The VLDB Journal (2004) 13: 294–315 / Digital Object Identifier (DOI) 10.1007/s00778-004-0137-1

The BEA streaming XQuery processor

Daniela Florescu[1], **Chris Hillery**[1], **Donald Kossmann**[2], **Paul Lucas**[1], **Fabio Riccardi**[1], **Till Westmann**[1], **Michael J. Carey**[1], **Arvind Sundararajan**[1]

[1] BEA Systems, 2315 North First Street, San Jose, CA 95131, USA
e-mail: {daniela.florescu, paul.lucas, fabio.riccardi, till.westmann, michael.carey, arvind.sundararajan}@bea.com
[2] University of Heidelberg, Institut für Informatik, 69120 Heidelberg, Germany
e-mail: kossmann@informatik.uni-heidelberg.de

Edited by J.-C. Freytag. Received: January 12, 2004 / Accepted: March 29, 2004
Published online: August 12, 2004 –

Abstract. This paper describes the design, implementation, and performance characteristics of a commercial XQuery processing engine, the BEA streaming XQuery processor. This XQuery engine was designed to provide high performance for message-processing applications, i.e., for transforming XML data streams. The engine is a central component of the 8.1 release of BEA's WebLogic Integration (WLI) product. The BEA XQuery engine is fully compliant with the August 2002 draft of the W3C XML Query Language specification and we are currently porting it to the latest version of the XQuery language (July 2004). A goal of this paper is to describe how a fully compliant yet efficient XQuery engine has been built from a few relatively simple components and well-understood technologies.

1 Introduction

After several years of development in the W3C, XQuery is starting to gain significant traction as a language for querying and transforming XML data. Though the W3C XQuery specification has not yet attained Recommendation status, it is already beginning to appear in a variety of products. Examples to date include XML database systems, XML document repositories, and XML-based data integration offerings. In addition, of course, XPath – of which XQuery is a superset – is used in various products including Web browsers. In this paper, we focus on a new commercial incarnation of the XQuery language in an XML-centric enterprise application integration system. In particular, we provide a detailed overview of a new XQuery processing engine that was designed specifically to meet the requirements of application integration.

The XQuery language, in the tradition of prior query languages such as SQL and OQL, is a closed, declarative, and strongly typed language. In contrast to these traditional query languages, however, XQuery was designed from the start for use in querying both structured data (e.g., purchase orders) as well as unstructured data (e.g., Web pages). XQuery is a powerful query language: it has native support for handling over 40 built-in data types, powerful constructs for bulk data processing (i.e., for expressing joins, aggregation, and so on), support for text manipulation, and a notion of document ordering that provides a foundation for a variety of interesting document-oriented queries [3]. For added power as well as extensibility, support is provided for the definition and use of XQuery functions. The language is compatible with other W3C standards (e.g., XML Namespaces and XML Schema). Finally, to make the language user-friendly, particularly for prior XPath users, many XQuery expressions provide implicit existential quantification, schema validation, and/or type-casting in order to relieve programmers from always having to invoke these operations explicitly in their queries.

Because of the wide range of applications for which XQuery is intended, coupled with its powerful semantics and type system, something of a myth has emerged that the full XQuery language is going to be very difficult to implement and that it may be almost impossible to achieve both performance and scalability when implementing the language. Indeed, most existing XQuery implementations have tackled only a subset of XQuery and have made a number of simplifying assumptions. One of the key goals of this paper is to help dispel this myth by demonstrating that it is indeed possible to implement the entire XQuery language specification, types and all, in a manner that is performant for a set of target applications. To this end, this paper covers the design, implementation, and performance characteristics of the BEA streaming XQuery engine, which is now embedded in BEA's WebLogic Integration 8.1 product.[1] The engine implements the entirety of the August 2002 specification of XQuery and is now being adapted to the latest version. We describe some of the unique requirements that drove the design of the engine, particularly in the area of streaming XML data handling, and we discuss the ways in which the engine's architecture was influenced by these requirements.

The remainder of this paper is organized as follows. Section 2 lists requirements that drove the design of the BEA streaming XQuery engine. Section 3 gives an overview of

[1] The BEA streaming XQuery engine was formerly known as the XQRL (pronounced "squirrel") engine because it was developed by a startup called XQRL, Inc.

BEA's WLI 8.1 product, of which the engine is a central component. Section 4 gives an overview of the architecture of the engine. Section 5 defines the internal representation of XML data as token streams. Section 6 describes the implementation of the XQuery type system in the engine. Section 7 contains details of the compiler and optimizer. Section 8 presents the runtime system. Section 9 shows the Java interface of the engine. Section 10 describes how the engine interoperates with XML data stores. Section 11 presents the results of performance experiments. Section 12 discusses related work. Finally, Sect. 13 contains conclusions.

2 Requirements

BEA is a standards-focused company. As a result, the BEA streaming XQuery engine had a major requirement from the outset to implement the entire XQuery recommendation in a compliant manner. Performance was also a major requirement for the engine; specifically, the BEA XQuery engine was designed to provide high performance for message-processing applications (i.e., for streaming XML data). The major needs for message processing applications of XQuery include: (i) an efficient internal representation of XML data, (ii) the use of streaming execution (i.e., pipelining) to the largest extent possible, and (iii) the efficient implementation of transformations that involve the use of many node constructors. Basically, the target XML message-processing applications need to quickly reshape XML message payloads as they pass through the system, requiring a small memory footprint, the ability to handle a wide range of message sizes, and efficiency both for locating data within a message as well as generating new and/or transformed message content.

While XQuery completeness and excellent message transformation performance were the top two design goals, there were a number of additional requirements that influenced the XQuery engine's design and implementation as well:

- Limited resources: An initial version of the engine had to be developed by a team of six engineers in about 6 months. This level of productivity only seemed achievable using Java (vs. C or C++) as the programming language for the implementation.
- Integration into BEA products: The engine was designed to be an embedded component within a set of BEA products (in particular WLI 8.1). This definitely mandated the use of Java, and it also required the development of a powerful Java-to-XQuery interface (referred to as the XDBC interface in Sect. 9).
- Usability with other components: The engine was designed to be usable with third-party XML parsers, schema validators, persistent XML stores, etc.
- Deployment environment: The engine must operate properly in a clustered environment and on multiprocessor machines.
- Evolvability: Since the XQuery specification was unstable during the time when the BEA XQuery engine was initially being developed, it had to be feasible (and affordable) to evolve the engine over time in response to changes in the XQuery language specification.

3 XQuery in WebLogic Integration 8.1

The BEA WebLogic Platform 8.1 is an integrated product suite built around WebLogic Server, a leading high-end J2EE application server. The suite includes products for application and process integration (WebLogic Integration), portal development (WebLogic Portal), and data integration (Liquid Data for WebLogic) as well as an integrated development environment (WebLogic Workshop) that simplifies J2EE application development and serves as the design center for all of the other Platform components.

As mentioned earlier, the target application that really drove the design of the BEA streaming XQuery engine is WebLogic Integration (WLI), BEA's enterprise application integration product. WLI is the portion of the BEA Platform that provides tools to enable companies to rapidly develop and deploy integration-based applications that communicate with business partners, automate enterprise business processes, orchestrate existing Web services and packaged and/or legacy applications, and receive, transform, and send bits of data from/to applications throughout an enterprise. WLI 8.1 is a major new release of WLI that focuses heavily on Web services and on XML-based data handling and manipulation [2]. As such, the XQuery language plays a central internal role in WLI 8.1. XQuery is used for specifying data transformations on messages and on process variables, i.e., for transforming data such as purchase orders as they flow through the system. XQuery expressions can also be used to specify the data-driven flow logic (i.e., the looping and branching) of WLI business processes.

One of the main features required of an application integration platform is strong support for data transformations – both at design time and at runtime. This is the primary role of the XQuery engine in WLI 8.1, and BEA is making a significant bet on XQuery being the right technology for this task. To provide a simple design-time experience, WLI provides a built-in mapping tool that enables integration developers to create XQuery-based data transformations without coding (i.e., without having to work with the syntax of XQuery). Transformation developers work in terms of a graphical "map view" that shows the schemas of a transformation's XML inputs on the left-hand side and the schema of the desired XML output on the right-hand side of the view. Data transformations are specified graphically by drawing lines between the input and output schemas, adding functions to the lines when computations are required, and so on.

Based on the graphical "map view", WLI 8.1 autogenerates a corresponding XQuery expression. The resulting query can then be viewed and optionally source-edited. (The WLI 8.1 data transformation editor supports limited two-way editing of XQuery source queries.) In addition, WLI 8.1 includes a test view that generates sample data from schemas and allows transformations to be tested right in the IDE.

Single-document XML-to-XML transformations are the most common form of data transformations for WLI, but WLI 8.1 actually supports a much broader range of transformations with the data mapping tool and the BEA streaming XQuery engine. In addition to single-input transformations, whose input is one XML message, the mapper supports the graphical construction of transformations that take multiple XML arguments as input (i.e., XML joins). Moreover,

WLI supports the development of data transformations that consume and/or produce Java objects and binary data objects (i.e., not just XML arguments). In these non-XML cases, the mapper still shows the transformation's input and/or output types as trees, so the design model remains consistent across a wide range of potential data types. In the case of Java objects, WLI 8.1 infers a default XML Schema corresponding to the Java class of interest. In the case of binary data, WLI 8.1 relies on the use of another WLI component, called FormatBuilder, which allows developers to separately specify, test, and persist a set of parsing rules to convert a given binary record format into a structurally isomorphic XML Schema. In these cases, when a data transformation's input or output format is non-XML, a transformation step into or out of XML form occurs prior to the central XQuery-based transformation; for efficiency, in all cases the actual internal XML data representation is the token stream format described in Sect. 5.

The other use of XQuery in BEA WLI involves business process logic. A typical WLI 8.1 business process can include a number of XQuery expressions that define XML-based flow logic for the process. These expressions can appear in conditional nodes (decision nodes in the process) that control which branch of the flow should be processed next. They can also appear in iteration loops (loop nodes in the process) that drive the process to do something once for each piece of something else, e.g., once for each line item in a purchase order. Unlike data transformation queries, which tend to involve a number of nested for-loops and return clauses with lots of node construction, flow queries mostly fall within the simple XPath subset of the XQuery language. These queries are basically simple path expressions over process variables. This use of XQuery is tool-based as well; a special expression editor helps developers to construct XPath expressions for their branching and looping node conditions in business processes.

In addition to WLI, XQuery is used in several other components of the BEA Platform as well. One such component is XmlBeans, which is a Java binding that provides friendly yet efficient programmatic access to XML data (see http://xml.apache.org/xmlbeans/). The XmlBean APIs in the BEA Platform include two methods, *selectPath()* and *executeQuery()*, that permit developers to use XQuery expressions to declaratively navigate to a set of nodes of interest in an XmlBean or to declaratively construct a new XmlBean out of an existing XmlBean. The BEA streaming XQuery engine is used within the 8.1 XmlBeans implementation to process the *executeQuery()* requests as well as all but the simplest of *selectPath()* requests.[2]

The final BEA Platform component that utilizes the XQuery language is Liquid Data for WebLogic, a distributed, XML-based enterprise information integration product that BEA offers for use with the Platform. Liquid Data 8.1 is based on a different XQuery engine, one that does database query pushdown and distributed query processing, that was developed independently from the engine that is the focus of this paper. However, work is currently in progress to merge the features of these two XQuery engines by extending the BEA streaming XQuery engine with the requisite pushdown and distributed query-processing capabilities from the current Liquid Data engine. This merge is motivated by a desire to have a single XQuery implementation going forward – avoiding the potential for multiple XQuery dialects, subtle differences in semantics, and avoiding the engineering cost of maintaining and evolving more than one set of XQuery engine components going forward.

[2] Simple predicate-free path expressions are handled directly by the XmlBeans implementation.

