XBird/D: Distributed and Parallel XQuery Processing using Remote Proxy

Makoto Yui†, Jun Miyazaki†, Shunsuke Uemura‡, and Hirokazu Kato†
{makoto-y|miyazaki|uemura|kato}@is.naist.jp

† Graduate School of Information Science, Nara Institute of Science and Technology
Keihanna Science City, Nara 630–0192, Japan

‡Faculty of Informatics, Nara Sangyo University
Sango–cho, Nara 636–8503, Japan

ABSTRACT

In this paper, we focus on an aspect of distributed XQuery processing that involves data exchanges between processor elements. We first address problems of distributed XML query processing and explain how the problems differ from traditional database problems. Then, in order to achieve efficient and transparent data exchange, we adopt the use of *remote proxy*, in which each shipped data is wrapped in a proxy sequence, and the proxy sequence is returned to the remote peer. When accessing the proxy sequence, actual results (possibly partial results) are fetched from a data provider, and then the data provider evaluates its entity sequence in a *call-by-need* fashion. Our scheme allows parallel query execution and reduces network traffic and redundant buffer utilization by exchanging required data directly between a consumer and a provider.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—
Distributed databases, Query processing

General Terms

Design and Implementation of Distributed Database

Keywords

Distributed and parallel database, XML query processing, Lazy evaluation

1. INTRODUCTION

As a result of wide adoption of XML, XML data has been spread over computer networks. It has produced the need to integrate distributed and dynamic XML documents. For example, users might want XML feed-readers to show more fresh and/or on-the-fly information (e.g., recent disaster information or latest results of sport games) through their thousands of RSS/ATOM subscriptions. However, current feed-readers do not aggregate their subscriptions in real-time but at hourly intervals (Bloglines [1] currently checks subscriptions once an hour). Another example is found in integration of biological databases. Most biological databases today have the ability of publishing XML to support integrations among heterogeneous data-sources, and each of them receives frequent updates/corrections from individual laboratories around the world. Therefore, integration systems of biological databases must take data freshness into account.

Considering such situations, it is doubtful that on-the-fly processing for thousands of XML data is realistic. It would be impossible with current XML query processor technologies. Because, as far as we know, there is no XML query processor tackling both inter-operator parallelism [2] and distributed query processing, both of which are indispensable to solving the underlying problems as explained below.

To achieve on-the-fly XML query processing of thousands of XML data that cannot be processed by an XQuery processor on a single computation node, we apply a divide-and-conquer approach that divides a query into sub-queries, and then, executes these sub-queries on multiple computation nodes as in Figure 1. However, such a hierarchical distributed system must deal with the following technical issues:

- First, partitioned computation requires that query processors exchange intermediate results in order to produce its final result. The time for exchanging intermediate results cannot be ignored. In particular, data exchanging of XQuery Data Model (XDM) [3] instances requires long CPU time because, in contrast to relational databases, XML databases must deal with non-scalar data types such as XML trees, which basically treat scalar data types. Therefore, encoding and decoding of XDM instances, in general, spend more CPU time than those for a relational model. According to the findings in [4], parsing an XML document typically takes 175K CPU instructions per kilobyte, which is as of the same order as inserting a row into a relational table (30K to 200K instructions). Hence, each edge of two operators can potentially be a bottleneck.

- The second issue is that CPU-utilization of current query processors cannot exploit inter-operator parallelism in addition to divide-and-conquer parallelism. Regarding the divide-and-conquer strategy, a sub-query q_a can execute its sub-queries $(q_h \cdots q_n)$ in parallel and even out-of-order (see Figure 1).

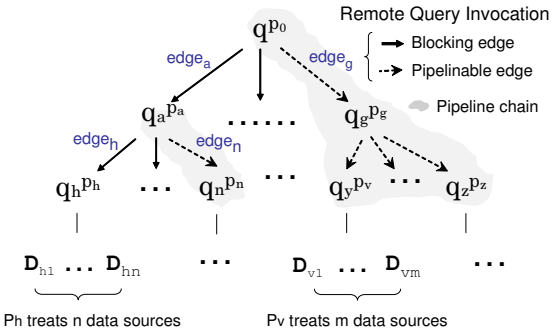
  Here, we use the notations $T(q_a)$, $LT(q_a)$ and $T(edge_a)$ for elapsed time of a query q_a at peer p_a, that of lo-

