XRPC: Interoperable and Efficient Distributed XQuery

Ying Zhang
Centrum voor Wiskunde en Informatica
P.O.Box 94079, 1090 GB
Amsterdam, the Netherlands
Y.Zhang@cwi.nl

Peter Boncz
Centrum voor Wiskunde en Informatica
P.O.Box 94079, 1090 GB
Amsterdam, the Netherlands
P.Boncz@cwi.nl

ABSTRACT

We propose XRPC, a minimal XQuery extension that enables distributed yet efficient querying of heterogeneous XQuery data sources. XRPC enhances the existing concept of XQuery functions with the Remote Procedure Call (RPC) paradigm. By calling out of an XQuery `for`-loop to multiple destinations, and by calling functions that themselves perform XRPC calls, complex P2P communication patterns can be achieved. The XRPC extension is orthogonal to all XQuery features, including the XQuery Update Facility (XQUF). We provide formal semantics for XRPC that encompasses execution of both read-only and update queries.

XRPC is also a network SOAP sub-protocol, that integrates seamlessly with web services and Service Oriented Architectures (SOA), and AJAX-based GUIs. A crucial feature of the protocol is *bulk RPC*, that allows remote execution of many different calls to the same procedure, using possibly a single network round-trip. The efficiency potential of XRPC is demonstrated via an open-source implementation in MonetDB/XQuery. We show, however, that XRPC is not system-specific: every XQuery data source can service XRPC calls using a wrapper.

Since XQuery is a pure functional language, we can leverage techniques developed for functional query decomposition to rewrite data shipping queries into XRPC-based function shipping queries. Powerful distributed database techniques (such as semi-join optimizations) directly map on bulk RPC, opening up interesting future work opportunities.

1. INTRODUCTION

The main contribution of this paper is the proposal of a minimal yet powerful XQuery extension, XRPC, that enables efficient distributed querying with a focus on interoperability between heterogeneous data sources. In this paper we provide in-depth insight in the consequences of this proposal for the formal semantics of XQuery (inclusive updates), the ease and potential efficiency of its implementation in existing XQuery systems, and the expressiveness of XRPC to specify distributed query processing strategies.

In more detail, we view our contributions as follows: *(i)* to establish the XRPC language syntax extension, including its SOAP-based XRPC network protocol, and provide a formal semantics for XRPC. *(ii)* the idea of set-at-a-time RPC (a.k.a. Bulk RPC) to make XRPC truly efficient. *(iii)* identifying various isolation levels for distributed XRPC updates, that result from calling *updating functions*, as defined by the XQuery Update Facility (XQUF) W3C Draft, over XRPC. *(iv)* showing that XRPC is sufficiently powerful to be used as the target language for a distributed query optimizer that can generate query plans for heterogeneous XQuery systems.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

XRPC Language Extension. XQuery only provides a *data shipping* model for querying XML documents distributed on the Internet. The built-in function `fn:doc()` fetches an XML document from a remote peer to the local server, where it subsequently can be queried. The recent W3C working draft of XQuery Update Facility (XQUF) introduces a built-in function `fn:put()` for remote storage of XML documents, which again implies data shipping.

There have been various proposals to equip XQuery with *function shipping* style distributed querying abilities [28, 30, 33], and on the syntax level, we consider our XRPC proposal an incremental development of these. XRPC adds RPC to XQuery in the most simple way, adding a destination URI to the XQuery equivalent of a procedure call (i.e. function application).

The design goal of XRPC is to create a distributed XQuery mechanism with which different XQuery processors at different sites can jointly execute queries. This implies that our proposal also encompasses a *network protocol*. Network communication in XRPC uses SOAP (i.e. XML messages) over HTTP. XML is ideal for distributed environments (think of character encoding hassles, byte ordering), XQuery engines are perfectly equipped to process XML messages, and an XML-based message protocol makes it trivial to support passing values of any type from the XQuery Data Model [15]. The choice for SOAP brings as additional advantages seamless integration of XQuery data sources with web services and Service Oriented Architectures (SOA) as well as AJAX-style GUIs.

Bulk RPC. Our SOAP XRPC protocol allows to compute multiple applications of the same function (with different parameters) in a single request/response network interaction. Bulk RPC is much more efficient than repeated single RPC as network latency is amortized over many calls, and performance becomes bounded by network bandwidth or CPU throughput (hardware factors that scale much better than network latency).

We implemented XRPC in the relational open-source XQuery DBMS MonetDB/XQuery [9] based on the *Pathfinder* compiler [18]. The essence of the compilation technique employed by Pathfinder is *loop-lifting* [18], which translates XPath/XQuery expressions inside `for`-loops into single bulk relational query plans that process all iterations of the loop independently of each other. In case of Pathfinder, with its loop-lifted approach to XQuery translation, it was trivial to generate Bulk RPC requests for any XRPC call found

in an XQuery. That is, a XRPC call nested in a `for`-loop taken many times leads to only a single Bulk XRPC request/response, which invokes the function for all iterations of the loop in bulk.

Another way to look at Bulk RPC is that it exposes bulk execution opportunities, such that e.g. a function that selects with a constant argument is turned into a join against the sequence of all arguments. Bulk RPC thus has a direct correspondence with set-oriented processing as offered by query algebras, and we believe can be generally applied any algebraic XQuery implementation.

XRPC and Updates. During a single XRPC query, it may happen that multiple read-only XRPC requests are sent to the same site. In the *repeatable read* isolation level we define, each request from the same query is guaranteed to see the same database state.

XRPC queries may themselves also update the databases, by invoking XQUF "updating functions" over XRPC. Note that XQUF queries only perform side-effecting actions *after* all query execution has finished, such that during query execution the database state is constant, and updating queries behave much like read-only queries. Obviously, atomically committing a distributed transaction requires a protocol like two-Phase Commit (2PC). We decided not to add 2PC to the XRPC network protocol, but rather rely on the recent industry standard Web Services Atomic Transaction (WS-AtomicTransaction) [3, 4] that provides exactly this feature for distributed web-service transactions.

XRPC as target language. One of the design goals of XRPC is – besides it being directly useful as an explicit instrument to write distributed queries – to have it serve as the target language for a distributed XQuery optimizer that takes queries without XRPC as input (thus data shipping only), and produces a decomposed query as output that uses XRPC for function shipping.

Our choice to make distributed execution explicit in terms of remote functions and their dependencies (parameters), aligns well with XQuery being a pure functional language. Query decomposition techniques [21] can thus be applied to decompose the full query (function) into sub-queries (again functions), that each can in theory be executed on any of the participating sites.

Note that automatic query decomposition techniques are beyond the scope of this paper (but part of our future work). We limit ourselves here to showing how some well-known distributed query execution strategies, such as the distributed semi-join strategy, can be elegantly expressed in XRPC. To demonstrate the performance opportunities of XRPC, as well as its interoperability, we provide some initial performance experiments with one peer running MonetDB/XQuery, and another running Saxon.[1]

Outline. This paper is organized as follows. In Section 2 we give a definition of the XRPC language extension, including the SOAP sub-protocol it uses, and spend considerable time in rigorously defining the formal semantics of XRPC. Section 3 outlines the initial implementation of XRPC in MonetDB/XQuery, inclusive the correspondence of Bulk RPC with the loop-lifting technique applied by the pathfinder compiler. In Section 4 we demonstrate in case of Saxon how XRPC can be used already with *any* XQuery system, using an XRPC wrapper that is capable of translating Bulk RPC requests into XQuery. Section 5 then shows how XRPC can be used to elegantly express various distributed query processing strategies, including experiments in which MonetDB/XQuery and Saxon work together over XRPC, using e.g. the distributed semi-join strategy. Finally, we discuss related work in Section 6 before outlining our conclusions and future work in Section 7.

[1]Section 4 outlines a simple *XRPC wrapper* that allows arbitrary XQuery data sources to handle XRPC calls.

2. THE XRPC LANGUAGE EXTENSION

Syntax. Remote function applications take the XQuery syntax:

`execute at` { *Expr* }{ *FunApp* (*ParamList*) }

where *Expr* is an XQuery `xs:string` expression that specifies the URI of the peer on which *FunApp* is to be executed. The function to be applied can be a built-in or user-defined. For user-defined functions, we currently restrict ourselves to functions defined in an XQuery Module. A small (future) extension to the network protocol would also allow functions defined inside the query to be executed over XRPC.

For a precise syntax definition, we show the rules of the XQuery 1.0 grammar that were changed:

```
PrimaryExpr ::= ...| FunctionCall | XRPCCall | ...
XRPCCall ::= "execute at" "{" ExprSing "}" "{" FunctionCall "}"
FunctionCall::= QName "(" (ExprSingle("," ExprSingle)*)?  ")"
```

Example. As a running example, we will assume a set of XQuery database systems (peers) that each store a movie database document `"filmDB.xml"` with contents similar to:

```
<films>
 <film><name>The Rock</name><actor>Sean Connery</actor></film>
 <film><name>Goldfinger</name><actor>Sean Connery</actor></film>
 <film><name>Green Card</name><actor>Gerard Depardieu</actor></film>
</films>
```

We assume an XQuery module `film.xq` stored at `x.example.org`, that defines a function `filmsByActor()`:

```
module namespace film="films";
declare function film:filmsByActor($actor as xs:string) as node()*
{ doc("filmDB.xml")//name[../actor=$actor] };
```

We can execute this function on remote peer `"y.example.org"` to get a sequence of films in which Sean Connery plays in the remote film database:

```
import module namespace f="films" at "http://x.example.org/film.xq";
<films> {
 execute at {"xrpc://y.example.org"}
           {f:filmsByActor("Sean Connery")}                    (Q1)
} </films>
```

We introduce here a new `xrpc` network protocol, accepted in the destination URI of `execute at`. The generic form of such URIs is: `xrpc://`$<$ *host* $>$ [: *port*] [/[*path*]]. The `xrpc://` indicates the network protocol. The second part, $<$ *host* $>$ [: *port*], indicates the remote peer. The third part, [/[*path*]], is an optional local path at the remote peer.