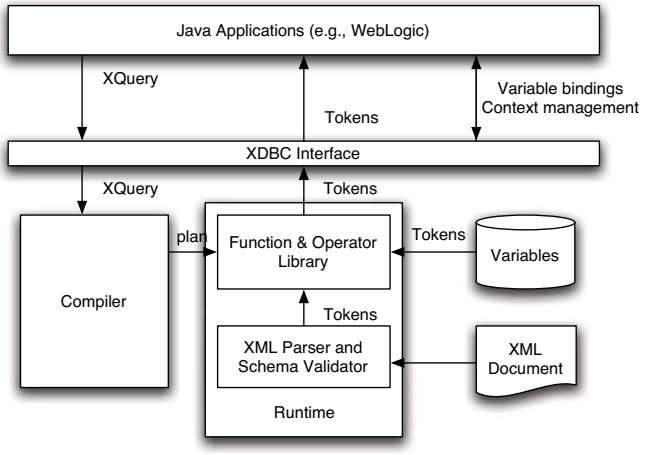

Fig. 1. Overview of the BEA streaming XQuery engine

4 XQuery engine overview

A high-level overview of the BEA streaming XQuery engine is given in Fig. 1. In terms of packaging, the BEA XQuery engine is implemented as a library so that it can be embedded within any Java application or server that wishes to use it to manipulate XML data. Java applications execute XQuery queries and consume their queries' results through an interface referred to as XDBC; the name XDBC is derived from JDBC. Each query is parsed, type-checked, and optimized by the query compiler. The compiler generates a query plan, which is a tree of operators that consume data from one another in a cascading fashion. The plan is interpreted by the runtime system, which consists of implementations of all the functions and operators of the XQuery library [18] and of the XQuery core (e.g., sorts and joins). Additionally, the runtime system contains an XML parser and an XML Schema validator; these are required when external XML data must be processed as part of a query or when XQuery's validate function is explicitly or implicitly used in a query.

In the BEA XQuery runtime, all XML data are represented as streams of tokens that are loosely equivalent to SAX events in their semantics (i.e., a token stream is essentially a depth-first linearization of an XML tree). Use of the token stream, as opposed to querying against materialized XML trees such as DOM, minimizes the runtime memory footprint of the query engine. In addition, use of the token stream enables lazy evaluation of queries. At runtime, each runtime operator consumes its input one token at a time, and any input data not required are eventually discarded. The token stream design follows the XQuery data model [12]. Finally, since the XQuery language ultimately consumes and produces XQuery data model instances, the BEA XQuery engine uses the token stream model for its XML inputs and outputs as well as for its runtime. The token stream itself is defined as a Java interface to allow for

different implementations; the default implementation uses simple Java objects. For external consumption, the engine has adaptors that developers can use to serialize the token stream as XML text or to construct a DOM or any of several other popular XML representations (e.g., SAX, StaX, or XmlBeans) from a token stream (not shown in Fig. 1).

5 XML token stream

The performance of a data-intensive language like XQuery depends very much on the data representation used to implement the language's underlying data model. Because of this, significant effort went into the design of the data representation used in the BEA XQuery engine. This section describes the relevant design goals and then describes the resulting data representation, the XML token stream, in detail. Note that the XQuery language is closed under the abstract XQuery data model [12] – meaning that the inputs, intermediate results, and output of an XQuery expression are all instances of this data model. As a consequence, a single, uniform data representation capable of faithfully capturing the abstract data model can be used throughout the system in order to simplify the runtime system.

The first design consideration for an XML data representation is the granularity at which data are to be accessed and processed. Traditional relational database engines process data at the tuple level [13]. This works well for the relational model, where tuples have fairly uniform (and bounded) sizes, but it is not appropriate for XML. A given XML node can be arbitrarily large (e.g., an entire database of 1 GB can be a single document node), and sizes can vary dramatically across nodes within a given sequence of nodes, so streaming at the level of top-level XML nodes (or items) in a sequence is not satisfactory. Given the importance of effective stream processing for our use cases, a much finer granularity of data streaming is a must.

Another important consideration for an efficient XML data representation is the avoidance of copying of data, when possible, to minimize memory requirements and query running times. As described earlier in the paper, the most important use cases for the XQuery engine in WLI 8.1 involve XML transformations. Typically, transformations involve many node constructions; creating new trees and/or copying old trees for each node construction operation would lead to unacceptable performance. A high degree of sharing and reuse of XML data instances is a must for efficiency.

The third representation-related design consideration involves the question of how much redundancy should be permitted. Some properties of data in the XQuery data model can be derived from others, but doing so might be expensive. A prominent example is the *typed value* of an element, which can be computed from the element's *children* and *type annotation*. The cost of recomputing properties must be balanced against the space overhead of representing properties redundantly. Ideally, the data representation should be flexible so that the query processor can choose which information to store redundantly based on the given workload.

Based on the above considerations, particularly the streaming emphasis, we decided to represent XML instances (sequences of atomic values and nodes [12]) conceptually as arrays; we call the sequence of array entries the *XML token stream*. In the token stream, special tokens represent the beginnings and endings of documents, elements, and attributes. Single tokens are used to represent processing instructions, comments, and text nodes. Special tokens represent simple values of XML Schema types (e.g., integer values or date values), including instances of qualified names (which are also used for name annotations and for type annotations). An instance of the token stream is generated through a preorder traversal of the XQuery data model tree structure. Element nesting is modeled by the nesting of the tokens that denote the beginnings and endings of elements and documents.

The token stream design was inspired by the popular SAX interface for XML parsing. SAX events correspond roughly to tokens in the token stream, but there are two major differences. The first major difference is that the token stream supports the full XQuery data model, whereas SAX is based on the XML Infoset model. The token stream can represent both typed and untyped data, and it can represent general sequences of items, freely intermixing nodes of all kinds and simple values. SAX is less powerful; it cannot represent typed XML data or general sequences, rendering it insufficient for the needs of an XQuery engine (or for an XQuery interface for that matter). The second major difference is that the token stream was designed to work well in the context of a pull-based API, whereas the SAX API is push-based. As discussed in Sect. 8, a pull-based interface is essential in order to permit lazy evaluation in the runtime system of the XQuery engine.

The actual token stream is relatively straightforward and can be best described using an example.

5.1 Example

Consider the following element declaration:

```
<judgement index="11">43.5</judgement>
```

The parser translates this element into the following token stream (serialized). Note that while the notation used here is textual, the actual tokens can be represented in various ways, e.g., as Java objects or encoded in a binary and very compact manner (Sect. 5.4).

```
[ELEMENT [judgement], [xdt:untyped] ]
[ATTRIBUTE [index],  [xdt:untypedAtomic] ]
[CharData "11", [xdt:untypedAtomic] ]
[END ATTRIBUTE]
[TEXT ("43.5", [xdt:untypedAtomic]) ]
[END ELEMENT]
```

This example shows that there is an `ELEMENT` token whose name is judgment and that is not qualified by a namespace in this example. The type of the judgement element is type `xdt:untyped`. (*xdt* refers to `http://www.w3.org/2003/11/xpath-datatypes`, the name space of the XPath specifications in which this specific type is defined.) The `ELEMENT` token is followed by an `ATTRIBUTE` token whose name is index and whose type is `xdt:untypedAtomic`. Following this token is the value of the attribute (11), which is represented as a `CharData` token. An `END ATTRIBUTE` token closes the attribute declaration. (Note that the value of an attribute can be a list of values.) A `TEXT` token follows to represent the content of the element,

and finally an END ELEMENT token closes the element declaration.

The data in the previous example have not been schema-validated, so the most general types appear in the type annotations (e.g., xdt:untyped for the element content). The example data can be validated against the following XML Schema snippet [9], which describes a complex type "vote" with an attribute "index" of type xsd:int and content of type xsd:float:

```
<xsd:complextype name="vote">
  <xsd:simpleContent>
    <xsd:extension base="xsd:float">
      <xsd:attribute name="index"
                     type="xsd:int"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:element name="judgement" type="vote"/>
```

The result after validation would be the following slightly different token stream:

```
[ELEMENT [judgement],
  [vote@http://www.bea.com/example]
]
[ATTRIBUTE [index], [xsd:int] ]
[int 11]
[END ATTRIBUTE]
[TEXT ("43.5", [xdt:untypedAtomic]) ]
[float 43.5]
[END ELEMENT]
```

After validation, the type annotation of the ELEMENT token is "vote" and that of the ATTRIBUTE is xsd:int. (*xsd* refers to the name space of XML Schema.) The value of the attribute is now a binary int, and the element TEXT value is now augmented with the binary float representation. It should be noted that the XQuery standard does not mandate having both the original lexical representation of elements and their typed values, but some WLI use cases require preservation of the original lexical representation of messages. Because of these use cases, both the "untyped" (lexical) and "typed" (binary) values can be kept in the token stream if desired. When preservation of the original lexical representation is not required, the token stream preserves only the binary values and the lexical representation is computed on the fly whenever needed.

While the XML token stream was mainly designed to serve as the XQuery processor's internal data representation, it turns out to be a useful XML format for application-level use as well. The token stream allows XML fragments to be managed easily, and it is easy to serialize, both for transmission on the network and storage on disk (Sect. 5.4).

5.2 Node identifiers

The XQuery data model requires each node to have a unique node identifier. Node identifiers are semantically required both for duplicate elimination and for comparisons (i.e., sorting in document order). Node identifiers can also be used internally to optimize certain operations; e.g., parent node references can be cached inside node identifiers, speeding up reverse traversals.

In the token stream, every token that represents a simple node (i.e., PI, COMMENT, and TEXT) or the beginning of a complex node (i.e., DOCUMENT, ELEMENT, ATTRIBUTE) *can* be annotated with a node identifier. However, not every query *needs* node identifiers, so node identifiers are optional in the token stream. The decision of whether node identifiers may be needed for a given query is made by the compiler (Sect. 7) using semantic information about operations (i.e., which operations require node identifiers and of which kind) together with dataflow analysis. The compiler introduces the required node identifier generation operations explicitly in the appropriate places in the expression tree. If node identifiers are not needed by a query, the token stream is particularly compact. In this case, no space is needed for node identifiers; moreover, the same Java ELEMENT token object can be (re)used in this case to represent all elements of the same kind (i.e., all elements with the same name and type).

Since not every query needs reverse traversals, different kinds of node identifiers can be used for different kinds of queries. As a consequence, the BEA XQuery engine has two implementations of node identifiers for use in processing those queries that actually do require node identifiers. The engine uses *light* node identifiers for simple queries and *heavy* node identifiers for more complex queries. Light node identifiers are based on a simple preorder numbering of the nodes; they can be used for duplicate elimination and for sorting nodes into document order. Heavy node identifiers are based on a combination of preorder numbering, postorder numbering, and parent reference materialization. As a result, heavy node identifiers can be used in queries that involve reverse traversals and special comparison functions like *precedes* and *follows*.

Node identifiers are realized as a Java interface in the BEA XQuery engine. This was done so that different users of the engine can implement their own kinds of node identifiers for their own internal purposes while enabling them to be used by the BEA engine during query processing. Any implementation must of course be semantically compliant with the XQuery data model (i.e., node identifiers must identify nodes uniquely and must be usable for identity and ordering comparisons). Support for externally defined node identifiers turns out to be important to make it possible to efficiently integrate the XQuery engine with an XML data store (Sect. 10).

5.3 Special tokens

As described so far, the token stream is a fairly straightforward serial implementation of the XQuery data model as defined by the W3C [12]. For performance reasons, the BEA engine supports the following additional kinds of tokens.

FullNode *tokens:* Streaming at the level of individual tokens is advantageous for memory footprint minimization and lazy evaluation. However, it can also lead to performance penalties in some cases since every operation must handle every token individually. To reduce this overhead, the token stream additionally supports FullNode tokens. A

FullNode token represents an entire node, i.e., it stands for a whole XML subtree. A given FullNode token contains its name, type, and, optionally, the node identifier of the node; additionally, it contains a reference to a token stream for the subtree, which itself may contain other FullNode tokens as well. FullNode tokens are particularly useful for improving the performance of queries that look only at the top-level nodes of a sequence when the underlying data have already been materialized (e.g., when the data already reside in an XmlBean or in a DOM tree). For example, FullNode tokens would be particularly helpful for a query that returns the one thousandth PurchaseOrder from a sequence of PurchaseOrders:

/PurchaseOrder[1000]

If each PurchaseOrder is represented in the engine's XML input by a single FullNode token (rather than the potentially large sequence of tokens containing all of the line items and other details of the PurchaseOrder), evaluating this query involves skipping 999 FullNode tokens rather than tens or hundreds of thousands of fine-grained tokens. In essence, FullNode tokens provide a way to efficiently support navigation to the next sibling, a very common operation in XML query processing, within the token stream. The downside of FullNode tokens is that they do imply a bit of additional overhead during parsing (if they are being generated at that time) when all of the details of a node end up being required anyway (in which case the additional overhead is not paid for by a subsequent query-processing cost savings). Choosing whether or not to use FullNode tokens in situations other than when data are coming from a materialized XML store is a decision left to the optimizer.

