XQuery Reloaded

Roger Bamford[3] Vinayak Borkar[5] Matthias Brantner[2] Peter M. Fischer[4]
Daniela Florescu[3] David Graf[2] Donald Kossmann[2,4] Tim Kraska[4]
Dan Muresan[1] Sorin Nasoi[1] Markos Zacharioudakis[3]

[1]FLWOR Foundation www.flworfound.org
[2]28msec, Inc. www.28msec.com
[3]Oracle Corp. www.oracle.com
[4]Systems Group, ETH Zurich www.systems.ethz.ch
[5]University of California, Irvine www.cs.uci.edu

Contact: contact@flworfound.org

ABSTRACT

This paper describes a number of XQuery-related projects. Its goal is to show that XQuery is a useful tool for many different application scenarios. In particular, this paper tries to correct a common myth that XQuery is merely a query language and that SQL is the better query language. Instead, XQuery is a full-fledged programming language for Web applications and services. Furthermore, this paper tries to correct a second myth that XQuery is slow. This paper gives an overview of the state-of-the-art in XQuery implementation and optimization techniques and discusses one particular open-source XQuery processor, Zorba, in more detail. Among others, this paper presents an XQuery Benchmark Service which helps practitioners and XQuery processor vendors to find performance problems in an XQuery processor.

1. INTRODUCTION

XQuery is more than ten years old. Its origins go back to the QL 1998 workshop held in Boston. Even though the W3C only recently released the XQuery 1.0 recommendation [8], the first public working drafts were published in 2001.

Originally, XQuery was designed as a *query language* for XML data. The goal was to provide the expressive power of a query language like SQL and, in addition, to support XML-specific operations such as navigation in hierarchical data. From the very beginning, an important feature of XQuery has been the capabilty to process untyped data. Furthermore, XQuery has been designed to support the processing of data on the fly or of data stored in the file system; it is not necessary that the data be stored in a database.

In its early stages, XQuery was quite popular both in industry and academia. At the beginning of this decade, XML and, as a consequence, XQuery, was a hot topic at the major database conferences (SIGMOD and VLDB). Furthermore, all the major database vendors integrated XQuery into their database products and made XQuery, like SQL, one possible way to extract data from a database. Based on user feedback, Oracle has at least 7000 customers using this feature.

Arguably, XQuery has been most successful in the middle-tier. In the middle-tier, XQuery can serve several purposes. One prominent example is the transformation and routing of XML messages [20]. Another example is enterprise information integration [11]. A third example involves the manipulation and processing of configuration data represented in XML. Obviously, the architecture of an XQuery processor designed for the middle-tier can vary greatly from the architecture of an XQuery processor which was specifically designed for the database.

A recent trend which potentially changes the adoption of XQuery is that XQuery is being extended by a number of additional features. These features go beyond message transformation and XML query processing for which XQuery was initially designed. Most prominently, the XQuery Update Facility [18] and XQuery Full-Text [3] (for XML information retrieval) have been devised and have reached recommendation status by the W3C. Furthermore, the XQuery Scripting Facility [17] and extended features such as the processing of windows for streaming data [16] are under development. With all these extensions, XQuery is much more than merely a query language; it has become an extremely powerful tool for development of any kind of data processing application.

The purpose of this paper is to revisit the advantages of XQuery and clarify some of the myths about XQuery which were created in the early days of XQuery when, indeed, XQuery was just a query language. Furthermore, this paper gives an overview of XQuery implementation techniques and presents a number of XQuery-related activities that are carried out by a group of people at various places (academia and industry) which were once called (in friendly terms) "XML extremists" by Jim Gray. These activities include the implementation of two open source XQuery engines, Zorba [27] and MXQuery [53].

VLDB '09, August 24-28, 2009, Lyon, France
https://doi.org/10.14778/1687553.1687560.

The remainder of this paper is organized as follows. Section 2 revisits the pros and cons of XQuery. Section 3 gives an overview of XQuery implementation techniques. Section 4 presents the design of one particular XQuery processor, Zorba. Section 5 lists some projects which were carried out with Zorba. Section 6 contains conclusions and avenues for future work.

2. WHAT IS XQUERY?

This section revisits XML and XQuery and why we believe that both are useful tools. Basic familiarity with both XML and XQuery is assumed. For an introduction to XQuery, the interested reader is referred to the XQuery specification [8] or some tutorials [25, 56].

2.1 Why XML?

Putting it bluntly, XML is useful because it reduces cost by increasing the flexibility of data management in various ways. Technically, XML is a *universal* syntax to serialize data. It is universal for two reasons. First, XML is platform-independent; i.e., XML works on all hardware and operating systems. Second, XML is based on UNICODE so that it supports all languages and alphabets.

The first kind of flexibility XML provides is to dissociate schema from data. This way, data can exist first and schema can be added later in a pay-as-you-go manner. Furthermore, this property helps to process data from legacy applications and archived data. For example, XML allows to generate data with one application according to the specific schema of that application and to process the data with another application, thereby using a different schema. This property has made XML an attractive data format for communication data (i.e., messages or data exchange).

The second kind of flexibility arises because XML is able to represent a large spectrum of data, from totally unstructured, semi-structured, to totally structured data. Furthermore, XML is able to represent data, meta-data, and even code that operates on the data and meta-data. This kind of flexibility has, for example, made XML the data format of choice for configuration data.

All these advantages have led to a wide adoption of XML; clearly, XML is here to stay. However, XML is also heavily criticized and many application developers avoid the use of XML whenever they can. First, XML is perceived to be slow, big, and clumsy. That is, XML data is typically much larger than the equivalent data represented in a proprietary format. Furthermore, XML processors have typically worse performance than processors that were specifically geared towards a specific (proprietary) format. A possible solution to these problems is the emerging *binary XML* recommendation of the W3C [49] which provides a pre-parsed and compressed universal representation of XML data.

The second critique against XML is that it is perceived to be complicated. This criticism is mostly directed against XML Schema [55] which indeed has many features that are rarely used. In practice, the work-around against this complexity is to use only those XML and XML Schema features which are really needed.

Obviously, XML is not the only syntax to serialize data. Traditionally, data exchange between business applications has been effected with the help of EDIFACT. EDIFACT is fast, but unfortunately, it lacks the flexibility of XML so that it can only be applied for a specific set of applications. Recently, JSON has been devised as a way to effect data exchange on the Internet. JSON is particularly popular for Web mash-ups. It turns out that JSON is equivalent to XML with similar pros and cons. In fact, for the purpose of this paper which focusses on XQuery, it is safe to use XML and JSON interchangeably as two different data formats which can both be processed by XQuery.

2.2 Why XQuery?

As for XML, the goal of XQuery is to reduce cost. What XML is for the representation of data, XQuery is for the processing of data and development of data-intensive applications. Again, the magic lies in increased flexibility.

The first kind of flexibility provided by XQuery is that XQuery operates on any kind of data. Naturally, XQuery is able to process XML data. However, as stated in the previous section, XQuery is just as well able to process JSON, EDIFACT, CSV, or data stored in a relational database. The XQuery processing model [8] specifies that XQuery expressions operate on instances of the XDM data model [22] and these instance can be generated from any kind of data and from any kind of data source.

Secondly, XQuery inherits all the flexibility provided by XML. In particular, XQuery can process untyped data so that XQuery supports the "data first - schema later" (or pay as you go) paradigm. Furthermore, XQuery is a natural choice to process archived data, communication, and configuration data represented in XML. Furthermore, XQuery is able to operate on the whole spectrum of unstructured to structured data. For unstructured data, the XQuery Full-Text extension [3] can come particularly handy. It is noteworthy that XQuery programs can be serialized into XML themselves by the means of the XQueryX recommendation [43] and that most XQuery processors (including Zorba which is described in detail in this paper) support an *eval* function which takes an XQuery program as a parameter for execution.

