Distributed XQuery

Christopher Re* James Brinkley*,† Kevin P. Hinshaw† Dan Suciu*

1 Motivation

XQuery is increasingly being used for ad-hoc integration of heterogeneous data sources that are logically mapped to XML. For example, scientists need to query multiple scientific databases, which are distributed over a large geographic area, and it is possible to use XQuery for that. However, the language currently supports only the *data shipping* query evaluation model (through the `document()` function): it fetches all data sources to a single server, then runs the query there. This is a major limitation for many applications, especially when some data sources are very large, or when a data source is only a virtual XML view over some other logical data model. We propose here a simple extension to XQuery that allows *query shipping* to be expressed in the language, in addition to data shipping.

Example 1.1 For a simple illustration, consider the following example:

```
Q1:
 for $x in document("X.xml")/a/b[@c="123"],
  $y in
   document("http://Y.com/Y.xml")/d/e[@f=$x/@g]/h
  return <answer> <x> { $x/u } </x>
                  <y> { $y/v } </y>
         </answer>
```

This XQuery performs a join between two documents `X.xml` (residing on the local server) and `Y.xml` (on a remote source). The query can be executed today by any standard XQuery processor. However, the problem is that the query processor implements data shipping only: it reads the entire file `Y.xml`, only to extract a small fragment. Worse, when `Y.xml` is not a real XML document but an XML virtual view over, say, a relational database, then this query cannot be executed at all, because relational databases often do not export the entire XML view. For example, SQL Server supports simple XPath expressions over a relational database, but will not export a complex, multi-table database in XML.

In our proposed extension, XQueryD, the same query can be expressed using a new expression, `execute`:

```
Q2:  for $x in document("X.xml")/a/b[@c="123"]
     let $y := (execute at "http://Y.com"
                xquery {
                 for $z in document(Y.xml)/d/e[@f=$x/@g]
                     return $z
                   })/h
     return <answer> <x> { $x/u } </x>
                     <y> { $y/v } </y>
            </answer>
```

The new `execute` expression instructs the processor to send a subquery to `Y.com` and wait for the result. The remote site has to execute a normal XQuery expression, and return only the answer to that query (as opposed to the entire document). For example, if the value of `$x/@g` is `"456"`, then the following query is shipped to `Y.com`:

```
for $z in document("Y.xml")/d/e[@f="456"]
return $z
```

This is a simple XPath expression supported by any XML data source. Once it returns to the original server, the result is interpreted as an XML fragment and the variable `$y` is bound to its `h` children.

In this paper we propose a simple extension to XQuery that supports the query shipping paradigm in addition to the data shipping paradigm. The extension consists of a single new expression:

```
execute at <URL>
xquery { <EXPR> }
( handle <VAR>:<NAME-SPACE> <EXPR> )*
```

We call the resulting language XQueryD. In this paper we illustrate the power of this simple construct through several examples, describe the exception-handling mechanisms (`handle`), describe a simple implementation, and discuss some source-to-source optimizations. We also illustrate with a real application from the University of Washington Human Brain Project [BMP+97, RM03].

1.1 Related Work

While simple, the language extension we propose distills several ideas from distributed databases and process calculi, which deserve a brief discussion here.

Optimizer vs. Language Approach Existing techniques in distributed query optimization can transform a declarative query like `Q1` into a distributed execution plan with query shipping, much

* Department of Computer Science, University of Washington.

† Structural Informatics Group, Department of Biological Structure, University of Washington

like Q2 [Kos00, Won00]. One does not need a language extension in order to be able to run distributed queries efficiently. Our proposal goes against the data independence principle, requiring users to be aware of the physical data distribution. However, in order to support true data independence one needs a sophisticated query processor with a distributed optimizer, and one also needs central control over all data sources. This is definitely not possible in our intended application: the exchange of scientific data. In this case scientists use publicly available XQuery query processors, which do not have distributed optimizers. Even if distributed optimizers were widely available, they would still not be of immediate use for the scientists, since the remote sources are outside of their control and thus are unlikely to expose all the information required by the query optimizer (source capabilities, statistics, etc). On the other hand, scientists are rather savvy users: they won't mind writing queries in XQueryD if this allows them to retrieve results immediately, rather than wait for a sophisticated infrastructure to be deployed. A final appeal of our light-weight approach is the ability to handle exceptions, which are common in a distributed query environment, see Sec. 2.2.