The above example yields:

```
<films><name>The Rock</name><name>Goldfinger</name></films>
```

More Examples. A more elaborate example demonstrates the possibility of multiple remote function calls to a peer:

```
import module namespace f="films" at "http://x.example.org/film.xq";
<films> {
  for $actor in ("Julie Andrews", "Sean Connery")
  let $dst := "xrpc://y.example.org"                             (Q2)
  return execute at {$dst} {f:filmsByActor($actor)}
} </films>
```

and to make it a bit more complex, we could do multiple function calls to multiple remote peers:

```
import module namespace f="films" at "http://x.example.org/film.xq";
<films> {
  for $actor in ("Julie Andrews", "Sean Connery")
  for $dst in ("xrpc://y.example.org", "xrpc://z.example.org")  (Q3)
  return execute at {$dst} {f:filmsByActor($actor)}
} </films>
```

2.1 SOAP XRPC Message Format

SOAP (Simple Object Access Protocol) is the XML-based message format used for web services [26, 19, 20], and we propose the use of SOAP messages over HTTP as the network protocol underlying XRPC. SOAP web service interactions usually follow an RPC (request/response) pattern, though the SOAP protocol is much richer and allows multi-hop communications, and highly configurable error handling. For the simple RPC use of SOAP over HTTP, a sub-protocol called "SOAP RPC" is in common use [20]. SOAP RPC is oriented towards binding with programming languages such as C++ and Java, and specifies parameter marshaling of a certain number of simple (atomic) data types, and also allows passing *arrays* and *structs* of such data-types. However, its supported atomic data types do not match directly those of the XQuery Data Model (XDM) [15], and the support for arrays and structs is not relevant in XRPC, where there rather is a need for supporting arbitrary-shaped XML nodes as parameters as well as sequences of heterogeneously typed items. This is the reason, why our SOAP XRPC message format, while supporting the general SOAP standard over HTTP with the purpose of RPC, implements a new parameter passing sub-format (SOAP XRPC $\neq$ SOAP RPC). The most often used form of SOAP RPC is called *rpc/encoded*, while our SOAP XRPC protocol belongs to the family of *document/literal*. It was shown in [12] that rpc/encoded in general is significantly slower than document/literal, and suffers from scalability problems when the message size increases.

XRPC Request Message. SOAP messages consist of an envelope, with a (possibly empty) header and a body. Inside the body, we define a `request` that specifies a module URI `module`, an at-hint `location`, a function name `method` and its `arity`. The actual parameters of a single function call are enclosed by a `call` element. Each individual parameter consists of a `sequence` element, that contains zero or more values.

Below we show the XRPC request message for the first example query, that looks for films with Sean Connery:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
 xmlns:env="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://monetdb.cwi.nl/XQuery
             http://monetdb.cwi.nl/XQuery/XRPC.xsd">
 <env:Body>
  <xrpc:request module="films" method="filmsByActor" arity="1"
                location="http://x.example.org/film.xq">
   <xrpc:call>
    <xrpc:sequence>
     <xrpc:atomic-value
      xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
    </xrpc:sequence>
   </xrpc:call>
  </xrpc:request>
 </env:Body>
</env:Envelope>
```

Atomic values are represented with `atomic-value`, and are annotated with their (simple) XML Schema Type in the `xsi:type` attribute. Thus, the heterogeneously typed sequence consisting on an integer `2` and double `3.1` would become:

```
<xrpc:sequence>
  <xrpc:atomic-value xsi:type="xs:integer">2</xrpc:atomic-value>
  <xrpc:atomic-value xsi:type="xs:double">3.1</xrpc:atomic-value>
</xrpc:sequence>
```

XML nodes are passed by value in an `<element>` element:

```
<xrpc:sequence>
 <xrpc:element><name>The Rock</name></xrpc:element>
 <xrpc:element><name>Goldfinger</name></xrpc:element>
</xrpc:sequence>
```

Similarly, the XML Schema `XRPC.xsd`[2] defines enclosing elements for document, attribute, text, processing instruction, and comment nodes. Document nodes are represented in the SOAP message as a `<document>` element that contains the serialized document root. Text, comment and processing instruction nodes are serialized textually inside the respective elements `<text>`, `<comment>` and `<pi>`. Attribute nodes are serialized *inside* the `<attribute>` element: `<xrpc:attribute x="y"/>`.

XRPC fully supports the XQuery Data Model, a requirement for making it an orthogonal language feature. This implies XRPC also supports passing of values of user-defined XML Schema types, including the ability to validate SOAP messages. XQuery already allows importing XML Schema files that contain such definitions. Values of user-defined *named* types are enclosed in SOAP messages by `<element>` elements, with an `<xsi:type>` attribute annotating their type. The XQuery system implementing XRPC should include an `xsi:schemaLocation` declaration as well as an `xmlns` namespace definition inside the `<Envelope>` element, when values of such imported element types occur in the SOAP message. If a parameter has an *anonymous* user-defined schema type, however, its type information is lost, but this can be avoided exploiting a future protocol extension[4] (discussed later) by including the lowest ancestor-or-self element with a *named* schema type in the SOAP message.

XRPC Response Messages follow the same principles. Inside the body is now an `xrpc:response` element that contains the result `sequence` of the remote function call, e.g.:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
 xmlns:env="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://monetdb.cwi.nl/XQuery
                     http://monetdb.cwi.nl/XQuery/XRPC.xsd">
 <env:Body>
  <xrpc:response module="films" method="filmsByActor">
   <xrpc:sequence>
    <xrpc:element><name>The Rock</name></xrpc:element>
    <xrpc:element><name>Goldfinger</name></xrpc:element>
   </xrpc:sequence>
  </xrpc:response>
 </env:Body>
</env:Envelope>
```

XRPC Error Message. Whenever an XRPC server discovers an error during the processing of an XRPC request, it immediately stops execution and sends back an XRPC error message, using the format of the SOAP Fault message ([26], [19]). Thus, any error will cause a run-time error at the site that originated the query. As an example, the following SOAP Fault message indicates that a required module could not be loaded (we show only the `env:Fault` element):

```
<env:Fault>
  <env:Code><env:Value>env:Sender</env:Value></env:Code>
  <env:Reason>
    <env:Text xml:lang="en">could not load module!</env:Text>
  </env:Reason>
</env:Fault>
```

Outlook. Our discussion of SOAP XRPC message is not fully done yet. In the next subsection, we will extend the format with support for isolation and updates. Then, in Section 3.2 we describe the *Bulk RPC* feature, that allows a single message to request multiple function calls.

[2] See `http://monetdb.cwi.nl/XQuery/XRPC.xsd`

2.2 XRPC Formal Semantics

In defining the semantics of XRPC, we take care to attach proper database semantics to the concept of RPC, to ensure that all RPCs being done on behalf of a single query see a consistent distributed database image and commit atomically. It is known that full serializability in distributed queries can come at a high cost, and therefore we also define certain less strict isolation levels that still may be useful to certain applications.

We use the following notation and terms:

– $\mathcal{P}$ denotes a set of *peer identifiers*. We use the peer identifier p_0 to denote the *local peer*, on which a particular query is started. All other peers $p_i \in \mathcal{P}$ are *remote peers*. In practice, a peer identifier is a URI from the `xrpc` protocol, that contains a host and (optionally) a port number.

– $\mathcal{F}$ denotes a set of *XRPC function applications*. An XRPC call $f^{p_i \to p_j}$ that triggered from p_i that causes function f to be executed at p_j is an *updating XRPC call* ($f_u^{p_i \to p_j} \in \mathcal{F}_u$), if it calls an updating function; otherwise, it is a *non-updating XRPC call* ($f_r^{p_i \to p_j} \in \mathcal{F}_r$). If the evaluation of an XRPC call $f^{p_i \to p_j}$ requires evaluation of other XRPC call(s) at p_j, we term $f^{p_i \to p_j}$ a *nested XRPC call*.

– $\mathcal{M}$ denotes a set of XQuery *modules*. A module consists of a number of function definitions d_f. Each XRPC call $f^{p_i \to p_j}$ must correspond to a definition d_f from some module $m_f \in \mathcal{M}$.

– An *XRPC query* is an XQuery query q which contains at least one XRPC call $f^{p_i \to p_j} \in \mathcal{F}_q$, where $\mathcal{F}_q$ denotes the set of all function calls performed during execution of q. We call a query in which only one, non-nested XRPC call appears a *simple XRPC query*. An XRPC query q is an *updating XRPC query*, if it contains at least one update command or a call to an updating (XRPC) function.

– Each query operates in a *dynamic context*. The XQuery 1.0 Formal Semantics [13] defines that each expression is normalized to a *core* expression, which then is defined by a semantic judgment $dynEnv \vdash Expr \Rightarrow val$. The semantic judgment specifies that in the dynamic context $dynEnv$, the expression $Expr$ evaluates to the value val, where val is an instance of the XQuery Data Model (XDM). For now, we simplify the dynamic environment to a database state db (i.e. the documents and their contents stored in the XML database): $dynEnv \simeq db$. The $dynEnv.docValue$ from the XQuery Formal Semantics [13] corresponds to db used here. To indicate a context at a particular peer p, we write db^p.

– When considering that a database may be changed by updates, we can view it as a function over time t as $db^p(t)$. In our formal rules, the default assumption on database states is that they stay equal over time, unless otherwise stated. When the time context t is clear, the shorthand notation db^p is used to refer to the current database state.