XQuery provides a special kind of architectural flexibility in the sense that XQuery runs on all application tiers. As mentioned in the introduction, XQuery runs in the database layer as it has been implemented by all major database vendors (e.g., IBM, Microsoft, and Oracle) as part of their database products. Furthermore, XQuery runs in the middle-tier; e.g., as part of the BEA enterprise service bus or in separate XQuery products such as Saxon, MXQuery, or Zorba. Finally, as shown in Section 5, XQuery also runs in the client as part of a Web browser plug-in [30]. This flexibility allows to move code between the application tiers, thereby reducing operational cost. Data providers, for instance, can chose to move computation to clients in order to reduce the load on their servers.

A myth about XQuery is that it is not powerful enough to build whole applications in XQuery and that, as a result, XQuery needs to be embedded into a host language such as Java or C#. This myth came from the time in which XQuery was merely a query language and this myth is supported by the name, XQuery. [36] reports on experiences gained by implemented a full-fledged enterprise Web application entirely in XQuery. One particularly valuable advantage of XQuery is that XQuery makes it much easier to customize enterprise Web applications. The same application code can be applied to data in different schemas by using XQuerys flexible data model and the "schema-later" approach of XML. For instance, if one variant of the application added a field to a specific business object, then all the existing code is still applicable to the extended (as well as the original) business object. As a result, XQuery and XQuery database are naturally multi-tenant and do not require heavy weight-lifting as is necessary to implement multi-tenancy in relational database systems [5]. A second advantage of implementing a whole application in XQuery in a single tier is that code for, say, error handling and checking of integrity constraints need not be duplicated across tiers.

Like XML, XQuery is conceived by many to be slow and complicated. One of the goals of the authors of this paper is to address these concerns by building high performance XQuery processors and by providing best practices and examples that demonstrate the power and usefulness of XQuery as a programming tool.

Obviously, XQuery has a great deal of competition. For any class of application, XQuery competes with a number of other programming languages. For instance, Ruby and PHP are particularly popular in order to rapidly develop (simple) Web applications. Java and .Net are still the gold standards for enterprise-scale Web applications. A survey that compares XQuery with other programming languages and programming paradigms is given in [26].

2.3 What is XQuery?

As mentioned in the introduction, XQuery is a family of recommendations of the W3C. It extends XPath and was co-designed with XSLT 2.0. As a formula, XQuery can be characterized as follows:

XQuery = Query + Update + Fulltext +
Scripting + Streaming + Libraries

Again, this paper will not give a tutorial on XQuery, but it is worth comparing XQuery to other programming languages. Database languages such as SQL typically cover the "Query", "Update", and potentially the "Fulltext" and "Streaming" aspects. General-purpose programming languages like Java or C# cover the "Scripting" and "Libraries" aspects. XQuery does it all. Even though XQuery is not an object-oriented programming language[1], XQuery is well suited for large-scale structured programming with modules, as discussed in the previous section and shown in [36]. Classes in Java correspond to modules in XQuery. Based on our experience, Java programers learn XQuery very quickly and achieve the same (and higher) level of productivity for enterprise Web applications with XQuery. Furthermore, as it will be discussed in Section 4, modern XQuery processors come with sophisticated libraries and XQuery allows users to create and publish their own libraries for re-use.

One noticeable omission in the XQuery family is a data definition language (DDL) which allows the specification of integrity constraints, the declaration of schemas, and the definition of a physical database design with indexes. SQL, obviously, provides such a DDL and such a DDL is also needed for XQuery applications. As shown in Section 4, one of the goals of the Zorba project is to devise and support such a DDL for XQuery applications.

In summary, it can be concluded that XQuery tries to combine the features of existing programming languages like SQL, Java, or even PHP. In this way, XQuery allows to implement sophisticated applications in a single tier and with a single uniform technology, thereby avoiding impedance mismatches and improving flexibility and customizability. Like SQL, XQuery supports declarative queries and updates; XQuery is able to specify bulk queries and updates which are best executed inside a database. In addition, XQuery supports the processing of streams and continuous queries with windows [13]. XQuery is, thus, able to support applications which involve complex event processing or the processing of data from sensor networks. Its full-text extension make XQuery also a good candidate to process RSS and Atom data feeds or any other form of unstructured and semi-structured data. Furthermore, XQuery extends XPath and was co-designed with XSLT 2.0 in order to support message transformations and routing in the middleware. In the middle tier, XQuery is also a good candidate for information integration as shown by several XQuery-based EII products [11]. XQuery is also a good candidate to implement enterprise Web applications and sophisticated application logic with strong typing and libraries. In this respect, XQuery competes with Java and .Net. In addition, XQuery can be used as a scripting language with untyped and possibly unstructured data. XQuery is also a viable option to implement mash-ups and RESTful services. Finally, XQuery supports the event-based programming of graphical user interfaces; here, XQuery competes with JavaScript as a programming language for the Web browser [30].

[1]XQuery is a functional programming language. It does not support inheritance and the bundling of methods and data in classes.

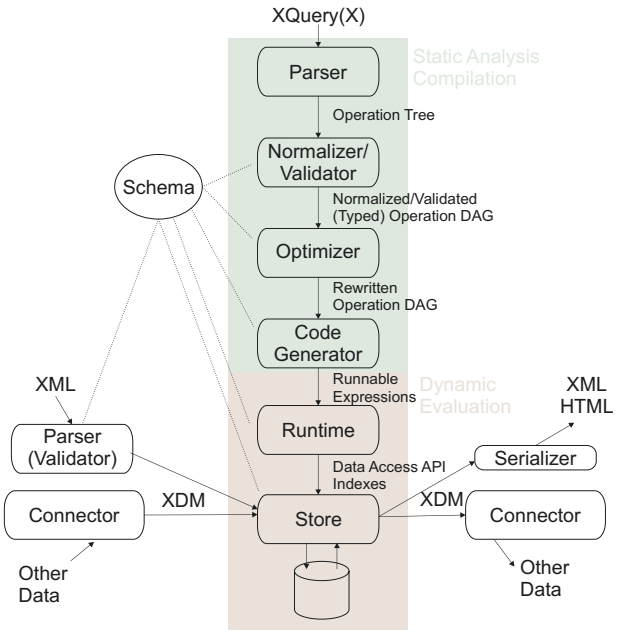

Figure 1: XQuery Processor Architecture

3. XQUERY PROCESSING TECHNIQUES

3.1 Architecture of an XQuery Processor

The XQuery specification specifies a processing model to evaluate XQuery programs [8]. This processing model prescribes particular operations and interactions, but does not specify how to implement them.

Figure 1 gives a generic architecture that most XQuery processors have adopted. This architecture is also related to the architecture used by most query processors of relational database systems and compilers/runtime systems of general purpose programming languages [2].

The most remarkably difference to traditional database architectures is the explicit use of XML parsers or other connectors in order to process data from external data sources and produce query results that can be consumed by other applications. Internally, all data is processed as instances of the XDM data model [22]. An XDM instance is an ordered sequence (or list) of *items* where an item is either an atomic value (e.g., an integer or string) or a node (e.g., an XML element or an XML attribute). Naturally, any XML data, a JSON object, or a relational table can be represented as an XDM instance which makes XQuery a candidate to process any of these kinds of data. If the XQuery processor is associated to a database, then that database would store its data as XDM instances. As explained below, the "Store" component of the XQuery processor provides a uniform way for the XQuery processor to access all data.

The following briefly summarizes the most important components of an XQuery processor:

1. *Parser:* The parser retrieves a textual representation of an XQuery program and generates an "operator tree" as an internal representation of the program. Since XQuery is a functional programming language, each node of the operator tree represents an "expression" of the XQuery program. Building an XQuery parser is quite challenging as the language has no keywords; this feature is part of the XPath legacy of XQuery.

2. *Normalizer+Validator:* This component checks references to namespaces, types, variables and functions. By resolving the variable references, it effectively turns the operator tree into a directed, acyclic graph. Furthermore, implicit operators such as casts are added to the operator tree according to the XQuery formal semantics [21]. If static typing is supported (which is an optional feature of XQuery), type information is inferred and checked for the expressions.

