XRPC: efficient distributed query processing on heterogeneous XQuery engines
Zhang, Y.

Citation for published version (APA):
Zhang, Y. (2010). XRPC: efficient distributed query processing on heterogeneous XQuery engines

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
In this chapter, we present techniques to automatically decompose any XQuery query – including updating queries specified by XQUF – into subqueries, that can be executed near their data sources, i.e., function-shipping. The main challenge addressed here is to ensure that the decomposed queries properly respect XML node identity and preserve structural properties, when (parts of) XML nodes are sent over the network, effectively copying them. We first precisely characterise the conditions, under which pass-by-value parameter passing causes semantic differences between remote execution of an XQuery expression and its local execution. We then formulate a conservative strategy that effectively avoids decomposition in such cases. To broaden the possibilities of query distribution, we extend the pass-by-value semantics to a pass-by-fragment semantics, which keeps better track of node identities and structural properties. The pass-by-fragment semantics is subsequently refined to a pass-by-projection semantics by means of a novel runtime XML projection technique, which safely eliminates semantic differences between the local and remote execution of an XQuery expression, and strongly reduces message sizes. Finally, we discuss how these techniques can be used for updating queries, both under the standard W3C XQUF specification, as well as under an extended semantics that allows updating remote documents. The proposed techniques are implemented using XRPC. Experiments on MonetDB/XQuery establish the performance potential of our XQuery decomposition techniques.

5.1 Motivation

Decomposing queries to address multiple data sources is a well-studied optimisation problem in relational [175], object-oriented [115, 105], and semi-structured databases [166, 167]. While it is natural (and correct) to assume that many of the existing techniques can be carried over, the XML data model and the XQuery language introduce a number of particular challenges not met elsewhere that revolve around XML node identities and structural (rather than value-based) relationships between nodes. Previous work on distributed XML [53, 63, 170] only focused on a restricted subset of XQuery queries, and did not address the problem of transparent query decomposition, such that these challenges did not arise.

In this chapter, we introduce ways to decompose any XQuery query that consults multiple XML documents residing on multiple peers into subqueries that can be executed on those peers, i.e., function shipping. In principle, we do not want to restrict the form of these queries in any significant way: the full W3C recommended XQuery language [38] including its XQUF extension [58] is the starting point of our decomposition. Our only requirement for
peers to participate is running an XML database system (XDBMS) that complies with these W3C recommendations. The goal of this chapter is to exploit the computational power of heterogeneous XML engines on the Web to jointly execute XQuery and XQUF queries. In our decomposition, we use a functional abstraction, which is a good match for XQuery, as it is a functional language. This provides ultimate flexibility in the way queries can be decomposed. One can chop up an XQuery query in any possible way, view the chops as function compositions, and potentially execute each of these functions on a different peer.

**Shipping XML Messages** Without loss of generality, we view the subexpressions to be executed by remote peers as XQuery functions that may have parameters and produce a result. During remote function execution, the calling peer (i.e., query originator) will send a request message containing parameters to a remote peer, which executes the subexpression, and sends back a response message containing the result. When XML nodes must be shipped over the network, pieces/snippets of the XML documents must somehow be copied into the messages, changing the “holistic” structural properties and identities of nodes, which may affect the semantics of XQuery execution on such shipped nodes. In Section 5.2, we identify all semantic differences between evaluating XQuery expressions on the original XML nodes and on the shipped (i.e., copied) nodes. Alternatively, one can choose to ship the entire XML document in order to preserve all structural relationships (which defeats the purpose of function shipping). Naively, when shipping a node, one would ship its descendants (i.e., XML subtree), but other solutions are also possible, and will in fact be proposed in this chapter (especially, the idea to use XML projection techniques). In particular, the run-time projection approach contributed in this chapter tunes the shape of the shipped XML messages to the characteristics of the query, such that a minimal amount of data is shipped and those structural relationships that are actually needed are preserved.

**XRPC** While our problem statement covers distributed XQuery in general, the techniques proposed in this chapter stem from the particular context of XRPC. As XQuery is a compositional functional language, each query can be chopped up in arbitrary pieces. One can then view the pieces as functions connected together by function parameters and results. With XRPC, we have in principle ultimate flexibility in the way queries can be decomposed, as it allows each function to be executed on an arbitrary peer. An important feature on the network protocol level is Bulk RPC that allows multiple calls to the same function (with different parameters) to be handled in a single network interaction. Bulk RPC is exploited when a query contains a function call nested in an XQuery `for`-loop, which in a naive implementation would lead to as many synchronous RPC network interactions as loop iterations.

Figure 5.1 shows a query $Q$ that performs a single XRPC function call to $\text{fcn}(\cdot)$ with a single parameter (a node $\$n$ from some document $\mathcal{D}$). To make an XRPC call, the local
peer formulates a SOAP request message which contains a deep copy \( P \) of the node \( \$n \). The Simple Object Access Protocol (SOAP) is an XML-based message format commonly used by web services [128, 89, 90]. XRPC follows the previously mentioned approach of copying the XML subtree of a node parameter, which implies a pass-by-value parameter passing strategy. The message is sent as a synchronous HTTP POST request. The remote peer runs an HTTP server, which parses the request message and constructs a separate XML fragment for each node parameter (in this example a single fragment \( P' \)). The remote peer then evaluates the function and serialises the result into a response message (here, a deep copy of the result node, denoted \( R \)). Finally, the local peer parses the response message and constructs a separate XML fragment for each node-typed result (here \( R' \)), which is the result of \( Q \).

**Problem Statement** Our goal is to rewrite an XQuery \( Q \) that uses XML documents with \( \text{xrpc://} \) URIs stored at remote peers, into an equivalent query \( Q' \) that uses XRPC calls to execute parts of the query (expressed as XQuery functions) on those remote peers. For a query \( Q \), \( Q(D) \) denotes the result of evaluating \( Q \) over a (possibly distributed) database \( D \). Two queries \( Q \) and \( Q' \) are equivalent, if \( Q(D) = Q'(D) \) for any given database \( D \) (under the XQuery deep-equal semantics).

We illustrate XQuery decomposition as follows:

```xml
for $e in doc("employees.xml")//emp
where $e/@dept = doc("xrpc://example.org/depts.xml")//dept/@name
return $e
```

the URL \( \text{xrpc://example.org/depts.xml} \) implies that the remote peer \( \text{example.org} \) supports XRPC, so the predicates could be pushed as:

```xml
declare function fcn($n as xs:string) as xs:boolean
[$n = doc("depts.xml")//dept/@name];

for $e in doc("employees.xml")//emp
where execute at {"example.org"} {fcn($e/@dept)}
return $e
```

In this example, the parameter and return value of the function \( \text{fcn}() \) are of atomic types. In more complex cases, nodes may be involved, such that potential semantic differences due to pass-by-value should be considered (discussed in Section 5.2), which is our main challenge.

### 5.2 Semantic Differences with Pass-By-Value

There are well-defined semantic differences [180] between evaluating an XQuery expression locally and executing it remotely under pass-by-value parameter passing. We discuss these
differences with a query $Q_1$ in Table 5.1. This query evaluates three functions: `makenodes()`, `overlap()` and `earlier()`.

**Problem 1: Non-downward XPath Steps** Reverse and horizontal XPath axis navigation (e.g., `parent`, `ancestor`, `preceding(-sibling)` and `following(-sibling)`) from remote function parameters always produces empty results, as pass-by-value node serialisation only includes the descendants of a node inside the message. Consider the following:

```xquery
7 let $bc := execute at {"example.org"} {makenodes()}, $abc := $bc/parent::a
```

here, $abc$ evaluates to the empty sequence, instead of the correct $\langle a \rangle \langle b \rangle \langle c/ \rangle \langle /b \rangle \langle /a \rangle$.

It is possible to evaluate downward XPath steps on a sequence of remote nodes, but only if we are sure that these nodes are ordered and non-overlapping (otherwise, the results of such XPath steps will fail to respect node identity and order, as described below).

**Problem 2: Node Identity Comparisons** If a remote function returns a sequence with twice the same node, or the same node is passed twice as function parameters, pass-by-value represents them as two different copies. This leads to problems with duplicate elimination (as shown in Problem 4 below) and any node identity comparison will always yield false. For instance:

```xquery
10 where execute at {"example.org"} {overlap($first, $node)}
```

yields false, while the local query evaluation gives true.

**Problem 3: Document Order** The parameters of a function call on a remote peer are serialised into the message in parameter order, in separate XML fragments. Even if the parameter nodes are disjoint (making Problem 2 irrelevant), the relative order between these XML fragments may differ from their original order. Thus, inter-parameter node comparisons ("≪", "≫") may behave differently from the local semantics. Consider the usage of `earlier()` in $Q_1$ as:

```xquery
9 let $first := execute at {"example.org"} {earlier($bc,$abc)}
```

In both iterations, the variable $first$ binds to a copy of $bc$, instead of $abc$, although $abc$ is the parent of $bc$.

Another problem with document order, not revealed by this example, could occur when comparisons of nodes from different XML documents are executed on remote peers. The XQuery/XPath Data Model (XDM) [71] defines that the relative order of nodes in different documents is implementation-dependent, but must be stable during the processing of the same query. Consider the query

```xquery
declare function earlier2() as boolean
{doc("xrpc://a.example.org/a.xml")/a ≪ doc("xrpc://b.example.org/b.xml")/b};
execute at {"a.example.org"} {earlier2()} = execute at {"b.example.org"} {earlier2()}
```

which, depending on how documents are ordered by the remote peers, could return true or false\(^1\), while XDM requires it to always return true. Note, however, that a query containing a single call to `earlier2()` may return either true or false, in accord with XDM. In such a query, `earlier2()` could be executed at a remote peer.

\(^1\)Even if the two calls to `earlier2()` were executed on the same remote peer, without any guarantees for consistency, the results could be different, since each call is a separate query on the remote peer.
CHAPTER 5. XQUERY DECOMPOSITION

Problem 4: Interaction Between Different Calls  Additional semantic differences can occur when XQuery subexpressions (sequences) may contain nodes that were obtained as results from different remote function calls, and these function calls, directly or indirectly, accessed the same XML document on some peer. Node sequences can become intermixed by any XQuery construct that accepts multiple inputs, namely: sequence construction, and the built-in functions `union`, `except`, and `intersect`. A special source of call-mixing is the return clause of a `for`-loop in which remote function evaluation is performed, because the return clause implicitly creates a sequence that concatenates the expression result of all loop iterations (each of which performed a semantically separate remote function call). The result of such “mixed-call expressions” is that nodes returned by different calls may in fact stem from the same document. However, node identity and ordering between nodes from different calls is not preserved, leading to semantic differences. For example, even if a downward XPath step is applied on an input sequence containing nodes obtained from different remote calls, the result can have the wrong order (placing the results from the first call always before those of the second call) and will fail to properly eliminate duplicates:

```
(for $node in ($bc, $abc)
  let $first := execute at "example.org" {earlier($node,$abc)}
return $node)//c
```

The above two XRPC calls produce nodes belonging to separate XML fragments. Under pass-by-value, evaluating `//c` produces two separate copies of `c` nodes, while in local execution the nodes returned from `earlier()` are from the same XML fragment, such that XPath steps return a duplicate-free result.

Problem 5: XQuery Built-in Functions  Various problems may occur when evaluating certain built-in functions remotely.

1. `static-base-uri()`, `default-collation()` and `current-datetime()`: depend on the static XQuery context.
2. `base-uri()` and `document-uri()`: depend on the dynamic context of node expressions.
3. `root()`: accesses the document root.
4. `id()` and `idref()`: return all nodes in a document with certain ID/IDREF values.
5. `lang()`: accesses the `xml:lang` attribute of the context node and its ancestors.