Basic read-only XRPC, i.e. the semantics of executing a read-only function $f^{p_0 \to p_x}$ ($f \in \mathcal{F}_r$), is defined by extending the XQuery 1.0 semantic judgments with a new rule [3]:

$$\frac{\begin{array}{c} db^{p_0}(t_0) \vdash \texttt{<call>}\{\mathbf{s2n}(v_1), \cdots, \mathbf{s2n}(v_n)\}\texttt{</call>} \Rightarrow call; \\ send^{p_0 \to p_x} request(m, f_r, call); t_x \geq t_0 \\ db^{p_x}(t_x) \vdash \mathbf{s2n}(f_r(\mathbf{n2s}(call/*[1]), \cdots, \mathbf{n2s}(call/*[n]))) \Rightarrow res; \\ send^{p_x \to p_0} reply(res); \\ db^{p_0}(t_0) \vdash \mathbf{n2s}(res) \Rightarrow v_{res}; \end{array}}{db^{p_0}(t_0) \vdash f_r^{p_0 \to p_x}(v_1, \cdots, v_n) \Rightarrow v_{res}}$$

$$(\mathcal{R}_{\mathcal{F}_r})$$

This rule $\mathcal{R}_{\mathcal{F}_r}$ states that execution at p_0 of the (read-only) XRPC call $f^{p_0 \to p_x}(v_1, \cdots, v_n)$ in the dynamic context $db^{p_0}(t_0)$ (without further assumption on t_0) starts with constructing a `<call>` element that contains the SOAP representation of all parameters v_i. This XML representation, described in the previous Section 2.1 is created by the sequence-to-node marshaling function `s2n()`, discussed below. Then, the request $(m, f, call)$ is sent to peer p_x. Here, m is the module URI (plus at-hint) in which function f_r is defined. The function f_r is then evaluated as a normal local function in the dynamic context of the remote peer $db^{p_x}(t_x)$, where we only assume $t_x \geq t_0$. The parameters of f_r, are obtained by using the inverse node-to-sequence marshaling function `n2s()` to produce the result node *res*. This result *res* is sent back to peer p_0, which finally converts *res* into the result sequence v_{res}.

This definition inductively relies on the XQuery Formal Semantics to evaluate f locally at p_x, and thus may trigger the evaluation of additional XRPCs if these happen to be present in the body of f. Also, this definition covers execution of XRPC calls in the current database state db^{p_0}, which we need for our basic purpose of defining the semantics XRPC queries (in which case t_0 is the current time point). Finally, this XRPC rule does not produce any new current local database state db^{p_0} nor new remote database state db^{p_x} (i.e. it defines read-only semantics).

Parameter Marshaling. The SOAP representation of a sequence `$seq` is created in a new `<sequence>` node by the function:

```
declare function s2n($seq as item*) as node()
```

The inverse transformation (from `<sequence>` representation to real item sequence) is provided by:

```
declare function n2s($n as node()) as item*
```

For example, we get `("abc",42)` from calling:

```
n2s(<xrpc:sequence>
    <xrpc:atomic-value xsi:type="xs:string">abc</xrpc:atomic-value>
    <xrpc:atomic-value xsi:type="xs:integer">42</xrpc:atomic-value>
    </xrpc:sequence>)
```

An important characteristic of the function `n2s()` is that it guarantees that for node-typed parameters (i.e. those represented as `<element>`, `<text>`, `<document>`, `<attribute>`, `<comment>` and `<pi>`) an XDM node of the correct type is returned **as a separate XML fragment**. This guarantees that evaluating the upwards and horizontal XPath axes on such nodes will return empty results. It may be tempting to return element nodes under the identity found in the message (i.e. $\$request/call/xrpc:sequence[i]/xrpc:element/*$), but this would allow a query to navigate e.g. to the SOAP envelope element, or the other function parameters.

One should note that `n2s()` and `s2n()` are internal functions only that do not need to be exposed to XRPC users, and in fact do not need to exist in reality, as each XRPC system implementation may have its own internal (efficient) mechanisms to process SOAP messages. In case of MonetDB/XQuery, beyond shredding the SOAP request and response messages, we do not spend any effort in `n2s()` nor `s2n()` on element construction to retrieve node values of the correct type, as our implementation directly chops up the shredded XML message in separate XML fragments per function parameter, and modifies node types internally (as the SOAP messages are invisible to the user, their integrity can be compromised at will by the system). It is possible, though, to implement `n2s()` and `s2n()` purely in XQuery, as we will show when we discuss the XRPC wrapper, that allows arbitrary XQuery processors to participate in distributed XRPC queries in Section 4.

A final detailed remark on parameter marshaling is that XRPC requires the *caller* to perform parameter up-casting. The rationale is that such casting is already part of the standard function application code generated by any XQuery system, thus easy to do at the caller for XRPC calls, and makes it easier to implement XRPC

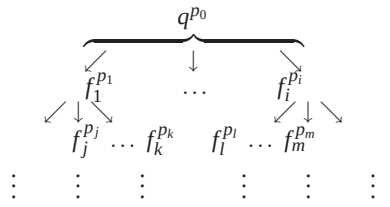
[3] In our rules, we use the ';' sign to suggest an order in the evaluation of the statements.

handlers that have no or limited XQuery capabilities (e.g. wrapped outside web services as in [28]).

Call-by-Value. An important choice implied by making `n2s()` and `s2n()` explicit in our Formal Semantics is to enforce *by-value* parameter passing in XRPC. If nodes are passed as parameters of an XRPC call, they will be serialized into a SOAP message, shipped to the remote side, and there new nodes will be constructed (or: appear) of the correct type with equal-valued contents, but with different node identifiers. This can lead to a number of semantic differences between local and remote function application. We already mentioned that XPath navigation from node parameters over non-downwards axes (e.g. `parent`, `following`) will always produce empty results on the remote side. More subtly, if a function is invoked over XRPC with two nodes as parameters that have a `descendant-or-self` relationship, XRPC parameter marshaling will destroy this relationship at the remote side[4] Finally, the XQuery Formal Semantics specifies that some consistent order should be enforced over nodes from different documents, but our semantics will not respect this order on their copies when shipped over XRPC.

The rationale behind this by-value choice is that *by-reference* semantics would lead to complications when the upwards or sideways XPath axes are invoked on node parameters (or results) of XRPC calls. Correctly supporting that would either lead to the need to ship the full XML data fragment for all node parameters upfront (defeating the purpose of function shipping) or cause implicit communication when navigating beyond the descendants of such nodes.

Obviously, call-by-value semantics complicate life when XRPC is used as the target language for automatic query distribution (as opposed to explicit XRPC query processing, where we can assume the query writer to be aware of the call-by-value semantics). In that case, the query optimizer has the task to make sure by-value parameter passing does not affect query semantics. The simplest solution is to refrain from function shipping in problematic cases, but more sophisticated solutions may be found for some query patterns.



Nested XRPC Calls. The general pattern of XRPC function applications generated by a query is a *tree*, as each XRPC call may again perform more XRPC calls. This happens when a query contains multiple XRPC function applications, or when such a function application occurs inside a `for`-loop. In the above diagram, the arrow '$\rightarrow$' should be read as "XRPC call".

[4]We are considering a future XRPC protocol extension that allows node parameters to be referred to using an `xrpc:nodeid` attribute that holds a node identifier. This alternative node representation can be used for nodes that are a descendant-or-self of another parameter that is fully serialized in the SOAP message. The `s2n()` function would then be altered to return nodes from the XML fragment that corresponds with that fully serialized parameter. This change of semantics ensures that ancestor/descendant relationships among parameters at the calling peer are preserved at the remote XRPC peer. This *indirect addressing* is useful for compressing the SOAP message. Moreover, if applied maximally, the resulting call-*by-fragment* result/parameter passing, allows an distributed XRPC rewriter to relocate parts of certain query predicates that *do* depend on node identity (i.e. node-valued join conditions whose predicates only contain descendant/ancestor XPath steps).

The peers $p_0, p_1, \cdots, p_i, p_j, \cdots, p_k, \cdots, p_l, \cdots, p_m$ are not necessarily unique: some peer p_i (or in fact many such peers) may occur multiple times in this tree. When considering rule $\mathcal{R}_{\mathcal{F}_r}$, the dynamic environment $dynEnv^{p_i}$ containing the *current* database state db^{p_i} may thus be seen multiple times during query evaluation. In between those multiple function evaluations, other transactions may update the database and change db^{p_i}. Thus, those different XRPC calls to the same remote peer p_i from the same query q may see different database states. This will not be acceptable for some applications and therefore, we deem it worthwhile to define *repeatable read* isolation for queries that perform XRPC calls.

Repeatable Read. XQuery users can control per query which semantics is used by using the XQuery `declare option` feature, setting `xrpc:isolation` either to `"none"` (rule $\mathcal{R}_{\mathcal{F}_r}$) or `"repeatable"`, defined by rule $\mathcal{R}'_{\mathcal{F}_r}$:

$$\frac{\begin{array}{c} db^{p_0}(t_q^{p_0}) \vdash \texttt{<call>}\{\mathbf{s2n}(v_1),\cdots,\mathbf{s2n}(v_n)\}\texttt{</call>} \Rightarrow call; \\ send^{p_0 \rightarrow p_x} request(q, m, f_r, call); \\ db^{p_x}(t_q^{p_x}) \vdash \mathbf{s2n}(f_r(\mathbf{n2s}(call/*[1]),\cdots,\mathbf{n2s}(call/*[n]))) \Rightarrow res; \\ send^{p_x \rightarrow p_0} reply(q, res); \\ db^{p_0}(t_q^{p_0}) \vdash \mathbf{n2s}(res) \Rightarrow v_{res}; \end{array}}{db^{p_0}(t_q^{p_0}) \vdash f_r^{p_0 \rightarrow p_x}(v_1,\cdots,v_n) \Rightarrow v_{res}}$$

$$(\mathcal{R}'_{\mathcal{F}_r})$$

The above rule $\mathcal{R}'_{\mathcal{F}_r}$ specifies that for evaluating XRPC calls on behalf of query q, peer p_x uses always the same database state $db^{p_x}(t_q^{p_x})$. Time $t_q^{p_x}$ is typically time that the first XRPC request of query q reached p_x; but we place no specific restriction on it. Observe that a unique query identifier q is now passed as an extra parameter in the XRPC request, such that a peer can recognize which XRPC calls belong to the same query and it can associate an isolated database state with it.