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

cal query processing of q_a, and that at an edge $edge_a$, respectively. When we do not consider pipeline parallelism, $T(q_a)$ is recursively defined as follows:

$$T(q_a) = \underbrace{max((T(q_h) + T(edge_h)), \cdots, (T(q_n) + T(edge_n)))}_{\text{elapsed time of the most time-consuming edge}} + LT(q_a)$$

According to this formula, computation time of a node depends on the most time-consuming edge. Taking Figure 1 as an example, local query processing of q^{p_0} is blocked and its CPU resource tends to be idle until the last intermediate result is returned. Moreover, the non-parallelized part of queries, e.g., $LT(q_a)$ in the above example, restricts the theoretical maximum speedups. According to Amdahl's law [5], the expected speedups by parallel query execution are often limited by the non-parallelized part. Therefore, to leverage the computation power of current multi-processors including multi-core processors, pipelining is indispensable.



A user query q is divided into subqueries $q_a, \cdots, q_z$ recursively. The symbols $p_0, p_a, \cdots, p_h, \cdots, p_z$ denote the peers in which a query is executed. The symbol D such as D_{hn} represents a data-source.

Figure 1: Divide-and-conquer and pipeline parallelism

Considering the above aspects, in this paper, we focus on the data exchanges between processor elements in distributed XQuery processing, which so far have not been carefully discussed in the literature. As an alternative to previous methods, we propose an efficient data exchange method using *remote proxy*, in which each entity or result sequence is wrapped in a proxy sequence and the proxy is returned to the remote peer. When accessing the proxy sequence, actual results (possibly partial results) are fetched from the data provider. The use of *remote proxy* brings the following three advantages:

- Our scheme allows parallel query execution, which supports several kinds of parallelisms, e.g., independent-operator and pipeline parallelism.
- Entity sequences are computed in a demand-driven manner. A new FIFO entry of the entity sequence [1] is requested when the entry is consumed and dropped to the low watermark level. Its sophisticated and built-to-order mechanism utilizes the server's resources such as memory and CPUs.
- Our method can reduce network traffic and redundant buffer occupation by directly exchanging required data between a consumer and a provider, which are mandatory for the previous *pass-by-value* data exchanges [6, 7, 8]. Avoiding intermediary trades reduces both network traffic and network latency as well as redundant computations such as encoding and decoding in mediator nodes.

We have implemented the proposed method on the top of our practical implementation of a native XML database system [2]. Our experimental results show up to 22x speedups compared with competing methods. We demonstrate the importance for distributed XML database systems to take *pass-by-reference* semantics into (re-)consideration.

The rest of this paper is organized as follows: Section 2 introduces related works and identifies open problems of distributed XML query processing, and then, we briefly mention our solutions for the problems. In Section 3, we describe details of our implementation including our language extension to XQuery and distributed query optimizations. In Section 4, we provide experimental results and their evaluation, and conclude in Section 5.

2. RELATED WORK AND OPEN PROBLEMS

In this section, we address the open problems of distributed XQuery processing in previous works, and propose our solution for these problems. We also refer to related works.

Figure 2 depicts a typical remote query execution flow with *pass-by-value* semantics. *Pass-by-value* semantics has been used for the evaluation strategy of previous distributed XML query processors [6, 7, 8]. The problems underlying these strategies are as follows:

- Limited inter-operator parallelism
  With a *pass-by-value* evaluation strategy, a remote query is executed sequentially as in Figure 2. Therefore, during processing on Server A, Server B becomes idle. In contrast, Server A tends to be idle while Server B processes a query. In addition, as we mentioned with Figure 1, current XML query processors using *pass-by-value* strategy cannot exploit pipeline parallelism, and thus, inter-operator parallelism is limited for nested query executions. We examined how long of a query processing time it takes to execute the following query at a remote peer where $doc locates an XML document of scale factor (SF) 5 or 10 of XMark [9]. The transferred data size of the resulting XDM instances were 2164 KB (SF=5) and 4307 KB (SF=10).

```
for $a in $doc/site/closed_auctions/closed_auction
where $a/price/text() >= 40 return $a/price
```

  The result, as in Figure 3, shows that, at least in our experience, the latency including encoding and decoding is the as same as the query execution time at a remote peer. This supports our claim that each edge of Figure 1 can be a potential bottleneck. Moreover, since each edge becomes a blocking edge (see Figure 1), it remains a limited inter-operator parallelism with the *pass-by-value* strategy. To solve this problem, we

[1] In XDM [3], a result is a sequence containing zero-or-more items. The items are consumed by an operator (iterator) in a FIFO manner. We call the remaining items "FIFO entry."