3. *Optimizer:* As in any other programming language, the goal of the optimizer is to transform the operator tree into an equivalent operator tree with lower expected running time or resource consumption. There are a myriad of different approaches and techniques in order to improve the performance of an XQuery program, including almost all techniques known from relational databases (e.g., access path selection and join ordering). As in traditional database systems, query optimization at compile-time can be based on heuristics or on cost estimations. This aspect is revisited in Section 3.2.

4. *Code Generator:* Again, as in any other system the generated code can be interpreted (using a runtime system) or compiled directly to a specific target hardware. Both approaches can be found in state-of-the-art XQuery processors.

5. *Runtime:* As in traditional (relational) database systems, the runtime system of an XQuery processor is typically organized using the iterator model [31]. That is, each basic expression of the XQuery language is implemented as an iterator with an *open, next, close* interface. Again, like in a traditional relational database system, there may be alternative implementations for the same kind of expression (e.g., different join algorithms) and the selection of the most beneficial implementation is made by the optimizer and the code generator.

6. *Store:* The Store maintains a collection of all XDM instances and provides a uniform interface for accessing items (e.g., nodes) of these XDM instances. As shown in Figure 1, the Store also integrates data from external data sources (push and pull). In order to do that, the Store contains a URI-resolver and fetches documents and collections identified by their URI from the Internet or its local database. The Store is also the component which synchronizes concurrent accesses to the data if an XQuery processor is used in a multi-threaded mode or several instances of an XQuery processor work on the same data concurrently. Depending on the usage scenarios (database, ad-hoc transformations, streaming) and the required functionality, the store requirements and optimizations in the Store can vary significantly.

7. *Schema:* In contrast to relational databases, schema management is an optional feature of an XQuery implementation. Nevertheless, all serious XQuery implementation support schema because schema information is needed for many applications (in particular, enterprise applications) and it can be useful for optimization.

3.2 Implementation Variants

This section gives an overview of implementation variants of the individual components of an XQuery processor and outlines the current best practices. As mentioned in Section 2.3, XQuery can be used in a wide spectrum of scenarios with varying requirements and thus different design decisions. In general, most implementations can be classified in one of the following three categories: *lightweight*, *full*, or *relational*. *Lightweight* implementations are typically used for ad-hoc transformations, embedding in other platforms or scripting. *Full* implementations are often used in the context of native XML databases and are more concerned about compliance and index usages, whereas relation engines are XQuery implementations based on relational databases. Of course, this classification might not fit for all implementations, but it demonstrates well the design space of implementations.

3.2.1 Parser, Normalizer & Validator

XQuery is a keyword-free language. As a result, lexer and parser generators are harder to apply if meaningful error codes and messages are required. Therefore, most XQuery processors, independent of the previously established classification, use hand-written parsers in order to maintain full control over the parsing phases.

Two common approaches exist to express the operator tree create by the parser: Lightweight and full implementations typically use an operator tree format which is designed along the lines of the XQuery operator specification, as this is a good way to achieve compliance. In contrast, relational implementations use an extended relational model as their intermediate representation. This extension of (conventional) relational algebra is required to deal with some of the specifics of XQuery, such as path expressions, element construction or general comparison. Not stating those constructs explicitly in the tree would make the optimization process much harder [7].

3.2.2 Compiler Architecture & Optimizer

Although the stages of parsing, normalization, and optimization with an accompanying framework and graph representation are state-of-the-art, lightweight implementations often avoid the related overhead by just having a single stage and representation. As shown in the case studies performed on XQRL [23], for many scenarions the compilation time exceeds query execution time by an order of magnitude if a multi-stage approach is used.

Therefore, lightweight engines parse the query directly into the runtime operators and perform transformations and optimizations directly on this operator tree. This yields fast compilation times while at the same time making the engine smaller and, hence, suited for small devices (e.g., MXQuery). The optimization itself is typically rule-based without schema awareness: A set of carefully ordered transformation rules is used to optimize the operator tree. Only the type information present in the expression definition is considered, but not additional, derived information provided by schema. However, in such an engine, adding additional optimization rules soon becomes very complicated. Optimization rules often depend on each other and require a carefully set execution order which is hard to control without an appropriate optimization framework.

Given their more traditional (database) workloads, full implementations typically prefer to use multiple stages and separate representations of logical and physical plans. The logical representations closely follow the XQuery expressions as defined in [8], thus simplifying translation and re-ordering of expressions. Again, rule-based optimization is common practice in this category of implementations. Furthermore, full XQuery implementations provide

schema support, as annotating the query plan with schema-derived information allows for optimizations such as general comparison rewrites. Given the complexity of XML Schema, almost all implementations try to build on top of an existing library. Xerces [4] has become the de-facto standard in this regard.

Relational implementations try to leverage their existing optimization frameworks and cost-baseds models as much as possible. Schema information is particularly important in this regards, hence almost all relational implementations rely on it and thus provide the necessary support. An extended version of the relational algebra is typically used as logical representation. The extension is made for the XQuery specific operators such as path expressions. Stating those expressions explicitly simplifies the optimization process. Even though relational implementations try to use their existing cost-based optimizer, the relevant cost models for XQuery have not yet reached a sufficient degree of maturity. Therefore, even for relational implementations, most of the XQuery-specific optimizations are still overwhelmingly rule-based instead of cost-based.

3.2.3 Optimizations

Standard XQuery-specific optimizations include join detection, constant folding, avoiding duplicate elimination, document ordering and node identifier operations [23]. Join detection allows replacing nested-loop joins (implied by the XQuery syntax and the ordering constraints) by more efficient join algorithms, which is particularly important for large data sets. Constant folding allows pre-computation of partial results and simplification of expressions. Duplicate elimination and document ordering are implied by the ordered nature of XDM and the semantics of many expressions, most prominently path and set expressions. Since their implementations tend to be expensive and pipeline-breaking, it is therefore important to only instantiate them when absolutely necessary. Since they typically rely on XDM node identifiers, eliminating them also helps in avoiding to generate node identifiers, which is one of the most expensive operations at the store level.

Other notable optimizations are the elimination of non-forward axes (parent, pre-sibling etc.) of path expressions and optimizations regarding general comparison. Using only forward axes allows tremendous simplifications and optimizations for the store, in particular enabling streaming execution of data accesses. Optimization regarding general comparison is one of the most performance-critical factors. Given the syntax of XQuery, users tend to write general comparisons (e.g, =, <, <=) even though a simple comparison (e.g., eq, lt, gt) would have been sufficient. General comparisons are expensive in two ways: Expressing the type casting and existential quantification required by the semantics of general comparison leads to complex implementations that can cost up to orders of magnitudes more than simple values expressions. In addition, general comparison is neither transitive nor reflexive, thus prohibiting many other optimizations and complicating the use of indexes. XML Schema often provides the information needed to rewrite general comparisons into value comparisons.

3.2.4 Runtime

Runtime implementations differ mainly in the following techniques, independent of the targeted use-case: iterator model (pull vs. push), runtime operators (relational vs. XQuery expressions), and internal data model representation (tokens vs. items).

Iterator Model. Most runtime implementations follow a pull-based iterator model as described earlier. Iterators allow for lazy evaluation and streaming execution, so that the runtime can deal with recursive function calls and data streams - also infinite data in the case of stream queries. Since materialization of intermediate results is avoided, the required memory footprint for processing is minimized. Unfortunately, the depth and nesting of XQuery operator trees often cause bad cache behavior and a high number of function calls between iterators. This is particularly bad if a fine-grained internal data representation, such as tokens, is used in combination with a lazily evaluated iterator model.

Therefore, some engines forego the iterator model and compile the code into native code or their own virtual machine code (e.g., MonetDB or Oracle). Consequently, such runtimes are rather push-based and apply a single operation at the time on the whole input data set before they forward the complete result to the next operations. Such an approach increases the cache locality and better utilizes the pipeline architecture of modern CPUs. On the other hand, this requires materializing the intermediate results, thus increasing the memory footprint and prohibiting lazy evaluation. Saxon is an example for an engine positioned between those extremes: It partly compiles the iterator tree into Java code and mixes push and pull depending on optimization heuristics. Oracle also performs this mixture of push and pull, using different operator implementations for different requirements.

