The BEA/XQRL Streaming XQuery Processor

Daniela Florescu[1] Chris Hillery[1] Donald Kossmann[1,2] Paul Lucas[1]
Fabio Riccardi[1] Till Westmann[1] Michael J. Carey[1] Arvind Sundararajan[1]
Geetika Agrawal[3]

[1]BEA Systems
San Jose, CA, USA
http://www.bea.com

[2]Technical University of Munich
Munich, Germany
http://www3.in.tum.de

[3]Stanford University
Palo Alto, CA, USA
http://www.cs.stanford.edu

Abstract

In this paper, we describe the design, implementation, and performance characteristics of a complete, industrial-strength XQuery engine, the BEA streaming XQuery processor. The engine was designed to provide very high performance for message processing applications, i.e., for transforming XML data streams, and it is a central component of the 8.1 release of BEA's WebLogic Integration (WLI) product. This XQuery engine is fully compliant with the August 2002 draft of the W3C XML Query Language specification. A goal of this paper is to describe how an efficient, fully compliant XQuery engine can be 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, and the definition of the language has not entirely stabilized, 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 such traditional query languages, 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 forty 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 [Cas02]. 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.

Because of the wide range of applications for which XQuery is intended, and the powerful semantics and type system of the language, something of a myth has emerged that the complete XQuery language is going to be very difficult to implement and that it may be almost impossible to achieve 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 show that this myth about XQuery is indeed just a myth; i.e., that it is indeed quite possible to implement the entire XQuery language specification, types and all. 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 en-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003

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

gine implements the entirety of the August 2002 specification of XQuery. We also 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 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 presents the results of performance experiments. Section 11 discusses related work. Finally, Section 12 contains conclusions and avenues for future work.

2 Requirements

BEA is a very standards-focused company. As a result, from the outset, the BEA streaming XQuery engine had a major requirement to be fully standards-compliant and to implement the entire XQuery recommendation. Performance was also a major requirement for the engine; in particular, the engine was designed to provide very high performance for message processing applications (i.e., for streaming XML data). Among the major needs for message processing XQuery applications are: (i) an efficient internal representation of XML data, (ii) the use of streaming execution (i.e., pipelining) to the extent possible, and (iii) the efficient implementation of XQuery transformations that involve the use of many node constructors.

While completeness and high performance were the top two major design goals for the BEA streaming XQuery engine, there were a number of additional requirements that influenced the 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 six months. This productivity was only possible by using Java (vs. C or C++) as a programming language for the implementation.

- Integration into BEA products: The engine was designed to be an embedded component of other BEA products (in particular WLI 8.1). Again, this 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 Section 9).

- Usability with other components: The engine was designed to be usable with third-party parsers, schema validators, persistent XML stores, etc.

start-up called XQRL, Inc.

- The engine must operate properly in a clustered environment and on multi-processor machines.

- Since the XQuery specification is not yet stable (and was definitely unstable during the time when the BEA streaming XQuery engine was developed), it must be affordable to adapt the engine down the road to incorporate changes in the XQuery language specification.

3 XQuery in WebLogic Integration 8.1

As mentioned in the Introduction, the application that drove the design of the BEA streaming XQuery engine is BEA WebLogic Integration (WLI) 8.1, BEA's enterprise application integration product. WLI is the portion of the BEA Platform Suite that provides tools that 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 [CBTN02]. As such, the XQuery language plays a central role in WLI 8.1. XQuery is used for specifying data transformations on messages and workflow variables, i.e., for transforming data such as purchase orders as it flows through the system. XQuery is also used to specify the data-driven process flow logic (i.e., the looping and branching) of WLI workflows.

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 most important role of the XQuery engine in WLI 8.1. BEA is making a significant bet on XQuery being the right technology for this task. To provide a good design-time experience, WLI provides a built-in tool that enables integration developers to create XQuery-based data transformations without coding (i.e., without having to remember the syntax of XQuery). Figure 1 shows a "map view" screen shot taken from the beta version of WLI 8.1. In the example shown, the tool is being used to create an XML-to-XML transformation that converts an XML purchase order in one format to an XML purchase order in a different format; the source and target formats are specified as XML schemas that are shown as source and target trees in the user interface. At the time of the screen shot, the workflow developer was putting the finishing touches on the handling of addresses, which are complex structures in the input format but simple atomic values in the output format. Based on this "map view", WLI 8.1 automatically generates the corresponding XQuery query. This query can then be hand-edited if desired. (The WLI 8.1 data transformation editor supports limited two-way editing of XQuery queries.)

