="foo" attribute2="bar"></element>
```

Comments

Comments look like so:

```
<!-- This is a comment -->
```

Comments do not contain further markup, i.e., their content is not parsed in a special way. Comments should not be used for expressing data, as the XML specification does not require them to be forwarded to the main application by an XML parser.

Processing instructions

Processing instructions look like so:

```
<?my-app Instructions for my-app ?>
```

A processing instruction contains instructions that are passed to the application making use of the XML parser, but their semantics is outside of the scope of the XML specification.

Character references

Special characters can be expressed using their Unicode code points in character references. A character reference starts with &# and ends with a semi-colon. For example, Π encodes the Greek uppercase Π character (code point 3A0 in hexadecimal notation).

Entity references

Entities are resources such as text, XML fragments, or even non-XML resources (images, ...). Entity references are shortcuts to them. Like a character reference, an entity reference begins with an ampersand and ends with a semi-colon.

Most of the time, an entity reference in XML will be a reference to one of the five XML pre-defined entities, each of them being a single special character:

<	<
>	>
"	"
'	'
&	&

This allows to escape characters that would otherwise be recognized as markup. For example,

```
&lt;element>
```

will not be recognized as an opening tag.

2.1.2 Namespaces

URIs (Universal Resource Identifiers) are first-class citizens on the Web. They identify resources. For example, URLs are a kind of URI and are used to identify/locate Web pages.

In the XML world, URIs are used for identifying many other kinds of resources than just Web pages. XML namespaces are one of them. XML namespaces are defined in the Namespaces in XML 1.0 specification [50]. An XML namespace can be seen as a family name for element and attribute names.

As URIs are quite long, a namespace URI is mostly written only once to bind it to a prefix, and it is then this prefix which is used for all elements and attributes in the corresponding namespace.

The binding occurs using special attributes beginning with “xmlns”. For example:

```
<ns:element xmlns:ns="http://www.example.com"/>
```

binds the namespace “http://www.example.com” to the prefix “ns”. The element “ns:element” above is hence in the “http://www.example.com” namespace. “ns:element” is called a *QName*, it has the *prefix* ns and the *local name* element.

It is also possible to define a default namespace to avoid the use of prefixes. The following element is also in the “http://www.example.com” namespace:

```
<element xmlns="http://www.example.com"/>
```

If no default namespace is defined, then unprefixed names are in no namespace.

For simplicity and readability, we will make use of default namespaces (or even of no namespaces) as much as possible.

2.1.3 Well-formedness constraints

In addition to grammatical rules, XML defines a number of additional constraints for a document to be well-formed. One of them is for example that the document must contain a single root element tag pair (it cannot contain directly text, neither can it contain several top-level element tags).

2.1.4 XML Information Set (an XML data model)

XML as described above is pure syntax (text parsing according to a grammatical production and fulfilling some well-formedness constraints). The XML Information Set specification defines a logical data model on top of an XML document. It sees an XML document as a tree: the root is a document node, its child is the root element node, and each element node has several children (other element nodes, attribute nodes, comment nodes, text nodes, ...).

2.1.5 Valid XML documents

In addition to well-formedness, an XML document can fulfill additional conditions (such as which element names are allowed where). In this case, the XML document is said to be valid.

There are several technologies allowing the definition of such additional constraints. Natively, XML supports DTD (Document Type Definition) validation. DTD constraints can be directly written in the XML document.

DTD has limitations which were addressed with the more powerful XML Schema specification [37]. An XML Schema is written using the XML syntax. XML Schema bootstraps, i.e., it is especially interesting that an XML Schema itself must be valid against the “XML Schema Schema”.

For us, the most important aspect of the XML Schema specification is that it defines types. After validation, an XML document is expressed in an extended data model called Post Schema Validation Infoset (PSVI), in which each element and attribute has a type.

XML Schema is described in more details in chapter 6.

2.2 XQuery

XQuery is an XML Query language. It can process existing XML and produce new XML content. XQuery is to XML what SQL is to relational tables.

2.2.1 The XQuery Data Model

XQuery shares its data model with other languages like XPath (of which it is a superset) and XSLT. This data model is called XPath and XQuery Data Model and is abbreviated as XDM [61].

Each instance of the XDM is a sequence of items, where an item can be either an atomic item or an XML node.

An atomic item has an (atomic) type and an (atomic) value. For example, 2 is an **`xs:integer`**, 3.14 is an **`xs:decimal`** and "Hello" is an **`xs:string`**. There are also **`xs:untyped`** atomic items.

Unlike atomic items, XML nodes have an identity and can be compared (even ordered). XML nodes share a core set of accessors which give access to their children, their attributes, their name, ...

The empty sequence is a valid XDM instance. A single item is identical to a (singleton) sequence containing this item. Sequences are flat and cannot nest: (1, (), (2, 3)) is the same as (1, 2, 3).

2.2.2 XQuery 1.0 and XQuery 3.0

XQuery expressions

XQuery is a declarative language relying on full composability of its expressions. Each expression takes as input one or several sequences of items through its operands, and results in an output sequence of items.

The following is a (non-exhaustive) list of XQuery expressions.

String and number literals The hello world program in XQuery is simply:

```
"Hello, World"
```

which returns a sequence of one item with the type **`xs:string`** and the value "Hello, World". In string literals, entity and character references (like "&") are parsed. Number literals are also valid XQuery expression and return **`xs:integer`**s, **`xs:decimal`**s or **`xs:double`**s.

Variables Variables in XQuery are preceded with a dollar sign: $var. A variable is *bound* to a sequence of items. The terminology of “bound” is quite important: in XQuery core, variables cannot be reassigned.

Function calls XQuery supports functions, including user-defined functions but also built-in functions. The XQuery Functions and Operators specification [72] contains a list that all implementations should support. One of them is the doc function.

```
doc("file.xml")
```

returns the top-level document node resulting from parsing the document file.xml and converting it to an XDM instance.

Path navigation Given a node (for example a document node), it is possible to navigate through an XML document. For example, if file.xml contains:

```
<Stock>
  <Name>XML Corp stock</Name>
  <Symbol>XMLC</Symbol>
  <Company>XML Corporation</Company>
  <Country>U.S.A.</Country>
  <Country>Switzerland</Country>
  <Quote>3.14</Quote>
</Stock>
```

then the following path expression:

```
doc("file.xml")/Stock/Country[2]
```

returns

```
<Country>Switzerland</Country>
```

Context item Like when navigating with directories on a file system, one can refer to the context item (analogous to the current directory) with a dot . and to the parent item with a double dot .. in a path expression.

Parenthesized expressions Parentheses may be used to override precedence and carry no further semantics.

Comma expressions A comma concatenates the sequences output by its operands into a single sequence, as in (1, 2), (3, 4).

Arithmetic expressions XQuery supports the arithmetic operators +, - (unary and binary), *, div, idiv (integer division), mod.

If one of the operands is an XML node, something particular in XQuery happens, which is called *atomization*. An atomic value is extracted from the node and used

for the operation. For example, in the following query, the XML node is atomized to the **xs:untyped** atomic item "2" and then converted to an **xs:double**.

```
<tag>2</tag> + 1
```

The result of the operation is then the **xs:double** 3.

It is also possible to explicitly atomize a node with the function **data**.

Comparison expressions XQuery supports

- binary atomic value comparison operators: eq (equality), le (lower or equal), lt (lower than), ge (greater or equal), gt (greater than)
- binary node comparison operators: is (node identity), <<, >> (document order comparison)
- binary sequence comparison operators which return true if at least one item in the left sequence and one item in the right sequence fulfill the comparison: =, <, >, <=, >=.

For example, the following query returns true if the stock bound to the variable $stock has at least one Country element with the content “Switzerland”.

```
$stock/Country = "Switzerland"
```

Logical expressions XQuery supports the logical operators *or* and *and* as well as **not** (the latter as a function call).

If one of the operands is not an **xs:boolean**, then its so-called *Effective Boolean Value (EBV)* is taken. For example, the EBV of a sequence beginning with an XML node is true, which allows to test whether a sequence of nodes is empty or not with a very simple syntax.

Node constructors XQuery can build new XML nodes with a syntax very similar to that of XML. It is possible to switch back to XQuery mode using curly braces.

```
<Mystock>
  <Code>{ doc("file.xml")/Stock[1]/data(Symbol) }</Code>
  <Value>{ doc("file.xml")/Stock[1]/data(Quote) }</Value>
</Mystock>
```

FLWOR expressions One of the most powerful constructs in XQuery is the FLWOR expression (For Let Where Order by Return), which is analogous to SQL’s SELECT FROM WHERE.

The following query builds an XHTML table with two columns (stock code and quote) by iterating over all stocks from the input document that are exchanged at least in Switzerland (note the use of the = operator).

```
<table>
  <th>
    <td>Code</td>
    <td>Value</td>
  </th>
{
  for $stock in doc("file.xml")/Stock
  where $stock/Country = "Switzerland"
  return
  <tr>
    <td>{ $stock/data(Symbol) }</td>
    <td>{ $stock/data(Quote) }</td>
  </tr>
}
</table>
```

Control flow expressions XQuery supports conditional, typeswitch, switch, try/catch expressions. The following query has the same semantics as the query above, but uses a conditional expression instead of the where clause.

```
<table>
  <th>
    <td>Code</td>
    <td>Value</td>
  </th>
{
  for $stock in doc("file.xml")/Stock
  return
  if ($stock/Country = "Switzerland") then
    <tr>
      <td>{ $stock/data(Symbol) }</td>
      <td>{ $stock/data(Quote) }</td>
    </tr>
  else ()
```

```
    }
    </table>
```

QNames in XQuery

Like XML, XQuery supports namespaces. This means that it supports QNames in path navigation, but also that function and variable names are QNames as well. For simplicity, we will use the default namespace as often as possible, so that there will be no prefix. For functions, the default namespace is the namespace containing most built-in XQuery functions. For convenience, XQuery defines a namespace for defining functions local to a query and binds it to the prefix "local". We will use it when defining functions.

User-defined functions

XQuery allows programmers to define their own functions. User-defined functions can either be defined in library modules (containing only variables and functions), or in a main module (containing variables, functions, and finally a query to be evaluated).

```
declare function local:add($x, $y as xs:integer) {
  $x + $y
}
```

Parameter types and return types may be omitted (like for the parameter $x above), in which case the most general type is assumed (a sequence of any items `item()*`).

Chapter 6 gives more details about typing in XQuery.

2.2.3 XQuery Update Facility

XQuery is declarative, only reads input XDM instances from documents and collections to output an XDM instance (which can be part of the input documents, or new). Its only side effects are the creation of new XML nodes. The XQuery Update Specification [54] allows to declaratively prepare a list of updates (called a Pending Update List) that is to be applied to an XML instance (*after* the execution of the updating query, i.e., there are no side effects during query evaluation other than the creation of new nodes).

It is described in more details in Chapter 3

2.2.4 XQuery Scripting Extension

XQuery Scripting allows an XQuery program to have side effects during its execution. It allows the application of Pending Update Lists during the execution of a program (which is a side effect) and gives control about the order in which side-effecting expressions are evaluated.

The XQuery Scripting extension makes XQuery a general-purpose programming language, as it allows side effects to the outside world. For example, it supports side-effecting HTTP communication (POST, PUT, DELETE requests), access to the file system, event binding, etc.

It is described in more details in Chapter 5

Part I

Time

Chapter 3

Composing Pending Tree Updates

This chapter introduces a composition operator on Pending Update Lists. Pending Update Lists model changes that can be applied to a tree in order to create a new tree (Pending Update List application). This operator composes two Pending Update Lists to create a single Pending Update List, the application of which is equivalent to the application of one operand followed by the application of the other operand. The challenge is that the paradigm behind the application of a Pending Update List must be formalized in order to provide a proof of correctness.

3.1 Introduction

The XQuery Update Facility extends XQuery in order to allow updates against documents. However, instead of each updating expression actually performing the update, the XQuery Update Facility is still a declarative language, in which updating expressions are side-effect free and return a list of pending changes, called a Pending Update List (PUL). A Pending Update List can be *applied*, which means that the changes it contains are made effective, changing the original tree in a new tree.

This suggests that Pending Update Lists are ideal candidates for modeling changes to be made against a tree, for example in order to ship them over HTTP to update an XML resource.

In practice though, several updates can be made successively (for example with XQuery Scripting), i.e., several PULs can be applied successively, and shipping each of them over the network consumes a lot of network bandwidth. Ideally, one could write a single, equivalent XQuery Update program which directly generates all changes in one shot,

but this can become very cumbersome for the programmer (for example for implementing an algorithm which generates updates conditionally to former update effects).

Instead, we suggest that the burden of summarizing all changes modeled by an initial sequence of PULs be algorithmically automated, and provide a way of building a single PUL from a sequence of PULs. The XQuery Update specification has only been a recommendation since 2011, so that this had not been done before. The contribution of this chapter is to introduce a new associative binary operator on PULs, called PUL composition. This in effect turns the set of PULs into a monoÃ¯d.

3.2 Introduction to the XQuery Update Facility

This section gives an overview of the existing XQuery Update Facility recommendation.

An XQuery program is side-effect-free and returns an XDM instance. An XQuery Update program in itself is also side-effect free and returns an XDM instance and a PUL. In the current specification, it is forbidden for both to be non-empty, which means that either a result is returned with no pending changes, or pending changes are returned without a result. The PUL is applied after the execution of the program.

The XQuery Update Facility specification is based on a snapshot semantics. During the execution of the program, the input XML trees are frozen (*snapshot*), and updates are never visible until after the end of the execution, when the PUL is actually applied.

3.2.1 Pending Update Lists

A Pending Update List is an ordered list of update primitives. Update primitives can be one of the following:

Inserting One of

upd:insertInto
upd:insertBefore
upd:insertAfter
upd:insertIntoAsFirst
upd:insertIntoAsLast
upd:insertAttributes.

These update primitives have a target node as well as new content (XDM instance) to insert.

Deleting `upd:delete`. These update primitives have a target node to be deleted.

Replacing One of

`upd:replaceNode`
`upd:replaceValue` (of an attribute)
`upd:replaceElementContent` (to replace the content of an element with text).

These update primitives have a target as well as a replacement XDM instance.

Rename `upd:rename` to rename a target node to a new name (QName).

Put `upd:put` to bind an XDM instance to a URI, so that it can later be retrieved with `doc`.

In the remainder of this chapter, we will not consider the `upd:put` update primitive, as it is not related to directly modifying an XDM instance.

3.2.2 Updating expressions

In order to produce these update primitives and PULs that contain them, the XQuery Update Facility introduces new expressions (called primitive updating expressions), of which we now give a few examples.

The following expression creates an `upd:insertIntoAsFirst` update primitive (and a PUL that contains it):

```
insert node <a/> as first into doc("file.xml")/root
```

The following expression creates an `upd:delete` update primitive (and a PUL that contains it):

```
delete nodes doc("file.xml")/Stocks
                  /Stock[@country = "Switzerland"]
```

These primitive PUL blocks are then merged when using (existing) comma expressions:

```
[Expr1], [Expr2]
```

will return a single PUL merging the PULs return by each of the two expressions.

Other constructs such as conditional expressions, typeswitch expressions, ... are composable with updating expressions (in which case they are updating expressions as well). Others (such as path expressions) are not and their operand may not be updating expressions. Non-updating expressions are called simple expressions.

3.3 PUL Composition

This section introduces the basics behind the new PUL composition.

The difficulty behind PUL composition is that the PULs might be interdependent, e.g., the target of a PUL might be in the contents previously inserted by another PUL. Had they been completely disjoint, they could just be merged (with *upd:mergeUpdates*, described in the XQUF specification).

We have here a time component which has to be taken into account. At first, it seems that deltas must be expressed by sequences of PULs (e.g., (a) and (b) on Figure 3.1). But (c) shows that, for this example, a single, equivalent PUL can be given.

Again, PUL composition is different from PUL merging (*upd:mergeUpdates*), since the PULs are possibly not independent, i.e., they are not applied to the same snapshot.

First, we will give an operational definition of PUL composition, which is an abstract algorithm which describes *how* to compose a sequence of PULs. Then, we will give a formal definition of the composition operator with a proof of correctness.

A local PUL is maintained, which summarizes all local changes composed so far. It is normalized (see Section 3.6), and each time a new PUL is applied, a copy of this new PUL (right operand) is composed with the local PUL (left operand). This is described in Section 3.6.

The idea behind the PUL composition operator is very simple. For each update primitive in the right-hand PUL, there are two cases:

Accumulation Either the target of this update primitive was already here before the left-hand PUL was generated, in which case the update primitive is accumulated against it. Accumulation is shown on Figure 3.2 and detailed in Section 3.5.1.

Aggregation Or its target is in the contents of the left-hand PUL, in which case the extended semantics of the update primitive is made effective. Aggregation is shown on Figure 3.1 and detailed in Section 3.5.2.

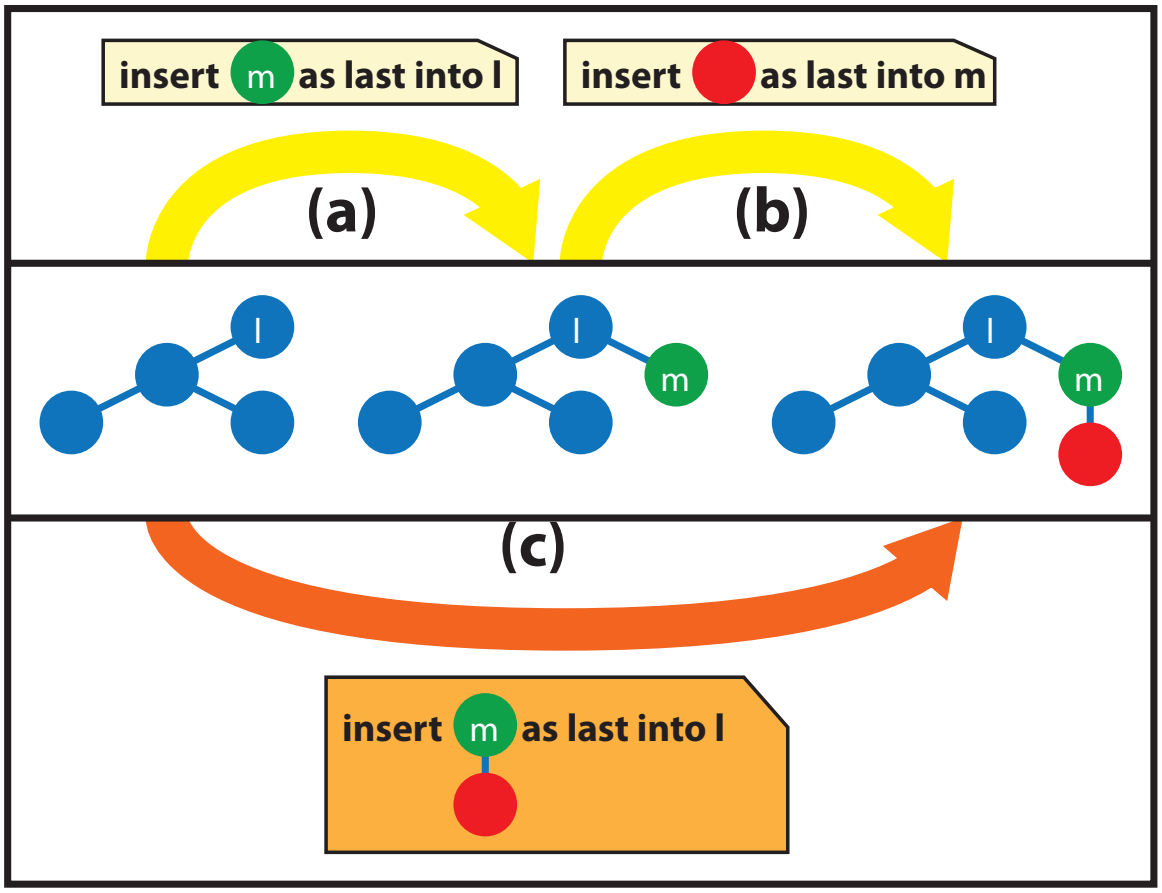

Figure 3.1: *Two PULs (a, b) can be summarized in a single PUL (c) (Aggregation Case)*

3.4 Normalized PUL

The composition operator works under the assumption that at least the left-hand PUL is *normalized*. This section defines PUL normalization.

A normalized PUL means that, without loss of generality, for each target, there is at most one update primitive of each kind with this target:

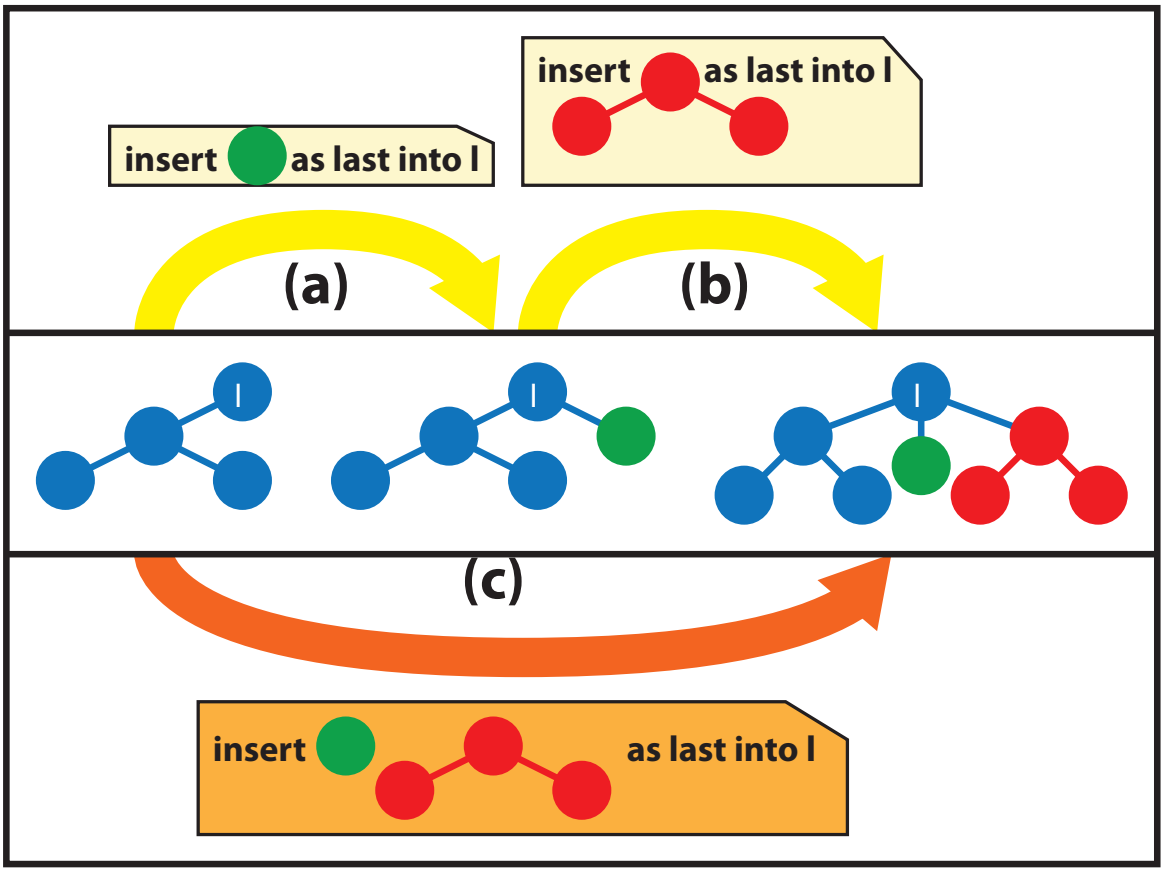

Figure 3.2: *Two PULs (a, b) can be summarized in a single PUL (c) (Accumulation Case)*

`upd:insertBefore`
`upd:insertAfter`
`upd:insertIntoAsFirst`
`upd:insertIntoAsLast`
`upd:insertAttributes`
`upd:replaceNode`
`upd:replaceValue`
`upd:replaceElementContent`
`upd:delete`
`upd:rename`

Note that `upd:put` is out of the scope of this thesis, `upd:insertInto` deserves a special treatment and can always be replaced with another inserting update primitive.

The uniqueness constraint already holds for replacing and renaming update primitives in the specification, and in case there are several inserting update primitives of the same kind with the same target, the XQuery Update Facility specification says that "ordering of nodes within each group is preserved but ordering among the groups is implementation-dependent." Whenever a PUL contains several inserting update primitives of the same kind sharing the same target, this allows us to group the contents of these update primitives inside a single update primitive of that kind, in an implementation-dependent way, without altering the semantics of the PUL.

It is not necessary for the right-hand PUL to be normalized. In any case, the composition operator produces a normalized PUL.

3.5 Operational definition of PUL composition

We now give details about accumulation and aggregation of an update primitive (from the right-hand PUL) against a PUL (the left-hand PUL, thereafter “the PUL” for concision). These two concepts are defined in a destructive way, meaning, as changes to the left-hand PUL, but algebraically a copy thereof can be made before.

3.5.1 Accumulation

When the target of an update primitive was already here before the PUL was generated (i.e., it is not to be found anywhere in the contents of this PUL - to put it simply, it is not encircled in black on the Figures), then this update primitive is accumulated against the left-hand PUL. It is mostly like merging them (upd:mergeUpdates), with the additional constraint that PUL remains normalized (Figure 6).

For **`upd:insertBefore`** and **`upd:insertAttributes`**, the contents of the update primitive are inserted at the end of the contents of the update primitive of the same kind in the PUL (or the update primitive is inserted in the PUL if not available).

For **`upd:insertIntoAsLast`**, the contents of the update primitive are inserted at the end of the contents of the update primitive of the kind **`upd:replaceElementContent`** in the PUL if it is available. Otherwise, they are inserted at the end of the contents of the updating primitive of the kind **`upd:insertIntoAsLast`** (or the update primitive is inserted in the PUL if not available).

For **`upd:insertAfter`**, the contents of the update primitive are inserted at the begin-

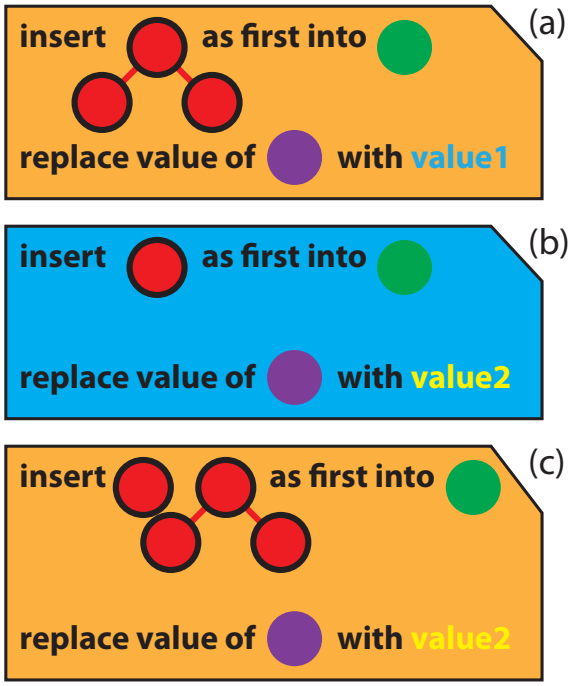

Figure 3.3: *Accumulating the update primitives in (b) against a PUL (a) leads to the new PUL (c).*

ning of the contents of the update primitive of the same kind in the PUL (or the update primitive is inserted into the PUL if not available).

For `upd:insertIntoAsFirst` , the contents of the update primitive are inserted at the beginning of the contents of the update primitive of the kind `upd:replaceElementContent` in the PUL if it is available. Otherwise, they are inserted at the beginning of the contents of the updating primitive of the kind `upd:insertIntoAsFirst` (or the update primitive is inserted in the PUL if not available).

For `upd:delete, upd:replaceNode`, the update primitive is inserted into the PUL if there is not already an update primitive with the same target and the same kind in the PUL. Otherwise, an error is raised (this should never happen during the normal execution of an XQuery program).

For `upd:replaceValue, upd:replaceElementContent`, the update primitive is inserted into the PUL if there is not already an update primitive with the same target and the same kind in the PUL. Otherwise, the contents (or string) of the update primitive in the PUL are replaced with the contents of the update primitive being accumulated.

For `upd:rename`, the update primitive is inserted into the PUL if there is not already an *upd:rename* update primitive with the same target in the local PUL. Otherwise, the name mentioned in the update primitive of the PUL is replaced with the name of the update primitive being accumulated.

3.5.2 Aggregation

When the target of an update primitive was put into place by the PUL, it means that it is to be found somewhere in the contents of this PUL (to put it simply, it is encircled in black on the figures). In this case, the update primitive is *aggregated* against the PUL (Figure 3.4).

The semantics of aggregation is the same as the semantics of applying an update primitive, with just the following modifications for `upd:insertBefore, upd:insertAfter` and `upd:replaceNode`.

For these three update primitives, the specification says that the parent property of the

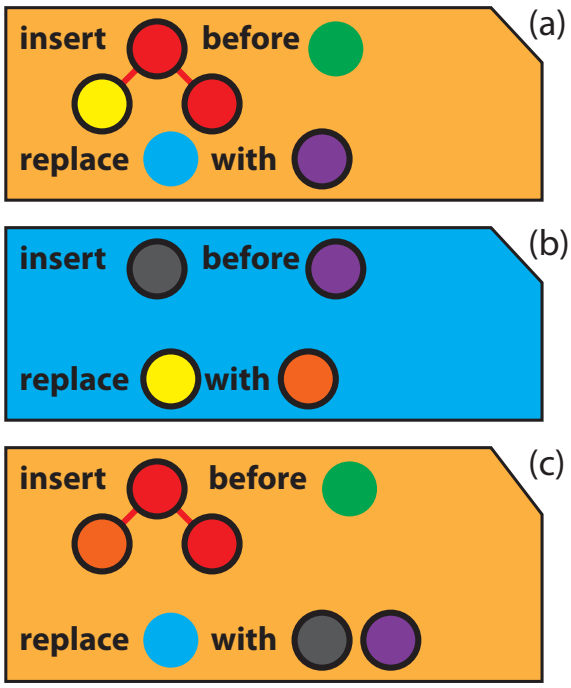

Figure 3.4: *Aggregating the update primitives in (b) against the local PUL (a) leads to the new local PUL (c).*

target must be non-empty (because it seems to make no sense to insert a node before another if the latter is the root of the tree). We relax this constraint. However, if the parent property is empty, the target must be in the contents of an update primitive in the PUL (encircled in black). This corresponds to (a) on Figure 3.4.

In case the parent property of the target is empty, the semantics of applying these three update primitives is extended as follows: for `upd:insertBefore` (`upd:insertAfter`), the content of the update primitive is inserted right before (after) the target in the contents of the update primitive of the PUL to which this target belongs (like on Figure 3.4); for `upd:replaceNode`, the content of the update primitive replaces the target of the update primitive of the PUL to which this target belongs.

3.6 Extension to XQuery Scripting

This section gives additional semantics in order to introduce PUL composition in Scripting programs, which apply several updates.

In XQuery, there is a dynamic context which contains dynamic information about the ex-

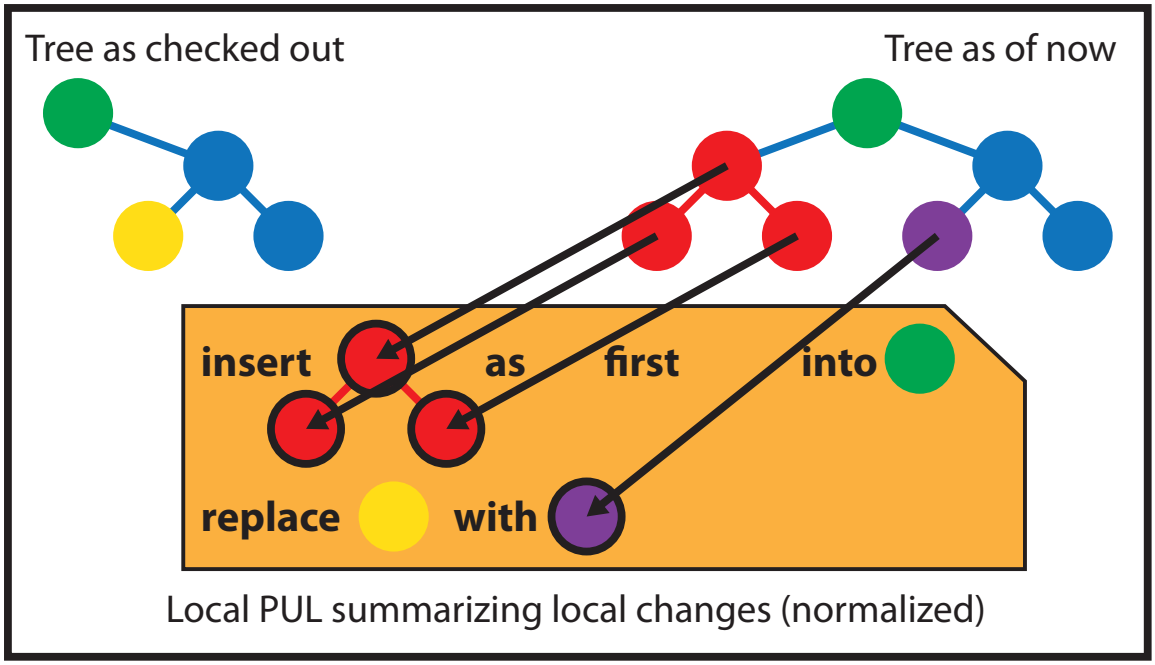

Figure 3.5: *The local PUL: overview*

ecution of a program. We extend the dynamic context with a local PUL, which contains information about all changes which have been made since a certain point in time (e.g., the beginning of the program). The local PUL can be seen as the delta between this point in time and now. Each time an update is applied, the composition operator is used with the local PUL on the left-hand-side, and a copy of the PUL being applied on the right-hand side. The local PUL is kept normalized when aggregating or accumulating update primitives, so that it will still be normalized at the end of the composition.

In addition, the XQuery engine must keep track, for newly created nodes since the beginning of the program, of their counterpart in the local PUL, which indicates how they were created (see Figure 3.5). That way, when a PUL is to be applied, with targets in the data being processed, a copy of this PUL can be made with the corresponding targets in the local PUL, in order to facilitate PUL composition.

On Figure 3.5, we take the convention that the nodes encircled in black in update primitive contents or targets are counterparts to nodes in the processed data. The identities of these copies are distinct from those of the original nodes.

The XQuery Scripting Extension specification introduces apply expressions, which apply a PUL. In an apply expression, after each operand is evaluated, the PUL it returns is applied with `upd:applyUpdates`. We perform PUL composition right here. Whenever a PUL is applied, a copy of it is also composed with the local PUL. This is described on

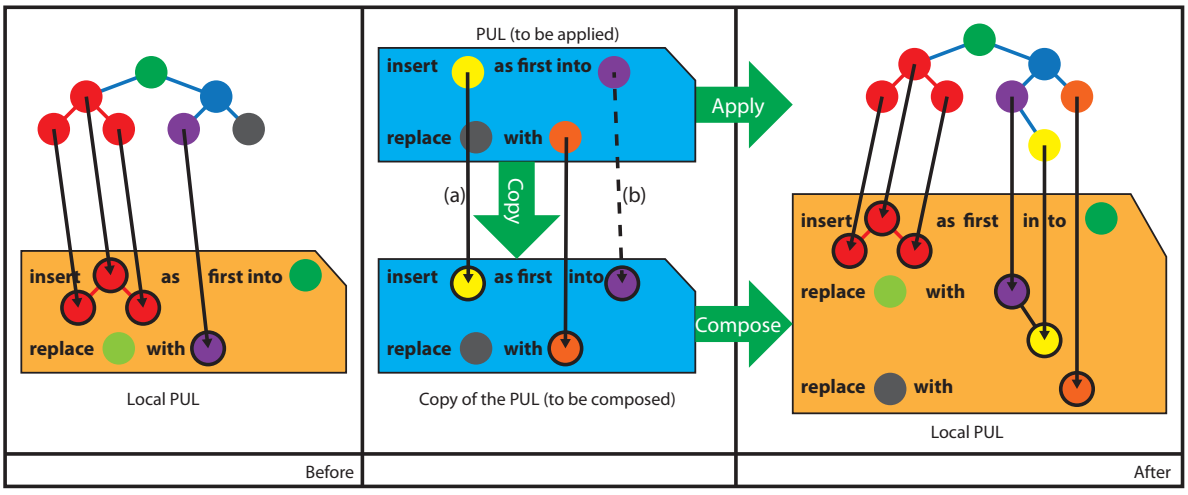

Figure 3.6: *PUL: applying, copying and composing. (a) shows generated provenance information, (b) shows how provenance information is used to retrieve the target in the local PUL.*

Figure 3.6.

More precisely, we extend the semantics of Apply Expressions as follows. For each operand, before the PUL is applied:

Copy This PUL is copied to a new PUL, where:

- The contents of the update primitives in the PUL are copied as well, and the engine remembers that such copies were made and keeps track of their location ((a) on Figure 3.6). These copies will eventually land in the local PUL, whereas the original nodes will land in the processed data.
- The targets of the update primitives in the PUL (which are references to processed data) are replaced in the copy with a reference to their counterparts in the local PUL. ((b) on Figure 3.6).

Composition The copied PUL (right-hand-side) is then composed with the local PUL (left-hand-side). The result of this composition is the new local PUL.

Then the PUL is applied (with `upd:applyUpdates`, as defined in the XQUF specification).

3.7 Formal Model

Finally, this section gives a formal definition of the PUL composition operator, and a proof of correctness on a formal modelization of the PUL framework.

3.7.1 References

In the formalization, the XML content is represented as forests. Each XML node has a reference taken from an alphabet $\mathcal{L}$. PULs reference target XML nodes using these references.

In practice, references are unique, but this assumption is not needed. We hence simplify and allow several nodes to have the same references (in which case an update primitive with a given target reference will be applied to all nodes with this reference). This will in particular be true for unique references.

3.7.2 Forests

We define the set of all forests $\mathcal{F}(\mathcal{L}, \mathcal{N})$ given an alphabet of references $\mathcal{L}$ and an alphabet of names $\mathcal{N}$ recursively as follows.

Empty forest The empty forest is a forest: $[O] \in \mathcal{F}(\mathcal{L}, \mathcal{N})$

Tree building A forest can be put under a node with reference l and name n (meaning, the trees in this forest become children of this node) to form a tree, which is a (singleton) forest. We represent this with a fraction: $\forall l \in \mathcal{L}$, $n \in \mathcal{N}$, $\forall [f] \in \mathcal{F}(\mathcal{L}, \mathcal{N})$, $\frac{l, n}{[f]} \in \mathcal{F}(\mathcal{L}, \mathcal{N})$

Concatenation The concatenation of two forests is also a forest: $\forall [f], [g] \in \mathcal{F}(\mathcal{L}, \mathcal{N})$ $[f][g] \in \mathcal{F}(\mathcal{L}, \mathcal{N})$

Note that we use square brackets to denote forests: $[f]$, whereas references l and names n are letters alone.

3.7.3 Update Primitives

We model the following update primitives:

insert as first The trees of a forest can be inserted as the first children of a node.

$\mathcal{IF}(\mathcal{L}) = \{\textbf{insert } [f] \textbf{ as first into } l, l \in \mathcal{L}, [f] \in \mathcal{F}(\mathcal{L}, \mathcal{N})\}$

insert as last The trees of a forest can be inserted as the last children of a node.

$\mathcal{IL}(\mathcal{L}) = \{\textbf{insert } [f] \textbf{ as last into } l, l \in \mathcal{L}, [f] \in \mathcal{F}(\mathcal{L}, \mathcal{N})\}$

insert before The trees of a forest can be inserted before a node.

$\mathcal{IB}(\mathcal{L}) = \{\textbf{insert } [f] \textbf{ before } l, l \in \mathcal{L}, [f] \in \mathcal{F}(\mathcal{L}, \mathcal{N})\}$

insert after The trees of a forest can be inserted after a node.

$\mathcal{IA}(\mathcal{L}) = \{\textbf{insert } [f] \textbf{ after } l, l \in \mathcal{L}, [f] \in \mathcal{F}(\mathcal{L}, \mathcal{N})\}$

delete A node can be deleted.

$\mathcal{D}(\mathcal{L}) = \{\textbf{delete } l, l \in L\}$

replace node A node can be replaced with a forest.

$$\mathcal{R}(\mathcal{L}) = \{\textbf{replace } l \textbf{ with } [f], l \in \mathcal{L}, [f] \in \mathcal{F}(\mathcal{L}, \mathcal{N})\}$$

replace value The children of a node can be replaced with a forest.

$$\mathcal{V}(\mathcal{L}) = \{\textbf{replace value of } l \textbf{ with } [f], l \in \mathcal{L}, [f] \in \mathcal{F}(\mathcal{L}, \mathcal{N})\}$$

rename A node can be renamed with a new name n.

$$\mathcal{N}(\mathcal{L}) = \{\textbf{rename } l \textbf{ to } n, l \in \mathcal{L}, n \in \mathcal{N}\}$$

A PUL is then defined as a list of update primitives, i.e., the (unrestricted) set of PULs is:

$(\mathcal{IF}(\mathcal{L}) \cup \mathcal{IL}(\mathcal{L}) \cup \mathcal{IB}(\mathcal{L}) \cup \mathcal{IA}(\mathcal{L}) \cup \mathcal{D}(\mathcal{L}) \cup \mathcal{R}(\mathcal{L}) \cup \mathcal{V}(\mathcal{L}) \cup \mathcal{N}(\mathcal{L}))*$

There are constraints on how a PUL looks like, though. Within a PUL, the XQuery Update specification:

- forbids two renaming, two node-replacing or two value-replacing update primitives with the same reference.
- allows two inserting update primitives of the same kind and with the same reference, but "if multiple groups of nodes are inserted by multiple insert expressions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation-dependent."

In our formalism, we group inserting update primitives of the same kind and with the same reference as follows (the initial order is arbitrary, since it is implementation-dependent, but once the grouping has been made the order is well-defined):

- (**insert** $[f]$ **as first into** l, **insert** $[g]$ **as first into** l) $\rightarrow$ **insert** $[f][g]$ **as first into** l
- (**insert** $[f]$ **as last into** l, **insert** $[g]$ **as last into** l) $\rightarrow$ **insert** $[f][g]$ **as last into** l
- (**insert** $[f]$ **before** l, **insert** $[g]$ **before** l) $\rightarrow$ **insert** $[f][g]$ **before** l
- (**insert** $[f]$ **after** l, **insert** $[g]$ **after** l) $\rightarrow$ **insert** $[f][g]$ **after** l

Finally, two deleting update primitives with the same reference are superfluous and equivalent to a single one.

- (**delete** l, **delete** l) $\rightarrow$ **delete** l

With these considerations, we can assume that there is at most one update primitive of each kind with the same target reference. We call $\mathcal{P}(\mathcal{L})$ the set of (so-called normalized) PULs satisfying this assumption.

We are actually more general than the XQuery Update specification in the following ways:

- We allow the value-replacing update primitive to have any forest (not just a single tree) in its content.
- We allow situations like

  insert a **as first into** b, **insert** c **as first into** a

  which would never appear in XQuery Update programs. For these we apply a strict snapshot semantics, i.e., update primitives do not see the effects of each other, they all only see the nodes in the original XML forest being updated. In the example above, the second update primitive, targeting node a, will not see the node a inserted by the first one.

3.7.4 Partition of PULs with respect to targets

The update primitives in a PUL can be partitioned according to their targets: a PUL p can be expressed as $(p_l)_{l\in\mathcal{L}}$ where each p_l is a list of update primitives of distinct kinds with the same target l.

Defining a PUL p is equivalent to defining p_l for each reference l.

It is always possible to assume that p_l contains each of the inserting update primitives (if it is absent, it is as it were present with an empty content). This means that each p_l contains each of the four kinds of insert update primitives, and each of the other four kinds (delete, replace, replace value, rename) may be present or not.

We introduce the following getters on the contents of a partition p_l:

replace node $R(p_l)$ is the content replacing l, i.e., **replace** l **with** $R(p_l) \in p_l$. It is set to $\perp$ if there is no such update primitive in p_l.

delete $D(p_l)$ is set to $\perp$ if there is no such update primitive (**delete** l) in p_l, and to $\top$ if it is present.

replace value $V(p_l)$ is the content replacing the value of l, i.e., **replace value of** l **with** $V(p_l) \in p_l$. It is set to $\perp$ if there is no such update primitive in p_l.

insert before $B(p_l)$ is the content inserted before l, i.e., **insert** $B(p_l)$ **before** $l \in p_l$

insert after $A(p_l)$ is the content inserted after l, i.e., **insert** $A(p_l)$ **after** $l \in p_l$

insert first $F(p_l)$ is the content inserted as first into l, i.e., **insert** $F(p_l)$ **as first into** $l \in p_l$

insert last $L(p_l)$ is the content inserted as last into l, i.e., **insert** $L(p_l)$ **as last into** $l \in p_l$

rename $N(p_l)$ is set to the new name for l, i.e., **rename** l **to** $N(p_l) \in p_l$. It is set to $\perp$ if there is no such update primitive in p_l.

3.7.5 The PUL application operator

PULs can act on (update) a forest. This is formalized by an operator $.$ which takes a PUL p and a forest $[f]$, and returns the forest obtained by applying p on $[f]$.

Since forests are defined by induction, the $.$ operator is also defined by induction on $\mathcal{F}(\mathcal{L}, \mathcal{N})$ as follows:

Empty forest Applying a PUL to the empty forest does not alter it: $p.[O] = [O]$

Concatenation Applying a PUL to a concatenation of two forests is the same as applying it to each of the two forests, and concatenating the results$p.([f][g]) = [p.f][p.g]$

Tree building The results of applying a PUL to a tree, $p.\frac{l,n}{[f]}$, is defined depending on the content of the PUL p (the first matching case is used):

Case R if $R(p_l) \neq \perp$, i.e., p replaces the tree (but the insert-before and insert-after update primitives have an effect, too):

$$p\frac{l,n}{[f]} = [B(p_l)][R(p_l)][A(p_l)]$$

Case D otherwise if $D(p_l) \neq \perp$, i.e., p deletes the tree (but the insert-before and insert-after update primitives have an effect, too):

$$p\frac{l,n}{[f]} = [B(p_l)][A(p_l)]$$

Case V otherwise if $V(p_l) \neq \perp$, i.e., p replaces the children of the tree (since the root remains, the effects of a renaming are visible. The operator | is defined below):

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l,N(p_l)|n}{[V(p_l)]}[A(p_l)]$$

Case I otherwise in all other cases, new siblings (after, before) and new children (first, last) are inserted, and the root is possibly renamed:

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)]}[A(p_l)]$$

The renaming operator | is defined for convenience as

$$a|b = \text{ if } a \neq \perp \text{ then } a \text{ else } b$$

It means that b is renamed to a if a is not undefined. Note that it is associative:

$$a|(b|c) = (a|b)|c$$

so that one can write $a|b|c$ without ambiguity. $a|b|c$ is actually equal to the first name (a, b or c) that is not $\perp$.

3.7.6 PUL Composition theorem

We now state the main PUL composition theorem.

Theorem 1. *There exists a composition operator* $*$ *such that*

$$\textit{for any } p, p' \in \mathcal{P}(\mathcal{L}) \textit{ and } f \in \mathcal{F}(\mathcal{L},\mathcal{N}), q.(p.f) = (q * p).f$$

In other words, applying several PULs in a row is equivalent to applying one single PUL, which can be computed using the composition operator. PULs can hence be composed.

3.7.7 Proof of the PUL composition theorem

Explicit definition of the composition operator

The proof is done by construction, i.e., we give a definition of a candidate operator for $*$ and then prove that it fulfills the condition.

Given two PULs p and q, the result $q * p$ of applying the operator is defined by defining $(q * p)_l$ for each l as follows:

replace node If p replaces l, then q is applied to the replacement. If p deletes l, nothing is done, if q replaces l, this replacement is forwarded to $q * p$.

$$R((q*p)_l) = \begin{cases} q.R(p_l) & R(p_l) \neq \bot \\ \bot & D(p_l) \neq \bot \\ R(q_l) & \text{otherwise} \end{cases}$$

delete If p or q deletes l, this deletion is forwarded to $q * p$.

$$D((q*p)_l) = \begin{cases} \top & D(p_l) \neq \bot \text{ or } D(q_l) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

replace value If q replaces the value of l, it is forwarded to $q * p$. If p replaces the value of l, q is applied to this new value, and possible insertions as first or last by q are forwarded as well in addition to the new value.

$$V((q*p)_l) = \begin{cases} V(q_l) & V(q_l) \neq \bot \\ F(q_l)q.V(p_l)L(q_l) & V(p_l) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

insert before If p makes an insert before l, q is applied on it. An insert-before by q is only forwarded if p did not delete or replace l.

$$B((q*p)_l) = \begin{cases} q.B(p_l) & R(p_l) \neq \bot \text{ or } D(p_l) \neq \bot \\ q.B(p_l)B(q_l) & \text{otherwise} \end{cases}$$

insert after If p makes an insert after l, q is applied on it. An insert-after by q is only forwarded if p did not delete or replace l.

$$A((q*p)_l) = \begin{cases} q.A(p_l) & R(p_l) \neq \bot \text{ or } D(p_l) \neq \bot \\ A(q_l)q.A(p_l) & \text{otherwise} \end{cases}$$

insert first If p inserts children (first) into l, q is applied on it and an insert-as-first by q is forwarded as well.

$$F((q * p)_l) = F(q_l)q.F(p_l)$$

insert last If p inserts children (last) into l, q is applied on it and an insert-as-last by q is forwarded as well.

$$L((q * p)_l) = q.L(p_l)L(q_l)$$

rename The leftmost rename wins.

$$R((q * p)_l) = N(q_l)|N(p_l) = \text{if } N(q_l) \neq \perp \text{ then } N(q_l) \text{ else } N(p_l)$$

Another way of seeing it is that seven cases are possible. They are shown below. The update primitives that are ineffective are shown in gray (for example, a deleting update primitive is ineffective when there is a replacing update primitive).

Case 1-*-R if $R(p_l) \neq \perp$, i.e., p replaces l, q is applied recursively to all nodes coming from p:

insert $[q.B(p_l)]$ **before** l
insert $[q.A(p_l)]$ **after** l
replace l **with** $[q.R(p_l)]$
(**delete** l)
(**replace value of** l **with** $[V(q_l)]$)
(**rename** l **to** $N(q_l)|N(p_l)$)

Case 2-*-D otherwise if $D(p_l) \neq \perp$, i.e., p deletes l, q is applied recursively to all nodes coming from p:

insert $[q.B(p_l)]$ **before** l
insert $[q.A(p_l)]$ **after** l
delete l
(**replace value of** l **with** $[V(q_l)]$)
(**insert** $[F(q_l)][q.F(p_l)]$ **as first into** l)
(**insert** $[q.L(p_l)][L(q_l)]$ **as last into** l)
(**rename** l **to** $N(q_l)|N(p_l)$)

Case 3-R-VI otherwise if $R(q_l) \neq \perp$, i.e., q replaces l, only nodes inserted after and before by p are kept, q is applied on them, and possibly q also inserts nodes before and after l:

insert $[q.B(p_l)][B(q_l)]$ **before** l,
insert $[A(q_l)][q.A(p_l)]$ **after** l,
replace l **with** $[R(q_l)]$
(**delete** l)
(**replace value of** l **with** $[V(q_l)]$)
(**insert** $[F(q_l)][q.F(p_l)]$ **as first into** l)
(**insert** $[q.L(p_l)][L(q_l)]$ **as last into** l)
(**rename** l **to** $N(q_l)|N(p_l)$)

Case 4-D-VI otherwise if $D(q_l) \neq \perp$, i.e., q deletes l, only nodes inserted after and before by p are kept, q is applied on them, and possibly q also inserts nodes before and after l:

insert $[q.B(p_l)][B(q_l)]$ **before** l,
insert $[A(q_l)][q.A(p_l)]$ **after** l,
delete l
(**replace value of** l **with** $[V(q_l)]$)
(**insert** $[F(q_l)][q.F(p_l)]$ **as first into** l)
(**insert** $[q.L(p_l)][L(q_l)]$ **as last into** l)
(**rename** l **to** $N(q_l)|N(p_l)$)

Case 5-V-VI otherwise if $V(q_l) \neq \perp$, i.e., q replaces the value of l, only nodes inserted after and before by p are kept, q is applied on them, and possibly q also inserts nodes before and after l, in addition q or p may have renamed l:

insert $[q.B(p_l)][B(q_l)]$ **before** l,
insert $[A(q_l)][q.A(p_l)]$ **after** l,
replace value of l **with** $[V(q_l)]$,
rename l **to** $N(q_l)|N(p_l)$
(**insert** $[F(q_l)][q.F(p_l)]$ **as first into** l)
(**insert** $[q.L(p_l)][L(q_l)]$ **as last into** l)
(**rename** l **to** $N(q_l)|N(p_l)$)

Case 6-I-V otherwise if $V(p_l) \neq \perp$, i.e,, p replaces the value of l, q is applied to all nodes coming from p, nodes inserted after and before l by q are forwarded, nodes inserted as first or last by q are put in the value-replacing primitive, and p or q may have renamed l:

insert $[q.B(p_l)][B(q_l)]$ **before** l,
insert $[A(q_l)][q.A(p_l)]$ **after** l,
replace value of l **with** $[F(q_l)][q.V(p_l)][L(q_l)]$,
rename l **to** $N(q_l)|N(p_l)$
(**insert** $[F(q_l)][q.F(p_l)]$ **as first into** l)
(**insert** $[q.L(p_l)][L(q_l)]$ **as last into** l)
(**rename** l **to** $N(q_l)|N(p_l)$)

Case 7-I-I otherwise, in all other cases, q is applied to all nodes coming from p, each inserting update primitive is completed with the contents inserted by q, and l is possibly renamed:

insert $[q.B(p_l)][B(q_l)]$ **before** l,
insert $[A(q_l)][q.A(p_l)]$ **after** l,
insert $[F(q_l)][q.F(p_l)]$ **as first into** l,
insert $[q.L(p_l)][L(q_l)]$ **as last into** l,
rename l **to** $N(q_l)|N(p_l)$

Proof that the previous definition fulfills the theorem predicate

Let p and q be two PULs. We prove, very mechanically, that $(q * p).h = q.(p.h)$ for any forest h by induction on forests, using the definition of the PUL application operator and of the composition operator.

- $q.(p.[O]) = [O] = (q * p).[O]$
- $q.(p.([f][g])) = q.([p.f][p.g]) = ([q.(p.f)][q.(p.g)]$
  By induction,$q.(p.f) = (q * p).f$ and $q.(p.g) = (q * p).g$.
  It follows $q.(p.([f][g])) = ([(q * p).f][(q * p).g]) = (q * p).([f][g])$.
- For $p.\dfrac{l, n}{[f]}$, we consider all cases in the definition of the $*$ operator.

Case 1-*-R if $R(p_l) \neq \perp$, i.e., p replaces l (case R):

$$p.\frac{l, n}{[f]} = [B(p_l)][R(p_l)][A(p_l)]$$

from which it follows by definition of the . operator on a concatenation of forests:

$$q.(p.\frac{l,n}{[f]}) = [q.B(p_l)][q.R(p_l)][q.A(p_l)]$$

Now,

$$(q*p)_l = \begin{array}{c}\textbf{insert } [q.B(p_l)] \textbf{ before } l,\\ \textbf{insert } [q.A(p_l)] \textbf{ after } l,\\ \textbf{replace } l \textbf{ with } [q.R(p_l)]\end{array}$$

so that by definition of the . operator on a tree (case R):

$$(q*p).\frac{l,n}{[f]} = [q.B(p_l)][q.R(p_l)][q.A(p_l)]$$

and

$$(q*p).\frac{l,n}{[f]} = q.(p.\frac{l,n}{[f]})$$

Case 2-*-D if $D(p_l) \neq \perp$, i.e., p deletes l (case D):

$$p.\frac{l,n}{[f]} = [B(p_l)][A(p_l)]$$

from which it follows by definition of the . operator on a concatenation of forests:

$$q.(p.\frac{l,n}{[f]}) = [q.B(p_l)][q.A(p_l)]$$

Now,

$$(q*p)_l = \begin{array}{c}\textbf{insert } [q.B(p_l)] \textbf{ before } l,\\ \textbf{insert } [q.A(p_l)] \textbf{ after } l,\\ \textbf{delete } l\end{array}$$

so that by definition of the . operator on a tree (case D):

$$(q*p).\frac{l,n}{[f]} = [q.B(p_l)][q.A(p_l)]$$

and

$$(q * p).\frac{l,n}{[f]} = q.(p.\frac{l,n}{[f]})$$

Case 3-R-VI if $R(q_l) \neq \perp$, i.e., q replaces l (case R).

p is either in case V, in which case:

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l,N(p_l)|n}{[V(p_l)]}[A(p_l)]$$

or in case I, in which case:

$$p\frac{l}{[f]} = [B(p_l)]\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)]}[A(p_l)]$$

Applying q on any of these expressions (case R) gives:

$$\begin{aligned} q.(p\frac{l,n}{[f]}) &= q.([B(p_l)]\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)] \text{ or } [V(p_l)]}[A(p_l)]) \\ &= [q.B(p_l)]q.\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)] \text{ or } [V(p_l)]}[q.A(p_l)]) \text{(forets concatenation)} \\ &= [q.B(p_l)][B(q_l)][R(q_l)][A(q_l)][q.A(p_l)]) \text{(case R)} \end{aligned}$$

Now,

$$(q * p)_l = \begin{array}{c} \textbf{insert } [q.B(p_l)][B(q_l)] \textbf{ before } l, \\ \textbf{insert } [A(q_l)][q.A(p_l)] \textbf{ after } l, \\ \textbf{replace } l \textbf{ with } [R(q_l)] \end{array}$$

so that by definition of the . operator (case R):

$$(q * p).\frac{l,n}{[f]} = [q.B(p_l)][B(q_l)][R(q_l)][A(q_l)][q.A(p_l)])$$

and

$$(q * p).\frac{l,n}{[f]} = q.(p.\frac{l,n}{[f]})$$

Case 4-D-VI if $D(q_l) \neq \perp$, i.e., q deletes l (case D):

p is either in case V, in which case:

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l, N(p_l)|n}{[V(p_l)]}[A(p_l)]$$

or in case I, in which case:

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l, N(p_l)|n}{[F(p_l)][p.f][L(p_l)]}[A(p_l)]$$

Applying q on any of these expressions (case D) gives:

$$\begin{aligned} q.(p\frac{l,n}{[f]}) &= q.([B(p_l)]\frac{l, N(p_l)|n}{[F(p_l)][p.f][L(p_l)] \text{ or } [V(p_l)]}[A(p_l)]) \\ &= [q.B(p_l)]q.\frac{l, N(p_l)|n}{[F(p_l)][p.f][L(p_l)] \text{ or } [V(p_l)]}[q.A(p_l)]) \text{(forest concatenation)} \\ &= [q.B(p_l)][B(q_l)][A(q_l)][q.A(p_l)]) \text{(case D)} \end{aligned}$$

Now,

$$(q*p)_l = \begin{array}{c} \textbf{insert } [q.B(p_l)][B(q_l)] \textbf{ before } l, \\ \textbf{insert } [A(q_l)][q.A(p_l)] \textbf{ after } l, \\ \textbf{delete } l \end{array}$$

so that by definition of the . operator (case D):

$$(q*p).\frac{l,n}{[f]} = [q.B(p_l)][B(q_l)][A(q_l)][q.A(p_l)])$$

and

$$(q*p).\frac{l,n}{[f]} = q.(p.\frac{l,n}{[f]})$$

Case 5-V-VI if $V(q_l) \neq \perp$, i.e., q replaces the value of l (case V):

p is either in case V, in which case:

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l, N(p_l)|n}{[V(p_l)]}[A(p_l)]$$

or in case I, in which case:

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)]}[A(p_l)]$$

Applying q on any of these expressions (case V) gives:

$$\begin{aligned} q.(p\frac{l,n}{[f]}) &= q.([B(p_l)]\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)] \text{ or } [V(p_l)]}[A(p_l)]) \\ &= [q.B(p_l)]q.\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)] \text{ or } [V(p_l)]}[q.A(p_l)]) \text{(forest concatenation)} \\ &= [q.B(p_l)][B(q_l)]\frac{l,N(q_l)|N(p_l)|n}{[V(q_l)]}[A(q_l)][q.A(p_l)]) \text{(case V)} \end{aligned}$$

Now,

insert $[q.B(p_l)][B(q_l)]$ **before** l,
insert $[A(q_l)][q.A(p_l)]$ **after** l,
replace value of l **with** $[V(q_l)]$,
rename l **to** $N(q_l)|N(p_l)$

so that by definition of the . operator (case V):

$$(q*p).\frac{l,n}{[f]} = [q.B(p_l)][B(q_l)]\frac{l,N(q_l)|N(p_l)|n}{[V(q_l)]}[A(q_l)][q.A(p_l)])$$

and

$$(q*p).\frac{l,n}{[f]} = q.(p.\frac{l,n}{[f]})$$

Case 6-I-V if $V(p_l) \neq \perp$, i.e„ p replaces the value of l (case V):

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l,n}{[V(p_l)]}[A(p_l)]$$

from which it follows:

$$\begin{aligned} q.(p\frac{l,n}{[f]}) &= q.([B(p_l)]\frac{l,N(p_l)|n}{[V(p_l)]}[A(p_l)]) \\ &= [q.B(p_l)]q.\frac{l,N(p_l)|n}{[V(p_l)]}[q.A(p_l)]) \text{(concatenation of forests)} \\ &= [q.B(p_l)][B(q_l)]\frac{l,N(q_l)|N(p_l)|n}{[F(q_l)][q.V(p_l)][L(q_l)]}[A(q_l)][q.A(p_l)]) \text{(case I)} \end{aligned}$$

Now,

$$(q*p)_l = \begin{array}{c} \textbf{insert } [q.B(p_l)][B(q_l)] \textbf{ before } l, \\ \textbf{insert } [A(q_l)][q.A(p_l)] \textbf{ after } l, \\ \textbf{replace value of } l \textbf{ with } [F(q_l)][q.V(p_l)][L(q_l)], \\ \textbf{rename } l \textbf{ to } N(q_l)|N(p_l) \end{array}$$

so that by definition of the . operator (case I):

$$(q*p).\frac{l,n}{[f]} = [q.B(p_l)][B(q_l)]\frac{l,N(q_l)|N(p_l)|n}{[F(q_l)][q.V(p_l)][L(q_l)]}[A(q_l)][q.A(p_l)])$$

and:

$$(q*p).\frac{l,n}{[f]} = q.(p.\frac{l,n}{[f]})$$

Case 7-I-I otherwise, in all other cases (I):

$$p\frac{l,n}{[f]} = [B(p_l)]\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)]}[A(p_l)]$$

from which it follows:

$$\begin{aligned} q.(p\frac{l,n}{[f]}) &= q.([B(p_l)]\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)]}[A(p_l)]) \\ &= [q.B(p_l)]q.\frac{l,N(p_l)|n}{[F(p_l)][p.f][L(p_l)]}[q.A(p_l)]) \text{(concatenation of forests)} \\ &= [q.B(p_l)][B(q_l)]\frac{l,N(q_l)|N(p_l)|n}{[F(q_l)]q.([F(p_l)][p.f][L(p_l)])[L(q_l)]}[A(q_l)][q.A(p_l)]) \text{(case I)} \\ &= [q.B(p_l)][B(q_l)]\frac{l,N(q_l)|N(p_l)|n}{[F(q_l)][q.F(p_l)][q.p.f][q.L(p_l)][L(q_l)]}[A(q_l)][q.A(p_l)]) \end{aligned}$$

Now,

$$(q * p)_l = \begin{array}{c} \textbf{insert } [q.B(p_l)][B(q_l)] \textbf{ before } l, \\ \textbf{insert } [A(q_l)][q.A(p_l)] \textbf{ after } l, \\ \textbf{insert } [F(q_l)][q.F(p_l)] \textbf{ as first into } l, \\ \textbf{insert } [q.L(p_l)][L(q_l)] \textbf{ as last into } l, \\ \textbf{rename } l \textbf{ to } N(q_l)|N(p_l) \end{array}$$

so that by definition of the . operator (case I):

$$(q * p).\frac{l}{[f]} = [q.B(p_l)][B(q_l)]\frac{l, N(q_l)|N(p_l)|n}{[F(q_l)][q.F(p_l)][(q * p).f][q.L(p_l)][L(q_l)]}[A(q_l)][q.A(p_l)])$$

and since by induction $(q * p).f = q.(p.f)$:

$$(q * p).\frac{l, n}{[f]} = q.(p.\frac{l, n}{[f]})$$

In all cases, we have the equality.

3.8 Conclusion

With this new composition operator on PULs, it is possible to summarize the semantics of several successive PULs in a single one, without introducing any order between the update primitives. This will prove very useful in the next chapter, which introduces a version control system for data represented as a tree. It means that it is legitimate to assume that the delta between two versions of a tree can be expressed by a single PUL, even if the changes were made by a full-fledged XQuery Scripting program applying several PULs.

Chapter 4

A Version Control System for semi-structured data

This chapter provides an extension to XQuery and its data model in order to support time travel. The challenge is that, in addition to accessing past versions of the data, it must be possible to query this data, and the querying syntax must integrate seamlessly with the existing XQuery language.

4.1 Introduction

Over the last few decades, hardware trends have been going towards cheaper, denser storage. In addition, software allows compression, virtually increasing the amount of data stored. Several times, Jim Gray [68] [79] pointed out that with so much storage capacity available, it becomes possible to keep track of crucial information: instead of updating in place, a new version can be created.

Versioning information allows to go back in time, compare, analyze, and produce even more information. In addition, versioning can serve as a transaction control mechanism, allowing applications to undo or merge concurrent changes. Such features could be very useful, for example, for collaboration tools, which are proliferating. Google, Adobe and Microsoft are offering Web applications for word processing, spreadsheets, presentations, which allow users to view, share and edit documents in real time [1] [15] [25]. Applications like Microsoft Word already have basic versioning (reviewing) capabilities. In 2011, Apple introduced Versions [6] in its desktop operating system, which automatically performs version control on text processing, spreadsheet and presentation documents.

Some versioning systems are already widespread. On the one hand, there exist document versioning systems such as CVS, SVN [4], which are centralized, and git [13], Bazaar [7], Mercurial [23], which are distributed.

This systems analyze documents line by line, compare and compress them.

On the other hand, there are also versioning systems for structured data, like versioned databases (Oracle Flashback [28], Microsoft ImmortalDB [70]).

There are two basic approaches to keeping track of versioned data. The first one is to use valid time [43][67][73][78], i.e., the time at which the data is valid (which can even be far away in the past, before computers existed at all). The second one, which we are taking in this paper, is transaction time, i.e., the time at which the data is written or committed.

The general contributions of our work are:

(i) to provide a unified model for data versioning, document versioning and anything (semi-structured data) between data and documents.

(ii) to have a programming language suited for processing (powerful querying as well as updating) this versioned data.

(iii) to perform measurements on a prototype that demonstrate that time travel can be implemented with reasonable storage and query time overhead.

We choose to base our work on XML technologies and extend XQuery, which is a Turing-complete programming language standardized by the W3C.

What distinguishes our approach from existing literature is the combination of two factors. First, time is often modeled as an explicit parameter in an application or document. Extra code is needed to explicitly refer to time in the queries, which can lead to poor performance. Rather, we choose to add time to the logical data model and hide physical implementation and storage details from the programmer, so that it is simple to go back in time and query any, or even several versions. Also, we ensure backward compatibility, allowing the user to completely ignore the versioning features. Second, our data and query model seamlessly integrates with the updating model: instead of collecting snapshots over time, the database is constructed and updated using the XQuery Update Facility, which has the nice side effect of providing us with the deltas as Pending Update Lists and of unambiguously keeping track of node identities.

The remainder of this chapter is organized as follows: in Section 4.2, we present a motivating example. Section 4.3 introduces our extension of the XQuery data model to obtain an XML-aware versioning system. Section 4.4 introduces our extension of the

XQuery programming language with new functions and time axes, so that it is possible to query against versioned data. In Section 4.5, we give an overview of the processing model for checking out from and checking in to the repository, to allow collaborative work. Section 4.6 presents data structures to implement the versioning system together with a high-level overview of querying and updating algorithms, as well as a storage scheme based on existing literature. Section 4.7 presents the performance measurements showing that extending a non-versioning engine with versioning capabilities does not alter performance on traditional queries. Section 4.8 discusses related work.

4.2 Motivating example

The running example for this paper is the following: an investment bank maintains data about stocks, bonds, funds, and portfolios thereof in a database. Users can collaboratively modify the data. This is semi-structured information (for example, it has optional comments and nested elements). The bank would like to set up a versioning system for its data.

Initially, the data could look as shown in Fig. 4.1(a). There are three sections: users, securities and portfolios. The users are Alice, Bob, and Charles. The securities part contains Apple stock (NASDAQ Symbol AAPL), a bond of the Swiss Confederation (Valor 3141) and a fund managed by Charles (ISIN CH123456789) investing in the former two assets. Bob has a portfolio containing bonds and shares in Charles' fund. Fig. 1(b) shows a subsequent database state after some modifications: Alice changed her hometown, the fund's content was updated, Bob updated his data to use references (Symbol, Valor, ISIN) to the underlying securities, and Alice created her portfolio with AT&T stocks.

While keeping all the functionality already available to query and update the current version, the bank would like to be able to keep track of and analyze past data as well. For example, it could be interested in the evolution of the diversification of a portfolio, or in checking that a balanced fund has always allocated the right stock/bond ratio over time, or in comparing the performance of a portfolio between two versions (using an external source to retrieve the latest pricings). Versioning functionality is needed. We emphasize that not only versioning data is important - reasons are widespread in literature [78], [81] - but also the ability to flexibly query against past versions.

Versioning it as a document, e.g., in SVN (diffing lines of text) would miss the continuity of the tree structure: the identity of the nodes is lost between the versions, drastically

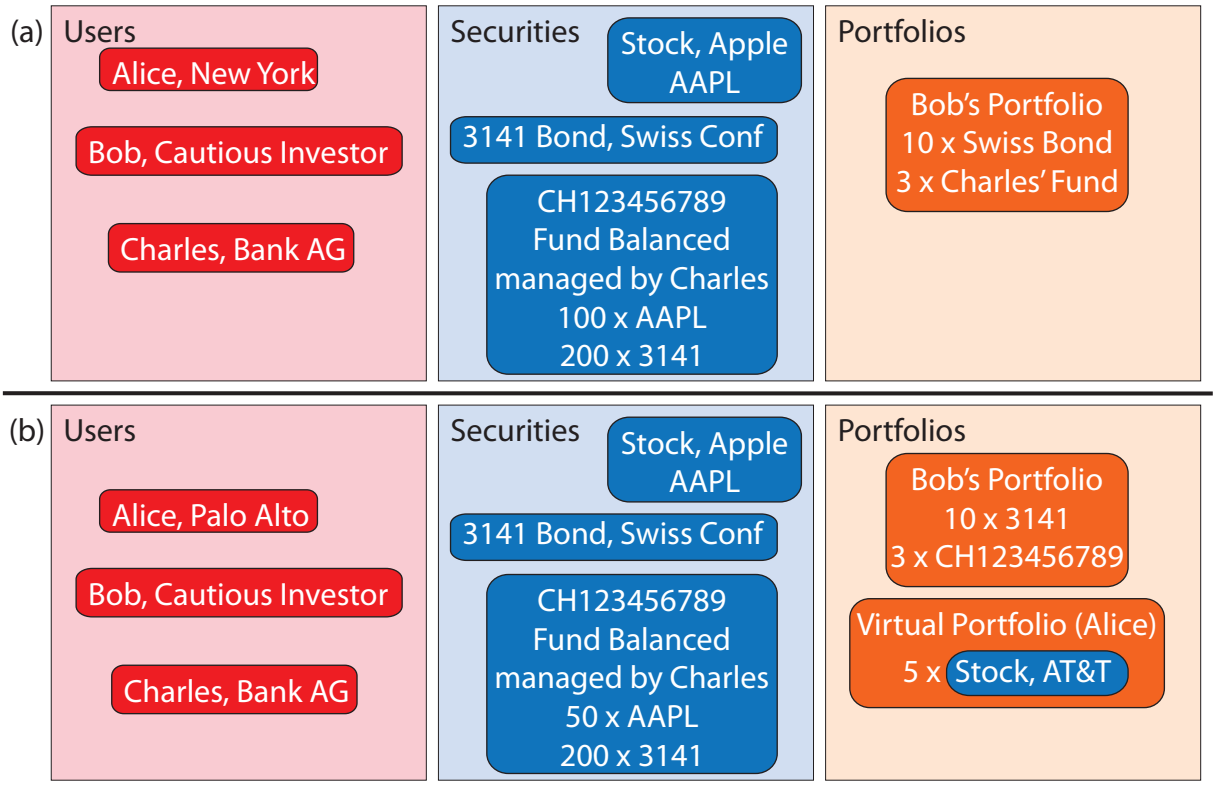

Figure 4.1: *Two versions of the data (a) and (b).*

limiting the querying functionality. A versioned database would not be of great help either, as this is semi-structured information. For example, the users do not provide the same information (hometown, company, or even comments). The keys used to identify securities are not using the same standard (NASDAQ Symbol, Swiss Valor, ISIN). Then, there are levels of nesting (like the AT&T stocks nested in Alice's portfolio). Finally, since there is no schema, the evolution of the data structure is unpredictable. For all these reasons, versioning semi-structured data cannot be reduced to a versioned database with uniform rows and columns in a straightforward way.

There are already many technologies around for storing and manipulating semi-structured information: this data can be stored as XML (e.g., a large XML document, or as three collections of XML trees) or JSON. XML can be queried with XPath, XQuery, XSLT.

There is also literature about storing versioned, semi-structured information [56] [57] or query it [73] [60]. We suggest to leverage these technologies and techniques and provide a unified framework which is (1) XML-aware to handle any kind of data. The versioning system should be aware of XML trees and nodes, which can be done by extending the XDM. (2) XQuery-aware for powerful queries and updates. It should be

possible to easily navigate to past versions with XQuery code, which can be done by extending the XQuery syntax. It should be possible as well to create new versions with XQuery Update (which outputs a PUL) or XQuery Scripting (which produces a sequence of PULs). We showed in Chapter 3 that a sequence of PULs can be composed to a single PUL, so that a delta can always be represented by a PUL. Serializing these PULs as XML can even allow querying the deltas.

A typical setting to start with for the implementation would be to use an XQuery Web Application Server like MarkLogic [22] or Sausalito [49]. The latter runs on and stores its XML data in the Amazon cloud. We chose to extend Sausalito with versioning capabilities.

4.3 Extensions to the data model

Our broader aim is bridging the gap between data and document versioning. Because we want to query versioned semi-structured data, we need to work at the level of the data model. For XML- and XQuery-aware versioning, we need to extend the XQuery Data Model (XDM) with a time dimension.

4.3.1 Tree timelines

We introduce the concepts of *node timeline*, *tree timeline* and *version*. In the context of a Web application, we are interested in linear versioning. Branching can be emulated by setting up a tree of repositories, and merging can be done with the processing model defined in Section 4.5.

Node Timeline A *node timeline* is a succession of all node items sharing a given identity. It models the lifetime of a node item on which updates are applied. A node timeline is identified by a URI.

Tree Timeline A *tree timeline* is a succession of trees whose roots share a given identity. It models the lifetime of an XML tree on which updates are applied. All roots of these trees belong to the same node timeline, which is called the root node timeline of the tree timeline. A tree timeline is identified by the URI of its root node timeline.

Version A *version* uniquely identifies a tree among a tree timeline. Each version has a number. There is also a special version with no number, called the local version.

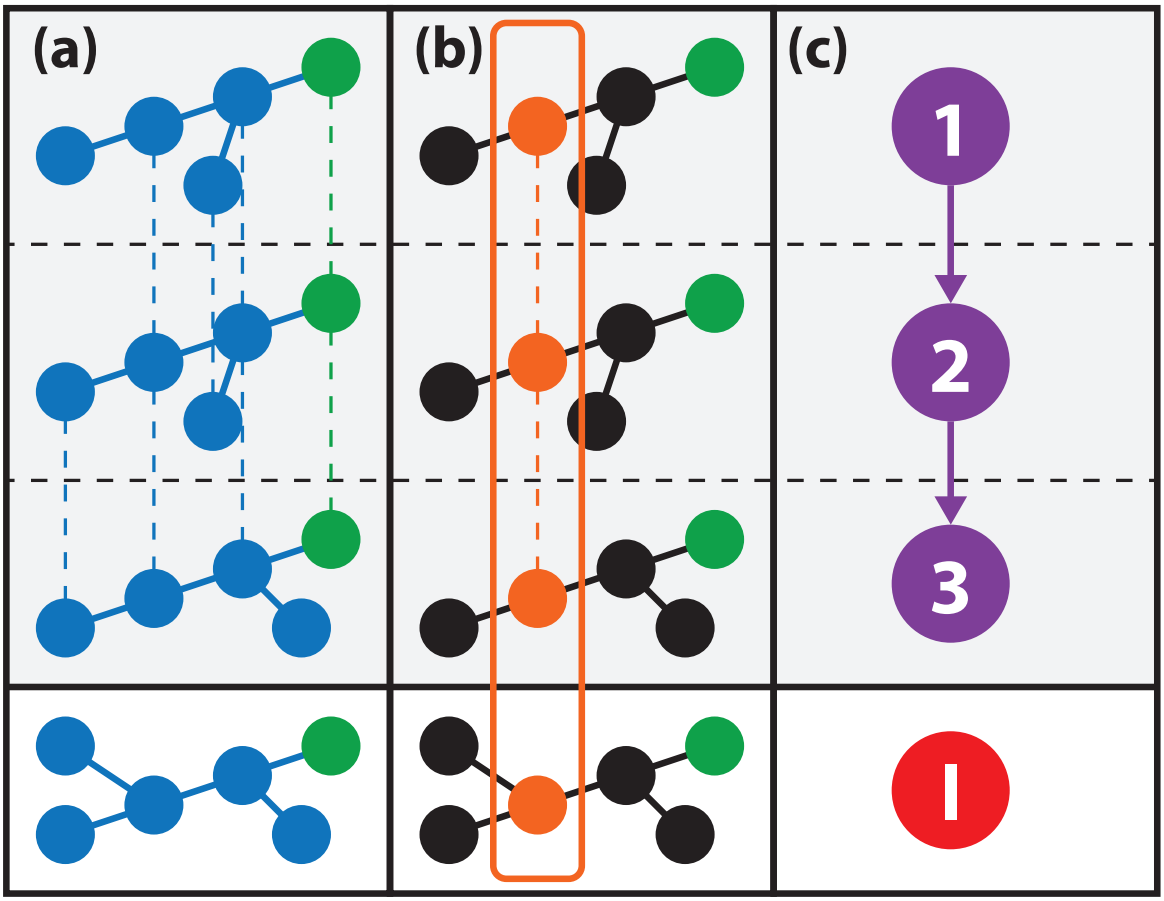

Figure 4.2: *A tree timeline (a), a node timeline (b), a sequence of versions (c).*

This is the version which is currently being modified by the user and which has not been committed yet. Each user can have their own local version. A version is identified by a URI.

Fig. 4.2 shows how tree timelines, node timelines and versions are related. The version shown at the bottom in red is the local version.

Node items are defined as in the original XQuery Data Model. Two new accessors are defined on them:

- `dm:node-timeline` returns the URI of the node timeline this node item belongs to
- `dm:version` returns the URI of the version that uniquely identifies this node item within its node timeline.

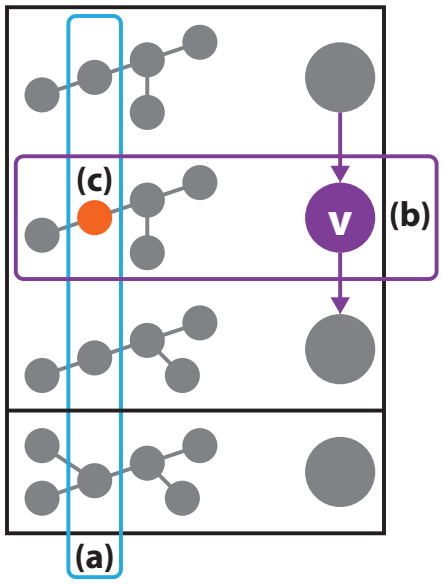

Figure 4.3: *A node timeline URI (a) together with a version URI (b) uniquely identify a node item (c).*

The node timeline URI and the version URI uniquely identify a node item (Fig. 4.3). The other accessors (`dm:children`, `dm:attributes`, `dm:node-name`, ...) are defined as stated in the XDM specification. Document order is extended to allow time-awareness: node items are first sorted in time, and then in space. Newly created nodes (with element constructors or nodes copied in updating expressions) belong to no timeline and are in no version, so that these accessors are reserved.

In the remainder of this chapter, when we talk about "a version of a node (or tree) timeline", we actually take a shortcut and refer to the node item (tree) uniquely identified by this version and the node (tree) timeline. This should be clear from the context.

4.3.2 Collection timelines

In some XQuery implementations like Sausalito, collections are first-class citizens (for example, they can be updated with extra update primitives). Thus we also extend the corresponding data model.

Collection Timeline A *collection timeline* is a succession of collections of node items. It models the lifetime of a collection to which nodes are inserted or deleted. Collections are defined in the XQuery specification. Versions can also uniquely identify a collection in a collection timeline (Fig. 4.4).

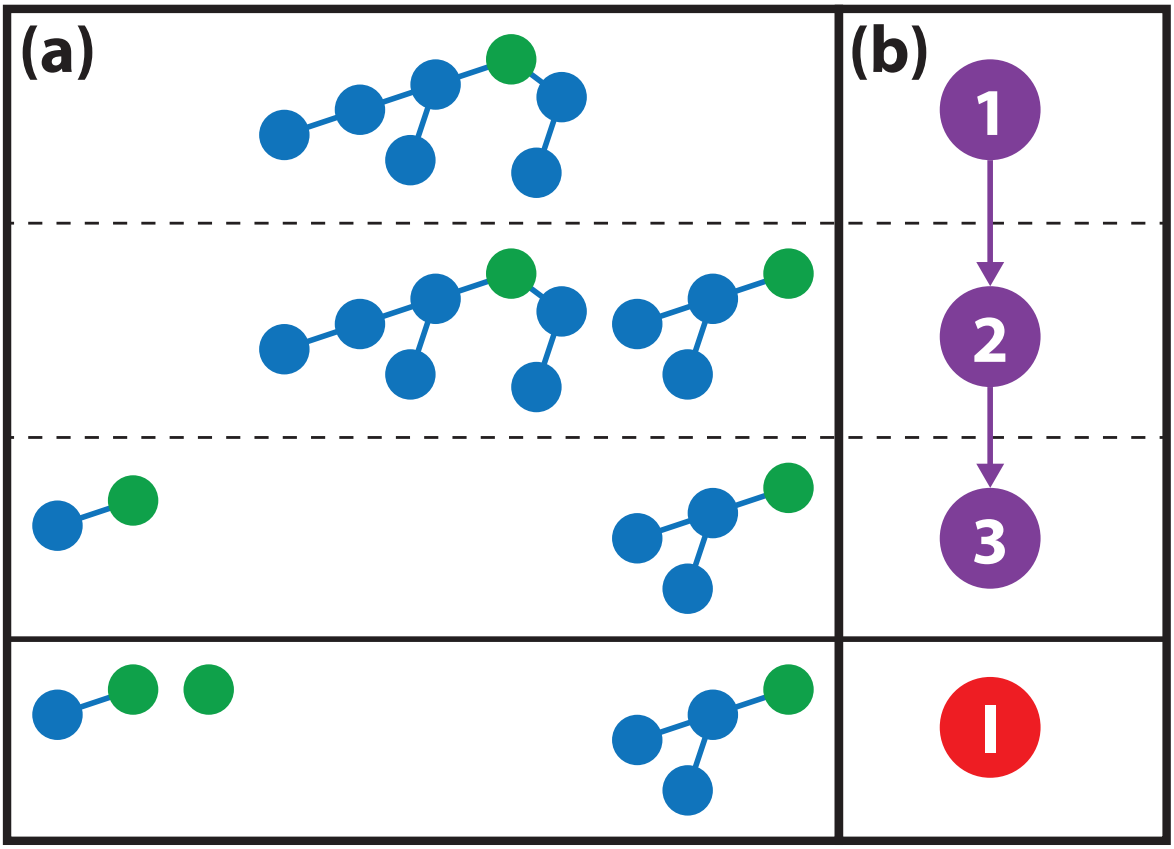

Figure 4.4: *A collection timeline (a), and the corresponding versions (b).*

We restrict collection timelines to two different kinds: (i) collection timelines which only contain non-local (frozen) versions of the node timelines, and (ii) collections timelines which only contain local versions of the node timelines. Collections of the second kind "follow" the evolution of the node timeline.

The implementation of collections differs between XQuery engines. In Sausalito, collections and trees are tied together: collections own their trees on the storage layer, i.e., a collection points to the roots of its trees, and a tree can only belong to a single collection [45]. To allow for this entanglement, we can synchronize tree and collection (type (i)) version numbers, which means that the versioning granularity is at the collection level ("synchronized collection timeline").

The framework for synchronized collection timelines can very often be deduced from the framework for tree timelines as such a collection can be represented as a forest of trees (equivalently, a tree without its root). The framework for other collection timelines (type (ii), or unsynchronized (i)) is very similar to document-oriented versioning as each

slice is a sequence of identifiers.

4.3.3 Serialized Pending Update Lists

The XQuery Update standard [54] defines how to update an XML document. The updates themselves are organized as a set of update primitives (of the kinds insert, delete, replace, rename) standardized as a Pending Update List (PUL). For example, it is possible to delete Bob's portfolio like so:

```
delete node //Portfolio[@owner="Bob"]
```

This XQuery Update program outputs a PUL containing, in this case, one update primitive `upd:delete($ref)` with a variable `$ref` bound to the node to delete (the one computed by the XPath expression above). Such a PUL can then be applied to the XML document, which deletes the node.

An updating XQuery program always returns a PUL. XQuery Scripting is an extension of XQuery Update which allows to sequentially apply several PULs - however, as explained in chapter 3, this can always be reduced to a single PUL. PULs are hence ideal candidates for deltas in an XQuery-based versioning system, as shown on Fig. 4.5.

In order for these deltas to be queryable, we choose to serialize the PULs to the XML format. For example, the PUL output by the program above can be serialized as:

```
<pending-update-list xmlns="http://www.example.com/pul">
  <delete>
    <target>http://www.example.com/Doc1#1.3.3</target>
  </delete>
</pending-update-list>
```

Technically, a *serialized Pending Update List* is a Pending Update List (as defined in [54]) in which the target is replaced with the URI reference to its node timeline and the content is replaced with its XDM serialization. Serialized Pending Update Lists, since they are XML, are queryable with XQuery.

An XML Schema for serialized PULs is available in a technical report [63]. It also covers the serialization of an XDM instance and provides an example of serialized PUL.

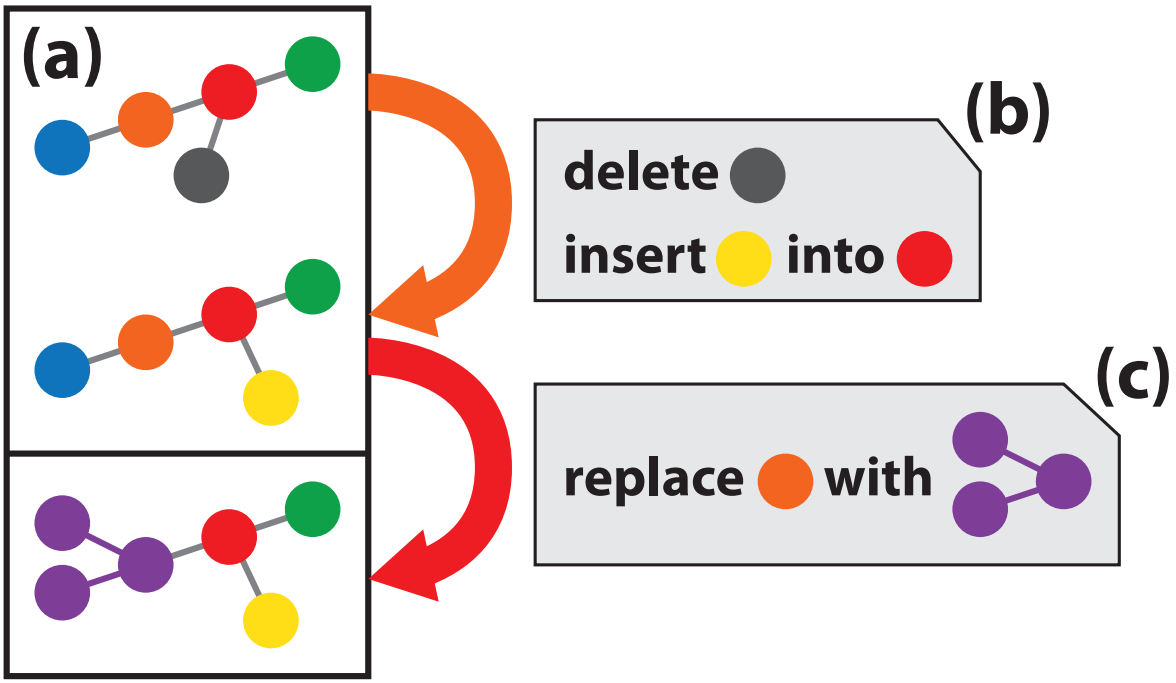

Figure 4.5: *Three versions (one being local) of a tree timeline (a), and serialized PULs modeling the deltas (b).*

4.4 Extensions to the programming model

4.4.1 Tree timelines

To navigate through a tree, i.e., within a version of a tree timeline, the user can use the various axes defined in XQuery (`child::`, `descendant-or-self::`, ...). To navigate in time, the following functions are added to the XQuery functions and operators [72] and are available to the user (they are defined in a new versioning namespace, represented with the *vng* prefix):

- **`vng:reference`** which takes a node and returns its node timeline URI (using **`dm:node-timeline`**).
- **`vng:version`** which takes a node and returns its version URI (using **`dm:version`**).
- **`vng:dereference`** which takes a node timeline URI and returns its local version.
- **`vng:ttdereference`** which takes a node timeline URI, a version URI and returns the associated node item.

- **vng:node-versions** which takes a node and returns a list of all version URIs of its node timeline.
- **vng:version-number** which takes a node and returns its version number.
- **vng:time** which takes a node and returns the creation date and time of its tree (when it was committed).
- **vng:is-local** which takes a node and returns true if it is local, false otherwise.

For example:

```
vng:dereference(vng:reference($node))
```

gets the local version of node $node.

