XRPC

Efficient Distributed Query Processing on Heterogeneous XQuery Engines

张颖
Zhang Ying

XRPC

Efficient Distributed Query Processing on Heterogeneous XQuery Engines

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de

Universiteit van Amsterdam,

op gezag van de Rector Magnificus

prof. dr. D. C. van den Boom

ten overstaan van een door het college

voor promoties ingestelde commissie,

in het openbaar te verdedigen in de Agnietenkapel

op donderdag 8 juli 2010, te 12:00 uur

door Zhang Ying (张颖)
geboren te ChongQing, China

Promotiecommissie:

Promotor:	Prof.	dr. Martin L. Kersten
CoPromotor:		Dr. Peter A. Boncz
Overige leden:	Prof.	dr. Torsten Grust
	Prof.	dr. Arjen P. de Vries
	Prof.	dr. Maarten de Rijke
		Dr. Maarten Marx

Faculteit:

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam

The research reported in this thesis was finished at current position of the author at CWI, the Dutch national research laboratory for mathematics and computer science, within the theme Data Mining and Knowledge Discovery, a subdivision of the research cluster Information Systems.

SIKS Dissertation Series No. 2010-26.
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-90-9025263-6

Contents

1 Introduction

1.1 Motivation

The use of the Internet as the means for individuals and organizations to interact and exchange information has significant consequences for data management technologies. The data management research and industry communities have therefore embraced the W3C recommendations around XML, which is the de facto standard for information exchange on the Internet[1]. Around XML an ecosystem of other Web standards has emerged, including XML Schema, SOAP, WSDL, XSLT, XPath and XQuery. While the design of these Web standards is in principle not a question of computer science research, rather one of design by committee, the fact that the Internet has adopted these standards as the means of (semi-) structured data exchange creates a reality that provides relevance and urgency for finding answers to new research questions, such as presented here in this thesis.

Generally speaking, one can view XML data on the Internet as a huge wide-area XML database containing semi-structured information. The individual machines on the Internet belong to truly billions of different individuals and millions of different organizations, and consequently act as independent peers. The term "peer" is used here, as in principle there is no hierarchy among these machines. Cooperative software systems where multiple peers work together without necessarily a central point of control, are called Peer-to-Peer (P2P) systems. Also, building systems that combine data from multiple peers on the fly is a new way to create rich applications with little effort ("mashups"). As such, the question is how the creation of such cooperative peer systems can be facilitated.

To ease the development of data-intensive P2P applications, we envision a P2P Database Management System (P2P DBMS) that acts as a database middle-ware system. It manages dynamic collections of heterogeneous data sources (peers, with different software installed) and provides a uniform database abstraction to the application. The goal of this Ph.D. work at large is to research which features such a database abstraction should offer, and how it can be realized efficiently by extending and combining existing Database Management Systems with P2P technologies.

Please note here, that P2P DBMS means something different than the common notion of P2P systems, which the general public knows best as P2P file sharing systems, often used for illegal downloading of copyrighted media. Whereas a P2P file sharing system typically manages (binary) files, a P2P DBMS manages (semi-)structured information. In the context

[1] We could not resist using this by now infamous phrase.

of this study we focus on data in XML format, i.e., on P2P XML Database Management Systems (P2P XDBMS). File sharing systems typically only allow simple queries, based on meta data of the files, e.g. using keywords or simple properties. P2P XML database systems, in contrast, allow users to query the contents and structures of the XML data, using an XML query language. In the context of this thesis, we assume the query language to be XQuery (which is a super-set of XPath).

In our quest for creating P2P XDBMS technology, we first focused on Distributed XDBMS technology. The distinction between Distributed and P2P technology is that in the former, users (e.g., application programmers) are aware of on which sites (i.e., peers) data are located. Distributed queries typically involve specific and explicit locations where data is to be queried from. In P2P systems that also target large environments, where users cannot keep track which data is on which peer, and where the group membership is highly volatile (peers enter and leave continuously and unpredictably), users are typically shielded from explicit knowledge where data is located. We focused first on Distributed XDBMS technology as this area also was unexplored, and Distributed XDBMS technology can be seen as a building block for P2P XDBMS technology.

In this research work, we have hence focused on distributed XML DBMS aspects including query execution, query optimisation, and transaction management. The result of this work is XRPC (stands for Xquery RPC), a minimal XQuery extension that enables efficient distributed querying of heterogeneous XQuery data sources. In the remainder of this section, we motivate the choices we made in the design and implementation of XRPC, and in how XRPC is used for distributed XQuery processing over heterogeneous data sources.

1.1.1 Interoperability

The primary design goal of XRPC is to create a distributed query mechanism with which *different* query processors at different sites can jointly execute queries. An important way for a system to achieve interoperability with other systems is to adhere to published interface standards. For this reason, our choice for the basic building bricks of XRPC naturally falls on XML, XQuery, SOAP and HTTP – all well-defined and generally accepted Web standards.

After its first public revealing at the SGML 1990 Conference, XML has quickly gained enormous popularity as a data exchange format among *different* applications and organizations. XML owns its success to several properties: *(i)* it is hardware and (to some extent) software independent; *(ii)* it is a self-documenting format, i.e., a single document contains both description of the structure and field names, and values; *(iii)* it is suitable for data with or without a clear structure, thus one can capture plain text, and text with some structure, e.g., e-mails, all the way to very structured information such as tuples; and *(iv)* XML schemas are extensible, which makes it easy to support backward compatibility. Consequently, the choice for XML as the data model and XQuery as the query language – and web standards in general – eases many aspects of distributed data management.

The only way for two different systems to communicate is to use an open protocol. However, to the best of our knowledge, none of the existing proposals of distributed XML query processing use or define such an open standard protocol. Some of them use a non-open protocol, e.g., AXML [9], and some of them use a proprietary protocol, e.g., DXQ [70]. In such systems, heterogeneity is hard to achieve, or impossible. Therefore, we choose to use an open network protocol to enable communication among different XQuery engines. Although W3C has a Candidate Recommendation *XML Fragment Interchange* [87] and SOAP has a *SOAP*

RPC subprotocol, these two standards have a common problem: the data types they support do not match the data types defined by the XQuery Data Model (XDM) [71]. The XML Fragment Interchange defines a way to send fragments of an XML document – regardless of whether the fragments are predetermined entities or not – without having to send all of the containing document up to the fragments in question. Thus, XML Fragment Interchange only supports XML elements, but no atomic values; while SOAP RPC only supports exchanging of atomic values without XML elements. Neither approach deals with (XQuery) sequences of heterogeneous types.

As a result, XRPC also encompasses a SOAP-based network protocol, the *SOAP XRPC* protocol. Network communication in XRPC uses XML messages over HTTP. There is ubiquitous support for URIs, specifically HTTP networking, and XQuery engines are perfectly equipped to process XML messages. Moreover, an XML-based message protocol makes it trivial to support passing values of any type from XDM. The choice for SOAP brings as additional advantages seamless integration of XQuery data sources with web services and Service Oriented Architectures (SOA) as well as AJAX-style GUIs.

1.1.2 Extending XQuery with Query Shipping

For efficient processing of XQuery queries in our target environments, the first task is to extend XQuery with a query shipping model.

By default, the XQuery 1.0 standard already allows querying XML documents distributed over the Internet, using a *data shipping* model. The built-in function `fn:doc()` fetches an XML document from a remote peer to the local server, where it subsequently can be queried. The recent W3C Candidate Recommendation XQuery Update Facility (XQUF) introduces a built-in function `fn:put()` for remote storage of XML documents, which again implies data shipping. In P2P settings such a data shipping model has several serious drawbacks [91, 92, 180]. It is highly *inefficient* in terms of network latency. For instance, aggregation queries on huge remote XML documents that produce only small results incur large network costs, because complete documents have to be transferred to queries' local peers. This directly leads to *poor scalability* both in sizes and in the number of remote documents involved in a query. *Bad load balancing* is another drawback, because all query execution happens locally, i.e., at the query originator, missing possibilities of exploiting query processing capabilities of remote peers.

There have been various proposals to equip XQuery with a *query shipping* model, especially the *function shipping* style distributed querying abilities [74, 134, 144, 172]. In this research work, we choose to extend XQuery with a Remote Procedure Call (RPC) mechanism, which we call *XRPC* (i.e., XQuery RPC). On the syntax level, we consider XRPC an incremental development of the existing extensions, with specific advantages concerning *simplicity* and *optimisability*. Considering simplicity, XRPC adds RPC to XQuery in the most simple way: adding a destination URI to the XQuery equivalent of a procedure call (i.e., function application). XRPC is optimisable in that it makes explicit the input data (parameters) of a remote query and its result type through the function signature, routinely identified during query parsing in existing XQuery systems. Also, functions can be defined in XQuery modules, and compiled separately in advance, making it easy to do query plan caching and thus accelerate distributed query processing.

1.1.3 Efficiency

In P2P settings (e.g., WAN), query execution times are mainly determined by network latency. The study in [79] demonstrates on a real system that, in distributed systems, the key to scalability is minimising the number of messages. Therefore, in the design of XRPC, we have paid special attentions to minimize the number of messages exchanged among peers and also the sizes of messages. Two new concepts have been introduced: *Bulk RPC* and *runtime XML projection*.

Bulk RPC The XRPC extension allows maximal flexibility of making XRPC calls. An XRPC call may be placed anywhere in an XQuery query, where a normal XQuery function call is allowed. For example, XRPC calls can be included in `for`-loops, which often contain large numbers of iterations. Clearly, a naive implementation that handles XRPC calls one-at-a-time will not scale. The SOAP XRPC protocol supports a so-called Bulk RPC concept, which allows the system to compute multiple applications of the same function (with different parameters) in a single request/response network interaction[2]. Bulk RPC is much more efficient than repeated single RPC as network latency is amortized over many calls, and performance becomes bounded by network bandwidth or CPU throughput (hardware factors that scale much better than network latency). Another way to look at Bulk RPC is that it exposes bulk execution opportunities, such that e.g. a function that selects with a constant argument is turned into a join against the sequence of all arguments. Bulk RPC thus has a direct correspondence with set-oriented processing as offered by query algebras, and we believe it can be generally applied in any algebraic XQuery implementation.

Runtime XML Projection XML projection[3] [24, 60, 66, 125, 52, 31, 56, 83, 61, 111] is a technique popularly used, e.g., by streaming systems, to reduce the amount of data that needs to be processed for a query and to reduce memory usage. The basic idea of XML projection is, for a given XQuery query Q and an XML document $\mathcal{D}$, to extract a smaller part $\mathcal{D}'$ of $\mathcal{D}$, which is used to execute Q such that $Q(\mathcal{D}) = Q(\mathcal{D}')$. A projection technique usually conducts a compile-time path analysis on Q, to derive a set of XPath expressions $\mathcal{P}$ that over-estimate the nodes that Q touches. Then, a loading algorithm applies $\mathcal{P}$ on $\mathcal{D}$ (from a file or a stream) to generate the projected document (or stream) $\mathcal{D}'$, which is queried with Q.

XML projection is also extremely interesting for distributed XQuery processing, as we will see in Chapter 5, where we introduce a new concept called *runtime XML projection*. When sending XML nodes, pruning huge subtrees, which will remain untouched at the remote sites, can strongly reduce network bandwidth usage, as well as serialisation and deserialisation effort. Our runtime XML projection technique has as additional advantages, compared with compile-time techniques, higher *accuracy* and *flexibility*.

Considering accuracy, with the runtime technique, we are able to restrict the projected document using the selection predicates found in expressions, thus, the resulting projected document $\mathcal{D}'$ is often much smaller. For instance, consider the expression `//person[@id=$pid]`. A compile-time technique, lacking the ability to execute the predicate "`@id=$pid`", has to keep *all* person elements in $\mathcal{D}'$, while our runtime technique only projects the `person` element, whose `id` attributed matches the given `$pid`.

[2] Note that Bulk RPC should not be confused with semi-join, although they both aim at improving efficiency. Bulk RPC achieves this by reducing the impact of network latency, while semi-join reduces the amount of shipped data.

[3] Also called XML filtering or XML pruning.

Runtime projection is more flexible, because it can handle XPath steps on *all* axes and the built-in functions that need to access XML nodes outside the subtree of their parameters (i.e., `fn:root()`, `fn:id()`, `fn:idref()` and `fn:lang()`). While handling non-downwards XPath steps (e.g., `parent` and `ancestor`) is a cumbersome task for many streaming systems, the runtime projection technique makes it easy. Applying a `parent` step on a document at runtime is only marginally different from applying a `child` step.

Thanks to the runtime projection technique, we can stick to a copy-based, stateless approach in the design of XRPC, which minimise network interactions (hence the impact of WAN latency).

1.1.4 Stateless versus Stateful

When designing the basic SOAP XRPC protocol and its extensions, we made a conscious choice for a *by-value* parameter passing mechanism. That is, during remote function execution, the calling peer (i.e., query originator) will send a request message containing a deep-copy of the parameters to a remote peer, which executes the subexpression, and sends back a response message containing a deep-copy of the result. If the function parameters or results contain XML node-typed items, only the subtrees rooted at these items are serialised (i.e., deep-copied) into the XRPC messages. As a result of this, node-typed items lose their original node identities and structural properties, when they are exchanged among peers, which may affect the semantics of XQuery execution on such shipped nodes. This is an inevitable situation, because when XML nodes must be shipped over the network, it means that, unless one chooses to ship the entire XML document in order to preserve all structural relationships (which defeats the purpose of function shipping), pieces/snippets of the XML document must somehow be copied into the messages.

To illustrate the challenges of distributing XQuery, yet preserving XML node identity, consider a subexpression `f($a,$b)` with two parameters `$a` and `$b` of type `node()`, that is executed remotely. Complications may arise, for instance, if the subexpression `f()` tests structural XML relationships among its parameters, such as `$a/parent::b is $b`. Similar complications arise when transporting result values back over the network, and when the results of two different function calls to the same remote peer end up at the same peer. It therefore depends on the characteristics of the subexpressions `f()`, as well as on the way parameters are marshalled in and out of the network messages, whether the distributed query will behave correctly, that is, whether the distributed query is identical to local execution (blindly copying all parameters into the message does not work in this example).

One could consider a simple "callback" way of handling XQuery distribution by not sending XML snippets at all, but just some (global) node identifiers. Each time when a peer needs to execute node-specific XQuery/XPath expressions on such node identifiers, this alternative approach would communicate with the peer where the nodes originally came from, executing the node-specific expressions on that peer and returning the results. While such an approach circumvents semantic problems, it has many drawbacks: *(i)* it basically gives up on the desire to move computation to more powerful peers; *(ii)* it introduces additional network round-trips; *(iii)* it makes all distributed queries – even read-only queries – *stateful*: a single query might consist of multiple (potentially many) network requests and the query processor on each peer must keep a session context open to guarantee repeatable reads consistency, which causes extra memory consumption and lock contention; and finally *(iv)* additional protocols would be needed to properly terminate such stateful distributed queries, adding extra protocol com-

plexity, bookkeeping overhead and network latencies. In contrast, the techniques introduced with XRPC lead to flexible query distribution where subexpressions can be moved to the peer that can most efficiently process them. Typically, each peer is visited only once, thus network interactions are minimised and peers can handle the subqueries in a *stateless* manner.

Let us have a more precise look at the difference between a callback approach (i.e., stateful) and the XRPC approach (i.e., stateless). XRPC leads to $O(P_Q)$ number of network *round trips*, where P_Q is the number of different documents opened by the query Q. This is obviously the minimal number of network round trips. A naive callback approach leads to $O(P_Q * (X_Q + 2B_Q))$ network round trips, where X_Q is the number of XPath steps in the query Q, and B_Q the number of binary operators (e.g., `is`, $\ll$, `union`, etc). Although it is possible to reduce the number of network round trips caused by a naive callback approach, what we are trying to make clear here is that, in this research, the question is not so much whether XRPC is better than a callback approach, but whether the XRPC approach is *possible*. This is a major research question.

Additionally, one could also envision a network protocol that *combines* "callback" query processing with our techniques, something that might be interesting for handling distributed updating queries (because in the default XQUF semantics only locally stored documents can be updated, hence one would have to "callback" to the peer where a node originated from, to apply the update actions). However, given our target of Internet-wide P2P query processing with high network latencies, we decided against the "callback" approach in our own prototype construction, and fully focused on XQuery execution by moving XML snippets and computations on them over the network. In this research work, we have solved the problem brought by this approach in preserving semantic correctness, and also demonstrate the efficiency of this approach (Chapter 5).