Class 1 of the above built-in functions is handled by extending the XRPC message format with extra attributes such that the remote side can declare identical values for these context attributes. Class 2 is dealt with by adding these properties as attributes in the XRPC nodes (such as `xrpc:element`) that enclose serialized parameter/result nodes in the SOAP messages. Use of the `fn:base-uri()` and `fn:document-uri()` in XRPC is substituted by `xrpc:base-uri()` and `xrpc:document-uri()` wrappers that take these attributes into account when invoked on XRPC parameter nodes. As solutions for Class 1-2 are available, the main problem with built-in functions is posed by Classes 3-5, which access non-descendants of parameter nodes, and thus cannot be supported by pass-by-value.

In the remainder, we present decomposition techniques and extensions to enhance the pass-by-value semantics that solve the aforementioned problems.

---

2If `static-base-uri()` is not set, we ship the value `xrpc://P/doc/`, so that `fn:doc()` calls with a relative document URI call back to the originating peer `P`. 
Table 5.2: XCore grammar rules

5.3 XQuery Core Rewrite Framework

XQuery Core [67] (abbreviated XCore) is a subset of XQuery in which all implicit operations are made explicit. We adopt a subset of XCore expressions in Table 5.2 which is sufficient to capture XPath 1.0 and XQuery FLWOR expressions [67]. Additionally, we support all updating expressions (rule `UpdExpr`) and the transform expression (`TransformExpr`) as defined by XQUF. We use a representation of XPath paths in our XCore grammar that keeps consecutive steps together, rather than nesting each step in a separate `for`-loop (when allowed – the use of `position()` precludes this). Such an optimisation is common in XQuery engines, and is part of XQuery normalisation, further described in Section 5.4. Additionally, we define two new rules for the XRPC extension [180]:

\[
\text{XRPCExpr} ::= \"execute\" \"at\" \"("ExprSingle\")\" \"function\" \text{XRPCParam} \"("Expr\")\"
\]

\[
\text{XRPCParam} ::= \text{Var} \ | \ \text{VarRef} \ | \ \text{VarRef}(\text{ParamList})
\]

Rule `XRPCExpr` identifies an `xrpc://` URI in expression `ExprSingle`, and declares a new anonymous function that is to be executed remotely. It should be noted that these grammar rules lack the expressive power to define recursive functions. This does not matter for XQuery decomposition, as our decomposition strategies will not generate recursive functions. It should also be noted that the syntax defined by the rules `XRPCExpr` and `XRPCParam` differs from the actual XRPC syntax ("execute at \{ExprSingle\}\{FunApp\(ParamList\)\}"). The syntax used here is only for presentation purposes to avoid the need to define all rules concerning declaration of user-defined functions. Thus, our simple XCore rule without explicit user-
CHAPTER 5. XQUERY DECOMPOSITION

Basic XQuery query

```
(let $s := doc("xrpc://A/students.xml")/people/person,
  $c := doc("xrpc://B/course42.xml"),
  $t := $s[tutor = $s/name]
for $e in $c/enroll/exam
where $e/@id = $t/id
return $e)/grade
```

XCore variant

```
(let $s := doc("xrpc://A/students.xml")/child::people/child::person return
let $c := doc("xrpc://B/course42.xml") return
let $t := for $x in $s return if ($x/child::tutor = $s/child::name) then $x else ()
return for $e in $c/child::enroll/child::exam
return if ($e/attribute::id = $t/child::id) then $e else ()/child::grade
```

Normalised XCore variant

```
(let $t := (let $s := doc("xrpc://A/students.xml")/child::people/child::person
  return for $x in $s return if ($x/child::tutor = $s/child::name) then $x else ()
return for $e in (let $c := doc("xrpc://B/course42.xml") return $c/child::enroll/child::exam)
return if ($e/attribute::id = $t/child::id) then $e else ()))/child::grade
```

Table 5.3: Example query $Q_2$

defined function declarations can express all queries in a single ExprSingle, which in turn can be mapped to a query graph. This simplifies the formulation of analysis steps.

### 5.3.1 XCore Dependency Graph

We introduce a dependency graph ($d$-graph) for an XCore query. Consider the XQuery query $Q_2$ in Table 5.3, which asks for the grade in course42 of students having a tutor who is also a student, and its XCore equivalence $Q_2^c$ normalised as $Q_2^n$.

A **dependency graph** is a directed, ordered and connected graph $G$ with vertices $V(G)$ and edges $E(G)$. Each vertex $v$ is denoted as $v_i$; $\text{rule}[\text{val}]$, where $v_i$ is a unique vertex identifier, $\text{rule}$ is the grammar rule represented by $v_i$, and $\text{val}$ is an optional value indicating the right-hand-side of $\text{rule}$. There is a single $v_{\text{root}}$ vertex without incoming edges. $E(G)$ consists of parse edges $E_p(G)$ and varref edges $E_v(G)$. Each parse edge is an ordered vertex pair $(u, v)$, where $u$ corresponds to a parsing rule $r_u$ that directly causes the use of another parsing rule $r_v$. A varref edge is an ordered vertex pair $(w, x)$ denoting a variable usage. When a VarRef rule is used, an additional edge is created between the VarRef vertex and the Var vertex that defines the variable.

**Example 5.3.1.** Figure 5.2 shows the $d$-graph of $Q_2^n$ in Table 5.3. Solid and dashed lines represent parse and varref edges, respectively. The variable binding in the first let expression corresponds to vertices $v_2, \ldots, v_{21}$, and vertices $v_{22}, \ldots, v_{39}$ depict its return clause. The edge $(v_6, v_7)$ is a parse edge. The edge $(v_{30}, v_{25})$ is a varref edge, as the variable used by $v_{30}$ is a reference of variable $\$c$ introduced by $v_{25}$. Thus, a $d$-graph is in essence a parse-tree with additional (dashed) edges to indicate variable usages.

We define three types of dependency relationships upon the reachability between two vertices $x, y$ in $V(G)$: (1) $x$ **“parse-depends on”** $y$, denoted as $x \xrightarrow{\text{parse}} y$, if $y$ is reachable from $x$ via only parse edges; (2) $x$ **“varref-depends on”** $y$, denoted as $x \xrightarrow{v} y$, if $y$ is reachable from $x$ via at least one varref edge; and (3) $x$ **“depends on”** $y$, denoted as $x \sim y$, if either $x \xrightarrow{\text{parse}} y$ or $x \xrightarrow{v} y$ holds. The compositional nature of XQuery means that $x \sim y$ concisely captures all semantic dependencies between subexpressions.
Figure 5.2: $d$-graph of the normalised XCore variant $Q_n^2$ in Table 5.3
Consider Figure 5.2, $v_{15}$ is a parse edge; $v_{16}$ is reachable from $v_{15}$ via $(v_{15}, v_{16})$, $v_{11}$ and $(v_{16}, v_{11})$ is a varref edge.

For a $d$-graph $G$ and a vertex $v_r \in V(G)$, we use the term subgraph to mean the vertex-induced subgraph of $r_s$, denoted $G_{r_s}$, including $r_s$ and all $u \in V(G)$ where $r_s \sim u$; $r_s$ is called the root of the subgraph. For instance, the subgraph rooted at vertex $v_{22}$ contains vertices $v_{22}, \ldots, v_{39}$, but does not contain vertices $v_3, \ldots, v_{21}$. Throughout this chapter, we use the terms (sub)graph and (sub)query interchangeably, as a (sub)query is represented by the induced subgraph rooted at some vertex.

### 5.3.2 XRPCExpr Insertion

We can decide to evaluate a certain subgraph $G_{r_s}$ rooted at $r_s$ remotely over XRPC, by inserting a $v_x$ node above it. This may only be done if we can ensure that the result of the rewritten query is equivalent to the original query. Such an insertion means that a new function will be defined that contains $G_{r_s}$ as its body. In the main query graph, $G_{r_s}$ is replaced by a remote XRPC call to this function, which receives as parameters all variable references in $G_{r_s}$ that resolve to variable bindings outside $G_{r_s}$.

1. Insert a vertex $v_x$ with a parse edge $(v_x, r_s)$, and replace each incoming edge $(v_{in}, r_s)$ with a new edge $(v_{in}, v_x)$.
2. For each outgoing varref edge from vertex $v_j \in V(G_{r_s})$ to $v_j \in V(G) \setminus V(G_{r_s})$, where edge $(v_i, v_j) \in E_{v}(G)$ is a varref edge as $(v_i, v_j) = (v_i, v_{qname})$, we insert a new vertex $v_k$, a new parse edge $(v_k, v_x)$, and replace the varref edge $(v_i, v_j)$ by $(v_i, v_k)$ and $(v_k, v_j)$. Here, $v_k$ has the form $v_k \in \text{XRPCParam}[^\{'p' := 'qname'\}]$, which introduces a new variable $p$ and binds it to $qname$ in $v_j$.
3. If there are no outgoing edges as stated in step 2, we insert a vertex $v_l$ with the form $v_l \in \text{XRPCParam}[^\langle\rangle]$ (i.e., empty parameter), and a parse edge $(v_x, v_l)$.

**Example 5.3.2.** Consider the d-graph in Figure 5.2. Suppose that the subgraph rooted at $v_{22}$ is identified for an XRPCExpr insertion (Figure 5.3). First, insert vertex $v_{40}$ and replace edge $(v_2, v_{22})$ by $(v_2, v_{40})$ and $(v_{40}, v_{22})$. For the outgoing varref edge $(v_{36}, v_3)$, vertex $v_{41}$ is inserted below $v_{40}$ and the varref edge is replaced by two new varref edges: $(v_{36}, v_{41})$, $(v_{41}, v_3)$.

### 5.4 Conservative Decomposition

In this and the next two sections, we first describe algorithms to decompose read-only XCore queries. We will delay the discussion of decomposing queries containing any UpdExpr and TransformExpr expressions until Section 5.7.

#### 5.4.1 By-Value Insertion Conditions

Given a $d$-graph $G$ and a subgraph $G_{r_s}$ of $G$ rooted at $r_s$, under the pass-by-value semantics, vertex $r_s$ is in the set $I(G)$ of valid decomposition points ($d$-points), iff $r_s$ satisfies all of the following conditions:

---

This is similar to the “lambda lifting” technique in the programming language domain [104].