Clearly, XRPC with repeatable reads requires more resources to implement, as some *database isolation* mechanism (of choice) will have to be applied to retain $db^{p_x}(t_q^{p_x})$ across calls. The transaction mechanism of MonetDB/XQuery, for example, uses snapshot isolation based on shadow paging, which keeps copies of modified pages around. Systems that provide the isolation levels serializable or repeatable reads (obviously) can also provide this semantics.

A quite common reason why a peer is called multiple times in the same query and the need for repeatable reads arises, is when an XRPC call appears inside a for-loop. In Section 3.2 we describe how *Bulk RPC* helps avoid these costly isolation measures in case of *simple* XRPC queries (i.e. those that contain only one non-nested function application).

Other Isolation Levels. If we would suppose that all peers involved in q support the isolation level *snapshot isolation*, and all would use the *same* timestamp t_q as the one in which the original query executes, i.e. $t_q^{p_0} = \cdots = t_q^{p_x} = \cdots = t_q^{p_m} = t_q$, we could obtain the isolation level *distributed snapshot isolation*. Just using a globally consistent query timestamp is actually not enough for that, extra effort is needed to enforce distributed commits to happen at the same time point (one way to do that is to block or abort incoming reads while a node is in prepared stated – this is called the pessimistic approach in [32]). For this to be meaningful in practice, however, we would have to have a representation of t values (until now, this is left opaque) that allows a full ordering, thus enabling us to define a "happened before " query/transaction order $t_{q_1} \ll t_{q_2}$. However, as XRPC is also intended for use in P2P settings, we make no assumptions on a centralized distributed transaction coordinator that could give out unique and monotonically increasing t numbers. In absence of that, one could think of t numbers generated by Lamport Clocks [23], but while this method guarantees that

a transaction that depends on a previous one ("happened before") has a smaller Lamport clock value, the reverse inference cannot be made (i.e. meaningfully enforcing a transaction order depending on such t-s) unless all peers participate in all queries (which again is not a reasonable assumption in P2P). Of course, we can think of t as being "exact" (UTC) time, but as we do not want to assume either that all participating peers possess (synchronized!) Stratium grade precision clock hardware, this is only a theoretical notion.

For this reason, we leave the maximum XRPC isolation currently at the repeatable read level, though finding a distributed isolation level useful in P2P is on our future work agenda.

SOAP XRPC Extension: Isolation. XRPC uses repeatable reads semantics for requests that have the optional `queryID` child element in the `xrpc:request` element. The `queryID` in the SOAP message contains `host` and `timestamp` attributes that state on which host and at UTC time the query started initially, and a `timeout` attribute that specifies a local number of seconds during which to conserve the isolated database state. Note that the timeout is relative, it is a number of seconds – this mitigates problems caused by different peers having big clock synchronization differences. When the timeout passes, the isolated database state can be discarded, freeing up system resources. However, the local XRPC handler should still remember expired `queryID`s, such that it can give errors on XRPC requests that arrive too late. The purpose of sending the `timestamp` of the originating host is to ease the administration of expired `queryID`s, as per host only the latest timestamp needs to be retained, and can be restricted to some sane time interval.

A timeout mechanism is inevitable, even if XRPC would use a 2PC-like coordination protocol to signal the finishing of a query (for updates, XRPC actually does so via WS-AtomicTransaction), because such a coordination protocol also needs a timeout to conclude that remote hosts are no longer responding. Automatically computing a good timeout value requires a cost model that takes into account the query, data-distribution, network, and peer characteristics – a task we leave for our future work on automatic query distribution. Therefore, the timeout to use is specified in the query using `declare option xrpc:timeout <sec>`, so users and applications can set them according to their needs.

2.3 XRPC Update Semantics

The XRPC language extension is fully orthogonal to all XQuery features, and thus one can also make XRPC calls to user-defined *updating functions*, as defined by the XQuery Update Facility (XQUF). The XQUF syntax ensures that if a user-defined function contains one updating function, it must itself be an updating function. XQuery updates (and thus updating functions) determine which nodes to change (and how), purely based on the database state before the update, and produce a *pending update list* Δ. Only after query execution has finished, all updates in the pending update list are to be applied and committed. This concept is quite similar to IO monads, used in functional languages like Haskell, that cleanly separate functional execution from any side-effecting actions.

$$\frac{\begin{array}{c} db^{p_0}(t_0) \vdash \texttt{<call>}\{\mathbf{s2n}(v_1),\cdots,\mathbf{s2n}(v_n)\}\texttt{</call>} \Rightarrow call; \\ send^{p_0\rightarrow p_x} request(m, f_u, call); t_x \geq t_0 \\ db^{p_x}(t_x) \vdash f_u(\mathbf{n2s}(call/*[1]),\cdots,\mathbf{n2s}(call/*[n])) \Rightarrow \Delta; \\ db^{p_x}(t_x) \vdash \texttt{applyUpdates}(\Delta) \Rightarrow db^{p_x}; \\ send^{p_x\rightarrow p_0} reply() \end{array}}{db^{p_0}(t_0) \vdash f_u^{p_0\rightarrow p_x}(v_1,\cdots,v_n) \Rightarrow (), db^{p_x}} \quad (\mathcal{R}_{\mathcal{F}_u})$$

The above rule $\mathcal{R}_{\mathcal{F}_u}$ states that update functions apply the pending update list Δ immediately, producing a new current remote database state db^{p_x}. For this purpose, we use the internal function `applyUpdates()` defined in the XQUF [11] that carries through all changes in a pending update list. Note that this rule executes an updating call between p_0 and p_x in databases states from t_0 resp. t_x without other assumptions than $t_x \geq t_0$. Typically, an implementation may choose to use db^{p_x}, i. e. the latest database state to handle each XRPC request.

Remote execution of an XQUF updating function causes no new db^{p_0} state directly (it returns an empty pending update list), but does yield a new db^{p_x}. This is a simplification, because $f_u()$ itself may perform XRPC calls that modify database states of other peers involved in q – and potentially even db^{p_0} itself. While the local query q at p_0 always operates in $db^{p_0}(t_0)$, if it performs multiple XRPC calls to the same peer p_x, these calls will thus potentially see different states $db^{p_x}(t_{x1}), db^{p_x}(t_{x2}), \ldots$, which may even include the updates caused by the previous XRPC calls made for q. Therefore, while easy to implement, this semantics does not guarantee repeatable reads, even allows lost updates at the same peer between multiple calls performed on behalf of the same query, and will cause non-atomic distributed commits to happen if XRPC execution is aborted halfway due to an error.

Atomic Updates with Isolation. We now define an improved XRPC isolation level that provides repeatable reads as well as atomic distributed commit. Recall that the effects of XQUF updates are invisible until query execution finishes; only then `applyUpdates()` is invoked on the pending update list. In the previous rule $\mathcal{R}_{\mathcal{F}_u}$, updates were visible directly after handling each individual XRPC request. The new rule $\mathcal{R}'_{\mathcal{F}_u}$ given below, thus corresponds more closely to the intent of the XQUF, in that no side effects of query q are visible at any involved peer p_x until the query commits.

The repeatable read isolation implies that peers defer applying pending update lists created by individual XRPC calls made on behalf of the same query q until the point that q actually commits. Thus, peers p_x must not only keep track of the database state $db^{p_x}(t_q^{p_x})$, but also of a collection of pending update lists $\Delta_q^{p_x} = \cup_{\forall i \in \{1,\cdots,U_q^{p_x}\}} \Delta_q^{p_x}(i)$, where $U_q^{p_x}$ is the number of updating XRPC calls p_x has handled so far for q.

$$\frac{\begin{array}{c} db^{p_0}(t_q^{p_0}), \Delta_q^{p_0} \vdash \texttt{<call>}\{\mathbf{s2n}(v_1),\cdots,\mathbf{s2n}(v_n)\}\texttt{</call>} \Rightarrow call; \\ send^{p_0\rightarrow p_x} request(q, m, f_u, call); \\ db^{p_x}(t_q^{p_x}), \Delta_q^{p_x} \vdash f_u(\mathbf{n2s}(call/*[1]),\cdots,\mathbf{n2s}(call/*[n])) \Rightarrow \Delta_q^{p_x}(U_q^{p_x}); \\ send^{p_x\rightarrow p_0} reply() \end{array}}{db^{p_0}(t_q^{p_0}), \Delta_q^{p_0} \vdash f_u^{p_0\rightarrow p_x}(v_1,\cdots,v_n) \Rightarrow ()} \quad (\mathcal{R}'_{\mathcal{F}_u})$$

The translation of isolated updating XRPC calls is depicted in the inference rule $\mathcal{R}'_{\mathcal{F}_u}$ above. Like rule $\mathcal{R}'_{\mathcal{F}_r}$, this rule again provides for proper isolation by keeping the database state $db^{p_x}(t_q^{p_x})$ constant throughout the query. The execution of a function $f_u()$ at p_x causes a new pending update list to be created, that becomes part of the collection $\Delta_q^{p_x}$.