While XML-to-XML transformations are the most common form of data transformations expected, WLI 8.1 actually supports a much broader range of transformations using the data mapping tool and the BEA streaming

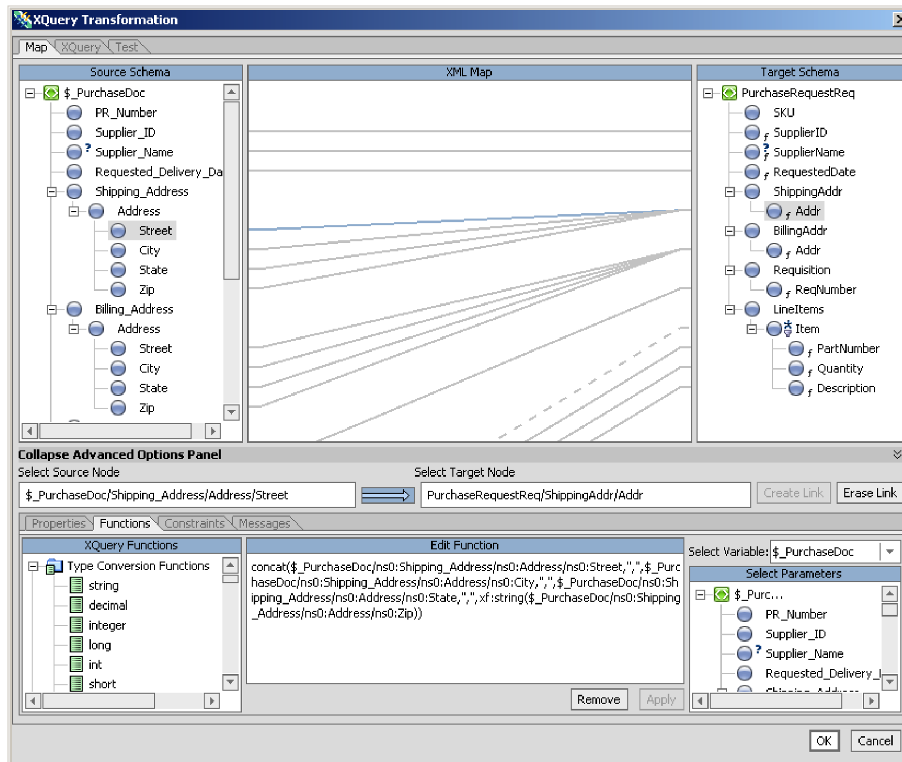

Figure 1: Graphical View of an XML Data Transformation

XQuery engine. In addition to single-document-in, single-document-out use cases, the mapper supports the graphical construction of queries that accept multiple input arguments (e.g., a purchase order and a customer profile). It also supports the design of transformations that begin and/or end with Java objects or binary data formats rather than just documents that are instances of XML schemas. In such cases, the mapper still shows the transformation's input and/or output types as trees, so the design model is 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 such cases, where a data transformation's input or output format is non-XML, a transformation step into or out of XML occurs prior to the central XQuery-based transformation. In all cases, for efficiency, the actual internal data representation is the TokenStream format described in section 5.

The other major use of XQuery in BEA WLI involves workflow process logic. Specifically, a typical WLI 8.1 workflow can include a number of XQuery expressions that serve to define the flow logic for the workflow. These expressions can be used in conditional nodes (decision nodes in the workflow) that control which branch of the flow should be processed next. They can also be used in iteration loops (loop nodes in the workflow) that drive the workflow to do something once for each piece of something else, e.g., once for each line item in a purchase order. These uses of XQuery are also tool-based, in terms of how they are specified by a developer. In this case, there is a special editor that helps the developer to edit XPath expressions in the conditions of branches of a workflow.

4 XQuery Engine Overview

An overview of the BEA streaming XQuery engine is given in Figure 2. Java applications submit XQuery queries and consume query results through an interface referred to as XDBC; the name XDBC is derived from JDBC. The query is then parsed 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 [F02] and of the XQuery core (e.g., sorts and joins). Furthermore, the runtime system contains an XML parser and an XML schema validator which are required when external XML data must be processed as part of a query. In WLI 8.1, incoming XML messages are parsed and schema-validated once and then stored in a special format so that they can be used in many XQuery queries without paying the high cost for parsing and schema validation for each query invocation. These messages are bound as free variables to queries and variable

bindings are also carried out by the means of the XDBC interface.