[2] It will be released at:
http://db-www.naist.jp/~makoto-y/proj/xbird/

take an advantage of pipeline parallelism and inter-operator parallelism into our *remote proxy*. Due to the *remote proxy*, our systems can make each edge of Figure 1 pipelinable, and thus, all computation nodes can theoretically run in parallel.

- Poor resource utilization
  In a multi-user environment, multiple concurrent queries consume lots of system resources. Thus, it is important to allocate adequate CPU and memory resources especially under current multi-processor or multi-core architectures. For example, selecting low degrees of an operator parallelism can lead to under-utilization of the system and reduce throughput. On the other hand, high degrees of parallelism can spend "too many" resources on one query and lead to high resource contention. With the current *pass-by-value* strategy, such resource contentions frequently occur, for instance, at A and B in Figure 2. Suppose that encoding entire parametric sequences at A, Server A consumes lots of CPU cycles and memory space, and then remote query execution at Server B is blocked until the encoding finishes. On decoding receiving parameters at Server B, Server B may suffer from less available memory. Such a situation actually happened in our later experiment in reality (see Section 4). To deal with the problem, we propose an efficient resource utilization scheme using *remote blocking-queue* with which processing rates of operators are managed.

- Encoding and decoding overhead
  Previous research used to take XML formats for data-exchanges between processor nodes [6, 7, 8]. However, as mentioned in [4], decoding XML inputs consumes lots of CPU cycles. In [10], Bayardo et al. asserted that binary encoding of XML would appear to provide performance benefits to most applications without any significant drawbacks other than compromising a view-source principle. On the contrary, a naive (blocking) binary encoding may prevent pipelined XML stream processing. We thus take an incremental encoding/decoding scheme that incrementally converts an XDM instance to a SAX-like event (binary) stream. In addition, we propose an efficient *direct result forwarding* mechanism for the *pass-by-reference* in Section 3.

3. IMPLEMENTATION OF XBIRD/D

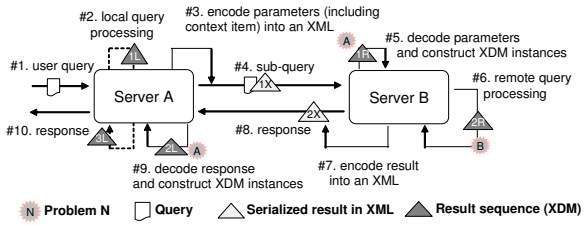
3.1 The Language Extension: BDQ

XBird/D extends the XQuery language [11] to support remote query execution. We call the extension for ***XBird Distributed Query***, *BDQ* for short. Figure 4 describes the grammatical extension to *PrimaryExpr* of XQuery. *BDQ-Expr* means that $Expr_2$ is to be executed at a remote peer $\mathcal{P}$, where the endpoint of $\mathcal{P}$ is *fn:string*(*fn:exactly-one*($Expr_1$)). The endpoint takes a URL format of the form: *//host:port/name* where *name* is the binding name of remote service that is bounded at the service registry identified with the pair of *host* and *port*.

3.2 Remote Proxy

As mentioned in Section 2, our distributed XQuery processor *XBird/D* employs *remote proxy* to achieve an inter-operator parallelism.

The base (single) XQuery processor, which is noted as *XBird*, is based on the *Volcano* iterator model and employs an iterative query processing model based on an iterator



Server A accepts a query including *BDQExpr* (explained later Section 3.1)[1], then a local query processor evaluates its query[2]. At the evaluation of *BDQExpr*, its computed parameters $1L$ and compiled expression are shipped (i.e., encoded[3] and transferred[4]) to the remote Server B. Then, Server B decodes the encoded parameters $1X$ and constructs its XDM instance $1R$[5]. After that, Server B evaluates the received compiled expression[6]. Then, the intermediate result $2R$ at Server B is returned to the Server A[7,8]. Server A decodes the response $2X$ of Server B and constructs its XDM instance $2L$[9]. Finally, Server A continues to evaluate the entire expression[2] and return the result $3L$ to a user program[10].