Determined by the algorithms (Section 5.4-5.6) that compute the insertion points (i.e., determine if a vertex may be an $r_s$), $r_s$ is the only vertex in the subgraph $G_{r_s}$ that has incoming edges from vertex outside $G_{r_s}$.
5.4. CONSERVATIVE DECOMPOSITION

i. $\exists n \in V(G) : n \cdot \text{rule} \in \{\text{RevAxis, HorAxis}\} \land (\text{useResult}(n, r_s) \lor \text{useParam}(r_s, n))$

ii. $\exists n \in V(G) : n \cdot \text{rule} \in \{\text{NodeCmp}, \text{NodeSetExpr}\} \land ((r_s \cdot \text{rule}) \in \{\text{NodeCmp}, \text{NodeSetExpr}\} \land 
\quad n \notin V(G_{r_s}) \lor
\quad \text{useResult}(n, r_s) \lor \text{useParam}(r_s, n))$

iii. $\exists n \in V(G), \exists m \in V(G) : n \cdot \text{rule} = \text{AxisStep} \land m \cdot \text{rule} \in \{\text{ForExpr, OrderExpr, ExprSeq, NodeSetExpr, AxisStep}\} \land 
\quad (\text{useResult}(n, m) \land m \scriptstyle{\Rightarrow_r} \scriptstyle{\Leftarrow_m} r_s) \land
\quad (m \notin V(G_{r_s}) \land r_s \scriptstyle{\Rightarrow_r} \scriptstyle{\Leftarrow_m} m) ;$

iv. $\exists n \in V(G) : n \cdot \text{rule} = \text{FunCall} \land n \cdot \text{val} \in \{\text{fn:root()}, \text{fn:id()}, \text{fn:idref()}, \text{fn:lang()}\} \land 
\quad (\text{useResult}(n, r_s) \lor \text{useParam}(r_s, n) \land
\quad n \in V(G_{r_s}), \exists v \in V(G) \setminus V(G_{r_s}) : n \scriptstyle{\Rightarrow_r} v)$

where we impose these restrictions symmetrically both on expressions that use the result of the remote expression $r_s$, as well as on the way remote expressions (below $r_s$) use their shipped parameters:

\[
\text{useResult}(n, r_s) \leftrightarrow n \scriptstyle{\Rightarrow_r} r_s \\
\text{useParam}(r_s, n) \leftrightarrow n \in V(G_{r_s}), \exists v \in V(G) \setminus V(G_{r_s}) : n \scriptstyle{\Rightarrow_r} v
\]

Conditions i and ii guard against using any node comparisons as well as horizontal and reverse XPath steps on shipped nodes, avoiding Problems 1-3 described in Section 5.2. Condition iii also disallows decomposing any node comparisons, when a query contains multiple such expressions, to avoid the problem with document order of nodes from different documents. Condition iii avoids using downwards XPath steps (per condition i) on shipped nodes stemming from expressions that might be so-called “mixed-call sequences” (ForExpr, ExprSeq, NodeSetExpr), avoiding Problem 4. It also guards against sequences not in node order (ForExpr, OrderExpr) or with nodes that may be overlapping (the restrictions on NodeSetExpr and XPath steps). This ensures that downwards XPath steps can be used on shipped node sequences that are ordered and non-overlapping. Condition iv states that shipped nodes may not be used as parameters of the listed built-in functions (Problem 5).

Example 5.4.1. In the $d$-graph of example query $Q_2^5$ (Figure 5.2), we mark in shades of grey the $d$-points identified by the conservative decomposition strategy. The XPath step /grade that is performed on the result of a for-loop, matches condition iii and causes all vertices that depend on $v_{10}$ and $v_{22}$ (the ForExprs) as well as all their descendants to be excluded from $I(G)$, leaving $v_1$ and the subgraphs rooted at $v_5$ as $d$-points.

5.4.2 Interesting Decomposition Points

While a $d$-point may be semantically valid, remote evaluation of the subquery below it might not be useful from a performance perspective. Consider the $d$-point $v_8$, which contains only an fn:doc() function call in its subgraph. Executing this function remotely provides no performance gain, as it only demands the shipping of a whole document. Similarly, remote execution of expressions that do not involve any XML documents should be avoided. Therefore, we filter $d$-points by first annotating each vertex $v_x \in V(G)$ with the URI dependency set $D(v_x)$. Here, $D(v_x)$ represents the set of URLs that are used as parameters of fn:doc() in vertices that the vertex $v_x$ can reach via parse edges:

\[
D(v_x) = \{\text{uri} : v_y \mid (v_y, v_x) \in E(G) : v_y \scriptstyle{\Rightarrow_r} v_x \land v_x \cdot \text{rule} = \text{FunApp} \land v_x \cdot \text{val} = \text{"doc"} \land
\quad ((v_x \cdot \text{rule} = \text{Literal} \land v_x \cdot \text{uri} = \text{"doc"}) \lor (v_x \cdot \text{rule} \neq \text{Literal} \land v_x \cdot \text{uri} = \text{"*"}))\}
\]

We tag each $\text{uri}$ with the vertex $v_y$ where the document is opened, to be able to distinguish the use of the same document through multiple fn:doc() calls. If the parameter of fn:doc() is
Table 5.4: Query decomposition and code motion

an expression instead of a literal, we use a wildcard symbol “*” as uri. In this chapter, the built-in function fn:collection() is treated as an fn:doc(*), and an element construction is assigned an artificial unique URI fn:doc(vi::vi).

One can use the URI dependency set to partition the V(G) into equivalence classes, i.e., those vertices with the same URI dependency set belong to the same class. Using all vertices in an equivalence class, we can consider its induced subgraph in G, and try to handle it in a single XRPC subquery. Thus, we define interesting decomposition points (i-points) I'(G) as those valid insertion points that (a) are a root vertex in their induced subgraph, (b) contain at least one fn:doc() and (c) execute at least one XPath step on the fn:doc() function:

\[ I'(G) = \{ v_x | v_x \in I(G) : \exists v_y : v_y \xrightarrow{L} v_x \wedge D(v_y) = D(v_x) \wedge \exists v_z : v_z \xrightarrow{L} v_x \wedge v_z.rule = \text{AxisStep} \wedge \exists \text{xrpc://uri} \in D(v_x) \} \]

Note that this definition is also used by the next two algorithms to filter the d-points.

Example 5.4.2. In Figure 5.2, the two subtrees rooted at v5 and v25 correspond to two different equivalence classes D(v5) = {xrpc://A/students.xml::v9} and D(v25) = {xrpc://B/course42.xml::v27}. However, v25 is not a valid insertion point. The vertices in I'(G) (coloured dark grey) are v6 (the highest non Var vertex in the subtree rooted at v5) and the root v1. Thus, I'(G) = \{v1,v6\}.

5If the root node happens to be a Var vertex, we consider its value expression instead as root.
5.4.3 Normalisation

Rewriting algorithms that operate on the XCore level are vulnerable to syntactic variation. In the case of our decomposition strategy, an important vulnerability comes from the behaviour of the strategy to ship subgraphs consisting of parse-edges only. That is, varref-edges are not pushed, but rather become parameters to the function. The syntactic freedom one has in XQuery of defining subexpressions, e.g., inline or via a variable reference to a previous let-binding, therefore affects our strategy. For this purpose, as part of XCore normalisation, we re-order let-bindings, moving them as deep into the query as possible. More specifically, let-bindings are moved to just above the lowest common ancestor vertex (defined in terms of parse-edges) of all vertices that reference its variable. The query $Q^c_2$ (Table 5.3) can be normalised to $Q^n_2$ (Table 5.3), which can thus be rewritten as $Q^v_2$ in Table 5.4.

The main achievement of normalisation in the above case is to relate the call to \texttt{doc(".../course42.xml")} through parse-edges (directly calling $c$ in $Q^n_2$), instead of varref edges (referencing $c$ in $Q^c_2$), with its use in the /child::enroll/child::exam XPath steps. However, these being part of a ForExpr with the /grade step on top, causes insertion condition ii to prohibit pushing it. In the next section on pass-by-fragment, however, we will see that normalisation was not in vain, and the query can be decomposed into $Q^f_2$ (Table 5.4).

5.4.4 Distributed Code Motion

The let-normalisation phase has the effect of pushing expressions that depend on the same documents downwards, potentially below an interesting insertion point (which causes them to be executed remotely). However, it can happen that some of the expressions initially found below an interesting insertion point can in fact better be moved above it (to be executed locally). In particular, it is safe to assume that expressions that solely depend on a parameter of a function, can better be evaluated on the caller side. Moving a subexpression out of a function can be done by passing that subexpression as an additional parameter to the function. With pass-by-value passing, such a rewrite may not always be safe, however if only $d$-points are moved, the technique is semantically safe. Analogous to the well-known compiler technique of moving invariant statements out of the loop (and its use in parallel processing [110]) we call this technique distributed code motion.

Example 5.4.3. Consider the function fcn2() in Table 5.4, we may observe that the expression $\$\texttt{para1/child::id}$ only depends on the function parameter $\$\texttt{para1}$. Shipping full person nodes $\$\texttt{para1}$ from peer A to B, only to extract the string value of its id child at B, may waste bandwidth, especially if person carries much more data than just an id. Instead, it would be better to extract the string value of id at peer A and only ship the strings. This optimisation can be realised by adding a new parameter $\$\texttt{para2}$ to the function, and substituting $\$\texttt{para1/child::id}$ in the body with it. In the function fcn0() that calls fcn2new(), we save the original function parameter $\$\texttt{t}$ in a new let-binding $\$\texttt{l}$, and pass $\$\texttt{l}$ instead of $\$\texttt{t}$. The additional function parameter is passed as $\$\texttt{l/child::id}$. Finally, the affected function parameter $\$\texttt{para1}$ is no longer used, so we remove it, arriving at the result as the code motion part in Table 5.4.
5.5  By-Fragment Decomposition

The node copying done by pass-by-value is the main source of semantic differences. This, in turn, leads to serious restrictions in the way the decomposition strategy can push expressions remotely. For this reason, we extend the pass-by-value message passing semantics into a new pass-by-fragment message passing semantics that better preserves structural relationships of XML nodes.

The basic idea is to avoid serialising the same nodes twice, by grouping all node-valued data in the message in a preamble element fragments. In principle, each node parameter is serialised below a separate fragment child element. However, if a sent node is a descendant of another one, it is not serialised twice, as we can reuse the XML fragment of the other node. We also ensure that the XML fragments are sorted in original document order, which means that ancestor/descendant relationships in the same message, as well as node identity and document order, are preserved.

Later in the message, where XQuery sequences are serialised (inside sequence tags), we just provide references to the nodes that were previously serialised in the fragments. In particular, an element tag, which is used to contain as a child the fully serialised copy of a node, now just carries two numeric attributes, fragid (pre-order of the fragment containing this element within the fragments section) and nodeid (pre-order of this element within the fragment referred to by fragid). In order to keep XRPC an interoperable protocol that is easy to implement for XQuery engines and the XRPC Wrapper [180], node referencing is also expressible in XQuery. Supposing $msg$ is the root of the message, with $fragid$ and $nodeid$ numbers, we can identify the referenced nodes as follows:

\[
\text{msg}/\text{fragment}[\text{fragid}]/\text{descendant::node}()[\text{nodeid}]
\]

**Example 5.5.1.** Going back to $Q_1$ in Table 5.1, the lower part of Table 5.5 shows the XRPC request message sent for the call execute at {"example.org"} {earlier ($bc$, $abc$)} from the discussion of Problem 3. Recall that the node $bc$ with value ⟨b⟩⟨c/⟩⟨/b⟩ is contained in the $abc$ fragment ⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩. The lower part of the figure shows an excerpt from the message as produced for pass-by-fragment. Here, both node parameters $bc$ and $abc$ are represented in element nodes with fragid and nodeid attributes. The XQuery engine handling the call will use these attributes to evaluate:

\[
\begin{align*}
\text{bc} & := \text{msg}:\text{fragment}[1]/\text{descendant::node}()[2], \\
\text{abc} & := \text{msg}:\text{fragment}[1]/\text{descendant::node}()[1]
\end{align*}
\]

such that earlier ($bc$, $abc$) correctly returns $abc$, because $abc \ll $bc, just like on the peer that invoked this function. The upper part, with the changed part of the old pass-by-value message (element call), shows that node parameters were previously repeatedly serialised, causing node order and identity relationships between parameters to be lost.

We made a conscious choice not to rely on ID/IDREF for referencing nodes, since this would require adding ID attributes to the XML data in the fragments. As XRPC is designed to respect and conserve XML SCHEMA type information, this would cause the XRPC message to no longer respect user-defined schemas.