LazyText *tokens:* In the example shown earlier in this section, the postvalidation token stream contained both a TEXT token with a string that represents the value "43.5" and a FLOAT token with a float that represents the value 43.5. In the most general case, both are retained because both representations might be needed by a query. In addition, it is not always possible to derive the original text representation from the float representation; for instance, the original text in the XML document could have been "0043.500". (In contrast, it is always possible to derive the float value 43.5 from the text "0043.500", but it is generally worth materializing the float value to save the repeated conversion cost.) These observations led to the current design of the token stream, which by default contains both representations. In order to operate more efficiently in cases where both representations are not in fact needed, the engine supports LazyText tokens. These tokens represent text nodes but, unlike TEXT tokens, do not carry the actual text value. Instead, they contain a simple pointer to the typed (e.g., float) value of the data and will generate a (canonical) text representation on demand.

Container *tokens:* Some XQuery expressions involve packaging the results of different expressions into a *tuple* or (to use a model-neutral term) a *container*. One example is the ORDER BY clause of FLWOR expressions, which requires sorting a multivariable binding sequence. Another example is representing the results of joins from relational data sources in an XQuery-based XML data integration scenario. To represent such tuples or containers of XML data, the token stream supports an additional pair of tokens: CONTAINER, which marks the beginning of such a container, and END CONTAINER, which marks the end. Containers can be nested, like elements, and a CONTAINER token has an associated name, again just like an ELEMENT token. The main difference is that a CONTAINER token needs neither a type annotation nor a node identifier under any circumstances. Unlike an element, it is just a lightweight physical "wrapper token" that has no constraints on what it can wrap (and no logical side effects on what it is wrapping).

5.4 Token stream implementation

We now discuss several alternative ways to implement the tokens of the token stream. Just as the BEA engine permits different kinds of node identifiers (built-in and externally defined), the BEA engine also permits different implementations for the various kinds of tokens. There are built-in default implementations for each kind of token, but external clients can define their own way to represent tokens. Again, such externally defined implementations for tokens are particularly important if the BEA engine is ever to be packageable with an external XML data store (Sect. 10). Technically, each kind of token is an interface that supports a set of methods that are used at execution time in order to interpret the token. For instance, the ELEMENT token is an interface with methods to return the name, type, and node identifier (possibly *null*) of the element. Node Identifier, in turn, is an interface that contains methods to compare (equality, document order) two node identifiers. The engine currently provides two built-in implementations of these token interfaces.

Java objects: The simplest way to implement tokens is to provide a Java class for each kind of token. For instance, a Java class that represents ELEMENT tokens would have private members for the name, type, and node identifier (possibly *null*). A Java class that represents TEXT tokens would have private members for the node identifier (possibly *null*) and the text value as a char array. A Java class that represents INTEGER tokens would simply have an integer value as a private member. This implementation of tokens is used in BEA's WLI product in order to process incoming XML messages. The BEA XML parser is a SAX parser, and for each SAX event generated by the parser, a Java object (i.e., token) that represents that event is generated.

Binary: A more sophisticated and compact implementation is used if XML data are supposed to be stored on disk, or serialized in order to be processed by another instance, without the need to reparse and revalidate the data later. This representation is referred to as *binary token stream* . The binary token stream format is used in the WebLogic Integration product to persist XML messages so that they can be used by multiple activations of a business process, or even multiple business processes, at different times without having to reparse and/or revalidate the messages.

A binary token stream consists of a header (`<?binxml version"2.0"?>`) and a series of encodings of tokens. Each token is encoded by one byte that indicates the kind of the token and by encodings of the additional data carried by the token (e.g., the value of an `INTEGER` token or the name and type of an `ELEMENT` token). One particular feature of this implementation of the token stream is its use of dictionary compression for the names and types of elements and attributes. To enable dictionary-based compression, the binary token stream makes use of *pragma tokens*. Pragma tokens are instructions for clients that consume the binary token stream. Whenever a new element name (i.e., a string) is encountered, a pragma token containing the name plus a short encoding for the name is created to tell the client to insert this name/encoding pair into the dictionary so that the encoding can subsequently be used in all elements of that name. Pragma tokens may also instruct clients to flush the dictionary in order to make room for new encodings, and a special pragma token is used to signal the end of the binary token stream. This format is designed to achieve decent compression while still supporting fast processing; compression factors in the range of 2–4 are not unusual from what we have seen on customer data.

6 Type system

XQuery is a strongly typed language with a rich type system, so a key responsibility of a standard-compliant XQuery compiler is to verify the type consistency of a query with respect to its input sources and derive its result type by deriving the partial types of each subexpression using type inference rules. Type information is also important during the compiler's query optimization phase, as we will see later. The fact that the type system of XQuery consists of a mixture of named and structural typing makes this task interesting. Structural typing is limited to the types of input parameters and return values of functions and operators in XQuery, as both simple and complex XML types are named; however, during the compiler's type inference and query optimization phases, complex type operations must be performed to infer result types (type derivation and construction) and to determine whether a particular type is acceptable as input for a function or an operator (type subsumption).

The XQuery type system [6] is based on XML Schema [9]. In addition to the usual data types found in conventional programming languages, such as integer, string, etc., and user-defined structures, XQuery's type system allows new types to be created using sequences (e.g., integer followed by string), alternation (e.g., integer or string), shuffle-product (e.g., integer and string in either order), and occurrences of those (zero or one, zero or more, and one or more). Types are used by XQuery to determine whether XML data that are given as input to a certain operation are in the required form, hence the proper type. To determine this, and for additional optimization purposes, the main questions that our type system implementation has to be able to answer are:

1. Is one type a subtype of another?
2. Are two types equal?
3. Do two types intersect?

An XQuery type (e.g., `(xs:integer | xs:string)*`) is similar in spirit to a regular expression. Regular expressions and XQuery types are naturally represented using trees. Using trees to represent types allows them to be constructed easily (by a compiler parsing an XQuery expression, for example). While representing types as trees seems natural, trees do not allow the questions mentioned above to be answered easily. One reason why not is because many different trees can represent the same type.

Regular expressions, and thus XQuery types, can also be represented using an extension of finite-state automata (FSA), where an XQuery type corresponds to a language accepted by such an FSA. Simple XQuery types such as `xs:integer` and `xs:string` are symbols comprising the alphabet of the language. Unlike traditional FSAs, the transitions in these extended FSAs can be labeled with FSAs themselves, thereby providing a recursive composition of FSAs for recursive types. As a result, recursive variants of all algorithms to operate on FSAs are required and employed in the BEA streaming XQuery engine.

It can be shown that with an appropriate FSA representation (minimized deterministic FSA, mDFA) it is possible to answer the questions above by performing algebraic operations on FSAs (union, intersection, complement). The details are beyond the scope of this paper; here we aim to convey a basic sense of what the type system does and some of the challenges that had to be overcome to implement it.

To answer the question of whether two types are equal, we exploit the following observation: type T is equal to type U only if T is a subtype of U and U is also a subtype of T. Thus,"type equality" is easily mapped to "subtyping".

To deal with subtyping, we exploit the following observation: type T is a subtype of type U only if the intersection of T and the complement of U is empty. Since the FSA intersection operation is extremely expensive, we use DeMorgan's laws to compute intersection: type T intersects type U if the complement of the union of the complements of U and T is not empty.

Due to recursions, FSA algebra operations are very expensive. It can be shown that the computational complexity of these operations grows exponentially with the complexity of the FSA. To alleviate this problem, the engine uses caching – it aggressively caches all results computed by the type system. Therefore, if the same pair of types is compared several times during the compilation (and execution) of a query, expensive FSA computation is carried out only once. Caching is very effective because the total number of types involved in a given query is limited.

7 Query compilation and optimization

The first step in query processing is query compilation. Given the complexity of this component, our main goal while designing the compiler was to make it *effective* but also *extensible*, *flexible*, and *simple*. As described below, we consistently used the principles of avoiding hard-coded information and algorithms and using a declarative approach whenever possible to keep the compiler simple and extensible.

Architecturally, the XQuery compiler is composed of six major components. Three of the compiler's components are

managers: the *operation manager*, the *context manager*, and the *expression manager*. These components manage the major data structures employed by the compiler. The other three components are functional components: the *query parser*, the *query optimizer*, and the *code generator*. These components implement the three main phases of query compilation and use the services provided by the managers. We now examine each of these six components in more detail.

7.1 Operation manager

The first manager component is the *operation manager*, which holds information about the *first-order* functions and operators available for the processing of a particular query. We define the first-order functions to be those functions that require only XML data model instances as arguments in order to compute their results. In other words, first-order functions do not take artifacts such as functions, types, or schema components as arguments. Comparisons and arithmetics are good examples of first-order functions. In contrast, the XQuery ORDER BY operator is not a first-order function because it takes functions as arguments; these functions determine the values of each item to sort by.[3] Similarly, type casts and the XQuery treat-as operator are also not first-order operators because they take XQuery types as arguments. XQuery core algebra operators like conditionals and node constructors could be indeed seen as first-order operators (they only take data as input). However, we preferred not to categorize them as "normal" first-order operators due to their special role and the special treatment they require during XQuery compilation.

The operation manager maintains information about the set of in-scope first-order operators, indexed by name (the QName) and arity. The first-order operators managed by the operation manager include both the built-in XQuery operators and any external functions, functions imported from modules, and local XQuery functions. A single Java object models such an operator, and adding or removing such an operator description from the set of in-scope functions and operators can be done by simply adding this object to the Operation Manager's hash table, i.e., without recompiling the query processor code. Hence, our query processor is trivially extensible in terms of first-order operators. (Extensibility in terms of the second-order operations is not as trivial but is still relatively simple.)

The information that the operation manager maintains about each first-order operator includes the operator name and signature, semantic properties (see below), and pointers to the runtime class implementing each operator (required for code generation) and to the Java code for type derivation of polymorphic operators (required for type checking). The semantic information includes (but is not limited to) the property of preserving or introducing document order in the result, the property of preserving or creating duplicate-free results, the commutativity with the `unordered` operator, the property of the operator to create new nodes in the result, whether the operator is a map function,[4] whether the operator can raise errors at runtime, and, finally, whether or not the operator is a real function (i.e., returns the same result given the same input).[5] This semantic information is used during the optimization phase for equivalent expression rewriting. The information describing the semantic properties of the first-order operators that are part of the built-in library is loaded while bootstrapping the XQuery engine out of a declarative description.

7.2 Context manager

The second manager, the *context manager*, is used in the entire query processing lifecycle (compilation and execution). Each phase of query processing (parsing, type checking, optimization, execution) is done in a certain *context*. The context holds a variety of environmental properties. The XQuery standard defines which information needs to be maintained in a static and dynamic context of a query, including the in-scope variables (with their types and values), the schema validation context, the in-scope definitions (e.g., namespaces, functions, types, schemas, collations), and the current default processing specifications (e.g., element name namespace, function namespace, strip whitespace parameter, collations).

The BEA XQuery engine's context is an extension of the static and dynamic context as specified in the XQuery standard in the sense that it also includes all of the other environmental properties and components that allow the query processor to execute. For example, it includes the current managers that perform various tasks during compilation and execution (e.g., the *type manager*, the *schema manager*, and the *node identifier manager*): the entity resolver, which helps in resolving URIs into physical resources, the error handler, which holds the logic of responding to errors and warnings raised during query processing, the thread manager, which provides the query processor with new threads during asyncronous calls, and JDBC connection manager, which provides the query processor with JDBC connections for external SQL calls. The Context interface is one of the most fundamental APIs in the BEA XQuery processor. Since the engine code is structured as a library that can be embedded in a variety of software components (Sect. 3), we designed the Context in such a way that it can serve as the *only* link between the calling software component and the query processor library.

The same context is passed through all query-processing phases; thus, expressions (at compile time) and iterators (at runtime) only exist in a certain context. The context is logically composed of a hierarchy of local contexts. Searching for information in the context translates into searching from the local context recursively up to the parent until the desired information is found or the root is reached. The root of the hierarchy is the base context, which holds all of the XQuery engine's default parameters (e.g., all functions in the XQuery function library [18]). This information is bootstrapped from a declarative specification, as mentioned earlier. The base context is not augmentable or changeable, whereas local contexts can be augmented and/or modified using the XDBC API (Sect. 9) or by compiling XQuery prologs.

[3] Note the difference between taking a *function* as argument and taking the *result of the application of a function* as argument.

[4] A map function, as known from the functional programming language world, is a function that takes a sequence as input and whose application can be distributed over sequence concatenation.

[5] Some XQuery operators like `getCurrentDate()` do not have this property.