Optimizations Our language-based approach does not preclude optimizations, however. Instead, it makes them much easier to implement, as source-to-source query rewritings, focused around the execute expression: we illustrate this in Sec. 5 and in the Appendix. Importantly, the distributed optimizer is no longer a prerequisite to achieve query shipping, since this is already expressed in the language.

Process calculi Our approach is mostly inspired by the work on process calculi and their application to database queries [ST01, GM03]. Unlike ubQL [ST01], we use only one communication primitive, namely the migration operator execute, and omit channels and pipelining. Our emphasis is on obtaining the highest benefits by having to do as few changes to the language as possible.

Active XML This is a project for Webservices developed recently at INRIA [ABC$^+$03, ABM04, ABC$^+$04]. In the Active XML data model, an XML tree may contain on its leaves calls to Webservices which return other XML fragments, possible containing more Webservice calls. Evaluating a query over an Active XML document naturally leads to distributed execution. XQueryD differs from Active XML in several ways. In Active XML the data needs to be modified by inserting Webservice calls: this is perfectly reasonable in commercial systems, but it is impractical in scientific data exchange. Queries in Active XML are very simple, since users (in this case end-customers) only see plain XML data, while the distribution is handled completely by the system. By contrast, in XQueryD the data does not need to be modified, while queries require detailed knowledge of the sources and their capabilities: again this is better suited in the scientific domain. The technical challenges also differ: queries may not terminate in Active XML (although in practice this is rarely happens), while queries are guaranteed to terminate in XQueryD.

2 XQueryD

The single extension to XQuery is the execute expression shown in Sec. 1. We first illustrate execute with some examples, then describe the exception handling mechanism (the handle clause), and finally illustrate how to invoke foreign languages.

2.1 Execute

Dependent Joins The query Q2 in Example 1.1 contains a *dependent join*: for each value of the $x variable, a query is sent to the remote site. The disadvantage is that the number of queries executed on the remote site may be arbitrarily large, depending on the data.

Joins Alternatively, we may express the same query Q2 as join:

```
Q2':
let $to_ship :=
   (for $x in  document("X.xml")/a/b[@c="123"]
    return <x_at_g>$x/@g</x_at_g> <x_u>$x/u</x_u>)
return
  execute at "http://Y.com"
  xquery
   { for $t in $to_ship
     return
       <answer>
         <x>  { $t/x_u/child::node() } </x>
         <y>  { for $z in document("Y.xml")
                    /d/e[@f=$t/x_at_g/child::node()]
                return $z/h})/v
              }
         </y>
       </answer>
    }
```

Here only one query is sent to Y.com, making it more efficient than Q2, but it is also harder to write and harder to read. It is possible, however, to translate automatically Q2 to Q2' using simple rewrite rules: we show this in the appendix.

Joins over three sources Consider now:

```
Q3:
for $x in document("X.xml")/a/b[@c="123"]
let $y := (execute at "http://Y.com"
           xquery
            {for $z in document("Y.xml")/d/e[@f=$x/@g]
             let $u :=
                execute at "http://Z.com"
                xquery
                { for $w in document("Z.xml")/m/n[@p=$z/@q]
                  return $w/v[@u=$x/@z]
                }
             return
               <h> <z> { $z } </z>
                   <u> { $u } </u>
               </h>}
          )/h
 return <answer> <x> { $x/u } </x>
                 <y> { $y   } </y>
        </answer>
```

As before, a subquery is sent repeatedly to `Y.com`. However, this subquery contains in turn an `execute` statement, instructing `Y.com` to send a query to `Z.com`.