---

Note that descendant::node() does not return attribute nodes. We use the nodeid of its parent and include the name of the attribute in an attribute element, so it can be found back with an additional attribute step.
By-Fragment Insertion Conditions  Given a d-graph $G$ and a subgraph $G_r$ of $G$ rooted at $r_s$, under the pass-by-fragment semantics, vertex $r_i$ is in the set $I(G)$ of valid decomposition points, iff $r_s$ satisfies all of the following conditions:

I. $\not\exists n \in V(G) : n.\text{rule} \in \{\text{RevAxis, HorAxis}\} \land (\text{useResult}(n, r_s) \lor \text{useParam}(r_s, n))$

II. $\not\exists n \in V(G) : n.\text{rule} \in \{\text{NodeCmp, NodeSetExpr}\} \land
\quad \left((r_s.\text{rule} \in \{\text{NodeCmp, NodeSetExpr}\} \land n \not\in V(G_r) \land \text{hasMatchingDoc}(n, r_s)) \lor
\quad (\text{useResult}(n, r_s) \lor \text{useParam}(r_s, n)) \land \text{hasMatchingDoc}(n, n_s)\right)$

III. $\exists m \in V(G) : n.\text{rule} = \text{AxisStep} \land n.\text{rule} \in \{\text{ForExpr, ExprSeq, NodeSetExpr}\} \land
\quad \left((\text{useResult}(n, m) \land m \sim r_s) \lor (\text{useResult}(n, r_s) \land m \in V(G_r)) \lor
\quad (m \in V(G) \setminus V(G_r) \land r_s \sim n \sim m)) \land \text{hasMatchingDoc}(m, m)\right)$

IV. $\not\exists n \in V(G) : n.\text{rule} = \text{FunCall} \land n.\text{val} \in \{\text{fn:root()}, \text{fn:id()}, \text{fn:idref()}, \text{fn:lang()}) \land
\quad (\text{useResult}(n, r_s) \lor \text{useParam}(r_s, n))\right)$

Thus, with the pass-by-fragment semantics, we modify the pass-by-value decomposition conditions listed in Section 5.4 by restricting the prohibitions to decompose a node $r_s$ formulated in Conditions ii and iii to only those $r_s$, for which the predicate hasMatchingDoc() holds. Here, hasMatchingDoc() is defined as:

\[\text{hasMatchingDoc}(v_1, v_2) \iff \forall u_i : v_i \in D(v_1), \exists u_i : v_j \in D(v_2) :\]
\[v_i \neq v_j \land (u_i = u_i \lor u_i = \star \lor u_i = \star)\]

By stating that the given expressions depend on two different applications of fn:doc() with the same URI (taking into account computed URIs as wildcards), this predicate precisely isolates the problem of creating result sequences with remote nodes from multiple calls to the same document.

The ForExpr is a special form of combining the results of multiple calls. A remote call nested in a for-loop which depends on the same remote document, is treated as a single call, since Bulk RPC ensures that all iterations of the remote call nested in the for-loop are handled in a single message exchange (where pass-by-fragment now ensures proper conservation of node relationships). Finally, we remove from condition iii the restrictions that arbitrary ordering (OrderExpr) cannot be used and that all pushed AxisSteps should be of the non-overlapping kind (parent, preceding-sibling, following-sibling, self, child, and

---

Table 5.5: By-value vs. by-fragment messages. In the by-fragment message, the first element node refers to the second descendant node (i.e., nodeid="2") of the first fragment (i.e., fragid="1") in the fragments section earlier in the message.
attributem), as the pass-by-fragment message passing is able to properly conserve sequence order and the ancestor/descendant relationships between transported nodes. As the remaining problems with mixed-call sequences are related to dealing with multiple network message exchanges in the same query, this problem can not be solved inside the message passing semantics alone and is beyond our current scope. The restrictions to avoid horizontal and reverse XPath steps on remote nodes (Condition I) and on using built-in functions (Condition iv) will be addressed in the next section.

**Example 5.5.2.** Consider Figure 5.2, as the constraint hasMatchingDoc() in condition III does not hold, all vertices in the graph are identified as valid decomposition points under the pass-by-fragment semantics. However, most vertices will be filtered out by the definition of interesting decomposition points, which leads to $I^*(G) = \{v_1, v_2, v_4, v_6, v_22, v_24\}$.

### 5.6 By-Projection Decomposition

The basic idea of using XML projection [125] is, for a given XQuery query $Q$ and an XML document $D$, to extract a minimal subdocument $D'$ needed to execute $Q$ such that $Q(D) = Q(D')$. The projection technique conducts a compile-time path analysis on $Q$, to derive a set of simple path expressions that over-estimate the nodes that $Q$ touches. These simple paths are referred to as projection paths. Here, a projection path is an XML path that starts from the document root, containing forward navigation but not predicates (e.g., `doc("uri")/a/b/@id`).

Projection paths consist of returned paths and used paths. Returned paths describe the nodes that are returned by the expression. Used paths indicate the nodes necessary to answer the query but are never returned as results (e.g., predicates).

Based on the projected paths $P$ of query $Q$ from path analysis, a loading algorithm is applied to $P$ and an XML document (from a file or a stream) $D$. A projected XML document (or stream) $D'$ is then generated, which contains all used and returned nodes plus the descendants of the returned nodes, and is queried with $Q$.

There are three reasons why projecting XML is extremely interesting for distributed XML processing: (i) until now, when sending nodes, we had to serialise all descendants – which potentially contain huge subtrees that may remain untouched on the other side. This amounts to wasted network bandwidth as well as serialisation and shredding effort. (ii) if documents are projected into lean skeletons that only contain the relevant portions, it becomes feasible to serialise XML fragments from some lowest common ancestor on, possibly even the document root. Even with pass-by-fragment, the execution of reverse/horizontal XPath axes on remote nodes is impossible. By extending projecting XML with support for reverse and horizontal axes, however, we get a tool to precisely identify the lowest common ancestor of an XML document that needs to be included to allow correct remote execution of those axes. (iii) the projection technique can even be applied to support the built-in functions `fn:root(`, 

| ProjectionPath | ::= doc("Literal":"Literal") (/"SimplePath")* |
| SimplePath     | ::= AxisStep "=" NodeTest | SimplePath="/" AxisStep "=" NodeTest |
| AxisStep       | ::= "self" | "child" | "attribute" | "descendant" | "descendant-or-self" |
| NodeTest       | ::= "node()" | "text()" | QName | "*" |

Table 5.6: Grammar rule extension of ProjectionPath (bold)
fn:id(), fn:idref() and fn:lang(), i.e., by taking the lowest common ancestor of those, if a path contains one of these functions.

For these reasons, we further refine the pass-by-fragment message passing semantics into a so-called pass-by-projection semantics. XML projection can be used in both directions: to project the parameters in a request message, and to project the function’s result sequence before shipping back the response.

**Insertion Conditions** Pass-by-projection removes the by-fragment insertion conditions (in Section 5.5) I and IV, such that only II and III, i.e., the application of node comparison, node set operators and axis steps on top of multiple calls to fn:doc() with the same URI remains illegal. Hence, given a d-graph $G$ and a subgraph $G_{r_s}$ of $G$ rooted at $r_s$, under the pass-by-projection semantics, vertex $r_s$ is in the set $I(G)$ of valid decomposition points, if $r_s$ satisfies all of the following conditions:

$$
\begin{align*}
(a) \quad & \not\exists n \in V(G) : n.\text{rule} \in \{\text{NodeCmp, NodeSetExpr}\} \land \\
& \quad ((r_s.\text{rule} \in \{\text{NodeCmp, NodeSetExpr}\} \land n \notin V(G_{r_s}) \land \text{hasMatchingDoc}(n,r_s)) \lor \\
& \quad ((\text{useResult}(n,r_s) \lor \text{useParam}(r_s,n)) \land \text{hasMatchingDoc}(n,n)); \\
(b) \quad & \not\exists n \in V(G), 3m \in V(G) : n.\text{rule} = \text{AxisStep} \land m.\text{rule} \in \{\text{ForExpr, ExprSeq, NodeSetExpr}\} \land \\
& \quad ((\text{useResult}(n,m) \land m \sim r_s) \lor (\text{useResult}(n,r_s) \land m \in V(G_{r_s}) \land \\
& \quad (m \in V(G) \land V(G_{r_s}) \land r_s \not\sim m \land r_s \not\sim n \not\sim m)) \land \text{hasMatchingDoc}(m,m)).
\end{align*}
$$

**Message Extension: Projection Paths** We introduce an optional element as a sub-element of a request element: projection-paths, which in turn has zero or more child elements returned-path and used-path. In the new pass-by-projection semantics, the absence or presence of this element determines whether the message should be in the original pass-by-value or the new pass-by-projection format.

**Example 5.6.1.** To illustrate projected XRPC messages, the upper part of Table 5.7 shows part of the request message for the call from $Q_1$ (discussed in Problem 4):

```xml
let $bc := execute at "example.org" {makenodes()}
```

since the projection path analysis detects that $bc$ will subsequently be used as context node by a parent step: $abc := bc/parent::a$, the request message specifies parent::a as a returned path. Therefore, the response message contains the full fragment $\langle a \rangle \langle b \rangle \langle c / \rangle \langle / b \rangle \langle / a \rangle$ to which $abc$ then gets correctly bound.
5.6.1 Extending Projected XML

We extend the path grammar rules [125] and path annotations, to handle full-fledged XQuery involving reverse/horizontal XPath steps and built-in functions. The extended grammar rule for ProjectionPath is given in Table 5.6.

We denote path annotations in projected XML as follows:

$$\text{Env}(\nu_i) \vdash \text{Expr} \Rightarrow \text{Paths using } \text{UPaths}$$

The notation $\text{Env}(\nu_i)$ is used to identify the path annotation environment at a certain vertex $\nu_i$ in the XQuery $\mathcal{d}$-graph.

Path annotations are constructed bottom up by path analysis rules that derive the set of used ($\text{UPaths}$) and returned ($\text{Paths}$) paths for each XCore expression in terms of used and returned paths of its subexpressions. Therefore, we extend the notation of the vertices and use $\nu_i.\text{UPaths}$ and $\nu_i.\text{Paths}$ to refer to the path sets, with which the vertex $\nu_i$ is annotated.

Example 5.6.2. Assume that the subgraph $G_{v_{22}}$ rooted at $v_{22}$ in Figure 5.2 is identified to be evaluated remotely. The subgraph $G_{v_{22}}$ has one parameter, $\text{St}$, via the VarRef edge ($v_{36}, v_3$). We show the path annotations of $v_3$ and $v_{22}$ in Figure 5.4. Comparing the returned path of $v_3$ with all projection paths of $v_{22}$ and $v_1$, we know that $v_3$ is only used in the subgraph rooted at $v_{22}$ (i.e., it is not returned by $v_{22}$), and that only the id child elements of the person elements
are used. Thus, only those elements will be projected and serialised in the request message for \( v_{22} \).

The basic path analysis rules have been discussed in [125], such as literal values, sequences, for and let expressions and XPath steps, etc. Our extension to include reverse and horizontal XPath steps brings no changes for the path analysis rules, but must be supported by the loading algorithm, which is described in Section 5.6.2. We complement the rules for built-in functions, which apart from the unsolved cases mentioned under Problem 5 in Section 5.2 (fn:root(), fn:id(), fn:idref() and fn:lang()) also includes fn:doc(). The description of the basic projection technique assumes a single document. As in distributed query processing there are always multiple documents, our paths always start with \( \text{fn:doc}(\text{URI}) \).