7.3 Expression Manager and algebra

The central component of the XQuery compiler is the *expression manager*. The expression manager holds the internal representation for all kinds of XQuery expressions (e.g., constants, variables, first-order expressions, instance of, conditionals). XQuery expressions are roughly equivalent in functionality to the algebraic query representations used by most relational query engines. Our internal representation for expressions shamelessly borrows ideas from functional programming compilation, relational query compilation, and object-oriented query compilation – all adapted to XQuery, of course.

The BEA XQuery processor has an abstract class *expression* with subclasses for each kind of expression. The simplest XQuery expressions are constants and variables. The BEA XQuery engine differentiates between five types of variables: let, for, count, external, and function parameter variables. A variable has a name (represented by a QName), a type, and potentially also a value. All of the first-order expressions (e.g., boolean operators, comparisons, arithmetics, union, intersection, and user-defined functions) share a single internal representation. A first-order expression has a pointer to the operator object in the operation manager (see above) plus its list of arguments (which are themselves expression objects). This modeling of expressions is somewhat different from traditional relational query internal representations, but it is appropriate since XQuery is an expression language and is essential for keeping the code simple and extensible.

The engine does have separate representations for the following kinds of expressions:

- *Conditionals (i.e., IfThenElse)*: These expressions hold pointers to the three expressions that correspond to their if, then, and else clauses.
- *Treat as*, *cast*, and *instanceof*: These expressions each hold a pointer to the expression that computes their argument and a pointer to their target type.
- *Node constructors*: These expressions hold a pointer to the list of expressions that produce their name (if there is one), the content of the newly created node, and an indication of the kind of node to be created.
- *Match*: These expressions model a simple filter operation that takes a sequence of nodes and a node test as input and returns those nodes from its input sequence that match the given node test. Consequently, a match expression holds a pointer to the expression that creates the input sequence, plus a node test. A node test is a triple: node kind, node name, and node type test. The name and type tests can contain wildcards.

Second-order XQuery expressions are each handled individually. Each kind of second-order expression (e.g., `FLWOR`, map, let, sortby, and quantifiers) has its own internal representation. Consider the map expression as an example. A map in our internal algebra corresponds to a simplified version of a FLWOR with a single FOR variable and no LET, WHERE, or ORDER BY clauses. Such a map is internally represented as a pair: a FOR variable and the expression that corresponds to the RETURN clause. The model for the FOR variable holds the name of the variable plus the expression that corresponds to the sequence in the FOR clause. A LET expression is very similar to a map, the only difference being that the variable is labeled as being a LET variable rather than a FOR variable.

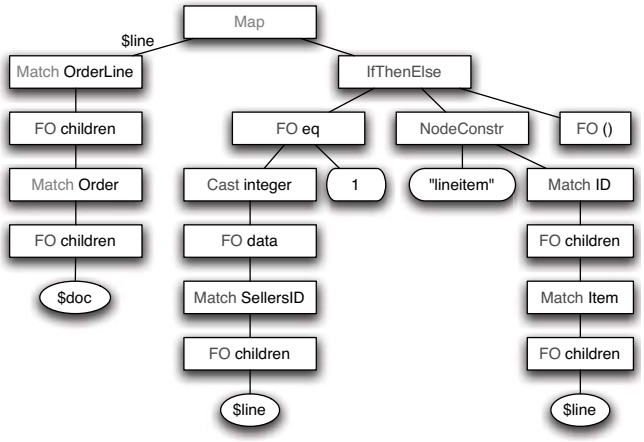

Fig. 2. Expression tree for query Q1

The translation of an XQuery query into an expression tree follows closely, but is not identical to, the recommendations of the W3C XQuery formal semantics [6]. For example, the FOR clause of a FLWOR query is translated into nested maps, each of which defines one variable; the LET clause into nested LET expressions; the WHERE clause into an If-Then-Else expression; and, the ORDER BY clause into a SortBy expression. The process is illustrated using the following example:

```
Q1:
for $line in $doc/Order/OrderLine
where xs:integer(data($line/SellersID))
  eq 1
return
  <LineItem> {$line/Item/ID} </LineItem>
```

Figure 2 shows the expression tree for this query. This query asks for line items in a purchase order document that have a particular seller. This example involves the following kinds of expressions:

- Map in order to represent the FOR clause in the query.
- First-order expressions (denoted as FO in Fig. 2) in order to implement the navigation and predicates in the query. For instance, () represents a first-order expression that generates the empty sequence for items that do not fulfill the predicate of the WHERE clause.
- Match in order to represent XPath node tests.
- Cast in order to represent type conversions.
- IfThenElse in order to implement the WHERE clause of the query.
- NodeConstructor for the RETURN clause of the query.
- Constant for the constant 1 in the predicate of the WHERE clause.
- For variable in order to represent the $line variable in the FLWOR expression.
- External variable in order to represent the $doc variable.

Our algebraic internal representation is redundant in the sense that we have models for both core and noncore expressions. For example, we have representations for `FOR` and `LET` expressions as well as for complex `FLWOR` expressions. However, we do not have an internal representation for path ex-

pressions, as they are normalized immediately during parsing. Another characteristic worth mentioning is that we do not make the distinction between a logical algebra and a physical algebra, unlike traditional relational query compilers. This distinction makes no sense for many kinds of XQuery expressions, e.g., conditionals, instance of, typeswitch, etc., or for most first-order expressions. For those operations where multiple possible physical implementations are available (e.g., node constructors and joins), the choice made by the optimizer is instead expressed via expression annotations. Those expression annotations are used during code generation to generate the right iterators to call at runtime.

The expression manager implements various functionality required for query compilation. Examples are:

- *Copy* and *replacement*: The expression manager supports a simple (recursive) copy of an expression as well as a copy through a substitution. A substitution is a mapping from variables to expressions. The copy of an expression *e* through a substitution *S* copies the original expression *e* and replaces all the occurrences of the variables in *e* with the corresponding expressions in *S*. This kind of copy is frequently used during rewriting, for example, during view unfolding. Many rewriting rules need a simple primitive for replacing a certain subexpression of an expression with an equivalent one.
- *Expression equality*: During optimization, there is frequently a need to detect if two expressions are the *same*; one important example is common subexpression factorization. The equivalence test is not a simple syntactical equality test; instead, it detects renamings of variables and returns the corresponding substitutions as part of the result of the equivalence test. However, the nature of the test is syntactical; that is, this test does not test for all kinds of *semantic* equality. Since XQuery is a Turing complete language, the general problem of detecting equivalent expressions is not decidable.
- *Variable usage analysis*: For each expression, the expression manager detects whether or not the expression uses a given variable, whether all of the variables are used inside the expression, and whether all of the free variables are used inside the expression, as well as more detailed information about the usage of a variable inside an expression (e.g., whether the variable is used as part of a loop, or is input to a recursive function, and how many occurrences of the variable the expression bears).
- *Operator usage analysis*: This functionality is similar to variable usage analysis.
- *Dataflow analysis*: Many rewriting rules need to detect the *sources* of the result of a certain expression. A source is a first-order function that computes new nodes, a node constructor operator, or an external variable. To do this analysis, we use hard-coded information about the dataflow of second-order operators and the semantic information about the first-order operators.

For each expression in the query tree, the type of its input expressions is computed bottom up and checked against the expected type for the given expression. Moreover, in the same pass, if the type-checking constraints are satisfied, the expected type of the result of the expression is computed. The type-checking and type inference algorithms follow the rules of the XQuery formal semantics [6] closely, with two major differences. First, in certain cases the type inferred by the standard XQuery rules is not precise enough for our needs, so our implementation computes a more precise type.[6] The second difference concerns the type-checking rules. The type-checking rules of the XQuery formal semantics are pessimistic, in the sense that a static typing implementation is required to raise a static type error for an operation if there is a *possibility* that the operation in question may fail at runtime. For many BEA applications, this behavior is too strict; thus, we support a more optimistic type-checking algorithm where warnings are raised in some situations, instead of static errors, and the operation is allowed to proceed at runtime. It is of course possible for an engine user to choose between the standard type-checking rules and the relaxed ones.

7.4 XQuery parser

In addition to the three managers described in the previous subsections, the XQuery compiler has three functional components: the *parser*, the *optimizer*, and the *code generator*. The parser translates an XQuery string into the corresponding expression in our internal representation (see, e.g., Fig. 2). During parsing, the current parsing context can be augmented with information from the query prolog like new namespace definitions, new function definitions, and so on. The main source of complexity in the XQuery parser comes from the necessity of keeping the language free of reserved keywords. Because of this requirement, the parser must keep multiple lexical states and perform complex state transitions during parsing. In this environment, a big challenge (one that we are still facing) is to provide users with high-quality debugging and error information. In terms of implementation, we used the ANTLR parser generator, which so far has been satisfactory for our needs.

As part of the parsing process, the XQuery expression is normalized. Some normalization transformations are required by the XQuery formal semantics [6]. Other transformations, such as the elimination of path expressions, are specific to the BEA engine and its algebra. We will revisit normalization in more detail in the next subsection.

7.5 Optimizer

The next important phase in query compilation is *query optimization*. The optimizer's task is the translation of the expression generated by the parser into an equivalent expression that is cheaper to evaluate. Traditionally, query optimization has been defined as the process of translating a *query* into an *efficient*, *equivalent query execution plan (QEP)*. In our case, both queries and execution plans share the same model: they are both expressions modeled in the same algebra. The optimizer's task is, thus, the translation of the expression generated by the parser into an equivalent expression, hopefully cheaper to evaluate.

First, we define expression equivalence for the purpose of optimization. Ideally, two expressions are *equivalent* if they

[6] An XQuery implementation is allowed to do so if the inferred type is a subtype of the standard inferred type.

have the same type and, for every possible input and in the same context, they either produce the same value as an output or they both produce the same error. Unfortunately, this definition precludes many important optimizations. As a result, the optimizer of the BEA engine relaxes this equivalence criterion. According to this relaxed definition, an expression E_1 can be rewritten into expression E_2 if the following conditions are fulfilled:

- The type of E_2 must be a subtype of the type of E_1.
- The set of free variables in E_2 must be a subset of the set of free variables in E_1.
- For every possible binding of the free variables, either E_1 or E_2 raises errors or both expressions return the same result. It is possible that both E_1 and E_2 return errors but that the errors are different. Most XQuery operations can result in errors that are not predictable at compile time. As a result, almost any operation reordering at compile time changes the query result in the event of an error.

At the heart of the query optimizer is a *library of rewriting rules*. Each rewriting rule takes an expression and returns an equivalent expression (if the rule is applicable) or null (if the rule is not applicable). Equivalence of expressions is as defined above. Applicability of a rule involves that the rewritten expression be expected to have lower costs than the original expression.

The optimizer itself is built by the successive application of such rewriting rules using heuristics. The *rule engine* is the component that carries out this process. The rule engine decides which rules are invoked and in which order and to which expression. To this end, the rule controller traverses the expression tree top to bottom and applies the local transformations dictated by each rewriting rule until saturation. The strategy of the rule engine is specified declaratively. In practice, different products that embed the XQuery engine have different strategies. For instance, the WLI product uses a different strategy than the BEA message broker and the BEA data integration product that is currently under development using this XQuery engine will certainly have yet another optimization strategy.

The set of rewriting rules that are currently in the rule library can be classified as: (a) normalization rules, whose purpose is to apply the basic normalization operations required by the correct XQuery semantics [6] (e.g., introducing the implicit atomization, boolean effective values, and casts); (b) simplification rules, whose purpose is to put the query into a simpler, "normal" form; and (c) cost-reduction rewriting rules that are supposed to translate an expression into another, less expensive expression to evaluate.

Examples of normalization rules are:

- Adding the implicit atomization operator whenever necessary;
- Adding the implicit type conversion operands (e.g., converting untyped values to the required type, adding casts and promotion operations);
- Normalizing away filter predicates that are part of path expressions and translating them into FLWRs and typeswitch;
- Translating treat-as operations into typeswitch operations
- Adding the boolean effective value.

Examples of simplification rules are:

- Inlining of XQuery user-defined nonrecursive functions,
- Replacing general generic comparisons with existential quantifiers,
- Dispatching generic comparisons and arithmetics to type-specific operators,
- Simplifying typeswitch operations by eliminating the unreachable causes,
- Translating typeswitch expressions into cascades of conditionals and instance-of expressions,
- Unfolding the quantifier expressions with multiple variables into a cascade of quantifiers with single variables,
- Eliminating the existential and universal quantifiers if applied on singleton sequences,
- Unnesting FLWOR expressions in the FOR and RETURN clauses,
- Minimization of the set of FOR variables of a FLWOR expression,
- Propagating boolean constants through the boolean operators,
- Putting predicates into conjunctive normal form.