Figure 2: A typical remote query execution flow with pass-by-value semantics

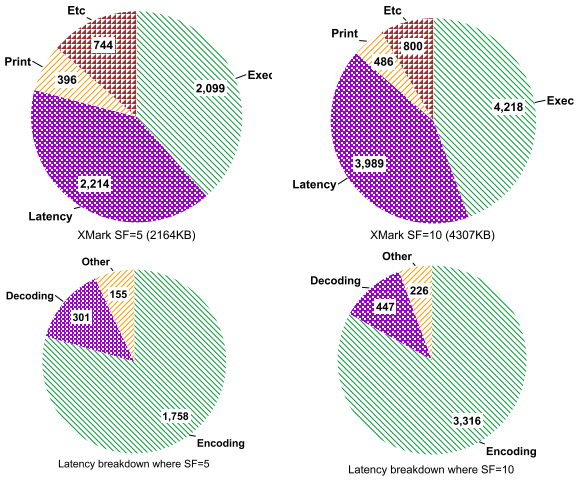
tree, which is similar to the BEA/XQRL XQuery processor [12] in which pipelined processing operators deal with loops, axis accesses, etc. The iterator model allows lazy evaluation of expressions, and also plays an important role in *XBird/D* architecture.

We describe how the *pass-by-reference* evaluation semantics is achieved by using *remote proxy*. *Remote proxy* is not a unique feature for *XBird/D*. It is an extension of the well-known proxy design pattern for distributed object communications [13], and was not developed in the context of distributed XML query processing. It is impossible to accomplish inter-operator parallelism only with *remote proxy*. Therefore, we try to use a combination of *remote proxy* and an asynchronous entity production of intermediate results, as is described next.

Asynchronous Production and Queue Management

We assume that items in a sequence are stored in FIFO queue in which each item is consumed from an another operator processed by a consumer.

Figure 5 gives an overview of our *remote proxy* implementation. When executing a remote query, the intermediate result (*result entity sequence*) is wrapped with a proxy and the proxy object (*result proxy sequence*) is returned to the caller ($Peer_1$). Then, the remote operator asynchronously produces the items of the *result entity sequence* until the queue becomes full. When the queue becomes full, the producer thread is blocked until space becomes available in the queue$^{(a)}$. On the other hand, retrieving items is blocked until the queue becomes non-empty$^{(c)}$. The *remote proxy* fetches remote entries$^{(c)}$ when the local queue is empty at timing (b). The size of fetched items can be configured for each query by specifying initial fetch size and its growth factor. The fetched size automatically grows according to the parameters up to a specified threshold. This feature aims at reducing client/server communications. Due to the advantages of this simple but effective queue management, relying on *remote blocking-queue*, our system can utilize system resources by avoiding oversupply and undersupply.



Exec: Elapsed time for remote query execution at a remote peer.
Latency: Latency of remote query execution including encoding/decoding and network latency.
Print: Elapsed time of serializing an XDM instance to a file.
ETC: Elapsed time including compilation and other (local) query execution footprints.

Figure 3: Breakdown of remote query processing time (in msec.)

$$\begin{array}{rcl} \mathit{BDQExpr} & ::= & \text{“}\mathit{execute\ at}\text{”}\ \mathit{Expr}_1\ \text{“}\{\text{”}\ \mathit{Expr}_2\ \text{“}\}\text{”} \\ \mathit{PrimaryExpr} & ::= & \mathit{Literal} \mid .. \mid \mathit{Constructor} \mid \mathit{BDQExpr} \end{array}$$

Figure 4: BDQ grammar extensions

Direct Result Forwarding

As we have already mentioned in Section 2, *XBird/D* has a *direct result forwarding* feature to reduce latency, e.g., encoding/decoding and network latency, in the distributed query execution. We explain how a query is processed in distributed and nested query execution in detail using Figure 6. In Figure 6, *filter*, *reduce*, *select1* and *select2* functions are executed at PE1, PE2, PE3 and PE4, respectively. The left half of Figure 7 expresses the nested operation tree. The *reduce* function collects closed_auction and open_auction and returns the first 1000 items for each. Since this *reduce* function does not cause local resource access (*fn:doc* and/or *fn:collection*), the execution is relocatable. This optimization is performed not at the compilation time but at execution time [3]. Since our XQuery processor is implemented based on the *Volcano* iterator model, the result iterators are calculated by lazy evaluation in call-by-need fashion. For the relocation of execution, we just forward the iterators P1 and P2 to the upper operator (on PE1, in this case), and evaluate them on PE1. The intermediate results are fetched from PE3 and PE4 directly. This optimization is effective since encoding and decoding on PE2 can be avoided. Recall that it takes the majority of total elapsed time in remote query execution for encoding and decoding (see Figure 3).