Obviously, atomically committing a distributed transaction requires a protocol like 2PC or one of its more advanced derivatives [29, 17]. We decided not to add 2PC to the XRPC network protocol, but rather rely on the recent industry standard WS-AtomicTransaction [3, 4] that provides exactly this feature for distributed web-service transactions. WS-AtomicTransaction [3] provides a rather vanilla SOAP-based 2PC interface with e.g. `Prepare()` and `Commit()` functions. It is embedded in the WS Coordinator framework [4] that allows to register a collection of peers that participate in a distributed transaction, and subsequently run a transaction protocol on those (in this case WS-AtomicTransaction). Thus, in order to support updates with this isolation level, XRPC systems

must implement support for these web service interfaces, and offer them over the same HTTP SOAP server that runs XRPC.

To implement proper 2PC, the `Prepare()` function brings q in prepared state. It may raise an error, if a conflicting transaction has reached this state already. Else, it logs the union of the pending update lists ($\Delta_q^{p_x}$) to stable storage, ensuring q can commit later:

$$\frac{\begin{array}{c} send_{p_0 \to p_x} request(q, Prepare); \\ db^{p_x}(t_q^{p_x}), \Delta_q^{p_x} \vdash \texttt{log}(\Delta_q^{p_x}) \Rightarrow r; \\ send_{p_x \to p_0} reply(r) \end{array}}{db^{p_0}(t_q^{p_0}), \Delta_q^{p_0} \vdash Prepare^{p_0 \to p_x}() \Rightarrow r}$$

`Commit()` carries through the updates, creating a new database state:

$$\frac{\begin{array}{c} send_{p_0 \to p_x} request(q, Commit); \\ db^{p_x}(t_q^{p_x}), \Delta_q^{p_x} \vdash \texttt{applyUpdates}(\Delta_q^{p_x}) \Rightarrow db^{p_x} \end{array}}{db^{p_0}(t_q^{p_0}), \Delta_q^{p_0} \vdash Commit^{p_0 \to p_x}() \Rightarrow db^{p_x}}$$

More SOAP XRPC Extensions. In XRPC, peer p_q that starts the query q is the one that registers the participating peers at the WS Coordinator service and initiates the Prepare and Commit phases. For this registration task, it thus needs to know a full list of peers that participate in the transaction. Due to nested XRPC calls, it may not be aware of all peers and therefore we extended the SOAP XRPC protocol to piggyback a list of all unique participating peers in the response message.

Finally, the XQUF specifies that when the same node is updated twice in the same query, the order in which the different update actions on that node are applied is *non-deterministic*! This means that we can simply union all individual $\Delta_q^{p_x}(i)$ pending update lists (one for each XRPC call handled in p_x for q) to get a full update list $\Delta_q^{p_x}$ without worrying about preserving some proper order on the update actions. In a separate work [35], omitted here for reasons of space, we have defined a deterministic update order for XQUF, and devised a way to enforce it over XRPC using a small XRPC protocol extension, despite the out-of-order execution effects of Bulk RPC, that will be observed at the end of Section 3.1.

3. MONETDB/XQUERY IMPLEMENTATION

We implemented XRPC in MonetDB/XQuery, an efficient yet purely relational XML database system [9]. It consists of the MonetDB relational database back-end, and the *Pathfinder* compiler [18], that translates XQuery into relational algebra as front-end.

The XRPC module contains an ultra-light HTTP daemon implementation [24] that runs a request handler (the XRPC server), and contains a message sender API (the XRPC client). We also had to add support for the `execute at` syntax to the Pathfinder XQuery compiler, and change its code generator to generate *stub code* that invokes the new message sender API.

The stub code uses the message sender API to generate a SOAP message from actual function parameters. This process reuses the normal sequence serialization mechanism in MonetDB/XQuery. The message sender API sends the XML message using HTTP POST and waits for a result message. The result message is subsequently shredded into a relational table, the way all XML documents are shredded in MonetDB/XQuery. The stub code retrieves atomic values from the SOAP document nodes; node-typed values just refer to the nodes in the newly shredded SOAP document.

The request handler, on the other side, behaves similarly. It listens for SOAP requests and shreds incoming messages into a temporary relational table, from which the parameter values are extracted. As MonetDB/XQuery is a relational system, XQuery values are all represented as (temporary) relational tables. The module function specified in the SOAP request is then executed locally with these parameter tables, producing a result table. The request handler then builds a response message in which this result table is serialized into XML, using the normal MonetDB/XQuery serialization mechanism onto the network socket.

Operator	Semantics
σ_a	select all rows with column $a = true$
$\pi_{a_1:b_1,\ldots,a_n:b_n}$	project columns $b_1,\ldots,b_n$ and possibly rename columns b_i to a_i (no duplicate removal)
δ	duplicate elimination
$\dot\cup$	disjoint union
$\bowtie_{a=b}$	equi-join
$\rho_{b:\langle a_1,\ldots,a_n\rangle/p}$	row numbering (DENSE_RANK SQL:1999)
ab	literal table

Table 1: Relational Algebra Generated By Pathfinder

3.1 Relational XQuery And Loop-Lifting

The *Pathfinder* compiler [18] translates XPath/XQuery expressions into bulk query plans formulated in the vanilla relational algebra, depicted in Table 1. All operators are well-known, except perhaps the row numbering operator ρ, which is similar to the SQL:1999 operator DENSE_RANK: $\rho_{b:\langle a_1,\ldots,a_n\rangle/p}(q)$ that assigns each tuple in q a rank (i.e. number), which is saved in column b. The constraint for the enumeration is the implicit order or q by the columns $a_1,\ldots,a_n$. Numbers consecutively ascend from 1 in each partition defined by the optional grouping column p.

Representing sequences as tables. The evaluation of any XQuery expression yields an *ordered sequence* of $n \geq 0$ items x_i, denoted $(x_1, x_2, \ldots, x_n)$. MonetDB/XQuery is a relational system, thus sequences are represented as tables, with schema pos|item. Since relations have (unordered) set-semantics, sequence order must be explicitly maintained using a `pos` column. In the XQuery data model, a single item x and the singleton sequence (x) are identical. Item x is represented as a single row table containing the tuple $\langle 1, x\rangle$. The empty sequence $()$ maps into the empty table.

pos	item
1	x_1
2	x_2
⋮	⋮
n	x_n

Loop-lifting. Each XQuery is translated bottom-up into a single relational algebra plan consisting only of the classical relational operations (select, project, join, etc); that is, the XQuery concept of nested `for`-loops is fully removed and a single bulk (=efficient and optimizable) execution plan is created.

The result of an XQuery at each step of bottom-up compilation is a relational plan that yields the result sequence *for each* nested iteration, all stored together. To make this possible, these intermediate tables have three columns: iter|pos|item, where `iter` is a logical iteration number, as shown in the tables below. For each scope, we keep a *loop* relation that holds all iters.[5] If we focus on the execution state in the innermost iteration body (marked as scope s_2) of $\mathcal{Q}_5$, there will be three such tables that represent the live variables `$x`, `$y` and `$z` respectively.

s_0
```
for $x in (10,20)
```
s_1
```
return for $y in (100,200)
```
s_2
```
let $z := ($x,$y)
return $z
```
$\mathcal{Q}_5$

As we can see from the `iter` columns, there are four iterations in scope s_2 (numbered from 1 to 4) and as expected, `$x` takes the value `10` in the first two iterations and the value `20` in the second two iterations. Similarly, `$y` takes the value `100` in the odd iterations and the value

loop

iter
1
2
3
4

x

iter	pos	item
1	1	10
2	1	10
3	1	20
4	1	20

y

iter	pos	item
1	1	100
2	1	200
3	1	100
4	1	200

z

iter	pos	item
1	1	10
1	2	100
2	1	10
2	2	200
3	1	20
3	2	100
4	1	20
4	2	200

[5] The *loop* relation allows to keep track of empty sequence values, encoded by the absence of tuples in the expression representation.

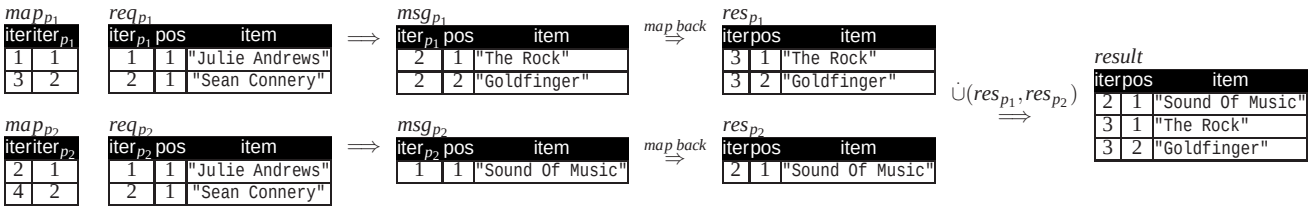


Figure 1: Relational Processing of Bulk RPC (Multiple Destinations Example)

`200` in the even ones. Finally, `$z` is a sequence of two values in all four iterations (having the value of `$x` concatenated with `$y`).

3.2 Bulk RPC

actor

iter	pos	item
1	1	"JulieAndrews"
2	1	"SeanConnery"

dst

iter	pos	item
1	1	"http://y.example.org/"
2	1	"http://y.example.org/"

Our earlier example query Q_2 contains a function application inside a `for`-loop. Inside this loop, the variables `$dst` and `$actor` yield relational tables shown left. Thus, the value of `$dst` is the same in both iterations of the `for`-loop, whereas `$actor` takes on values `"Julie Andrews"` in the first and `"Sean Connery"` in the second iteration.

SOAP XRPC Extension: Bulk RPC. The loop-lifted processing model of MonetDB/XQuery thus collects in a single table all XRPC function parameters needed by a remote function call, nested in one or more `for`-loops. This is exploited in SOAP XRPC by allowing *Bulk RPC*, in which a single XRPC message to the destination peer requests to perform *multiple* function calls. Each call is represented by an individual `xrpc:call` child element of the `xrpc:request`. Such a Bulk RPC also returns multiple results in the `xrpc:response` (one `xrpc:sequence` sequence for each call). From the shredded XRPC response message, it is quite straightforward to obtain the `iter|pos|item` table that represents an XDM result value for each iteration. Note that Bulk RPC exactly fits in the existing loop-lifted processing model of MonetDB/XQuery: without `execute at`, the local function translation mechanism already produced such a `iter|pos|item` table.