1.2 Research Objective

The focus of this thesis is processing and optimisation of distributed XQuery queries. The general research question addressed here is:

> *How to support efficient processing of full-fledged XQuery queries – including those containing XQUF expressions – on large amounts of XML data served by heterogeneous XQuery engines in P2P settings?*

This question identifies three main issues: *efficiency*, *interoperability*, and *scalability*. To better understand the research question, we have refined it into more specific questions:

1. *How should we extend XQuery with a query shipping mechanism that is suitable for the targeted environments?*

   Determined by the main research question, such an extension should provide the potential for efficient, interoperable and scalable XQuery processing. Additionally, the extension should be orthogonal to all XQuery features, and it should be kept as simple as possible. By allowing *any* kind of XQuery expressions to be executed remotely, the extension provides maximal possibilities for remote execution, which in turn provides more potential for query optimisation. A simple design might look easy, but it is important for efficiency (think of administrative overhead), interoperability (i.e., easy to understand) and scalability (an extension requiring minimal administration is usually much faster than a complex one).

2. *How can different XQuery engines be united to jointly evaluate a single query?*

   A single XQuery query could easily involve multiple remote XML documents served by peers that are possibly capable of processing XQuery queries. In general, the best way to handle such queries is to to exploit the query processing power on the remote peers, i.e., executing subexpressions of a query on remote peers close to data sources. An open protocol is a prerequisite for heterogeneous XQuery engines to communicate with each other.

3. *How are distributed updating queries supported?*

   With the introduction of XQUF, XQuery is no longer a read-only language. Since XRPC is designed to be an orthogonal extension of XQuery, it also allows updating expressions to be executed on remote peers. This requires a clear definition of the semantics of distributed updating queries (e.g., updates on remote documents), which isolation levels are supported, and how distributed transactions are supported.

4. *How can we automatically decompose XQuery queries for distributed execution?*

   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 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. For this reason, automatic XQuery decomposition must determine which subexpressions *can* be decomposed in order to guarantee the correctness of the decomposed queries.

5. *How can we integrate existing DBMS with P2P overlay networks to provide non-trivial data management facilities to P2P applications?*

   Both DBMS and P2P networks are mature research fields, thus, instead of inventing a P2P DBMS from scratch, our strategy for advancing the state-of-the-art in distributed DBMS is to research how to couple existing XDBMS with Distributed Hash Table based P2P overlay networks: which information should the underlying DHT overlays provide to the XDBMS, and how should this information be exposed? How can the DHT overlays benefit from the data management and query processing features supported by the XDBMS, to offer a finer grained data sharing feature than file-based data sharing, and more powerful searching facilities than keyword based search?

1.3 Thesis Outline

This thesis is further organised as follows. We start with a discussion of related work in Chapter 2.

In Chapter 3, we introduce the XRPC language extension, which adds a Remote Procedure Call (RPC) mechanism to XQuery. First, we specify the XRPC syntax and the SOAP XRPC network communication protocol. Then, we spend considerable time in rigorously defining the formal semantics of read-only as well as updating XRPC calls. Finally, we discuss the implementation of XRPC in MonetDB/XQuery, including the correspondence of Bulk RPC with the loop-lifting technique applied by the pathfinder compiler. This chapter addresses the research questions 1 and 2, and is based on the following paper:

- Y. Zhang, P. A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, September 2007.

In Chapter 4, we discuss various uses of XRPC for distributed XQuery processing on heterogeneous XQuery engines. First, by using Saxon, we demonstrate how XRPC can be used already with *any* XQuery system, using an XRPC wrapper that is capable of translating Bulk RPC requests into XQuery. We also show how XRPC can be used to elegantly express various distributed query processing strategies, including experiments in which MonetDB/XQuery and Saxon work together over XRPC, using e.g. the distributed semi-join strategy. Then, we turn our attention to the interaction between XRPC and XQUF. We first define a deterministic distributed update semantics and show that a small extension to the SOAP XRPC protocol enables the protocol to conform to the deterministic update semantics. We then describe how the industry standard Web Service Atomic Transaction[55] could be adapted to support atomic distributed commits of XQUF queries on heterogeneous XQuery engines. Finally, we discuss our first step towards integrating an XDBMS and DHT-based overlays in MonetDB/XQuery*. While XRPC already allows performing P2P queries, it still misses a number of vital P2P functionalities (robust connectivity, peer and resource discovery, approximate query/transaction processing). In Section 4.6, we propose two different ways to couple an XDBMS with DHTs, i.e., a loose-coupling and a tight-coupling. This chapter addresses the research questions 2, 3 and 5, and is based on the following papers (in the order of their appearance):

- Y. Zhang, P. A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, September 2007.

- Y. Zhang, P. A. Boncz. Loop-lifted XQuery RPC with deterministic updates. *Technical Report INS-E0607*, CWI, Amsterdam, The Netherlands, November 2006.

- Y. Zhang, P. A. Boncz. Distributed XQuery and Updates Processing with Heterogeneous XQuery Engines. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, (Demo Paper), June 2008.

- Y. Zhang, P. A. Boncz. Integrating XQuery and P2P in MonetDB/XQuery* *In Proceedings of the 1st Workshop on Emerging Research Opportunities for Web Data Management (EROW)*, volume 229 of *CEUR Workshop Proceedings*. CEURWS.org, Barcelona, Spain, January 2007.

In Chapter 5, we present an XQuery decomposition framework and three decomposition algorithms that automatically decompose any XQuery query into subqueries for remote execution under different parameter passing semantics. We start with identifying the semantic differences of remote XQuery pass-by-value function evaluation with respect to standard, local function evaluation. We then describe an XQuery Core based query decomposition framework. This leads in to a conservative XQuery decomposition strategy that avoids semantic problems simply by refraining from decomposition in all problem cases. To make our rewrites more effective and robust against syntactic variation, we also describe normalisation and code motion rewrite strategies. To broaden the possibility of query distribution, we

extend the pass-by-value semantics with a new *pass-by-fragment* message format that conserves more structural relationships between nodes passed in a message, and allows more predicates to be distributed. The pass-by-fragment semantics is subsequently refined to a *pass-by-projection* semantics by means of a novel *runtime XML projection* technique, which we use to generate messages that conserve all needed structural relationships between transferred XML nodes, and thus allows even more freedom in query decomposition. As a runtime technique, it is able to prune XML data much more than previously described compile-time projections[52, 31, 111]. Then, we discuss how updating queries can be handled, both in the normal XQUF semantics, as well as under an extension in which we allow non-local documents to be updated. Finally, we give an evaluation of the performance benefits of our techniques in the context of MonetDB/XQuery. This chapter addresses the research question 3 and it is based on the following papers:

- Y. Zhang, N. Tang, P. A. Boncz. Efficient Distribution of Full-Fledged XQuery. *In Proceedings of the 25th International Conference on Data Engineering (ICDE)*, pages 565-576. IEEE, ShangHai, China, April 2009.

- Y. Zhang, N. Tang, P. A. Boncz. Projective Distribution of XQuery with Updates. *IEEE Transactions on Knowledge and Data Engineering*, 2010. To appear.[4]

In Chapter 6, we formally prove the correctness of the three decomposition algorithms proposed in Chapter 5. We first prove, for each algorithm, that executing allowed subexpressions of a query remotely over XRPC will produce the same results as the original query, under the XQuery deep-equal semantics. Then, we prove the correctness of the algorithms on XCore queries containing XQUF expressions, and the correctness of the distributed code motion technique.

In Chapter 7, we illustrate how the described P2P XDBMS in Chapter 4 can be used in StreetTiVo. StreetTiVo is a demo application for multimedia meta-data for the so-called Home Theatre PCs. We first describe the current system architecture of StreetTiVo, and its major components, beside XRPC, ASR and PF/Tijah. This first version of StreetTiVo was chosen to be simple, so that we could quickly demonstrate the cooperation between XRPC, ASR, PF/Tijah in a non-trivial application setting, thus a distributed architecture is adopted, in which all StreetTiVo users (i.e., clients) are managed by a central server. Then, we explain the envisioned P2P model and discuss the challenges on our way to make StreetTiVo a truly P2P application. This chapter presents some preliminary ideas to address the research question 4, and is based on the following paper:

- Y. Zhang, A.P. de Vries, P.A. Boncz, D. Hiemstra, and R. Ordelman. StreetTiVo: Using a P2P XML Database System to Manage Multimedia Data in Your Living Room. *In Proceedings of the Joint International Conferences on Advances in Data and Web Management (APWeb/WAIM)*, volume 5446 of *Lecture Notes in Computer Science*, pages 404-415. Springer, SuZhou, China, April 2009

Finally, we conclude the thesis and discuss future work in Chapter 8.

[4]The extended version of the ICDE 2009 Conference paper accepted by the TKDE for its Special Issue on the Best Papers of ICDE2009, scheduled for publication in 2010.

2
Related Work

This Ph.D. work is related to a large body of research works in the area of query processing, query optimisation and transaction management. For more literature, we would like to refer the readers to the surveys [178] and [114]. The books [57] and [135] give comprehensive overviews of techniques involved in all aspects of distributed DBMSs. In this chapter, we first discuss in Section 2.1 related research work in Peer Data Management Systems (PDMSs), with focus on distributed XML querying. Then, we discuss in Section 2.2 two particular techniques that are often used for efficient distributed (XML) query processing, namely XML document filtering and query decomposition.

2.1 P2P Data Management Systems

The tremendous size of the Internet and the popular use of P2P applications have presented new challenges to distributed DBMS researchers. Several visionary papers and surveys have been published that identify new issues in data management in large-scale heterogeneous P2P systems, and introduce possibilities or survey the state of the art on PDMS.

In [86], Gribble et al. raise the question how data management can be applied to P2P and what the database community can learn from it and how they can contribute. The authors discuss the challenges of data placement in P2P environments and introduce the Piazza system [103], a peer data management system that enables sharing heterogeneous relational data using schema mapping. In [33], Bernstein et al. introduce the Local Relational Model (LRM) to also address issues in data management in P2P environments. LRM assumes that the data residing in different databases have semantic inter-dependencies, and allows peers to specify coordination formulas that explain how the data in one peer must relate to data in an acquaintance. Thus, LRM does not require a global schema. Sung et al. give a nice survey of data management in P2P systems in [168]. This paper gives a comprehensive overview of existing unstructured and structured P2P network systems. It also discusses the data integration issues that P2P systems must deal with when sharing structured data, query processing techniques in both P2P file sharing and data sharing systems, and data consistency issues that the cache manager and update manager must address when data duplication is supported. Bonifati et al. [44] carefully compare the characteristics of P2P databases with those of distributed, federated and multi-databases to gain better insight to the P2P data management technology. The authors provide a taxonomy of the most prominent research efforts toward the realisation of full-fledged P2P DBMSs for relational data. [99] is another extensive survey of the state of the art of PDMS. The authors examine open research directions in several areas of PDMS,

including system model, semantics, query planning schemes and maintenance.

Among the material published, [113] is a survey dedicated to distributed processing of XML data in P2P systems. It focuses on data management issues including indexing, clustering, replication and query processing and routing. In [113], P2P DBMSs are classified based on *(i)* the degree of decentralisation, *(ii)* the topology of the overlay network, *(iii)* the way information is distributed among the nodes, and *(vi)* the type of data they store.

In the remainder of this section, we first describe other XQuery extensions that equip XQuery with a query-shipping model (Section 2.1.1). We then discuss related work on distributed XML query processing in P2P systems. Proposals are grouped by the topology of their underlying overlay networks, i.e., structured P2P systems (Section 2.1.2) and unstructured P2P systems (Section 2.1.3). Finally, in Section 2.1.4, we discuss some prominent work of PDMSs of relational data.

2.1.1 Extending XQuery with Query-Shipping

XQueryD XQueryD [144, 29] extends XQuery with a new statement "`execute at`", which supports remote execution of free-form XQuery queries (specified in the "`xquery { ...}`" block following an "`execute at`" statement). XQueryD uses a runtime rewriter to scan the XQuery expressions in the `xquery` block for variables and to substitute them with the runtime values. XQueryD has two main features that distinguish it from other XQuery extensions.

XQueryD is the only XQuery (query-shipping) extension that has a mechanism to explicitly catch and handle exceptions. For each `xquery` statement, one can define multiple `handle` clauses to specify how to react to certain exceptions. The necessity of such an error handling mechanism in XQuery has been confirmed by the recently published W3C Working Draft XQuery 1.1 [149], which proposes to add `try...catch` expressions to the XQuery language.

The primary goal of the XQueryD project is to develop tools to help neuroscientists understand language organisation in the brain. In this context, data sources using totally different data models must be integrated. Therefore, XQueryD also allows an "`execute at`" statement to be used to invoke queries in a foreign language, other than XQuery.

Galax XButler and the Yoo-Hoo! Service Galax Yoo-Hoo! [72] is a user-presence service that uses XButler [134] to integrated user-presence information from multiple service providers and to provide Yoo-Hoo! clients with information such as "the requested subscriber is on-line at Jabber" or "the subscriber's cell phone is busy". XButler extends XQuery with a new statement "`import service`" to import WSDL [62] modules, which enables access to Web services from within XQuery queries:

```
import service namespace foo="http://example.org" name "UserProfile";
let $c := foo:getContact("user1", 4) return ($c/name, $c/tel)
```

In the above query, the call to `foo:getContact()` will trigger a SOAP call to the appropriate Web service. Parameter values and results are passed between the caller and the callee using SOAP RPC messages. Thus, once a service module is imported, all operations defined in this service module could be called in an XQuery query, as if they were normal XQuery functions.

When a WSDL module is imported, each operation defined in this module is compiled into an XQuery stub function with the same name and parameters as the WSDL operation. The stub function takes care of generating SOAP RPC request messages with the actual parameter values, exchanging messages with the Web service, and extracting results from the SOAP RPC response message. XButler also provides a tool `xquery2soap` to deploy a SOAP service

from a given XQuery module. In [134], a binding between XQuery and WSDL is given which is based on the relationship between XQuery `modules` and WSDL `portTypes`.

Related to XButler is the work proposed by Fourny et al. in [78, 77], which also adds a Web service importing feature to XQuery to allow (asynchronous) accessing of Web services. In [78, 77], the authors have extended JavaScript to allow execution of XQuery expressions (including updating expressions) in Web browsers. Such an extension could ease the development of AJAX-style applications, because programming the browsers involves mostly XML navigation and manipulation, and the XQuery family of W3C standards were designed exactly for this purpose.

DXQ DXQ [74, 70] is a distributed XQuery processing framework developed as an extension of Galax. The basic idea of DXQ is that functions in an imported module can be executed on an arbitrary number of remote DXQ servers. DXQ allows remote updates, by means of supporting XQueryP [59]. A small change in module importing is that DXQ distinguishes a module's interface from its implementation, thus, in DXQ, multiple servers can export the same module interface, but each provides its own, potentially unique implementation. Next to synchronous remote executions, DXQ also provides an asynchronous execution model. This useful feature opens more opportunities for (distributed) query optimisation.

A major difference between DXQ and other XQuery extensions is that DXQ depends on distributed query plans, in terms of the internal Galax execution algebra, generated by the Galax optimiser. This has certain advantages, such as better control over the capabilities of the distributed nodes and possibly better physical plans and optimisation, but the use of an internal algebra makes it hard to achieve cross-system DXQ.

2.1.2 Distributed XML Querying in Structured P2P Systems

We start with a detailed discussion of the Active XML project and its related KadoP system in a dedicated subsection, because it is by far the most important related work in distributed processing of XML data that has achieved notable results [28, 19, 20, 165, 12, 18, 5, 9, 36, 154, 11, 16, 37, 15, 14, 4, 6, 8, 13, 30, 155, 10, 3, 127, 138, 7].

2.1.2.1 Active XML and KadoP

Active XML Active XML (for short: AXML) is a declarative framework that uses Web services (WSDL) for distributed XML data management. The framework is based on *AXML documents*, which are XML documents that may contain embedded calls to Web services; and *AXML services*, which are Web services capable of exchanging AXML documents. An *AXML peer* is a repository of AXML documents that can act both as client when invoking the embedded service calls, and as server when providing AXML services. A comprehensive overview of AXML features can be found in [9]. In [18, 20], the authors give a formal analysis of the behaviour of AXML systems, in particular, to verify the temporal properties of AXML documents. In this section, we highlight several AXML aspects that are closely related to our work.

A service call in an AXML document is denoted using a special `axml:call` element, which has one `service` attribute to specify the service to be invoked. An `axml:call` element can also contain additional attributes to specify, for instance, the `mode`, in which the call is activated, and for how long its results are `valid`. Parameter values of the called services are included as the descendants of the `call` element [9, 30]. In an AXML document, `axml:call`

elements are allowed to appear anywhere in the document. This is not a problem for XML documents without schemas, as AXML documents are syntactically valid XML documents. However, adding `axml:call` elements to an XML document with a schema would either invalidate the original document, or require the schema to be modified.

After a service call has been invoked, its results are materialised, i.e., added into the document. The service call could either be replaced by its results, or be kept next to its results for re-evaluation, which enables data refreshing and nested calls. Service functions are defined in AXML using an XML query language X-OQL[22], in the open-source implementation [28], which itself does not allow distributed evaluation (an embedding piece of AXML is always needed). The SOAP protocol used for AXML uses a document/literal encoding to represent XML subtree values. However, it has not been specified formally.