A previously proposed system [8] can use an *intensional* expression that was originally proposed in [14]. The intensional answer can make a server shift its work to a client by mutating the result expression with the intermediate results. For example, the intensional expression transfers the computation of the *reduce* function by mutating the result

[3]The advantage of this dynamic relocation is that it can take execution time information into consideration.

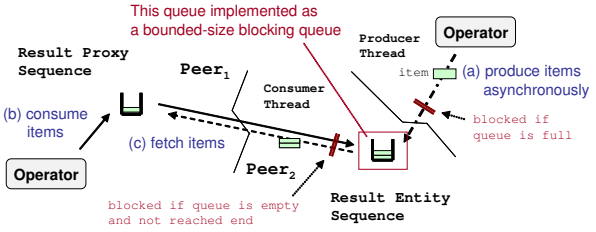


Figure 5: Server/Client interaction between processor elements using a remote proxy

expression as follows:

```
fn:sequence( (<closed_auction> ... </closed_auction>,
              <closed_auction> ... </closed_auction>), 1, 1000),
           | (<open_auction> ... </open_auction>,
              <open_auction> ... </open_auction>), 1, 1000) )
```

However, according to our experience, the benefit of an intensional answer is limited where the result is small, since it requires additional (and expensive) encoding and decoding of the intensional results. The encoding and decoding often waste more server resources than evaluation of a query. In contrast, our iterator forwarding can receive the benefits of lazy evaluation without such a significant drawback.

```
declare function bdq:select1() {
  execute at $PE3 {
    fn:collection($col)/site/closed_auctions/closed_auction
  }
};
declare function bdq:select2() {
  execute at $PE4 {
    fn:collection($col)/site/open_auctions/open_auction
  }
};
declare function bdq:reduce() {
  execute at $PE2 {
    (  fn:subsequence(bdq:select1(), 1, 1000)
     | fn:subsequence(bdq:select2(), 1, 1000) )
  }
};
declare function local:filter() {
  for $a in bdq:reduce()
  where $a/seller/@person >= "person10000"
     or $a/buyer/@person >= "person10000"
  return $a
};
local:filter()  (: execute at PE1 :)
```

Figure 6: Nested remote query execution example

4. EXPERIMENTAL EVALUATION

In order to evaluate the effectiveness of our enhancements, i.e., *remote proxy* and *direct result forwarding*, we conducted performance comparisons to MonetDB/XRPC [7] version 4.18.1, which is one of the state-of-the-art distributed XQuery processors.

As the experimental environment, we used four PCs. We denote the XQuery processor running on each PC as PE1 ... PE4. Each PC consists of Pentium D 2.8GHz CPU, 2GB of memory, SuSE Linux 10.2 as an OS, and JDK 1.6 as a runtime system, connected on 1Gb/s Ethernet, except for PE2, which is equipped with an Athlon 64 X2 2.4GHz CPU. We used a query in Figure 6 for the evaluation of *XBird/D* and the equivalent query for *MonetDB/XRPC*. As for the data set, we used a 1.1GB XML document generated by the data-generator of XMark [9] where the scale factor was set