The "cost-reduction" rewriting rules can be classified into seven categories:

1. Removing *unnecessary operations* whenever possible. A prominent example for such unnecessary operations is sort and duplicate elimination operations that are defined implicitly in a query due to the semantics of XPath expressions. Further examples are redundant self operators, concatenate operators with a single input, `FOR` expressions with a trivial `RETURN`, `FOR` expressions that iterate over a single item, computation of the effective boolean value, and unnecessary casts and treat-as operations, and validation operations are guaranteed to succeed.
2. Rewriting *constant expressions* and *common subexpressions*. For example, subexpressions that are executed as part of a loop (i.e., in a FLWR, sort or quantifier) but that do not depend on the loop variable are factored out of the loop. Subexpressions that appear multiple times in a query are also factored out and a `LET` variable is introduced.[7] Subexpressions that can be computed statically and whose results are small are computed statically and replaced by their precomputed value.
3. Enabling *streaming* whenever possible. An important example is rewriting expressions that use backward navigation into expressions that use only forward navigation whenever possible.
4. Dealing with *node identifiers*. As we mentioned earlier, a great deal of optimization and memory efficiency is derived from the fact that the query engine does not use node identifiers unless absolutely required by the particular query. Special rules in the optimizer do dataflow analysis in order to detect such a need and eventually introduce the generate node identifier operations.
5. Exploiting *schema information*. The most important rule in this class translates the expensive `descendant()` oper-

[7] A longer discussion is required with respect to the effectiveness of common subexpression factorization. In XQuery, common subexpression factorization is not always a good idea. Because of lazy evaluation and because of the streaming of intermediate results, the recomputation of a subexpression that is used multiple times can be *less* expensive than the materialization for reuse.

ator into a sequence of `children()` operators based on schema information, or introducing *topN* operators based on cardinality information obtained from schemas.
6. Carrying out *operation reordering*, if this is beneficial. Notably, the `FOR` variables in a `FLWR` are reordered if an unordered directive is present and there are no dependent joins. Similarly, the `FOR` variables of a quantifier can be freely reordered in the absence of any interdependency.
7. Transforming nested-loop (nondependent) joins and outerjoins into *hash-based* joins. In the absence of data statistics, this decision is based on heuristics and possibly results in worse query execution plans.

The BEA optimizer does not use a cost model; it only uses heuristics. There are several reasons for this. First, it is very difficult to get and maintain statistics in the Internet world. Obtaining statistics is particularly difficult for streaming XML data and message processing, for which the BEA engine was designed. Without good statistics, cost-based optimization is meaningless. Second, even in the presence of good statistics, defining an effective cost model for an XQuery engine is very difficult and probably as much work as developing the initial engine itself. Third, it seems that optimizations that require costing are less important in XQuery than, say, in SQL. For instance, `FOR` expressions (the dual to relational joins) can only be reordered in XQuery under certain circumstances. Obviously, cost-based optimization is very important for XQuery in many application scenarios, but we believe that an XQuery optimizer *needs* good heuristics to compensate for the uncertainties described above, and cost-based optimization simply has not been very important for our particular target applications.

During query rewriting, the BEA optimizer makes extensive use of both the semantic information associated with each operator (e.g., preserving document order, creating duplicates, commutativity, etc.) and the types inferred for expressions. Almost none of the cost-reduction rewriting rules could be applied in the absence of this information.

Another important task of the query optimizer is to detect (and minimize) the need for data materialization. The entire XQuery engine was designed with the chief goal of streaming data in and out of the query engine, thereby minimizing the data footprint and eliminating blocking points in the execution. Nevertheless, some queries require data materialization and/or blocking. In addition to the traditional causes such as sorting, duplicate elimination, and aggregation, the value of a variable must be materialized in three cases: when the variable is used multiple times in a query, when the variable is used inside a loop (`FOR`, sort, or quantifiers), or when the variable is an input of a recursive function. Another cause for materialization is backward navigation that cannot be transformed into forward navigation. Finally, the execution of operators like `descendant()` requires materialization under certain circumstances (Sect. 8).

7.6 Code generation

The expression produced by the query optimizer is the input to the next phase: *code generation*. The goal of this phase is to translate an expression into an executable plan. An executable plan is represented as a tree of token iterators. There is almost a one-to-one mapping between expressions and iterators, making this task quite simple: the iterator tree is built while traversing the expression tree bottom up.

Only recursive functions require special attention. Naively generating code for them results in infinite iterator trees. In order to avoid this situation, the engine delays code generation for recursive functions until they are actually executed (i.e., at runtime); the iterator tree corresponding to a new iteration is unfolded at runtime at the beginning of an iteration. We chose this way to evaluate recursive functions for two reasons. First, code generation is a relatively cheap operation (e.g., a clone of the already generated query execution plan tree). Second, the alternative would have been to have all the iterators in the query execution plan having to keep separately the state for each iteration and use the right state for each iteration. This alternative seemed to us to be more complicated to implement and more expensive to execute at runtime, and, more importantly, it would have penalized all operators, not only the ones used in recursive functions.

8 Runtime system

The task of the *runtime system* is to interpret a query execution plan, which is modeled as a tree of token iterators. The runtime system is composed of a library of iterators containing implementations for all functions and operators of the XQuery standard [18] and for all functions of the XQuery core (e.g., map) [6]. The main design goal of the runtime system is, of course, performance. In order to achieve good performance, the runtime system works in a stream-based dataflow way and avoids materialization of intermediate results whenever possible. Furthermore, the runtime system provides generic implementations of all functions and operators; at the same time, it is able to exploit certain function and operator knowledge (in particular, type-related information) obtained at compile time.

8.1 Iterator model

Like most SQL engines, the BEA streaming XQuery engine is based on an iterator model [13]. The reasons for this choice are the same as in the relational world: (a) modularity, (b) low main memory requirements, and (c) avoidance of the (CPU and I/O) costs to materialize intermediate results. Furthermore, the iterator model allows for lazy evaluation of expressions, which is particularly important for XQuery. Sometimes only a small fraction of the result of a subexpression must be computed; a frequent example is existential quantification.

In the iterator model of the BEA engine, every function and operator is implemented as an iterator that consumes zero, one, or multiple token streams produced by its input iterators and returns a single stream of tokens. As in the traditional iterator model, all iterators operate in three phases:

- *open()*: Prepare to produce results.
- *next()*: Produce next token of result stream; return *null* as end-of-stream indication.
- *close()*: Release allocated resources and do cleanup work.

In addition, our iterators provide a *peekNext()* method and a *skipNext()* method. The *peekNext()* function returns the next

token without consuming it. This function is convenient for the implementation of certain XQuery functions that need to look ahead in their inputs. The *skipNext()* function fast-forwards to the next item in the sequence being produced. This function is used by functions that only need to look at the "tip of the iceberg" (e.g., count or node test). Both *peekNext()* and *skipNext()* are implemented in a generic way for all iterators so that the need to support them does not increase the complexity of the code base or the work required to add new iterators. For *skipNext()*, however, specialized overrides are provided for certain functions for performance reasons; *skipNext()* can be made particularly fast if data are materialized or input is composed of `FullNode` tokens (Sect. 5.3), for example.

Another specific feature of the BEA iterator model is its error-handling mechanism. Every call to the *next()* method of an iterator can potentially result in a failure. Based on the semantics of XQuery, some failures can be ignored while other failures must be propagated to the application and terminate the execution of the query. In order to implement error handling, we made use of Java's exception-handling mechanism. Again, we were able to implement error handling in a generic way so that the specific XQuery error-handling rules did not have to be implemented for each iterator individually.

8.2 Example iterators

As mentioned at the beginning of this section, all functions and operators of the XQuery library [18] and core [6] are implemented as iterators. For expensive functions (e.g., node constructors and joins), several different implementations exist so that the best implementation can be chosen by the compiler depending on the characteristics of a query. In all, the runtime system contains implementations of more than 350 iterators. To provide a sense of the kinds of iterators found in the system, some examples follow.

Constant. One of the simplest iterators is the *constant* iterator, which is used to evaluate constant expressions. This iterator is used for XQuery literals such as "5" or "Feb-18-2003". For literals, the constant iterator produces a stream with a single token. The constant iterator is also used for other constant expressions such as "`<foo>boo</foo>`". In this example, the result of the element constructor is materialized at compile time and the constant iterator is used to return the materialized result at execution time.

Casts. The semantics of XQuery involve numerous implicit casts, some of which can be quite expensive. Some casts require value transformations, e.g., from strings to numerics, some involve the extraction of typed values from an element or attribute, while some require *atomization* [1], which takes a sequence as input and returns a simple value. As mentioned in Sect. 7, the compiler tries to determine the types of expressions statically as precisely as possible so that casts can be avoided or the most specific cast iterator can be used. To this end, the cast iterators in the runtime system are organized in a hierarchy. The most general cast iterator is expensive and is used when no static type can be inferred (i.e., when the static type is `xdt:untyped`). Cheaper, more specific casts are used when more information can be deduced statically (e.g., if an expression is known to have a simple type).

Materialization. Although the BEA streaming XQuery engine tries to stream data whenever possible, there are situations where the materialization of intermediate results is necessary. One important situation in which materialization is necessary is in a query that uses the results of a common subexpression several times and where (re-)computation of this subexpression is expensive. Such queries are implemented using an iterator factory. This factory takes the token stream produced by the common subexpression as input and allows the dynamic generation of iterators that consume that input. The factory consumes tokens from its input stream on demand, driven by the fastest consumer. The factory buffers the input stream and releases the buffered tokens only when the last (slowest) consumer is finished.

Item iterator. The item iterator is a very useful iterator; it is for cases where a sequence of items (according to the XQuery data model) needs to be read and processed item by item. The item iterator takes an arbitrary sequence as input. On its first use, the item iterator begins to consume its input sequence and emits the token substream for its first (complete) item. If the first item is an atomic value (e.g., an integer), the item iterator emits a single token (representing the atomic value). If the first item is instead an element node, the item iterator emits all of the tokens that represent that element – from its opening `ELEMENT` token through its matching `END ELEMENT` token. (Like any iterator, of course, the item iterator emits results one token at a time with each call to its *next()* method.) When the item iterator finishes emitting an entire item from its input sequence, it indicates end-of-stream; its caller then closes and reopens the iterator to advance to the next item in the sequence.

Node identifier generation. The XQuery data model assigns a unique ID to each node of an XML document [12]. Based on this ID, node comparisons, duplicate elimination, and sorting in document order (among other functions) are defined. In the BEA XQuery engine, IDs of nodes of incoming XML messages are generated on the fly using specialized GenerateId iterators. ID generation is an expensive operation, and the memory requirements for IDs can become prohibitive, so different types of IDs and GenerateId iterators are used depending on the requirements of a given query (Sect. 5.2). Most queries can be processed using simple, lightweight IDs that are based only on a preorder numbering of nodes; with such IDs, duplicate elimination and sorting in document order can be implemented. Some queries, however, cannot be processed using such lightweight IDs; examples are queries that involve backward traversals (e.g., the XPath parent axis) or special node comparisons. To evaluate such queries, and only for such queries, heavy IDs that are based on preorder and postorder numbering and materialization of parent/child relationships are generated. Furthermore, for certain queries, it is only necessary to generate IDs for root nodes (or for nodes up to a certain level); again, a special version of the GenerateId

iterator is used in order to improve the performance of such queries. The decision of which version of IDs and the GenerateId iterator to use is made at compile time based on the characteristics of the query. In fact, there are cases where no IDs are needed to evaluate a query; in such cases, the compiler generates no GenerateId iterator at all. Furthermore, GenerateId iterators are not needed if the input data are already parsed and annotated with node identifiers, e.g., data from a data store (Sect. 10).

XPath steps. Projections are implemented in XQuery as XPath steps, typically using the child ("/") and descendant-or-self ("//") axes. In order to exploit optimizations based on type inference at compile time, the runtime system provides different iterators for these axes. For example, there is a special version of child that stops early if it is known from the schema that only one child matches or if it is known that no more subelements are relevant as soon as one subelement of a particular type has been found. Furthermore, the runtime system implements a special algorithm to execute descendant-or-self. This algorithm starts optimistically and assumes that the data have no recursion (i.e., that an element is not nested inside another element with the same name). In this case, the algorithm is fully stream-based and no intermediate results need to be materialized. In bad cases, when recursion is detected in the data, the algorithm adapts and starts materializing data. In such cases, it behaves like a traditional, stack-based algorithm in order to compute descendants recursively.