```
let $ref := vng:reference($node)
for $version in vng:node-versions($node)
return vng:ttdereference($ref, $version)
```

builds a sequence of all versions of node $node.

```
for $n in $nodes
where vng:is-local($n)
return $n
```

filters nodes which are local within a sequence $nodes.

Because using these functions remains tedious for time travel, we also introduce the following axes:

- **first::** (the first version)
- **earlier::** (the former version)
- **past::** (all past versions)
- **past-or-current::** (the same, plus the current one)
- **last::** (the last version)
- **later::** (the next version)
- **future::** (all future versions)

- **future-or-current::** (the same, plus the current one)
- **current::** (the current version)
- **all-times::** (all versions)
- **local::** (the local version)

For example, **later::**node() navigates to the next version of the node timeline to which the current node belongs or the empty sequence if it does not exist. For example,

```
last::node()/later::node()
```

will always return the empty sequence, as well as

```
first::node()/earlier::node().
```

Time axes are compatible with node tests:

```
[time axis]::[node test]
```

is defined as

```
[time axis]::node()/self::[node test]
```

For example

```
future::mickey
```

is defined as

```
future::node()/self::mickey
```

and looks for all newer versions of the current node whose names are “mickey”.

Also,

```
if(vng:reference($temperature-left) =
    vng:reference($temperature-right))
then
  fn:avg(
    $temperature-left/future-or-current::*
    intersect
    $temperature-right/past-or-current::*
  )
else ()
```

checks whether the temperature nodes `$temperature-left` and `$temperature-right` belong to the same node timeline and if such is the case, it computes the average temperature between them.

Back to our motivating example in Section 4.2, the following query checks that balanced funds correctly invested their funds (according to `is-compliant`):

```
every $fund in $doc//all-times::Fund
satisfies ($fund/Type != "balanced" or is-compliant($fund))
```

The following query looks for all versions of Bob's portfolio which, today, would be worth more than the last version (**value** gives access to the latest quotes):

```
let $current-portfolio ::= $doc//Portfolio[owner = "Bob"]
for $portfolio in $current-portfolio/past::*
where value($portfolio) > value($current-portfolio)
return vng:version($portfolio)
```

4.4.2 Collection timelines

To navigate through time in collection timelines, the following functions are available to the user:

- **fn:collection** which takes a collection timeline URI and returns a sequence of nodes (its local version)
- **vng:collection-versions** which takes a collection timeline URI and returns a list of all its version URIs
- **vng:ttcollection** which takes a collection timeline URI and a version URI and returns the corresponding collection (sequence of nodes).

4.4.3 Retrieving deltas

We define the function **vng:pul** which takes a tree or collection timeline URI and two version URIs, and returns the corresponding delta (serialized PUL). This query counts how many security-related nodes have been deleted so far:

```
let $ref := $doc/vng:reference(Securities)
let $first := $doc/vng:version(first::Securities)
let $last := $doc/vng:version(last::Securities)
let $pul := vng:pul($ref, $first, $last)
return count($pul//*:delete/*:target)
```

4.5 Extensions to the processing model

One of the requirements for our framework is flexibility, which should also be true for collaborative work. We need flexibility to embed different collaboration models, for example

database transactions where queries are executed in an isolated way. An error is thrown if there are any concurrent changes

document editing where it is always attempted to merge changes into the repository even if other users modified the document.

In order to allow for several users to edit a document or data with different collaboration models, we need to extend the processing model of XQuery.

4.5.1 Checkout and checkin

We rely on a classical client-server architecture. The server maintains a central repository with all past versions. Each client can set up a local version of a tree timeline by checking out the tree timeline from the server repository. It is only allowed to manipulate this local version (keeping track of non-committed yet modifications as a local PUL). The nodes in the local version are only visible to the client.

The local modifications can be committed by checking in. New versions are appended to the repository only upon a successful checkin. Checkins and checkouts are atomic and isolated operations: there can only be one checkout or checkin at a time. This architecture is illustrated on Fig. 4.6.

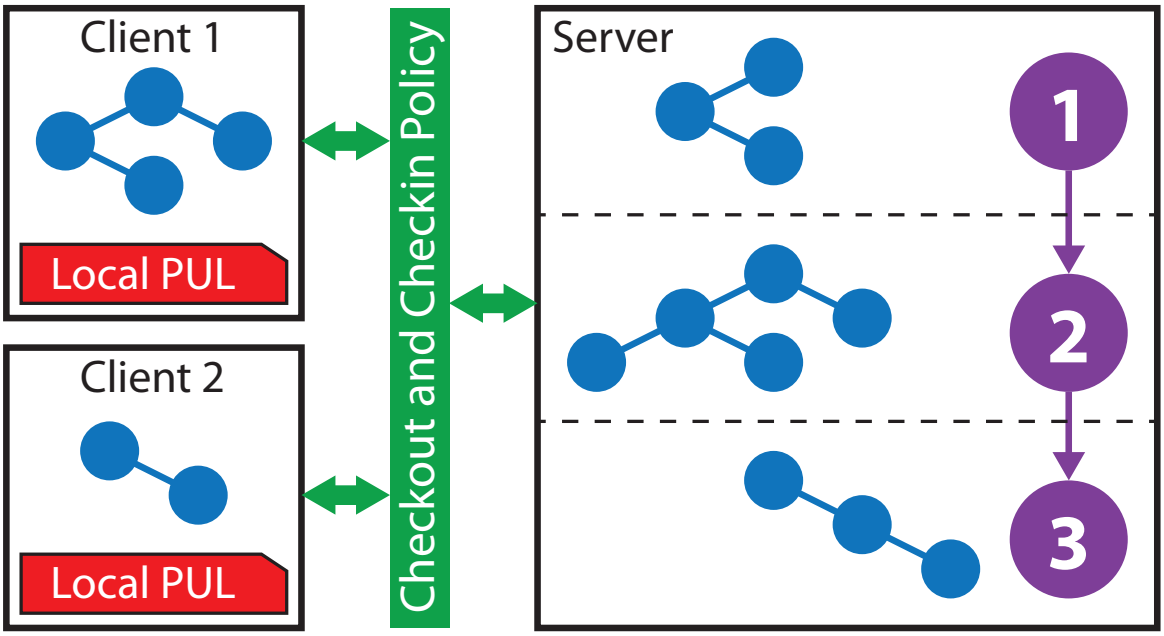

Figure 4.6: *The checkout-checkin processing model*

To make checkouts and checkins as flexible and configurable as possible, we use checkout and checkin policies. A client chooses a policy to decide how it wants to communicate with the server (read/write access to the repository, concurrent changes allowed or not, ...).

Both checkouts and checkins can be done implicitly, or explicitly with a function call. An implicit checkout occurs whenever it is attempted to access the local version of a tree timeline that has not been checked out yet. An implicit checkin occurs at the end of the program for all tree timelines which were checked out and modified during the program.

4.5.2 Checkout policy

A checkout policy can be seen as a black-box which checks out a version of a tree timeline so that the user can modify it. As one could have already checked out and made modifications before, the framework must allow for merging this version of the tree timeline with an existing local version.

Its input parameters are (i) the URI of the tree timeline to check out, (ii) the URI of the version to check out and (iii) the PUL containing local changes, read from the dynamic context (empty if it is the first checkout or if there are no local changes).

It outputs (i) the local version of the tree timeline which has just been checked out, (ii) the updated PUL containing local changes (written to the dynamic context) (iii) a boolean indicating whether it has been successful. If this boolean is false, then the local version and the local PUL are reverted and (iv) possibly an XDM instance containing unsolved conflicts if the checkout failed. How conflict resolution is done is outside of the scope of this work. There is work available on handling conflict resolution given several PULs [51].

In addition, it (v) saves the URI of the version which has been checked out in the dynamic context so that the checkin policy knows on what version the local PUL is based.

4.5.3 Checkin policy

A checkin policy can be seen as a black-box which attempts to check in the modifications done on the local version.

Its input parameters are (i) the URI of the tree timeline to check in, (ii) the PUL containing local changes, read from the dynamic context and (iii) the URI of the version which has been checked out, read from the dynamic context.

It outputs (i) the PUL which is to be applied to the last version in the repository, (ii) a boolean indicating whether it has been successful. If this boolean is true, then the local version and the local PUL are erased and the version URI is removed from the dynamic context. (iii) possibly an XDM instance containing unsolved conflicts if the checkin failed.

If the checkin succeeds, the output PUL is applied to the last version of the tree timeline to create a new version.

4.5.4 Using Checkout and Checkin policies

The reason why our framework allows checkout and checkin policies is to allow each user to choose her collaboration model. The user is provided with several checkout and checkin policies.

There are several possibilities for a checkout policy. Here are four examples of checkout policies:

no-checkout always throw an error (no checkouts possible)

single-checkout copy the version of the tree timeline to the local version when it is implicit, and subsequently return errors (only one checkout possible)

merge always copy the version of the tree timeline to the local version and then merge any former local changes.

discard always copy the version of the tree timeline to the local version and then discard local changes.

Also, there are several possibilities for a checkin policy, like for example:

no-checkin always throw an error (no write access to the repository)

conservative check whether the version which has been checked out is (still) the last version in the repository and output the local PUL if such is the case. Otherwise throw an error.

merge in any case attempt to merge local changes to changes between the version which has been checked out and the last version in the repository. Throw an error if conflicts arise so that the user solves them.

discard discard any changes done in the repository since the last check out and output a merge of a reverse PUL and of the local PUL.

Choosing which policy to use is done at the application level, rather than at the repository level, i.e., different users of the same repository may use different policies. Each policy has a URI and choosing a policy is done in the prolog, exactly like collations for string comparison.

For example, for a database transaction mode, one could use:

```
declare checkout policy "http://www.example.com/single-checkout";
declare checkin policy "http://www.example.com/conservative";
```

where only one checkout is made, and an error is thrown whenever other users concurrently wrote to the repository.

And for document editing:

```
declare checkout policy "http://www.example.com/merge";
declare checkin policy "http://www.example.com/merge";
```

where it is always attempted to merge any changes.

4.6 Algorithms and Data structures

The last three parts gave a logical view on the expected behavior of tree timelines for an XQuery program. The contribution of this part is threefold:

- introduce a new data structure, called π-tree, which efficiently implements tree timelines
- give algorithms (i) defining how versions of this tree timeline can be retrieved and (ii) defining how this data structure is modified when a new version is produced.
- reusing and completing an existing storage schema to allow for collection versioning.

4.6.1 Pi-Nodes, Pi-Trees and Pi-Forests

Keeping each version as whole tree in memory would be very expensive. Many document versioning systems (SCCS, RCS, CVS, SVN...) store reverse or forward deltas to reconstruct past versions. The thought process explaining why we are not doing this is as follows. Each XML data slice can be represented as a tree. With the XQuery Update model, over time, the parent of a node never changes. A node can be connected, or disconnected, only once: at no two points in time it has two distinct parents (XQuery Update does not support moving nodes around). For example, on Fig. 4.7(a), in any version, node β is always a child of node α, γ and δ of β, ϵ of α and ϕ of ϵ.

Hence, for each tree timeline, its node timelines are organized in a tree in a natural way as shown on Fig. 4.7(b). It is this materialization of the tree timeline as a tree of node timelines, called π-tree, which we use for our implementation. From now on, we focus on element nodes, but it is straightforward to extend to other kinds of nodes.

A node timeline is implemented as a π-node (Fig. 4.8(a)), which has an identity λ (here ORDPATH 1.3), a creation time c (here 2) and a deletion time d (here 5), a mapping n of the versions to the successive names of the node (“a” for 2 and 3 and “b” for 4). Creation and deletion times are positive integers, possibly infinite. A π-tree is then a tree of π-nodes. We call a (possibly empty) sequence of π-trees (Fig. 4.8(b)) a π-forest. Note that π-forests can be used to implement synchronized collection timelines (Section 3.2).

For convenience, the algorithms are described recursively on π-trees and π-forests, with the following definition: a π-tree (Fig. 4.8(c)) is obtained by attaching a π-forest

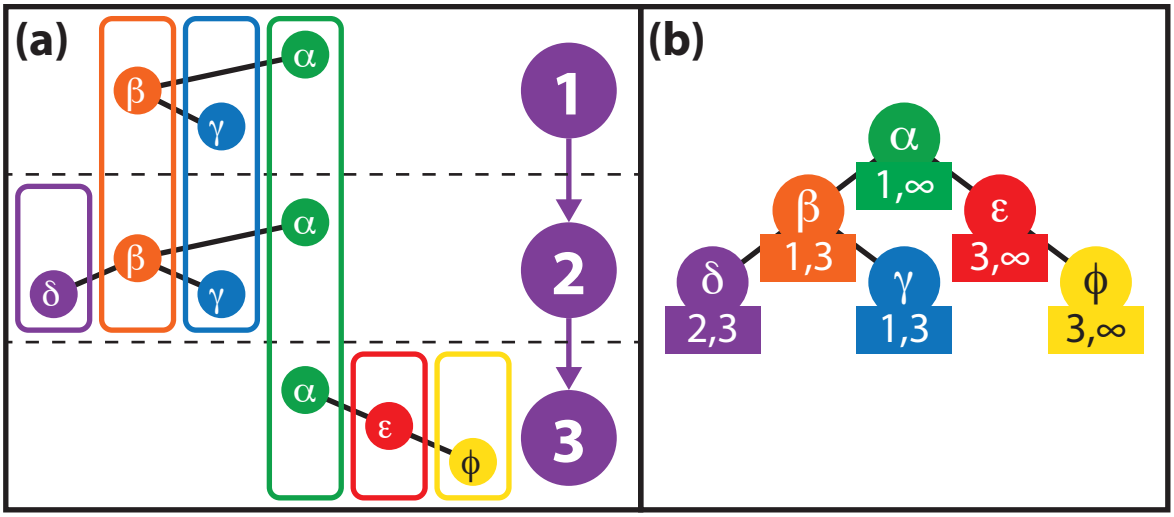

Figure 4.7: *Three versions of a tree timeline (a) and its implementation as a π-tree (b)*

below a π-node.

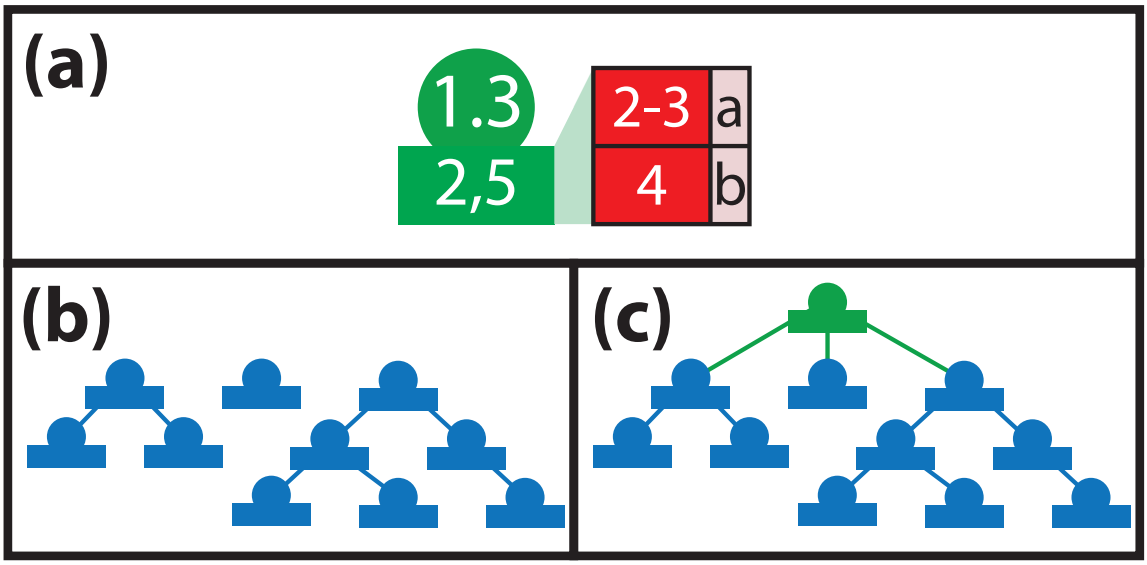

Figure 4.8: *A π-node (a), a π-forest (b), a π-tree (c).*

Retrieving a version (slice) v from the π-forest data structure is done with the instantiation function. This function is defined recursively as follows: the instantiation of a π-node (λ, c, d, n) is the node $(\lambda, n(v))$ if $c \leq v < d$, is an empty forest otherwise. The instantiation of a π-forest is the sequence of the instantiations of its π-trees, and the

instantiation of a π-tree is done by instantiating its root π-node and, if the result is not empty, recursively the underlying π-forest.

Using this algorithm, the three versions on Fig. 4.7(a) can be reconstructed from Fig. 4.7(b). The algorithm works in $O(n)$ where n is the size of the π-forest. This grows linearly with the number of versions, but it is possible to reduce this time to the average size of a slice (up to a constant factor) by using clustering, as shown in Section 6.4.

4.6.2 Updating a Pi-Forest

From a logical point of view, it is straight-forward to compute a new version for a tree timeline: the last version is copied, the serialized PUL is deserialized to a PUL with targets living in this copy, and the PUL is applied. The resulting tree is the new version. From a physical point of view, the π-forest needs to be updated. We defined an algorithm to apply updates (contained in a PUL) to our π-forest data structure at time t (assuming all creation and deletion times in the π-forest are lower than t). This algorithm is such that, at the logical level:

- it does not modify previous versions ($u < t$)
- it creates a new version t which differs from version $t - 1$ exactly according to the PUL.

An illustration is given in Fig. 4.9, where the new π-tree instantiates to an additional version obtained by applying the PUL to the former version.

The algorithm is defined as follows. Applying a PUL is done:

- for each deleting update primitive: by updating the deletion time of the target,
- for each renaming update primitive: by updating the time-to-name mapping,
- for each inserting update primitive: by inserting all contents at the corresponding locations (initializing the creation time to t and the deletion time to $+\infty$)
- for each replacing (content-replacing) update primitive: by inserting the content after (or as last child of) the target and updating the deletion time of the target (or of its children).

The complexity of the algorithm is $O(sd)$ where s is the size of the PUL and d the maximum depth of a tree, assuming a node can be reached in $O(d)$ with a reference.

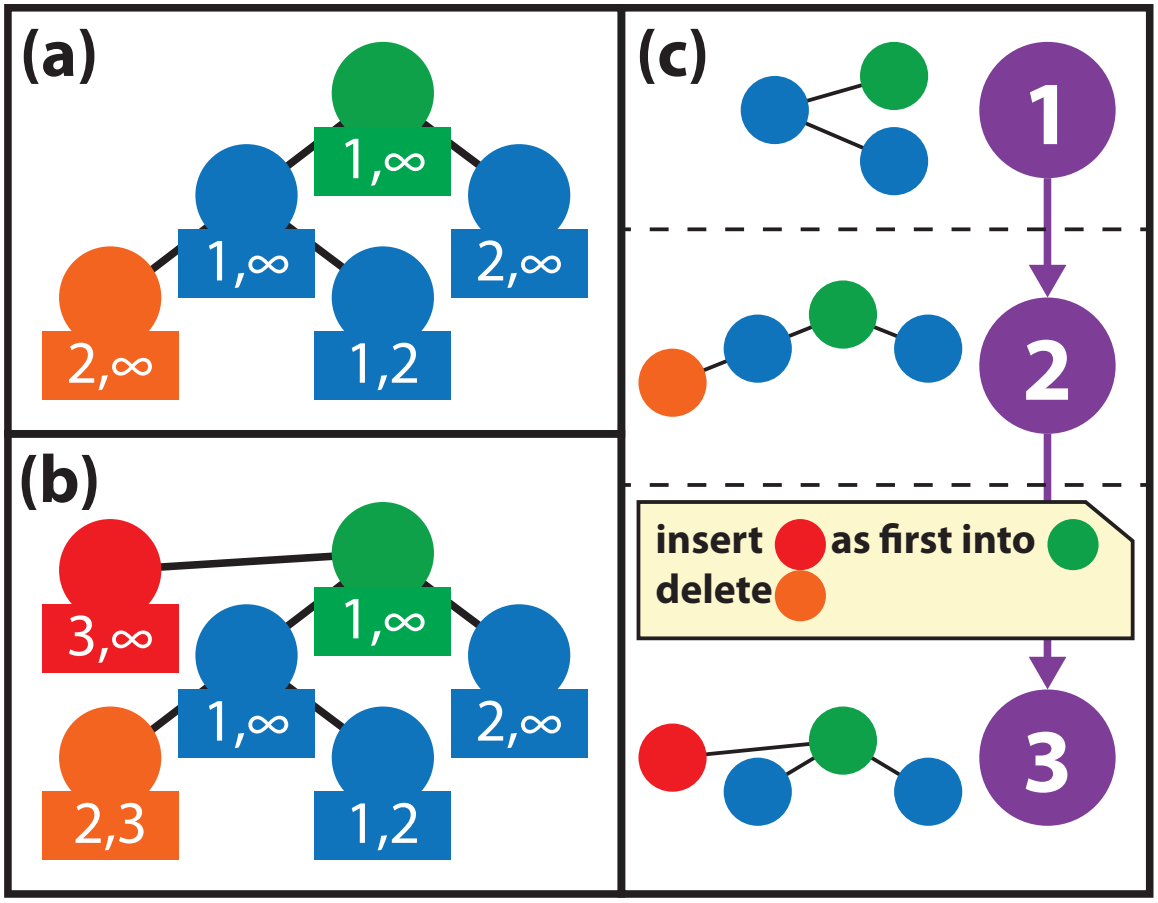

Figure 4.9: *The older π-tree (a) can be instantiated to the first two versions in (c). The updated π-tree (b) can be instantiated to the three versions.*

4.6.3 PUL retrieval

PUL retrieval may be implemented in two ways: (1) The PULs are stored besides each π-tree, and PUL composition is used to compute deltas. (2) No PUL is stored and deltas are computed automatically from the π-tree, as shown in existing literature, e.g. [55]. (1) and (2) might give different results, so the choice must be documented to the user.

4.6.4 Implementation of Timelines on Top or an RDBMS Layer

Sausalito storage layer

The Sausalito Web application server (used for deployed applications) stores persistent collections in a relational PostgreSQL database which lies on the same machine as the Web server.

A general two-column table maps collection QNames (identifiers) to internal integer collection IDs.

Each collection has its own two-column table (named after the collection ID), mapping:

- an exceptional ID to a CLOB serialization of the collection object
- each regular tree ID to a CLOB serialization of the corresponding tree object

Adaptation of the storage layer to version control

In the implemented prototype, we are reusing the same storage layout. Collection timeline objects are subclasses of collection objects, and tree timeline objects are subclasses of tree objects, so that a collection timeline table maps:

- an exceptional ID to a CLOB serialization of the collection timeline object
- each regular tree ID to a CLOB serialization of the corresponding tree timeline object

As becomes apparent in the measurements, the collection timeline CLOB (which contains a sorted list of all tree timeline ids as well as version information for each tree timeline) grows with time, so that we suggest an alternative layout for storing collection timelines.

Node timelines carry ORDPATH ids, but instead of using the ORDPATH 1 for the root, the root behaves like any other ORDPATH step, which extends ORDPATH-based document order computation to the entire collection. For example, the root of the first node timeline in the collection has ORDPATH 1, the root of the second node timeline has ORDPATH 3, and if a node is inserted between then, it will have the ORDPATH 2.1.

Then, each collection timeline has its own table, with the following columns:

Root ORDPATH The ORDPATH of the root of the tree timeline (NULL to store non-growing basic collection timeline information).

Creation time An integer with the number of the version at which this tree timeline begins to exist.

Deletion time An integer with the number of the version at which this tree timeline ceases to exist.

Value a CLOB serialization of the tree timeline object.

With this layout, it is possible to perform read and write operations more efficiently. For example, to access a version, n, the SQL query

```
SELECT *
FROM c1
WHERE creation <= n AND n < deletion
```

returns all relevant trees without accessing the others, relying on RDBMS optimizations.

Updating a single tree timeline is also less expensive, as it only requires to tamper with its entry, instead of having to additionally reserialize the entire collection timeline object.

4.6.5 Indices

Sausalito has proprietary support for indices. They are described in details at [45].

For example, the following index declaration:

```
declare index
on nodes collection(stocks)
by ./data(Country) as xs:string, ./data(Symbol) as xs:string
```

Evaluates the key values (country name, symbol string) for each node in the domain expression `collection`(my:stocks) set as the context item. It then stores them in a table (each row contains the node reference and the corresponding keys.)

It then allows efficient retrieval of the nodes given key values:

```
probe-index-point("United States", "AAPL")
```

Index construction can be done manually or automatically. An implementation may also use indices while optimizing a query.

These indices can be extended to support time if the domain expression evaluates to nodes within a single collection timeline. The index table is augmented with two additional columns modeling the validity domain of a (node reference, key values) entry: the version URI from which it applies, and the version URI from which it no longer applies.

The index generation algorithm is modified in such a way that:

- instead of deleting an entry, it sets the ending version URI thereof to the current version
- upon inserting a new entry, it sets the starting version URI thereof to the current version

The existing probing functions are kept and implicitly probe on the last version (i.e., entries for which the ending version URI is not set).

New probing functions are introduced to allow probing the past. For example, the following expression retrieves nodes with given key values from the first version of the collection:

```
probe-index-point(
  "United States",
  "AAPL",
  vng:collection-versions("stocks")[1]
)
```

4.6.6 Tree and Collection Timeline Clustering

If the storage layer is not stored on an RDBMS layer on the same computer as the Web server, but is stored remotely, then accessing an entire collection timelines (all its tree timelines) is expensive, but accessing each tree timeline individually might lead to inefficiency as well. An intermediate solution is provided by organizing the tree timelines in clusters. The issue then comes down to finding the best way to partition the data to minimize the number of pages read, or, equivalently, to maximize the usefulness of each page.

This problem was solved by Chien, Tsotras and Zaniolo in [56] with a technique called Usefulness-Based Copy Control (UBCC). With this technique, timestamped XML elements (in our case, these would be π-nodes) are organized in constant-size pages.

Each of them is identified with Sparse Preorder and Range (SPaR) [57] (in our case, we can use ORDPATH embedded in our URIs). Hence, using a snapshot index [77] to get all relevant pages, it is possible to reconstruct any version.

UBCC's main idea is that if the usefulness (i.e., the percentage of space which is relevant for the current version) of a page sinks below a threshold, the items of this page which are relevant for the current version are copied to a new page and the page is invalidated for the current version. Using this scheme allows version reconstruction in a time which is linear in the size of the target version.

If versioning is done at the collection level (see Section 3.2), we need to support synchronized collection timeline storage (as opposed to tree timeline storage originally). A synchronized collection timeline can be regarded as a "bigger tree" (with a virtual root), so that the same storage and reconstruction model as in UBCC can apply.

In the illustration on Fig. 4.10, we use ORDPATH identifiers [75], used only once within a collection. To generalize ORDPATH to a collection, we also use the ordering mechanism (1, 2.1, 3, ...) for the root of each tree. These identifiers allow to unambiguously reconstruct each π-tree, as well as the entire collection π-forest.

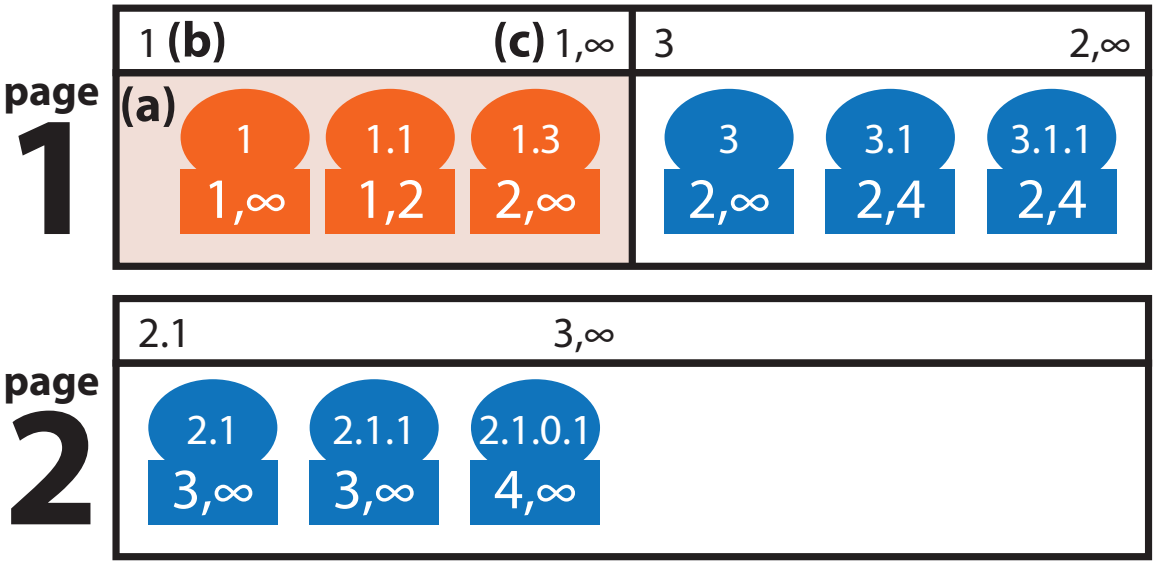

Figure 4.10: *Two sample UBCC pages containing three π-trees, like (a). A π-tree has a root ORDPATH (b) and an overall timestamp (c).*

However we suggest to introduce a "single-page" mode for large collections with small tree timelines, which reduces the number of pages used when retrieving small parts of such collections:

- The node timelines of a tree timeline which is smaller than a page are stored on the same page, in one piece, while collection timelines can be spread over several pages. New node timelines are inserted in place instead of being appended to the current page, which maintains the single-page consistency.

- The copy operation in UBCC (i.e., copying nodes that are still alive to a new page) is performed not only upon lack of usefulness, but also when a page becomes full because of in-place updates. Technically, this splits the corresponding tree timelines into an older, no longer updated, and a newer, fresh representation (see Fig. 4.11).

- For efficiency, each tree timeline stored on a single page can be assigned a timestamp interval (the smallest interval containing all intervals of its node timelines). That way, the snapshot index technique can be applied in a coarser way, with fewer intervals.

- If the current version of a tree timeline can no longer fit on a single page, we revert to the original UBCC model by relaxing the single-page constraint.

4.7 Performance Measurements

We performed measurements to check that adding versioning features to our engine did not lead to a significant loss of performance for traditional queries.

4.7.1 The implementation

We built a prototype based on Sausalito [49], implementing the π-tree data structure and the instantiation and update algorithms. We are not using clustering. Instead, each tree timeline as well as each collection index is stored as a CLOB in a relational DBMS. We extended Sausalito's index support to allow indices on the last version.

Our measurements compare the existing, non-versioning Sausalito with the versioning-enabled prototype.

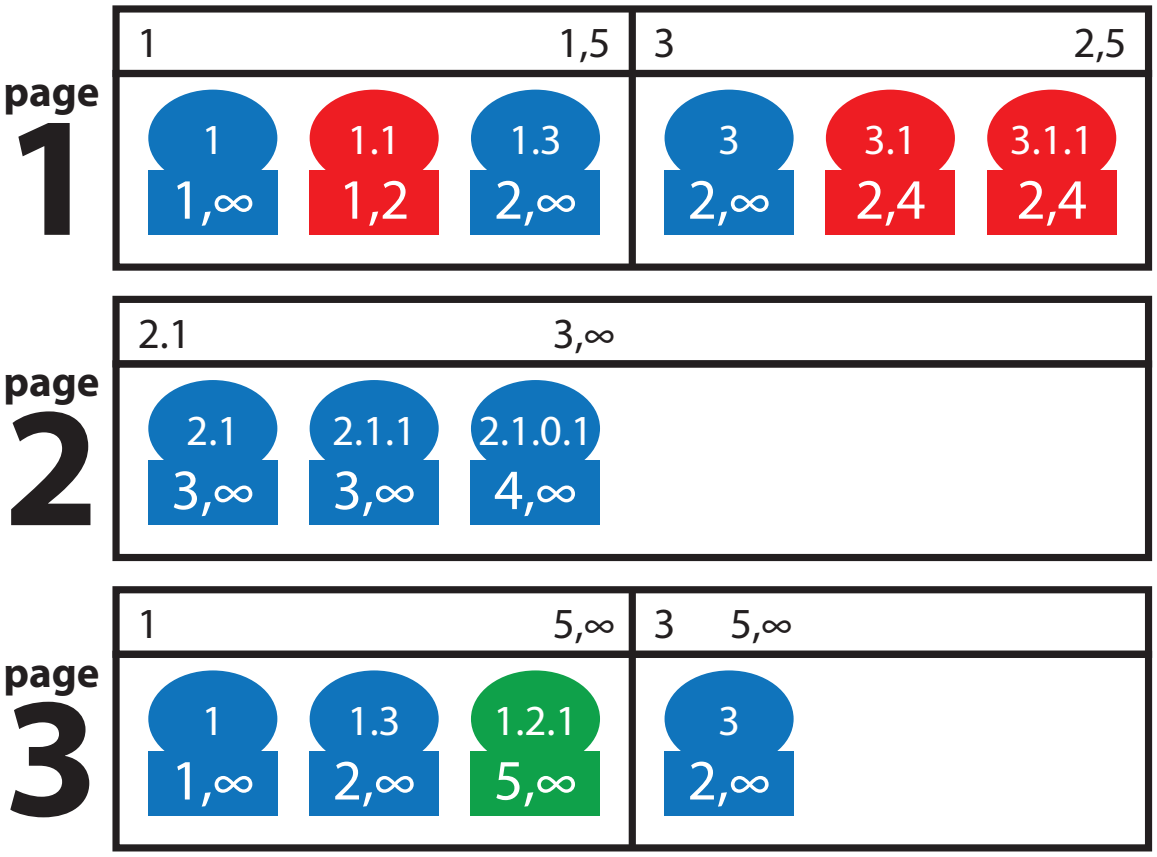

Figure 4.11: *Page 1 is full. To insert the green π-node 1.2.1, the current (not red) π-nodes 1, 1.3 and 3 need to be copied to a new page.*

4.7.2 Query classification

For convenience, we group queries into three kinds:

Traditional queries do not use versioning facilities (time axes, versioning functions), are automatically reading the local, checked out version and potentially creating a new version. They can be classical (XQuery core), updating or scripting queries.

Time-travel queries correspond to classical queries that are executed on a given past version.

Spacetime queries are the most general queries and can travel back to any version. They can be classical, updating or scripting (in the latter case they may perform explicit checkins or checkouts).

4.7.3 The TPoX benchmark

The measurements are based on the TPoX framework, which provides financial data and queries for measurement purposes.

TPoX relies on three collections:

Securities A collection of securities with related data (names, symbols, quotes, last bid, last ask).

Customers A collection of customers, with up to seven accounts (depots) per customer.

Orders A collection of buying or selling orders placed by customers.

for which it provides a larger set of sample data.

It provides transactions on top of this data:

Queries Read-only queries such as retrieving an order or a security, get an account statement or find the maximal order placed by a given customer.

Deletes Delete a customer or an order.

Inserts Insert a new customer or a new order.

Updates Open or close an account, update the price of a security, update or execute an order.

The general setting is that each of the possible transactions has a weight. There is an initial bulkload. A step then consists of the selection of a transaction with a probability proportional to its weight, its execution, and the measurement of the execution time. This step is repeated a large number of times. At least for a non-versioned database, a steady state should be reached, as the weights of insertions and of deletions compensate each other. At the end of the process, it is possible to analyze the data for each transaction kind.

4.7.4 Adaptation of TPoX for Version Control Measurements

With version control enabled, each of the three collections mentioned above is versioned. The exact same queries can be run on top of the repository, which will result

in checking out, reading and updating the last version of each collection. In addition, a new time-travel query (based on the one which gets the maximum order placed by a customer) is used. It goes to a past version of the orders' collection instead of working on the last version.

The measurements are conducted against the version-control-enabled Sausalito as well as against the original Sausalito (for which the time-travel query is executed against the last version for comparing the efficiency of time travel).

The initial bulkload populates the database with 200 customers, 2000 orders and 1000 securities (the logical size of the last version will actually remain at this level, since inserts and deletes are compensated). Then 100,000 transactions are executed, which involves roughly (as this is random) 1000 customer insertions and as many deletions, 7,000 order insertions and as many deletions, 8,000 updates, 70,000 read-only traditional queries and 1,000 time-travel queries. This involves data on a scale between 10 MB and 100 MB.

4.7.5 Design goals

Before implementing a prototype, we formulated three design goals aiming at almost no loss of performance for traditional queries (with "about the same", we mean in the same complexity class, with a constant factor as small as possible):

1. Traditional queries should be executed on a versioning engine in about the same time as on a traditional engine.
2. Traditional updating queries should be executed in about the same time as with a traditional engine.
3. The physical repository size of a versioning engine should grow no more than if there were no node deletions.

4.7.6 Results

The following graphs show measurements on a deleting query, an inserting query, a read-only query, an updating query and a time-traveling query. On each graph, the x-axis corresponds to the x-th time the given kind of query is executed. The y-axis shows, in milliseconds, the execution time of this x-th execution.

The x-axis can also be interpreted as the size of the physical repository of the time machine (for each graph in parallel, starting with the initial bulkload size, and reaching the maximum size at the end of the experiment), as it grows linearly with time.

Read-only queries

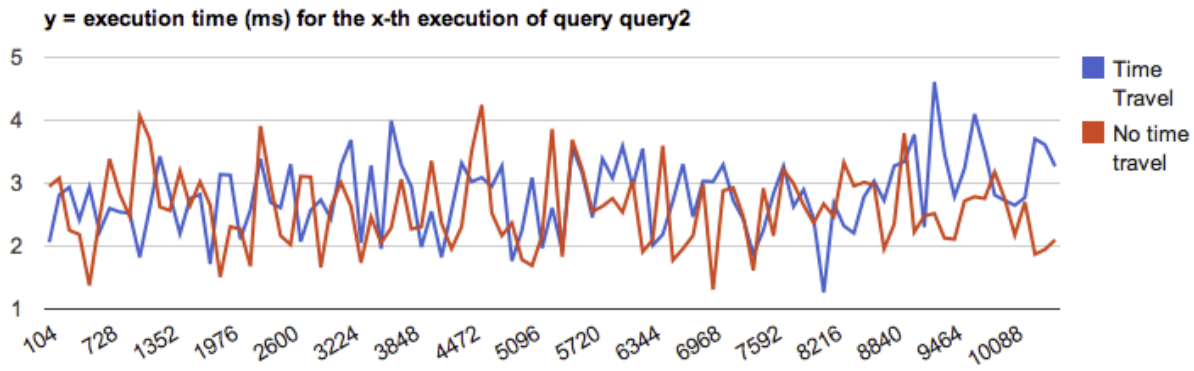

Figure 4.12: *Performance for an index-using read-only query.*

In read-only queries, Sausalito is in a steady state, because the collections are of constant size in average.

For read-only queries which make use of an index on the local version (Fig. 4.12), the time machine is also in a steady state, although the physical size of the collection timelines is increasing linearly. This is because the queries can be answered using an index and thus response time is independent of repository size. The average time is of the same order of magnitude as Sausalito. This shows that there is no loss of performance for point queries.

For read-only queries which iterate over a collection (no use of an index) such as on Figure 4.13, Sausalito is in a steady state, because the collection is of constant size in average, though it is slightly increasing.

In the early stage, the time machine is in a steady state and increases a bit as well, but stays in the same complexity class as the original Sausalito implementation. This shows that Design Goal 1 is fulfilled.

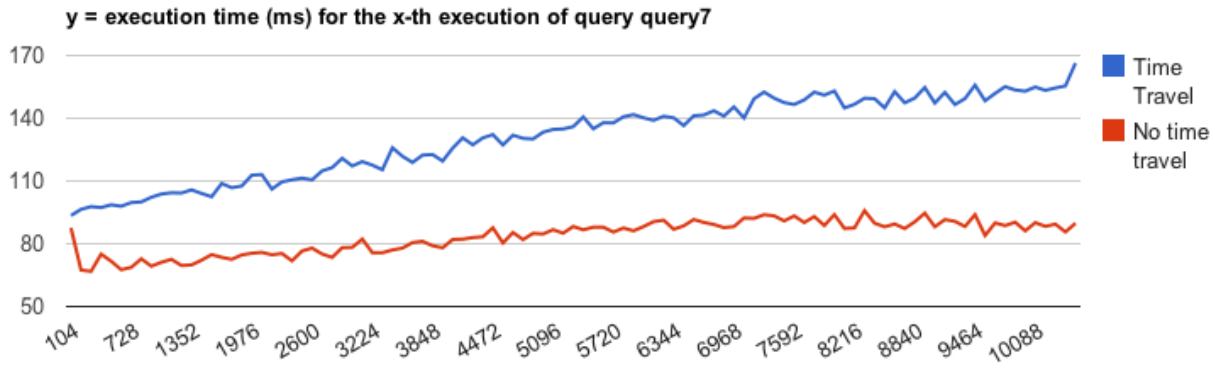

Figure 4.13: *Performance for an iterating read-only query.*

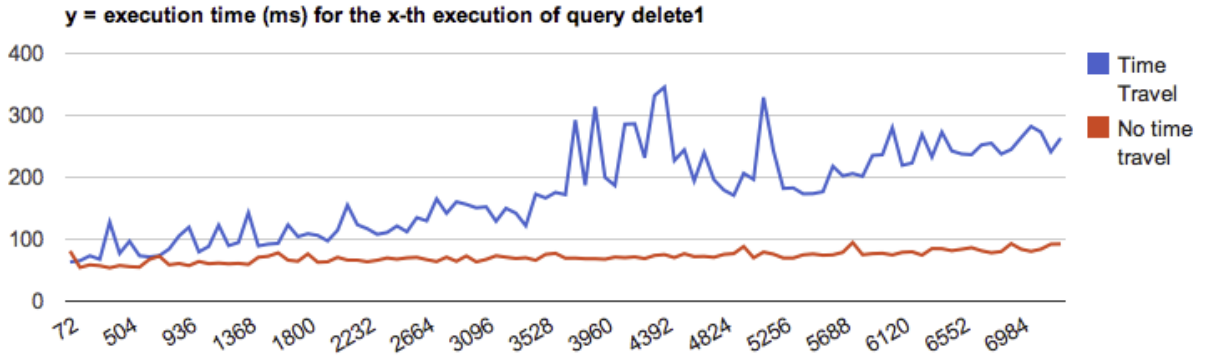

Figure 4.14: *Performance for deleting a node from a collection.*

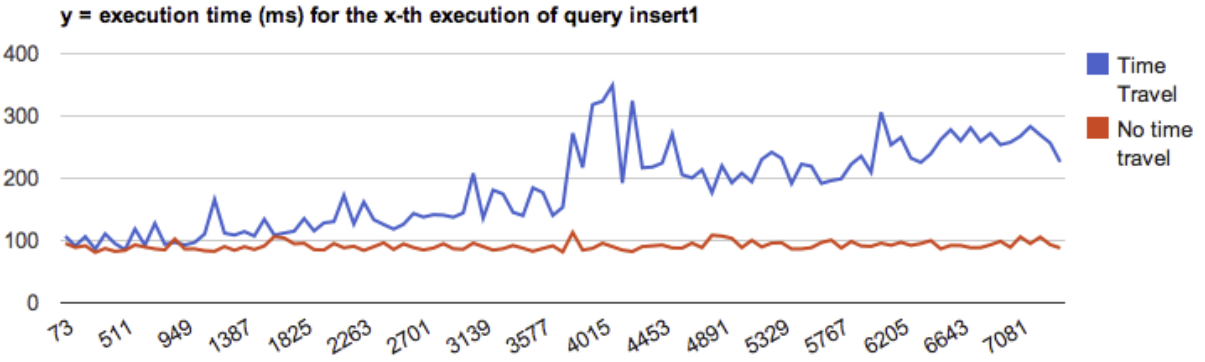

Figure 4.15: *Performance for inserting a node from a collection.*

Deleting and inserting queries

For inserting and deleting queries (4.14 and 4.15), Sausalito is in a steady state (as many inserts as deletes, so that collection size in constant). It is only slightly increasing, which might be due to memory leaks.

The time machine measurements are increasing linearly. This is due to the serialization format of a collection in Sausalito (as a single CLOB), whose size keeps increasing, as there are no deletes in a version control system. The entire collection index needs to be rewritten, because the number of the last version of the collection must be updated. This can be solved with a more adequate mapping in the database such as the one suggested in Section 4.6.4. Incidentally, this shows that the size of a collection increases linearly over time, which is a hint that Design Goal 3 is fulfilled.

Updating queries

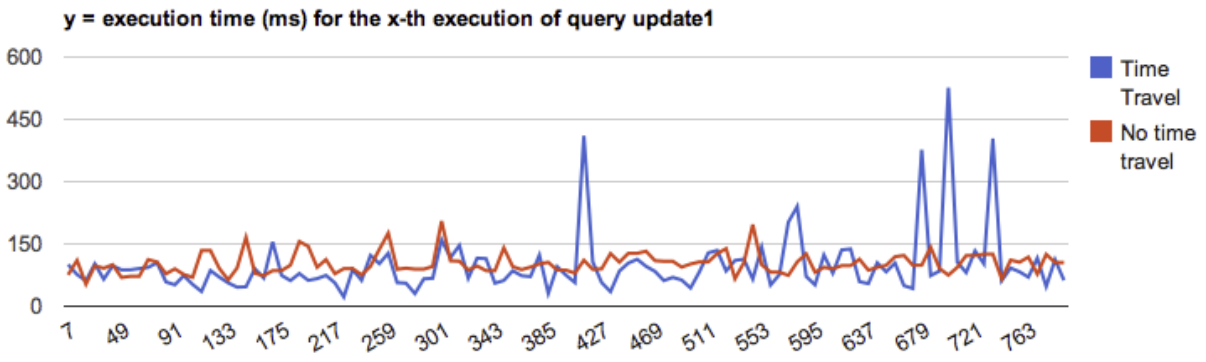

Figure 4.16: *Performance for updating a node in a collection.*

In the graph shown in Fig. 4.16, the time machine is in a steady state: Design Goal 2 is fulfilled.

For larger collections though, the time machine measurements begin to increase linearly. Like for inserting or deleting queries, this is due to the serialization format of a collection in Sausalito.

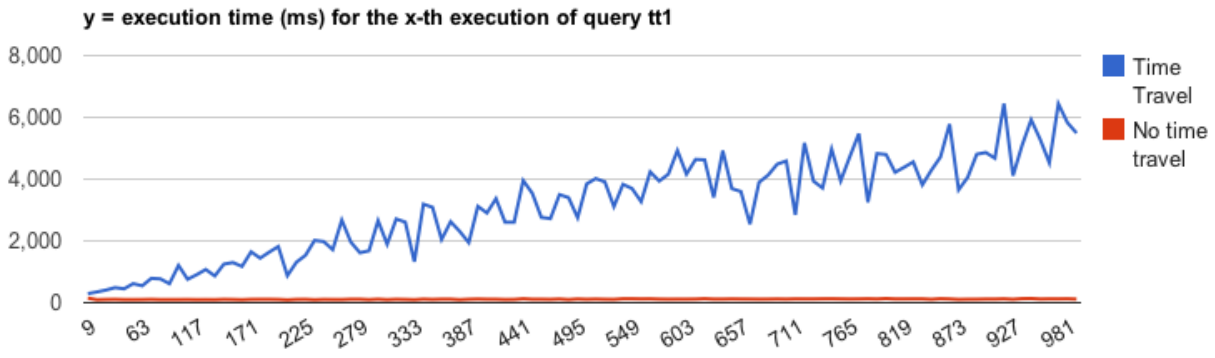

Figure 4.17: *Performance for an iterating time-travel query.*

Time-travel queries

Time-travel queries are compared against current-version traditional queries in Sausalito, so that Sausalito is also in a steady state. The time machine needs a linear time at first, then stabilizes. The reason for this is that tree timelines that still exist in the local version are cached, whereas older ones are dropped when not needed, and need to be loaded on demand. At the beginning of the benchmark execution, there are few versions, so that almost all trees are still in memory, whereas with time, more and more tree timelines need to be loaded from the storage layer. This is a compromise that has to be made if one wants to keep a stable memory usage over time. But of course, one could also load all tree timelines in memory, which would reduce time travel time.

4.8 Related work

Example of versioned database technologies are Oracle Flashback [28] and Microsoft Immortal DB [70], which allow non-destructive querying of the past versions of the database.

Zholudev and Kohlbase developed TNTBase [81] which is a versioning system for versioning XML documents on top of an SVN repository, linked to an XML database. TNTBase seems to focus on document-oriented versioning, e.g., deltas are regular SVN deltas, whereas we attempt to bridge the gap between documents and data.

There are non-XML proposals for versioning semi-structured data. Combi et al. [59] introduce a data model and an SQL-like query language.

There are also other proposals for extending XML data models. Many approaches work

with valid time intervals and introduce it explicitly in XML documents, so that queries have to be aware of the syntax chosen (i.e., do filtering with valid-time attributes). For example, Amagasa et al. [43] as well as Mendelzon et al. [73] extend the XPath data model with valid-time intervals. Dengfeng and Snodgrass [67] extend the XQuery language, also with valid-time intervals, and XQuery queries either need to be aware of the implementation (representational queries) or need to be transformed to a corresponding representational query. Fusheng and Zaniolo [78] suggest an XML versioning system which does not tamper with the data model, but here also, an explicit valid-time parameter is introduced in XML documents. Furthermore, the latter approach requires that the document is valid against a DTD: the DTD for the corresponding time-aware XML document is algorithmically deduced from the DTD of the original document. In our versioning system, XML data needs not be valid, as we are willing to leverage the fact that semi-structured data can exist without a schema.

Dyreson [60], like us, focuses on transaction time and extends XPath with time axes. However, his approach is based on the concept of an *observant system*, which has access to documents as well as reading and modifying times, but cannot update them. Our approach, on the other hand, follows very closely the evolution of XML data from the inside because it is seamlessly integrated with the XQuery Update Facility, which is the language to update XML data. Furthermore, our deltas are serialized XQuery Update's Pending Update Lists. Dyreson also suggests time axes, but the approaches are different: he defines the concept of known or assumed status (since the observer is outside), uses transaction time literals and introduces new node tests. We use the concept of version number and chose a more concise approach (all of the time travel information is in the axes).

Shu-Yao, Tsotras, Zaniolo and Donghui [56], [57], [77] provide an XML Document versioning storage scheme. They store the document as a set of nodes, clustered on several pages. Usefulness-based copying is used for efficiency, and each version of the document can be reconstructed thanks to the SPaR labeling scheme and the snapshot index. This is complementary to the contribution of this paper, and we base our clustering technique on their results, as explained in Section 4.6.6.

Finally, Ian Lynagh introduced the theory of patches [71], which defines operations on patches (composition, inversion...). darcs [76] is an example of changed-based versioning system using this formalism, as opposed to version-based. In our work, we chose to design a version-based time machine for efficient storage (a timeline can be efficiently stored as a single tree), but the patch theory seems general enough to support PUL-based patches. Federico Cavalieri worked on more operations on PULs [51],

in addition to composition (which he calls aggregation), like inversion, conflict resolution, reduction, which he implemented in an XQuery library for Zorba. This suggests the possibility of a more powerful, change-based XML time machine.

4.9 Conclusion

We presented an XML-aware, XQuery-aware versioning system which seamlessly integrates time to the data model. This versioning system bridges the gap between data and document versioning and allows for powerful queries against versioned data. We implemented a first basic prototype of this versioning system, based on the Sausalito Web Application Server [49] and the Zorba XQuery engine [35]. The extensions we made are backward-compatible with non-versioning queries and there is no significant loss of performance for traditional queries.

Part II

Scale

Chapter 5

An intuitive XQuery Scripting Grammar

This chapter proposes a new syntax for the XQuery Scripting extension, with the requirements that this syntax be intuitive for programmers used to languages like C or Java. The challenge is that the requirements for this new grammar lead to a cycle in the expression precedences. This cycle is avoided by introducing statements to the grammar, in addition to expressions.

5.1 XQuery Scripting

XQuery and the XQuery Update Facility are side-effect-free programming languages. XQuery declaratively constructs an XDM output. XQuery Update Facility declaratively constructs an XDM output and/or a Pending Update List which can then be applied to an underlying storage layer.

XQuery Scripting was introduced in order to support side effects. It is currently published as a W3C working draft. Its main features are:

The Apply expression This expression consists of semi-colon-separated (possibly updating) expressions. Its semantics is that each Pending Update List returned by an operand expression is applied.

In the following example, a PUL with an inserting update primitive is created and applied, then a PUL with a replacing update primitive is created and applied.

```
insert node <Stock><Symbol>AAPL</Symbol></Stock>
```

```
    into doc("stocks.xml")/Stocks;
replace value of node
  doc("stocks.xml")/Stocks/Stock[@Symbol="AAPL"]
  with "AAPL";
```

An evaluation order Since the order in which side effects occur is important, XQuery Scripting introduces an evaluation order on all XQuery expressions. In practice, this evaluation order is not relevant for non-side-effecting expressions, which leaves all optimizations open for core XQuery programs.

In the example query above, the second updating expression sees the effects of the first updating expression (it sees that a stock with the symbol AAPL has been inserted). Hence, it must be evaluated after the first one.

Variable assignment Variables can be declared (globally or in blocks) and assigned a new value during the execution of the program.

```
block {
  variable $stock;
  set $stock := doc("stocks.xml")/Stocks/Stock[@Symbol="AAPL"];
}
```

While loop Unlike FLWOR expressions (with for loops), in which the iteration tuples are computed in advance, a while loop reevaluations its test expression after each execution of its main expression and sees the side effects of the latter.

```
set $t := 0;
while($t < 10) {
  replace value of node
    $stock/Quote with $stock/Quote * pow(1.01, $t);
  if($stock/Quote > 100)
    then set $t := $t + 2
    else ();
}
```

The current working draft has a certain number of limitations, and this parts contributes a proposal to solve them.

5.2 The solved issues

5.2.1 State of the art

Semi-colon location In the current XQuery Scripting working draft, the scope of a semi-colon is an entire expression (Expr non terminal). This is very natural for simple expressions, as in:

```
delete node $x;
```

but in more complex control flow expressions, this leads to an unnatural syntax:

```
if (cond)
then
  insert node <a/> into $x
else
  delete node $y;
```

A programmer trained on languages like C or Java is used to semi-colons on both the then and the else clause.

Semi-colon precedence In the current XQuery Scripting proposal, the semi-colon has the lowest precedence of all operators and expressions.

The following example can be misleading, as the semi-colon is not only applying the delete expression, but the entire FLWOR expression.

```
for $x in local:set()
return
  delete node $x;
```

This is a crucial difference, as this means that the side effects caused by the delete expression are visible only after the evaluation of the entire FLWOR expression. If one wants the side effects to be visible after each evaluation of the return clause (for each tuple bound to $x), one needs to modify the precedence with parentheses:

```
for $x in local:set()
return
  (delete node $x;)
```

or with a block:

```
for $x in local:set()
return
  block { delete node $x; }
```

Doing nothing The current XQuery Scripting proposal does not allow empty blocks. Doing nothing is expressed with an empty parenthesized expression, as follows:

```
if (cond)
then block {
  delete node $x;
  delete node $y;
}
else ()
```

This leads to an asymmetry between blocks and parenthesized expressions.

Ignoring the result For each block, the current XQuery Scripting proposal returns the value returned by the last expression (among the semi-colon separated expressions). Putting a semi-colon or not after the last expression does not make any difference. Both of the following expressions return 3:

```
block {
  delete node $x;
  3;
}
```

```
block {
  delete node $x;
  3
}
```

5.2.2 Proposed syntax

The proposal presented in this section allows the following syntax for each of the aforementioned issues:

Semi-colon location The programmer can, and even must, use semi-colons both in the then and in the else clause (unless he is using blocks):

```
if (cond)
then
  insert node <a/> into $x;
else
  delete node $y;
```

Semi-colon precedence The semi-colon precedence is made higher than control flow expressions, i.e., in the following example, the scope of the semi-colon is only the delete expression:

```
for $x in local:set()
return
  delete node $x;
```

It is still possible to override this precedence with parentheses:

```
(for $x in local:set()
return
  delete node $x);
```

Do-nothing Empty blocks are allowed (and the keyword block is no longer necessary):

```
if (cond)
then {
  delete node $x;
  delete node $y;
}
else {}
```

The else clause is made optional:

```
if (cond)
then {
  delete node $x;
  delete node $y;
}
```

Ignoring the result A semi-colon always discards the result of the expression it applies. The following program returns the empty sequence:

```
{
  delete node $x;
  3;
}
```

whereas the following expression returns 3:

```
{
  delete node $x;
  3
}
```

5.3 Main challenge

The requirements above imply a cycle in the precedence ordering of expressions. Consider the following program:

```
delete node
  if ($c)
  then a
  else b;
```

First, the semi-colon must have lower precedence than a primitive updating expression (delete node...). This means that the program must be equivalent to:

```
(delete node
  if ($c)
  then a
  else b);
```

Second, the semi-colon must have higher precedence that a conditional expression, so that both the then clause and the else clause can have one, as seen in the last section. This means that the program must be equivalent to:

```
delete node
```

```
if ($c)
then a
else (b;)
```

Finally, a conditional expression and a primitive updating expression have equal precedence in the XQuery grammar.

Consequently, if p is the precedence function (associating each expression to an ordinal characterizing its precedence):

$$p(;) < p(\text{delete...}) = p(\text{if... then... else...}) < p(;)$$

which is contradictory.

This means that the requirements cannot be fulfilled using an Apply expression.

5.4 Detailed proposal

The XQuery Scripting extension proposal is a superset of XQuery 3.0 and XQuery Update Facility 3.0[1]. This means that any XQuery 3.0 program and any XQuery Update Facility 3.0 program are also XQuery Scripting Extension proposal programs.

5.4.1 Statements

The main idea behind the resolution of the challenge presented in the last section is the introduction of statements in the grammar, in addition to expressions.

Statements are different from expressions in that:

- They can have direct side effects (expressions can only have indirect side effects).
- They do not return anything (expressions return an XDM instance and/or a Pending Update List).
- They can be concatenated without a separator (expressions are concatenated using commas).

[1]The XQuery Update Facility 3.0 is still in early stage and no working draft has been made publicly available yet - we are assuming that its syntax is close to XQuery Update Facility 1.0 with minor adaptations to XQuery 3.0, such as the %fn:updating annotation.

The following constructs, available as expressions in the current XQuery Scripting working draft (or in earlier versions), are made statements. In particular, they all end with semi-colons. The almost same semantics is retained:

Apply statements An apply statement applies the Pending Update List returned by its operand expression.