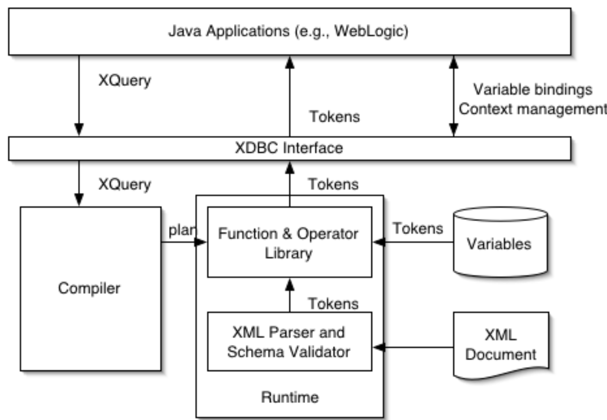

Figure 2: Overview of the BEA Streaming XQuery Engine

All XML data is represented as a stream of tokens that are roughly equivalent to SAX events in their semantics (i.e., a depth-first unfolding of an XML tree). This token stream minimizes the memory requirements of the engine. In addition, the token stream allows the lazy evaluation of queries. At runtime, each runtime operator consumes its input a token at a time and input data that is not required is simply discarded. The token stream matches the XQuery data model [M02]. Furthermore, the BEA engine provides tools that applications developers can use in order to serialize the token stream or construct a DOM representation from a stream of tokens (not shown in Figure 2).

The query engine is implemented entirely as a library, so it is embeddable in any application that might need to manipulate XML data. Input and output to the query engine is only supported using XML token streams, or by using a parser that translates some other form of XML into its token stream format. A toolkit of utility classes is provided to allow the instantiation of parsers, serializers, and adaptors that can efficiently couple the XQuery engine with most XML processing applications. The token stream itself is defined as a Java interface in order to allow for different implementations; the default implementation uses simple Java objects.

4.1 Compile-Time Issues

Because XQuery is a strongly typed language with a fairly complex type system, one of the important subtasks of the compiler is to verify the type consistency of the query with respect to its input sources and derive the type of the query result by deriving the partial types of each subexpression using type inference rules. Type information is also very 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 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).

Structural typing is found in functional languages (e.g., ML and Haskell) and it is considered algorithmically challenging since, in the general case, accurate type inference can be very costly (the type subsumption operation has exponential complexity). We have studied the patterns of utilization of the different type-related algorithms in the specific context of XQuery, and by intelligently caching type comparison results, we have been able to improve the performance of the type system implementation by more than two orders of magnitude with respect to the classical algorithms. We will say more about this in Section 6.

4.2 Run-Time Issues

In the XQuery world, queries are compiled and executed against execution contexts; these contain type definitions, XML schemas, function libraries, and variable definitions. In the BEA streaming XQuery engine, execution contexts can be made persistent and can survive query execution, though they are immutable objects from a logical point of view. Moreover, contexts in the XQuery world can be stacked, with a new context increasing or overriding the content of the previous context(s). The XQuery `LET` operator, for instance, defines a new context in a query, introducing a new variable and overriding previous variable declarations with the same name from outer contexts.

The main design goal of the runtime system is 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, but at the same time, it is possible to exploit all the knowledge available at compile-time (in particular, typing information).

5 XML Token Stream

In this section, we will briefly describe the structure of the XML token stream which is used in order to represent XML data. Since it is straightforward, we describe it by the means of a small example. Consider the following element declaration:

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

The parser translates this element into the following token stream (serialized); although the notation used here is textual, the actual tokens are of course binary and very compact.

```
[ELEMENT [judgement@http://www.bea.com/example],
  [anyType@http://www.w3.org/2001/XMLSchema]
```

```
]
[ATTRIBUTE [index],
  [anySimpleType@http://www.w3.org/2001/XMLSchema]
]
[CharData 11]
[END ATTRIBUTE]
[TEXT 43.5 ]
[END ELEMENT]
```

This example shows that there is an ELEMENT token whose name is judgement and type xs:anyType. The ELEMENT token is followed by an ATTRIBUTE token whose name is index and type is xs:anySimpleType. Then, follows the value of the attribute (11) 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 represents the content of the element and finally an END ELEMENT token closes the element declaration.