**Path Analysis Rules**

We provide one rule for \( \text{fn:doc}() \) with a constant parameter and another for computed URIs:

\[
\begin{align*}
\text{Env}(v_i) \vdash \text{doc}(\text{Literal}1) & \Rightarrow \text{doc}(\text{Literal}1::v_i) \text{ using } /\emptyset \\
\text{Env}(v_j) \vdash \text{Expr} \Rightarrow \text{Paths}_j \text{ using } U\text{Paths}_j \\
\text{Env}(v_i) \vdash \text{doc}(\text{Expr}) & \Rightarrow \text{doc}(*::v_i) \text{ using } \text{Paths}_j \cup U\text{Paths}_j \cup \text{Paths}_k \cup \text{descendant::text()} \\
\end{align*}
\]

As mentioned in Section 5.4, in the definition of \( D(v_i)^7 \), we use a wildcard URI \( * \) if the document name is an expression. Note that all paths start with \( \text{doc}(\text{uri}::v_i) \), thus, they identify both document URI as well as the vertex \( v_i \) where it is loaded. This notation facilitates the identification of situations where the same URI is loaded twice (the function hasMatchingDoc()). A similar rule can be formulated for XML element construction, producing a return path \( \text{doc}(v_i::v_i) \) with an artificial unique URI. Also note that because XQuery always automatically applies atomisation to node typed function parameters, we add a descendant::text() step to each returned path of a parameter in all rules in this section. The rule for \( \text{fn:root}() \) is:

\[
\begin{align*}
\text{Env}(v_i) \vdash \text{Expr} & \Rightarrow \text{Paths}_j \text{ using } U\text{Paths}_j \\
\text{Env}(v_i) \vdash \text{fn:root}(\text{Expr}) & \Rightarrow \bigcup_{p \in \text{Paths}_k} \text{p/root()} \text{ using } U\text{Paths}_j \\
\end{align*}
\]

The built-in function \( \text{fn:root}() \) with a single parameter is treated in the path annotations much like XPath axis steps, where the parameter has become the path prefix. In this path notation, functions remain easily recognisable by the parentheses. The rules for the built-in functions \( \text{fn:id()}/\text{fn:idref()} \), are highly similar (only \( \text{fn:id}() \) provided):

\[
\begin{align*}
\text{Env}(v_i) \vdash \text{Expr} & \Rightarrow \text{Paths}_j \text{ using } U\text{Paths}_j \\
\text{Env}(v_i) \vdash \text{Expr}_k & \Rightarrow \text{Paths}_k \text{ using } U\text{Paths}_k \\
\text{Env}(v_i) \vdash \text{fn:id}(\text{Expr}_j,\text{Expr}_k) & \Rightarrow \bigcup_{p \in \text{Paths}_j} \text{p/id()} \text{ using } \bigcup_{p \in \text{Paths}_k} \text{paths} \cup U\text{Paths}_j \cup U\text{Paths}_k \cup Paths_j/\text{descendant::text()} \\
\end{align*}
\]

The first parameter of \( \text{fn:id}() \) is ignored by the annotations as it contains string values, and the annotation framework only allows for the estimation of node sets. This has the consequence that our loading algorithm will conserve all elements with an ID/IDREF attribute. Finally, the rule for \( \text{fn:lang}() \) is:

\[
\begin{align*}
\text{Env}(v_i) \vdash \text{Expr} & \Rightarrow \text{Paths}_j \text{ using } U\text{Paths}_j \\
\text{Env}(v_i) \vdash \text{Expr}_k & \Rightarrow \text{Paths}_k \text{ using } U\text{Paths}_k \\
\text{Env}(v_i) \vdash \text{fn:lang}(\text{Expr}_j,\text{Expr}_k) & \Rightarrow \bigcup_{p \in \text{Paths}_j} \text{paths} \cup U\text{Paths}_j \cup U\text{Paths}_k \cup Paths_j/\text{descendant::text()} \\
\end{align*}
\]

We use the \( \text{doc}(...) \) prefixes of the returned paths annotations on \( v \) as a more precise form of the \( D(v) \) property. Documents that were only used but not returned will also be part of the original \( D(v) \), but these will not cause semantic problems.
The built-in function \( \text{fn:lang}() \) tests whether the language of its first parameter \( \text{Expr}_k \), as specified by \text{xml:lang} attributes, is the same as (or is a sublanguage of) the language specified by its second parameter \( \text{Expr}_j \). The language of \( \text{Expr}_k \) is determined by the value of the XPath expression: \( \text{(ancestor-or-self::*//attribute::xml:lang)}[\text{last()}] \). All paths are propagated as used paths, as this function returns a boolean value.

5.6.2 Runtime XML Projection

The extensions we made to XML projection, namely support for reverse/horizontal XPath axes and \( \text{fn:root}(), \text{fn:id}(), \text{fn:idref}() \) and \( \text{fn:lang}() \), could not be trivially integrated in the loading algorithm of [125]. However, in case of XRPC we are not really looking for a loading algorithm that efficiently reads (shreds) an XML file into a projected representation. Rather, the documents are already present (and indexed) in the XQuery engine, and runtime message projection is a serialisation task. Therefore, we propose a new runtime approach for projection, targeted at serialisation, rather than at shredding. Whereas the original loading algorithm starts at the document root, and evaluates absolute used and returned paths, our runtime projection algorithm starts in a run-time state, that is, with a real, materialised context sequence (e.g., the parameter values that are about to be serialised in a SOAP message), and executes only relative paths on them. Because the node sequence bound at run-time to a function parameter is only a subset of the node set characterised by its compile-time path annotation (e.g., its contents may well have been reduced by applying a selection predicate), this runtime projection technique can be much more precise than the original projection algorithm. As a final consideration, the projected XRPC messages trade projection effort for network bandwidth, which especially in WAN scenarios plays in the advantage of projection.

For these reasons, our runtime approach for projection simply relies on the normal XPath evaluation capabilities of the XQuery engine for fully evaluating all used and returned path annotations one-by-one (and uniting them with \( \text{union()} \)). Doing so, it produces a used node set \( U \) and a returned node set \( R \). These two sets are the input for the runtime projection algorithm listed in Algorithm 1.

The Runtime Projection Algorithm The runtime projection algorithm identifies all projection nodes in the XML tree representation of the original document, by traversing the tree top-down depth-first. During traversal, if the current node \( \text{cur} \) of the XML document is an ancestor of the current projection node \( \text{proj} \) (line 5), \( \text{cur} \) is added to output \( D' \) and moved to the next node in document order. If a \( \text{proj} \) is found (line 8), \( \text{proj} \) is added to \( D' \); if this \( \text{proj} \) is a returned node, all its descendants are also appended. Then \( \text{cur} \) is moved to its next following node in the document. Otherwise, if the current projection node \( \text{proj} \) is not a descendant of \( \text{cur} \), the subtree of \( \text{cur} \) can be skipped (line 21). Though this algorithm is formulated on an abstract level that is independent of the particular XML storage scheme used in an XQuery engine, it is safe to assume that skipping a subtree is fast (either \( O(1) \) or \( O(\log(|D|)) \)). At the end of the algorithm (lines 24-27), post-processing is performed to remove unnecessary nodes, as we are only interested in the lowest common ancestor of all input nodes in the projected document \( D' \).
Algorithm 1: \texttt{RuntimeXMLProjection}(U, R, D)

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{input} : \(U\)- used nodes
\State \(R\)- returned nodes
\State \(D\)- the original XML document
\State \textbf{output} : \(D'\)- the projection of \(U\) and \(R\) on \(D\)
\State \(P \leftarrow \text{sort}(U \cup R)\) \Comment{\(P\) is union of \(U\) and \(R\) sorted by document order}
\State \(\text{proj} \leftarrow \text{first node in } P\)
\State \(\text{cur} \leftarrow \text{first node of } D\), i.e., root node;
\While{\(\text{proj} \neq \text{end}\)}
\If{\text{proj} is a descendant of \text{cur}}
\State add \text{cur} to \(D'\);
\If{\text{proj} is a returned node}
\State add \text{cur} and all descendants of \text{cur} to \(D'\);
\State \text{cur} \leftarrow \text{next following node of } \text{cur} \text{ in } D;
\While{\text{proj}.\text{next} \text{ is a descendant of } \text{proj}}
\State \text{proj} \leftarrow \text{proj}.\text{next} \Comment{prune projection nodes;}
\EndWhile
\EndIf
\State \text{proj} \leftarrow \text{proj}.\text{next} \Comment{next projection node;}
\Else
\State add \text{cur} to \(D'\);
\State \text{cur} \leftarrow \text{next node in } D;
\EndIf
\EndWhile
\State \text{cur} \leftarrow \text{root node of } D';
\While{\text{cur} has only one child node \& \text{cur} \notin \{U \cup R\}}
\State \text{cur} \leftarrow \text{first child of } \text{cur};
\EndWhile
\end{algorithmic}
\end{algorithm}

\textbf{Example 5.6.3.} Consider an XML document \(D\) in Figure 5.5(a). Assume that the used node set \(U\) is \(\{i\}\), and the returned node set \(R\) is \(\{d, k\}\). Figure 5.5(b) shows the projected document \(D'\) of applying Algorithm 1 on \(U, R\) and \(D\).

The algorithm starts with \(P \leftarrow \{d, i, k\}\), \(\text{proj} \leftarrow d\) and \(\text{cur} \leftarrow a\). We traverse the tree using \(\text{cur}\) from \(a\) to \(d\). Nodes \(a, b\) and \(c\) are added to \(D'\), since they are ancestors of the current context node \(d\). Nodes \(d, e\) and \(f\) are also added to \(D'\), as \(d\) is a returned node. Then, \(\text{cur}\) is advanced to \(g\) (\(d\)'s next following node). Because the next context node \(i\) is not in the subtree of \(g\), the subtree is skipped by advancing \(\text{cur}\) to \(i\). Recall that \(i\) is a used node, thus only \(i\) is added to \(D'\). The last context node is \(k\). Our current document node \(\text{cur}\) traverses from \(i\) to \(j\), and then to \(k\), where we can add nodes \(k, l\) and \(m\) to \(D'\). The traversal can be terminated, because there are no more context nodes to process. However, the intermediate result \(D'\) contains all common ancestors of \(\{d, i, k\}\). The post-processing removes node \(a\) from \(D'\), which produces the final projected document \(D'\) as shown in Figure 5.5(b).

Relative Projection Paths At \textit{compile time}, the XQuery compiler builds a query graph \((d\text{-graph})\) with root \(v_{\text{root}}\), normalises it, and then does decomposition and code motion. For each inserted \texttt{XRPCExr} \(v_{\text{xrpc}}\), and for each \texttt{XRPCParam} parameter vertex \(v_{\text{param}}\), it then extracts the relative paths:
CHAPTER 5. XQUERY DECOMPOSITION

Figure 5.5: Runtime XML projection example

\[ U_{rel}(v_{xrpc}) = \text{allSuffixes}(R(v_{xrpc}), U(v_{\text{root}})) \]
\[ R_{rel}(v_{xrpc}) = \text{allSuffixes}(R(v_{xrpc}), R(v_{\text{root}})) \]
\[ U_{rel}(v_{\text{param}}) = \text{allSuffixesVia}(R(v_{\text{param}}), U(v_{xrpc}), U(v_{\text{root}})) \]
\[ R_{rel}(v_{\text{param}}) = \text{allSuffixesVia}(R(v_{\text{param}}), R(v_{xrpc}), R(v_{\text{root}})) \]

where:
\[ \text{allSuffixes}(\text{Paths}_x, \text{Paths}_y) = \{ s_x | p_x, s_x \in \text{Paths}_y : \exists p_x \in \text{Paths}_x \} \]
\[ \text{allSuffixesVia}(\text{Paths}_x, \text{Paths}_y, \text{Paths}_z) = \{ s_x, s_z | p_x, s_x \in \text{Paths}_y : \exists p_x, s_z \in \text{Paths}_z \land \exists p_z \in \text{Paths}_x \} \cup \{ s_y | p_y, s_y \in \text{Paths}_z \land p_y, s_y \in \text{Paths}_x \} \]

At runtime, \( U_{\text{param}} \cap U_{\text{rel}}(v_{\text{param}}) \) and \( R_{\text{param}} \cap R_{\text{rel}}(v_{\text{param}}) \) are used to project the parameters in the outgoing XRPC request message. \( U_{\text{rel}}(v_{\text{xrpc}}) \) and \( R_{\text{rel}}(v_{\text{xrpc}}) \) are passed in the projection-paths element such that a remote peer can appropriately apply these paths to project the response message. When computing the relative used and returned paths for \( v_{\text{param}} \), we need to take into account that (parts of) \( v_{\text{param}} \) could be returned by \( v_{\text{xrpc}} \), and thus will be used by vertices depending on \( v_{\text{xrpc}} \). Hence, in \( \text{allSuffixesVia}() \), we not only find the relative paths that \( v_{\text{xrpc}} \) will apply on \( v_{\text{param}} \), but also the relative paths that \( v_{\text{root}} \) will apply on \( v_{\text{param}} \). If both the relative used and returned paths for a vertex are empty sets, this vertex is not projected. To serialise such vertices, by-fragment semantics is used.