2.2 Exceptions

It is important to react to unforeseen events when evaluating queries in a distributed environment, such as time outs, server down, authorization failures, syntax errors, etc. XQueryD has a simple exception mechanism. An exception is defined by a namespace, and the `handle` clause specifies how to handle that specific exception. Each exception may return parameters, which can be arbitrary XML values (of a schema defined by that particular exception) and are bound to a variable. The following example illustrates this:

```
Q4:  for $x in document("file.xml")/a/b
     let $y := execute at "Y.com"
       xquery { . . . . }
       handle "www.cs.washington.edu/xqueryd/timeout"
        { <result> 123 </result> }
       handle "www.cs.washington.edu/xqueryd/serverdown"
        { execute at "backup.site.com"
           xquery { . . . .}
        }
       handle $e: "www.cs.washington.edu/xqueryd/syntaxerror"
        { <result>
           <line> { $e/line/text() } </line>
           <errorcode> { $e/errorcode/text() } </errorcode>
          </result>
        }
     return . . .
```

Exception names are namespaces, and are shown here as URLs; in general, XQuery namespaces can be used. The first `handle` clause checks for a `timeout` exception: if this happens, then a default result of `123` is returned. The second `handle` checks for a `serverdown` exception: in this case another remote XQuery is executed, at a backup site. Finally, the third `handle` checks for a syntax error. This exception has a parameter, which is bound to the variable `$e`. The result returned in this case includes the line where the error occurred and the error code, both extracted from the parameter `$e`.

2.3 Foreign

Finally, the `execute` statement can be used to invoke queries in a foreign language, other than XQuery. We found this to be necessary in order to integrate data from sources that do not have XML wrappers. The statement becomes:

```
execute at "site.com"
foreign { . . . . }
```

Any string can be included after `foreign`, and it will not be parsed by the XQuery parser. The pre-processor still substitutes variables, but now is limited to substituting only text values: any other value will generate a runtime error. We will uses `foreign` in the next section.

3 Application to the UW Human Brain Project

Our proposal is motivated by a real application from the University of Washington Human Brain Project [BMP+97, RM03]. This project seeks to develop tools to help neuroscientists understand language organization in the brain. The primary approach is to develop methods for accessing and integrating multiple types of brain mapping data located in multiple data sources. We illustrate here with three sources that were particularly difficult to integrate because they use totally different data models: a relational database, an ontology, and an XML file.

Source 1: CSM (Cortical Stimulation Mapping) This is a patient-oriented relational database stored in MySQL and recording data obtained at the time of neurosurgery for epilepsy. The data represents the cortical locations of language processing in the brain (detected by noting errors made by the patient during electrical stimulation of those areas). One piece of information in this database is the cortical locations, e.g. *middle part of superior temporal gyrus*. We have built an XML wrapper over CSM using SilkRoute [FKM+02, TKL+03], which converts automatically all XQuery expressions to SQL and returns results as XML. To illustrate, the XQuery below finds the names of all structures over all patients, in which a CSM error of type 2 (semantic paraphasia) occurred at least once in one patient:

```
Q5:
<results>
 {for $trial in PublicView("Scrubbed.pv")
             /patient/surgery/csmstudy/trial
  where $trial/trialcode/term/abbrev/text()="2"
  return $trial/stimsite/name()
 }
</results>
```

Here `PublicView` indicates the file containing the mapping from the relational database to XML. SilkRoute translates the query automatically into a complex SQL statement:

```
SELECT stimsite.name
FROM  trial, csm, . . ., stimsite
WHERE term.type = 'CSM error code' AND abbrev = '2' AND  . . .
```