AXML has shown the value of distributed query optimisation, identifying lazy evaluation schemes and various rewrite strategies [6]. AXML has also shown its value in many different areas of distributed XML processing, including mobile systems [138], data warehousing [13, 14, 5], event systems [4], and workflow systems [164, 165].

KadoP An important application of AXML in P2P systems is KadoP [13, 14], a distributed infrastructure for warehousing XML resources in a P2P framework. KadoP builds on DHTs as a peer communication layer and AXML as a model for constructing and querying the resources in the network. A KadoP query is a tree pattern whose nodes represent data items and whose edges represent containment relationships among the nodes. Both nodes and edges are indexed in the underlying DHT network. The usefulness of KadoP has been demonstrated by the EDOS system [11, 17] and the WebContent platform [5]. EDOS is a distribution system for efficient dissemination of open-source software through the Internet using a publish-subscribe infrastructure. WebContent is a platform for managing distributed repositories of XML and semantic Web data. In [12], the authors describe techniques to address some of the scalability limitations of KadoP. The authors introduce a technique based on partitioning and distributing index blocks to greatly reduce query response time. Structural Bloom Filters are used to reduce network traffic by filtering out peers that surely do not have certain XML nodes needed by a query.

2.1.2.2 XPath Query Processing over DHTs

A considerable number of proposals on supporting XPath queries over DHT overlays already exist, e.g., Galanis et al. [79], XP2P [46, 45], XPeer [142], Skobeltsyn et al. [161] and XCube [120]. These proposals are similar to each other, in the sense that they all focus on efficient processing of the '/' and '//' XPath steps, with minor differences in additionally supported XPath language features. The work proposed by [79, 46, 142, 120] support selection predicates, and the work proposed by [142, 161] support the wildcard '$*$'. In addition, XCube supports tag-based queries, e.g., `(title, "Database")`. These proposals differ mainly in the used indexing techniques, which, in turn, affect the way queries a processed. To the best of our knowledge, there is no existing system that supports full-fledged XQuery over DHTs.

Galanis et al. In [79], XML tag names are used as hash keys. The authors try to address the scalability issue in large-scale P2P systems by sending queries *only* to the repositories that have data relevant to the query without relying on a centralised catalog infrastructure. Therefore, the authors propose a catalog framework that is distributed across the data sources themselves. A fully decentralised catalog service allows data providers to join and make their

data query-able by all peers in the systems. To balance query workload, catalog information is split or replicated dynamically.

XP2P XP2P [46, 45] is built on top of Chord [162]. The system assumes that each peer stores a set of XML fragments. Each fragment is unambiguously identified by its *distinct linear absolute path*. The search key of a fragment is the hash value of its path. To compute the hash values, a special fingerprinting technique is used which produces shorter hash keys (than those produced by Chord) and supports a concatenation property that allows the computation of the tokens associated with path expressions to proceed incrementally. XPath queries that only contain child steps (potentially with positional predicates) can be answered in XP2P extremely fast, since the actual query can serve as the search key in the DHT network. For `descendant` steps, a separate algorithm is presented that achieves a $O(N_f \times log(N))$ complexity, where N_f is the number of fragments and N the number of peers in the network.

XPeer XPeer [142][1] maintains indices at three different levels of granularity, i.e., DHT hashes of complete XML documents, XPath expressions, and XML elements. Although these indices might accelerate query execution, the authors did not discuss the costs of creating and maintaining the indices.

Skobeltsyn et al. Similar to XP2P, Skobeltsyn et al. [161] also index XPath steps containing only the child axis '/'. The P-Grid [1, 2] DHT overlay network is used, which uses a binary trie topology. The hash key of a query is the longest sequence of element tags in the query that are only divided by '/'. If only one peer is responsibly for the hash key, the query result can be computed. Otherwise, a *shower algorithm* is used to broadcast the query to *all* peers in the subtrie defined by the hash key. To overcome the high costs imposed by large broadcasts (when the hash key is short), intermediate results are cached.

XCube XCube [120] is a tag-based scheme that manages XML data in a hyperCube overlay network to support XPath and tag-based queries. An advantage of tag-based queries is that they do not require users to know the structure of a document before querying the document. Like XPeer [142], documents in XCube are indexed at different levels of granularity. That is, an XML document is compactly represented as a triple: a bit vector derived from the distinct tag names in the document, a synopsis of the document and a bit map of the content summary. A query is processed in four phases. First, the bit vector derived from the query tags is used to locate the query's anchor peer, which contains a superset of the synopses of all potentially matching answers. Second, the query is compared against all synopses at its anchor peer and forwarded to the anchor peer of each document with matching synopsis according to its bit vector. Third, at the anchor peer of a document, the predicates in the query are examined based on the bit maps stored on this peer. Documents that satisfy the structural requirements but not the predicates in the query are pruned. Finally, the query is forwarded to all owner peers in the answer set for evaluation. Unlike previous approaches, e.g., [79, 46, 45, 142, 161], XCube does not put any limitations on the supported XPath expressions. Regretfully, the experiment results do not show how well XCube would perform when processing XPath steps on reverse and horizontal axes.

[1] This XPeer project should not be confused with another XPeer project proposed by Sartiani et al. [156], which supports the FLWOR expressions of XQuery on top of a superpeer network.

2.1.3 Distributed XML Querying in Unstructured P2P Systems

Mutant Query Plans Similar to AXML, query execution in Mutant Query Plans (MQPs) [136, 137] also happens by exchanging XML documents containing both resulting XML data and unevaluated subqueries, and it is also independent of any central coordinators. However, exchanging (system specific) query plans would suffer from the same interoperability problem as we have pointed out in DXQ. In an XML query language, the usual way to reference data source is to use their URLs. An MQP is a query plan graph serialized in XML that, in addition to URLs, can refer to abstract resources using URNs and include verbatim XML data. In a system using MQP, each peer maintains a local catalog that maps each URN to either a URL, or to a set of peers that know more about this URN. When a peer receives an MQP, it first resolves all URNs in the MQP it knows about. Then, the peer (re)optimises the plan and creates subplans that can be evaluated locally, with associated costs. Next, the peer's policy manager will decide to accept or reject the mutant plans (e.g., costs are too high), and how much of a (sub)plan can be evaluated locally. Finally, the peer substitutes each evaluated subplan with its results, as an XML fragment, to get a new *mutated* query plan. If the plan is not yet fully evaluated, the peer chooses the next peer to forward the plan to, by consulting its local catalog. Otherwise, the plan is the final result of the query (in the form of an XML document, without any URNs) and it is forwarded to the query's destination, which may be different from its origin.

XPeer XPeer [156][2] is a P2P XDBMS for sharing and querying XML data on top of a superpeer network. Peers export a tree-shaped DataGuide description of their data that is automatically inferred by a tree search algorithm. The query language supported is the FLWR subset of XQuery without universally quantified predicates and sorting operations. Query compilation is performed in two phases by the superpeers. First, the peer that issues the query translates it into a location-free algebraic expression. Then, the query is sent to the superpeer network for the computation of a location assignment. After the location assignment is completed, the query is sent back to the peer that issued it for execution to minimize the load of the superpeer network. The peer applies common algebraic rewriting and then starts query execution: the query is split into single-location subqueries that are sent to the corresponding peers. Subqueries are locally optimized and the results are returned to the initial peer, which executes operations such as joins involving multiple sources. The query algebra of XPeer takes data dissemination, data replication and data freshness explicitly into account.

HePToX HePToX [43, 42] is a heterogeneous P2P XDBMS that supports a subset of XQuery. A key idea is that whenever a peer enters the system, it provides a mapping between its schema and a small number of the existing peer schemas. The peers chosen by the entering peer are called its *acquaintances*. Although (semi-)automatic schema mapping tools could be used to provide the mappings, HePToX also allows a peer database administrator to supply simple correspondences between the peer's schema and the schemas of its acquaintances. The correspondences are used as a basis for automatically inferring a *mapping expression*, expressed in the form of Datalog-like rules. HePToX implements a more expressive extension of Global-Local-as-View (GLaV) mappings, called *data schema interplay*, where mappings exploit correspondences between attribute *values* and *names* of schema elements.

[2] This XPeer project should not be confused with another XPeer project proposed by Rao et al. [142], which is an XML-based content query system built on DHT systems.

Piazza Piazza [86, 95, 94, 93, 171, 103] is a well-known PDMS that enables sharing and integration of heterogeneous data. It can handle the mapping of both relational data [95, 171] and XML data [103]. Piazza assumes that participating peers are willing to share their data and define pairwise mappings between their schemas. In [103] Tatarinov et al. describe several methods for optimising schema-based reformulation of XML queries. In [103], data is represented in XML, peers schemas in XML Schema, and mappings are described as query expressions using a subset of XQuery. Peers are considered as connected through semantic paths of such mappings. Peers may store mappings, data or both. Instead of a local index, each peer maintains mappings between its own schema and the schemas of its immediate neighbours. Query evaluation is incremental with an additional logical-level search where data are located based on schema-to-schema mappings. Query processing starts at the issuing peer and is reformulated over its immediate neighbours, which, in turn, reformulate the query over their immediate neighbours and so on. Whenever the reformulation reaches a peer that stores data, the appropriate query is posed on that peer, and additional results may be appended to the query result. Various optimisations are considered regarding the query reformulation process such as pruning semantic paths based on XML query containment, minimizing reformulations and pre-computing some of the semantic paths.

Bremer et al. [51] introduces a distribution approach for a virtual XML repository. XML data are fragmented based on a *global conceptual schema* and allocated among peers. Fragments allocation is done using existing allocation models for relational databases [26, 135]. The information of fragments allocation, i.e., which fragments are allocated at which peer(s), is kept in a *global context*. Query processing is done by shipping index entries among nodes and evaluating chains of local joins of indices. By using the global context, the peer, on which a query is started, can compute the remote peers that contain fragments needed by the query.

Koloniari et al. In [112], the authors propose a content-based approach to route XPath queries in a hierarchical P2P network. In this network, peers with similar content are clustered together. Each peer maintains two types of filters: a *local filter* summarising the documents stored locally at the peer, and one or more *merged filters* summarising the documents of the peer's neighbours. These filters are used to route a query only to those peers that may contain relevant documents. Two multi-level Bloom filters are proposed for summarising hierarchical data which exploit the structure of data. Like many proposals for processing XPath queries over DHTs discussed in Section 2.1.2, the work in [112] only supports the `child` and `descendant` XPath steps. The proposed approach is more effective for simple linear XPath expressions, but not precise for finding answers for descendant axes. [112] is one of the few approaches that considers updates. When a document is updated, inserted or deleted at a peer, the peer updates its local filter and propagates the updates to merged filters on other peers that use this local filter. The propagation algorithm ensures that only the changed parts of the multi-level filter are transferred, not the whole filter.

Distributed Evaluation of Semistructured Data [167] is one of the first works about evaluation of path expressions on distributed (though non-P2P) semistructured data. In this work, semistructured data is modeled as a rooted, labeled graph, and queries are regular path expressions with complex data restructuring and subqueries. Distribution is implemented by having links from the local XML data to XML objects at remote peers. The model distinguishes between local links that point to local objects and cross-links that point to remote objects. Every peer determines which of its data have incoming edges from other peers (input data

nodes) and which have outgoing edges to remote objects (output data nodes). Given a query, an automaton is computed and sent to every node. Each node traverses only its local graph starting at every input data node and with all states in the automaton. When the traversal reaches an output data node, it constructs a new output data node with the given state. Similarly, new input data nodes are also constructed. Once the result fragments, which consist of an accessibility graph that has the input and output data nodes and edges between them, are computed they are sent to the origin of the query. The originating peer of the query assembles these fragments by adding missing cross-links, and computes all data nodes accessible from the root. The algorithms guarantee that the size of the data exchanged depends only on the number of cross links and the size of the query answer.

2.1.4 PDMSs of Relational Data

Much research work has been done on querying relational data in P2P settings. In this section, we will discuss some prominent work in this area.

UniStore UniStore [107, 108] is a triple storage system built on top of the P-Grid [2] DHT overlay, which is based on the ideas of a Universal Relational Model and the Resource Description Framework (RDF). It proposes a structured query language Vertical Query Language (VQL), which is derived from SPARQL [139]. Query plans are processed in a similar way as the Mutant Query Plans [136, 137].

Hyperion Hyperion [27, 150, 186] is a PDMS built on top of its own unstructured P2P network, which supports SQL queries over heterogeneous relational data. Like UniStore, Hyperion avoids the need of a global schema by defining mappings between acquaintances. However, unlike UniStore (which requires that acquaintances must be defined at the time a peer enters the system), acquaintances are formed dynamically at runtime, and it does not use a database administrator. Mappings are defined in so-called *mapping tables* which specify relations between values of attributes of data records residing on different peers rather than on schema elements. Distributed Event-Condition-Action (ECA) rules are used to enable and coordinate data sharing.

PeerDB PeerDB [132] is built on top of the BestPeer network [131], a two layer hierarchical network that integrates mobile agents. Thus, PeerDB adopts mobile agents to assist in query processing. On each peer, a MySQL database is used to manage data. PeerDB proposes a mechanism to share data of similar but different schemas. It uses a simple data integration algorithm with some user interference and it relies on a central directory server.

The APPA System The APPA (Atlas Peer-to-Peer Architecture) system [23, 126, 174] provides high-level data sharing services in large-scale distributed environments by combining Grid and P2P technologies. The architecture of APPA consists of three layers. The *P2P Network* layer provides network independence with services that are common to different P2P networks. Thus, APPA can combine different P2P networks (e.g., JXTA, Chord and CAN) to exploit their relative advantages. The *Basic Services* layer provides elementary services, e.g., persistent data management, communication cost management and group membership management. The *Advanced Services* layer provides advanced services for semantically rich data sharing including schema management, replication, query processing, security, etc., using the basic services.

System P In System P [152, 151], query plans are computed decentralised, locally at the peers. Based on the given Local-as-View (LaV) and Global-as-View (GaV) peer mappings, a

local rule-goal tree is created at the peer receiving the original query, as well as at every peer that is contacted during query processing. System P balances the completeness of the query result and execution cost by pruning the query plan at mappings that are estimated to yield only few result tuples. For query execution, the authors propose a budget driven approach, where peers are assigned a budget to use for query answering. This is similar to the economic execution model of Mariposa [163].

2.2 Related Query Processing Techniques

In this section, we discuss related work in two specific techniques for efficient distributed (XML) query processing, namely XML document filtering and query decomposition.

2.2.1 XML Document Filtering

XML document projection or filtering is a technique popularly used by both streaming systems and main memory XQuery processors to drastically reduce the size of the data model representation, which in turn can accelerate query processing.

Marian et al. [125] originally introduce this concept and propose a static analysis algorithm to compute, at compile time, for a given XQuery query the set of projection paths which include a set of used paths and a set of returned paths. Before the query is evaluated, a load algorithm uses the projection paths to compute projected documents, which could be much smaller than the original documents. The query is then applied on the projected documents instead. This compile-time XML projection technique is used to address memory limitations in main memory XQuery processors. Our runtime XML projection technique [183, 184] (Chapter 5) is an extension of the compile-time XML projection. Instead of only supporting XPath steps on downward axes (e.g, `self`, `child` and `descendant`), the runtime technique supports XPath steps on *all* axes and the built-in functions `fn:root()`, `fn:id()`, `fn:idref()` and `fn:lang()` which need to access nodes outside the subtree of their node parameters. Comparing with the compile-time projection technique, runtime projection is much more accurate, because predicates on XPath steps are processed before the projection. In such cases, the resulting projected documents of runtime projection are usually much smaller than the results of compile-time projection.

Much research work has been done on filtering XML documents in streaming systems for efficient query processing [24, 60, 66, 52, 31, 56, 83, 61, 111]. We will take a look at several of these approaches. XFilter [24] aims at efficient matching of XML documents to large numbers of user profiles that are expressed as XPath queries. It is the first approach that uses a Finite State Machine for each XPath query to quickly locate relevant profiles when an XML document arrives. Based on XFilter, Diao et al. propose YFilter [66] that combines all XPath queries into a single Nondeterministic Finite Automaton. Bressan et al. [52] introduce a precise XML pruning technique for a subset of XQuery FLWOR expressions, based on the *a priori* knowledge of a data guide for underlying XML data. However, it does not handle XPath predicates, backward axes and XQuery-like languages. A type-based XML projection technique [31] is studied to improve current solutions with comparable or higher precision and less pruning overhead, and supporting backward XPath axes. This technique is only applicable for XML documents that have a DTD. Koch et al. [111] also propose runtime XML projection techniques. Based on the static compilation of runtime lookup-tables and a runtime automaton from projection paths and a DTD, an input XML document can be filtered efficiently using

string matching algorithms. This technique improves efficiency, but still lacks the power of supporting reverse XPath axes and XQuery built-in functions. This reflects a common limitation of the XML filtering techniques: they only support efficient processing of subsets of XPath.