Projecting a document using Algorithm 1 requires pre-calculated used and returned node sets. These sets are simply computed using the XPath evaluation infrastructure of the underlying XQuery engine by feeding the intermediate result \( \$ctx_{\text{param}} \) corresponding to \( v_{\text{param}} \) as context sequence into all suffix paths \( s_i \in U_{\text{rel}}(v_{\text{param}}) \) (resp. \( R_{\text{rel}}(v_{\text{param}}) \)):

\[ \text{union}(\$ctx_{\text{param}}/s_1, \text{union}(\$ctx_{\text{param}}/s_2, \ldots \text{union}(\$ctx_{\text{param}}/s_n-1, \$ctx_{\text{param}}/s_n)) \ldots) \]

Paths \( \$ctx/\text{path}_i/\text{root}()/\text{path}_j \) with function \( \text{root}() \) are executed as \( \text{root}(\$ctx)/\text{path}_j \). Similarly, \( \$ctx/\text{path}_i/\text{id}()/\text{path}_j \) is executed as \( \text{root}(\$ctx)/\text{attribute}()::(\text{id}1|...|\text{id}_n)/../\text{path}_j \), where \( \text{id}1, ..., \text{id}_n \) are all ID attributes\(^8\) (resp. IDREF in case of idref()).

The request handler on the remote side uses the same method to evaluate the suffix paths \( U_{\text{rel}}(v_{\text{xrpc}}) \) and \( R_{\text{rel}}(v_{\text{xrpc}}) \) using the result sequence of the function as \( \$ctx_{\text{xrpc}} \) during serialisation of the response message.

**Interoperability** We have devised a way to support pass-by-projection in the XRPC Wrapper by substituting the projection algorithm with a variant that serialises the lowest common ancestor of the used and returned node sets. Since document projection is not expressible in

\(^8\)Note that these \( \text{id}_i \) should be determined at runtime by the XRPC projection algorithm. The impossibility to express selection of all ID/IDREF attributes in XQuery, and thus in the XRPC Wrapper, forces us to still avoid shipping expressions where the result of \( v_{\text{xrpc}} \) is used as input to \text{id}()//idref().
XQuery (not even with the TRANSFORM feature of XQUF), this is as far as a pure XQuery engine can get. We contemplate the possibility to let the XRPC Wrapper echo the SOAP response message it generates to a stream, and implement a streaming version of our projection algorithm (that first gets a stream of used and returned nodes, and then the to be projected fragments) inside the XRPC Wrapper java program.

In case of XML data with a user-defined XML SCHEMA, the default projection algorithm is likely to throw away mandatory elements and attributes. For this reason, the runtime projection algorithm should be made schema-aware. A simple solution is to ensure that only elements with a minoccurs declaration of zero (i.e., optional elements) are removed. One can also envision more advanced variants that further reduce the size of a typed XML document.

5.7 Decomposition of XQUF Queries

Since the introduction of the W3C XQUF [58] specification, which has been well-received and adopted by various XQuery engines (e.g., [64, 69, 129, 140, 141, 109, 176]), XQuery is no longer a read-only query language. We now show how we can leverage such update-capable XQuery engines to automatically rewrite purely local updates into queries that may push some computations to remote peers. We recall that the general processing model of XQUF is that first the read-only part of a query is executed that defines which nodes are going to be updated, and how. This first phase results in a pending update list (PUL). In the second phase all update actions in this list are executed. Therefore, the first phase of XQUF execution is identical to a read-only query, and can in principle be distributed in the same way as described in the previous sections. However, systems implementing the XQUF typically only allow updating persistently stored documents, e.g., updating documents on an HTTP URI is not allowed. In this section, we first explain the restrictions the XQUF imposes on XRPC query distribution. Then, we extend the semantics of XQUF to allow updates on documents opened with fn:doc() using xrpc://P/D URIs (in short: remote documents) and also to support the fn:put() XQUF built-in function to write entire new documents to such URIs. This extended semantics again creates a possible trade-off between data shipping vs. function shipping, namely retrieving and updating a local copy of a remote document followed by an fn:put() vs. executing an XQUF updating function over XRPC. We introduce the necessary constraints to our query distribution techniques that guarantee semantic equivalence for such queries.

5.7.1 Distributing Normal XQUF Queries

XQUF has extended the XQuery language with four kinds of updating expressions: UpdExpr = {InsertExpr, DeleteExpr, RenameExpr, ReplaceExpr} (Table 5.2). An XCore query containing at least one UpdExpr is an updating XCore query (in short: updating query). Each UpdExpr has a TargetExpr that identifies the target nodes to be updated, and (except for DeleteExpr) each has an ExprSingle that computes the new values. For simplicity, we refer to those ExprSingle as SourceExpr, although XQUF uses different names. The functionality of the first three kinds of expressions is self explanatory. With ReplaceExpr, one can replace the target node with a new sequence of nodes (ReplaceNode), or replace the value of the target node (ReplaceValue). The expressions RenameExpr and ReplaceValue only modify some properties of the target node without changing its node identity.
XQUF also defines a *transform expression* (TransformExpr) that creates (and possibly modifies) copies of existing XML nodes. Each node created by a TransformExpr has a new node identity. The result of a TransformExpr is an XDM (XQuery Data Model) instance that may include both new nodes created by the TransformExpr and existing nodes. TransformExpr has special semantics: it is *not* an updating expression, as it does not modify any existing nodes. Hence, an XCore query that merely contains UpdExpr as subexpressions of a TransformExpr is *not* an updating query.

In our XCore rewriting framework, all three algorithms use a by-value based semantics, which means that target nodes may not stem from an XRPC function result, or from a function parameter (if the updating expression occurs inside an XRPC function body). Hence, we enforce that all UpdExprs, denoted \( V_a \), must be executed on the same peer that opened the document using \texttt{fn:doc()}. This, in turn, enforces that all expressions \( V_a \) (except TransformExpr) which depend on a \( v_{ai} \in V_a \), must be executed on the local peer. This is because \( V_a \) could only parse-depend on a \( v_{ai} \), as updating expressions are not allowed in a variable binding. Decomposing an expression in \( V_a \) would cause the \( v_{ai} \) to be executed on a remote peer. To correctly identify the target nodes of an UpdExpr, all expressions \( V_a \) that produce target nodes for a \( v_{ai} \), must also be executed on the local peer. When decomposing an updating query, the vertices \( V_a, V_i, \) and \( V_a \) in the query’s \( d \)-graph are never valid decomposition points, regardless of the parameter passing semantics used by the decomposition algorithm. The following *XQUF insertion conditions* should be added to the insertion conditions of each decomposition algorithm.

**XQUF Insertion Conditions** Given a \( d \)-graph \( G \) and a subgraph \( G_{rs} \) of \( G \) rooted at vertex \( r \), under any semantics, \( r \) is in the set \( I(G) \) of valid decomposition points, iff \( r \) also satisfies all of the following conditions:

(a) \( r \), rule \( \notin \{ \text{UpdExpr}, \text{TargetExpr} \} \)

(b) \( \exists v \in V(G): v \), rule \( \notin \{ \text{UpdExpr} \} \land r \), rule \( \neq \text{TransformExpr} \land r \sim v \land (∃ \forall m \in V(G): v_m \), rule \( = \text{TransformExpr} \land r \sim v_m \sim v) \)

(c) \( \exists v \in V(G): v \), rule \( = \text{TargetExpr} \land v \sim r \land (∃ \forall p \in v \) \text{Paths} \land ∃ p \in r \) \text{Paths} \land \text{starts-with} (p_t, p_s) \)

Condition (a) avoids decomposing any UpdExpr and Target-Expr. Condition (b) states that if \( r \) is not a TransformExpr, \( r \) may not depend on an UpdExpr, unless the UpdExpr is a subexpression of a TransformExpr, on which \( r \) depends. Condition (c) states that \( r \) may not be decomposed, if \( r \) produces target nodes of an UpdExpr. We say \( r \) *produces target nodes*, iff a returned path \( p_s \) of \( r \) is a prefix of a returned path \( p_t \) of \( v \), i.e., nodes returned by \( r \) include target nodes. Note that although the path annotations were introduced under the by-projection semantics, the analysis of projection paths is orthogonal to all semantics described in this work. So now we add it to the by-value and by-fragment semantics as well. We use the rules defined in [81] to propagate projection paths of the UpdExprs. Note that condition (b) allows a TransformExpr to be decomposed by all three decomposition algorithms, as it always makes (deep) copies of its source nodes. If a TransformExpr is executed on peer \( P \), \( P \) becomes the “local peer” for all new nodes created by this TransformExpr. With condition (a), we prevent UpdExprs in the modify clause of a TransformExpr from being separated from the TransformExpr (i.e., executed on another peer than \( P \)). Thus, the UpdExprs in the modify clause will also be executed on \( P \), which is the local peer of their target nodes. This confirms the XQUF semantics that UpdExprs may only be applied to local nodes. In the remainder of
this section, we continue our discussion on processing UpdExprs that are not subexpressions of a TransformExpr.

5.7.2 Updating XCore Queries on Remote Documents

We now extend the semantics of XQUF to allow updates on remote documents (i.e., documents identified by an xrpc:// URI scheme). We first provide the semantics for such updates in normal non-distributed execution (i.e., data shipping): the read-only part of the query is evaluated first, retrieving (a copy of) all accessed remote documents to the local peer, which results in a PUL. Then, the standard XQUF function upd:applyUpdates() is executed to carry through all update actions in the PUL. This could modify (some of) the local copies of the remote documents. Finally, as an additional step, for each affected remote document, an fn:put() is executed by passing the document’s original URI and its new contents, effectively replacing the existing document on the remote peer with the modified one. Note that the semantics do not apply to XCore queries only containing transform expressions, as they are read-only queries. Thus, no additional fn:put() is executed to overwrite the existing documents.