We show the `xrpc:request` part of the SOAP message in our Bulk RPC example, which contains two calls:

```
<xrpc:request module="films" method="filmsByActor" arity="1"
              location="http://x.example.org/film.xq">
 <xrpc:call> <!-- first call -->
  <xrpc:sequence>
   <xrpc:atomic-value
    xsi:type="xs:string">Julie Andrews</xrpc:atomic-value>
  </xrpc:sequence>
 </xrpc:call>
 <xrpc:call> <!-- second call -->
  <xrpc:sequence>
   <xrpc:atomic-value
    xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
  </xrpc:sequence>
 </xrpc:call>
</xrpc:request>
```

In the previous example the `execute at` expression `$dst` happened to be constant, such that all loop-lifted function calls had the same destination peer, and could be handled by the single Bulk RPC request above.

actor

iter	pos	item
1	1	"JulieAndrews"
2	1	"JulieAndrews"
3	1	"SeanConnery"
4	1	"SeanConnery"

dst

iter	pos	item
1	1	"http://y.example.org/"
2	1	"http://z.example.org/"
3	1	"http://y.example.org/"
4	1	"http://z.example.org/"

Let us now consider our other previous example Q_3. We now have an inner `for`-loop with four iterations, but `$dst` takes on two *different* values, identifying peers `y.example.org` and `z.example.org`, in respective the odd and even iterations. The general rule to translate a loop-lifted XRPC call is shown in Figure 2 and Figure 1 shows the intermediate steps taken. The system establishes a list of unique peers, and for each p extracts from each parameter `iter|pos|item` those iteration (tuples) that invoke the function on p. The resulting request tables (req_p) are used to generate a Bulk RPC to p. Observe that using ρ a new $iter_p$ column is created, and a mapping table (map_p) that maps old to new iteration numbers. The mapping table is then again used to map the new iteration numbers back into old ones, and all result tables (res_p) are united with a (merge-)union on the `iter` column, to guarantee the correct order of the result.

$$\text{iter pos item } result \Leftarrow \dot{\cup}_{\forall p \in \delta(dst.item)}(res_p)$$

with :

$$\text{iter pos item } res_p = \pi_{iter,pos,item}(\bowtie_{iter_p = iter_p}(msg_p, map_p))$$

$$\text{iter iter}_p \; map_p = \pi_{iter,iter_p}(\rho_{iter_p}(\sigma_{item=p}(dst)))$$

$$\text{iter}_p \text{ pos item } msg_p = f(req_p^1, \cdots, req_p^n)@p$$

$$\text{iter}_p \text{ pos item } req_p^i = \pi_{iter_p,pos,item}(\rho_{pos}(\bowtie_{iter=iter}(map_p, param_i)))$$

$$\text{execute at } \{\text{iter pos item } dst\}$$

$$\{ f(\text{iter pos item } param_1, \cdots, \text{iter pos item } param_n) \} \Rightarrow \text{iter pos item } result$$

Figure 2: Relational Translation of XRPC

Parallel & Out-Of-Order. The XRPC execution in Figure 1 performs two Bulk RPC calls. The first call processes both values of `$actor` on `y.example.org`. Then a second call performs the same task on `z.example.org`. It is important to observe that this order of processing is different than what is suggested by the query (i.e. first Julie Andrews on both, then Sean Connery on both). If a loop-lifted XRPC function application has multiple destination peers, in fact MonetDB/XQuery improves performance by dispatching all Bulk RPC requests *in parallel*, which makes the exact order in which peers execute the query unpredictable. After all parallel results are united, the mapping of temporary $\texttt{iter}_p$ numbers into `iter`s guarantees that the final result is produced in the correct order.

The out-of-order processing effects of loop-lifting are most easily explained in a single-destination (hence non-parallel) query:

```
import module namespace f="films" at "http://x.example.org/film.xq";
for $name in ("Julie", "Sean")
let $connery := concat($name, " ", "Connery")
let $andrews := concat($name, " ", "Andrews")                    (Q6)
return (
  execute at {"xrpc://y.example.org"} {f:filmsByActor($connery)},
  execute at {"xrpc://y.example.org"} {f:filmsByActor($andrews)} )
```

Here, only the peer `y.example.org` is involved twice within the same query due to sequence construction. In the first Bulk RPC call, it will look for films by two actors with surname Connery, resp. surname Andrews in the second RPC. Note that the intuitive order suggested by the query would be to look for actors by the name Julie first, and those named Sean second.

The above is also a good example of a query that needs *isolation*, because it handles two RPC requests inside the same query. While in this particular case, those two requests could potentially be combined, this is much harder if two different functions would be executed, or downright impossible if the parameters of one de-

	No Function Cache		With Function Cache	
	$x=1	$x=1000	$x=1	$x=1000
one-at-a-time	133	2696	2.6	2696
bulk	130	134	2.7	4

Table 2: XRPC Performance (msec): loop-lifted v.s. one-at-a-time; function cache v.s. no function cache

pend on the outcome of the other. Certain classes of queries, such as those that contain only a single non-nested XRPC call, can be easily identified at compile time to send at most one XRPC request to each destination peer. For such queries, we can use the cheaper XRPC mechanism without `queryID` (see Section 2.2), while still guaranteeing repeatable reads.

Note that without Bulk RPC, the costly isolation mechanism would be required for any XRPC that performs more than a single XRPC call. Thanks to Bulk RPC, many queries have to send just a single message to each peer, thus not only reducing the number of network I/Os, but also lessening the overhead of isolation.

3.3 Performance Evaluation

We conducted some experiments to evaluate the performance of XRPC in MonetDB/XQuery. The test setup consisted of two 2GHz Athlon64 Linux machines connected on 1Gb/s Ethernet.

Efficiency of Loop-lifting. To study the effect of loop-lifting, we define an `echoVoid` function and call it over XRPC while varying the number of iterations:

```
module namespace tst = "test";
declare function tst:echoVoid() { () };

import module namespace t="test" at "http://x.example.org/test.xq";
for $i in (1 to $x)
  return execute at {"xrpc://y.example.org"} {t:echoVoid()}
```

While in MonetDB/XQuery loop-lifting of XRPC calls (i.e. Bulk RPC) is the default, we also implemented a one-at-a-time RPC mechanism for comparison. The left half of Table 2 (the "No Function Cache" column) shows the experiment where we compare performance of Bulk RPC with single RPC at-a-time, while varying the number of loop iterations `$x`. It shows that performance is identical at `$x=1`, such that we can conclude that the overhead of Bulk RPC is small. At `$x=1000`, there is an enormous difference, caused by *(i)* serialization/deserialization of the request/response messages, *(ii)* network communication cost and *(iii)* overhead of function call (1000 calls instead of 1 call). This is easily explained as the one-at-a-time RPC experiment involves performing 1000 times more synchronous RPCs.

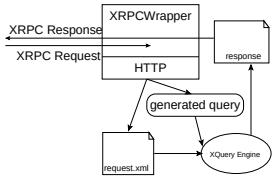
Throughput. We also carried out bandwidth experiments (details omitted for space) that scaled request and response payloads. Here we observed throughput of 8MB/s (large requests) and 14 MB/s (large responses), which correspond roughly with resp. the document shredding and serialization speed of MonetDB/XQuery [9]. Thus, like other SOAP-based messaging [16], XRPC data throughput on a fast local 1Gb network is CPU-bound rather than network-bound (though in a WAN it is likely to be the other way around).

Function Cache. XQuery Modules have the advantage that they may be pre-loaded and cached, and our choice to let XRPC use modules as the query transport mechanism also opens the possibility to reap performance profit from module pre-processing.

The feature of *prepared queries* is well-known for RDBMS, allowing a parametrized query plan to be parsed and optimized offline, such that an application can quickly enter actual parameters in the prepared plan and execute it. The ODBC and JDBC APIs export this functionality of relational databases using a programming language binding. MonetDB/XQuery has a mechanism for supporting prepared queries that does not need specific API support. Exploiting the fact that a prepared query is in essence a function with parameters, MonetDB/XQuery *caches* all query plans for (loop-lifted) function calls, for functions defined in XQuery Modules. Queries that just load a module and call a function in it with constant values as parameter, are detected by a pre-parser. The pre-parser then extracts the function parameters, and feeds them into a cached query plan. In MonetDB/XQuery, queries on small data sets can be accelerated ten-fold by this mechanism [9]. Note, that the function cache is **not** a query cache: queries are executed always on the latest data, and the performance improvement stems solely from the fact that query translation and optimization is avoided.

This same function cache mechanism is used by the XRPC request handler. This means that in MonetDB/XQuery an XRPC request usually does not need query parsing and optimization, just execution. The right half of Table 2 (the "With Function Cache" column) shows the impact of enabling the function cache: we see the processing time go down by 130ms (XQuery module translation time), improving both the single- and many-iteration Bulk RPC experiments. Thanks to the function cache, MonetDB/XQuery can achieve a minimum RPC latency of 3 msec – which is identical to that of commercial-strength software like .NET ([16, 27]).

4. XRPC WRAPPER



Cross-system distributed XRPC querying can be achieved even without XRPC being integrated into an XQuery processing engine. What is needed is a simple *XRPC wrapper* on top of the XQuery system. The XRPC wrapper is a SOAP service handler that stores the incoming SOAP XRPC request message in a temporary location, generates an XQuery query for this request, and executes it on an XQuery processor. The generated query is crafted to compute the result of a Bulk XRPC by calling the requested function on the parameters found in the message, and to generate the SOAP response message in XML using element construction. Such an XRPC wrapper only allows to *handle* calls with normally XRPC-incapable systems, but obviously does not allow to make outgoing XRPC calls from them.