2.2.2 Query Decomposition

Decomposing queries to address multiple data sources has been applied in a large variety of research areas, including relational databases [175, 106], object-oriented databases [34, 105, 115], distributed databases [98, 119], multi-databases [117], heterogeneous distributed databases [173, 123], P2P data management systems [33], and semi-structured databases [166, 167]. Decomposition techniques have been proposed based on ontologies [177], topics (i.e., Topical Query Decomposition) [40, 169], and (hyper-)trees [76, 157, 82].

Proposals exist that study decomposing XML queries, but none of them address the problems that revolve around XML node identity and structural (rather than value-based) relationships between nodes when queries are decomposed and distributed automatically. In [166, 167], the author discusses the decomposition of unstructured query languages on a semi-structured database (a rooted, labeled graph). In [118], Le et al. propose a bottom-up approach for distributed XDBMSs using Global-As-View to transform a global XPath expression into local XPath expressions executable in local schemas. This approach requires structural information about peers to supervise decomposition. In [160], Silveira et al. present a query decomposition mechanism that allows a query stated at the conceptual level, using CXPath (a Conceptual XPath language defined by the authors), to be decomposed into an XQuery statement at the XML level. Other works in distributed XML query evaluation, such as [53, 63, 170], only focus on a restricted set of XQuery/XPath queries, and do not address the problem of transparent query decomposition, such that these challenges do not play a role.

For instance, in [170], the authors consider optimizing the cost of communication in answering XPath queries over distributed data based on the client-server model. Minimal views that contain results of a single query or a set of queries are used to avoid the redundancy met in such results where the same data may appear many times. The system leaves part of the evaluation of the query to the client that may have to extract all the answers from the minimal view to obtain the results to the initial queries. At a receiving peer (i.e., the client side), the system ensures that only downwards XPath steps will be applied on the received data, avoiding problems around XML node identities and structural relationships between nodes that can be caused by executing non-downward XPath steps (e.g., `ancestor` and `preceding`) on the received data.

2.3 Conclusion

From the related research work, we can conclude that the topic of distributed XML querying has attracted quite some research interest. Within this area, two major approaches have been studied: *i)* extending the XQuery standard with a query shipping model, and *ii)* accelerating of XPath queries on distributed XML documents. XQuery/XPath has been generally accepted as the language to query XML data. DHT is a popularly used mechanism to manage underlying networks, because DHT networks have several properties (e.g., $O(logN)$ scalability) that make them particularly suitable for P2P settings. However, plenty of issues are still left open.

This Ph.D. research addresses three open issues. First of all, *interoperability* is an almost untouched topic by any of the existing proposals, while it is a main issue in P2P settings, where peers are highly heterogeneous. Although XButler uses the standard SOAP RPC protocol as its communication protocol, due to the limitations of SOAP RPC, it is restricted to only support XQuery functions with atomic value parameters and results. This in turn reduces the interoperability of XButler. Secondly, supporting full-fledged XQuery is an open issue. Most of the existing work only focuses on efficient distributed processing of small subsets of XQuery/XPath. This greatly reduces the expressive power of the XQuery language and also the interoperability of the proposed techniques. Moreover, with XQUF [58], XQuery is no longer a read-only language. This raises several questions, such as what are the semantics of remote updates, how can distributed transactions be supported, which consistency level should be provided and how can it be integrated into the language. Thirdly, the issue of dealing with challenges imposed by the XML data model is unaddressed. Query decomposition is an often used mechanism in distributed query processing. However, the existing techniques were developed for relational data, which are only value based, while in the XML world, XML nodes have node identities and structural properties. It has not been studied if and how the existing techniques can be applied to the XML data model, while respecting the semantics of XML data.

3 The XRPC Language Extension

In this chapter, we introduce the XRPC language extension. We start with a discussion of the design criteria that the XRPC extension must satisfy. Then, we give the definition of XRPC syntax in Section 3.2, and the SOAP XRPC message format in Section 3.3. In Section 3.4, we spend considerable time in rigorously defining the formal semantics of XRPC. Finally, in Section 3.5, we outline the initial implementation of XRPC in MonetDB/XQuery, including the correspondence of Bulk RPC with the loop-lifting technique applied by the pathfinder compiler.

3.1 Design Considerations

The XRPC language extension must satisfy the following design criteria:

- The extension *must* be orthogonal to all XQuery features, including XQUF.
- The extension *must* support all XDM data types.
- The extension *must* be unambiguous, i.e., if a query containing our extension is run on systems that do not support this extension, the query must not produce unexpected results.
- The extension *must* allow functions in the same module to be executed both locally and remotely, and it *must* also allow functions from different modules to be executed remotely.
- The extension *should* be clean, i.e., only require minimal changes to the XQuery standard.
- The extension *should* be well-defined and have easy to understand semantics.
- The extension *should* provide potentials for efficiency and scalability.
- The extension *should* be easy to support by heterogeneous XQuery engines.

Besides the existing proposals [74, 134, 144, 75], we have considered several alternatives, which all satisfy the first two design criteria:

1. Use a special namespace prefix, e.g., `rpc`, to indicate that all functions from an imported module, bound to this special namespace prefix, will be executed remotely (for short, we call modules that contain functions that will be executed at a remote peer as *RPC modules*):

```
import module namespace rpc = "rpc-functions"
    at "http://example.org/foo.xq", "http://example.org/bar.xq";
rpc:foo("foo.example.org", rpc:bar("bar.example.org", 42))
```

This approach is easy to understand. However, its semantics is ambiguous because namespace prefixes do not have significant meaning in XQuery. Query writers could accidentally use the special namespace prefix, causing queries to return unexpected results on different XQuery systems. Moreover, functions in the imported modules could only be run either locally or remotely, as the XQuery standard [38] does not allow the same module to be loaded twice in a query: "It is a static error [err:XQST0047] if more than one module import in a Prolog specifies the same target namespace.".

This approach allows multiple different modules to be imported by listing multiple module location URLs in the `at`-hint. The drawback of this approach is that multiple modules are loaded under the same namespace, which is restrictive and can lead to clashes. To overcome this problem, one could, instead of using one special namespace prefix, use namespace prefixes that start with a predefined string, e.g., "`rpc-`", to import RPC modules:

```
import module namespace rpc-foo = "rpc-functions" at "http://example.org/foo.xq";
import module namespace rpc-bar = "rpc-functions" at "http://example.org/bar.xq";
rpc-foo:foo("foo.example.org", rpc-bar:bar("bar.example.org", 42))
```

However, this alternative is even less clean: it is even more likely that query writers would accidentally use a namespace `rpc-`$*$ that causes queries to behave differently on systems with our extension.

2. Use a special namespace, e.g., "http://www.w3.org/TR/soap/", to indicate RPC modules:

```
import module namespace rpc =  "http://www.w3.org/TR/soap/" at "http://example.org/foo.xq";
import module namespace bar =  "http://example.org/bar" at "http://example.org/bar.xq";
rpc:foo("foo.example.org", bar:bar(42))
```

Comparing with the first approach, this approach is much cleaner by assigning special semantics to a namespace, which has significant meaning in XQuery. However, this approach allows only one module with the special target namespace to be imported in a query, which is a major limitation. Similar as for the first approach, this problem might be alleviated by defining a special namespace prefix, e.g., "`http://www.w3.org/TR/soap/`", and all modules, whose target namespaces start with this prefix, are recognised as RPC modules:

```
import module namespace foo =  "http://www.w3.org/TR/soap/foo" at "http://example.org/foo.xq";
import module namespace bar =  "http://www.w3.org/TR/soap/bar" at "ttp://example.org/bar.xq";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42))
```

On the other hand, this workaround intensifies a next problem: modules declared using this special SOAP namespace (as a prefix) cannot be used as normal, local modules, unless the module definitions are duplicated using a different target namespace (the same situation exists in the first approach). This is bad for software re-use and maintenance.

3. Use the `at`-hint to indicate an RPC module:

```
import module namespace foo = "http://example.org/foo.xq" at "http://www.w3.org/TR/soap/";
import module namespace bar = "http://example.org/bar.xq" at "http://www.w3.org/TR/soap/";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42))
```

Comparing with previous approaches and their alternatives, this approach is more favorable. First of all, the semantics of this approach is completely legal under the XQuery specification, because XQuery 1.0 [38] has specified that "The `URILiteral`s that follow

the `at` keyword are optional location hints, and can be interpreted or disregarded in an *implementation-defined* way.". Moreover, this approach allows multiple RPC modules to be imported using different namespace prefixes bindings, avoiding introducing unnecessary clashes. Nevertheless, modules still cannot be imported both as local modules and as RPC modules.

The major disadvantage of this approach is the `at`-hint, whose semantics is very counter-intuitive. The original intention of the XQuery specification is to use the `at`-hint to specify the *physical* location of the module, while in this approach, the `at`-hint contains a *logical* namespace. Additionally, this approach forces modules to have the same target namespaces as their physical location, defeating the purpose of target namespaces that should be logical identifiers of modules. A workaround of the latter disadvantage would be to include the special namespace "`http://www.w3.org/TR/soap/`" in the `at`-hint as an *additional* URL, whose semantics is different than other URLs in the `at`-hint:

```
import module namespace foo = "http://example.org/foo"
    at "http://example.org/foo.xq", "http://www.w3.org/TR/soap/";
import module namespace bar = "http://example.org/bar"
    at "http://example.org/bar.xq", "http://www.w3.org/TR/soap/";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42))
```

but this mix of `at`-hint containing both physical and logical hints makes the design even more messy.

4. Extend the XQuery language with a new module importing feature:

```
import rpc-module namespace foo =  "http://example.org/foo" at "http://example.org/foo.xq";
import rpc-module namespace bar =  "http://example.org/bar" at "http://example.org/bar.xq";
import module namespace foo-loc =  "http://example.org/foo" at "http://example.org/foo.xq";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42)),
foo-loc:foo(bar:bar("bar.example.org", 42))
```

With such a language extension, the semantics of the imported modules are clear. Modules can be imported both as local modules and as RPC modules. However, this approach is a language extension, while one of our design criteria is to limit the change to the XQuery standard to a minimum.

All four approaches discussed above suffer from a common problem: they do not have an elegant, flexible way to specify the destination of the remote peer on which a function is to be executed. The signature of each function is implicitly extended with an additional leading string parameter to hold the URL of the destination peer. Such a design could be considered unclean. In the next two approaches, more attention is paid on where and how the destination URLs should be specified.

5. Put RPC calls in extension expressions and specify in the pragmas of the extension expressions which functions should be executed remotely and on which peers:

```
import module namespace foo = "http://example.org/foo" at  "http://example.org/foo.xq";
import module namespace bar = "http://example.org/bar" at "http://example.org/bar.xq";
declare namespace rpc = "http://www.w3c.org/TR/SOAP";
(# rpc:rpc-call (foo:foo, "foo1.example.org", "foo2.example.org") #)
{foo:foo(bar:bar(42),
         (# rpc:rpc-call (bar:bar, "bar1.example.org", "bar2.example.org") #)
         {bar:bar(24)})} + 10
```

As shown in the example above, we identify in a pragma the exact function that should be executed remotely, followed by a list of URLs of the remote peers. In this way, modules only need to be imported once, but can be used both as local and as RPC modules. We can specify per function, instead of per module, whether it should be executed remotely or not. At the syntax level, this approach is much more flexible than the previous approaches, since pragmas are allowed everywhere where a path expression would be allowed. Thus, certain sub-expressions of a query could live within, while other sub-expressions could exist outside the scope of a pragma. Another advantage of using a pragma is that the semantics of the extension expressions is unambiguous, since it has been specified by the XQuery standard that: "An extension expression is an expression whose semantics are implementation-defined. Typically a particular extension will be recognised by some implementations and not by others." This makes pragma an ideal place to define additional language features.

However, pragmas are often considered to be difficult to understand, and have not been generally adopted. From the above example, it can already be seen that allowing pragmas everywhere in a query will quickly make a query unreadable. Query writers are also required to have a very good understanding of the exact semantics of pragmas, since they apply to the whole expression enclosed in the curly braces behind them. For instance, in the above example, the two calls to the function `bar:bar()` should be executed on different peers (i.e., the first one is a local function call while the second one is an RPC call), this requires that the second pragma is only specified for the second `bar:bar()`; otherwise the query will have very different semantics, e.g.,:

```
(# rpc:rpc-call (foo:foo, "foo1.example.org", "foo2.example.org") #)
(# rpc:rpc-call (bar:bar, "bar1.example.org", "bar2.example.org") #)
{foo:foo(bar:bar(42), bar:bar(24))} + 10
```

6. Extend the XQuery language with a new function application syntax, i.e., `QName(...)@(URILiteral (',' URILiteral)*)`:

```
import module namespace foo = "http://example.org/foo" at "http://example.org/foo.xq";
import module namespace bar = "http://example.org/bar" at "http://example.org/bar.xq";
foo:foo( bar:bar(42)@("bar.example.org") )@("foo1.example.org", "foo2.example.org")
```

The pros and cons of this approach are similar to that of the pragma approach. It allows modules to be imported once and used both as local and as RPC modules. The syntax is very flexible, as destinations can be specified for each function, and remote function calls can be made everywhere in a query where a function call is allowed. The semantics is easy to understand and is less error-prone, as the list of destinations only applies to its associated function.

Our final choice is to introduce a language extension by adding one new statement to XQuery to allow remote function application. At the syntax level, our language extension is inspired by that of [144]. By comparing different alternatives of adding a query shipping feature to XQuery, we conclude that a language extension results in a cleaner and more flexible design than extending existing XQuery features with new semantics. We consider one new statement to be a minimal change to the XQuery standard. Later in this chapter, we discuss

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

Table 3.1: The XQuery 1.0 grammar rules extended with `XRPCCall`.

in detail all features included in our XQuery extension, and it will be clear that this extension satisfies all criteria we have defined.

The above discussion concerns language design at the calling sites. At the remote sites (i.e., receivers of RPC calls), the following questions must be considered:

- How can remote peers know which functions to execute?
- How can remote peers access the implementation of the functions it should execute?
- How can remote peers get the actual parameter values?

The answers to these questions are presented in Section 3.3, where we also argue our choice to specify our own SOAP XRPC message format for function parameter and result (un)marshalling over using the standard SOAP RPC format.

3.2 XRPC Syntax

Remote function applications take the XQuery syntax:

```
"execute" "at" {ExprSingle} {FunApp(ParamList)}
```

where `ExprSingle` is an XQuery `xs:string` expression that specifies the URI of the peer on which `FunApp` is to be executed. The function to be applied can be a built-in or a user-defined function. For user-defined functions, we currently restrict ourselves to functions defined in an XQuery Module. A small (future) extension to the network protocol would also allow functions defined inside the query to be executed over XRPC. Thus, the defining parameters of an XRPC call are: *(i)* a module URI, *(ii)* a function name, and *(iii)* the actual parameters (passed by value). The module URI is the one bound to the namespace identifier in the function application. Just like an `import module` statement, the module URI may be supplemented by a so-called `at`-hint, which also is a URI. For a precise syntax definition, Table 3.1 shows the rules of the XQuery 1.0 grammar that were changed.

The current choice to allow functions defined in XQuery modules is due to efficiency and security reasons. XQuery modules have the advantage that they may be pre-loaded and cached, and our choice to let XRPC use modules as the query transport mechanism also opens the possibility to reap performance profit from module pre-processing. The feature of *prepared queries* is well-known for an RDBMS, allowing a parametrised query plan to be parsed and optimised off-line, such that an application can quickly enter actual parameters in the prepared plan and execute it. MonetDB/XQuery has a mechanism for supporting prepared queries that does not need specific API support. Exploiting the fact that a prepared query is in essence a function with parameters, MonetDB/XQuery *caches* all query plans for (loop-lifted) function calls, for functions defined in XQuery modules. Queries that just load a module and call a function in it with constant values as parameter, are detected by a pre-parser. The pre-parser then extracts the function parameters, and feeds them into a cached query plan. In MonetDB/XQuery, queries on small data sets can be accelerated ten-fold by this mechanism [41]. For security reasons, by allowing only modules, it is trivial to specify which

modules are allowed to be executed or not. XRPC can be easily extended to support free form queries, with some extra work on preserving the efficiency and security issues.

The XRPC URI Scheme We also introduce a new URI scheme, named `xrpc` to indicate that the remote peer specified in an `xrpc` URL is able to process XRPC requests. The generic form of such URIs is:

$$\texttt{xrpc://}\langle host\rangle[\texttt{:}\,port][\texttt{/}[path]]$$

The "`xrpc://`" indicates the network protocol. The second part "$\langle host\rangle[\texttt{:}\,port]$" identifies a remote peer. The third part "$[\texttt{/}[path]]$" is an optional local path at the remote peer.

The `xrpc` URI scheme is accepted in the destination URI of `execute at`. Moreover, we have extended the built-in functions `fn:put()` and `fn:doc()` to accept the `xrpc` URI scheme in their `$uri` parameters. Given a URL `xrpc://`$\mathcal{P}$`/`$\mathcal{D}$, `fn:put()` stores the XML tree rooted at its `$node` parameter on the remote peer $\mathcal{P}$ as document $\mathcal{D}$, which possibly overwrites the existing $\mathcal{D}$. With `fn:doc()`, $\mathcal{D}$ could then be retrieved (over HTTP) from (the XRPC server on) peer $\mathcal{P}$. As we will see in Section 5.7, this extension enables supporting updates on remote documents identified by `xrpc://` URIs.