Formal Semantics Let $Q_u$ denote an XCore query containing at least one UpdExpr on a remote document and $G_u$ its $d$-graph. $D_u(Q_u)$ denotes the set of affected documents that may be updated by $Q_u$:

$$D_u(Q_u) = \{(\text{uri})\mid \exists v, v_r, v_z \in V(G_u) : v_r \leadsto v_z \land \{v_r, v_z\} \in E(G_u) \land v_r.\text{rule} = \text{FunApp} \land v_r.\text{val} = \text{doc}'' \land v_z.\text{rule} = \text{Literal} \land v_z.\text{uri} = v_z.\text{val} \land v_z.\text{rule} = \text{TargetExpr} \land \exists p \in v.\text{Paths} \land \text{fn:starts-with}(p, \text{uri})\}$$

$D'_u(Q_u)$ is a subset of $D_u(Q_u)$, which contains the affected remote documents: $\forall d'_r \in D'_u(Q_u) : \text{starts-with}(d'_r.\text{uri}, \text{xrpc://"})$. The auxiliary functions host() and path() extract the peer identifier $P$ and the document name $D$ from an XRPC URI “xrpc://$P$/D”, respectively. Each query operates on a database state ($db^p$), which includes the documents and their contents persistently stored in the XML database on $p$. The dynEnv.docValue from [67] corresponds to $db^p$ used here. As a database may be changed by updates, we can view it as a function over time $t$ as $db^p(t)$. Time values $t$ are assumed to stem from some cardinal domain, and we are also assuming a fine granularity, such that each query execution action will take at least one time unit. 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.

The formal semantics of distributed updates is\footnote{We use the ‘;’ sign to suggest an order in the evaluation of the premises.}:

$$\forall d'_r \in D'_u(Q_u) : \text{fn:doc}(d'_r.\text{uri}) \Rightarrow D'_u(Q_u)$$

$$\forall d'_r \in D'_u(Q_u) : \text{fn:put}(d'_r.\text{node}, d'_r.\text{uri}) \Rightarrow ((), \text{db}^p), D'_u(Q_u) \Rightarrow (\text{db}^p, D'_u(Q_u))$$

$$\forall d'_r \in D'_u(Q_u) : \text{fn:delete}(d'_r.\text{uri}) \Rightarrow ((), \text{db}^p)$$

(\text{R}^u)

The rule $\text{R}^u$ states that the execution of an updating query $Q_u$ at the local peer $p_0$ in the database state $db^{p_0}$ starts with retrieving the remote documents $D'_u(Q_u)$, which could potentially be affected by $Q_u$, to $p_0$. This yields a set of local copies $D'_u(Q_u)$ of $D'_u(Q_u)$. Note that

\footnote{As explained in Section 5.4, computed URIs and invocations of fn:collection() are represented by $. During the runtime, when the actual values of the wildcard symbols are available, more URIs might be added to the set $D'_u(Q_u)$ on the fly.}
this step does not change $db^{p_0}$, as the documents in $D''_u(Q_u)$ are transient documents. Then, $Q_u$ is executed in $db^{p_0}$ with the additional documents $D'_u(Q_u)$ which first yields a PUL $\Delta$. Subsequently, $\text{upd:applyUpdates()}$ is executed to apply all update primitives in $\Delta$ to the affected documents. Updates in $\Delta$ that should be applied on remote documents $D''_u(Q_u)$ are applied on their local copies $D''_u(Q_u)$ instead. This step produces a new current database state $db^{p_0}$, which could differ from $db^{p_0}$ (if $\Delta$ contains updates on really local documents), and a set of changed local copies $D''_u(Q_u)$. Finally, an additional step is executed, which calls $\text{fn:put()}$ to store each $d''_t \in D''_u(Q_u)$ on its hosting peer and overwrite the existing $d''_t \in D''_u(Q_u)$. This step also creates a new current remote database state $db^{\text{host}(d''_t, \text{uri})}$ on each hosting peer. As the rule $R^u$ only applies $\Delta$ at the end of query execution, updates are not visible for the same query, which confirms the XQUF semantics. Hence, if $\Delta$ only contains updates on a single document, this rule already provides atomic updates.

**Isolation Levels** Note that the - potentially multiple - $\text{fn:put()}$ together with potential updates on some local documents constitute a distributed updating query. Depending on the semantics desired by the user, this distributed updating query could be run in a certain consistency level, which has been discussed in detail in our previous work [180]. One option is no consistency at all in which some documents may get updated, but other document updates may fail or get lost. By tagging queries with a unique ID, the repeatable read consistency level can be easily achieved. To ensure distributed atomic updates, [180, 181] shows how the WS-AtomicTransaction standard [55] can be integrated into XRPC to provide 2PC. In addition to repeatable reads and atomic commits, the lost updates anomaly can be avoided if participating peers abort the 2PC commit when another updating query or $\text{fn:put()}$ has modified an updated document already. Note that these semantics can also be supported by the XRPC Wrapper if the XQuery engine is XRPC oblivious. Given the design goal for XRPC of supporting P2P applications on the Internet, we refrained from attempting to define higher consistency levels (e.g. distributed serializability), as the overhead of these are impractical in such environments. We consider more advanced distributed consistency levels for P2P on the Internet a topic of future work, and consider it out of scope here, where we focus on semantically correct distributed query rewriting.

**Atomic Updates with Isolation** We now define an improved semantics that provides repeatable reads and atomic distributed commit, described by the rule $R^u_{\text{repeat}}$:

\[
D'_u(Q_u) = \varnothing; \\
\forall d_x \in D_u(Q_u): p_x = \text{host}(d_x, \text{uri}); \\
\text{send}^{p_0-\text{req}}(q_u, "\text{fn:doc"}, d_x, \text{uri}); t^{p_0}_q \geq t^{p_0}_q; \\
\text{db}^{p_0}(t^{p_0}_q) \models d_x, \text{node} = \text{fn:doc}(d_x, \text{uri}) \Rightarrow d'_x, \text{node}; \\
\text{send}^{p_0-\text{rep}}(d_x, d_x, d'_x, \text{node}); \\
\text{db}^{p_0}(t^{p_0}_q) \models D'_u(Q_u) = D'_u(Q_u) + (d_x, \text{uri}, d'_x, \text{node}); \\
\text{db}^{p_0}(t^{p_0}_q), D'_u(Q_u) \models \text{upd:applyUpdates}(\Delta) \Rightarrow D''_u(Q_u); \\
\forall d_x' \in D''_u(Q_u): p_x = \text{host}(d_x', \text{uri}); \\
\text{send}^{p_0-\text{req}}(q_u, \text{PREPARE}, "\text{fn:put"}, d_x', \text{node}, d'_x, \text{uri}); \\
\text{db}^{p_0}(t^{p_0}_q) \models \log("\text{fn:put"}, d'_x, \text{node}, d'_x, \text{uri}) \Rightarrow r; \\
\text{send}^{p_0-\text{rep}}(r_q, r); \\
\text{db}^{p_0}(t^{p_0}_q) \models Q_u \Rightarrow ()
\]

There are several differences between this rule $R^u_{\text{repeat}}$ and the previous rule $R^u$. First, each query is tagged with a unique query ID $q_u$, so that each peer will use the same database
state \( db^{p_i}(t_{q_u}^p) \) to handle requests originating from the same query. Usually, \( db^{p_i}(t_{q_u}^p) \) is the current state of peer \( p_i \) at the time \( t_{q_u}^p \), when query \( Q_u \) visits \( p_i \) for the first time. This ensures repeatable reads, if \( p_i \) is visited multiple times by \( Q_u \). Second, \( \text{fn:put()} \) is not executed immediately on a remote peer \( p_x \), instead, it is sent as a PREPARE request. The execution of \( \text{fn:put()} \) is first “prepared”, yielding a decision \( r \), which could be COMMIT or ABORT. The execution of \( \text{fn:put()} \) will be finalised after \( p_0 \) has received the decision \( r \) from all \( p_x \), with a separate COMMIT (or ABORT) message [180]. Finally, a minor difference: \( D'_{u}(Q_u) \) contains a copy of all potentially affected documents, including really local documents. This is for presentation purpose only. It indicates that updates on really local documents are also first applied to their copies, when executing \( \text{upd:applyUpdates()} \). The local peer also computes the decision \( r \). All updates (on both really local document and remote documents) will later be committed (or aborted) atomically. Hence, the rule \( R_{repeat}^u \) does not modify the database state \( db^{p_0}_u(t_{q_u}^{p_0}) \).

**XQUF Rewrites** Rather than using \( \text{fn:doc("xrpc://P/D")} \) followed by an additional \( \text{fn:put ("xrpc://P/D")} \) after \( \text{upd:applyUpdates()} \), i.e., *data shipping*, we can try to use updating functions that could be pushed with XRPC to do remote updates, i.e., *function shipping*. Note that XQUF as supported by XQuery engines, only supports updates on local XML nodes, so this is our target. In principle, we cannot push any \( \text{UpdExprs} \), except *homogeneous updating expressions*. An \( \text{UpdExpr} v_u^h \) is homogeneous, iff all returned paths of its \( \text{TargetExpr} v_u^h \) start with the same "xrpc://P", i.e., the update affects only nodes that stem from a single peer. Hence, the update can be pushed to that peer using an XQUF *updating function* such that it acts only on local documents there. Note that if \( v_u^h \) is decomposed, in principle, it should be executed on \( P \), because executing \( v_u^h \) on another peer than \( P \) implies the same semantics as executing \( v_u^h \) on the local peer, which makes remote execution not meaningful. The insertion conditions for updates formulated in Section 5.7.1 also applies for pushed updating expression: target nodes of an \( \text{UpdExpr} v_u \) may not be passed to a remote peer as function parameters or results. Decomposition of \( v_u^h \) thus requires that all expressions \( V_v^h \) that produce target nodes of \( v_u^h \) must be executed in the same remote function as \( v_u^h \). So, we need to find the smallest (super-)expression \( v_s \) that contains both \( v_u^h \) and \( v_u^h \):

Let \( Q_u \) be an updating query containing the homogeneous \( \text{UpdExpr} v_u^h \) and \( G_u \) its d-graph. Let \( v_w^h \) be the \( \text{TargetExpr} \) of \( v_u^h \) (i.e., \( (v_u^h, v_w^h) \in E(G_u) \land v_w^h.\text{rule}=\text{TargetExpr} \)). We define \( V_u^h \) as:

\[
\forall v_j \in V_u^h: v_j \sim v_i \lor (\forall p_i \in v_j, \text{Paths}, \forall p_u \in v_u^h.\text{paths}: \text{fn:starts-with}(p_u, p_i))
\]

and define \( v_s \) as:

\[
(v_s \sim v_i \lor v_s = v_i) \land \forall v_j \in V_u^h: v_j \sim v_i \land \exists v_s \in V(G_u): v_s \sim v_i \land v_s \sim v_u^h \land \forall v_l \in V_u^h: v_l \sim v_i
\]

Then, \( v_s \) could be a valid decomposition point. If no such point can be found, we fall back to the data shipping strategy (i.e., local execution and a \( \text{fn:put ("xrpc://P/D")} \) at the end of query execution). For updating queries containing both push-able and not push-able \( \text{UpdExprs} \), however, there is an additional issue to deal with: we can only push an \( \text{UpdExpr} v_{u_0} \) if we can guarantee that no other \( \text{UpdExprs} \) elsewhere in the query update nodes from the same documents (a *clash*), or, if another \( \text{UpdExpr} \) does, it can also be pushed. This is because all \( \text{UpdExprs} \) that are not pushed will generate an \( \text{fn:put()} \) in the end, which would potentially overwrite the pushed updating actions or other \( \text{fn:put()} \)s, from the same transaction. However, if all updates to the same document are pushed, the 2PC protocol used in XRPC ensures correct execution [180].
One more constraint must be added to the definition of $v$ above:

$$
\forall v \in V(G_u) : v, \text{rule} = \text{UpdExpr} \land \neg \text{isHomogen}(v) \land \\
\exists p_x \in v_x \cdot \text{Paths}, \exists p_u \in v_u \cdot \text{Paths} : \text{docPeer}(p_x) = \text{docPeer}(p_u)
$$

where, given a path $\text{doc("xrpc://\mathcal{P}/\mathcal{D}")/[\text{SimplePath}]})$, the function $\text{docPeer()}$ returns $\mathcal{P}$; and the function $\text{isHomogen()}$ is defined as:

$$
\text{isHomogen}(v_x) \iff \exists v_w \in V(G_u) \land (v_x, v_w) \in E(G_u) \land v_w, \text{rule} = \text{TargetExpr} \land \\
\forall p_i, p_j \in v_w \cdot \text{Paths} : \text{docPeer}(p_i) = \text{docPeer}(p_j)
$$

## 5.8 Evaluation in MonetDB/XQuery

We have implemented the proposed algorithms in MonetDB/XQuery [41], a purely relational XDBMS that uses the *Pathfinder* [88] XQuery compiler. We use the XRPC extension for remote function evaluation. Note that, as no other comparative results exist, the main goal of our experiments is to show the impact of the proposed techniques in a step-by-step fashion.