Runtime Operators. The types of operators also differ. In contrast to engines extended for supporting XQuery (e.g., MonetDB, DB2, Oracle), the runtime operators of pure XQuery engines such as Saxon, MXQuery or Sedna closely follow the XQuery expressions [8]. Most fundamentally, the issue of expressing the stream of variable bindings in FLWOR expressions (also called *tuples* needs to be handled. An example of a such a tuple stream would be ($x=1,$y=3),($x=2,$y=3), produced by the expression for $x in (1,2) let $y := 3. These tuples cannot be expressed using XDM, so several approaches have been proposed: 1) extending the internal data model beyond XDM to provide tuple boundaries or groups 2) Encapsulating the tuple binding logic in single FLWOR operator orchestrating the operations of the *for* and *let* implementations. 3) Eliminating the variables altogether and provding XDM streams created by operators like joins. Whichever variant (or combination of variants) is chosen, it needs to provide enough flexibility to incorporate special operators such optimized join implementations. Furthermoe, is should cater for the extended, general FLWOR semantics of XQuery 1.1, which add operators such as *outer for*, *count* or *group by* and allow to freely combine all FLWOR constructs in any number and order, including multiple *order by* or *where* clauses.

Internal Data Model. Finally, the internal data model representation, i.e. the representation of the smallest object transferred between the operators, varies from items (Saxon) to tokens (MXQuery, XQRL). The item representation is closest to the XQuery Data Model (and thus the specification). Items may represent a single atomic value such as a string or even a complete XML tree/ document, whereas tokens are of smaller granularity and can be compared to (typed) events generated by a SAX parser. Although tokens are fast to generate by a suitable parser and allow for lazy evaluation even inside an XML item, they often require invoking every iterator several times to produce the content of single item of the result set. Hence, a more coarse-grained model, such as items, is often superior to tokens, reducing the required amount of function calls to generate a result.

General Runtime Optimizations. Independent of the choice of the internal representation, the iterator model and and the runtime operators, the optimizations of general comparison, numerical operations and FLWOR expressions are applicable to all architectures:

As for the compiler, general comparison is also an issue inside the runtime. The optimizer is often not able to substitute general comparison by value comparison. As mentioned before, general comparison is especially complex because of the applied rules specified in the specification. Thus, spending time to optimize the general comparison quickly pays off. The standard approach is to optimize the implementation for the common case, i.e, a simple value comparison, with exceptions and fall-back mechanisms for the less likely truly general comparison situation.

Another experience concerns operators for numerical values, as the numeric types defined in XML Schema (and used by XQuery) do not completely correspond to native machine types. When striving for compliance, this means that very generic data types such as BigDecimal in Java or MAPM in C++ need to be used, causing significant performance degradation. As the precision applied by XML Schema is usually only required in corner cases, implementations often introduce two internal representations: one following the exact specification and another, faster one using native types. Furthermore, operations on numerics such as plus or minus, again have complex typing rules making the implementation slower even if the types are known and no casting is required. Hence, best practise is to implement special operators optimized for the different types. Actually, the same experience applies to string representations (e.g., ASCII, UTF8, UTF16 etc).

The FLWOR expression introduces additional complexity: the so-called tuples, which represent variable bindings (no to be confused with relational tuples). Older approaches like XQRL made the tuples explicit inside the data model representation, thus substantially complicating the process of bindings and other iterators. More recent approaches (such as MXQuery) use special FLWOR operators to encapsulate this logic, hence simplifying the overall engines design.

3.2.5 Store

Design aspects of store implementations can roughly be described along the following lines: General design principles, usage-specific implementations and indexing.

General Design Principles. Certain methods and aspects are common for all types of stores. First of all, since XDM mandates that all nodes must have a way to identify them, implementations of node identifiers need to be provided. It is now common practice to also express structural constraints such as document order and parent/child-relationships in the node identifiers to simplify path expressions, set operations, duplicate elimination and document ordering. For updatable stores, Ordpath [45] is the state-of-the-art method, for read-only stores Dewey IDs [54] are used, which both encode the document structure in a compact way. Therefore, all operation requiring structural constraints can be supported efficiently. Generating and maintaining node identifiers is expensive, both in terms of computational cost and memory overhead. As outlined above, in many use cases, an optimizer can decide to avoid generating them.

Furthermore, the store is responsible for generating the internal data representation of XDM. Parsing and object creation have been determined as the major cost drivers. This is of particular concern if only fragments of the documents are needed and/or in the case of one-time transformations and streaming. Hence, document projection [42], comparable to projection push-downs, is one method to speed up the processing of document parsing and at the same time minimizing the work for the runtime as much as possible.

In the same region of optimizations regarding the store is the use of object pools and dictionaries for namespaces, elements and strings. The latter allows performing comparisons based on pointers instead of, for example, the string representation for element names.

Usage-specific Implementations. The main differentiators for storage implementations are the usage scenarios and the supported functionality. Since several XQuery engines will cover a range of usage scenarios and functionalities, they provide multiple different store implementations, and choose the most appropriate one depending on the required workload. For example, MXQuery has dedicated stores for read-only queries, full-text, streaming and updates/scripting.

Usage scenarios range from the classical database scenario of preloaded and heavily indexed XDM collections (DB2, Sedna, MonetDB) over the XML transformations and scripting scenario with (partial) ad-hoc materialization of items (Saxon, XQRL, MXQuery) to the XML data stream processing scenario in which subsets of the infinite XDM stream need to be buffered (MXQuery).

Among the different sets of supported functionality, the difference between read-only (XQuery, XQuery Full Text) and updatable (Update Facility, Scripting) XDM stores is most important: updatable stores need to provide facilities to support snapshot semantics (in order to cater for concurrent updates), revalidation support if typed XDM (Schema) is used, updatable node identifiers and the possibility to push the XDM updates to external data (e.g. by performing XML file modifications).

Beyond updateable/non-updateable, the store implemenatations can be categorized in several dimensions: First, stores can be divided into in-memory (e.g., XQRL, Saxon and MXQuery) and persistent stores (MonetDB, DB2, Sedna). Second, the storing technique can be roughly grouped into binary XML stores (XQRL, Sedna, Saxon, MXQuery), relational edge-stores (MonetDB), and hybrid relation/XML binary stores (DB2, Oracle). The storing techniques itself may be split according to the various ways of shredding into relational tables [24] or the different XML binary encodings [35, 6]. Comparing all these approaches is beyond the scope of this paper.

Indexing. Indexes play a similarly important role for XQuery as for SQL engines. However, data types and general comparison often complicate the use of indexes in queries, especially temporary ad-hoc indexes. It is therefore essential to have a clean index interface to provide the necessary information for such optimizations. Three types of indexes are important for XQuery: structural, value and full-text indexes. While full-text indexes are rarely implemented, structural and value indexes can be found in most implementations. Value indexes particularly vary in the way they are created. An approach implemented in XQRL indexes certain path-expressions (e.g., */author/name*), thus creating one value index per path. Alternatively, e.g., in DB2, structural and value indexes are combined. For the actual index structure hash tables or B-Trees are the common approaches.