8.3 Query execution example

We now consider an example to demonstrate in more detail how the iterators of the runtime system stream data:

```
for $x in $exp
return foo($x, $x).
```

In this example, $*exp* is assumed to be an external variable containing an instance of the XQuery data model; *foo* is a user-defined function that takes two items as input, checks whether both inputs are PurchaseOrder elements, and if so returns the PurchaseOrder element with the earlier shipping date. If one of the inputs is not a PurchaseOrder element, *foo* returns the empty sequence as its result. (How *foo* is implemented as an iterator is not described here.) Our example query simply returns all PurchaseOrder elements in $*exp* and filters out all other items.

Figure 3 shows the plan for this query as generated by the compiler (Sect. 7). The query plan is based on expressions from the logical query algebra of the compiler. The `FOR` expression of the query is represented by a *map* exression, the user-defined function is represented by expression *foo*, and all variables are represented by variable expressions ($*exp* is an *external variable expression* and the occurrences of $*x* are *for variable expressions*, as discussed in Sect. 7.3).

Figure 4 shows the corresponding iterator tree and includes information about the dataflow dependencies between the iterators. The most complex aspect of Fig. 4 is its depiction of the mechanics to materialize tokens and to ensure that *foo* is applied to each item individually. The *map iterator* peeks

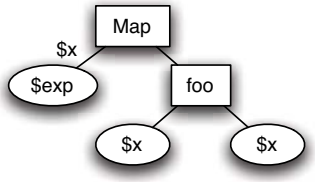

Fig. 3. Example query evaluation plan

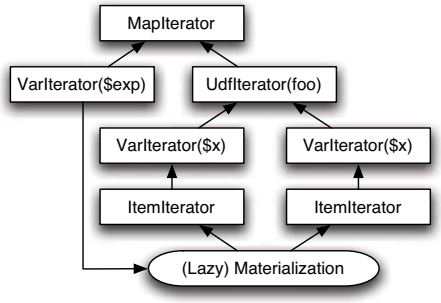

Fig. 4. Example iterator tree

into the output of the $*exp* iterator in order to coordinate the process. However, the real consumer of the tokens of the $*exp* iterator is the *materialization* point from which tokens are consumed in order to evaluate the function *foo* (i.e., the `RETURN` operator of the query).

The first point to observe is the role of the *item iterators* at runtime. The *item iterators* ensure that the $*exp* input is processed an *item* at a time. For instance, if the first item of the $*exp* variable is indeed a PurchaseOrder element (as desired by the *foo* function), then only the tokens of this PurchaseOrder element are returned by the item iterators. If the `END ELEMENT` token of that PurchaseOrder element has been processed, then the *item iterators* signal the end of the token stream (i.e., they return *null* when their *next()* method is called), and consequently, the *Udf iterator* signals the end of the token stream to the *map iterator*. At this point, only the first item (i.e., the first PurchaseOrder element) has been processed, only the first item has been consumed from the $*exp* iterator, and the rest of the sequence still needs to be processed. In order to process the second item, the *map iterator* restarts the whole process. That is, the buffer of the *materialization* is flushed (because it is no longer needed) and the states of the *Udf iterator*, *Var iterators* for variable $x, and *item iterators* are reset by calling the *close()* and *open()* methods on the *Udf iterator* (which propagates the calls to its children, which in turn propagate them down until the *materialization* point is reached). At this point, the entire iterator subtree rooted at the *Udf iterator* is ready to process the next item. In this way, each item of the $*exp* sequence is processed individually.

For this particular query, *materialization* is needed because the input tokens are consumed twice. As indicated in Fig. 4, materialization happens lazily; tokens are consumed and materialized from the $*exp* input only as they are actually needed. In the beginning, the *Udf iterator* that implements the *foo* function asks for just a single token. Using this token, it decides whether the input is a PurchaseOrder element, as requested. If it is not a PurchaseOrder element (e.g., if it is some other element, document, or atomic value), then the *Udf iterator* signals

end-of-stream to the *map iterator*. At this point only the first token of the item of the $*exp* variable has been processed and materialized. The other tokens of that item, if any, are never consumed or materialized. In order to process the next item of the $*exp* variable, which might indeed be a PurchaseOrder element that must be considered for the query result, the *map iterator* calls the *skipNext()* method of the $*exp* iterator. (It is here that an optimized *skipNext()* implementation has an opportunity to improve runtime performance by allowing the engine to avoid ever touching the skipped tokens.)

9 XDBC interface

The Java binding for the BEA XQuery engine is called XDBC, as it was designed to look and behave much like JDBC. The two APIs have similar features, including the ability to precompile statements for repeated execution, facilities for binding per-execution variables in a statement, and the ability to maintain separate execution contexts.

Because of these similarities, and because JDBC is a widely known and understood interface, we used the same basic set of classes and, where appropriate, even the same method names for XDBC. There is currently an initiative to standardize a Java interface for XML data called XQJ (JSR 225). BEA participates in this standardization activity. In its current design, XQJ has many important commonalities with XDBC. Once the XQJ standard has fully materialized, we will either make the XDBC interface upwardly compatible with it or deprecate the XDBC interface in favor of XQJ (at least for external users).

9.1 Connections and statements

The entry point into XDBC, as with JDBC, is a connection. A connection is obtained via a static method, *getConnection()*, on the DriverManager class. A connection maintains an execution context internally, keeping track of declared namespaces, types, and XQuery functions, among other things. In the current library implementation, XDBC and the XQuery engine run within the same JVM. Therefore, despite the name "connection", there is no networking involved, nor is there more than one type of driver to manage.

From a connection, applications can create two types of XQuery statement objects: Statement or PreparedStatement. Again, as with JDBC, a PreparedStatement differs from a simple Statement chiefly in the ability it provides to precompile an XQuery statement and then execute it multiple times, optionally assigning different values to the unbound variables (i.e., the parameters) of the XQuery statement for each execution.

9.2 Parameterized queries

With a PreparedStatement object, the application first performs any required variable bindings using various *setType()* methods. There are different *setType()* methods for the various XQuery primitive types, such as *setString(), setInteger(), setURI(), setDate()*, and so on. Of particular importance is *setComplex()*, which allows binding a variable to a token iterator that is potentially the result of a separate query execution. This way, chaining of XQuery queries is supported. It is also possible to specify the type of the token stream produced by an iterator bound using the *setComplex()* method.

In order to facilitate the binding and rebinding of external variables to an XQuery statement, we have extended the variable-related rules for XQuery so that it is no longer required that all variables mentioned within a statement be assigned values before use. As a result, such statements will now compile legally. However, attempting to execute a statement without binding all of its unbound external variables will result in an exception. PreparedStatement also has methods for determining the set of unbound variables discovered by query compilation, which is useful for tools (such as graphical query editors) that need to handle prepared queries in a generic fashion.

9.3 Execution of queries

Once all external variables have been bound, a PreparedStatement can be executed using *executeQuery()*. This method returns a token iterator representing the token stream resulting from the statement's execution. We provide utility classes for doing basic conversions on token iterators, such as serializing them as UNICODE or in a binary format. PreparedStatement also offers other features for specialized purposes, including a *cloneStatement()* method for creating a duplicate of a PreparedStatement that can then be rebound and reexecuted, e.g., in a different thread.

10 XML store interface

The BEA XQuery engine is a full implementation of the XQuery language. It is *not*, however, an XML database system; in particular, it has no storage functionality of its own. Instead, the engine was designed to be capable of working on top of different XML data stores and querying XML data stored in different stores in a seamless way. For this purpose, a special store interface was designed: data stores that implement this interface can be packaged together with the BEA XQuery engine in order to provide full XQuery support. There are two reasons that we adopted this essentially store-neutral approach. First, there is a wide variety of XML storage and indexing techniques in the world today, including relational [11, 21], native [10], and indexes for XPath expressions [5,14]. Second, there are many different XML stores driven by many different requirements, including XML caches, XML Beans, transactional XML stores, and legacy data stores with XML wrappers. Our aim is to make it possible to use the BEA engine for virtually all such stores. The key idea is to extend those stores so that they expose their data as a stream of tokens.

10.1 Collections

From the point of view of the BEA XQuery engine, a data store is a set of XML *collections*. Each XML collection is nothing more than an instance of the XQuery data model [12], i.e., a sequence of items. For instance, an XML document (like a

PurchaseOrder) could be a collection, or a sequence of nodes (say elements) and simple values (such as integers) could be a collection as well. In order to ensure deterministic updates, we impose the following constraint: nodes can only be part of one single collection. As a side effect of this constraint, of course, a node cannot have a parent that belongs to a different collection than its own.

Each collection is uniquely identified by a URI. As a result, the collection can be queried by the XQuery engine by using the XQuery *collection()* function, which takes a URI as parameter. In order to allow the XQuery engine to read a collection, the store prescribes the following pair of functions as part of its API:

- TokenIterator getCollection(URI source)
- TokenIterator getCollectionIds(URI source)

Both functions expose the specified collection as a token stream (Sect. 5) so that the data can be processed directly by the BEA XQuery engine. The difference between the two functions is that the second function ensures that all tokens that represent nodes in the token stream are annotated with node identifiers (Sect. 5.2), whereas the token stream returned by the store when the first function is used is not annotated with node IDs. The first function is thus called by the BEA XQuery engine when the compiler has detected that node IDs are not needed to evaluate a given query.

Exposing XML data as a token stream and implementing the interface described in Sect. 5 may seem like a high burden for implementors of data stores. However, data store implementors can typically use as tokens their own internal representation of XML data. As mentioned in Sect. 5.4, the BEA XQuery engine supports user-defined implementations of the token and node identifier interfaces. In particular, a data store that already supports a SAX interface can be extended easily in order to support the token stream interface. As a result, no significant data marshalling between the store and the XQuery engine is necessary, and very good performance can be achieved.

10.2 Indexes

In addition to collections, a data store can define indexes. An index is defined by means of two XQuery expressions: one is an expression that describes the domain that is indexed (e.g., all PurchaseOrder elements) and the second is a parameterized expression that describes some property of that domain (e.g., the shipping date). Note that this property can be computed using any XQuery expression that ranges over the indexed domain, not just on projected fields as in traditional relational databases. In order to allow the XQuery engine to make use of indexes, the store must provide these two expressions for all indexes as part of its API. This way, the compiler can detect which indexes are useful for a given query and rewrite queries accordingly. Rewriting XQuery queries in order to make use of indexes is a fairly straightforward application of answering queries using views techniques [16].

In order to use the indexes of a store at query execution time, the store must provide functions that probe the index. Just as in any ordinary database system, there should be several ways to probe the index, e.g., full index scan, range scan, or lookup of a certain key (e.g., a specific shipping date). In each case, probing an index results in the generation of a token stream, e.g., a token stream that represents all matching purchase orders.

In the current design, the BEA XQuery engine has several restrictions on the kind of indexes it can use. First, the domain that is indexed must be a sequence of nodes. Second, the property of that domain that is indexed must be an atomic value and the values must be homogeneous (i.e., they all must be instances of the same XML Schema simple type). These restrictions simplify significantly the integration of indexes at runtime. It is possible to relax these conditions, but at the moment, it seems that the most important use cases are covered despite these restrictions. For instance, any kind of indexing that is commonly carried out on relational data meets these restrictions.

10.3 Updates

To date, there is no standard to define updates on XML data. There are only proposals in the research community (e.g., [22]), in industry (e.g., all vendors of XML databases), and an internal draft from the W3C XQuery working group [4]. Nevertheless, applications can already implement their own update language using the BEA XQuery processor and an XML data store.

As an example, consider the following update statement using the syntax proposed in [4]:

```
INSERT <author>Florescu</author>
BEFORE document("www.dblp.org")//
    article[title = "The BEA XQuery
                     Processor"]/
    author[. = "Hillery"]
```

This statement adds a new author element to an article with the title “The BEA XQuery Processor” in the DBLP database. The new element is inserted just before the author element for “Hillery”. Obviously, the insertion of new data in the store must be carried out by the data store. However, the BEA XQuery engine can be used to evaluate the expression that specifies the data to be inserted (i.e., `<author>Florescu</author>` in the example) as well as the expression that describes the update target (i.e., the expression that specifies the element for “Hillery” in the `BEFORE` clause). Both of these expressions can be arbitrarily complex.

In order to evaluate `<author>Florescu</author>`, the XML store must provide a *token factory*. In this particular example, this expression involves generating an `ELEMENT` token, a `TEXT` token, and an `END ELEMENT` token. All these tokens must be generated by the XML store using its particular implementation for representing XML data. The role of the XQuery engine is to drive the generation of new tokens (e.g., deciding which tokens to generate, setting types, etc.) as part of its implementation of node constructors.