```
delete node $x;
```

Variable declaration statements A variable declaration statement declares (and possibly assigns a value to) a new variable. Its scope is the remainder of its innermost containing block or control flow clause. Unlike in the current working draft, variable declarations can be made anywhere in a program, not necessarily at the beginning of a block.

```
variable $counter := 0;
```

A variable can also be made constant with an annotation:

```
%ann:nonassignable variable $pi := 3.14;
```

Variable assignment statements A variable assignment statement assigns a new value to an assignable variable in scope.

```
$counter := $counter + 1;
```

Control flow statements: While, Break, Continue and Exit These statements give the programmer some control about the instruction flow. In a while loop, each evaluation of the test expression sees the side effects of former evaluation of the main statement:

```
while ($counter < 10)
  $counter := $counter + 1;
```

A break expression exits a while or FLWOR statement, a continue expression goes to the next evaluation of the test expression (while) or to the next tuple (FLWOR). An exit expression exits a function, returning the specified value:

```
break;
continue;
exit returning 0;
```

There are also statements which are counterparts to existing expressions:

Block statements A block statement groups several statements in one grammatical unit, and serves as a scope for variables. The last statement must end with a semi-colon (or be a block statement). The keyword block is removed.

```
{
  variable $x := 1;
  $x := $x * $x;
  replace value of node $y with $x;
}
```

Conditional statements Both the then clause and the else clause must be statements.

```
if ($x/flag > 1)
then
  $y := 2;
else {
  delete node $x;
  $y := 1;
}
```

FLWOR statements The return clause is a statement.

```
for $x in $nodes
return {
  delete node $x/child;
  $counter := $counter + 1;
}
```

Try-Catch statements The try and catch clauses are block statements.

Switch statement The return clauses are statements.

Typeswitch statements The return clauses are statements.

5.4.2 Composability of statements and expressions

The last section showed how expressions can be used in statements. It is also possible to nest statements in expressions. The main construct allowing this is a block expres-

sion. A block expression is like a block statement, except that there is an additional expression after the statement list.

The following is an example of a block expression, consisting of three statements (variable declaration, variable declaration, while loop) followed by an expression (variable reference):

```
{
  variable $x := 1;
  variable $y := 1;
  while ($y <= 10) {
    $x := $x * 2;
    $y := $y + 1;
  }
  $x
}
```

which returns 1024 (the final value of $x).

Block expressions are allowed as a standalone expression, and replace in the grammar all curly-brace-based constructs, such as node constructors:

```
<result>{
  variable $x := 1;
  variable $y := 1;
  while ($y <= 10) {
    $x := $x * 2;
    $y := $y + 1;
  }
  $x
}</result>
```

or try-catch expressions:

```
try{
  variable $x := 1;
  variable $y := 1;
  while ($y > -1) {
    $x := $x div $y;
    $y := $y - 1;
```

```
  }
  $x
}
catch * {
  "Division by zero!"
}
```

as well as ordered/unordered, validate, extension expressions, and (global and inline) function body expressions.

Furthermore, at all these places (except as a standalone block expression or in a try-catch expression[2]), the final expression is optional. If there is no final expression, the empty sequence is returned by the operand.

5.4.3 Sequential expressions

The last section showed that expressions may contain statements. The direct consequence of this is that they can have indirect side effects, such as applying updates to an XDM instance, altering the dynamic context, or affecting the flow of control. Such expressions are called *sequential*.

The current working draft classifies expressions in three classes: simple, updating and sequential. The classification is made by defining composability on each kind of expression.

In order to simplify this, we move sequentiality to another, separate dimension. A simple expression is now a non-updating, non-sequential expression. An updating expression is now an updating, non-sequential expression, a sequential expression is a non-updating, sequential expression.

Sequentiality is defined as follows. An expression is sequential if it contains, directly or nested:

- a call to a sequential function.
- an assignment to a free variable (i.e., declared outside of the expression).
- an apply statement, the operand of which is updating.
- a free break or continue statement (i.e., without FLWOR or while statement at an intermediate level in the expression tree).

[2]to avoid a parsing conflict with block statements or try-catch statements

- an exit statement.

5.4.4 Nested snapshots

The XQuery Update Facility introduces the concept of snapshot semantics. An XQuery Update Facility program is evaluated on a single snapshot, i.e., there are no side effects visible during its evaluation. The produced Pending Update List is applied to the underlying storage layer *after* the program has completed its execution.

We define a snapshot as a lapse between two consecutive changes. This forbids nested snapshots. Since an updating expression is evaluated on a single snapshot, there is a global composability constraint that an expression may not be both updating and sequential (this is similar in spirit to the existing working draft).

5.4.5 Evaluation order

The side effects of a sequential expression are immediately effective and are visible to subsequent expressions. Because of their side effects, sequential expressions must be evaluated in evaluation order.

The evaluation order of the newly introduced statements, as well as control flow expressions, is defined explicitly according to their semantics.

The evaluation order for existing expressions is defined as being from left to right. The parameters passed to a function must be evaluated before its call. An expression must be evaluated exactly once for each evaluation of its parent expression, except when specified otherwise (for example in a conditional expression).

As in the current working draft, optimizations may still be made on non-sequential expressions: operands may be evaluated in any order as this does not affect the semantics of these expressions.

5.4.6 The complete grammar

This section presents the complete scripting grammar proposal.

New query body for main modules

```
MainModule  ::=  Prolog Program
Program  ::=  StatementsAndOptionalExpr
```

Mixing Expressions and Statements

```
Statements  ::=  Statement*
StatementsAndExpr  ::=  Statements Expr
StatementsAndOptionalExpr  ::=  Statements Expr?
```

Statements

```
Statement  ::=  ApplyStatement
                | AssignStatement
                | BlockStatement
                | BreakStatement
                | ContinueStatement
                | ExitStatement
                | FLWORStatement
                | IfStatement
                | SwitchStatement
                | TryCatchStatement
                | TypeswitchStatement
                | VarDeclStatement
                | WhileStatement
ApplyStatement  ::=  ExprSimple ";"
AssignStatement  ::=  "$" VarName ":=" ExprSingle ";"
BlockStatement  ::=  "{" Statements "}"
BreakStatement  ::=  "break" "loop" ";"
ContinueStatement  ::=  "continue" "loop" ";"
ExitStatement  ::=  "exit" "returning" ExprSingle ";"
```

```
FLWORStatement  ::=  InitialClause
                     IntermediateClause*
                     ReturnStatement

ReturnStatement  ::=  "return" Statement

IfStatement  ::=  "if" "(" Expr ")"
                  "then" Statement
                  "else" Statement

SwitchStatement  ::=  "switch" "(" Expr ")"
                      SwitchCaseStatement+
                      "default" "return" Statement

SwitchCaseStatement  ::=  ("case" SwitchCaseOperand)+
                          "return" Statement

TryCatchStatement  ::=  "try" BlockStatement
                        ("catch" CatchErrorList BlockStatement)+

TypeswitchStatement  ::=  "typeswitch" "(" Expr ")"
                          CaseStatement+ "default"
                          ("$" VarName)?  "return" Statement

CaseStatement  ::=  "case" ("$" VarName "as")?  SequenceType
                    "return" Statement

VarDeclStatement  ::=  Annotation* "variable"
                       "$" VarName TypeDeclaration?
                       (":=" ExprSingle)?
                       ("," "$" VarName TypeDeclaration?
                       (":=" ExprSingle)?)* ";"

WhileStatement  ::=  "while" "(" Expr ")" Statement
```

Expressions

(Separating control-flow expressions)

```
ExprSingle ::= ExprSimple
             | FLWORExpr
             | IfExpr
             | SwitchExpr
             | TryCatchExpr
             | TypeswitchExpr

ExprSimple ::= QuantifiedExpr
             | OrExpr
             | InsertExpr
             | DeleteExpr
             | RenameExpr
             | ReplaceExpr
             | TransformExpr
```

(Direct element constructors)

```
CommonContent ::= PredefinedEntityRef
                | CharRef
                | "{{"
                | "}}"
                | BlockExpr
```

(Computed element constructors)

```
ContentExpr ::= StatementsAndExpr

CompDocConstructor ::= "document" BlockExpr

CompAttrConstructor ::= "attribute" (EQName | ("{" Expr "}"))
                        ("{" "}" | BlockExpr)

CompPIConstructor ::= "processing-instruction"
                      (NCName | ("{" Expr "}"))
                      ("{" "}" | BlockExpr)

CompCommentConstructor ::= "comment" BlockExpr

CompTextConstructor ::= "text" BlockExpr
```

(Block expression)

```
PrimaryExpr  ::=  Literal
                | VarRef
                | ParenthesizedExpr
                | ContextItemExpr
                | FunctionCall
                | OrderedExpr
                | UnorderedExpr
                | Constructor
                | FunctionItemExpr
                | BlockExpr
BlockExpr  ::=  "{" StatementsAndExpr "}
```

Function body

```
FunctionDecl  ::=  "function" EQName "(" ParamList?  ")"
                   ("as" SequenceType)?
                   ("{" StatementsAndOptionalExpr "}"
                   | "external"))
```

5.4.7 An LL grammar

The grammar presented in the last section is LR(1), but requires an arbitrary lookahead for top-down parsers. This parsers can nevertheless implement a subset of the language. The grammar changes are as follows.

For implementors using LL-parsers, the grammar described in this specification (which is LR(1)) needs an arbitrary lookahead (as opposed to a lookahead of 1 for LR-parsers). This arbitrary lookahead can be removed by using a subset of this proposal specified by the following changes. This makes the grammar LL(2).

In a nutshell, block expressions are not available, and statements that could be mistaken for the beginning of an expression (with a lookahead of 2) need to be put in block statements. This happens in function bodies, node constructors and block expressions.

Primary expressions unchanged, no block expression

```
PrimaryExpr  ::=  Literal
                | VarRef
                | ParenthesizedExpr
                | ContextItemExpr
                | FunctionCall
                | OrderedExpr
                | UnorderedExpr
                | Constructor
                | FunctionItemExpr
```

Statement non-terminal split into two non-terminals

```
Statement  ::=  Statement1 | Statement2

Statement1  :=  AssignStatement
                | BlockStatement
                | BreakStatement
                | ContinueStatement
                | ExitStatement
                | VarDeclStatement
                | WhileStatement

Statement2  :=  ApplyStatement
                | FLWORStatement
                | IfStatement
                | SwitchStatement
                | TryCatchStatement
                | TypeswitchStatement
```

Only non-ambiguous statements allowed before an expression

```
StatementsAndOptionalExpr  ::=  Statements1 Expr?

Statements1  :=  Statement1*
```

5.5 Conclusion

This Scripting proposal has been implemented in the Zorba 2.0 engine and is officially part of its release. It will also be available in the Sausalito Web application server when it upgrades to Zorba 2.0.

Chapter 6

Object-orientation in XQuery

This chapter analyzes how much of the object-oriented paradigm XQuery already supports, and proposes an extension adding behavior to XML nodes, leading to improved code reuse. The challenge is that this must integrate seamlessly with the already existing typing paradigm in XQuery and XML Schema.

6.1 Introduction

Besides the new requirements for semi-structured data, the last decades have seen the object-oriented paradigm rise. The main requirements behind it, as explained in a lecture about object-orientation given by Peter Müller at ETH [74] are:

- Modeling the real world with high fidelity
- Reusing existing code without modification
- Support in writing correct programs
- Cooperating components with well-defined interfaces

The object-orienting paradigm answers to these requirements with three main concepts, described in the lecture as well:

Object model Objects have a state, an identity, a lifecycle, a location, and behavior.

Interfaces and Encapsulation Objects have well-defined interfaces, behind which the implementation is hidden.

Classification and Polymorphism The objects can be classified in a hierarchy of abstract or concrete classes (tree or DAG). Objects belonging to a class can also be used wherever a less specific (supertype) class is expected.

The goal of this chapter is to investigate how XQuery presently integrates these concepts, and to extend it to give it more object-orientedness. The approach taken (attempt to emulate object orientation with the existing language in order to show limitations, then extend the language) is also similar in spirit to what is done with C in the introduction of the lecture mentioned above.

6.2 State of the art in XQuery and XML Schema

6.2.1 XQuery types

In XQuery, a node item or an atomic item can have a type among two kinds:

Simple types These are types for atomic items (such as 1 or "string") or for element nodes with simple content (such as `<element>value</element>`) or even attribute nodes (such as `attribute="value"`). This includes predefined types (such as `xs:integer` or `xs:string`, but also user-defined types (restricting the value space of other simple types).

Complex types These are types that define a layout for element nodes which have children, such as

```
<element>
  <myinteger>1</myinteger>
  <mystring>foobar</mystring>
</element>
```

This is very comparable to the difference between primitive types and classes in Java.

6.2.2 Complex types

Complex types define the layout in an element. There are several types of layout. One of them is the complex content layout, illustrated on Figure 6.1 by an element with six element children and an (anonymous) complex definition against which it is valid. Note that, unlike in Java classes, it is also possible to allow a certain number of children with the same name, for example no, one, two or three `Country` children in the example.

XML Schema also allows more powerful constructs, such as choices (similar to C++ unions).

The layouts shown in these example are complex content, these are the closest to classes in the object-oriented world. A complex type can also have a simple content (i.e., a single child text node, but possibly also attributes), as in

```
<element attr="value">other value</element>
```

or mixed content (children text and element nodes), as in

(a)

```
<Stock>
  <Name>XML Corp stock</Name>
  <Symbol>XMLC</Symbol>
  <Company>XML Corporation</Company>
  <Country>U.S.A.</Country>
  <Country>Switzerland</Country>
  <Quote>3.14</Quote>
</Stock>
```

(b)

```
<xs:complexType name="stock">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Symbol" type="xs:string"/>
    <xs:element name="Company" type="xs:string"/>
    <xs:element
      name="Country" type="xs:string"
      minOccurs="0" maxOccurs="3"/>
    <xs:element name="Quote" type="xs:decimal"/>
  </xs:sequence>
</xs:complexType>
```

Figure 6.1: *An element and the definition of its complex type (stock).*

`<element>this is <b>bold</b></element>`.

6.2.3 Validating an XML document

If an XML document is parsed correctly, it is said to be well-formed. If such a document is used in XQuery, all its nodes are annotated with the type `xs:untyped` and the user must manually convert its data to the desired format (`xs:integer`, `xs:string`...).

In addition to well-formedness, it is possible to validate an XML document against a schema. A schema defines the structure an XML document must have, e.g., what the children of an element must be. More than that, a schema defines user-defined complex types. Upon validation, the nodes will be annotated with a complex type.

For example, Figure 6.2 shows an example of document validated against a schema.

Once validated, the node `Security` will be annotated with the type **`security`**, and the node `Symbol` will be annotated with the type **`xs:string`**.

6.2.4 Some more examples

Let us take the scenario of an investment bank which maintains a database of securities and wishes to display a listing with all prices. The stock schema was defined in Figure 6.1. A schema for another kind of security, funds, is defined in 6.3. Figure 6.4 shows two functions computing the price for each of these kinds of security.

(a) security-schema.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="security">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Symbol" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Security" type="security"/>
</xs:schema>
```

(b) security-sample.xml

```
<?xml version="1.0"?>
<Security>
  <Name>Apple Corp.</Name>
  <Symbol>AAPL</Symbol>
</Security>
```

Figure 6.2: *A schema (a) and a document (b) valid against it.*

(a)

```
<xs:complexType name="fund">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Symbol" type="xs:string"/>
    <xs:element name="Manager" type="xs:string"/>
    <xs:element name="Underlyings" type="weighted-security-list"/>
  </xs:sequence>
</xs:complexType>
```

(b)

```
<xs:complexType name="weighted-security-list">
  <xs:sequence>
    <xs:element
      name="Underlying" type="xs:string"
      minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Weight" type="xs:decimal"/>
          <xs:choice>
            <xs:element name="Stock" type="stock"/>
            <xs:element name="Fund" type="fund"/>
          </xs:choice>
        </xs:sequence>
      </xs:complexType>
    </xs:element/>
  </xs:sequence>
</xs:complexType>
```

Figure 6.3: *Complex types for a fund*

6.2.5 Using XML Schema in XQuery

It is possible to write XQuery programs which do not use schemas. However, XQuery is strongly typed, so that even in this case, all XML elements have the type **xs:untyped**, and attributes have the type **xs:untypedAtomic**. Variables and function parameters do not need to be annotated with a type. Path expressions allow dynamic navigation in XML documents (the layout of which does not need to be known at compile time).

However, complex types defined in XML Schema can also be imported and used in XQuery programs. An XML node can be validated against a schema as shown on Figure 6.5. If the schema specifies a root element `Stock` with a type **stock** (a), a strict validation as shown in (b) will succeed. Otherwise, if only the complex type **stock** is defined (as in Figure 6.1), a lax validation is possible, but it requires an explicit type declaration in the XML hierarchy (c).

It is also possible to annotate function parameters and variables as shown on Figure

```
declare function get-stock-price($stock) {
  $stock/Quote
};
declare function get-fund-price($fund) {
  sum(
    for $underlying in $fund/Underlyings/Underlying
    let $weight := $underlying/data(Weight)
    let $security := $underlying/*[2]
    return
    $weight * (typeswitch($security)
      case element(*, stock)
      return
        get-stock-price($security)
      case element(*, fund)
      return
        get-fund-price($security)
      default return ()
    )
  )
};
```

Figure 6.4: *Functions for computing the price of stocks and funds*

(a)
```
<xs:element name="Stock" type="stock"/>
```

(b)
```
validate {
  <Stock>
    <Name>XML Corp stock</Name>
    <Symbol>XMLC</Symbol>
    <Company>XML Corporation</Company>
    <Country>U.S.A.</Country>
    <Country>Switzerland</Country>
    <Quote>3.14</Quote>
  </Stock>
}
```

(c)
```
validate type stock {
  <Stock>
    <Name>XML Corp stock</Name>
    <Symbol>XMLC</Symbol>
    <Company>XML Corporation</Company>
    <Country>U.S.A.</Country>
    <Country>Switzerland</Country>
    <Quote>3.14</Quote>
  </Stock>
}
```

Figure 6.5: *The XQuery validation instruction.*

6.6. For example, the function `get-stock-price` takes as a parameter an element with the complex type stock. At runtime, it will be checked that the (dynamic) type of the actual parameter matches this (static) annotated type.

Note how the price of a fund is obtained using a typeswitch construct to invoke the one or the other price getter function (Figure 6.4). This can be slightly refactored using the type hierarchy. Before going more in depth into static and dynamic types, we shortly introduce the type hierarchy in XML Schema/XQuery.

At this point, we could model stocks and funds, but adding a new kind of security re-

```
declare function
get-stock-price($stock as element(*, stock)) as xs:decimal {
  $stock/Quote
};
declare function
get-fund-price($fund as element(*, fund)) as xs:decimal {
  sum(
    for $underlying in $fund/Underlyings/Underlying
    let $weight as xs:decimal := $underlying/data(Weight)
    let $security := $underlying/*[2]
    return
    $weight * (typeswitch($security)
      case element(*, stock)
      return
        get-stock-price($security)
      case element(*, fund)
      return
        get-fund-price($security)
      default return ()
    )
  )
};
```

Figure 6.6: *Functions for computing the price of stocks and funds*

quires modifying the fund schema (to allow a new kind of underlying) as well as the typeswitch. This can be improved using the XML Schema type hierarchy and the XQuery subtype polymorphism.

6.2.6 Type hierarchy and subtyping

The XML Schema type hierarchy supports subtyping in two ways:

By restriction The value space of the subtype is a subset of the supertype value space. For example, `xs:integer` is derived by restriction from `xs:decimal`. Subtyping by restriction is available for simple types as well as complex types.

By extension Each subtype value is made of a supertype value augmented with additional children elements or attributes appended at the end of the element children. Subtyping by extension is only available for complex types.

Figure 6.7 shows how the stock complex types can be rewritten as a derivation by extension of the security complex type. Three additional allowed children are appended, i.e., a value of the extended type subsumes a value of the base type.

Figure 6.8 shows how the security complex type could be derived by restriction from a hypothetical security type allowing any number of symbols. Note that the allowed interval for the number of symbols ($[1, 1]$) is a subset of the allowed interval of the base type ($[0, +\infty]$), i.e., the value space of the restricted type is subsumed by the value space of the base type.

(a)

```
<xs:complexType name="security">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Symbol" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

(b)

```
<xs:complexType name="stock">
  <xs:complexContent>
    <xs:extension base="security">
      <xs:sequence>
        <xs:element name="Company" type="xs:string"/>
        <xs:element
          name="Country" type="xs:string"
          minOccurs="0" maxOccurs="3"/>
        <xs:element name="Quote" type="xs:decimal"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Figure 6.7: *A complex type and another complex derived by extension.*

(a)

```
<xs:complexType name="security-many-symbols">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Symbol" type="xs:string"
      minOccurs="0" maxOccurs="unbounded"
    />
  </xs:sequence>
</xs:complexType>
```

(b)

```
<xs:complexType name="security">
  <xs:complexContent>
    <xs:restriction base="security-many-symbols">
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Symbol" type="xs:string"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Figure 6.8: *A complex type and another complex derived by restriction.*

In order to validate an element which has a subtype of the expected type, one needs to specify the type to validate against, as shown in Figure 6.9. Assuming the validator expects the type **`security`**, if no type were specified, it would fail when reading the `Company` element. If the subtype is specified (here **`stock`**), it will validate it against this type and accept the element as valid since it is a subtype of the expected type.

6.2.7 Static type vs. dynamic type

In XQuery, all atomic or node values have a dynamic type. XQuery allows static typing, but does not requires (nor forbids) an XQuery engine to perform static analysis, i.e., detect type errors at compile time. At runtime however, it must be checked that the value of a variable or of a function parameter matches its static type if a static type is specified.

```
<Stock>
  <Name>XML Corp stock</Name>
  <Symbol>XMLC</Symbol>
  <Company>XML Corporation</Company>
  <Country>U.S.A.</Country>
  <Country>Switzerland</Country>
  <Quote>3.14</Quote>
</Stock>
```

Figure 6.9: *An element valid against the subtype.*

For example, in the following expression:

```
let $i := 1
let $a as xs:integer+ := 1
return $a[$i] + 2
```

at runtime, it will be checked that 1's dynamic type (`xs:integer`) matches `xs:integer+`.

An implementation supporting static typing will infer that $a has the static type `xs:integer+`, that $a[$i] has static type `xs:integer` and that the final result has static type `xs:integer`. An implementation not supporting static typing will let the program compile and raise an error at runtime.

The dynamic type can also be a complex type. With the documents presented in Figure 6.2, the expression

```
(validate type security {doc("security.xml")/Security})/Symbol
```

has the dynamic type `xs:string` (simple type).

whereas the result of the following expression:

```
validate type security { doc("security.xml")/Security }
```

has the dynamic type `security` (complex type).

The static type of a variable may differ from the dynamic type of the bound item(s), i.e., an item whose dynamic type is a subtype of the static type will successfully match, as shown in the following example (where the file stock.xml contains a document valid against the complex type `stock`).

```
let $security as element(*, security)
    := validate type stock { doc("stock.xml")/Stock }
```

6.2.8 Polymorphism in XQuery

XQuery allows subtype polymorphism. This works for subtypes derived by extension, but also for subtypes derived by restriction.

For example, considering the function signature:

```
declare function
local:get-price($security as element(*, security)) as xs:decimal
```

the parameter $security can have the dynamic type **security**, but an element with the dynamic type **stock**, derived from **security** by extension, can also be passed as a parameter.

XQuery does not support however function overloading.

6.2.9 Using polymorphism in the portfolio example

Figure 6.10 shows how the fund complex type can be redefined using the new **security** supertype.

Using subtype polymorphism, the function **get-fund-price** can be split and the type-switch can be factored out in a separate function (Figure 6.11).

(a)

```
<xs:complexType name="fund">
  <xs:complexContent>
    <xs:extension base="security">
      <xs:sequence>
        <xs:element name="Manager" type="xs:string"/>
        <xs:element name="Underlyings"
                    type="weighted-security-list"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

(b)

```
<xs:complexType name="weighted-security-list">
  <xs:sequence>
    <xs:element
      name="Underlying" type="xs:string"
      minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Weight" type="xs:decimal"/>
          <xs:element name="Security" type="security"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element/>
  </xs:sequence>
</xs:complexType>
```

Figure 6.10: *Redefinition of the fund schema using subtyping*

```
declare function
get-fund-price($fund as element(*, fund)) as xs:decimal {
  sum(
    for $underlying in $fund/Underlyings/Underlying
    let $weight as xs:decimal := $underlying/data(Weight)
    let $security as element(*, security) :=
      $underlying/*[2]
    return
      $weight * get-security-price($security)
  )
};
declare function
get-security-price($security as element(*, security))
as xs:decimal{
  typeswitch($security)
    case element(*, stock)
    return
      get-stock-price($security)
    case element(*, fund)
    return
      get-fund-price($security)
    default return ()
};
```

Figure 6.11: *Functions for computing the price of stocks and funds*

6.3 Motivations for more object orientation in XQuery

We are now reaching some limitations of XQuery as it is. While the typeswitch has been isolated, it is not possible to emulate dynamic binding (like it would be possible in C): Although function items are allowed, they cannot be embedded in XML.

If one wants to add a new kind of a security, like a bond, the existing code cannot be reused without any modification.

Figure 6.12 shows the new schema complex type as well as the price function for a bond.

(a)

```
<xs:complexType name="bond">
  <xs:complexContent>
    <xs:extension base="security">
      <xs:sequence>
        <xs:element name="IsCorporate" type="xs:boolean"/>
        <xs:element name="Issuer" type="xs:string"/>
        <xs:element name="IssueDate" type="xs:date"/>
        <xs:element name="InitialValue" type="xs:decimal"/>
        <xs:element name="Coupon" type="xs:decimal"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

(b)

```
declare function
get-bond-price($bond as element(*, bond)) as xs:decimal {
  $stock/(
    InitialValue * math:pow(
      1 + Coupon div 100,
      years-from-duration(fn:current-date() - IssueDate)
  ))
};
```

Figure 6.12: *A schema and a function to get the listing price of a bond position.*

Adding bonds requires tampering with the function `get-security-price` as shown on Figure 6.13. A new case must be added in the typeswitch.

XQuery already possesses many properties of the object model. XML nodes have:

a state An XML node contains information in the subtree of which it is the root.

an identity An XML node has an identity of its own, it is not a value. Two XML nodes with the same content do not have necessarily the same identity. An XML node keeps its identity when it is updated.

a lifecycle An XML node can be read from a document, it can be built during program execution with a node constructor, it can be disposed of when no longer referenced, or it can be made persistent in a database or on a file system.

a location An XML node has a location in memory. In the REST paradigm, XML resources can also be accessed remotely and updated.

XML Schema provides XQuery with a full-fledged classification scheme: complex and simple types are organized in a subtype hierarchy which forms a tree. It supports subtype polymorphism, in that an XML node can be used wherever one of its supertypes is expected.

XQuery allows the definition of modules with well-defined interfaces (function signatures). With the new XQuery 3.0, functions can be made private, which allows encapsulation by hiding implementation details at the module level.

However, XML nodes do not have behavior. They are not bound to code specific to their type. They do not exchange messages. This is the limitation addressed in this chapter.