### 5.8.1 Read-Only Queries

For all our experiments, the test platform consisted of three 2GHz Athlon64 Linux machines connected in a local network (LAN). Each was equipped with 2GB RAM. The benchmark data used is XMark [159], a popular XML benchmark for evaluating XQuery efficiency and scalability. The data set was generated using scale factors 0.1, 0.2, 0.4, 0.8 and 1.6. A data set is stored on each remote peer. We conducted three groups of experiments: bandwidth usage, query execution time and runtime projection precision.

We slightly modified the query $Q_2^i$ (in Table 5.3) so that it conforms to the XMark schema as the following:

```xml
(let $t:= let $s:=\text{doc("xrpc://peer1/xmk_nn_MB.xml")/child::site/child::people/child::person}
  return for $x in $s return if ($x/descendant::age < 40) then $x else ()
  return for $e in (let $c := \text{doc("xrpc://peer2/xmk_nn_MB.auctions.xml")}
    return $c/descendant::open_auction)
  return if($c/child::seller/attribute::person = $t/attribute::id)
    then $c/child::annotation else () )/child::author
```

All techniques discussed in this paper are applied to the above query: (i) under the pass-by-value semantics, only the expression

```xml
\text{doc("xrpc://peer1/xmk_nn_MB.xml")/child::site/child::people/child::person}
```

can be decomposed and executed on peer1; (ii) under the pass-by-fragment semantics, we can decompose both the second let clause ("let $s := \ldots") and the second for-loop ("for $e in \ldots"), and execute them on peer1 and peer2 respectively. The variable $t$ becomes the parameter of the generated function containing the second for-loop (see also Table 5.4); (iii) under the pass-by-projection semantics, the query is decomposed in the same way as using pass-by-fragment, however, when serialising the request messages, a projection of $t/attribute::id$ (parameter projection) and $c/child::annotation/child::author$ (result projection) is calculated. The test set thus contains four queries in total, and each of them is executed on 2 documents of sizes 10, 20, 40, 80 and 160MB.

In this case, code motion is ideal, as it is able to send just strings, not nodes. However, if we would replace the final step $\text{child::author by parent::*}$, then just applying code motion and no projection provides mediocre performance similar to by-value. It is the ability
of projection to decompose almost any query at little cost, that makes it the overall method of choice.

**Bandwidth Usage** Figure 5.9 shows the bandwidth used by each benchmark query on different sets of documents, i.e., the total size of XML documents plus total size of XML messages transferred among peers, in its y-axis. The x-axis is the total size of the XML documents used by each query. The pure data-shipping XQuery query (the leftmost bar) has the largest bandwidth usage, as both documents used by the query have to be shipped. By-value decomposition can push the XPath step:

```xml
doc("xrpc://peer1/xmk_nn_MB.xml")/child::site/child::people/child::person
```

to be evaluated on peer1, which reduces the amount of data sent from peer1 to the local peer. However, the second document "xmk_nn_MB.auctions.xml" still has to be sent fully. The by-fragment passing semantics allows the local peer to push predicates to both peers, achieving a distributed semi-join plan. Also, it strongly reduces message size by avoiding duplicating the same XML node multiple times. Pass-by-projection further brings down message sizes due to reduced response message size. For example, when sending the result of the remote execution of the second for-loop, the response message will only contain annotation nodes with their author child nodes. When applied to pass-by-fragment, code motion has a larger effect in reducing message size than when it is applied to pass-by-projection. This is because in pass-by-fragment, complete person nodes (i.e., including all their descendants) are serialised, while in pass-by-fragment with code motion, only the values of the id attributes are serialised. In pass-by-projection, however, the message size has already minimised the data to be sent, i.e., only person nodes and their id attributes, hence, the effect of applying code motion here is negligible. In general, we observe good scalability of pass-by-fragment and pass-by-projection in bandwidth usage.

**Execution Time** Figure 5.10 shows the execution time breakdown of all four queries on documents of 320MB in total. The execution time is divided into five parts: *shred* is the time to receive a document from the remote peer and shred it in to the XML database; *local exec* is the execution time of the query at the local peer, including query parsing, module loading, etc; *(de)serialise* is the time spent on generating/shredding the XML messages and extracting parameter/result values from the messages; *remote exec* is the time to execute the called functions on remote peers; and *network* is the time spent on sending/receiving the XML messages. From Figure 5.10, the following observations can be made: (i) in the data-shipping only query and the by-value decomposed query, data shredding is the main bottleneck, ei-
ther because the whole document will be shipped (data-shipping), or an XML node might be shredded multiple times (by-value). Especially in the data-shipping query, more than 99% of the total execution time is spent on getting the documents from remote peers and shredding them; (ii) when pass-by-fragment and pass-by-projection semantics are used, the total execution time is significantly reduced (about 84∼94%, compared with data-shipping and pass-by-value). This is easily explained as these techniques reduce the amount of data exchanged to be less than 10% of the original document sizes. Even with the overhead introduced by remote execution (i.e., ‘(de)serialise’+‘remote exec’), pass-by-fragment or pass-by-projection are preferred over the data-shipping method. (iii) pass-by-projection performs even better than pass-by-fragment (about 35% improvement), which is again explained by the reduced bandwidth usage, as shown in Figure 5.9.

Figure 5.8 shows the execution time of all queries on documents of increasing sizes. It indicates that the two enhanced parameter passing techniques achieve good scalability. On average, pass-by-fragment and pass-by-projection achieve a performance improvement of roughly 94%, compared with data-shipping; this is proportional to the decrease in bandwidth usage, which is approximately 96%. Even on small documents (20MB), the proposed techniques are preferred over the data-shipping methods.

**Runtime Projection Precision** Our new runtime projection technique combines intermediate query results with runtime execution and relative XPath paths. Due to selections (by e.g., predicates and value comparisons), the run-time projection node sets obtained may be much smaller than suggested by compile-time projection paths, used in [125]. We used our by-projection benchmark query to compare runtime projection with compile-time projection on various sizes of the XMark document “xmk_nn_MB.xml”. In this experiment, the compile-time technique projects all person elements and their age, while our runtime projection technique will only project those person elements that have an age descendant larger than 45. Figure 5.6 shows runtime projection to be 5 times more precise in terms of the size of the projected document. In this experiment, the investment in run-time XPath evaluation pays off due to the more precise results, as shown in Figure 5.7.

### 5.8.2 XQUF Queries

For the updating XCore queries, we have conducted two groups of experiments to compare performance of updating remote documents with or without XRPC. The first group corresponds to the generic strategy discussed in Section 5.7 where a remote document is first retrieved to the local machine (with fn:doc()), then the updates are applied on the local copy of the remote document, and finally the updated document is written to the remote peer using fn:put(). We call queries in this group “GUP queries” (i.e., Get-Update-Put). In the second group, called “XRPC queries”, updates are applied directly on the original document at the remote peer with XRPC using so-called updating functions as specified by the XQUF:

```xml
module namespace fcn = "foo";

declare updating function fcn:doInsert($d as xs:string, $node as node())
    {do insert $node into doc($d)/site};

declare updating function fcn:doDelete($d as xs:string, $pid as xs:string)
    {do delete doc($d)//person[@id=$pid]};

declare updating function fcn:doRename($d as xs:string, $pid as xs:string, $nm as xs:string)
    {do rename doc($d)//person[@id=$pid] into $nm};
```
We tested all four kinds of updates, keeping the granularity of the updates constant, affecting 100 \textit{person} nodes. For example, the insert query in XRPC looks as follows:

\begin{verbatim}
import module namespace fcn="foo" at "http://example.org/foo.xq";
fcn:doInsert("xrpc://p2/xmark200mb.xml", doc("Persons100.xml")/persons)
\end{verbatim}

All updating queries were applied on XMark documents of 200, 400, 600, 800 and 1000 MB, respectively. The data set is stored on one peer, which acts as the remote peer. The total execution time of all queries are shown in Figure 5.11.

For all four kinds of update queries, XRPC is significantly faster than GUP. The relatively small performance differences between different kinds of updates reflects the MonetDB/XQuery implementation of the XQUF. We can conclude that with increasing document sizes, the absolute benefits of XRPC grow linearly, which is caused by the additional full serialisation, network copy, and shredding for the “Get” phase, followed by full serialisation and network copy steps in the “Put” phase, performed by the GUP approach. As the number of updates is small, the total bandwidth usage of all GUP queries are approximately twice the documents size, as shown in Figure 5.12, whereas the XRPC query only sends the function parameters and results (tens of KB). In Figure 5.13, the bars at the left-hand-side show the time breakdown of GUP queries, while the bars at the right-hand-side show the time breakdown of XRPC queries; all were run on a 1GB document. From Figure 5.13, it can be seen that the GUP queries spend a large amount of time on adding the document to the local and remote database (shown as “gup add doc remote” and “gup add doc local”). They also spend a significant amount of time on exchanging the document between the local peer and the remote peer (shown as “gup network”). However, the times spent on actually applying the updates (shown as “gup exec update”) are only a very small portion of the total execution times. On the other hand, for the XRPC queries, the only dominant factor in the total execution time is the time spent on applying the updates (shown as “xrpc remote exec”), while the times spent on processing the request and response messages (i.e., serialise, send and deserialise) are negligible.

We finally recall that in all experiments (including the read-only ones) we used a local area network (LAN); but in a WAN environment, where much lower network performance is common, the benefits of our query decomposition techniques will be larger, as we showed by their strongly reduced network bandwidth use.
CHAPTER 5. XQUERY DECOMPOSITION

Figure 5.8: Execution time of read-only queries
Figure 5.9: Bandwidth usage of read-only queries
Figure 5.10: Time breakdown of read-only queries on 320MB data

Figure 5.11: Execution time of updating queries
Figure 5.12: Bandwidth usage of updating queries
Figure 5.13: Time breakdown of updating queries on 1000MB data
5.9 Conclusion

In this chapter, we have described a framework for distributed execution of full-fledged XQuery including XQUF, focusing on the issue of providing equivalent query decompositions, in the face of semantic differences when (parts of) nodes are shipped across the network in XML messages. We first carefully characterised the problems that may occur regarding node identity and structural XPath relationships in such a distributed setting. Then, we proposed a series of techniques such as pass-by-fragment and the use of a novel runtime XML projection method for serialising XML messages, that remove all but one semantic problems and strongly improve performance, as shown by experiments on the open-source MonetDB/XQuery XDBMS (http://monetdb.cwi.nl). We also discussed the semantics of updating both local and remote documents using XQUF expressions, and additional constraints that should be added to the proposed techniques to guarantee semantic equivalence for such queries.

Our main future work is an issue left out-of-scope here: deciding on distributed query placement after decomposition. In this area, we also contemplate using runtime methods to improve optimisation quality.