The previous data can be validated against the following XML Schema snippet [Sch01]:

```
<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 of doing so is the following slightly different token stream:

```
[ELEMENT [judgement@http://www.bea.com/example],
  [vote@http://www.bea.com/example]
]
[ATTRIBUTE [index],
  [int@http://www.w3.org/2001/XMLSchema]
]
[int 11]
[END ATTRIBUTE]
[TEXT 43.5 ]
[float 43.5]
[END ELEMENT]
```

The type of the ELEMENT token is now ”vote” and that of the ATTRIBUTE is ”xs:int”. The value of the attribute is a binary int and the element TEXT value is enriched with the binary float representation. Note that the XQuery standard mandates that the original lexical representation of elements is preserved, so both the “untyped” and “typed” values are kept in the token stream.

Although the XML token stream is mainly designed for the processor's internal data representation, it turns out to be a very convenient XML format for application interchange. The token stream allows XML fragments to be managed easily, and it is easy to serialize it, both for the network and on disk. In fact, WLI uses a serialized token stream format whenever it needs to persist the values of XML workflow variables; since it uses XQuery as its principal means for XML processing, its intermediate results are best kept in a format directly amenable to efficient XQuery processing.

6 Type System

XQuery has a rich (and complicated) type system [Sem02] which is compatible with XML Schema [Sch01]. In addition to 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 is in the required form, hence type. To determine this, the questions that an XQuery processor needs to be able to answer are:

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

An XQuery type (e.g., `(xs:integer | xs:string)*`) can be represented as 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, for example). While representing types as trees seems natural, trees don't allow the questions mentioned above to be answered easily. One reason why not is because many different trees can represent the same (non-trivial) type.

Regular expressions, and 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 our 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 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 above mentioned questions using algebraic operations on FSAs (union, intersection, complement). Unfortunately space restrictions do not allow us to go into the details of such operations. A forthcoming publication will describe them in detail. In this section, we will only try to give a feeling for what the type system does and the challenges that we encountered while implementing it.

To answer the question ”Are two types 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. So “type equlity” 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 in order to compute intersection: type T intersects type U if the complement of the union of the complements of U and T is empty.

Due to recursions, FSA algebra operations are very expensive. It can be shown that the computational complexity grows exponentially with the complexity of the FSA. To alleviate this problem, the engine aggressively caches all the results of the type system. Therefore, if the same two types are compared several times during the compilation of a query, expensive FSA computation is carried out only once. Caching is effective because the total number of types involved in a 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 the declarative approach whenever possible in order to keep our compiler simple and extensible.

The XQuery compiler is composed of three managers: the *Expression Manager*, the *Context Manager*, and the *Operation Manager*. Furthermore, there are three functional components: the *Query Parser*, the *Query Optimizer*, and the *Code Generator*.

7.1 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). Its expressions are equivalent in functionality with the algebraic query representation used by most relational query engines. Our internal representation for expressions borrows ideas from functional programming compilation, relational query compilation, and object-oriented query compilation, all adapted to XQuery, of course.

The simplest XQuery expressions are constants and variables. We support four types of variables: let, for, count and external variables. All the first order expressions (e.g. boolean operators, comparisons, arithmetics, union, intersection, and user defined functions) share a single internal representation. This is different from the traditional relational query internal representation but it is essential for keeping the code simple and extensible. Each kind of second order expression (e.g. `FLWR` expressions, sortby and quantifiers) has a separate internal representation.

XQuery is formally defined in terms of a small core set of algebra operators into which the full language is mapped [Sem02]. Our internal representation is redundant, in the sense that we have models for both core and non-core expressions. For example, we have both a representation for `FOR` and `LET` expressions, as well as for the complex `FLWR` expressions. However, we do not have an internal representation for path expressions, 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 like in traditional relational query processing. This distinction makes no sense for many kinds of XQuery expressions like conditionals, instance of, typeswitch, etc, and for most first order expressions. For the operations for which multiple possible physical implementations are available (e.g. node constructors and joins) the choice made by the optimizer is expressed by expression annotations.

Finally, the Expression Manager implements various functionalities required for query optimization like variable and substitution management, type derivation, semantic properties derivation, copying, subexpression cut and paste, etc.

7.2 Operation Manager