3.3 Overview of XQuery Processors

The W3C (http://www.w3.org/XML/Query/) lists more than 50 XQuery implementations. Most XQuery processors are targeted either for XML message transformation or for use with a native XML store. Only a few XQuery processors are integrated into an

Engine	XQRL/BEA [23]	MXQuery [53, 13]	Saxon [37, 38]	Sedna [34, 29]	MonetDB [19, 9, 10]	DB2 [33, 47, 7]	Oracle [41]	Zorba [27]
Features	XQuery 1.0[1] Update[2] Scripting[2]	XQuery1.0 (99%) XQuery 1.1[4] Update Full-Text[4] Scripting	XQuery1.0 (100%) Update	XQuery1.0 (98.8%) Update[3] Full-Text[3]	XQuery1.0[1] Update	XQuery1.0[1] Update	XQuery 1.0[1] Update[4] Full-Text[4] Scripting[4]	XQuery1.0 (99%) XQuery 1.1 Update Scripting
Parser Optimizer	•XQuery operator representation •Rule-based optimization	•Direct Translation •No optimizer framework •Simple Re-Write rules	•Direct Translation •Simple optimizer framework •Simple Re-Write rules	•XQuery operator representation •Rule-based optimization	•XML Ext. Relational Algebra •Rule-based optimization	•XML Ext. Relational Algebra •Rule and cost-based optimization	•XML Ext. Relational Algebra •Rule and cost-based optimization	•XQuery operator representation •Rule-based optimization
Internal data representation	Token	Token	Item	Item	Token/Relational	Item/Relational	Item/Relational/ VM Sequences	Item
Processing Model	•Pull •Streaming •Iterator	•Pull •Streaming •Iterator	•Pull&Push •Streaming & Materialization •Iterator & compilation	•Pull&Push •Streaming & Materialization •Iterator	•Push •Materialization •VM language	•Pull •Streaming •XML-Extended Relational DB2 engine	•Pull&Push •Streaming & Materialization •Hybrid Relation/VM	•Pull •Streaming •Iterator
Store	Fixed Store	•Plugable Store •Specialized Stores for Full-Text, Streaming, Updates etc.	•Plugable Store •Different read-only implementations	Binary Store	Relational Column Store	Dedicated Binary XML Stores	Hybrid Approach: Relational and Binary XML	Plugable Store
Indexes	No index support - pure streaming	•Full-Text index •Restricted value index	•Structural index •Value index	•Structural index •Value index	Relational Index	Combined structural and value index	Structural and value indexes	Value index

[1]No conformance numbers available [2]Not officially published [12] [3]Own syntax, not compliant [4]Partially implemented

Table 1: XQuery Implementations (as of June 2009)

existing relational database system.

Table 1 gives an overview of the features of eight XQuery processors. XQRL, MXQuery, and Saxon are middleware products which do not include a persistent store. Sedna is based on a native XML store. MonetDB, DB2, and Oracle include XQuery as part of their relational database products. Zorba is explained in more detail in the next section.

One of the first commercial XQuery processors was XQRL [23]. Its algebra and operators are closely aligned with the XQuery operators. XQRL implements a traditional pull-based iterator model with tokens as internal representation.

MXQuery was designed following the same principles as XQRL, but in contrast to XQRL it runs on small devices. Hence, MXQuery does not have the complexity of an optimizer framework and directly translates the query into operators. The store of MXQuery is only responsible for materializing input documents, intermediate results and ad-hoc indexes. To deal with the different requirements and indexes, MXQuery makes use of specialized stores for different features (such as streaming or full-text).

Saxon is one of the most mature XQuery implementations. Saxon combines several interesting architectural aspects. First, Saxon applies a direct compilation of XQuery programs into an (executable / interpretable) operator tree, similar to the approach taken by MXQuery. Other than MXQuery, Saxon performs more advanced transformations and optimization on the (executable) operator tree. Furthermore, Saxon partly compiles iterators of the operator tree into Java Code and switches between pull- and push-based execution. The store is exchangeable, but no persistent store exits to date.

Sedna is an XQuery implementation based on a native XML store. Sedna implements the standard layered architecture. Interestingly, the execution is pull- and pushed-based, dynamically switched at run-time.

MonetDB is a relational XQuery processor. XQuery expressions are first compiled by the *Pathfinder* frontend into relational algebra expressions extended by certain functionality (e.g. staircase-join) and afterwards from MonetDB into the so-called MonetDB Assembly Language (MAL). The MAL code is then interpreted with an engine particularly tuned for array-processing in a one-operator at a time (per CPU) manner forcing to materialize all intermediate results. As storage, MonetDB's relational column store together with a pre-/post-based tree encoding into relational tables is used. Thus, no extra XML encodings or indexes exist.

DB2 is another XQuery implementation based on a relational database system. Like MonetDB, DB2's XQuery processor extends the relational algebra by XQuery-specific operators such as path expressions and element constructors. As a result, both SQL and XQuery are compiled into the same internal (extended) representation and (in theory) the same optimizations are applied to both kinds of programs. The DB2 store is a hybrid relational and native XML store. That is, a dedicated XML encoding exists inside the database. Furthermore, DB2 provides special region indexes for faster XML lookups. In addition, XML data can be indexed by a combined structural and value index.

Oracle XML DB is a hybrid query processor, combining XQuery push-down to an extended relational algebra (lazy evaluation) and a pure XQuery virtual machine (eager evaluation). In particular, XScript is compiled into virtual machine code. The query engine does XML storage/index dependent optimizations for different XML storage/index forms: structured XML with an object relational storage, semi-structured and content XML with a binary XML storage.

4. ZORBA

This section describes Zorba, an XQuery processor which was recently developed as a joint effort between the FLWOR Foundation, 28msec Inc., Oracle Corp., and ETH Zurich.

4.1 Motivation

Zorba is an open-source XQuery processor which can be freely copied, distributed, and altered (Apache 2.0 license). It is written in C++, and adopts the latest optimization techniques in order to achieve good performance (Section 3). In the same way as performance, standard-compliance has been a primary goal of the Zorba project from the very beginning. Zorba fully implements all XQuery-related recommendations of the W3C, including scripting and continuous queries with windows, which are W3C recommendations still under development. The only exception is XQuery Full-Text; we will start implementing Full-Text hopefully at the end of 2009.

One particular feature of Zorba is that it provides a public Web-based interface at http://try.zorba-xquery.org. This way, anybody can experiment and test XQuery programs without the need to install an XQuery processor, especially with regard to features and extensions that have not seen adoption in other implementations. Query plans and optimizations can be studied on the fly, since this information is also exposed in an easy-to-understand graphical format.

Furthermore, Zorba features a debugger and a performance profiler which lists how much time was spent in each XQuery function for a given execution of an XQuery program. Such tools are typically provided for general-purpose programming languages such as Java, C#, and C/C++. Supporting them in Zorba shows how close XQuery has already come to be a general-purpose programming language.

As shown on the W3C Web pages, there are dozens of XQuery processors already. Many of these processors are mature and have very high quality. Why yet another XQuery processor? There were two main reasons that motivated the work and design of Zorba:

- Pluggability: We wanted to have an XQuery engine that can be embedded into other systems. Given the broad range of applications that could benefit from XQuery (Section 2), it is clear that there is no one-size-fits all system that will support all these applications. The most important components of an XQuery processor, however, should be reused across all these systems.
- XQuery Extensions and Libraries: We wanted to have a platform that allows us to implement the necessary extensions and libraries that we believed are still missing in order to help XQuery come to a break-through.

The remainder of this section shows how Zorba achieves this pluggability and the XQuery extensions and libraries that we have added to Zorba.

4.2 Pluggable Store

Revisiting Figure 1, Zorba implements all components of an XQuery processor. As will be shown in Section 5, there is one component which plays a particular role when embedding Zorba into another system: the Store. In order to use Zorba as an XQuery front-end for a MySQL database, for instance, the parser, normalizer, etc. of Zorba need not be changed. Only the Store needs to be changed in order to serve XDM instances to the Zorba runtime system from the MySQL database.

In order to make the Store pluggable, the Zorba Store API was designed carefully. As stated in Section 3.1, the main purpose of the Store is to serve instances of the XQuery data model. As a consequence, the Store API involves all accessors of the XQuery data model, as defined in [22]. Furthermore, the Store API involves functions in order to apply *pending update lists (PULs)*, as defined in the XQuery Update specification [18].

When designing the Store API, there is a delicate trade-off with regard to the internal representation of atomic values such as strings, numerics, and dates. Different Stores may wish to have different representations for these atomic values. For instance, a string is represented differently in a Web browser than in a relational database, if one would like to embed Zorba both in a Web browser and in a relational database system (Section 5). Likewise, the implementation of node identifiers which are required in order to compare two nodes ([22]) may vary between Store implementations. Allowing Store implementers to have their own representation for these data types provides Store implementers with the maximum flexibility to implement these data types and their operations (e.g., string concatenation) in the most efficient way. On the other hand, however, many Store implementers do not wish to implement their own data types and functions on them. As a result, Zorba has default implementations for all data types as part of its default Store implementation and provides the flexibility to override these default implementations with other implementations. Unfortunately, this flexibility comes at a cost, both in terms of performance and development effort, if Store implementers wish to provide their own implementation. In terms of performance, each access to the store involves a virtual (C++) function call. In terms of Store implementation effort, implementers must provide implementation of primitives to atomic data types (e.g., arithmetics) if they wish to provide their own implementation of atomic types and avoid paying the price for data marshalling.