Source 2: FMA (Foundational Model of Anatomy) This is an ontology representing a large semantic network [MBR03] of the entire human anatomy. It is processed by a server called OQAFMA, which accepts queries written in StruQL [FFLS00], and returns results in XML. For a simple illustration, the StruQL query below answers the following: what is the *middle part of the superior temporal gyrus* part of?

```
Q6: WHERE
        Y->":NAME"->"Middle part of superior temporal gyrus",
        X->"part"*->Y,
        X->":NAME"->Parent
    CREATE  Concept(Parent);
```

The answer is:

```
<results>
 <Concept> <Ancestor>Neocortex</Ancestor>
 </Concept>

 <Concept> <Ancestor>Telencephalon</Ancestor>
 </Concept>

 <Concept> <Ancestor>Forebrain</Ancestor>
 </Concept>

 <Concept> <Ancestor>Temporal lobe</Ancestor>
 </Concept>

 <Concept> <Ancestor>Nervous system</Ancestor>
 </Concept>

 <Concept> <Ancestor>Superior temporal gyrus</Ancestor>
 </Concept>

  . . . . . .

</results>
```

Source 3: Image Manager This is a single XML document containing collections of images. Each image can have associated with it one or more annotation sets, consisting of one or more annotations. An annotation consists of a closed polygon specified by a sequence of image coordinates on the image and an anatomical name. We use Galax [FS02] to query IM. For example the query "Find all images annotated by the *middle part of the superior temporal gyrus*" is expressed as:

```
Q7:  for $image in document("image_db.xml")//image
     where $image/annotation_set/image_annotation/name/text()
             ="middle part of the superior temporal gyrus"
     return
         <image>
             {$image/oid}
         </image>
```

Two Sources Query `Q7` on IM returns the empty set ! The reason is that *middle part of the superior temporal gyrus* is a too specific anatomical name, and the images stored in the database are annotated with higher level names. Users unfamiliar with the details of the IM data can get around that by refering to FMA, the authority on the human anatomy. This is expressed as a distributed query, essentially combining `Q6` and `Q7`.

```
for $image in document("image_db.xml")//image,
    $n in
     (execute at "http://csm.biostr.washington.edu/oqafma"
      foreign
       {WHERE Y->":NAME"->"Middle part of superior temporal gyrus",
              X->"part"*->Y,
              X->":NAME"->Ancestor
        CREATE  Concept(Ancestor);
       }
     )/results/Concept/text()
where $image/annotation_set/image_annotation/name/text()
            ="middle part of the superior temporal gyrus"
return
    <image>
        {$image/@oid}
    </image>
```

Notice the use of the `foreign` keyword, needed to express a subquery in a different language (StruQL). The query with out the use of the ontology returned the empty set. The modified query returns 191 image oids.

All Three Now Scientists are especially interested in integrating CSM with IM. For example, find the names of all structures over all patients, in which a CSM error of type 2 (semantic paraphasia) occurred at least once in one patient. For each of these names find the IDs of all images annotated with this name. However, it is not possible to join CSM and IM directly because the regions in CSM are more detailed than those in Image Manager (hence, we would again obtain an empty result). The solution is to use the FMA, to relate the anatomical names. This results in the following query spanning three different sources:

```
for $image in document("image_db.xml")//image
let $region_name :=
      execute at "http://csm.biostr.washington.edu/axis/csm.jws"
      xquery
       { for $trial in PublicView("Scrubbed.pv")
                   /patient/surgery/csmstudy/trial
         where $trial/trialcode/term/abbrev/text()="2"
         return $trial/stimsite/name()
       },
    $surrounding_regions :=
        for $term in $region_name
        return
          <term>
           {(execute at "http://csm.biostr.washington.edu/oqafma"
             foreign
              {WHERE Y->":NAME"->"$term",
                     X->("part")*->Y,
                     X->":NAME"->Ancestor
               CREATE  Concept(Ancestor);
              }
             )/results/Concept/text()
           }
          </term>
where $image/annotation_set/image_annotation/name/text()
          = $surrounding_regions/text()
return $image/oid/text()
```

At this point the reader may appreciate the power of our simple language extension. This last query integrates data from three different sources: MySQL, an ontology, and an XML file. All three sources are important for the scientists in order to retrieve relevant results. While writing the query requires extensive knowledge of all three sources, we found that scientists are willing to do that if that helps them obtain the data they need. On the other hand, the infrastructure needed to support such queries is minimal, and requires only minimal cooperation from the source owners: we show this next.