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

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

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

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

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

```
import module namespace f="films" at "http://x.example.org/film.xq";
<films> {
    execute at {"xrpc://y.example.org"} {f:filmsByActor("Sean Connery")}
} <films>                                                                    (Q3-1)
```

This example yields: ⟨films⟩⟨name⟩The Rock⟨/name⟩⟨name⟩Goldfinger⟨/name⟩⟨/films⟩.

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

```
import module namespace f="films" at "http://x.example.org/film.xq";
<films> {
    for $actor in ("Julie Andrews", "Sean Connery")
    let $dst := "xrpc://y.example.org"
    return execute at {$dst} {f:filmsByActor($actor)}
} </films>                                                                   (Q3-2)
```

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

```
import module namespace f="films" at "http://x.example.org/film.xq";
<films> {
    for $actor in ("Julie Andrews", "Sean Connery")
    for $dst in ("xrpc://y.example.org", "xrpc://z.example.org")
    return execute at {$dst} {f:filmsByActor($actor)}
} </films>                                                                   (Q3-3)
```

Complex communication patterns may be programmed with XRPC, especially if recursive functions are used. The query below executes the RPC on a set of destination peers, uniting all results, and does so by constructing a binary spanning tree of recursive RPC calls.

```
module namespace film="films";
declare function film:recursiveActor($destinations as xs:string*, $actor as xs:string) as node()
{ let $cnt := fn:count($destinations)
  let $pos := ($cnt / 2) cast as xs:integer
  let $dsts1 := fn:subsequence($destinations, 1, $pos)
  let $dsts2 := fn:subsequence($destinations, $pos+1)
  let $peer1 := $destinations[1]
  let $peer2 := $destinations[$pos]
  return
    (if ($cnt $>$ 1) then execute at {$peer1} {film:recursiveActor($dsts1, $actor)} else (),
     doc("filmDB.xml")//name[../actor=$actor],
     if ($cnt $>$ 2) then execute at {$peer2} {film:recursiveActor($dsts2, $actor)} else ())
};                                                                                    (Q3-4)
```

3.3 SOAP XRPC Message Format

The Simple Object Access Protocol (SOAP) is the XML-based message format used for web services [128, 89, 90], and we propose the use of SOAP messages over HTTP as the network protocol underlying XRPC. SOAP web service interactions usually follow an RPC (request/response) pattern, though the SOAP protocol is much richer and allows multi-hop communications, and highly configurable error handling. For the simple RPC use of SOAP over HTTP, a subprotocol called "SOAP RPC" is in common use [90]. SOAP RPC is oriented towards binding with programming languages such as C++ and Java, and specifies parameter marshalling of a certain number of simple (atomic) data types, and also allows passing *arrays* and *structs* of such data-types. However, its supported atomic data types do not match directly those of the XQuery Data Model (XDM) [71], and the support for arrays and structs is not relevant in XRPC, where there rather is a need for supporting arbitrary-shaped XML nodes as parameters as well as sequences of heterogeneously typed items. This is the reason, why our SOAP XRPC message format, while supporting the general SOAP standard over HTTP with the purpose of RPC, implements a new parameter passing subformat (SOAP XRPC $\neq$ SOAP RPC).

3.3.1 XRPC Request Messages

SOAP messages consist of an envelope, with a (possibly empty) header and a body. Inside the body, we define a `request` that specifies a `module` URI, an `at`-hint `location`, a function name `method` and its `arity`. The module definition must be accessible (via an HTTP connection) for the remote peer at the location given by the `at`-hint. In this way, we can rely on the XQuery facility to import the module from an arbitrary URL. The actual parameters of a single function call are enclosed by a `call` element. Each individual parameter consists of a `sequence` element, that contains zero or more values. Below we show the XRPC request message for the first example query that looks for films with Sean Connery:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
              xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://monetdb.cwi.nl/XQuery http://monetdb.cwi.nl/XQuery/XRPC.xsd">
  <env:Body>
```

```
    <xrpc:request xrpc:module="films" xrpc:method="filmsByActor" xrpc:arity="1"
                  xrpc:location="http://x.example.org/film.xq" xrpc:updCall="false">
      <xrpc:call>
        <xrpc:sequence>
          <xrpc:atomic-value xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
        </xrpc:sequence>
      </xrpc:call>
    </xrpc:request>
  </env:Body>
</env:Envelope>
```

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

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

Node Typed XML Elements XML nodes are passed by value in an ⟨`element`⟩ element:

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

Similarly, the XML Schema "XRPC.xsd" defines enclosing elements for `document`, `text`, `attribute`, `processing-instruction`, and `comment` nodes. A `document` node is represented in the SOAP message as a ⟨`document`⟩ element that contains the serialised document root. The `text`, `comment` and `processing-instruction` nodes are serialised textually inside the respective elements ⟨`text`⟩, ⟨`comment`⟩ and ⟨`processing-instruction`⟩. An `attribute` node is serialised *inside* an ⟨`attribute`⟩ element, for example, the attribute node `x="y"` is serialised as: ⟨`xrpc:attribute x="y"/`⟩.

User-Defined Types XRPC fully supports the XDM, a requirement for making it an orthogonal language feature. This implies that XRPC also supports passing of values of user-defined XML Schema types, including the ability to validate SOAP messages. XQuery already allows importing XML Schema files that contain such definitions. Values of user-defined *named* types are enclosed in SOAP messages by ⟨`element`⟩ elements, with an `xsi:type` attribute annotating their type. The XQuery system implementing XRPC should include an `xsi:schemaLocation` declaration as well as an `xmlns` namespace definition inside the ⟨`Envelope`⟩ element when values of such imported element types occur in the SOAP message. If a parameter has an *anonymous* user-defined schema type, its type information is lost. However, this can be avoided exploiting a future protocol extension (discussed in Section 5.5) by including the lowest ancestor-or-self element with a *named* schema type in the SOAP messages.

3.3.2 XRPC Response Messages

XRPC response messages follow the same principles. Inside the body is now an XRPC `response` element that contains the result `sequence` of the remote function call:

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

3.3.3 XRPC Error Message

If an XRPC server discovers an error during the processing of an XRPC request, it immediately stops execution and sends back an XRPC error message, using the format of the SOAP Fault message ([128], [89]). Thus, any error will cause a run-time error at the site that originated the query. Updating queries with 2PC enabled behave similarly, since update effects will only be applied if a query succeeds. If 2PC is not enabled, a failed updating query might already have applied changes somewhere. The exact semantics of updating queries is discussed in Section 3.4.2. As an example, the following SOAP Fault message indicates that a required module could not be loaded:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  <env:Body>
    <env:Fault>
      <env:Code><env:Value>env:Sender</env:Value></env:Code>
      <env:Reason><env:Text xml:lang="en">Could not load module!</env:Text></env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Remarks Our discussion of SOAP XRPC message is not fully done yet. In the next section, we will extend the format with support for isolation and updates. Then, in Section 3.5.2 we describe the *Bulk RPC* feature, that allows a single message to request multiple function calls. Finally, in Section 4.4.1 we describe a small extension to include `tag` attributes in `call` elements that allows keeping track of a deterministic distributed update order. The XML Schema Definition of the XRPC SOAP messages is given in Appendix A,

3.4 XRPC Formal Semantics

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

Notations We use the following notation and terms in this section:

- $\mathcal{P}$ denotes a set of *peer identifiers*. We use the peer identifier p_0 to denote the *local peer*, on which a particular query is started. All other peers $p_i \in \mathcal{P}$ are *remote peers*. In practice, a peer identifier is a URI from the `xrpc` protocol that contains a host and (optionally) a port number.
- $\mathcal{F}$ denotes a set of *XRPC function applications*. An XRPC call $f^{p_i \to p_j}$ that triggered from p_i that causes function f to be executed at p_j is an *updating XRPC call* ($f_u^{p_i \to p_j} \in \mathcal{F}_u$), if it calls an updating function; otherwise, it is a *non-updating XRPC call* ($f_r^{p_i \to p_j} \in \mathcal{F}_r$). If the evaluation of an XRPC call $f^{p_i \to p_j}$ requires evaluation of other XRPC call(s) at p_j, we term $f^{p_i \to p_j}$ a *nested XRPC call*.
- $\mathcal{M}$ denotes a set of XQuery *modules*. A module consists of a number of function definitions d_f. Each XRPC call $f^{p_i \to p_j}$ must correspond to a definition d_f from some module $m_f \in \mathcal{M}$.
- An *XRPC query* is an XQuery query q which contains at least one XRPC call $f^{p_i \to p_j} \in \mathcal{F}_q$, where $\mathcal{F}_q$ denotes the set of all function calls performed during execution of q. We call a query in which only one, non-nested XRPC call appears a *simple XRPC query*. An XRPC query q is an *updating XRPC query*, if it contains at least one update command or a call to an updating (XRPC) function.
- Each query operates in a *dynamic context*. The XQuery 1.0 Formal Semantics [67] defines that each expression is normalised to a *core* expression, which then is defined by a semantic judgment $dynEnv \vdash Expr \Rightarrow val$. The semantic judgment specifies that in the dynamic context $dynEnv$, the expression $Expr$ evaluates to the value val, where val is an instance of the XQuery Data Model (XDM). For now, we simplify the dynamic environment to a database state db (i.e., the documents and their contents stored in the XML database): $dynEnv \simeq db$. The *dynEnv.docValue* from the XQuery Formal Semantics [67] corresponds to db used here. To indicate a context at a particular peer p, we write db^p.
- When considering that a database may be changed by updates, we can view it as a function over time t as $db^p(t)$. In our formal rules, the default assumption on database states is that they stay equal over time, unless otherwise stated. When the time context t is clear, the shorthand notation db^p is used to refer to the current database state.

3.4.1 Read-Only XRPC Semantics

Basic Read-Only XRPC The semantics of executing a read-only function $f^{p_0 \to p_x}$ ($f \in \mathcal{F}_r$) is defined by extending the XQuery 1.0 semantic judgments with a new rule[1]:

$$\frac{\begin{array}{c} db^{p_0}(t_0) \vdash \langle \texttt{call} \rangle \{ \texttt{s2n}(v_1), \ldots, \texttt{s2n}(v_n) \} \langle / \texttt{call} \rangle \Rightarrow call; \\ send^{p_0 \to p_x} request(m, f_r, call);\ t_x \geq t_0 \\ db^{p_x}(t_x) \vdash \texttt{s2n}(f_r(\texttt{n2s}(call/*[1]), \ldots, \texttt{n2s}(call/*[n]))) \Rightarrow res; \\ send^{p_x \to p_0} reply(res); \\ db^{p_0}(t_0) \vdash \texttt{n2s}(res) \Rightarrow v_{res}; \end{array}}{db^{p_0}(t_0) \vdash f_r^{p_0 \to p_x}(v_1, \ldots, v_n) \Rightarrow v_{res}} \quad (\mathcal{R}_{\mathcal{F}_r})$$

This rule $\mathcal{R}_{\mathcal{F}_r}$ states that execution at p_0 of the read-only XRPC call $f^{p_0 \to p_x}(v_1, \ldots, v_n)$ in the dynamic context $db^{p_0}(t_0)$ (without further assumption on t_0) starts with constructing a $\langle \texttt{call} \rangle$ element that contains the SOAP representation of all parameters v_i. This XML representation,

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

described in the previous Section 3.3, is created by the sequence-to-node marshalling function `s2n()`, discussed below. Then, the request $(m, f, call)$ is sent to peer p_x. Here, m is the module URI (plus `at`-hint) in which function f_r is defined. The function f_r is then evaluated as a normal local function in the dynamic context of the remote peer $db^{p_x}(t_x)$, where we only assume $t_x \geq t_0$. The parameters of f_r, are obtained by using the inverse node-to-sequence marshalling function `n2s()` to produce the result node *res*. This result *res* is sent back to peer p_0, which finally converts *res* into the result sequence v_{res}.

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

Parameter Marshalling The SOAP representation of a sequence `$seq` is created in a new ⟨`sequence`⟩ node by the function:

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

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

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

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

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

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

One should note that `n2s()` and `s2n()` are internal functions only that do not need to be exposed to XRPC users, and in fact do not need to exist in reality, as each XRPC system implementation may have its own internal (efficient) mechanisms to process SOAP messages. In case of MonetDB/XQuery, beyond shredding the SOAP request and response messages, we do not spend any effort in `n2s()` nor `s2n()` on element construction to retrieve node values of the correct type, as our implementation directly chops up the shredded XML message in separate XML fragments per function parameter, and modifies node types internally (as the SOAP messages are invisible to the user, their integrity can be compromised at will by the system). It is possible, though, to implement `n2s()` and `s2n()` purely in XQuery, as we will show when we discuss the XRPC wrapper, that allows arbitrary XQuery processors to participate in distributed XRPC queries in Section 4.2.

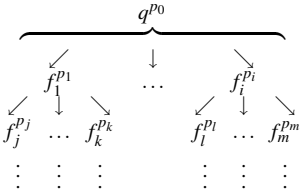


Figure 3.1: Nested XRPC calls

A final detailed remark on parameter marshalling is that XRPC requires the *caller* to perform parameter up-casting. The rationale is that such casting is already part of the standard function application code generated by any XQuery system, thus it is easy to do at the caller for XRPC calls, and it makes it easier to implement XRPC handlers that have no or limited XQuery capabilities (e.g. wrapped outside web services as in [134]).

Pass-By-Value An important choice implied by making `n2s()` and `s2n()` explicit in our Formal Semantics is to enforce *by-value* parameter passing in XRPC. If nodes are passed as parameters of an XRPC call, they will be serialised into a SOAP message, shipped to the remote side, and there new nodes will be constructed virtually of the correct type with equal-valued contents, but with different node identifiers. This can lead to a number of semantic differences between local and remote function application. We already mentioned that XPath navigation from node parameters over non-downwards axes (e.g. `parent`, `following`) will always produce empty results on the remote side. More subtly, if a function is invoked over XRPC with two nodes as parameters that have a `descendant-or-self` relationship, XRPC parameter marshalling will destroy this relationship at the remote side[2]. Finally, the XQuery Formal Semantics specifies that some consistent order should be enforced over nodes from different documents, but our semantics will not respect this order on their copies when shipped over XRPC.

The rationale behind this by-value choice is that a *by-reference* semantics would lead to complications when the upwards or sideways XPath axes are invoked on node parameters (or results) of XRPC calls. Correctly supporting that would either lead to the need to ship the full XML data fragment for all node parameters upfront (defeating the purpose of function shipping) or cause implicit communication when navigating beyond the descendants of such nodes. Obviously, call-by-value semantics complicate life when XRPC is used as the target language for automatic query distribution (as opposed to explicit XRPC query processing, where we can assume the query writer to be aware of the call-by-value semantics). In that case, the query optimiser has the task to make sure by-value parameter passing does not affect query semantics. The simplest solution is to refrain from function shipping in problematic cases, but more sophisticated solutions may be found for some query patterns.

[2]In Section 5.5, we discuss a future XRPC protocol extension that allows node parameters to be referred to using an `xrpc:fragid` and an `xrpc:nodeid` attribute that together identify a node serialised earlier in a special ⟨`fragment`⟩ section of an XRPC message. This alternative node representation can be used for nodes that are a descendant-or-self of another parameter that is fully serialised in the SOAP message. The `s2n()` function would then be altered to return nodes from the XML fragment that corresponds with that fully serialised parameter. This change of semantics ensures that ancestor/descendant relationships among parameters at the calling peer are preserved at the remote XRPC peer. This *indirect addressing* is useful for compressing the SOAP message. Moreover, if applied maximally, the resulting *pass-by-fragment* result/parameter passing, allows an distributed XRPC rewriter to relocate parts of certain query predicates that *do* depend on node identity (i.e., node-valued join conditions whose predicates only contain descendant/ancestor XPath steps).

Nested XRPC Calls The general pattern of XRPC function applications generated by a query is a *tree*, as each XRPC call may again perform more XRPC calls. This happens when a query contains multiple XRPC function applications, or when such a function application occurs inside a `for`-loop. In Figure 3.1, the arrow '$\rightarrow$' should be read as "XRPC call". The peers $p_0, p_1, \ldots, p_i, p_j, \ldots, p_k, \ldots, p_l, \ldots, p_m$ are not necessarily unique: some peer p_i (or in fact many such peers) may occur multiple times in this tree. When considering rule $\mathcal{R}_{\mathcal{F}_r}$, the dynamic environment $dynEnv^{p_i}$ containing the *current* database state db^{p_i} may thus be seen multiple times during query evaluation. In between those multiple function evaluations, other transactions may update the database and change db^{p_i}. Thus, those different XRPC calls to the same remote peer p_i from the same query q may see different database states. This will not be acceptable for some applications and therefore, we deem it worthwhile to define *repeatable read* isolation for queries that perform XRPC calls.

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

$$\frac{\begin{array}{c} db^{p_0}(t_q^{p_0}) \vdash \langle \texttt{call} \rangle \{ \mathbf{s2n}(v_1), \ldots, \mathbf{s2n}(v_n) \} \langle / call \rangle \Rightarrow call; \\ send^{p_0 \rightarrow p_x} request(q, m, f_r, call); \\ db^{p_x}(t_q^{p_x}) \vdash \mathbf{s2n}(f_r(\mathbf{n2s}(call/*[1]), \ldots, \mathbf{n2s}(call/*[n]))) \Rightarrow res; \\ send^{p_x \rightarrow p_0} reply(q, res); \\ db^{p_0}(t_q^{p_0}) \vdash \mathbf{n2s}(res) \Rightarrow v_{res}; \end{array}}{db^{p_0}(t_q^{p_0}) \vdash f_r^{p_0 \rightarrow p_x}(v_1, \ldots, v_n) \Rightarrow v_{res}} \quad (\mathcal{R}'_{\mathcal{F}_r})$$

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

Clearly, XRPC with repeatable reads requires more resources to implement, as some *database isolation* mechanism (of choice) will have to be applied to retain $db^{p_x}(t_q^{p_x})$ across calls. The transaction mechanism of MonetDB/XQuery, for instance, uses snapshot isolation[32] based on shadow paging, which keeps copies of modified pages around. Systems that provide the isolation levels serialisable or repeatable reads (obviously) can also provide this semantics.

A quite common reason why a peer is called multiple times in the same query and why the need for repeatable reads arises, is when an XRPC call appears inside a for-loop. In Section 3.5.2 we describe how *Bulk RPC* helps avoid these costly isolation measures in case of *simple* XRPC queries (i.e., those that contain only one non-nested function application).

Other Isolation Levels If we would suppose that all peers involved in q support the isolation level *snapshot isolation*, and all would use the *same* timestamp t_q as the one in which the original query executes, i.e., $t_q^{p_0} = \cdots = t_q^{p_x} = \cdots = t_q^{p_m} = t_q$, we could obtain the isolation level *distributed snapshot isolation*. Just using a globally consistent query timestamp is actually not enough for that, extra effort is needed to enforce distributed commits to happen at the same time point (one way to do that is to block or abort incoming reads while a node is in the prepared state – this is called the pessimistic approach in [158]). For this to be meaningful in practice, however, we would have to have a representation of t values (until now, this

is left opaque) that allows a full ordering, thus enabling us to define a "happened before " query/transaction order $t_{q_1} \ll t_{q_2}$.

However, as XRPC is also intended for use in P2P settings, we make no assumptions on a centralised distributed transaction coordinator that could give out unique and monotonically increasing t numbers. In absence of that, one could think of t numbers generated by Lamport Clocks [116], but while this method guarantees that a transaction that depends on a previous one ("happened before") has a smaller Lamport clock value, the reverse inference cannot be made (i.e., meaningfully enforcing a transaction order depending on such t-s) unless all peers participate in all queries (which again is not a reasonable assumption in P2P). Of course, we can think of t as being "exact" (UTC) time, but as we do not want to assume either that all participating peers possess (synchronised!) Strontium grade precision clock hardware, this is only a theoretical notion. For this reason, we leave the maximum XRPC isolation currently at the repeatable read level, though finding a distributed isolation level useful in P2P is on our future work agenda.

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

A timeout mechanism is inevitable, even if XRPC would use a 2PC-like coordination protocol to signal the finishing of a query (for updates, XRPC actually uses a 2PC protocol via Web Services Atomic Transaction [55]), because such a coordination protocol also needs a timeout to conclude that remote hosts are no longer responding. Automatically computing a good timeout value requires a cost model that takes into account the query, data-distribution, network, and peer characteristics – a task we leave for our future work on automatic query distribution. Therefore, the timeout to use is specified in the query using `declare option xrpc:timeout` ⟨`sec`⟩, so users and applications can set them according to their needs.

3.4.2 XRPC Update Semantics

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

Basic Updating XRPC The semantics of executing a single updating function $f^{p_0 \to p_x}$ ($f \in \mathcal{F}_u$), is defined by extending the XQuery 1.0 semantic judgments with a new rule:

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

The above rule $\mathcal{R}_{\mathcal{F}_u}$ states that update functions apply the pending update list Δ immediately, producing a new current remote database state db^{p_x}. For this purpose, we use the internal function `applyUpdates()` defined in the XQUF [58] that carries through all changes in a pending update list. Note that this rule executes an updating call between p_0 and p_x in database states from t_0 resp. t_x with no other assumptions than $t_x \geq t_0$. Typically, an implementation may choose to use db^{p_x}, i.e. the latest database state to handle each XRPC request.

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

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

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

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

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

Obviously, atomically committing a distributed transaction requires a protocol like 2PC or one of its more advanced derivatives [135, 85]. We decided not to add 2PC to the XRPC network protocol, but rather rely on the recent industry standard Web Services Atomic Transaction [55] that provides exactly this feature for distributed web-service transactions. The Web Services Atomic Transaction [55] standard provides a fairly vanilla SOAP-based 2PC interface with e.g. `Prepare()` and `Commit()` functions. It is embedded in the Web Services Coordinator framework [54] that allows registering a collection of peers that participate in a distributed transaction, and subsequently run a transaction protocol on those peers (in this case WS-AtomicTransaction). Thus, in order to support updates with this isolation level, XRPC systems must implement support for these web service interfaces, and offer them over the same HTTP SOAP server that runs XRPC.

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

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

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

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

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

Finally, the XQUF specifies that when the same node is updated twice in the same query, the order in which the different update actions on that node are applied is *non-deterministic*! This means that we can simply union all individual $\Delta_q^{p_x}(i)$ pending update lists (one for each XRPC call handled in p_x for q) to get a full update list $\Delta_q^{p_x}$ without worrying about preserving some proper order on the update actions. In Section 4.4, we define a deterministic update order for XQUF and devise a way to enforce it over XRPC using a small XRPC protocol extension, despite the out-of-order execution effects of Bulk RPC that will be observed at the end of Section 3.5.1.

3.5 Loop-lifted Implementation of XRPC

We have implemented XRPC in open-source MonetDB/XQuery, an efficient yet purely relational XDBMS [41]. It consists of the MonetDB relational database back-end, and the *Pathfinder* compiler [88], that translates XQuery into relational algebra, as front-end. The essence of the compilation technique employed by Pathfinder is *loop-lifting* [88], which translates XPath/XQuery expressions inside `for`-loops into single bulk relational query plans that process all iterations of the loop independently of each other. Loop-lifting makes MonetDB/XQuery inherently different (and often faster) than those XQuery *interpreters* that tend

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

Table 3.2: Relational algebra generated by Pathfinder

to strictly follow the `for`-loop order syntactically suggested by a query. In case of Pathfinder, with its loop-lifted approach to XQuery translation, it was trivial to generate Bulk RPC requests for any XRPC call found in an XQuery. Hence, an XRPC call nested in a `for`-loop taken many times leads to only a single Bulk XRPC request/response, which invokes the function for all iterations of the loop in bulk. This optimisation dramatically reduces the number of request/response messages sent and thus the impact of the network latency on query performance.

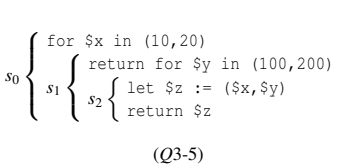
The XRPC module contains an ultra-light HTTP daemon implementation [122] that runs a request handler (the XRPC server), and contains a message sender API (the XRPC client). We also had to add support for the `execute at` syntax to the Pathfinder XQuery compiler, and change its code generator to generate *stub code* that invokes the new message sender API.

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

The request handler, on the other side, behaves similarly. It listens for SOAP requests and shreds incoming messages into a temporary relational table, from which the parameter values are extracted. As MonetDB/XQuery is a relational system, XQuery values are all represented as (temporary) relational tables. The module function specified in the SOAP request is then executed locally with these parameter tables, producing a result table. The request handler then builds a response message in which this result table is serialised into XML, using the normal MonetDB/XQuery serialisation mechanism onto the network socket. As we re-used the shredding and serialisation functionality already in MonetDB/XQuery, as well as an off-the-shelve open source HTTP daemon [122], implementation was limited to a small parser extension, and stub code generation.

3.5.1 Relational XQuery and Loop-Lifting

The *Pathfinder* compiler [88] translates XPath/XQuery expressions into bulk query plans formulated in the vanilla relational algebra, depicted in Table 3.2. All operators are well-known, except perhaps the row numbering operator ρ, which is similar to the SQL:1999 operator DENSE_RANK: $\rho_{b:\langle a_1,\ldots,a_n\rangle/p}(q)$ assigns each tuple in q a rank (i.e., number), which is saved in column b. The constraint for the enumeration is the implicit order of q by the columns $a_1,\ldots,a_n$. Numbers ascend consecutively from 1 in each partition defined by the optional grouping column p.



(*Q*3-5)

loop

iter
1
2
3
4

x

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

y

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

z

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

Figure 3.2: Example query (*Q*3-5) and its loop-lifted relational representation

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

pos	item
1	x_1
2	x_2
⋮	⋮
n	x_n

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

The result of an XQuery at each step of bottom-up compilation is a relational plan that yields the result sequence *for each* nested iteration, all stored together. To make this possible, these intermediate tables have three columns: iter|pos|item, where `iter` is a logical iteration number, as shown in the tables below. For each scope, we keep a `loop` relation that holds all `iter`-s.[3] Figure 3.2 shows an example query (*Q*3-5) and its loop-lifted relational representation. If we focus on the execution state in the innermost iteration body (marked as scope s_2) of (*Q*3-5), there will be three such tables that represent the live variables `$x`, `$y` and `$z` respectively. As we can see from the `iter` columns, there are four iterations in scope s_2 (numbered from 1 to 4) and as expected, `$x` takes the value `10` in the first two iterations and the value `20` in the second two iterations. Similarly, `$y` takes the value `100` in the odd iterations and the value `200` in the even ones. Finally, `$z` is a sequence of two values in all four iterations (having the value of `$x` concatenated with `$y`).

3.5.2 Bulk RPC