4.3 XQuery Extensions

We have proposed many XQuery extensions in the past. Some of which are currently under development as part of XQuery Scripting and XQuery 1.1. This section lists additional extensions to the XQuery language that we feel are important and that we have implemented as part of Zorba.

4.3.1 REST and Web Services

Service-oriented software architectures are beginning to play a dominant role, both in large-scale enterprise applications and in more light-weight Internet mash-ups. Obviously, XQuery programmers should be able to integrate (i.e., call) services provided by others using both REST and Web Services protocols. Furthermore, XQuery programmers should be able to easily expose their own modules as services which can be called, again, using REST and/or Web Services protocols.

An initial design how to expose XQuery modules as Web Services and call other Web Service using XQuery was devised in [46]. We have extended this design and adopted it to REST. The corresponding specifications have been submitted to the W3C [50, 28]. Furthermore, the REST proposal has been implemented in Zorba and both REST and Web Services have been implemented in MXQuery. Implementing these extensions in an XQuery processor like Zorba or MXQuery is straightforward.

4.3.2 Data Definition Language

As stated in Section 2, XQuery currently does not provide a syntax to define integrity constraints and declare indexes or collections. Clearly, such features are very useful and as a result, we have developed our own syntax. The following shows how a unique integrity constraint can be specified in a collection with a given URI;

in XQuery, a collection (of items) plays the same role as a table (of tuples) in SQL:

```
declare integrity constraint URI
         on collection URI $var
         check unique keys
          (<expr1>, ..., <expr n>)
```

In this clause, "$var" is bound to each item of a collection and can be used as a free variable in the expressions which specify the key of the collection. This syntax allows the definition of composite keys by giving more than one expression in the list of expressions. As a typical example, this syntax could be used in order to define that the social-security number (i.e., *$var//ssn*) is a key in a collection of employees.

The following syntax can be used in order to define referential integrity between foreign keys in items of a collection, identified by URI1, and keys of items in a collection, identified by URI2:

```
declare [unchecked] integrity constraint
     foreign key
        from collection URI1 node $x
             keys  ( $x/a, $x/b)
        to collection URI2 node $y
             keys  ( $y/foo, $y/bar)
```

4.3.3 Time Travel and Versioning

Recently, time travel has become popular in relational database systems [52]. The idea is to ask queries *as of* a certain point in time. This technology could, for instance, be used in order to look up the old address of a customer who has recently moved. The key idea is to extend the data model and make it a temporal data model in which each data item is associated a range of time stamps that indicate in which time frame the data item was valid.

This concept of a temporal data model which keeps versions of each data item can be naturally applied to XQuery. The XDM data model extensions are along the lines of the extensions for relational data. In order to implement time travel queries, we are proposing to extend XPath steps by an additional dimension which allows navigation in time (in addition to navigation along parent/child relationships). For instance, the following query would refer to the first (outdated) address which was ever recorded in the database for the customer with id 4711:

```
//customer[id eq "4711"]/first::address
```

Just as for the navigation of XML trees, there are a number of different steps which are supported along the time dimension: (a) *first* navigates to the first version and corresponds to the *root* step in the navigation of the tree; (b) *former* navigates to the previous version (pendant to *parent*); (d) *current* denotes the current version (pendant to *self*) (c) *all-former* returns all previous versions, without the current (*ancestors*); (e) *all-former-or-current* returns all previous version, including the current version (f) *next* navigates to the next version (*child*); (g) *all-next* returns all succeeding versions, excluding the current version (*descendant*); and (h) *all-next-or-current* returns all succeeding versions, including the current version (*descendant-or-self*). There are two time steps which have no equivalent in the tree navigation world: First, *last* navigates to the last version of an element which has been committed to the database. Second, *latest* navigates to a possibly uncommitted version of an element that has been created as part of the (running) transaction.

4.4 XQuery Libraries

This section describes several functions and libraries that we have implemented as part of Zorba. Providing such libraries does not require a change or extension to the XQuery programming language and fits nicely into the XQuery programming model. Many of these libraries can also be found in Java and implementing them in XQuery is an attempt to close the gap to Java. It is worth to note that a great deal of Java libraries (e.g., Java util or threads) is not needed in XQuery because the corresponding functionality (e.g., handling collections) is already built into XQuery.

4.4.1 Collection Functions

As mentioned before, collections in XQuery play the same role as tables in SQL. The *name* of a collection is represented as a URI and the standard XQuery function library provides the "fn:collection" function which returns all items of a collection given the URI of the collection. Unfortunately, the XQuery standard does not provide functions in order to manipulate collections; i.e., insert and delete items from a collection, creating and dropping collections. This gap is closed by the *collection* library implemented in Zorba.

4.4.2 Graphs and N:M Relationships

As in SQL, it is possible to implement keys and foreign keys using values; e.g., the SSN of an employee, as described in Section 4.3.2. An alternative way which is advocated by RDF is to implement references to nodes using URIs. In order to support this approach, Zorba implements two functions:

- *ref(node):* Given a node (e.g., an XML element or a JSON object), provide the URI of that node.
- *deref(URI):* Given a URI, return the referenced node.

As mentioned in Section 3.1, URIs are generated and resolved by the Store.

4.4.3 Eval Function

Zorba implements a special (higher-order) function, called *eval*, which takes an XQuery program as a parameter and executes it. The XQuery program may have free variables (i.e., parameters) and bindings can be specified as part of the call to the eval function.

4.4.4 Tidy

Zorba (and also MXQuery) have a built-in library that takes arbitrary HTML and XHTML as input (including non-standard and erroneous HTML), cleans it, and returns it as a string or XDM instance which can then be used for XQuery processing. The Zorba implementation is based on the open source *Tidy* library available at "http://tidy.sourceforge.net/". Such a library is extremely useful in order to implement Web mash-ups with the help of XQuery.

4.4.5 JSON

As mentioned in Section 3, XQuery was designed to process any kind of data. The W3C recommendations specify how to consume XML data and generate XML or HTML as query results. Zorba, however, is able to consume and generate JSON data just as well. This way, XQuery can be used in order to write Web mash-ups which involve JSON and/or XML data sources.

In order to effect JSON as a first-class citizen of every Zorba program, Zorba adopts the XML-JSON mappings proposed in [51]. The basic idea is to map JSON key-value pairs into a pair of XML elements having a name and type attribute. For example,

```
{ key1: value1, key2: value2 }
```

map to

```
<pair name="key1" type="typeOfValue1">
   xmlOfValue1
</pair>
<pair name="key2" type="typeOfValue2">
  xmlOfValue2
</pair>.
```

Here, *typeOfValue1* is the XML type associated to `value1` (e.g., xs:string if `value1` is a simple string and not nested); accordingly, *typeOfValue2* is the XML type assocated to `value2`. *xmlOfValue1* is a serialization of `value1` in XML, thereby recursively applying the same JSON to XML mapping rules if `value1` is nested. This way, the nesting of objects in JSON is naturally implemented by the nesting of elements in the mapped XML.

JSON arrays are mapped into sequences of XML elements; in such a sequence, each element has a "type" attribute. "values", "numbers", and "booleans" are mapped to XDM atomic values of type "string", "decimal", and "boolean", respectively.

The mapping works in both directions and, as a consequence, all results of an XQuery program can be serialized to JSON. How to map XML to JSON is specified as part of the JSonML specification [15].

4.4.6 E-Mail

Zorba also provides a library to send and receive E-Mails. This library can be configured to use specific SMTP, POP, and IMAP servers and to provide the user credentials (for security) whenever interacting with these servers. The Zorba library is able to set header information (e.g., the "to" or "reply-to" fields) and is able to handle multipart mime-encoded content (i.e., attachments). Similar libraries have also been implemented for Java. The Zorba implementation is based on the "c-client library" from the University of Washington.