4 Implementation

The implementation consists of two parts: (1) a modified XQuery processor to support the XQueryD language, and (2) a Webservice wrapper over each data source.

(1) took a couple of weeks of work. We modified Galax [FS02], a freely available XQuery processor, adding the following two components: an extension of the **parser** to accept an `execute` statement, and a runtime **rewriter** that accepts the XQuery expression in the `execute` statement, scans it for variables, and substitutes them with the current runtime values. Recall that in XQuery a variable may refer to

a sequence consisting of arbitrary XML elements or scalars. Textual substitution is not possible. Instead we introduce a `let` binding in the query being shipped, which assigns the same value. Of course, this is not possible for foreign languages. Here the substitution is textual, and is limited only to variables that are bound to strings. If a variable is bound to an XML fragment, a runtime error is generated. We currently support only very limited exceptions, and have not implemented optimizations yet.

For (2), adding a Webservice interface to a data source takes less than one hour of work. The interface needs to accept an `execute` method, accepting an XQuery or XQueryD string as input, and outputting an XML value, representing the answer. We found that most data sources accept some form of a query API (e.g. odbc, or some proprietary code), and it is easy to write a uniform Webservice over each such API.

5 Distributed Optimizations

We are currently implementing several rewrite rules that allow the following optimizations to be expressed: replacing dependent joins with joins, semijoin reduction, and pushing computations to the sources. All three are well known from the database literature, and they are all implemented as source-to-source query rewritings. To illustrate, a typical rewriting rule is:

```
execute at s xquery{ . . .  E . . .}
        --->
 let $y := E
   return execute at s xquery{ . . . $y . . . }
```

where `E($x)` is an expression without any occurrence of a local variable in `execute`. The appendix illustrates more rewrite rules, showing how they can be used to optimize `Q2` into `Q2'`.

6 Conclusions

We have proposed a simple, lightweight extension to XQuery that allows users to express queries over multiple, heterogeneous data sources. While simple, our approach is quite expressive, and, when used by knowledgeable users, can become a very powerful tool. We are now starting to use this in the UW Human Brain Project. The approach is not well suited in applications where queries are written by end users, with little knowledge of the domain.

Future work will include further development of the exception handling mechanism and the implementation of a few optimizations.

Acknowledgments This work was partially funded by Human Brain Project Grant DC02310. Suciu was partially supported by the NSF CAREER Grant IIS-0092955, NSF Grants IIS-0140493 and IIS-0205635, a gift from Microsoft, and a Sloan Fellowship.

References

[ABC+03] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic xml documents with distribution and replication. In *SIGMOD*, pages 527–538, 2003.

[ABC+04] S .Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for active xml. In *SIGMOD*, 2004.

[ABM04] S .Abiteboul, O. Benjelloun, and T. Milo. Positive active xml. In *PODS*, 2004.

[BMP+97] J. F. Brinkley, L. M. Myers, J. S. Prothero, G. H. Heil, J. S. Tsuruda, K. R. Maravilla, G. A. Ojemann, and C. Rosse. A structural information framework for brain mapping. In *Neuroinformatics: An Overview of the Human Brain Project*, pages 309–334. Mahwah, New Jersey: Lawrence Erlbaum, 1997. See also `http://sig.biostr.washington.edu/projects/brain/`.

[FFLS00] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Declarative specification of web sites with strudel. *VLDB Journal*, 9(1):38–55, 2000.