```
import module namespace f="films" at "http://x.example.org/film.xq";
for $actor in ("Julie Andrews", "Sean Connery")
let $dst := "xrpc://y.example.org"
return execute at {$dst} {f:filmsByActor($actor)}          (Q3-2)
```

actor

iter	pos	item
1	1	"Julie Andrews"
2	1	"Sean Connery"

dst

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

Our earlier example query (*Q*3-2) (repeated above) contains a function application inside a `for`-loop. Inside this loop, the variables `$dst` and `$actor` yield relational tables shown on the right. Thus, the value of `$dst` is the same in both iterations of the `for`-loop, whereas `$actor` takes on values "`Julie Andrews`" in the first and "`Sean Connery`" in the second iteration.

[3]The `loop` relation allows keeping track of empty sequence values, encoded by the absence of tuples in the expression representation.

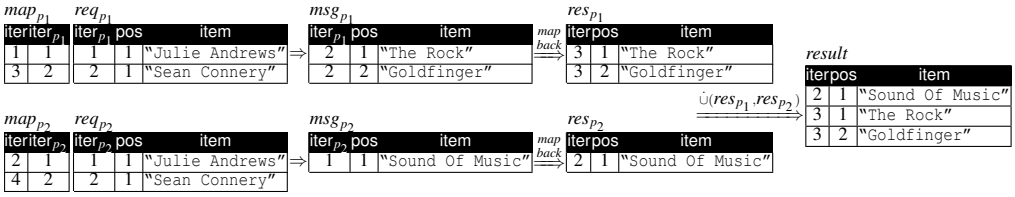


Figure 3.3: Relational processing of Bulk RPC (multiple destinations example)

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

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

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

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

actor

iter	pos	item
1	1	"Julie Andrews"
2	1	"Julie Andrews"
3	1	"Sean Connery"
4	1	"Sean Connery"

dst

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

Let us now consider our other previous example (*Q*3-3). We now have an inner `for`-loop with four iterations, but `$dst` takes on two *different* values, identifying peers "y.example.org" and "z.example.org", in respectively the odd and even iterations. The general rule to translate a loop-lifted XRPC call is shown in Figure 3.4, and Figure 3.3 shows the intermediate steps taken. The system establishes a list of unique peers, and for each p extracts from each parameter `iter|pos|item` those iteration (tuples) that invoke the function on p. The resulting request tables (req_p) are used to generate a Bulk RPC to p. Observe that using ρ a new $iter_p$ column is created, and a mapping table (map_p) that maps old to new iteration numbers. The mapping table is then again used to map the new iteration

$$
\begin{array}{rl}
\boxed{\texttt{iter}}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ result & \Leftarrow \dot{\cup}_{\forall p \in \delta(dst.item)}(res_p) \\
with: & \\
\boxed{\texttt{iter}}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ res_p & = \pi_{iter,pos,item}(\bowtie_{iter_p = iter_p}(msg_p, map_p)) \\
\boxed{\texttt{iter}}\boxed{\texttt{iter}_p}\ map_p & = \pi_{iter,iter_p}(\rho_{iter_p}(\sigma_{item=p}(dst))) \\
\boxed{\texttt{iter}_p}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ msg_p & = f(req_p^1, \ldots, req_p^n)@p \\
\boxed{\texttt{iter}_p}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ req_p^i & = \pi_{iter_p,pos,item}(\rho_{pos}(\bowtie_{iter=iter}(map_p, param_i)))
\end{array}
$$

$$
\texttt{execute at } \{\boxed{\texttt{iter}}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ dst\}\{f(\boxed{\texttt{iter}}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ param_1, \ldots, \boxed{\texttt{iter}}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ param_n)\} \Rightarrow \boxed{\texttt{iter}}\boxed{\texttt{pos}}\boxed{\texttt{item}}\ result
$$

Figure 3.4: Relational translation of XRPC

numbers back into old ones, and all result tables (res_p) are united with a (merge-)union on the `iter` column, to guarantee the correct order of the result.

Parallel & Out-Of-Order The XRPC execution in Figure 3.3 performs two Bulk RPC calls. The first call processes both values of `$actor` on "y.example.org". Then a second call performs the same task on "z.example.org". It is important to observe that this order of processing is different than what is suggested by the query (i.e., first Julie Andrews on both, then Sean Connery on both). If a loop-lifted XRPC function application has multiple destination peers, MonetDB/XQuery improves performance by dispatching all Bulk RPC requests *in parallel*, which makes the exact order in which peers execute the query unpredictable. After all parallel results are united, the mapping of temporary $\texttt{iter}_p$ numbers into `iter`s guarantees that the final result is produced in the correct order.

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