The second manager is the *Operation Manager*, which holds all the information about the first order functions and operators available to the query engine. This information includes the operator names and signatures, semantic properties (see below), pointers to the class implementing each operator and to the Java code for type derivation of polymorphic operators. The semantic information includes: 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, and, finally, whether the operator is a real function or not (i.e. returns the same result given the same input)[2], etc. This semantic information is exploited during the optimization phase for equivalent expression rewriting. All this information is loaded while bootstrapping the XQuery engine out of a declarative description.

7.3 Context Manager

The third manager used during the compilation is the *Context Manager*. Each phase in query processing (parsing, type checking, optimization, execution) is done in a certain context; such contexts hold a variety of environmental properties. Examples are: the current managers that perform various tasks during compilation and execution (e.g. the *Type Manager*, the *Schema Manager* and the *NodeIdentifier Manager*), the in-scope variables, the schema validation context, the in-scope definitions (e.g. namespace, functions, types, schemas, collations) and the current default specifications (e.g. element name namespace, function namespace, strip whitespace parameter, collations). The same context is passed through all query processing phases; expressions and iterators only exist in a certain context.

The context is composed of a hierarchy of local contexts. Searching for information translates into searching

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

from the local context recursively up to the parent until the information is found or the root is reached. The root of the hierarchy is the base context that holds all of the XQuery engine default parameters. This base context is also bootstrapped out of a declarative specification.

7.4 Parser

In addition to these three managers, as mentioned above, 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. During parsing, the current parsing context can be augmented with certain information like new namespace definitions, new function definitions, etc. Some normalization is performed during parsing; in particular, path expressions are eliminated at this time.

The main source of complexity in the XQuery parser comes from the necessity of keeping the language free of reserved words. Because of this requirement, the parser must keep multiple lexical states and perform complex state transitions during parsing. In this environment, a big challenge (that we are still facing) is to provide users with high quality debugging and error information. We used the ANTLR parser generator, which proved to be satisfactory for our needs.

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.

First, we have to address the expression equivalence problem. Two expressions are *equivalent* if they 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. This definition is idealistic; unfortunately while building a real query optimizer we realized that we have to relax it in important ways. For example, our optimizer will translate an expression E_1 into an expression E_2 even if the two expression E_1 and E_2 do not have the same inferred type[3]. Moreover, preserving errors is too restrictive. Most XQuery operations can result in errors which are not predictable at compile time. As a result, almost any operation reordering could change the potential result in the event of an error. Our rewriting methodology guarantees that no *new errors* are being introduced as a result of query rewriting, but it is possible that the original expression returns an error whereas the rewritten expression does not. A prominent example in which this can happen is a query with a Boolean AND expression if one of the subexpressions returns false and the other one returns an error.

[3] The two expression could still produce the same output even if the inferred type is not the same.

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). The optimizer itself is built by the successive application of such rewriting rules using heuristics. How the rewriting rules are composed in an optimizer strategy is also specified declaratively.

Rewriting rules can be either (a) normalization rules, whose purpose is to put the expression into a "normal" form without reducing the cost, and (b) cost-reduction rewriting rules that are supposed to translate an expression into another, less expensive expression. Examples of normalization rules are: putting predicates into a conjunctive normal form, dispatching general comparisons to the type-specific operators, dispatching arithmetics to the type-specific operations, inlining non-recursive XQuery functions, transforming typeswitch expressions into a cascade of conditionals, and unnesting FLWR expressions found in `FOR` and `RETURN` clauses.

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

1. Remove *unnecessary operations* whenever possible. A prominent example for such unnecessary operations are sort and duplicate elimination operations which 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 casts for function parameter and results; these casts are also defined implicit in the XQuery semantics.

2. Rewrite with *constant* and *common subexpressions*. For example, subexpressions that are executed as part of a loop (i.e., in a `FOR`, 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. Subexpressions that can be computed statically and whose results are small, are computed statically and replaced with their value.

3. Enable *streaming* whenever possible. An important example is rewriting expressions that use backward navigation into expressions that use only forward navigation whenever possible.

4. Exploit *schema information*. The most important rule in this class translates the expensive `descendant()` operator into a sequence of `children()` operators based on schema information, or introducing *topN* operators based on cardinality information obtained from schemas.