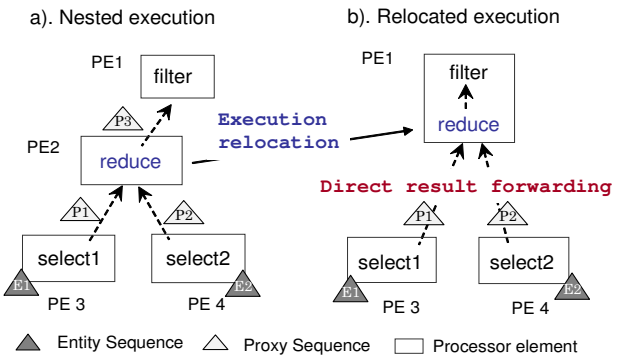


Figure 7: Execution relocation and direct result forwarding

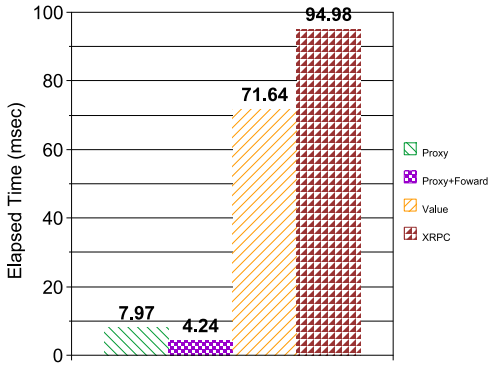


Figure 8: Comparison of four evaluation strategies

to 10. We bound the generated document to variable $col in Figure 6. The documents were loaded to each database instance in advance on PE3 and PE4 where both *MonetDB* and *XBird* were running.

The summarized results are shown in Figure 8. We compared four evaluation strategies including *remote proxy* (Proxy), *remote proxy* with *direct result forwarding* (Proxy+Forward), our implementation of *pass-by-value* semantics (Value), and *MonetDB/XRPC* by *pass-by-value* semantics [7] (XRPC).

As in Figure 8, our *pass-by-reference* implementation using *remote proxy* shows a significant improvement on the elapsed time. This is because our system could eliminate unnecessary computation at PE3 and PE4, as a result of applying lazy evaluation to distributed XQuery processing. The remote query evaluation by *pass-by-value* semantics computed and produced the entire results at one time, while not all of them are used in the later computation. It is clearly inefficient, and explains why our *remote proxy* evaluation strategy (Proxy) executed about 9 times faster than our *pass-by-value* evaluation strategy. Moreover, *direct result forwarding* eliminated the redundant encoding/decoding on PE2 and the overhead of mediated communications. As a result, our system could finally obtain about 22 times better performance than the competitive method (XRPC).

In addition, only our system using *remote proxy* can process 100 concurrent query requests using 30 threads in 160 seconds where the maximum and average elapsed times are 53.76 and 36.75 seconds, respectively. Both of the *pass-by-value* semantics implementations (Value and XRPC) suffer from a frequent swap-in/out [4] due to poor resource utilization. We, thus, confirm the advantage of our methods in a certain situation.

5. CONCLUSION

In this paper, we proposed an efficient distributed XML query execution strategy using *remote proxy*. Our experimental results show up to 22x speedups compared with competitive methods, and demonstrated the importance for distributed XML database systems to take *pass-by-reference* semantics into consideration. Furthermore, our enhancements (asynchronous production managed by *remote blocking-queue*) can utilize system resources efficiently with supporting inter-operator parallelisms.

Issues to be explored include dynamic execution dispatching of remote query processors taking system resources and utilizations (e.g., CPU utilizations, free memory space, and specs) of the participating nodes into account, and development of a selection model of execution strategies.

[4]XRPC did not return the first response of 5 concurrent queries in 10 minutes.

Acknowledgment

This work was partly supported by the CREST Program of JST, MEXT (Grant-in-Aid for Scientific Research on Priority Areas #19024058), and JSPS (Grant-in-Aid for Young Scientists (B) #19700094).

6. REFERENCES

[1] Bloglines. http://www.bloglines.com/.
[2] Tamer M. Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd ed.)*. Prentice Hall, 1999.
[3] W3C. XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/.
[4] M. Nicola and J. John. XML Parsing: A Threat to Database Performance. 2003.
[5] Gene M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. Morgan Kaufmann Publishers Inc., 2000.
[6] Christopher Ré, Jim Brinkley, Kevin Hinshaw, and Dan Suciu. Distributed XQuery. In *Proc. IIWeb*, pages 116–121, 2004.
[7] Y. Zhang and P. A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *Proc. VLDB*, 2007.
[8] Mary F. Fernandez, Trevor Jim, Kristi Morton, Nicola Onose, and Jerome Simeon. DXQ: A Distributed XQuery Scripting Language. In *Proc. XIME-P*, 2007.
[9] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.
[10] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki. An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing. In *Proc. WWW*, pages 345–354, 2004.
[11] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.
[12] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, and Arvind Sundararajan. The BEA streaming XQuery processor. *VLDB Journal*, 13(3):294–315, 2004.
[13] Hans Rohnert. *The proxy design pattern revisited: Pattern languages of program design 2*. Addison-Wesley, Inc., 1996.
[14] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Exchanging Intensional XML Data. volume 30, pages 1–40, 2005.