5. XQUERY PROJECTS

This section gives an overview of a variety of alternative projects that were carried out using XQuery and/or Zorba as tools. These projects show the power of XQuery and that XQuery has developed way beyond being merely a query language for XML data.

5.1 Sausalito: An XQuery Application Server in the Cloud

Sausalito is a commercial product offered by 28msec Inc. It integrates the Zorba XQuery processor and the Apache Web Server into an XML database system. The goal is to provide a complete infrastructure in order to run any kind of XQuery application, including database applications, Web mashups, and data streaming applications. Sausalito extends the Zorba XQuery processor in the same way as an application server (e.g., WebLogic, WebSphere, or JBoss) extends a Java virtual machine; whereas in the latter case an additional relational database system (e.g., Oracle, SQL Server, or DB2) is required.

One special feature of Sausalito is that it runs in the Amazon cloud or any other private (enterprise) cloud which provides a scalable infrastructure to dynamically provision (and unprovision) servers and a scalable key/value store such as S3, Cassandra [40], or a relational database which can also be used as a key/value store. The architecture and design trade-offs of Sausalito have been presented in a recent paper [14] and talk [39] and are beyond the scope of this paper.

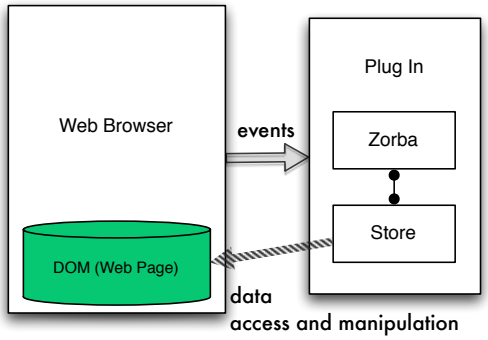

Figure 2: XQIB Architecture

Sausalito is a nice example of how Zorba can be embedded into different systems. Revisiting the architecture of Figure 1, Sausalito re-uses all components of Zorba with the one noticeable exception of the "Store". Even the XML parser, XML Schema Validator and Connectors (e.g., for JSON) of Zorba are reused in the Sausalito product. That is, Sausalito only implements its own Store. The Sausalito Store stores all data as blobs into a key/value store such as S3, as described in [14]. Furthermore, Sausalito runs many (possibly thousands) Zorba XQuery processors in parallel in the cloud and the Sausalito Store synchronizes concurrent updates; again, the distributed protocols used by Sausalito are described in [14].

Sausalito has been released as a beta version in February 2009 and in this beta period, Sausalito can be freely tested. The getting started guide and a number of demos are available at the 28msec Web site: http://www.28msec.com

5.2 XQIB: XQuery in the Browser

XQIB is another example which demonstrates how the Zorba XQuery processor can be embedded into another system. XQIB is a Web browser plug-in which allows to embed XQuery scripts into Web pages and implement AJAX-style user interfaces using XQuery in the same way as with JavaScript. XQuery is a natural candidate to implement AJAX applications in the browser because XQuery has all the important ingredients: It supports an event-based programming model, it has scripting capabilities, it supports HTTP calls to servers (like the XMLHttpRequest of JavaScript), and most importantly, it provides a declarative way to query and update a Web page in the browser. Inside the browser, a Web page is represented using DOM which is just another way of representing XML data.

The details of XQIB are described in [30]. XQIB is an open-source project and the plug-in can be freely downloaded from the XQIB Web site http://www.xqib.org. At this Web page, there are also a number of demos available that show how to embed XQuery into an HTML Web page: It is exactly the same mechanism as with JavaScript and the demo shows the expressive power of XQuery for this purpose (e.g., drag & drop, asynchronous events, style sheets, etc.).

The architecture of XQIB is shown in Figure 2. The Web browser loads and renders Web pages and stores them internally using DOM. When the Web browser loads a Web page, it notifies the XQIB plug-in so that the plug-in can extract, compile, and execute all XQuery scripts embedded in the Web page. Furthermore, the Web browser notifies the plug-in of any other events such as mouse clicks, mouse movements, keyboard events, or requests and responses from the server. Accordingly, the plug-in executes the

XQuery functions which were registered as event handlers for each kind of event.

Again, the XQIB plug-in reuses all components of Zorba with the only noticeable exception of the Store. XQIB implements its own Store on top of the DOM of the Web browser which implements the Web page. As a result, all reads and updates are directed to the Web page which is thereby implicitly modified.

5.3 MDQ: Pay-as-you-go Data Integration with XQuery

Another XQuery-related project is called "mapping data to queries" or MDQ, for short. The idea is to make XQuery work in heterogeneous environments in which diverse data sources publish the same kind of data in different formats. A typical example are airlines which all sell plane tickets, but the structure of the flight information (e.g., XML element names) is different. The idea is to provide mapping rules that express the equivalences; e.g., a mapping rule could specify that the "flight-number" in one data source is equivalent to the "fid" in another source.

At http://fifthelement.inf.ethz.ch:8080/rules/, a demo of MDQ can be found. The details of the approach are described in a technical report [32]. Again, however, the magic of MDQ lies in modifications done to the "Store" component of an XQuery processor. All other components of an XQuery processor need not be changed. Currently, MDQ is implemented using Saxon, but we are currently working on porting it to Zorba.

5.4 XQuery Benchmark Service

As mentioned in Section 2 one of the issues of XQuery is that it is perceived to be slow. The same argument was made against the relational data model, SQL, and Java in their early days. Indeed, most XQuery processors (including Zorba) are not mature enough to show good performance in all situations. The purpose of the XQuery Benchmark Service is to help users experiment with the performance of different XQuery processors and database systems for generic, standard benchmarks and for user-defined benchmarks. The hope is that with this service feedback can be given to the vendors of XQuery processors so that they can improve their products.

Concretely, the XQuery Benchmark Service is an online service available through a Web-based interface at http://xqbench.org. In a server farm, a number of XQuery processors and database systems are pre-installed; e.g., MonetDB, Zorba, MXQuery, Saxon, etc. Furthermore, a number of benchmark datasets (with different scaling factors) and queries are pre-installed; e.g., XMark [48], TPoX [44], and a home-grown XQuery benchmark which models an online bookstore and involves complex XQuery queries and updates. With the help of the Web-based interface, users can carry out their own benchmarks. Users do this by selecting a sub-set (or all) of the pre-installed XQuery processors for the experiment. Furthermore, users select a data set for the experiment, either one of the pre-defined databases or, if desired, a user-defined database that can be uploaded through the Web-based interface. Finally, the user selects a set of queries and updates to be executed on the XQuery processors and data sets; again, the queries can be chosen among the pre-defined queries, queries uploaded by the user, or a mix. Once, an experiment has been defined by a user, the experiment is scheduled for execution on the server farm and the user is informed by E-Mail when the execution of the experiment has been completed. A Web-based report (with graphs and comparisons between the various XQuery processors) is generated using the XCheck tool [1]. Users can also search the results of previous experiments.

5.5 XQDT: XQuery Eclipse Plug-in

An important component for the success of any programming language is its support with tools, in particular, an IDE. XQDT (XQuery Development Tools) is an Eclipse plug-in for XQuery and it has all the features one would expect from an Eclipse plug-in: syntax highlighting, code outline, error handling, code completion, etc. Furthermore, XQDT provides an environment for running XQuery programs using any (user-configured) XQuery processor and debugging XQuery programs using Zorba or any other XQuery processor which implements the XQDT debug API. The debug API is similar to the debug API of Eclipse used for Java. In its latest version, XQDT also allows to test and debug Sausalito applications inside Eclipse and to deploy Sausalito projects in the cloud.

XQDT is open source and freely available under an Eclipse license. It can be installed and downloaded from http://www.xqdt.org.

5.6 XQuery on Hadoop

All the projects listed above are available and can be downloaded and tested using the URLs indicated above. A project that we just started is to integrate Zorba into Hadoop in order to provide facilities for large-scale data analysis with XQuery. The aim is to position XQuery as a standardized language for writing complete analytic programs for structured and semi-structured data. We expect to have first results on this project at the end of 2009.