5. Carry 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.

6. Transform the nested loop (non-dependent) joins and outerjoins into *hash-based ones* whenever possible. Given our absence of data statistics this decision is based on heuristics.

Our current 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 essentially 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 think that an XQuery optimizer *needs* good heuristics to compensate for the problems described above, and cost-based optimization simply has not been important for our particular target application.

During query rewriting we make extensive use of both the semantic information associated with the operators that we described above 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. We designed our entire XQuery engine with the main goal to stream data in and out of the query engine, therefore 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 like sort, duplicate elimination and aggregates, the value of a variable must be materialized in three cases: when the variable is used multiple times in the 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 (see Section 8).

A noticeable absence in the BEA engine is the choice of physical operators for joins and selections. In the current version, a simple scan for selections is supported. For joins, nested-loops and hashing are supported. Again, these decisions have been made with regard to the particular application domain (processing of XML message in the range of a few kilobytes up to about a megabyte) for which the BEA streaming XQuery engine was designed. For other applications (in particular, large-scale XML databases), more methods can be supported and the optimizer must be extended accordingly.

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 Token Iterators so that this task is quite simple: the Token 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 Token Iterator trees. In order to avoid this situation, we delay code generation for recursive functions until they are actually executed; the Token Iterator tree corresponding to a new iteration is unfolded at runtime at the beginning of an iteration. While the recursion could require some extra materialization, it is itself not blocking the operator pipeline.

8 Runtime System

The task of the *Runtime System* is to interpret a query execution plan, 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 [F02] and for all functions of the XQuery core (e.g., map) [Sem02].

8.1 Iterator Model

Like most SQL engines, the BEA streaming XQuery engine is based on an iterator model [Gra93]. The reasons for this choice are the same as in the relational world: (a) modularity, (b) low main memory requirements, and (c) avoidance of (CPU and I/O) costs to materialize intermediate results. Furthermore, the iterator model allows lazy evaluation of expressions which is particularly important for XQuery. Sometimes, only a small fraction of the result of a sub-expression 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 the next token of the result stream; return “null” in order to indicate the end of the stream.
- *close*: release allocated resources and do clean-up work.

In addition, 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 carries out a fast forward to the next item in a sequence. This function is conveniant in functions that only look at the "tip of the iceberg" (e.g., count or nodetests). Both *peekNext()* and *skipNext* can be implemented in a generic way for all iterators so that this additional method does not increase the complexity of the code base. For *skipNext*, however, it is beneficial for performance reasons to provide specific implementation for certain functions; for example, a *skipNext* can be carried out particularly fast if the data is materialized.

Another specific feature of the BEA iterator model is its error handling mechanism. Every call of the *next()* method of an iterator can result in a failure. Based on the semantics of XQuery, some failures can be ignored whereas 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. Furthermore, 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, every function and operator of the XQuery library [F02] and core [Sem02] 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 depending on the characteristics of a query. In all, the runtime system contains the implementation of 350 iterators. In order to get an impression of the kind of iterators found in the system, some examples are listed in the following:

Constant

One of the simplest iterators is the Constant iterator that 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 a great deal of implicit casts. These casts can be very expensive. Some of these casts require transformations, e.g., from strings to numeric values. Some casts involve the extraction of typed values from an element or attribute. Finally, some casts call a function called *atomization* [Que02] which takes a sequence as input and returns a simple value.

As mentioned in Section 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 used when no static type can be inferred (i.e., the static type is "any"); more specific and cheaper casts are used if more information can be deduced statically (e.g., an expression has a simple type).

Materialization

Although the BEA streaming XQuery engine tries to stream data whenever possible, there are situations in which the materialization of intermediate results is necessary. One important situation in which materialization is necessary is a query that uses the results of a common sub-expression several times and (re-) computation of this sub-expression is expensive. Such queries are implemented using an iterator factory. This factory takes the token stream produced by the common sub-expression as input and allows the dynamic generation of iterators that consume this 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 when the last (slowest) consumer is done.

Node Id Generation