The second expression, which specifies the target of insertion, can be evaluated just like any other XQuery expression. The BEA XQuery engine implies no restrictions and has no specific requirements. In order to be able to implement such an insert statement, however, it is important that the XML store

be able to identify the target for insertion based on the query result returned by the XQuery engine. The simplest way to do this is to annotate the query result with node identifiers; in this example, the `ELEMENT` token for the author element for "Hillery" needs to be annotated with the right node identifier so that the data store knows where to place the new data into the collection. Although the example above is very specific to the semantics of the particular *insert* statement of [4] and to the BEA XQuery engine, the approach is very general. The same principles can be applied in order to implement other update statements such as the delete, replace, and complex update statements defined in [4]. In all cases, the BEA engine can be used to evaluate the expressions that are part of these update statements.

11 Performance experiments and results

This section presents the results of a set of performance experiments that assess the compilation time, running time, and memory footprint of the BEA streaming XQuery engine for XML transformations based on use cases drawn from BEA WLI customer scenerios and on the XMark benchmark [20]. These experiments were carried out on a PC with a 2.4-GHz Pentium IV processor and 1 GB main memory. The operating system was SuSE Linux 9.0. The Java 2 Runtime Environment, Standard Edition (build 1.4.2-b28), and the Java HotSpot Client VM (build 1.4.2-b28) from Sun were used.

11.1 XML transformations: Xalan vs. BEA engine

The first experiment examines the performance of the BEA streaming XQuery engine for typical operations performed by customers of the WebLogic Integration product, i.e., for transformations for which the engine was designed to work well. As an alternative, XSLT stylesheets [25] were used to implement these same use cases; since XSLT is a stable W3C recommendation, XSLT is commonly used in practice today to implement these kinds of transformations. The use cases tested here explore various basic XML transformations on different kinds of XML messages. The XSLT stylesheet tests were executed using Xalan-J version 2.5.2 [24]. The XQuery queries were (of course) executed using the BEA streaming XQuery engine. In both cases (XSLT and XQuery), the best possible formulation was chosen if a transformation could be expressed in different ways. Furthermore, to focus on relative engine performance, we factored out XML parsing times and only measured the running times of the transformations after the XML input had been parsed. This net cost of XML transformations is the most relevant metric for BEA's WebLogic Integration product, as an XML message is typically parsed once and then transformed and processed several times over the course of its lifetime in the system.

Table 1 shows the results. In all cases, executing the XQuery expression on the BEA engine was much faster than executing an equivalent XSLT stylesheet using Xalan. In the best case, the speedup was a factor of 82; there was no particular pattern or transformation type for which the speedup was especially high. There are several factors that contributed to these speedups, the two most important being that XQuery is easier to optimize (more declarative) than XSLT and that the token streams used by the BEA engine can be processed more efficiently than the document table model used in Xalan. (Note: The document table model replaced DOM as the XML representation in Xalan starting with version 2.4 for the purpose of improving performance; older, DOM-based versions of Xalan had worse performance on our tests.)

11.2 XML transformations: compilation time, memory footprint

Table 2 shows the compile times for the BEA transformation queries using the BEA XQuery engine. As a reference, the table also contains the running times of the queries. Two observations can be made. First, XQuery transformations can be compiled and optimized in less than 1 s. In this experiment, it never took more than a quarter of a second to compile and optimize a query, and it took less than 50 ms to compile most of the queries. (In general, type-checking and type-related operations are the most expensive part of compilation in the BEA XQuery compiler). The compilation times for the BEA XQuery processor are essentially in the same ballpark as the compilation times of commercial SQL engines.

The second observation to be made from Table 2 is that the compilation time was always significantly higher than the running time for these particular XML transformations. The reason for this is that these transformations are all relatively small, in-memory operations (all of which need to execute very quickly when run repeatedly), as opposed to database queries that incur client/server communication times, I/O times, transaction costs, and so on. This observation makes it clear why it is so important to precompile queries in a product like WebLogic Integration and why it is critical to provide an interface like the XDBC interface in order to support such precompilation. For this reason, WebLogic Integration precompiles the transformation queries for a given type of business process at deployment time; the precompiled plans are cloned when a new instance of the process is created and are simply parameter-bound and then executed when the transformations need to be invoked at running time.

Table 2 also shows the memory footprint taken at running time for executing the XML transformations. Here, the token stream was implemented as a Java object (Sect. 5.4). For most transformations, the memory requirements were just a few kilobytes. The heaviest use case here involved complex manipulation of strings and required almost 400 KB of main memory. This set of customer use cases also involved two transformations with joins; these two required 100 KB and 70 KB of main memory.

11.3 XML transformations: significance of optimization

In the third experiment, we studied the impact of the most important optimization rules of the XQuery compiler (Sect. 7). In order to study the impact of an optimization rule, we disabled that rule in the XQuery compiler, compiled the queries, and measured the running times of the resulting query evaluation plans. Comparing the running times of these query evaluation plans with the running times of the (regular) plans that

Table 1. BEA customer use cases – BEA engine (XQuery) vs. Xalan (XSLT): running time (ms), speedup

Description	*XQuery (ms)*	*XSLT (ms)*	*Speedup*
Straight element mapping	0.06	5.18	82.24
Element mapping to different names	0.13	4.27	32.58
Element combination	0.11	4.09	36.21
Element explosion	0.16	5.66	35.18
Element to attribute mapping	0.19	4.15	22.30
Attribute to element mapping	0.18	3.92	21.30
Attr. to attr. mapping – straight copy	0.37	4.12	11.13
gq2-2-3. gq2-2-3 Attr. to attr. mapping – name mapping	0.39	4.08	10.45
gq2-2-4. gq2-2-4 Repeating group to repeating group	0.17	3.89	23.32
Static fields and rep. grp. to rep. grp.	0.25	4.08	16.59
Regrouping by key fields	1.47	5.66	3.85
Decreasing loop nesting	0.31	4.89	15.87
Incr. loop nesting	0.16	4.03	26.00
Incr. loop nesting using an input key	1.51	5.66	3.75
Conditional repeating group transf.	0.39	3.83	9.84
String functions	2.74	5.26	1.92
Aggregation of data	0.18	3.79	21.55
Parameterized queries	0.18	3.75	21.20
Parameterized transformations	0.08	3.83	50.38
Union: docs of the same schema	0.15	4.02	26.81
Union: docs of different schemas	0.66	4.13	6.24
Joining multiple docs	1.46	5.70	3.89
Joining with substitution	1.41	5.54	3.92
Repeated key value lookup	0.46	4.31	9.47

Table 2. BEA customer use cases: compile time (ms), running time (ms), memory (KB)

Description	*Compiler (ms)*	*Runtime (ms)*	*Memory (KB)*
Straight element mapping	12.3	0.06	1.1
Element mapping to different names	28.0	0.13	3.6
Element combination	27.7	0.11	2.5
Element explosion	30.5	0.16	9.8
Element to attribute mapping	36.4	0.19	4.8
Attribute to element mapping	53.9	0.18	4.0
Attr. to attr. mapping – straight copy	67.0	0.37	11.5
Attr. to attr. mapping – name mapping	71.1	0.39	12.0
Repeating group to repeating group	20.9	0.17	5.8
Static fields and rep. grp. to rep. grp.	36.8	0.25	8.6
Regrouping by key fields	66.0	1.47	52.3
Decreasing loop nesting	29.8	0.31	9.1
Incr. loop nesting	19.7	0.16	3.4
Incr. loop nesting using an input key	42.2	1.51	58.1
Conditional repeating group transf.	40.9	0.39	12.5
String functions	253.2	2.74	396.9
Aggregation of data	16.6	0.18	5.5
Parameterized queries	12.9	0.18	6.1
Parameterized transformations	16.6	0.08	1.8
Union: docs of the same schema	11.2	0.15	1.5
Union: docs of different schemas	52.7	0.66	36.7
Joining multiple docs	48.0	1.46	99.7
Joining with substitution	66.9	1.41	68.3
Repeated key value lookup	57.4	0.46	120.0

Table 3. BEA customer use cases: impact of optimization rules: running time (ms), speedup

Description	*full*	*doc order*		*dup-elim*		*IDs*	
	(ms)	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup
Straight element mapping	0.06	1.15	18.25	0.99	15.67	0.72	11.46
Element mapping to different names	0.13	1.28	9.76	1.01	7.70	0.77	5.90
Element combination	0.11	1.28	11.28	1.01	8.94	0.80	7.04
Element explosion	0.16	1.33	8.27	1.08	6.72	0.81	5.04
Element to attribute mapping	0.19	1.43	7.67	1.13	6.08	0.87	4.66
Attribute to element mapping	0.18	1.46	7.91	1.19	6.48	0.98	5.34
Attr. to attr. mapping – straight copy	0.37	1.67	4.51	1.39	3.75	1.13	3.06
Attr. to attr. mapping – name mapping	0.39	1.71	4.39	1.39	3.57	1.01	2.58
Repeating group to repeating group	0.17	1.31	7.86	1.37	8.23	0.82	4.94
Static fields and rep. grp. to rep. grp.	0.25	1.40	5.70	1.51	6.15	0.89	3.63
Regrouping by key fields	1.47	1.92	1.30	1.70	1.15	1.48	1.01
Decreasing loop nesting	0.31	1.50	4.88	1.27	4.13	0.99	3.21
Increasing loop nesting	0.16	1.32	8.52	1.02	6.61	0.88	5.65
Incr. loop nesting using an input key	1.51	2.16	1.43	1.92	1.27	1.57	1.04
Conditional repeating group transf.	0.39	1.91	4.91	1.29	3.31	1.06	2.73
String functions	2.74	4.80	1.75	4.29	1.57	3.69	1.35
Aggregation of data	0.18	1.34	7.60	1.17	6.62	0.81	4.61
Parameterized queries	0.18	1.69	9.55	1.14	6.42	0.86	4.83
Parameterized transformations	0.08	1.25	16.47	1.21	15.89	0.74	9.68
Union: docs of the same schema	0.15	1.23	8.22	1.00	6.66	0.79	5.25
Union: docs of different schemas	0.66	2.61	3.94	2.13	3.22	1.27	1.92
Joining multiple docs	1.46	2.22	1.51	2.03	1.39	1.62	1.11
Joining with substitution	1.41	2.85	2.01	2.44	1.73	1.97	1.40
Repeated key value lookup	0.46	1.76	3.86	1.55	3.40	1.19	2.61

were produced with all rules enabled shows how important that rule is for a given query workload. The following three optimization rules were studied in this way:

- *doc order:* This rule eliminates unnecessary operators that sort data into document order. As mentioned in Sect. 7, XPath semantics involve sorting in document order after every path step. In many situations, however, sorting is unnecessary (e.g., for a simple *child* step) because the result is already sorted. (This is similar to "interesting order" rules in relational optimization.)
- *dup elim:* This rule eliminates unnecessary operators for duplicate elimination. Again, the relevance of this rule is due to the special semantics of XPath steps.
- *IDs:* This rule controls whether, and which, node identifiers are generated for a query. If this rule is turned off, node identifiers will be generated at running time to evaluate all queries, even though only a few of the queries may really need node identifiers.

Disabling these optimization rules had little impact on the compilation time, but doing so increased the running times of the queries, in several cases dramatically (by a factor of 18 in one extreme case). Table 3 shows the resulting running times (depending on the optimization rule) and the factor by which the running time increased, again for the WebLogic Integration data transformation use cases. As a baseline, Table 3 shows the running times of the queries if all optimizations are turned on (the *full* column). Almost all queries benefited significantly from all three of these optimization rules, so it is safe to conclude that all these rules should be enabled at all times. (Fortunately, disabling an optimization rule never resulted in any running time improvements.)

11.4 XMark benchmark

Table 4 shows the compilation times, running times, and memory requirements of the BEA XQuery engine for the XMark benchmark [20]. The XMark benchmark was designed for testing the performance of XML database systems using rather traditional database workloads (e.g., selections on large collections of data) rather than data transformations. The benchmark includes a suite of 20 benchmark queries that test a large variety of features of the XQuery language. Furthermore, the XMark benchmark specifies how the XML data must be generated and defines a scaling factor in order to produce databases of different sizes. Table 4 shows the running times and memory requirements of the 20 XMark queries on databases of size 120 KB and 3.8 MB. Again, only the running times of the BEA engine on parsed XML input are reported. As a baseline, using the Xerces parser [23], parsing the 120-KB XML database takes 73 ms and parsing the 3.8-MB XML database takes 1580 ms.