We illustrate how such an XRPC wrapper works by an example. The following function returns the `person` node from an XMark document (`$doc`) whose `@id` attribute matches a given `$pid`:

```
declare function getPerson($doc as xs:string,
                           $pid as xs:string) as node()?
{ zero-or-one(doc($doc)//person[@id=$pid]) };
```

Figure 3 shows the query generated by XRPC wrapper to handle the `getPerson` request.

The XRPC protocol includes information about the arity of the function (as well as its return type), so it is easy to generate the right amount of *param* parameters in the call. The brunt of the work is done by the `n2s()` and `s2n()` marshaling functions, introduced in Section 2.2. These functions (omitted for reasons of space), can be implemented purely in XQuery.

The `n2s()` function, used here to process all parameters, needs to convert a SOAP XRPC element into an item sequence, where each item has the right type. This is done by going over all children of the `xrpc:sequence` using a series of `if..then` XQuery statements that select on the `xsi:type` attribute found in the `xrpc:atomic-value` nodes. In case of `xrpc:element` nodes with an `xsi:type`, XQuery validation is performed. The `s2n()` function is used here only to convert the function return value into a correct SOAP XRPC node. It iterates over the input item sequence, and for each item uses an

```
import module namespace func = "functions"
               at "http://example.org/functions.xq";

declare namespace env = "http://www.w3.org/2003/05/soap-envelope";
declare namespace xrpc = "http://monetdb.cwi.nl/XQuery";

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://monetdb.cwi.nl/XQuery
                        http://monetdb.cwi.nl/XQuery/XRPC.xsd">
<env:Body>
<xrpc:response xrpc:module="functions" xrpc:method="getPerson">{

  for $call in doc("/tmp/requestXXX.xml")//xrpc:call
  let $param1 := n2s($call/xrpc:sequence[1])
  let $param2 := n2s($call/xrpc:sequence[2])
  return s2n(func:getPerson($param1,$param2))

}</xrpc:response>
</env:Body>
</env:Envelope>
```

Figure 3: XQuery generated for the getPerson() XRPC request

XQuery `typeswitch()` to generate the right SOAP node. If the return type is a sequence of nodes that have a schema type (this information is supplied in the SOAP request) we insert the correct `xsi:type` attribute in it.

Saxon Experiments. Using the wrapper, we can run a number of experiments on the Saxon XSLT/XQuery processor [1] (Saxon-B 8.7). Like the experiments in Section 3.3, we put the `execute at` inside a `for`-loop with a varying number of iterations (`$x`) to study the performance impact of Bulk RPC. By absence of a function cache, Saxon latency is dominated by startup and compilation time, so we focus here on the internal Saxon timings (compile, treebuild, exec) and disregard network communication cost, which is a few msec at most. For the `echoVoid` experiment, we see that Bulk RPC again allows to amortize XRPC latency really well: instead of 1000 times the latency, with a 1000 times more work total latency increases just over a factor 2. As the execution time still is increased by a factor 30, the low impact is due to other amortized latencies, in parsing the XML request document, compiling the query, etc.

We also show the results of the `getPerson()` example above. This exposes an additional benefit of Bulk RPC over just amortized fixed latencies: whereas in the single-call case, `getPerson()` behaves like a selection over the XMark document, the Bulk version of `getPerson()`, that iterates over all calls in the request, becomes an *equi-join*. Again, the total time for a Bulk RPC with 1000 calls is only about twice as much as a single call, but here we see that the execution time impact has increased only by a factor of 3 (was 30 in `echoVoid`). The explanation is that Saxon is able to detect the join condition and builds a hash-table such that performance remains linear in the size of the XMark document, just like it was in the single call selection.

5. DISTRIBUTED XQUERY WITH XRPC

One of the design goals of XRPC is to have it serve as the target language for a distributed XQuery optimizer that takes queries without XRPC calls as input (hence, only data shipping) and produces a decomposed query as output that uses XRPC for function shipping. In this section, we show how some well-know distributed query execution strategies, such as distributed semi-join, can be elegantly expressed in XRPC. We also outline several future work issues in the area of automatic query distribution techniques using functional decomposition.

Let us assume a distributed XDBMS system with two peers $\{p_a, p_b\}$. An XMark document is distributed between these two peers, with p_a stores all persons in "`persons.xml`", and p_b stores all items and (open/closed) auctions in "`auctions.xml`".

	total	compile	treebuild	exec
echoVoid $x=1	275	178	4.6	92
echoVoid $x=1000	590	178	86	325
getPerson $x=1	4276	185	1956	2134
getPerson $x=1000	8167	185	1973	6010

Table 3: Saxon Latency via the XRPC Wrapper (msec)

```
for $p  in doc("persons.xml")//person,
    $ca in doc("xrpc://B/auctions.xml")//closed_auction          (Q7)
where $p/@id = $ca/buyer/@person
return <result>{$p,$ca/annotation}</result>
```

The above query is executed at peer p_a. For each person and for every item this person has bought, query Q_7 returns the `person` node and the `annotation` node of the bought item in a new `result` node. For the moment, assume that `fn:doc()` is invoked with a compile-time known constant URI from our `xrpc://` URI name scheme, which indicates the peer supports XRPC.

Predicate Pushdown. A first heuristic optimization is to push predicates that depend only on a single `fn:doc('xrpc://p/..')` into data source p. Thus, instead of transferring the whole document "`auctions.xml`" from p_b to p_a, we define a function to return all `closed_auction` nodes and execute this function on p_b

```
module namespace b = "functions_b";
declare function b:Q_B1() as node()*
{ doc("auctions.xml")//closed_auction };

--- Rewritten query Q7_1 ---
import module namespace b="functions_b" at "http://example.org/b.xq";

for $p  in doc("persons.xml")//person,
    $ca in execute at {"B"} { b:Q_B1() },
where $p/@id = $ca/buyer/@person
return <result>{$p,$ca/annotation}</result>
```

This heuristic rewrite can simply be triggered by the presence of `fn:doc()`. The required analysis how much of the XQuery (Core) expression is dependent on that `fn:doc()` alone, and can be pushed, is highly similar to [25].

Advanced Pushdown. We could even push expressions that depend on a `fn:doc()` application even if that function application has a non-constant URL argument, and even could depend on a `for`-loop variable. That is, using the helper functions:

```
declare function xrpc:host ($url as xs:string) as xs:string
declare function xrpc:path ($url as xs:string) as xs:string
```

where by default `host()` returns `"localhost"` and `path()` returns its argument – except for `xrpc://` URLs, where they would separate the URL in a host prefix and path suffix – we could rewrite calls to `fn:doc($url)` into:

```
execute at { xrpc:host($url) } { fn:doc(xrpc:path($url)) }
```

This approach, however, does require a refinement of the work in [25]. One must bear in mind that any of the rewrites discussed here should only be made by an automatic rewriter *if* it can establish that the call-by-value semantics of XRPC will not compromise the semantics of the query. This at least involves a check whether nodes that come from pushed expressions are only navigated downwards, and also involves checking against node identity tests and order-dependent (XPath, order by) processing of node sequences that stem from multiple `fn:doc()` calls pushed to different sources.

Execution Relocation. The possibilities of query rewriting do not stop at push-down of `fn:doc('xrpc://..')`-dependent expressions. Even if a query depends on a set $\mathcal{P}$ XRPC peers that contribute documents, one could decide to select one peer p_i from $\mathcal{P}$ and put *all* execution on p_i. We call this mechanism *Execution Relocation*. For example, it might be beneficial to relocate all execution on p_b, if "`auctions.xml`" is much larger than "`persons.xml`":

	Total Time	MonetDB Time	Saxon Time
data shipping	28122	16457	11665
predicate push-down	25799	2961	22838
execution relocation	53184	69	53115
distributed semi-join	10278	118	10160

Table 4: Execution time (msecs) of query $\mathbb{Q}_7$ distributed on MonetDB/XQuery and Saxon (Saxon Time includes network).

```
module namespace b = "functions_b";
declare function b:Q_B2() as node()*
{ for $p  in doc("xrpc://A/persons.xml")//person,
      $ca in doc("auctions.xml")//closed_auction
  where $p/@id = $ca/buyer/@person
  return <result>{$p, $ca/annotation}</result>
};
```

Then peer p_a needs only to call this function to get the results:

```
import module namespace b="functions_b" at "http://example.org/b.xq";
execute at {"B"} { B:Q_B2() }
```

Distributed Semi-join. The classical distributed semi-join strategy [7, 34] can be employed as well. The XRPC equivalent of the semi-join strategy uses a XRPC function call with a loop-dependent parameter. In this case, the person `@id` for all persons could be passed in a loop to a function executed at p_b that returns those closed auctions with buyers having that `@id`:

```
module namespace b = "functions_b";
declare function b:Q_B3($pid as xs:string) as node()*
{ doc("auctions.xml")//closed_auction[./buyer/@person=$pid] };

--- Rewritten query Q7_3 ---
import module namespace b="functions_b" at "http://example.org/b.xq";

for $p  in doc("persons.xml")//person
let $ca := execute at {"B"} {b:Q_B3($p/@id)}
return if(empty($ca)) then ()
       else <result>{$p, $ca/annotation}</result>
```

This shows that federating data sources with XRPC (even via the XRPC Wrapper) is more powerful than the "wrapper-architecture" [22] used in federated database systems. Such wrappers typically lack the possibility to push table-valued parameters into data sources, which is required for the semi-join optimizations.

Saxon and MonetDB/XQuery Joined by XRPC. To demonstrate the interoperability, expressiveness and performance potential of XRPC we run query $\mathbb{Q}_7$ on two peers using all four mentioned strategies. On peer p_a (the local peer), we run MonetDB/XQuery with the document "persons.xml" (1.1MB, 250 `person` nodes); on peer p_b the Saxon XSLT/XQuery processor with the document "auctions.xml" (50MB, 4875 `closed_auction` nodes). There are 6 matches between the `person` nodes and the `closed_auction` nodes.