The XQuery data model assigns a unique id to each node of an XML document [M02]. 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 (i.e., non-blocking) using specialized GenerateId iterators. Id generation is an expensive operation and the memory requirements of ids can become prohibitive. Therefore, different types of ids and GenerateId iterators are used, depending on the requirements of a query. Most queries can be processed using simple, light-weight ids that are based on a pre-order numbering of nodes only; with such ids, duplicate elimination and sorting in document order can be implemented. Some queries, however, cannot be processed using such light-weight ids; such queries, for instance, involve backward traversals (e.g., the XPath parent axes) or special node comparisons. To evaluate such queries and only such queries, heavy ids that are based on pre-order and post-order 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 GenerateId iterator to use is made at compile time, depending on the characteristics of the query. In fact, it is possible that no ids are necessary to

evaluate a query; in this case, the compiler does not generate a GenerateId iterator.

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 sub-elements are relevant as soon as a sub-element of a particular type has been found. Furthermore, the runtime system implements a special algorithm in order to execute descendant-or-self. This algorithm starts optimistically and assumes that the schema has no recursion. In this case, the algorithm is fully stream-based and no intermediate results need to be materialized. In bad cases, the algorithm adapts and materializes data and behaves just like a traditional algorithm in order to compute descendants recursively.

9 XDBC Interface

The Java binding for our XQuery engine is called XDBC. It is designed to look and behave much like JDBC. The two APIs have similar features, such as 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, the same method names for XDBC. There is currently an initiative to standardize a Java interface for XML data called JXQI. Once a standard interface has materialized, we plan to make the XDBC interface upwardly compatible with it.

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 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. Future revisions will likely introduce the concept of remote XDBC services.

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

9.2 Parameterized Queries

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

The ability to bind and re-bind external variables to an XQuery statement is a BEA-specific extension to the XQuery language. We extended the variable rules for XQuery so that it is no longer required that all variables mentioned within a statement must be assigned values before use. Such statements will still compile in the BEA implementation. Attempting to execute a statement without binding all unbound external variables will result in an exception, however. PreparedStatement has methods for determining the set of unbound variables discovered by query compilation.

9.3 Execution of Queries

Once all external variables have been bound, the PreparedStatement is executed via *executeQuery()*. This method returns a Token Iterator representing the token stream resulting from the statement's execution. We provide utility classes for doing basic manipulations on Token Iterators, such as serializing them as UNICODE or in a binary format. PreparedStatement also offers other features for specialized purposes, such as a *cloneStatement()* method for creating a duplicate PreparedStatement that can be rebound and re-executed; e.g., in a different thread.

10 Performance Experiments and Results

This section presents the results of performance experiments that assess the running time of the BEA streaming XQuery engine for XML transformations based on use cases of BEA customers and for the XMark benchmark [SWK+02]. All experiments were carried out on a PC with a 1.8 GHz Pentium 4 processor and 1 GB of main memory. The Java Virtual Machine was from Sun using JDK 1.4.1_01. For all experiments reported here, nested-loops were used in order to implement nested `FOR` expressions (i.e., joins and group by) because hash-based algorithms were not beneficial in these particular experiments.

10.1 XML Transformations

The first experiment studies the performance of the BEA streaming XQuery engine for typical use cases of customers of the WebLogic Integration product; i.e., transformations for which the engine was designed to work well. As an alternative, XSLT stylesheets [XSL02] were used to implement these use cases; since XSLT is already a stable W3C recommendation, XSLT is today commonly used in

description	*XQuery*	*XSLT*	*speedup*
Straight element mapping	0.7	5.4	7.66
Element mapping to different names	0.6	5.1	8.04
Element combination	0.6	5.1	8.74
Element explosion	0.7	6.3	9.09
Element to attribute mapping	0.8	4.1	5.27
Attribute to element mapping	0.8	4.4	5.82
Attr. to attr. mapping - straight copy	1.1	4.7	4.48
Attr. to attr. mapping - name mapping	1.1	4.7	4.33
Repeating group to repeating group	0.8	4.4	5.74
Static fields and rep. grp. to rep. grp.	0.7	5.0	6.69
Re-grouping by key fields	1.6	7.0	4.28
Decreasing loop nesting	1.1	6.7	5.90
Incr. loop nesting	0.9	4.5	4.96
Incr. loop nesting using an input key	1.9	8.2	4.20
Conditional repeating group transf.	1.2	6.1	5.24
String functions	3.7	6.9	1.88
Aggregation of data	1.7	4.8	2.90
Aggregation of data	0.8	4.8	6.16
Parameterized queries	0.8	4.9	6.37
Parameterized transformations	0.7	4.2	6.13
Union: docs of the same schema	0.9	5.4	5.73
Union: docs of different schemas	1.7	5.7	3.31
Joining multiple docs	2.1	7.2	3.46
Joining with substitution	1.9	5.9	3.10
Repeated key value lookup	1.0	5.3	5.22

Table 1: Time [ms]: Xalan (XSLT) vs. BEA Engine (XQuery)

Query	*120 KB*	*3.8 MB*
Q1	7.8	107.8
Q2	6.9	124.2
Q3	10.1	309.0
Q4	9.8	259.3
Q5	5.0	249.6
Q6	8.8	285.1
Q7	66.3	11,279.5
Q8	32.9	21,971.6
Q9	38.0	24,867.0
Q10	20.1	4,741.0
Q11	26.7	15,621.9
Q12	16.5	6,452.9
Q13	4.6	88.0
Q14	25.2	982.1
Q15	4.7	62.8
Q16	5.5	119.3
Q17	6.1	141.1