6. CONCLUSION

XML is here to stay because it makes data management more flexible. We believe that XQuery is here to stay, too. XQuery is great for processing XML. XQuery is also great for many other application that do not involve XML data because XQuery makes applications more flexible. As a result, it is easier to evolve and customize XQuery applications than applications based on more traditional programming paradigms. XQuery has adopted a great deal of concepts from general-purpose programming languages such as Java or C#, scripting languages such as PHP, Python, or JavaScript, and database programming languages such as SQL. In addition, XQuery provides a number of additional features (e.g., construction of data) which cannot be found in any of these languages. Probably the biggest advantages of XQuery are its unique processing model and its data model which enables XQuery to integrate any kind of data and process any kind of data in a possible way (on the fly and in a database). As a result, XQuery is an extremely powerful tool with the potential to implement a wide variety of different applications in a single tier using XQuery only.

This paper gave an overview of XQuery-related projects carried out by a bunch of extremists from various different institutions from academia / research, open source, start-ups, and large companies. One of these projects is the Zorba XQuery processor which is embeddable into various different systems and provides a number of useful XQuery extensions and libraries. Furthermore, this paper presented a number of recent projects that demonstrated how XQuery can be used as a tool in different scenarios. The authors hope that this paper helped to correct some myths about XQuery: (a) XQuery is not necessarily slow, and (b) XQuery is not complicated.

Obviously, XQuery has not reached a break-through yet and a great deal of work is needed before XQuery technology has reached the maturity of, say, Java or SQL technologies. The hope of this paper is to encourage other people to consider XQuery as a tool and possibly get other people interested in one of the projects listed in this paper. If you are interested, please, contact any of the authors or all of us at "contact@flworfound.org".

Acknowledgments. We would like to thank the following contributors to the projects listed in this paper: Cezar Andrei, Nicolae Brinza, William Candillion, Ghislain Fourny, Martin Hentschel, Dennis Knochenwefel, Alexander Kreutz, Paul Kunz, Paul Pedersen, Markus Pilman, Gabriel Petrovay, and Daniel Turcanu. Furthermore, we would like to thank the users of the various products described in this paper. Without their feedback, bug reports, and patience, none of these products would have reached maturity.

7. REFERENCES

[1] L. Afanasiev, M. Franceschet, and M. Marx. XCheck: A Platform for Benchmarking XQuery Engines. In *VLDB*, 2006.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] S. Amer-Yahia et al. XQuery and XPath Full Text 1.0, 2008. W3C Candidate Recommendation.

[4] Apache Software Foundation. The Xerces Project. http://xerces.apache.org/.

[5] S. Aulbach et al. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *SIGMOD*, 2008.

[6] R. J. Bayardo et al. An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing. In *WWW*, 2004.

[7] K. Beyer et al. System RX: One Part Relational, One Part XML. In *SIGMOD*, 2005.

[8] S. Boag et al. XQuery 1.0: An XML Query Language, 2007.

[9] P. Boncz et al. Pathfinder: XQuery — The Relational Way. In *VLDB*, 2005.

[10] P. Boncz et al. MonetDB/XQuery: a Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, 2006.

[11] V. Borkar et al. The BEA AquaLogic Data Services Platform. In *SIGMOD*, 2006.

[12] V. Borkar et al. XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform. In *ICDE*, 2008.

[13] I. Botan et al. Extending XQuery with Window Functions. In *VLDB*, 2007.

[14] M. Brantner et al. Building a Database on S3. In *SIGMOD*, 2008.

[15] M. Brown. Get to know JsonML. http://jsonml.org/, 2007.

[16] D. Chamberlin et al. XQuery 1.1, 2008. W3C Working Draft.

[17] D. Chamberlin et al. XQuery Scripting Extension 1.0, 2008. W3C Working Draft.

[18] D. Chamberlin et al. XQuery Update Facility 1.0, 2008. W3C Candidate Recommendation.

[19] CWI. MonetDB. http://monetdb.cwi.nl.

[20] Y. Diao et al. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *TODS*, 28(4):467–516, 2003.

[21] D. Draper et al. XQuery 1.0 and XPath 2.0 Formal Semantics, 2007.

[22] M. Fernandez et al. XQuery 1.0 and XPath 2.0 Data Model (XDM), 2007.

[23] D. Florescu et al. The BEA Streaming XQuery processor. *VLDB Journal*, 13(3):294–315, 2004.

[24] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[25] D. Florescu and D. Kossmann. XML Query Processing. In *ICDE*, 2004.

[26] D. Florescu and D. Kossmann. Programming for XML. In *SIGMOD*, 2006.

[27] FLWOR Foundation. Zorba - The XQuery Processor. http://www.zorba-xquery.org.

[28] FLWOR Foundation. Zorba documentation: Rest functions. http://www.zorba-xquery.com/doc/zorba-latest/zorba/html/rest.html.

[29] A. Fomichev, M. Grinev, and S. D. Kuznetsov. Sedna: A Native XML DBMS. In *SOFSEM*, 2006.

[30] G. Fourny et al. XQuery in the Browser. In *WWW*, 2009.

[31] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[32] M. Hentschel et al. Mapping Data to Queries: Semantics of the IS-A Rule. Technical report, ETH Zurich, 2007.

[33] IBM. XML Database - DB2 pureXML. http://www.ibm.com/software/data/db2/xml/.

[34] Institute for System Programming RAS. Sedna: Native XML Database System. http://modis.ispras.ru/sedna/.

[35] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *ICDE*, 2000.

[36] M. Kaufmann and D. Kossmann. Developing an Enterprise Web Application in XQuery. http://www.28msec.com/tech_reading.html, 2008.

[37] M. Kay. Saxon: The XSLT and XQuery processor. http://saxon.sourceforge.net/.

[38] M. Kay. Ten Reasons Why Saxon XQuery is Fast. *IEEE Data Engineering Bulletin*, 31(4):65–, 2008.

[39] D. Kossmann. Building Web Applications without a Database System. In *EDBT*, 2008.

[40] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. In *SIGMOD*, 2008.

[41] Z. H. Liu, A. Novoselsky, and V. Arora. Towards a Unified Declarative and Imperative XQuery Processor. *IEEE Data Engineering Bulletin*, 31(4):33–40, 2008.

[42] A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, 2003.

[43] J. Melton and S. Muralidhar. XML Syntax for XQuery 1.0 (XQueryX), 2007.

[44] M. Nicola, I. Kogan, and B. Schiefer. An XML Transaction Processing Benchmark. In *SIGMOD*, 2007.

[45] P. O'Neil et al. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.

[46] N. Onose and J. Simeon. XQuery at your Web Service. In *WWW*, 2004.

[47] F. Özcan, N. Seemann, and L. Wang. XQuery Rewrite Optimization in IBM DB2 pureXML. *IEEE Data Engineering Bulletin*, 31(4):25–32, 2008.

[48] A. Schmidt et al. XMark: a Benchmark for XML Data Management. In *VLDB*, 2002.

[49] J. Schneider and T. Kamiya. Efficient XML Interchange (EXI) Format 1.0, 2007. W3C Working Draft.

[50] K. Sheykh Esmaili, P. M. Fischer, and J. Simeon. XQuery 1.0 Web Services Facility (Proposal). https://www.dbis.ethz.ch/research/publications/WSDL-SOAP_Proposal.pdf, 2008.

[51] J. Snelson. Parsing JSON into XQuery. http://snelson.org.uk/archives/2008/02/parsing_json_in.php, 2008.

[52] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[53] Systems Group, ETH Zurich. MXQuery A lightweight, full-featured XQuery Engine. http://www.mxquery.org.

[54] I. Tatarinov et al. Storing and Querying Ordered XML using a Relational Database System. In *SIGMOD*, 2002.

[55] H. S. Thompson et al. XML Schema Part 1: Structures Second Edition, 2004.

[56] P. Walmsley. *XQuery*. O'Reilly, 2007.