```
declare function
get-security-price($security as element(*, security))
as xs:decimal{
  typeswitch($security)
    case element(*, stock)
    return
      get-stock-price($security)
    case element(*, fund)
    return
      get-fund-price($security)
    case element(*, bond) return
      get-bond-price($security)
    default return ()
};
```

Figure 6.13: *The updated security price getter function*

6.4 A Human-readable format for XML Schema

6.4.1 Human-readable versus XML

XQuery exists in two variants: a human-readable form, and an XML representation known as XQueryX. XML Schema only exists in XML representation.

To introduce behavior in XML nodes, we hence are faced with two possibilities:

- Extend XML Schema and embed XQueryX code in type definitions (all XML),
- or provide a human-readable version of XML Schema in which XQuery code can be embedded (all human readable).

The first alternative is unsatisfactory: while writing an XML Schema in XML format can be reasonably done, XQuery developers write (human readable) XQuery code and not directly XQueryX code. Besides, if existing XQuery code is to be bound to a type, this would require converting it to XQueryX. The latter can be performed automatically, nevertheless XQueryX is not (and not intended to be) easy to read and debug by a human.

The second alternative requires reinventing the XML Schema wheel in human-readable form. However, it is possible to make a compromise and provide a small subset of XML Schema, close to classical classes in the object-oriented world (complex types with complex content) while still allowing binding XQuery code with an XML Schema written in XML form.

We hence contribute a human-readable form of a subset of XML Schema which can be used in the prolog of XQuery programs to define new complex types with complex content, which are the pendent of Java classes or C++ structs in XQuery.

6.4.2 Complex type declarations

In addition to variable and function declarations, an XQuery prolog is extended to declare complex type declarations. A complex type declaration is made of the complex type QName, possibly an extension or restriction declaration (for subtyping), and a sequence of semi-colon-terminated attribute and element declarations.

Figure 6.14 shows a human-readable equivalent form of the complex type definitions in the schemas defined in Figure 6.7.

(a)
```
declare complex type lib:security {
  element Name as xs:string;
  element Symbol as xs:string;
};
```

(b)
```
declare complex type lib:stock extends lib:security {
  element Company as xs:string;
  element Country as xs:string*;
  element Quote as xs:decimal;
};
```

Figure 6.14: *A complex type and another complex derived by extension, shown in human-readable form.*

If it is in a library module, the namespace of a complex type declaration must match the module namespace (i.e., like variables and functions).

A complex type declaration is mapped to a schema complex type declaration as shown on Figure 6.15.

6.4.3 Element and attribute declarations

Complex type declarations in our extension of XQuery contain a sequence of attributes and elements. Elements and attributes are directly mapped to schema elements and attributes. Some care is required for occurrence indicators. XML Schema allows any (integer) lower bound and any (integer or infinite) upper bound for the number of elements with a given name in a complex type sequence. The XQuery sequence type syntax only allows 0, 1 and infinity as bounds. For simplicity, the human-readable syntax uses the XQuery sequence types. Figure 6.16 shows how the mapping is done.

(a)

```
module namespace lib = "http://www.example.com";
declare complex type lib:foo {
  ...
}
declare complex type lib:bar1 extends other:type {
  ...
}
declare complex type lib:bar2 restricts other:type {
  ...
}
```

(b)

```
<xs:schema targetNamespace="http://www.example.com">
  <xs:complexType name="foo">
    <xs:sequence>
      ...
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="bar1">
    <xs:complexContent>
      <xs:extension base="other:type">
        <xs:sequence>
            ...
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  <xs:complexType name="bar2">
    <xs:complexContent>
      <xs:restriction base="other:type">
        <xs:sequence>
            ...
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Figure 6.15: *Mapping of a human-readable complex type declaration (a) to an XML Schema document (b).*

(a)

```
element foo as xs:string;
element bar as other:type+;
attribute attr as xs:integer;
```

(b)

```
<xs:element name="foo" type="xs:string"/>
<xs:element name="bar" type="other:type"
minOccurs="1" maxOccurs="unbounded"/>
<xs:attribute name="attr" type="xs:integer"/>
```

Figure 6.16: *Mapping of a human-readable complex type declaration (a) to an XML Schema document (b).*

6.5 Methods: Introducing behavior in XQuery

Now that a human-readable version of a subset of XML Schema has been defined, we can embed methods inside complex types. The static context is augmented with method signatures (name, arity and static types of the parameters, but also the name of the complex type containing the method). The dynamic context is augmented with method implementations. A method implementation maps an instance of the complex type as well as the actual parameters to its return type.

6.5.1 Method declarations

A method declaration is found at the same location as element and attribute declarations within a complex type. Its syntax, i.e., signature and body, is very similar to that of a function. Unlike a function, though, a method can be abstract, in which case case its signature appears in the static context, and its implementation in the dynamic context is present but empty.

(a)

```
declare complex type lib:security {
  element Name as xs:string;
  element Symbol as xs:string;
  method get-price() as xs:decimal abstract;
};
```

(b)

```
declare complex type lib:stock extends lib:security {
  element Company as xs:string;
  element Country as xs:string*;
  element Quote as xs:decimal;
  method get-price() as xs:decimal {
    Quote
  };
};
```

Figure 6.17: *Declaration of the method* ***get-price****.*

6.5.2 Inheritance

When a complex type is derived from another complex types, it inherits its method signatures and implementations. It may also override a supertype method by defining a method with the same name and arity.

When a complex type is defined, the signatures of its methods are added to the static context and their implementations to the dynamic context. If the complex is derived from a supertype, this (recursively) includes the inherited methods that have not been overridden.

6.5.3 Method resolution and dynamic binding

Given an instance of a complex type, it is possible to invoke a method defined for this complex type. The syntax is the same as function calls. The idea is that a matching method is used if available, otherwise a matching function is used.

The semantics of a function call is changed as follows.

During static typing analysis (if supported), a static error is thrown if there is:

1. neither a method signature in the static context
    - for the dynamic type of the current context item
    - with the same name
    - with the same arity
2. nor a function signature in the static context
    - with the same name
    - with the same arity

During dynamic evaluation,

1. if there is a matching method implementation in the dynamic context
    - for the dynamic type of the current context item
    - with the same name
    - with the same arity

then it used as described below, except if it is empty (abstract method) in which case a dynamic error is thrown.

2. if there is a matching function implementation in the dynamic context, it is used and the function call is evaluated as described in the current specification.

6.5.4 Method call evaluation

In XQuery, during a function evaluation, the context item is not set. In comparison to Java, it is equivalent to not having any *this* construct like in static methods. Since we are introducing behavior and methods in XQuery, we need to slightly adapt the semantics of function calls to define that of method calls, as follows.

A method call is evaluated like a function call, except that the body expression is evaluated with a context item set to the node item, with a context size of 1 and a context position of 1.

Figure 6.18 shows the implementation of the method `get-price` for the type `fund`. It contains a method call (`get-price`) on each underlying security. Not that the semantics we have given correspond to dynamic binding: for each of the securities, the method corresponding to its dynamic type (`stock` or `fund`) will be selected. An error will be raised if a direct instance of the supertype is found in the fund.

6.5.5 Binding method to existing XML Schemas

For behavior to be available with any complex type (even those not representable in human-readable format), it is possible to split the declaration between the layout in XML Schema and the methods and constructors in XQuery.

Figure 6.19 shows how this is done using the keywords "defined in schema".

```
declare complex type lib:fund extends lib:security {
  element Manager as xs:string;
  element Underlyings as list:weighted-security-list*;
  method get-price() as xs:decimal {
    sum(
      for $underlying in Underlyings/Underlying
      let $weight as xs:decimal := $underlying/data(Weight)
      let $security as element(*, security) :=
        $underlying/*[2]
      return
        $weight * $security/get-price()
    )
  };
};
```

Figure 6.18: *Declaration of the method* ***get-price*** *for the type fund. It itself calls a method.*

```
import schema namespace lib = "http://www.example.com/securities";

declare complex type lib:fund defined in schema {
  method get-price() as xs:decimal {
    sum(
      for $underlying in Underlyings/Underlying
      let $weight as xs:decimal := $underlying/data(Weight)
      let $security as element(*, security) :=
        $underlying/*[2]
      return
        $weight * $security/get-price()
    )
  };
};
```

Figure 6.19: *Binding a method to a complex type declared in a schema.*

6.6 Constructors for complex types

XQuery has a constructor syntax for simple types, which has the same semantics as a cast. For example,

```
xs:integer("1")
```

delivers an atomic item with the atomic type `xs:integer` and the atomic value 1. It actually casts its parameter to the target type.

We contribute an extension that allows constructors for complex types.

6.6.1 Constructor declarations

Constructors can be declared together with elements, attributes and methods in a complex type. The static context is extended with constructor signatures (name of the complex type, arity, static type of the parameters). The dynamic context is extended with constructor implementations (mapped to complex types).

6.6.2 Constructor call

When a constructor is used, as shown on Figure 6.21, the implementation is retrieved from the dynamic context (same complex type, same arity), and an error is raised if it is not found. The body expression is then evaluated, with no context item set. The resulting value is validated against the complex type before being returned.

Figure 6.22 shows two constructors for the complex type `fund`. One of them builds an empty fund, the other one accepts underlyings.

```
declare complex type lib:stock extends lib:security {
  element Company as xs:string;
  element Country as xs:string*;
  element Quote as xs:decimal;
  method get-price() as xs:decimal {
    Quote
  };
  constructor (
    $name as xs:string,
    $symbol as xs:string,
    $company as xs:string,
    $countries as xs:string+,
    $quote as xs:decimal)
  {
    <Stock>
      <Name>{$name}</Name>
      <Symbol>{$symbol}</Symbol>
      <Company>{$company}</Company>
      {
        for $country in $countries
        return
          <Country>{$country}</Country>
      }
      <Quote>{$quote}</Quote>
    </Stock>
  };
};
```

Figure 6.20: *Declaration of a constructor for the complex type* ***stock****.*

```
stock(
  "XML Corp Stock",
  "XMLC",
  "XML Corporation",
  ("U.S.A.", "Switzerland"),
  3.14)
```

Figure 6.21: *Use of a constructor for the complex type* ***stock****.*

```
declare complex type lib:fund extends lib:security {
  ...
  constructor (
    $name as xs:string,
    $symbol as xs:string,
    $manager as xs:string,
    $underlyings as element(*, security)+,
    $weights as typenamexs:decimal+)
  {
    <Fund>
      <Name>{$name}</Name>
      <Symbol>{$symbol}</Symbol>
      <Manager>{$manager}</Manager>
      <Underlyings>
      {
        for $security at $p in $underlyings
        return
          <Underlying>
            <Weight>{$weights[$p]}</Weight>
            {copy $s := $security
            modify rename node $s to "Security"
            return $s}
          </Underlying>
      }
      </Underlyings>
    </Fund>
  };
  constructor ($name as xs:string, $symbol as xs:string
    $manager as xs:string)
  {
    <Fund>
      <Name>{$name}</Name>
      <Symbol>{$symbol}</Symbol>
      <Manager>{$manager}</Manager>
      <Underlyings></Underlyings>
    </Fund>
  };
};
```

Figure 6.22: *Declaration of constructors for the type fund.*

6.7 Code reuse

We have introduced behavior in XQuery. XML nodes are now capable of processing messages with method calls.

A consequence of this contribution is that code can now be reused without modification. If we want to introduce a new complex type for bonds, all is needed is a complex type declarations as shown on Figure 6.23.

For example, a new fund containing a bond and a stock (stored in variables) can be created with the expression shown on Figure 6.24, without any modification in the code concerning funds.

```
declare complex type lib:bond extends lib:security {
  element IsCorporate as xs:boolean;
  element Issuer as xs:string;
  element IssueDate as xs:date;
  element InitialValue as xs:decimal;
  element Coupon as xs:decimal;
  method get-price() as xs:decimal {
    InitialValue * math:pow(
      1 + Coupon div 100,
      years-from-duration(fn:current-date() - IssueDate)
    )
  };
  constructor (
    $name as xs:string,
    $symbol as xs:string,
    $is-corporate as xs:boolean,
    $issuer as xs:string,
    $initial-value as xs:decimal,
    $coupon as xs:decimal)
  {
    <Bond>
      <Name>{$name}</Name>
      <Symbol>{$symbol}</Symbol>
      <IsCorporate>{$is-corporate}</IsCorporate>
      <Issuer>{$issuer}</Issuer>
      <IssueDate>{current-date()}</IssueDate>
      <InitialValue>{$initial-value}</InitialValue>
      <Coupon>{$coupon}</Coupon>
    </Bond>
  };
};
```

Figure 6.23: *A new complex type declaration for bonds.*

```
fund(
  "Fund F",
  "FUNDF",
  "Mr.  Smith",
  ($bond, $stock),
  (10, 20))
```

Figure 6.24: *Creation of a fund containing instances of the new complex type* **bond**.

6.8 Encapsulation

XQuery 3.0 introduced encapsulation at the module level. Annotations (`%public`, `%private`) can be placed in front of functions to prevent them being exported to other modules.

Since we introduced methods, we allow the same mechanism on them. A method annotated as `%private` may only be used inside the same complex type, and it is not inherited. A method annotated as `%protected` may only be called inside the complex type or any of its subtypes. A method annotated `%public` (this is the default) can be called anywhere in the program. If an attempt is made to call a method where it is not visible, a static error is raised.

Elements declared in human-readable complex type declarations can also be made `%public`, `%private` or `%protected`. No error is raised if trying to access non-visible elements or attributes, they will simply appear as an empty sequence during path navigation. However, they are visible when copies are made for consistency.

```
import schema namespace lib =
  "http://www.example.com/securities";

declare complex type lib:fund defined in schema {
  %private method get-underlying-prices() as xs:decimal {
    for $underlying in Underlyings/Underlying
    let $weight as xs:decimal := $underlying/data(Weight)
    let $security as element(*, security) :=
      $underlying/*[2]
    return
      $weight * $security/get-price()
  };
  method get-price() as xs:decimal {
    sum(get-underlying-prices())
  };
};
```

Figure 6.25: *An example of private method.*

6.9 Complete Grammar

This section gives the complete set of changes made against the XQuery grammar behind the explanations given in the chapter.

```
AnnotatedDecl  ::=  "declare" Annotation*
                    (VarDecl | FunctionDecl | ComplexTypeDecl)

ComplexType  ::=  "complex" "type" EQName
                  ("defined" "in" "schema"
                  | ("extends" | "restricts") EQName)?
                  ComplexTypeBody

ComplexTypeBody  ::=  "{" (ComplexTypeElement
                      | ComplexTypeAttribute
                      | ComplexTypeMethod)* "}"

ComplexTypeElement  ::=  Annotation* "element"
                         NCName TypeDeclaration ";"

ComplexTypeAttribute  ::=  Annotation* "attribute"
                           NCName TypeDeclaration ";"

ComplexTypeMethod  ::=  Annotation* "method"
                        NCName "(" ParamList?  ")"
                        ("as" SequenceType)?  FunctionBody ";"

ComplexTypeConstructor  ::=  Annotation* "constructor"
                             "(" ParamList?  ")"
                             ("as" SequenceType)?
                             FunctionBody ";"
```

6.10 Implementation

Part of the design described in this chapter was implemented as a cross-compiler to regular XQuery code. Complex type declarations made in XQuery prologs are cross-compiled to XML Schema. The rest of the code (including method calls) is cross-compiled to the current state of XQuery. Though, in this implementation, methods are bound statically and method calls are rewritten by passing the context item as an explicit parameter.

Part III

Space

Chapter 7

XQuery on the client-side

This chapter introduces XQuery as a client-side (browser) programming language, providing an alternative to JavaScript which natively supports semi-structured data such as XML and HTML. The challenge is that it must be easy to learn (simple API), and it must not require any installation for it to be accepted by users and Web developers.

7.1 Introduction

Over the years, the code producing Web pages has kept moving back and forth between the client and the server. Many applications are server-side, many others are client-side, and the borders are blurred. After a period in which almost all the code was on the server (thin clients), we are now experiencing the AJAX and Web 2.0 trend, in which a great deal of code is executed on the client again. Client-side software today means browser-embedded software, so that there is no installation. The browser is no longer just a rendering engine, it has become a programming platform, more powerful than ever.

The most popular programming language for the browser today is JavaScript. JavaScript was specifically designed to run in a Web browser and, thus, JavaScript is a good match for the development of client-side programs. In particular, JavaScript is well-suited for programming event-based user interfaces in the Web browser. All main-stream Web browsers today support the execution of JavaScript natively.

In addition to JavaScript, there are several alternatives to effect browser-embedded application programming. First, JavaScript has been extended in order to embed XPath code into a JavaScript program. XPath is useful in order to declaratively navigate the

DOM that represents the Web page. Second, the Google Web Toolkit (GWT) is a popular tool that allows cross-compilation from Java to JavaScript. GWT helps Java programmers to get started with programming the Web browser without the need to learn a new programming language. Furthermore, GWT enables the development of both server-side and client-side application code in a uniform way. Another extension provided by Google is the Gears framework which, among others, supports persistent data (i.e., database access with SQL) and threading as part of JavaScript programs. Finally, Adobe Flash and Flex are popular ways to program powerful user interfaces in the Web browser.

The purpose of this paper is to provide another alternative to program applications inside the Web browser: XQuery. The goal is to combine the advantages of the existing alternatives (e.g., JavaScript, XPath, GWT, and Gears) into a single, uniform, and powerful offering. Because it is Turing-complete, XQuery is powerful enough to implement anything that can be implemented by JavaScript or GWT. However, XQuery can also co-exist with existing technologies in the Web browser, thereby re-using existing JavaScript and Flex application code and using XQuery to implement functionality that is not well supported by the existing technologies. This also provides a graceful evolution of existing systems and to compensate for features that are not well supported by XQuery today.

XQuery has several compelling reasons why it should be used in the browser. First, XQuery is a super-set of XPath which is already heavily used inside Web browsers in order to navigate the DOM. XQuery is not only useful to navigate the DOM; it can also manipulate the DOM in a declarative way, it supports declarative access to persistent data (like SQL in Gears), it has a powerful function and operator library (e.g., for dates and times), and it supports scripting (like JavaScript). In a nutshell, XQuery was designed to process and correlate data on the Web and that is exactly what AJAX-style programs and *Web mash-ups* need to do. Section 7.3.7 gives an example application that show-cases this property. Furthermore, XQuery is carefully designed to be highly optimizable.

Second, XQuery runs on all application tiers (database, middleware, and Web browser) and is thus highly portable. For instance, all major database vendors (e.g., IBM, Microsoft, and Oracle) have implemented XQuery as part of their database product and many middleware products (e.g., BEA's workflow engine, information integration product, and enterprise service bus) support XQuery. Hence, XQuery code can be shipped between different tiers which can be exploited in order to reduce cost. Section 7.5.1 presents an example application that show-cases this advantage.

Third, XQuery is a family of standards endorsed by the W3C so that it is standardized and the whole XQuery family interoperates well with other W3C recommendations; e.g., XML, XML Schema, REST/Web Services. The XQuery family is a complete and powerful programming model.

Finally, there is a significant industry push for XQuery. A great deal of tools are developed and products are maturing. XQuery is taught in the curricula of the Computer Science programs of many universities. As a consequence, it can be expected that more and more XQuery programmers will soon appear on the job market.

Again, the goal of this work is to study XQuery as an alternative and complementary technology in order to implement client-side applications inside the Web browser. That is, the purpose is to study the potential of this technology and see, with the help of examples, how it co-exists with existing technology such as JavaScript. To this end, this chapter reports on the following contributions:

Show that XQuery is a viable option for client-side, browser-embedded applications. We present examples which show the advantages and expressive power of XQuery for this purpose.

Extend XQuery for the browser , thereby providing a syntax for event-based programming, CSS, and access to the Browser Object Model.

Run XQuery in the Web browser . An early version of XQuery in the Browser was available as plug-ins for each browser, requiring installation, which raised some concerns. Hence, we cross-compiled the *MXQuery* engine to JavaScript to avoid this chicken-and-egg problem. The JavaScript library allowing XQuery code execution is available at `http://www.xqib.org` and works on all five major modern browsers as well as on major mobile browsers as of 2011.

Give two application examples , a Web mash-up and a publishing application, which demonstrate the flexibility of XQuery to integrate horizontally and vertically into an application stack. Additional examples that demonstrate the power of XQuery as a browser language are available at `http://www.xqib.org`.

Show that building an XQuery-on-all-tiers application from scratch significantly simplifies the technology stack.

The remainder of this chapter is organized as follows: Section 7.2 describes the state of the art and gives the main features of JavaScript, XPath embedded in JavaScript, GWT, Gears, and Flex. The goal of our work is to combine these features into a more powerful

programming framework based on XQuery. Section 7.3 shows the proposed extensions for XQuery so that it becomes a candidate for Web browser-embedded programming. Section 7.4 gives the design of an XQuery/JavaScript library. Section 7.5 describes our experience in using XQIB for two example applications as well as notes on XQuery-only web applications. Section 7.6 contains conclusions and shows avenues for future research.

7.2 State of the art

Given the growing importance of the Web browser, a number of alternative techniques to develop Web browser-based applications have been developed. This section gives an overview of the most prominent languages and tools: JavaScript, XPath embedded into JavaScript, GWT, Gears, and Flex. The goal of this work is to combine the advantages of these languages and tools into an XQuery application development framework.

7.2.1 JavaScript

JavaScript was developed in 1995. The initial motivation was to validate forms at the client-side without the need to exchange data with the server (which was much slower than today). Today, JavaScript has become a popular language and several extensions have been added by vendors to make it a powerful programming tool for the Web browser. AJAX (Asynchronous JavaScript And XML) is probably the best example of how JavaScript can help build powerful client-side applications. JavaScript is a great programming language for the browser because JavaScript was specifically designed for this purpose.

An important feature of JavaScript is that it supports event-based programming which is needed for modern user interfaces. A second feature of JavaScript is that it supports DOM, which is the API used in order to navigate and manipulate Web pages in a Web browser. A third feature of JavaScript is its availability in all browsers.

XQuery naturally supports the navigation and manipulation of Web pages because XQuery supports the navigation and manipulation of XML and, thus, any kind of data stored behind a DOM API. The support for event-based programming in XQuery will be detailed in Section 7.3. As a result, XQuery is a viable candidate to replace JavaScript. As will become clear in the remainder of this paper, however, both XQuery and JavaScript can also co-exist in a single application.

7.2.2 Embedded XPath in JavaScript

Since it is quite cumbersome to navigate Web pages using DOM, Web browsers have supported XPath for a long time. In particular, it is possible to embed XPath expressions into JavaScript programs. Many browsers support this feature. The following example demonstrates this feature for Firefox:

```
var allDivs, newElement;
allDivs = document.evaluate(
  "//div[contains(., 'time')]",
  document,
  null,
  XPathResult,
  UNORDERED_NODE_SNAPSHOT_TYPE,
  null);
if (allDivs.snapshotLength > 0) {
  newElement = document.createElement('img');
  newElement.src = 'http://.../watch.gif';
  document.body.insertBefore(newElement,
    document.body.firstChild);
}
```

This JavaScript code uses XPath as part of the `document.evaluate` function. This JavaScript function is called with an XPath expression which looks for all divs containing the word “time”. If such an occurrence is found, an image with a heart (i.e., “watch.gif”) is inserted into the Web page.

Obviously, XQuery naturally supports navigation of Web pages with XPath because XPath is a sub-set of the XQuery programming language. That is, all XPath expressions can be executed by an XQuery processor.

7.2.3 Google Web Toolkit (GWT)

As mentioned in the introduction, GWT enables Java programmers to implement client-side applications which run in the Web browser. With GWT, programmers can program in Java and compile their code to an AJAX application. Hence, programmers no longer need to worry about browser incompatibilities, and a lot of JavaScript inconveniences such as errors (type mismatch, etc) are caught by the compiler at compile-time instead

of been caught at runtime. Furthermore, GWT programmers can be supported by the same IDE tools as regular Java programmers, e.g., Eclipse.

One prominent advantage of GWT is that GWT blurs the distinction between the presentation layer and the "middle-tier" of an application: in principle, GWT enables the movement of code between these two tiers in both directions based on technology trends. As described in Section 7.5, this is an important feature. Our work on using XQuery on the client facilitates the same feature because XQuery also runs in the middle-tier and now in the presentation layer. In fact, XQuery runs on all three application tiers (including the database), facilitating even more code movement.

7.2.4 Gears

Gears is an offering from Google in order to enable the development of full-fledged applications inside the Web browser. For instance, Google Apps (i.e., Google's competition to Microsoft's Office products) were developed with the help of Gears. Among the features provided by Gears are support for databases inside the browser. With the help of this feature, browser-based applications can run even if the client is not connected to the Internet. Furthermore, such a client-side database improves performance if fine-grained changes are made, e.g., to a text document.

Again, our work on enabling XQuery in Web browsers targets in exactly the same direction as Gears. XQuery can also be used to facilitate client-side database access. Google is currently stopping the use of Gears and moving towards an equivalent counterpart in HTML5. We are working on making these features available in XQIB.

7.2.5 Flex

Flex is a popular tool in order to develop graphical user interfaces for Web browsers. It is based on a programming language called ActiveScript which is similar to JavaScript. In order to run ActiveScript, a Web browser must install a Flex plug-in. Like JavaScript, ActiveScript and XQuery can co-exist in the Web browser - in fact, all three programming languages can be embedded into a Web page. This way, it is possible to design fancy user interfaces with Flex and program more advanced application logic and Web page manipulation with, say, XQuery. Such an approach has been taken already in several projects at ETH Zurich.

7.3 XQuery in the Browser

The previous section showed that XQuery has all the ingredients needed to develop Web-based applications. It processes XML data natively, it is declarative, yet powerful enough for complex applications, and it supports remote calls via REST and Web Services. This section presents extensions to XQuery so that XQuery becomes a viable option for AJAX-style applications in the Web browser. Furthermore, it shows how XQuery can be embedded into HTML pages for execution inside the Web browser.

7.3.1 Overview

Like JavaScript, we propose to embed XQuery scripts into HTML documents with a `<script/>` tag.

XQuery expressions in the browser are executed in a browser-specific context. This context contains all the namespaces and function libraries (Section 7.3) as well as browser-specific extensions such as the predefined functions.

The following shows how a hello world looks like:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Hello World Page</title>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      b:alert("Hello, World!")
    </script>
  </head>
  <body/>
</html>
```

In the header, the script pops up a “Hello, World!” message. This code is called when the page is loaded.

The examples in the next subsection show that XQuery is as expressive as JavaScript; the main difference is that it takes much less code to process and manipulate a Web page using XQuery than using JavaScript because XQuery was designed for that pur-

pose. In the rest of this section, we will also detail browser-specific extensions to XQuery. The most critical ones are event handling and CSS handling.

7.3.2 Program structure

XQuery code can be included in HTML script tags with a MIME type application/xquery. A page may contain several of these tags, and each of these may contain either a main module or a library module. The example in the last subsection was a main module.

Below is an example with a main module and a library module. The main module imports the library module.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      module namespace mod = "http://www.example.com/module";

      declare %ann:sequential function mod:messagebox($evt, $loc)
      {
        b:alert(//div[1])
      };
    </script>
    <script type="application/xquery">
      import module namespace my = "http://www.example.com/module";

      b:addEventListener(
        b:dom()//input,
        "onclick",
        my:messagebox#2)
    </script>
  </head>
  <body>
    <div>some text</div>
    <input type="button" value="Button"/>
  </body>
```

```
</html>
```

The semantics of a library module is that it is made available in the context. Any other module on the page may import it. We also allow importing modules from an external location instead of declaring in a script tag. In this case, a location hint needs to be specified, and the same-origin policy will be followed:

```
import module namespace m = "http://www.xqib.org/module"
  at "module.xquery";
```

The semantics of a main module is that it will be evaluated as soon as the page has finished loading. The XDM instance returned by the query body is then inserted in the page after the script tag. Of course a main module may define functions (local namespace), but their scope will only be this module itself, and they cannot be imported by other modules. For each module, the XHTML namespace is set as the default element namespace. A built-in module with browser-specific functionality, the browser module, is also imported.

7.3.3 Browser Context

Each XQuery expression is evaluated in a context. By default from the XQuery recommendation, this context already comes with a rich function library [72]. Embedding XQuery into the browser, extends the context with one additional namespace (i.e., “http:// www.xqib.org/browserapi” which is bound to the prefix “b”). Furthermore, the context includes browser-specific functions. For instance, the `b:alert`() function triggers a popup. It is equivalent to the JavaScript `alert` function. This way, browser-specific components can be accessed using XQuery just as using JavaScript. The remainder of this section gives examples for some of the pre-defined functions and types in the browser-specific XQuery context.

7.3.4 Events

One of the most important features of client-side programming is event-handling. The basic principle of event-handling is that some functions (called listeners) are called when an event occurs at a certain location. For this call to occur, the listener is registered in advance.

In JavaScript, there are two ways to register for events. The first, simple way is to use e.g. the onclick, onload properties. The second way is to add listeners manually.

In XQuery, we use a special function in the browser context to register listeners for events. Those functions are high-order functions as they take functions as an argument.

Managing event listeners in XQuery

To demonstrate the mechanism, let us assume we have the following function which we would like to be called when the user clicks on a button.

```
declare %ann:sequential function
    local:myEventListener ($evt, $obj) as xs:boolean {
  let $message :=
    <message>Event occurred:{$evt/type} at {$obj}</message>;
  return b:alert(data($message))
};
```

This very simple function raises a message box with information on the event (this is the information which is passed as parameters: the event itself, and its location).

We would like to register this function as a listener for a click event at the button named “button”.

In JavaScript, such a listener is registered with:

```
document.getElementById("button").addEventListener
  ("onclick", myEventListener, false)
```

For XQuery, it is done like so:

```
b:addEventListener(
  //input[@id="button"],
  "onclick",
  local:myEventListener#2)
```

(This means that local:myEventListener shall be called whenever the user clicks at the location specified.)

Deregistering an event is done in a similar fashion:

```
b:removeEventListener(
  //input[@id="button"],
  "onclick",
```

```
  local:myEventListener#2)
```

(This means that we cancel the registration, i.e., the listener shall no longer be called if the events occurs.)

It is also possible to trigger an event:

```
b:dispatchEvent(
  //input[@id="button"],
  "onclick")
```

(This simulates a user clicking at the specified location.)

Event Node

When an event occurs, the listener is called and receives two parameters $evt and $obj. The first parameter is an XML element which contains information about the event, for example whether the alt key was pressed, which mouse button was used, etc. the same information which is available in an Event Object in the DOM [80]. The second element is the DOM node where the event occurred.

As in the DOM, there are several event properties which can be queried: $event/target, $event/type, $event/altKey, $event/button, etc., so that one can adapt the behavior of the listener:

```
declare %ann:sequential function local:listener($evt, $obj) {
  if($evt/button eq 1) then do-something()
  else () do-something-else()
};

b:addEventListener(
  //input[@name="submit"],
  "onclick",
  local:listener#2)
```

In this example, when the user clicks on the submit button, `local:listener` is called and receives an event node as well as the location node as parameters. It reads the event node and does something if the user clicked with the left button, something else if she clicked with the right button.

7.3.5 CSS

Another important aspect of browser programming is handling stylesheets. Stylesheets could be actually manipulated directly using XQuery by modifying the style attribute of an element (which is a string containing the list of style properties and their values), with an updating expression. However, the style attribute contains all style components and modifying them individually would be cumbersome.

Hence, we introduce library functions to handle styles.