```
import module namespace f="films" at "http://x.example.org/film.xq";
for $name in ("Julie", "Sean")
let $connery := concat($name, " ", "Connery")
let $andrews := concat($name, " ", "Andrews")
return (execute at {"xrpc://y.example.org"} {f:filmsByActor($connery)},
        execute at {"xrpc://y.example.org"} {f:filmsByActor($andrews)} )                    (Q3-6)
```

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

The above is also a good example of a query that needs *isolation*, because it handles two RPC requests inside the same query. While in this particular case, those two requests could potentially be combined, this is much harder if two different functions would be executed, or downright impossible if the parameters of one depend on the outcome of the other. Certain classes of queries, such as those that contain only a single non-nested XRPC call, can be easily identified at compile time to send at most one XRPC request to each destination peer. For such queries, we can use the cheaper XRPC mechanism without `queryID` (see Section 3.4), while still guaranteeing repeatable reads.

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

	no function cache		with function cache	
	`$x=1`	`$x=1000`	`$x=1`	`$x=1000`
one-at-a-time	133	2696	2.6	2696
bulk	130	134	2.7	4

Table 3.3: XRPC performance (msec): loop-lifted vs. one-at-a-time; no function cache vs. with function cache.

3.5.3 Performance Evaluation

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

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

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

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

While in MonetDB/XQuery loop-lifting of XRPC calls (i.e., Bulk RPC) is the default, we also implemented a one-at-a-time RPC mechanism for comparison. The left half of Table 3.3 (the "No Function Cache" column) shows the experiment where we compare performance of Bulk RPC with single RPC at-a-time, while varying the number of loop iterations `$x`. It shows that performance is identical at `$x=1`, such that we can conclude that the overhead of Bulk RPC is small. At `$x=1000`, there is an enormous difference, caused by *(i)* serialisation/deserialisation of the request/response messages, *(ii)* network communication cost and *(iii)* overhead of function call (1000 calls instead of 1 call). This is easily explained as the one-at-a-time RPC experiment involves performing 1000 times more synchronous RPCs.

Throughput We also carried out bandwidth experiments (details omitted for space) that scaled request and response payloads. Here we observed throughput of 8MB/s (large requests) and 14 MB/s (large responses), which correspond roughly with resp. the document shredding and serialisation speed of MonetDB/XQuery [41]. Thus, like other SOAP-based messaging [84], XRPC data throughput on a fast local 1Gb network is CPU-bound rather than network-bound (though in a WAN it is likely to be the other way round).

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

The feature of *prepared queries* is well-known for RDBMS. It allows a parameterised query plan to be parsed and optimised off-line, such that an application can quickly enter actual parameters in the prepared plan and execute it. The ODBC and JDBC APIs export this functionality of relational databases using a programming language binding. MonetDB/XQuery has a mechanism for supporting prepared queries that does not need specific API support. Exploiting the fact that a prepared query is in essence a function with parameters, MonetDB/XQuery *caches* all query plans for (loop-lifted) function calls, for functions defined in XQuery Modules. Queries that just load a module and call a function in it with constant values as parameter, are detected by a pre-parser. The pre-parser then extracts the function parameters, and feeds them into a cached query plan. In MonetDB/XQuery, queries

XMark document	1MB	10MB	100MB	500MB	1000MB
Bandwidth (MB/sec)	2	12	28	14	15

Table 3.4: XRPC bandwidth for serialising XML documents

on small data sets can be accelerated ten-fold by this mechanism [41]. Note, that the function cache is **not** a query cache: queries are executed always on the latest data, and the performance improvement stems solely from the fact that query translation and optimisation is avoided.

This same function cache mechanism is used by the XRPC request handler. This means that in MonetDB/XQuery an XRPC request usually does not need query parsing and optimisation, just execution. The right half of Table 3.3 (the "With Function Cache" column) shows the impact of enabling the function cache: we see the processing time go down by 130ms (XQuery module translation time), improving both the single- and many-iteration Bulk RPC experiments. Thanks to the function cache, MonetDB/XQuery can achieve a minimum RPC latency of 3 msec – which is identical to that of commercial-strength software like .NET ([84, 130]).

Document Serving To examine the performance of XRPC in serialising XML documents, we used an HTTP client (wget) that retrieves a number of XMark documents of increasing size. Such requests are automatically handled as a call to `fn:doc()` over XRPC. Table 3.4 shows that XRPC achieves a bandwidth of 14MB/sec on average. Only for small documents ($<$ 10MB) the bandwidth is lower, due to fixed start-up cost.

3.6 Conclusion

In this chapter, we introduced XRPC, a minimal XQuery extension that enables distributed query execution with a focus on efficiency and interoperability. We first gave a formal definition of the syntax and the semantics of XRPC, including the semantics of distributed updates, that follow from the use of XQUF updating functions over XRPC. This includes the definition of two isolation levels for read-only and updating XRPC queries. Since interoperability is a major goal, the XRPC proposal also comprises a message protocol, which we chose to base on SOAP. Such a SOAP protocol has the additional advantage of seamless integration with web services and AJAX-based GUIs.

Our experiences in MonetDB/XQuery suggest that adding XRPC to existing XML database systems is easy; as shredding, serialisation and HTTP functionality are usually already present, the work is limited to a small parser extension and stub code generation. The SOAP XRPC protocol supports the concept of Bulk RPC, the execution of multiple function calls in a single message exchange. This amortises network and parsing latencies, and can make XRPC a quite efficient communication mechanism. We have shown that the loop-lifting technique, pervasively applied in our MonetDB/XQuery system for the translation of XQuery expressions to relational algebra, can easily generate such Bulk RPC requests. In the next section, we will show in our Saxon experiments that Bulk RPC enables set-oriented optimisations such that Bulk RPC execution of a selection function can be handled using a join strategy.

4

Distributed XQuery With XPRC

4.1 Introduction

In this section, we discuss various uses of XRPC for distributed XQuery processing on heterogeneous XQuery engines.

First, we show that XRPC is not system-specific: every XQuery data source can service XRPC calls using a simple wrapper. Since XQuery is a pure functional language, we can leverage techniques developed for functional query decomposition to rewrite data shipping queries into XRPC-based function shipping queries. Powerful distributed database techniques (such as semi-join optimisations) map directly onto Bulk RPC, opening up interesting future work opportunities. We demonstrate this with experiments in which MonetDB/XQuery and Saxon work together over XRPC.

Second, we turn our attention to the interaction between XRPC and XQUF. We first define a deterministic distributed update semantics and show that a small extension to the SOAP XRPC protocol enables the protocol to conform to the deterministic update semantics. We then describe how the industry standard Web Service Atomic Transaction[55] can be adapted to support atomic distributed commits of XQUF queries on heterogeneous XQuery engines.

While XRPC already allows XQuery engines to perform P2P queries, it still misses a number of vital P2P functionalities (robust connectivity, peer and resource discovery, approximate query/transaction processing). In the final part of this chapter, we present preliminary work on MonetDB/XQuery*, in which we integrate existing XDBMS and P2P structures to provide P2P data management facilities.

XRPC as Target Language One of the design goals of XRPC is – besides it being directly useful as an explicit instrument to write distributed queries – to have it serve as the target language for a distributed XQuery optimiser that takes queries without XRPC as input (thus data shipping only), and produces decomposed queries as output that use XRPC for function shipping. Our choice to make distributed execution explicit in terms of remote functions and their dependencies (parameters), aligns well with XQuery being a pure functional language. Query decomposition techniques [105] can thus be applied to decompose the full query (function) into sub-queries (again functions), that each can in theory be executed on any of the participating sites.

Automatic query decomposition techniques are discussed in the next chapter. In this chapter, we limit ourselves to showing how some well-known distributed query execution strategies, such as the distributed semi-join strategy, can be elegantly expressed in XRPC.

To demonstrate the performance opportunities of XRPC, as well as its interoperability, we provide some initial performance experiments with one peer running MonetDB/XQuery, and another running Saxon.

The implementation of XRPC in the open-source XML database system MonetDB/XQuery (`http://monetdb.cwi.nl`) already allows query writers to experiment with distributed query processing strategies, but we show using Saxon that even without XRPC being integrated into other XQuery systems, we can achieve our goal of cross-system distributed querying using an *XRPC wrapper* (Section 4.2). This wrapper is a SOAP service handler which generates an XQuery query that uses the incoming SOAP request message as an input, iterates over all function call requests in it, applying the local XQuery function on the supplied parameters, and uses element construction to produce a SOAP response message that is sent back by the wrapper.

One should note that the capabilities of such an XRPC wrapper outclass that of the well-known wrapper architecture applied in federated database systems [114]. Not only can this architecture do without a centralised integrating engine (XRPC allows for true P2P query processing), we will also show that the possibility to submit sets of requests (that can each have sequence-typed correlated parameters) allows query writers to, e.g., express the well-known distributed query processing strategy of *semi-join reduction* by simply passing a key parameter to the remotely called function.

Deterministic Updates The W3C Candidate Recommendation proposal for the XQuery Update Facility leaves it undetermined how to handle multiple updates to the same node. For example, if we have an XML document ⟨a⟩ named "a.xml", then its value after executing the update expression

```
for $n in (<b/>,<c/>) return insert node $n as first into doc("a.xml")
```

can be either ⟨a⟩⟨b/⟩⟨c/⟩⟨/a⟩ or ⟨a⟩⟨c/⟩⟨b/⟩⟨/a⟩. Arguably, this semantics does not match the transactional semantics in databases very well. In MonetDB/XQuery, we thus choose to implement the XQUF deterministically, by respecting the `for`-loop order (respectively the sequence construction order), in which the multiple update statements occur in the query (in the above case yielding ⟨a⟩⟨b/⟩⟨c/⟩⟨/a⟩).

The question we address here is how to achieve deterministic semantics in *distributed* updates using our loop-lifted RPC technique. Note that XRPC is fully orthogonal to XQuery, thus it is allowed to call user-defined *updating functions* over XRPC. Updating functions can contain `for`-loops and sequence constructors, which might again make (multiple) other XRPC updating function calls to other peers. Thus, distributed update queries generally involve a group of peers and within a single query the same peer may even be involved multiple times, potentially through different function call sequences. Our loop-lifting approach to XRPC ("bulk" RPC requests) changes the order in which RPC function calls are evaluated. This means that the order in which updates must be applied may differ from the order in which the XRPC function calls were received. To address this issue, we formulate an extension to our bulk SOAP XRPC protocol that allows keeping track of deterministic update order, while conserving the performance advantages of loop-lifted RPC.

Distributed Transactions During a single XRPC query, it may happen that multiple read-only XRPC requests are sent to the same site. In the *repeatable read* isolation level we define, each request from the same query is guaranteed to see the same database state. XRPC queries may themselves also update the databases by invoking XQUF "updating functions"

```
import module namespace func = "functions" at "http://example.org/functions.xq";
declare namespace env = "http://www.w3.org/2003/05/soap-envelope";
declare namespace xrpc = "http://monetdb.cwi.nl/XQuery";
⟨env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://monetdb.cwi.nl/XQuery http://monetdb.cwi.nl/XQuery/XRPC.xsd"⟩
  ⟨env:Body⟩
    ⟨xrpc:response xrpc:module="functions" xrpc:method="getPerson"⟩{
      for $call in doc("/tmp/request_nnn.xml")//xrpc:call
      let $param1 := n2s($call/xrpc:sequence[1])
      let $param2 := n2s($call/xrpc:sequence[2])
      return s2n(func:getPerson($param1, $param2))
    }⟨/xrpc:response⟩
  ⟨/env:Body⟩
⟨/env:Envelope⟩
```

Table 4.1: XQuery generated for the `getPerson()` XRPC request

over XRPC. Note that XQUF queries only perform side-effecting actions *after* all query execution has finished, such that during query execution the database state is constant, and updating queries behave much like read-only queries. Obviously, atomically committing a distributed transaction requires a protocol like two-Phase Commit (2PC). We decided not to add 2PC to the XRPC network protocol, but rather rely on the recent industry standard Web Services Atomic Transaction (WS-AtomicTransaction) [55, 54] that provides exactly this feature for distributed web-service transactions.

Integrating XQuery and P2P Our approach to equip XRPC with P2P facilities is to integrate services offered by diverse P2P network structures, such as the Distributed Hash Tables (DHTs), into existing XDBMS. In MonetDB/XQuery*, we propose different ways of integration that avoid any further intrusion into the XQuery language and semantics. We also show how the proposed approaches, similarly to Bulk RPC, will lead to further query optimisation opportunities where the XDBMS interacts with the underlying P2P network. XRPC and MonetDB/XQuery* are adopted by StreetTiVo, a P2P collaborative video analysis and metadata distribution application. We discuss the architecture of StreetTiVo in Chapter 7 and show how XRPC and MonetDB/XQuery* enable quick development of complex P2P application such as StreetTiVo.

4.2 Cross-System Distributed XQuery

Cross-system distributed XRPC querying can be achieved even without XRPC being integrated into an XQuery processing engine. What is needed is a simple *XRPC wrapper* on top of the XQuery system, as shown in Figure 4.1. The XRPC wrapper is a SOAP service handler that stores the incoming SOAP XRPC request message in a temporary location, generates an XQuery query for this request, and executes it on an XQuery processor. The generated query is crafted to compute the result of a Bulk XRPC by calling the requested function on the parameters found in the message, and to generate the SOAP response message in XML using element construction. Such an XRPC wrapper only allows its underlying XQuery engine to *handle*

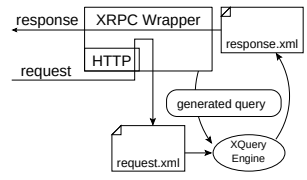


Figure 4.1: XRPC wrapper architecture

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

Table 4.2: Saxon latency via the XRPC Wrapper (msec)

calls with normal XRPC-incapable systems, but obviously does not allow making outgoing XRPC calls from them.

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

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

Table 4.1 shows the query generated by an XRPC wrapper to handle the `getPerson()` request.

The XRPC protocol includes information about the arity of the function (as well as its return type), so it is easy to generate the right number of *param* parameters in the call. The brunt of the work is done by the `n2s()` and `s2n()` marshalling functions, introduced in Section 3.4.1. These functions can be implemented purely in XQuery.

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

Saxon Experiments Using the wrapper, we can run a number of experiments on the Saxon XSLT/XQuery processor [109] (Saxon-B 8.7). The results are shown in Table 4.2. Like the experiments in Section 3.5.3, we put the `execute at` inside a `for`-loop with a varying number of iterations (`$x`) to study the performance impact of Bulk RPC. By absence of a function cache, Saxon latency is dominated by start-up and compilation time, so we focus here on the internal Saxon timings (compile, treebuild, exec) and disregard network communication cost, which is a few msec at most. For the `echoVoid` experiment, we see that Bulk RPC again allows amortising XRPC latency really well: instead of 1000 times the latency, with a 1000 times more work, total latency increases just over a factor of 2. As the execution time still is increased by a factor of 30, the low impact is due to other amortised latencies, in parsing the XML request document, compiling the query, etc.

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

that performance remains linear in the size of the XMark document, just like it was in the single call selection.

4.3 Distributed XQuery Optimisation

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

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

```
for $p  in doc("persons.xml")//person, $ca in doc("xrpc://B/auctions.xml")//closed_auction
where $p/@id = $ca/buyer/@person
return <result>{$p,$ca/annotation}</result>                                    (Q4-1)
```

The above query is executed at peer p_a. For each person and for every item this person has bought, query $Q7$ returns the `person` node and the `annotation` node of the bought item in a new `result` node. For the moment, assume that `fn:doc()` is invoked with a compile-time known constant URI from our `xrpc://` URI name scheme, indicating that the peer is known to support XRPC.

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

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

```
import module namespace b="functions_b" at "http://example.org/b.xq";
for $p  in doc("persons.xml")//person, $ca in execute at {"B"} {b:Q_B1()},
where $p/@id = $ca/buyer/@person
return <result>{$p,$ca/annotation}</result>                    Rewritten query Q4-1-1
```

This heuristic rewrite can simply be triggered by the presence of `fn:doc()`. The required analysis to determine how much of the XQuery (Core) expression is dependent on that `fn:doc()` alone, and therefore can be pushed, is highly similar to the analysis method developed for XML projection [125].

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

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

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

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

However, this approach does require a refinement of the work in [125]. One must bear in mind that any of the rewrites discussed here should only be made by an automatic rewriter *if* it can establish that the call-by-value semantics of XRPC will not compromise the semantics of the query. This at least involves a check whether nodes that come from pushed expressions are only navigated downwards, and also involves checking against node identity tests and order-dependent (e.g., `order by`) processing of node sequences that stem from multiple `fn:doc()` calls pushed to different sources.

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

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

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

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

Distributed Semi-Join The classical distributed semi-join strategy [25, 178] can be employed as well. The XRPC equivalent of the semi-join strategy uses an XRPC function call with a loop-dependent parameter. In this case, the person `@id` for all persons can be passed in a loop to a function executed at p_b that returns those closed auctions with buyers having that `@id`:

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