All communication between MonetDB/XQuery and Saxon happens via XRPC. The XRPC wrapper described in Section 4 is used to generate the XQuery query from an XRPC request message.

The measured execution times are shown in Table 4. In the column "MonetDB Time" are execution times on peer p_a and in the column "Saxon Time" are execution times on peer p_b. The Saxon time was measured by subtracting MonetDB time from total time, such that it also included communication. We should stress that this experiment is not a rigorous evaluation of distributed query execution strategies, rather a demonstration of the possibilities of XRPC. The results here show that the "data shipping" query is relatively expensive, since it spends quite some Saxon time on shipping the 50MB document and then still needs to do the join. The "predicate push-down" approach improves the performance, as we would expect. The "execution relocation" largely relieves the MonetDB peer from execution responsibilities, but still ships a significant amount of data and tasks Saxon with the whole join and result construction effort (where it takes longer than on MonetDB). The "distributed semi-join" is the strategy that incurs least data shipping, and is most efficient in this case.

6. RELATED WORK

In the area of extending XQuery with distributed querying, our work is mostly related to Galax DXQ [14], Active XML [2, 8, 6], Galax Yoo-Hoo! [28], XPeer [31] and XQueryD [30].

DXQ [14] is developed as an extension of Galax. We are not aware of a formal semantics defined in DXQ, especially considering updates. Also, XRPC is intended for use in P2P projects and targets any data source that supports XQuery, whereas DXQ depends on distributed query plans, in terms of the internal Galax execution algebra, generated by the Galax optimizer. This will have certain advantages, such as better control over the capabilities of the distributed nodes and possibly better physical plans and optimization, but the use of an internal algebra makes it much harder to achieve cross-system DXQ.

In Active XML (AXML) [6], calls to Web or AXML service functions are embedded in XML documents. The evaluation of a service call results in an XML fragment, which is inserted into the original XML document, and gets re-evaluated (allowing for nesting in case of AXML service calls). AXML has shown the value of distributed query optimization, identifying lazy evaluation schemes and various rewrite strategies [5]. Service functions are defined in AXML using an XML query language (X-OQL in the open-source implementation [2]), which itself does not allow distributed evaluation (an embedding piece of AXML is always needed). The SOAP protocol used for AXML services has not been specified formally; like XRPC it uses a document/literal encoding to represent XML subtree values. We think the ideas proposed in XRPC, namely *(i)* a well-defined XQuery extension that allows specification of distributed queries *inside* the service function , *(ii)* a semantics for distributed XQUF transactions, and *(iii)* a SOAP network protocol that supports the full XQuery Data Model as well as Bulk RPC, could all be exploited in AXML.

Galax Yoo-Hoo! [28] is related to our work in the sense that web services are accessed using remote procedure calls and SOAP messages are used as the communication protocol. As Yoo-Hoo! focuses on web service calls, it uses the simple SOAP RPC protocol, which lacks support for XML element (sub-tree) parameters and sequences, as well as Bulk RPC. Another difference is that Galax Yoo-Hoo RPC calls support only *one* fixed destination URI for each imported web service module (not a computed one).

XPeer [31] is a P2P XDBMS for sharing and querying XML data, on top of a super-peer network. The query algebra of XPeer takes explicitly into account data dissemination, data replication and data freshness. We think our future work on query decomposition can build on some of the techniques employed in XPeer.

The syntax of XRPC is inspired by that of XQueryD [30], which support shipping of free-form XQuery queries. XQueryD thus requires a runtime rewriter to scan the XQuery expressions in the `execute` statement for variables and substitute them with the runtime values. Such a rewriter is not needed in XRPC, since the binding of the parameters of an XRPC function application is done by the compiler as if it was a normal function application. Unlike XRPC, XQueryD does not define an network protocol.

In the area of distributed query processing and transactions, much prior work is surveyed in [22, 34] and parts of the book of [29]. Distributed snapshot isolation has received some attention in federated situations with a single coordinator [32].

7. CONCLUSION

In this paper, we presented XRPC, a minimal XQuery extension that enables distributed query execution with a focus on interoperability. We first gave a formal definition of the syntax and the semantics of XRPC, including the semantics of distributed updates, that follow from the use of XQuery Updating Functions over XRPC. This includes the definition of two isolation levels for read-only and updating XRPC queries.

Since interoperability is our goal, the XRPC proposal also comprises a message protocol, which we chose to base on SOAP. Such a SOAP protocol has the additional advantage of seamless integration with web services and AJAX-based GUIs. To enhance adoption of XRPC, we described a XRPC wrapper that allows any XQuery data source to handle XRPC calls.[6]

Our experiences in MonetDB/XQuery suggest that adding XRPC to existing XML database systems is easy; as shredding, serialization and HTTP functionality are usually already present the work is limited to a small parser extension and stub code generation.

The SOAP XRPC protocol supports the concept of Bulk RPC; the execution of multiple function calls in a single message exchange. This amortizes network and parsing latencies, and can make XRPC a quite efficient communication mechanism. We have shown that the loop-lifting technique, pervasively applied in our MonetDB/XQuery system for the translation of XQuery expressions to relational algebra can easily generate such Bulk RPC requests. During our Saxon experiments, we also saw that Bulk RPC enables set-oriented optimizations, such that Bulk RPC execution of a selection function can be handled using a join strategy.

In the near future, we plan to work on a distributed query optimizer that converts non-XRPC data shipping queries into function-shipping queries using XRPC. In this paper, we showed some initial experiences in distributed query execution, where MonetDB/XQuery and Saxon cooperated via XRPC. To allow automatic query rewrites into distributed plans, future research needs to address the problem of finding rewrites that hide the by-value parameter passing of XRPC, to correctly preserve XQuery node identity semantics. Other future work goes into applying XRPC for P2P data management [10], both in the area of designing a sound yet practical P2P update semantics, as well as integrating XRPC with advanced P2P data structures such as Distributed Hash Tables (DHTs).

8. REFERENCES

[1] SAXON The XSLT and XQuery Processor. http://saxon.sourceforge.net.

[2] The ActiveXML Project. http://activexml.net.

[3] Web Services Atomic Transaction (WS-AtomicTransaction), August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.

[4] Web Services Coordination (WS-Coordination), August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[5] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for active xml. In *SIGMOD*, pages 527–538, 2004.

[6] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed xml data management. In *EDBT*, March 2006.

[7] P. Apers, A. Hevner, and S. Yao. Optimization algorithms for distributed queries. *IEEE Trans. Software Eng.*, 9(1):57–68, 1983.

[8] O. Benjelloun. *Active XML: A data-centric perspective on Web services*. PhD thesis, September 2004.

[9] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, June 2006.

[6]XRPC and the XRPC wrapper are available in the open-source XDBMS MonetDB/XQuery (www.monetdb.nl).

[10] P. Boncz and C. Treijtel. AmbientDB: relational query processing in a P2P network. In *DBISP2P*, Sep. 2003.

[11] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update Facility. W3C Working Draft 11 July 2006. http://www.w3.org/TR/2006/WD-xqupdate-20060711.

[12] F. Cohen. Discover SOAP encoding's impact on web service performance, March 2003. http://www-128.ibm.com/developerworks/webservices/library/ws-soapenc.

[13] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Candidate Recommendation 8 June 2006. http://www.w3.org/TR/2006/CR-xquery-semantics-20060608.

[14] M. Fernández, T. Jim, K. Morton, N. Onose, and J. Siméon. Highly distributed xquery with dxq. In *SIGMOD demo*, June 2007.

[15] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Candidate Recommendation 11 July 2006. http://www.w3.org/TR/2006/CR-xpath-datamodel-20060711.

[16] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. van Engelen, and M. J. Lewis. Toward Characterizing the Performance of SOAP Toolkits. In *GRID '04*, 2004.

[17] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

[18] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, pages 252–263, September 2004.

[19] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation 24 June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624.

[20] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation 24 June 2003. http://www.w3.org/TR/2003/REC-soap12-part2-20030624.

[21] V. Josifovski and T. Risch. Query Decomposition for a Distributed Object-Oriented Mediator System. *Distributed and Parallel Databases*, 11(3):307–336, 2002.

[22] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[24] S. Lyubka. SHTTPD: Simple HTTPD. http://shttpd.sourceforge.net.

[25] A. Marian and J. Siméon. Projecting XML Documents. In *VLDB*, September 2003.

[26] N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer. W3C Recommendation 24 June 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624.

[27] A. Ng, S. Chen, and P. Greenfield. An Evaluation of Contemporary Commercial SOAP Implementation. In *AWSA*, April 2004.

[28] N. Onose and J. Siméon. XQuery at Your Web Service. In *WWW*, pages 603–611, 2004.

[29] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., NJ, USA, 1999.

[30] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *IIWeb*, pages 116–121, September 2004.

[31] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self -Organizing XML P2P Database System. In *EDBT Workshops*, 2004.

[32] R. Schenkel, G. Weikum, N. Weißenberg, and X. Wu. Federated transaction management with snapshot isolation. In *Proceedings of the 8th International Workshop on Foundations of Models and Languages for Data and Objects - Transactions and Database Dynamics '99*, Dagstuhl Castle, Germany, 1999.

[33] C. Thiemann, M. Schlenker, and T. Severiens. Proposed Specification of a Distributed XML-Query Network. *CoRR*, cs.DC/0309022, 2003.

[34] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, 1984.

[35] Y. Zhang and P. A. Boncz. Loop-Lifted XQuery RPC with Deterministic Updates. Technical Report INS-E0607, CWI, Amsterdam, The Netherlands, November 2006.