For example, one modifies the style of an element with:

```
b:setStyle(
  b:dom()//table[@id="thistable"],
  "border-margin",
  "2px")
```

And one can query its style with:

```
$my-style := b:getStyle(
  b:dom()//table[@id="thistable"],
  "border-margin")
```

Classes can be queried, added and removed as well:

```
b:getClasses(b:dom()//table[@id="thistable"]) = "main"
b:addClass(
  b:dom()//table[@id="thistable"],
  "main")
b:removeClass(
  b:dom()//table[@id="thistable"],
  "secondary-table")
b:toggleClass(
  b:dom()//table[@id="thistable"],
  "selected")
```

IDs can be modified using XQuery Update, as they are atomic types.

7.3.6 REST

MXQuery implements the EXPath standard for synchronous REST calls and the corresponding functions are available in the browser.

The following page sends synchronous GET requests on each keystroke to gather suggestions:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      import module namespace http-client =
        "http://expath.org/ns/http-client";

      declare %fn:updating function
          local:showHint($evt, $obj) {
        let $str := //*[@id="txt1"]/data(@value)
        let $url :=
            concat("http://www.example.com/hints?q=", $str)
        return replace value of node //*[@id="txtHint"]
        with
          if(length($str) eq 0) then ""
          else
            http-client:send-request(
              <http-client:request href="{$url}" method="get"/>
            )[2]//hint
      };

      b:addEventListener(//*[@id="txt1"], "onkeyup",
                         local:showHint#2)
    </script>
  </head>
  <body>
    <form>
      First Name:
      <input type="text" id="txt1">
```

```
    </form>
    <p>Suggestions:  <span id="txtHint"></span></p>
  </body>
</html>
```

7.3.7 AJAX

In JavaScript, the XMLHttpRequest object allows to make asynchronous calls, at a low-level. In XQuery, we use library functions to implement asynchronous calls. The following example is the XQuery version of an AJAX example given at [3]:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      import module namespace ab = "http://example.com"
        at "http://www.example.com/hints-library";

      declare %fn:sequential function
        local:showHint($evt, $obj) {
        let $str := //*[@id="txt1"]/data(@value)
        return
        if(length($str) eq 0)
          then replace value of node //*[@id="txtHint"]
          with "";
          else
            b:async-request(
              <http-client:request href="{$url}" method="get"/>
              local:onResult#1);
      };

      declare %fn:updating function
        local:onResult($result) {
          replace value of node //*[@id="txtHint"]
```

```
          with $result[2]//int
      };
      b:addEventListener(//*[@id="txt1"],
        "onkeyup", local:showHint#2)
    </script>
  </head>
  <body>
    <form>
      First Name:
      <input type="text" id="txt1">
    </form>
    <p>Suggestions:  <span id="txtHint"></span></p>
  </body>
</html>
```

When the user types in the text box, `local:showHint` is called. If the textbox is not empty, then `local:onResult` is called asynchronously with `b:async-request`. An event is triggered when the computation of the result is completed; i.e., when the remote call returns with a result. Furthermore, the call is non-blocking; i.e., asynchronous. The user keeps control of the user interface.

7.3.8 Browser Object Model

Other aspects of JavaScript functionality (objects such as window, screen, document) are exposed as a function library. This provides support for:

Cookies `b:getCookieNames, b:getCookie, b:removeCookie, b:setCookie`

Page location `b:pageURI, b:setHref, b:getLocation`

Window `b:addWindowListener, b:removeWindowListener, b:windowInfo`

This function library will continue to expand as XQIB is developed further.

7.4 Implementation

For the implementation, the MXQuery engine, written in Java, has been cross-compiled to JavaScript code using the Google Web Toolkit. It can be used to write client-side XQuery code on all five major browsers (IE9+, Chrome, Firefox, Safari, Opera).

7.4.1 Architecture

The JavaScript library works as depicted in Figure 7.1. First, the browser receives an HTML document and parses it. It generates the DOM, renders the Web page and initializes the extension. Then the plugin obtains the XQuery scripts.

MXQuery is called with the XQuery prolog followed by the main query call. This call may register event listeners. As the plugin implements the XDM on top of the DOM, Zorba, by reading and modifying the XDM, will read and modify the DOM accordingly. The plugin then listens for IE events. When an event occurs, Zorba is called with the XQuery prolog followed by the listener call, and the plugin loops between listening for IE events and executing the corresponding listeners.

Code samples demonstrating the various features are available at [39].

7.4.2 Implementation status

MXQuery supports XQuery 1.0, the XQuery Update Facility 1.0, partly XQuery 3.0 and partly the Scripting Extension working draft. DOM navigation as well as updates to the

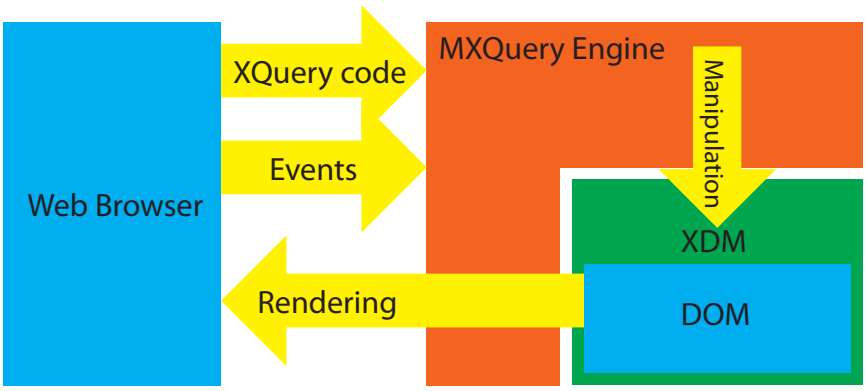

Figure 7.1: *Interaction with the XDM and the DOM*

DOM have been implemented by providing an XDM store wrapping the DOM.

Regular expressions use directly the JavaScript syntax, as GWT does not support the java.util.regex library used by MXQuery.

Major browsers support the HTML5 working draft. We mapped the following DOM5 HTML attributes to the XDM:

Namespaces we used the `node.namespaceURI` attribute

Local names we used the `node.localName` attribute (currently lowercase)

Navigation we used `node.attributes` to navigate to the attributes, `node.firstChild`, `node.nextSibling` and `node.parentNode` for element navigation.

XQIB involves a DOM which is a materialized, updateable tree, whereas MXQuery is based on immutable tokens. The generated tokens point back to the DOM so that they can follow its updates.

In the XQuery Data Model, nodes have unique identities. This is implemented with direct pointer comparisons on the Node class. Document order is computed with the DOM Level 3 `compareDocumentPosition` function when it is available, otherwise we use the Lowest Common Ancestor.

Updates are performed as follows. Deletion uses the DOM `removeFromParent` method. For insertion, the MXQuery token stream for the new node is converted to new DOM nodes (beginning with children creation).

Events are registered and unregistered using a hashmap from a (node, event name) pair to function names. When an event is triggered, all XQuery functions contained in the hashmap for the relevant key are called.

7.5 Application Scenarios

Several applications which make use of XQuery in the browser have been written in labs at ETH Zurich. Furthermore, one application has been developed in collaboration with Elsevier. This section gives three examples. The first one, based on the collaboration between ETH and Elsevier, shows how XQuery in the browser helps to migrate computation from servers to client machines; this application, thus, demonstrates the ability of XQuery to integrate different tiers (i.e., vertical integration). The second application is a typical Web mash-up, effected in the browser. It shows the ability of XQuery

to co-exist with JavaScript in the browser and the ability of XQuery to integrate different services horizontally. The third example shows how XQuery-only applications simplify the technology stack.

7.5.1 Code Mobility

The publishing industry has a long tradition of working with markup languages such as SGML and XML. Hence, the publishing industry has been using XQuery for a while in order to manage their XML data in the database and in the middle-tier. Consequently, the project Reference 2.0 was implemented from the beginning using XQuery in the middle-tier. With Reference 2.0, the user can browse through journals, volumes, issues and articles and then, for a given article, study the references (statistics, years...).

In the original architecture, an XQuery application server produces Web pages with data from an XML database available via REST calls. The XML database contains the article hierarchy and their contents. The generated Web pages contain client-side JavaScript code which reacts on events and allows interactivity.

In order to off-load Elsevier's servers, we are using our XQuery plug-in and migrating the XQuery code from the server to the client. As a result, the served Web pages contain both JavaScript and XQuery (see Figure 7.2). The JavaScript code improves the usability of the interface. The XQuery code takes care of the page layout and directly queries the XML from the database using REST calls. In order to improve performance, whole XML documents can be cached in the browser so that most user requests can be processed without any interaction with the Elsevier server.

The beauty of this migration project is that the JavaScript code is completely unchanged and that the XQuery code which runs in the client is almost the same as the XQuery code that previously ran in the server. In other words, this migration can be done with very little effort and shows how XQuery in the browser can help to make the software architecture of a Web-based application more flexible. In this particular case, reducing cost by off-loading servers was the main motivation for this project. Likewise, other business factors or trends could motivate to move more code back from the client to the server.

In more technical detail, we are carrying out the following changes in order to migrate the existing Reference 2.0 application from server-side to client-side software. The XQuery modules defined in the Reference 2.0 application code are directly published and made available and visible to clients. Furthermore, the REST interface of the Reference 2.0 modules has to be adjusted so that they serve whole documents

rather than individual queries to documents (to better enable caching). Furthermore, some specifics of the XQuery product used as an XMLDB (i.e., MarkLogic) have to be adjusted. Finally and most importantly, XQuery code is injected into the Web pages served by the Reference 2.0 server. For each dynamically generated Web page, the server-side expressions are moved to the script tag (which is created if it does not exist) as follows: the prolog is directly inserted into the script tag, whereas the contents enclosed in the outermost element constructors (formerly computed by the server) are put in main module script tags at the corresponding positions.

It would also be possible to translate the JavaScript code to XQuery, but it would need more time and it is not necessary for this project.

7.5.2 XQuery and JavaScript

There are several ways of building Web pages integrating both XQuery code and JavaScript, allowing for a smooth transition from traditional applications to all-layers XML-based Web applications.

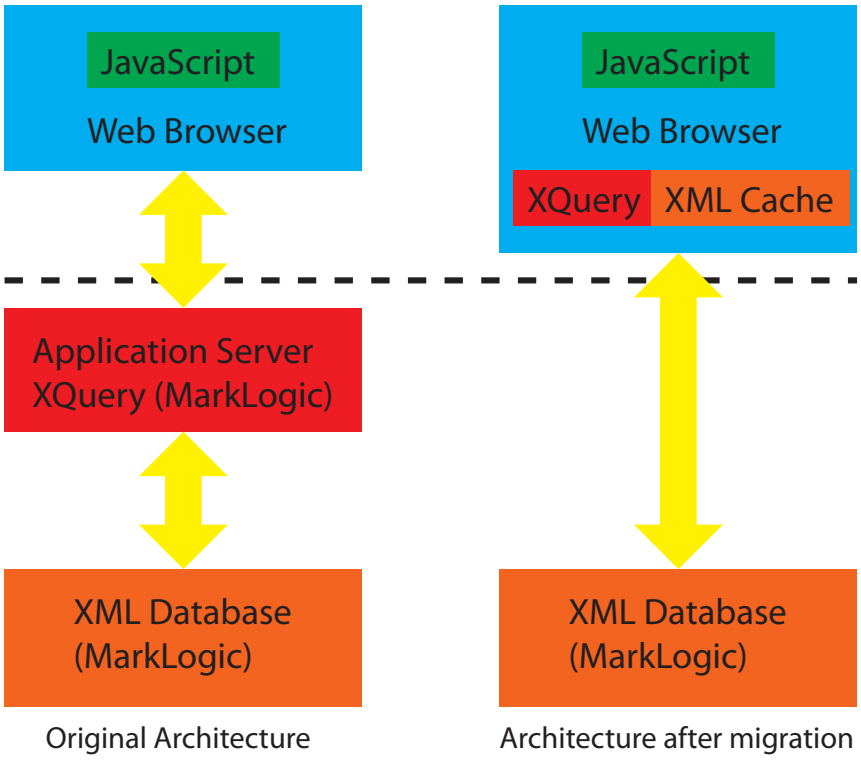

Figure 7.2: *Elsevier Reference 2.0: Server-to-client Migration*

Maps and Weather Mash-up

The first approach is to use the Web page as an interface between the languages, with event-based communication.

In a project, a Mash-up between Google Maps and a Weather service (actually, a selection of different weather services is used, depending on the used language and the region of interest) was developed. This application also integrates Webcams for the locations of interest. This application gives weather information for every location shown in Google Maps. Likewise, the service displays maps for all locations given and queried as part of one of the weather services.

In this project, the Web application uses both JavaScript and XQuery. JavaScript is used to run Google Maps and communicate with the Google server using AJAX. XQuery in the browser is used in order to initiate REST calls to the diverse weather services and integrate the results returned by the weather services. Furthermore, XQuery is used in order to search for Web cams at the location of interest. The interesting aspect of this co-existence of JavaScript and XQuery in the browser is that, in this application, code written in both languages listens to the same events. For instance, if the search button in Google Maps is clicked, then naturally, Google is called in order to serve the right map. At the same time, the XQuery code also handles this event (i.e., the click on the search button), extracts the location from the search box and initiates REST calls to weather services and Web cams. Also, code in both languages is used in order to query and manipulate the Web page (in the browser's DOM), as shown in Figure 7.3. In this figure, the Web page serves like a database and both JavaScript and XQuery code can be used in order to access and update that "database". The browser

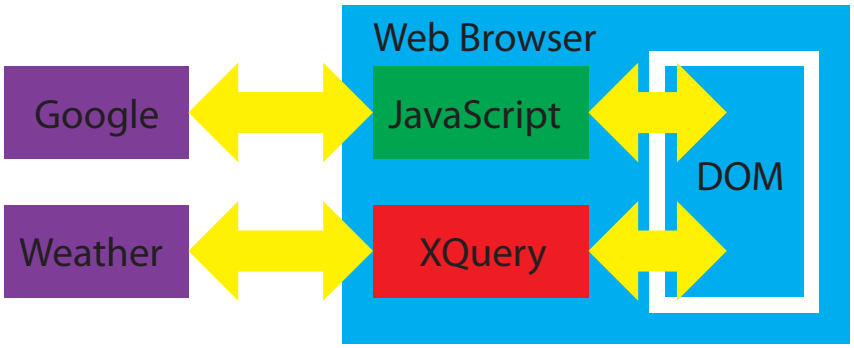

Figure 7.3: *Google Maps-Weather Mash-up: how XQuery and JavaScript can interact through the DOM.*

determines the order in which events are processed (by JavaScript and XQuery functions) in the same way as the browser serializes the order of event processing in the case that only JavaScript is used (and several different JavaScript functions are used to handle the same event).

Interlanguage calls

The second approach is to allow JavaScript to call XQuery functions, and vice versa (Fig. 7.4). This is done with a mapping between JavaScript datatypes and XQuery datatypes. For example, an XML node is exposed in JavaScript with the Node interface.

XQuery can invoke JavaScript functions with `b:js-call` as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="text/javascript">
      greet = function (name)
        { return 'Hello, "' + name+ '"';};
    </script>
    <script type="application/xquery">
      b:alert(b:js-call("window.greet", string(b:dom()//div))
```

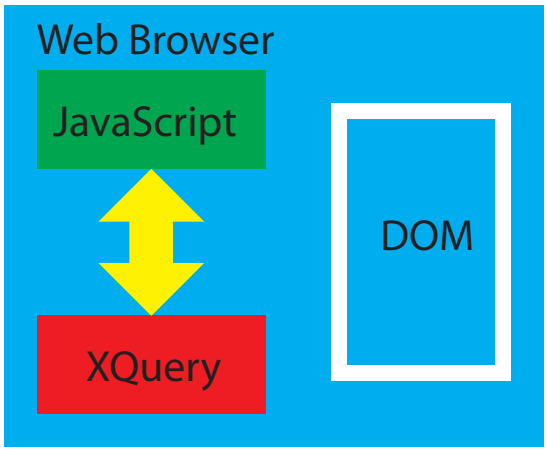

Figure 7.4: *How XQuery and JavaScript can interact directly.*

```
    </script>
  </head>
  <body>
    <div>World!</div>
  </body>
</html>
```

In the above page, a popup will appear with a “Hello, World!” message.

7.5.3 XQuery Only

The previous two applications showed how XQuery and JavaScript could co-exist in modern Web-based applications. JavaScript was mostly used because it was already there (i.e., legacy code) and it was too much effort to rewrite the JavaScript code. This section shows how XQuery-only can significantly simplify an application and result in less code, if a new application is designed from scratch.

The state of the art in Web application development is to have several languages in the same file. Here is an example of an application which simulates a shopping cart, taking products out of a database (We omitted the code to connect and disconnect). Clicking on the button next to a product adds it to the shopping cart.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript">
      function buy(e) {
        newElement = document.createElement("p");
        elementText = document.createTextNode(
          e.target.getAttribute(id));
        newElement.appendChild(elementText);
        var res = document.evaluate(
          "//div[@id='shoppingcart']",
          document,
          null,
          XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
          null);
          res.snapshotItem(0).appendChild("newElement");}
```

```
    </script>
  </head>
  <body>
    <div>Shopping cart</div>
    <div id="shoppingcart"></div>
    <% // Code establishing connection
      ResultSet results =
      statement.executeQuery("SELECT * FROM PRODUCTS");
      while (results.next()) {
        out.println("<div>");
        String prodName = results.getString(1);
        out.println(prodName);
        out.println("<input type='button' value='Buy'");
        out.println("id='"+prodName+"'");
        out.println("onclick='buy(event)'/></div>");
      }
      results.close();
    // Code closing connection %>
  </body>
</html>
```

In this example, the business logic is developed using JSP (Java nested into HTML). JavaScript is executed on the client, with embedded XPath. The server-side code even contains SQL.

If this application had been developed with XQuery only from scratch, it would look like the following XQuery expression:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="mxqueryjs/mxqueryjs.nocache.js"></script>
    <script type="application/xquery">
      declare %fn:updating function local:buy($evt, $obj) {
        insert node <p>{$obj/@id}</p>
        as first into //div[@id="shoppingcart"]
      };
```

```
      b:addEventListener(//input, "onclick", local:buy#2)
    </script>
  </head>
  <body>
    <div>Shopping cart</div>
    <div id="shoppingcart"></div>
    {
      for $p in doc("products.xml")//product
        <div>
          {$p/name}
          <input type="button" value="Buy" id="{$p/name}"/>
        </div>
    }
  </body>
</html>
```

The database is mapped to an XML document with the URI given as parameter to **doc**. A FLWOR expression inserts the products. The events are registered for all buttons with a single instruction. The entire code, client-side and server-side (even the HTML tags) is XQuery.

This shows that using XQuery for everything simplifies considerably the Web application. XQuery can be used for database access (instead of SQL), to define the application logic (instead of C# or Java) and inside the browser (instead of JavaScript). Furthermore, this example shows that an XQuery-only implementation can require fewer lines of code and avoid the technology jungle.

Another example that demonstrates how XQuery can reduce the number of lines of code is given on [39]. The multiplication table demoed on that site requires 77 lines of JavaScript code or alternatively only 29 lines of XQuery code. 28msec has also conducted experiments (on the scale of several thousands LOC) and showed XQuery to need less code than Java by a factor of up to 5.

7.6 Conclusion

The browser today is not only a rendering tool, but has become a programming environment. The most popular client-side programming language, JavaScript, is geared

towards this purpose. This paper showed that XQuery is also a viable candidate to program the Web browser. Just like JavaScript, XQuery is a full-fledged programming language with many features for processing and integrating data on the Web. In addition, XQuery has the advantage of being declarative and therefore better optimizable. In particular, XQuery can process XML data declaratively in order to query and manipulate the DOM that represents a Web page in the browser (e.g., HTML tables) or results of REST calls. Another reason to consider XQuery in the browser is to avoid the technology jungle found in many applications today: XQuery runs in all tiers (database, middleware, and Web browser) and therefore whole applications can be implemented using a single programming language, XQuery. All major (relational) database vendors already support XQuery and XQuery is a popular language in several middleware products. With the help of XQuery, it is thus possible to achieve flexible and largely simplified application architectures, thereby possibly eliminating the need for data marshalling and some layers in the application stack.

In order to make XQuery suited for the browser, XQuery had to be slightly extended. Most importantly, support for events had to be integrated in order to be able to implement reactive and asynchronous user interfaces. Other than that, XQuery was sufficient in order to build any kind of AJAX-style application. The XQuery/JavaScript library with a number of examples are available under a free open-source license at `http://www.xqib.org`.

Chapter 8

Conclusion

Traditionally, Web-application development was based on a three-layer architecture with a database back-end, business logic on the middleware accessible as a Web server, and client-side code in the browser. This involves cohabitation of many technologies, and of mappings between the different data paradigms.

Recently, solutions have appeared on the market to unify the first two layers with an semi-structured database (XML already, JSON increasingly) and an Web application server based on a language handling this data seamlessly (such as XQuery). We showed that it is possible to push this paradigm to the last layer, the browser, all the way down to the HTML UI which can itself be handled like semi-structured data as well.

We also provided a framework for introducing Version Control in the semi-structured database back-end. We introduced an operator on Pending Update Lists that allows to summarize the changes made by a program, which potentially allows making local changes and shipping them to a server for propagation to the persistent layer.

Then, we suggested several ways of extending the programming language used on all these layers, both with a more intuitive syntax for side-effecting programs, as well as with more object-orientation to provide code reuse.

We would like to finally emphasize that, even though this work has been conducted with the example of XML and XQuery, it would be potentially applicable to other declarative languages and tree-based data models (such as JSON) as well.

List of Figures

Bibliography

[1] Adobe Acrobat.com. http://acrobat.com.

[2] Adobe Flash Player. http://www.adobe.com/products/flashplayer.

[3] AJAX Suggest Example. http://www.w3schools.com/ajax/ajax_example_suggest.asp.

[4] Apache Subversion. http://subversion.apache.org.

[5] Apple iCloud. http://www.icloud.com.

[6] Apple Versions. http://www.apple.com/macosx/whats-new/auto-save.html.

[7] Bazaar Version Control System. http://bazaar.canonical.com/.

[8] Cascading Style Sheets specifications. http://www.w3.org/Style/CSS/Overview.en.html.

[9] Concurrent Versions System. http://www.nongnu.org/cvs.

[10] CSSC (GNU's replacement for SCCS). http://www.gnu.org/software/cssc.

[11] ECMAScript. http://www.ecmascript.org.

[12] eXist XML Database. http://www.exist-db.org.

[13] git. http://git-scm.com/.

[14] Google ChromeBook. http://www.google.com/chromebook.

[15] Google Docs. http://docs.google.com.

[16] Google Web Toolkit - Build AJAX apps in the Java language. http://code.google.com/webtoolkit/.

[17] HTML5 Specification (Working Draft as of 2011). http://www.w3.org/TR/html5.

[18] HyperText Transfer Protocol - HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt.

[19] Java Enterprise Edition. http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[20] jQuery. http://www.jquery.org.

[21] JSON: JavaScript Object Notation. http://www.json.org.

[22] MarkLogic XQuery Application Server. http://www.marklogic.com.

[23] Mercurial SCM. http://http://mercurial.selenic.com/.

[24] Microsoft ASP.NET. http://www.asp.net/.

[25] Microsoft Office Live. http://www.officelive.com.

[26] Microsoft Office Web Apps. http://office.microsoft.com/en-us/web-apps/.

[27] MXQuery engine. http://www.mxquery.org.

[28] Oracle Flashback. http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html.

[29] Oracle PLSQL. http://www.oracle.com/technetwork/database/features/plsql.

[30] PHP: HyperText Preprocessor. http://www.php.net/.

[31] Programmable Web. http://www.programmableweb.com.

[32] Revision Control System. http://www.gnu.org/software/rcs/rcs.html.

[33] Sedna XML Database. http://www.sedna.org.

[34] The Web of Things. http://www.webofthings.org.

[35] The Zorba XQuery Engine. http://www.zorba-xquery.com.

[36] XML Path Language (XPath) (Recommendation). http://www.w3.org/TR/xpath/.

[37] XML Schema Specification. http://www.w3.org/TR/xmlschema-0/.

[38] XML Specification (Recommendation). http://www.w3.org/TR/xml.

[39] XQIB Samples. http://www.xqib.org/samples.

[40] XQuery 3.0 Specification (Working Draft as of 2011). http://www.w3.org/TR/xquery-30.

[41] XSLT Specification (Recommendation). http://www.w3.org/TR/xslt20.

[42] *Datenbanksysteme - Eine Einführung*. Oldenburg, 1999.

[43] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A Data Model for Temporal XML Documents. In Mohamed Ibrahim, Josef Küng, and Norman Revell, editors, *Database and Expert Systems Applications*, volume 1873 of *Lecture Notes in Computer Science*, pages 334–344. Springer Berlin / Heidelberg, 2000.

[44] Sihem Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Holstege, Jim Melton, Michael Rys, and J. Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text 1.0. http://www.w3.org/TR/xquery-full-text/.

[45] Cezar Andrei, Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Markos Zaharioudakis. Extending XQuery with Collections, Indexes, and Integrity Constraints. In *XML Prague, Czech Republic*, 2010.

[46] Tim Berners-Lee. Twenty Years. WWW2009 Keynote Address.

[47] Tim Berners-Lee. Information Management: A Proposal. Technical report, CERN, May 1989.

[48] Scott Boag, Don Chamberlin, Mary Fernandez, Daniela Florescu, Jonathan Robbie, and Jérome Siméon. XQuery 1.0: An XML Query Language (Recommendation). http://www.w3.org/TR/xquery/, jan 2007.

[49] Matthias Brantner and Donald Kossmann. Sausalito: An Application Server for RESTful Services in the Cloud. In *Workshop on Database as a Service, Münster, Germany*, 2009.

[50] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.0. `http://www.w3.org/TR/xml-names//`.

[51] Federico Cavalieri, Giovanna Guerrini, and Marco Mesiti. Dynamic reasoning on XML Updates. In *EDBT Conference*, 2011.

[52] Don Chamberlin, Michael Carey, Daniela Florescu, Donald Kossmann, and Jonathan Robbie. XQueryP: Programming with XQuery. In *XML2006*, 2006.

[53] Don Chamberlin, Daniel Engovatov, Daniela Florescu, Giorgio Ghelli, Jim Melton, and Jérôme Siméon. XQuery Scripting Extension 1.0 Working Draft. `http://www.w3.org/TR/xquery-sx-10/`.

[54] Don Chamberlin, Daniela Florescu, and Jonathan Robbie. XQuery Update Facility. `http://www.w3.org/TR/xquery-update-10/`.

[55] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM.

[56] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Version Management of XML Documents. In *The World Wide Web and Databases: WebDB2000*, 2000.

[57] Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, and Donghui Zhang. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In *2nd International Conference on Web Information Systems Engineering (WISE)*, 2001.

[58] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26:64–69, January 1983.

[59] Carlo Combi, Nico Lavarini, and Barbara Oliboni. Querying Semistructured Temporal Data. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology – EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*, pages 625–636. Springer Berlin / Heidelberg, 2006.

[60] Curtis E. Dyreson. Observing Transaction-Time Semantics with TTXPath. In *WISE '01: Proceedings of the Second International Conference on Web Information Systems Engineering (WISE'01) Volume 1*, page 193, Washington, DC, USA, 2001. IEEE Computer Society.

[61] Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). `http://www.w3.org/TR/xpath-datamodel/`.

[62] Roy Thomas Fiedling. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[63] Ghislain Fourny. Pending Update List Serialization. Technical report, ETH Zurich, October 2011.

[64] Ghislain Fourny, Dana Florescu, Donald Kossmann, and Markos Zaharioudakis. A Time Machine for XML: PUL Composition. In *XML Prague*, 2010.

[65] Ghislain Fourny, Donald Kossmann, Markus Pilman, TIm Kraska, and Dana Florescu. XQuery in the Browser (Demo). In *SIGMOD Conference*, 2008.

[66] Ghislain Fourny, Donald Kossmann, Markus Pilman, TIm Kraska, Dana Florescu, and Darin McBeath. XQuery in the Browser. In *World Wide Web Conference, Madrid (Spain)*, 2009.

[67] Dengfeng Gao and Richard T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *In VLDB*, 2003.

[68] Jim Gray. Database Operating Systems: Storage and Transactions. In *SIGMOD Invited Talk*, 2006.

[69] Donald Kossmann. Building a Web Application without a Database Systems. In *Keynote for the EDBT 2008 Conference, Nantes, France*, March 2008.

[70] David Lomet, Roger Barga, Mohamed Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Transaction Time Support Inside a Database Engine. In *ICDE*, 2006.

[71] Ian Lynach. An Algebra of Patches, October 2006.

[72] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. `http://www.w3.org/TR/xpath-functions/`.

[73] Alberto O. Mendelzon, Flavio Rizzolo, and Alejandro Vaisman. Indexing Temporal XML Documents. In *proceedings of the 30th international conference on Very Large DataBases*, pages 216–227, 2004.

[74] Peter Müller. Concepts of Object-Oriented Programming. ETH Lecture, 2011.

[75] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908, New York, NY, USA, 2004. ACM.

[76] David Roundy. Darcs Advanced Revision Control System. `http://www.darcs.net/`.

[77] Vassilis J. Tsotras and Nickolas Kangelaris. The Snapshot Index: An I/O-Optimal access method for timeslice queries. *Information Systems*, 20(3):237–260, May 1995.

[78] Fusheng Wang and Carlo Zaniolo. Temporal queries and version management in XML-based document archives. *Data and Knowledge Engineering*, 65(2):304–324, May 2008.

[79] Marianne Winslett. Jim Gray Speaks Out. *Sigmod Record*, June 2008.

[80] Nicholas C. Zakas. JavaScript for Web Developers. Wrox, Wiley Publishing, Inc., 2005.

[81] Vyacheslav Zholudev and Michael Kohlhase. TNTBase: a Versioned Storage for XML. In Inc. Mulberry Technologies, editor, *Proceedings of Balisage: The Markup Conference 2009*, 2009.