[FKM+02] M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, and W. Tan. SilkRoute : a framework for publishing relational data in XML. *ACM Transactions on Database Technology*, 27(4), December 2002.

[FS02] M. Fernandez and J. Simeon. Galax: the XQuery implementation for discriminating hackers, 2002. available from `http://db.bell-labs.com/galax/`.

[GM03] P. Gardner and S. Maffeis. Modelling dynamic Web data. In *Proceedings of DBPL*, pages 75–84, Potsdam, Germany, 2003.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[MBR03] P. Mork, J. F. Brinkley, and C. Rosse. OQAFMA querying agent for the foundational model of anatomy: a prototype for providing flexible and efficient access to large semantic networks. *J. Biomedical Informatics*, 36(6):501–517, 2003.

[RM03] C. Rosse and J. L. V. Mejino. A reference ontology for bioinformatics: the foundational model of anatomy. *Journal of Bioinformatics*, 36(6):478–500, 2003.

[ST01] A. Sahuguet and V. Tannen. ubQL, a language for programming distributed query systems. In *WebDB*, pages 37–42, 2001.

[TKL+03] Z. Tang, Y. Kadiyska, H. Li, D. Suciu, and J. F. Brinkley. Dynamic xml-based exchange of relational data: application to the human brain project. In *Proceedings, Annual Fall Symposium of the American Medical Informatics Association*, pages 649–653, Washington, D.C., 2003.

[Won00] Limsoon Wong. The functional guts of the Kleisli query system. In *Proceedings of ICFP*, pages 1–10, 2000.

A An Optimization using Rewrite Rules

We show here how a series or rewrite rules can transform query `Q2`, which uses dependent joins, into `Q2'`, which uses a normal join. As we argued in the paper, `Q2` is natural to write and easy to read but inefficient, while `Q2'` is efficient but hard to read.

```
for $x in document("X.xml")/a/b[@c="123"]
let $y := (execute at "http://Y.com"
     xquery{ for $z in document("Y.xml")/d/e[@f=$x/@g] return $z})/h
return <answer> <x> { $x/u } </x>
                <y> { $y/v } </y>
       </answer>
```

Push `h` inside:

```
for $x in document("X.xml")/a/b[@c="123"]
let $y := (execute at "http://Y.com"
     xquery{ for $z in document("Y.xml")/d/e[@f=$x/@g] return $z/h})
return <answer> <x> { $x/u } </x>
                <y> { $y/v } </y>
       </answer>
```

Replace `y`:

```
for $x in document("X.xml")/a/b[@c="123"]
return <answer> <x> { $x/u } </x>
                <y> { (execute at "http://Y.com"
                       xquery{ for $z in document("Y.xml")/d/e[@f=$x/@g] return $z/h})/v } </y>
       </answer>
```

Push result across:

```
for $x in document("X.xml")/a/b[@c="123"]
return execute at "http://Y.com"
       xquery{ return
                <answer> <x> { $x/u } </x>
                         <y> { for $z in document("Y.xml")/d/e[@f=$x/@g] return $z/h})/v }
                <answer> }
```

Push down projections:

```
for $x in document("X.xml")/a/b[@c="123"]
let $x_at_g := $x/@g,
    $x_u := $x/u
return execute at "http://Y.com"
       xquery{ return
                   <answer> <x> { $x_u } </x>
                            <y> { for $z in document("Y.xml")/d/e[@f=$x_at_g] return $z/h})/v }
                   <answer> }
```

Combine dependents and push across:

```
let $to_ship := (for $x in for $x in document("X.xml")/a/b[@c="123"]
                           return <x_at_g>$x/@g</x_at_g>
                                  <x_u>$x/u</x_u>)
return  execute at "http://Y.com"
        xquery{ for $t in $to_ship
                return
                   <answer> <x> { $t/x_u/text() } </x>
                            <y> { for $z in document("Y.xml")/d/e[@f=$t/x_at_g/text()] return $z/h})/v }
```