Q18	7.2	162.3
Q19	10.1	413.4
Q20	10.1	365.9

Table 2: Time [ms]: XMark Benchmark

practice in order to implement these kinds of transformations. The use cases test different kinds of XML transformations on different kinds of XML messages.[4] The XSLT stylesheets were executed using Xalan-J version 2.4.1 [XJ02]. The XQuery queries were (of course) executed using the BEA streaming XQuery engine. In both cases the best possible formulation was chosen if a transformation could be expressed in different ways. Furthermore, we 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 because an XML message is typically parsed once and then transformed and processed several times.

Table 1 shows the results. In all cases, executing the XQuery expression on the BEA engine is much faster than executing an equivalent XSLT stylesheet using Xalan. In the best case, the speed-up was a factor of 9. There was no particular pattern or transformation type for which the speed-up was especially high and there were several reasons for this speedup. We believe that these were the two most important reasons: (a) XQuery is easier to optimize than XSLT; (b) token streams (as in the BEA engine) can be processed more efficiently than the document table model which is used in Xalan. (Note: The document table model replaced DOM as a representation in this Xalan version for the purpose of better performance.)

[4] We plan to publish queries and data on a Web page.

10.2 XMark Benchmark

Table 2 shows the running times of the BEA engine for the XMark benchmark [SWK+02]. The XMark benchmark was designed to test the performance of XML database systems using rather traditional database workloads (e.g., selections on large collections of data), rather than transformations. This 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 it provides a scaling factor in order to produce databases of different sizes. Table 2 shows the running times of the 20 queries on databases of size 120 KB and 3.8 MB. Again, only the running times of the BEA engine on parsed XML input is reported. As a baseline, parsing the 120 KB XML database cost 73 milliseconds and parsing the 3.8 MB XML database cost 1580 milliseconds using the Xerces parser [XJ00].

The purpose of these experiments was to stress test the BEA engine. Neither the workload nor the size of the databases were representative of the use cases for which the implementation of the BEA engine was tuned. Nevertheless, all queries could be executed in less than 70 milliseconds 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 very capable of processing XML messages of sizes up to a couple 100 KB, regardless of which type of queries need to be processed.

For the 3.8 MB XML database, the running times were in the range of 100 milliseconds up to 90 seconds (Q9). The BEA engine was robust in all cases, but the running

times can be improved. 3.8 MB is much larger than what the implementation of the engine was tuned for. The engine is extensible in order to scale for such scenarios, but doing so has not been a product requirement so far.

11 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 emerging [EM02]. 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 start-ups 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, various related aspects of XQuery have been addressed recently. To name just a few of the very latest results: [LMP02] describes a stream-based implementation of a subset of XQuery using transducers; [DFZF03] shows how information filters defined as XPath expressions (also a subset of XQuery) can be implemented using FSAs. Finally, [GS03] and [PC03] are two very recent papers on implementing XQuery for streaming XML data. (At the time this paper was completed, copies of this work were not yet available.)

12 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 August 2002 XQuery specification of the W3C. It is a central component of the BEA WebLogic Integration (WLI) 8.1 product. As such, it was tuned to provide high performance for XML message processing. Experiments using customer use cases have confirmed that it indeed has very good performance for such applications; in fact, much better performance than Xalan, a popular XSLT processor, that has been under development for several years and has been tuned for the same sorts of applications.

References

[Cas02] XML Query Use Cases. `http://www.w3.org/XML/Query`, August 2002.

[CBTN02] M. Carey, M. Blevins, and P. Takacsi-Nagy. Integration, web services style. *IEEE Data Engineering Bulletin, Special Issue on Web Services*, 25(4):17–21, December 2002.

[DFZF03] Y. Diao, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. Submitted for publication, 2003.

[EM02] A. Eisenberg and J. Melton. SQL/XML is making good progress. *ACM SIGMOD Record*, 31(3):101–108, September 2002.

[F02] XQuery 1.0, XPath 2.0 Functions, and Operations Version 1.0. `http://www.w3.org/TR/xquery-operators/`, August 2002.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GS03] A. Gupta and D. Suciu. Stream processing of XPath queries. In SIGMOD [SIG03].

[LMP02] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In VLDB [VLD02], pages 227–238.

[M02] XQuery 1.0 and XPath 2.0 Data Model. `http://www.w3.org/TR/query-datamodel/`, August 2002.

[PC03] F. Peng and S. Chawathe. XPath queries on streaming data. In SIGMOD [SIG03].

[Que02] XML Query. `http://www.w3.org/XML/Query`, August 2002.

[Sch01] XML Schema. `http://www.w3.org/XML/Schema`, May 2001.

[Sem02] XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`, August 2002.

[SIG03] *Proc. of the ACM SIGMOD Conf. on Management of Data*, San Diego, USA, June 2003.

[SWK+02] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. A benchmark for XML data management. In VLDB [VLD02], pages 974–985.

[VLD02] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Hong Kong, China, August 2002.

[XJ00] Xerces-J. `http://xml.apache.org/xerces-j`, 2000.

[XJ02] Xalan-J.2.4.1. `http://xml.apache.org/xalan-j`, 2002.

[XSL02] Extensible Stylesheet Language XSLT. `http://www.w3.org/Style/XSL/`, January 2002.