The purpose of trying these experiments was simply to stress-test the BEA engine, as neither the nature of the workload nor the sizes of the benchmark databases are representative of use cases for which the implementation of the BEA engine was tuned. Nevertheless, all XMark queries could be executed in 50 ms or less for the 120-KB XML database. In other words, executing the XMark queries was always cheaper than parsing the document. These results confirm that the BEA engine is capable of processing XML messages of sizes up to a few hundred kilobytes regardless of which type of query needs to be processed.

For the 3.8-MB XML database, the measured running times were in the range of 10 ms up to 34 s (Q9). The BEA

Table 4. XMark benchmark: compile time (ms), running time (ms), memory (KB), 120-KB and 3.8-MB database

		120 KB		*3.8 MB*	
Query	*Compiler (ms)*	*Runtime (ms)*	*Memory (KB)*	*Runtime*	*Memory*
Q1	46.5	3.43	135.9	56.3	2,063.3
Q2	47.6	0.93	5.5	20.4	156.6
Q3	119.7	9.47	491.6	273.7	2,165.4
Q4	62.3	8.78	476.8	267.7	2,166.8
Q5	23.3	1.24	62.7	33.7	1,208.7
Q6	11.1	2.12	0.6	71.6	0.6
Q7	15.2	20.27	216.6	630.8	5,113.4
Q8	69.3	37.08	640.8	26,060.2	7,945.0
Q9	114.8	51.12	821.6	33,999.7	12,278.1
Q10	211.7	18.99	520.2	3,547.5	13,154.6
Q11	65.5	22.54	535.1	14,279.6	8,011.6
Q12	71.7	9.39	423.5	4,981.0	8,010.4
Q13	28.4	0.35	8.5	11.5	324.8
Q14	19.6	8.49	434.5	283.1	3,141.2
Q15	100.7	0.39	2.3	13.0	58.6
Q16	96.4	0.99	48.2	29.2	898.8
Q17	26.4	1.62	82.5	46.7	1,581.5
Q18	20.7	0.94	44.3	28.3	434.7
Q19	30.3	4.15	105.4	144.2	3,131.2
Q20	59.8	5.35	199.3	185.8	4,124.3

Table 5. Impact of optimization rules, XMark benchmark (120 KB): running time (ms), speedup

Query	*full*	*doc order*		*dup-elim*		*ids*	
	(ms)	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup
Q1	3.43	13.58	3.96	9.44	2.75	7.89	2.30
Q2	0.93	14.19	15.26	9.50	10.22	5.34	5.74
Q3	9.47	20.09	2.12	13.95	1.47	10.10	1.07
Q4	8.78	22.73	2.59	15.14	1.72	8.42	0.96
Q5	1.24	11.52	9.29	7.75	6.25	5.39	4.35
Q6	2.12	11.05	5.21	7.83	3.69	5.65	2.67
Q7	20.27	28.14	1.39	23.30	1.15	21.98	1.08
Q8	37.08	68.75	1.85	54.48	1.47	48.49	1.31
Q9	51.12	76.90	1.50	67.34	1.32	55.67	1.09
Q10	18.99	41.43	2.18	37.26	1.96	19.79	1.04
Q11	22.54	50.40	2.24	40.96	1.82	31.14	1.38
Q12	9.39	37.13	3.95	25.65	2.73	16.61	1.77
Q13	0.35	10.40	29.71	7.01	20.03	4.68	13.37
Q14	8.49	17.25	2.03	12.77	1.50	10.10	1.19
Q15	0.39	10.06	25.79	8.26	21.18	5.14	13.18
Q16	0.99	11.58	11.70	8.37	8.45	5.13	5.18
Q17	1.62	12.49	7.71	9.51	5.87	6.93	4.28
Q18	0.94	11.67	12.41	9.55	10.16	6.47	6.88
Q19	4.15	15.69	3.78	11.43	2.75	8.60	2.07
Q20	5.35	16.47	3.08	13.09	2.45	10.10	1.89

engine was found to be robust in all cases, but it is clear to us from these numbers that the running times can be improved; 3.8 MB is a much larger input size than the current implementation of the engine was tuned for. (The engine was designed to be extensible in the future in order to scale to such scenarios, but doing so has not been an actual product requirement so far.)

Turning to the compilation times, it can be observed that, again, the query compilation times were always less than 1 s, even for the most complex XMark queries. Looking at the memory footprint, it can be seen that it was rather high in the case of the XMark experiments. In the most extreme case, the memory requirements were more than three times as high as the database alone (see Q9 and Q10 for the large database). Again, the engine can be tuned in the future to reduce the memory requirements in this situation (e.g., by using a different implementation for the token stream, as described in Sect. 5.4).

Lastly, Tables 5 and 6 show the impact of the three optimization rules studied in the previous subsection on the XMark

Table 6. Impact of optimization rules, XMark benchmark (3.8 MB): running time (ms), speedup

Query	*full*	*doc order*		*dup elim*		*ids*	
	(ms)	(ms)	Speedup	(ms)	Speedup	(ms)	Speedup
Q1	56.30	390.30	6.93	235.90	4.19	179.80	3.19
Q2	20.40	418.60	20.52	316.10	15.50	145.60	7.14
Q3	273.70	553.70	2.02	417.20	1.52	272.20	0.99
Q4	267.70	576.30	2.15	494.00	1.85	251.10	0.94
Q5	33.70	378.90	11.24	251.10	7.45	163.60	4.85
Q6	71.60	415.30	5.80	305.10	4.26	177.90	2.48
Q7	630.80	1,000.50	1.59	858.90	1.36	795.20	1.26
Q8	26,060.20	29,789.10	1.14	28,251.20	1.08	26,305.30	1.01
Q9	33,999.70	38,289.90	1.13	36,081.20	1.06	33,611.60	0.99
Q10	3547.50	4442.70	1.25	4316.70	1.22	3651.10	1.03
Q11	14,279.60	19,756.80	1.38	18,022.40	1.26	15,150.90	1.06
Q12	4,981.00	7,372.70	1.48	6,316.30	1.27	5,411.70	1.09
Q13	11.50	351.20	30.54	238.80	20.77	143.70	12.50
Q14	283.10	497.00	1.76	422.20	1.49	339.50	1.20
Q15	13.00	355.50	27.35	239.90	18.45	146.30	11.25
Q16	29.20	382.90	13.11	245.80	8.42	164.60	5.64
Q17	46.70	393.70	8.43	297.20	6.36	181.80	3.89
Q18	28.30	402.50	14.22	305.70	10.80	161.50	5.71
Q19	144.20	505.80	3.51	371.40	2.58	297.30	2.06
Q20	185.80	577.40	3.11	439.70	2.37	358.40	1.93

benchmark queries for both the small and large databases. Again, the key observation from before is the same: All three optimizations are necessary, but their effectiveness depends on the particular query. Comparing the speedup factors in the two tables, the optimization rules were, as an overall trend, more important for the large database than for the small database (as one would expect).

12 Related efforts

Although the XQuery language specification has not yet reached Recommendation status, there are significant efforts both in industry and academia to implement XQuery and to use it for different application scenarios. Virtually all major database vendors are currently working on extending their database products and/or establishing new products based on XQuery. In order to extend relational databases, the SQL/X standard is also emerging [8], and there is currently discussion in the SQL/X world about possibly embedding XQuery as an XML-processing sublanguage in SQL. Furthermore, vendors of native XML database systems (e.g., Software AG) are naturally using XQuery as a query interface for their product. In addition, there are a number of startups and open-source initiatives that are working on XQuery implementations. A list of public XQuery implementations and links to Web demos can be found on the home page of the W3C XQuery Working Group: `www.w3c.org/XML/Query`.

In the research community, a number of related aspects of XQuery have been addressed recently. To name just a few of the most recent results: Ludäscher et al. [17] describe a stream-based implementation of a subset of XQuery using transducers, and Diao et al. [7] show how information filters defined as XPath expressions (which are also a subset of XQuery, of course) can be implemented using FSAs. Finally, [15] and [19] are two very recent papers on techniques for implementing XQuery for streaming XML data.

13 Conclusion

This paper has described the design, implementation, and performance characteristics of the BEA streaming XQuery engine. Unlike most other XQuery implementations, this engine is fully compliant with the XQuery specifications of the W3C. Currently, the engine implements the August 2002 version of the XQuery specifications. We are currently amending the implementation taking the November 2003 version of the specifications into account; the November 2003 version is the last version before XQuery 1.0 formally reaches Recommendation status. As we have described, the BEA XQuery engine is a central component of BEA's WebLogic Integration (WLI) 8.1 product. As such, it was tuned to provide high performance for XML message-processing use cases. Experiments using real customer use cases confirm that it indeed has very good performance for such applications; in fact, it significantly outperforms Xalan, a popular XSLT processor that has been under development for several years and has been tuned for similar applications.

References

1. Boag S, Chamberlin D, Fernandez M, Florescu D, Robie J, Simeon J (November 2003) XML query 1.0: an XML query language. http://www.w3.org/XML/Query
2. Carey M, Blevins M, Takacsi-Nagy P (December 2002) Integration, web services style. IEEE Data Eng Bull 25(4):17–21
3. Chamberlin D, Fankhauser P, Florescu D, Marchiori M, Robie J (November 2003) XML query use cases. http://www.w3.org/XML/Query

4. Chamberlin D, Florescu D, Lehti P, Melton J, Robie J, Rys M, Simeon J (2002) XUpdate. http://www.w3.org/TR/2002/WD-xupdate-20021015
5. Cooper B, Sample N, Franklin M, Hjaltason G, Shadmon M (2001) A fast index for semistructured data. In: Proc. of the conference on very large data bases (VLDB), Rome, Italy, pp 341–350
6. Draper D, Fankhauser P, Fernandez M, Malhotra A, Rose K, Simeon J (2003) XQuery 1.0 formal semantics. http://www.w3.org/TR/query-semantics/
7. Diao Y, Altinel M, Franklin MJ, Zhang H, Fischer P (2003) Path sharing and predicate evaluation for high-performance XML filtering ACM Trans Database Sys 28(4):467–516
8. Eisenberg A, Melton J (September 2002) SQL/XML is making good progress. ACM SIGMOD Record 31(3):101–108
9. Fallside D (May 2001) XML schema part 0: primer. http://www.w3.org/XML/Schema
10. Fiebig T, Helmer S, Kanne C, Moerkotte G, Neumann J, Schiele R, Westmann T (2002) Anatomy of a native XML base management system. VLDB J 11(4):292–314
11. Florescu D, Kossmann D (1999) Storing and querying XML data using an RDBMS. IEEE Data Eng Bull 22(3):27–34
12. Fernandez M, Malhotra A, Marsh J, Nagy M, Walsh N (November 2003) XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/TR/query-datamodel/
13. Graefe G (1993) Query evaluation techniques for large databases, ACM Comput Surv 25(2):73–170
14. Grust T (June 2002) Accelarating XPath location steps. In: Proc. ACM SIGMOD conference on management of data, Madison, WI, pp 109–120
15. Gupta A, Suciu D (2003) Stream processing of XPath queries. In: Proc. of the ACM SIGMOD conference on management of data, San Diego, June 2003
16. Halevy A (2001) Answering queries using views: a survey. VLDB J 10(4):270–294
17. Ludäscher B, Mukhopadhyay P, Papakonstantinou Y (2002) A transducer-based XML query processor. In: Proc. of the conference on very large data bases (VLDB), Hong Kong, August 2002, pp 227–238
18. Malhotra A, Melton J, Walsh N (2003) XQuery 1.0 and XPath 2.0 functions and operations version 1.0. http://www.w3.org/TR/xquery-operators/
19. Peng F, Chawathe S (2003) XPath queries on streaming data. In: Proc. of the ACM SIGMOD conference on management of data, San Diego, June 2003
20. Schmidt A, Waas F, Kersten M, Carey M, Manolescu I, Busse R (2002) A benchmark for XML data management. In: Proc. of the conference on very large data bases (VLDB), Hong Kong, August 2002, pp 974–985
21. Shanmugasundaram J, Tufte K, Zhang C, He G, DeWitt D, Naughton J (1999) Relational databases for XML documents: limitations and opportunities. In: Proc. of the conference on very large data bases (VLDB), Edinburgh, UK, pp 302–314
22. Tatarinov I, Ives Z, Halevy A, Weld D (June 2001) Updating xml. In: Proc. of the ACM SIGMOD conference on management of data, Santa Barbara, CA
23. Xerces-J (2000) http://xml.apache.org/xerces-j
24. Xalan-J.2.5.2. (2003) http://xml.apache.org/xalan-j
25. Extensible Stylesheet Language XSLT (2002) http://www.w3.org/Style/XSL/