```
import module namespace b="functions_b" at "http://example.org/b.xq";
for $p  in doc("persons.xml")//person
let $ca := execute at {"B"} {b:Q_B3($p/@id)}
return if(empty($ca)) then () else <result>{$p, $ca/annotation}</result>     Rewritten query Q4-1-3
```

This shows that federating data sources with XRPC (even via the XRPC Wrapper) is more powerful than the "wrapper-architecture" [114] used in federated database systems. Such wrappers typically lack the possibility to push table-valued parameters into data sources, which is required for the semi-join optimisations. It is worth pointing out that the loop-lifted implementation of XRPC is essential for the efficiency of the distributed query plans discussed in this section. The XRPC calls in the inner `for`-loop of the rewritten query $Q7$-1 and $Q7$-3 require only *one* message exchange between p_a and p_b. Without the loop-lifted implementation, the network can easily get flooded by the huge amount of messages.

Saxon and MonetDB/XQuery Joined by XRPC To demonstrate the interoperability, expressiveness and performance potential of XRPC we run query $Q7$ on two peers using all four mentioned strategies. On peer p_a (the local peer), we run MonetDB/XQuery with the document "persons.xml" (1.1MB, 250 `person` nodes); on peer p_b the Saxon XSLT/XQuery processor with the document "auctions.xml" (50MB, 4875 `closed_auction` nodes). There are 6 matches between the `person` nodes and the `closed_auction` nodes.

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

Table 4.3: Execution time (msecs) of query $Q4$-1 distributed on MonetDB/XQuery and Saxon (Saxon time includes network).

All communication between MonetDB/XQuery and Saxon happens via XRPC. The XRPC wrapper described in Section 4.2 is used to generate the XQuery query from an XRPC request message.

The measured execution times are shown in Table 4.3. In the column "MonetDB Time" are execution times on peer p_a and in the column "Saxon Time" are execution times on peer p_b. The Saxon time was measured by subtracting MonetDB time from total time, such that it also included communication. We should stress that this experiment is not a rigorous evaluation of distributed query execution strategies, rather a demonstration of the possibilities of XRPC. The results here show that the "data shipping" query is relatively expensive, since it spends quite some Saxon time on shipping the 50MB document and then still needs to do the join. The "predicate push-down" approach improves the performance, as we would expect. The "execution relocation" largely relieves the MonetDB peer from execution responsibilities, but still ships a significant amount of data and tasks Saxon with the whole join and result construction effort (where it takes longer than on MonetDB). The "distributed semi-join" is the strategy that incurs least data shipping, and is most efficient in this case.

4.4 Deterministic Distributed Updates

The W3C Candidate Recommendation of the XQUF [58] does not determine the ordering among newly inserted nodes if those nodes are inserted into *the same* target node using the same kind of `insert` expression (`into` or `as first/last` or `into before/after`). The Candidate Recommendation specifies that this ordering is *implementation-dependent*.

Definition of Deterministic Updates The motivation in MonetDB/XQuery to exercise our liberty to implement the XQUF deterministically, is simply that order matters in XML. The solution chosen is that if the XQUF working draft leaves the ordering of updates actions undetermined, we respect the order in the pending update list. The XQUF working draft specifies how this list is built up incrementally. For two XQuery language constructs, namely `for`-loops and sequence construction, the working draft states that two pending updates must be merged with the `upd:mergeUpdates()` internal function. The XQUF leaves the working of this function unspecified, and our solution is to implement it with *concatenation*. Thus, each new pending update sublist (second parameter of `upd:mergeUpdates()`) is appended to the existing list (its first parameter). Note that this definition of update order is "intuitive" in that it respects the `for`-loop iteration order, as well as sequence construction order. Our rules $\mathcal{F}_u$ and $\mathcal{F}'_u$ further lead to synchronous function call semantics when updating functions are called over XRPC.

The Challenge Now that the MonetDB/XQuery implementation of XQUF cares about the update order, our challenge is to extend this deterministic update semantics to distributed updates. In the end of Section 3.5.2, we showed an example query that executed two Bulk RPCs on the same peer, and discussed how our loop-lifting technique causes the function to

be evaluated out of the intuitive order (this intuitive order is also followed by the XQUF to build the pending update list). The below query is the updating equivalent of that previous example, now using a hypothetical updating function `appendLog`, that appends entries to a log:

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

Our deterministic XQUF requires us to write first two Julie entries in the log, followed by two Sean entries. The loop-lifting, however, will process the two Connery invocations first, followed by the two Andrews. In this section, we describe an extension to the SOAP XRPC message format that allows re-ordering the pending update list at commit time such that the correct update order is followed.

4.4.1 Order-Correct Update Tags

We start by characterising the update actions a on behalf of query q that may be found in the pending update lists $\Delta_q@p$ at the various peers p. Second, we define a *conceptual* Distributed Pending Update Table (DPUT), that holds all $\langle p,a\rangle$ combinations in the required order. Then, we define an additional third $\mathcal{T}$ column for the DPUT that holds a *tag*, and explain how these tag values are constructed. We show that this $\mathcal{T}$ column will always appear in sorted order, given that the DPUT contains the required output order. From this, we can then conclude that if each peer orders its local $\Delta_q@p$ on $\mathcal{T}$ just before commit, it will apply the update actions in the correct order. As a last step we show how the tags are constructed during query execution and passed between peers using a small (and final) extension to the SOAP XRPC message protocol.

Update Actions There are four groups of updating primitives described in [58]:

- insert expressions = {`upd:insertInto`, `upd:insertIntoAsFirst`, `upd:insertIntoAsLast`, `upd:insertBefore`, `upd:insertAfter`, `upd:insertAttributes`};
- delete expressions = {`upd:delete`};
- rename expressions = {`upd:rename`};
- replace expressions = {`upd:replaceNode`, `upd:replaceValue`, `upd:replaceElementContent`}.

For our purposes here, we abstract from these different groups and consider them as single update actions, denoted $\mathcal{A}_s$. We denote $\mathcal{A}$ the set of all *update actions*. Composite update actions, denoted $\mathcal{A}_c$, are calls to an *updating function*, which itself can perform one or more update actions $\in \mathcal{A}$. We have $\mathcal{A} \equiv \mathcal{A}_s \cup \mathcal{A}_c$.

Distributed Pending Update Table Imagine that all update actions caused in a distributed update query are put in the correct deterministic update order, and attach to this global list Δ an additional peer column $\mathcal{P}$. The resulting table $\boxed{\mathcal{PA}}$ we call the Distributed Pending Update Table (DPUT). We should stress that this is a conceptual table only, we do not propose to materialise such a table in any way.

In Section 3.4 we described that when an updating XRPC query is started with isolation (i.e., following the semantics defined by $\mathcal{F}'_u$), each peer p keeps an isolated environment

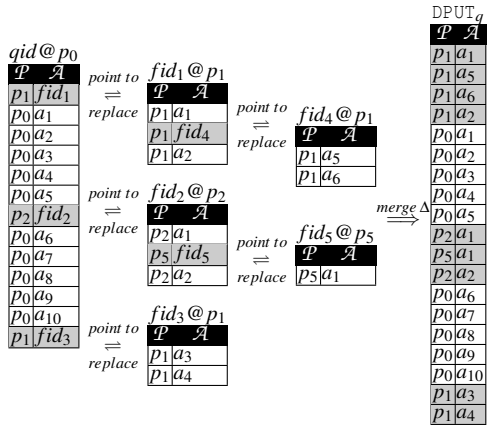


Figure 4.2: The conceptual Distributed Pending Update Table

$\langle db_q@p, \Delta_q@p\rangle$ around. Each XRPC function application $f_i(Params)@p$ causes a sublist $fid_i@p$ of pending update actions (just denoted Δ_f in rules $\mathcal{F}_u$ and $\mathcal{F}'_u$) that is merged into the overall list $\Delta_q@p$. We stress that $fid_i@p$ is just a conceptual list (not an implementation data structure) that represent the update actions caused at peer p by a single function call. The local list of pending updates at query site p_0 is denoted $qid@p_0$ here.

Corollary 4.4.1. *Iteratively substituting each $\langle p_x, fid_y\rangle$ in $qid@p_0$ by sublist $fid_y@p_x$, yields the DPUT in required order.*

Figure 4.2 shows $qid@p_0$ and all $fid_i@p_j$ caused by a single query, and the DPUT derived from those (the right-most table). In the lists, values a_i indicate single update actions, while the values fid_i points to another pending update sublist, that represents all update actions caused by the called function fid_i at the peer in column $\mathcal{P}$. The iterative substitution of the sublists in DPUT achieves the required *synchronous* semantics for remote function calls, as it inserts all update actions (recursively) caused by a function call in the DPUT at the point where the remote function was applied.

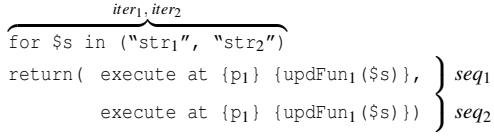
Body and Tags The XQUF restricts the locations in a query where update actions can be done. We abstract from the full XQuery syntax using the `body` concept, to define these places. `body` refers to the body of an updating XRPC query or the body of an updating XRPC function. The body grammar is shown below:

```
body ::= UpdateAction | "for" ... "return" body | body ("," body)*
```

A `body` can contain an expression in one of the three types, *(i)* an update action (possibly an XRPC updating function), *(ii)* a `for` expression which in turn contains a `body` in its `return` clause, or *(iii)* a sequence of one or more `body`s.

The tags in column $\mathcal{T}$ of the DPUT are concatenations of numbers, separated by a dot. We initialise $t_{prefix} = 1$ for executions done locally on behalf of the initiating query. The query body mimics the parse tree of the query, which is then "executed" recursively as follows (starting with b=root and $t_b = \emptyset$) to generate all tags:

- if b is a sequence constructor, we process all sequence expressions $s_1, \ldots, s_n$ while assigning $t_{s_i} = t_b.i$.



$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$
p1	updFun1(str_1)	1.1.1
p1	updFun1(str_1)	1.1.2
p1	updFun1(str_2)	1.2.1
p1	updFun1(str_2)	1.2.2

Figure 4.3: The body of the example query and its DPUT

$\Delta_q@p_1$

$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$	
"xrpc://y.example.org"	appendLog("Julie Connery")	1.1.1	$fid_1@p_1$
"xrpc://y.example.org"	appendLog("Sean Connery")	1.2.1	
$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$	$fid_2@p_1$
"xrpc://y.example.org"	appendLog("Julie Andrews")	1.1.2	
"xrpc://y.example.org"	appendLog("Sean Andrews")	1.2.2	

$\Downarrow sort_{\mathcal{T}}(\Delta_q@p_1)$

$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$
"xrpc://y.example.org"	appendLog("Julie Connery")	1.1.1
"xrpc://y.example.org"	appendLog("Julie Andrews")	1.1.2
"xrpc://y.example.org"	appendLog("Sean Connery")	1.2.1
"xrpc://y.example.org"	appendLog("Sean Andrews")	1.2.2

Figure 4.4: The pending update list $\Delta_q@p1$ was created by two XRPC calls executed after each other. Sorting those at commit time on $\mathcal{T}$ achieves deterministic update order.

- if b is a `for`-loop with iterations $1 \leq i \leq n$, we process each iteration of the body f with $t_f = t_b.i$.
- if b is an updating action, we put $tag = t_{prefix}.t_b$ in column $\mathcal{T}$ for all update actions it inserts in the pending update list.
- if b is an updating XRPC function, we also insert tag as an attribute of the `xrpc:call` in the XRPC request. The updating function body is executed remotely with initialisation $t_{prefix} = tag$.

Figure 4.3 shows how the tags are constructed from the body of our example update query. The initial t_{prefix} is 1. The `for`-loop with two iterations introduces the second number, 1 for the first iteration, and 2 for the second. Inside the loop body we find a sequence constructor, introducing a third number in the tag. Inside this sequence constructor, the update actions are found and tagged.

Note that the tag construction algorithm respects the `for`-loop and sequence construction order just like XQUF pending update list construction. Also, the tags generated by remote function applications are prefixed by the current tag and therefore must be bigger than all previous and smaller than all following locally generated tags, which mimics synchronous XRPC semantics. Therefore:

Corollary 4.4.2. *Column $\mathcal{T}$ in DPUT is ordered by definition.*

One should remember that the DPUT is only a concept used to define the required order, and there is no single place where we can afford to bring together the entire merged pending update list – each peer only has local information. But, if we could attach the correct tag values to the (partial) pending update lists $\Delta_q@p$ at each peer p in a $\mathcal{T}$ column, we can achieve correct update order by (stable) sorting the $\Delta_q@p$ on $\mathcal{T}$ locally at each peer at commit time.

XRPC SOAP Extension: Tag Attributes The tags are only constructed on demand, just before executing a Bulk RPC request. In local execution, the `iter` columns maintained by

MonetDB/XQuery for loop-lifting correspond with the iteration numbers in the tags. Thus by obtaining all `iter` numbers from the current scope through to the root level (by joining with so-called *map* relations [88]), the tags can be constructed whenever an update action needs to be executed. For sequence construction, these numbers are available in the Pathfinder XQuery Core parse tree, and can be inserted in the generated query plan. The tags are always prefixed by t_{prefix}, stored as a loop-lifted expression. The reconstructed tags are included as attributes in the `xrpc:call` elements in the Bulk SOAP XRPC request message. The remote peer uses this tag then as prefix for generating further tag numbers, as described before (i.e., as the loop-lifted t_{prefix} expression).

Below we show the first XRPC request message triggered by the RPC call in our example query, which leads to tags 1.1.1 and 1.2.1 (i.e., $t_{prefix}.iter_{\{1,2\}}.seq_1$):

```
<xrpc:request xrpc:module="filmdb" xrpc:method="appendLog" xrpc:arity="1"
              xrpc:location="http://x.example.org/film.xq" xrpc:updCall="false">
  <xrpc:queryID xrpc:host="x.example.org" xrpc:timestamp="32414232" xrpc:timeout="180"/>
  <xrpc:call xrpc:tag="1.1.1">    <!-- first call -->
    <xrpc:sequence>
      <xrpc:atomic-value xsi:type="xs:string">Julie Connery</xrpc:atomic-value>
    </xrpc:sequence>
  </xrpc:call>
  <xrpc:call xrpc:tag="1.2.1">    <!-- second call -->
    <xrpc:sequence>
      <xrpc:atomic-value xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
    </xrpc:sequence>
  </xrpc:call>
</xrpc:request>
```

The second function application leads to a similar XRPC request (logging actors with surname Andrews this time), with call tags 1.1.2 and 1.2.2 (not shown). Figure 4.4 shows the pending update list $\Delta_q@p_1$ at peer p_1 ("y.example.org") including the extra column $\mathcal{T}$. It depicts the situation at commit time. Both both XRPC requests have been executed successfully, and produced pending update sublists $fid_1@p_1$ and $fid_2@p_1$, which were concatenated in $\Delta_q@p_1$ as both executed with isolation semantics $\mathcal{F}'_u$ (note that the XRPC Request above includes the `queryID` element). Sorting the pending update list on $\mathcal{T}$ achieves the desired deterministic update order.

4.5 Distributed XRPC Transactions

XRPC allows XQUF [58] expressions to be executed on remote peers, by means of XRPC calls to updating functions, thus providing distributed transaction functionality. For such distributed updating queries, XRPC provides two different isolation levels, *no isolation* and *repeatable reads*, to meet the needs of different kinds of applications. The latter level provides *repeatable reads* for all XRPC requests to the same peer made in a single query and uses a distributed 2-Phase Commit (2PC) protocol to ensure atomic commit. The semantics of these levels have already been formally defined in Section 3.4.2, including necessary extensions to the basic SOAP XRPC protocol to support them. Each XRPC query can specify the desired isolation level using the XQuery `declare option` feature to set `xrpc:isolation` to `none` or `repeatable`. Here, we briefly explain how repeatable reads with atomic commit are supported for updating XRPC queries on different XQuery engines.

To ensure *repeatable reads*, during the execution of an updating XRPC query q_{up}, each participating peer maintains the same database state (i.e., all persistently stored XML documents) for the query. This can be done using systems that either use (lock-based) serialisation,

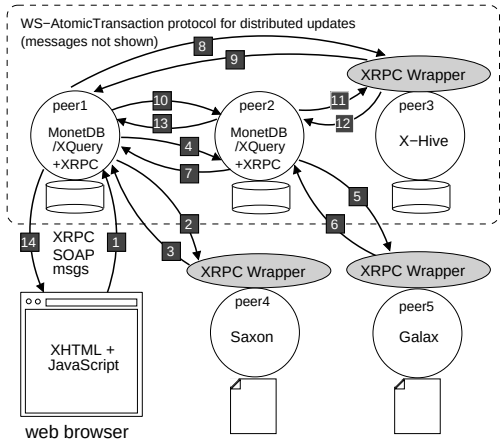


Figure 4.5: Browser-initiated XRPC query, visiting peers 2,3 twice; updating peers 1-3 with atomic commit (dashed box area).

snapshot isolation, or multi-version concurrency control. The XRPC update requests generated by a query are not applied immediately to the database state used by that query while it runs. Rather, these requests are collected, in correspondence with the XQUF formal definition of a *pending update list*, that grows while the query runs. When the update query decides to commit, all peers in the transaction effectuate all updates in this list.

To provide atomic distributed commit, we have chosen to use the SOAP-based 2PC industry standard WS-AtomicTransaction[55], which defines an API with functions such as `Prepare()` and `Commit()`. It is embedded in the WS-Coordinator framework [54] that allows registering a collection of peers that participate in a distributed transaction, and subsequently run a transaction protocol (in this case WS-AtomicTransaction) on those peers. In XRPC, peer p_q that starts the query q is the one that registers the participating peers at the WS Coordinator service and initiates the `Prepare` and `Commit` phases. For this registration task, it thus needs to know a full list of peers that participated in the transaction. Due to nested XRPC calls (i.e., remote functions calling in turn other remote functions), the query originator may not be aware of all peers involved and therefore we extended the SOAP XRPC protocol to piggyback a list of unique participating peers in their response messages.

To provide updating XRPC queries with *repeatable reads* and atomic commit, XRPC systems must implement these web service 2PC interfaces and offer them over the same HTTP SOAP server that runs XRPC (this is the case in MonetDB/XQuery).

4.5.1 Heterogeneous Distributed 2PC

To enable *heterogeneous* distributed transactions, that is, performing XQUF updates on multiple peers that run different XQuery engines, the WS-AtomicTransaction 2PC interfaces are implemented in our XRPC Wrapper. This involves extending the XRPC Wrapper with concurrency control, XRPC message logging, and recovery functionality, that is used on top of the transactional capabilities of the underlying XQuery engine.

To provide the *repeatable reads* isolation level, the underlying XQuery engine must provide repeatable read consistency or better, and support multi-query transactions (with explicit start-transaction and commit/abort commands). For read-only queries under repeatable reads,

the XRPC Wrapper keeps a separate client connection open to the XQuery engine in which all XQuery requests with the same query ID are executed. This connection is kept open for the timeout period as specified in the XRPC requests. The XRPC Wrapper also keeps a log of recently expired query IDs (and an in-memory hash-table for fast lookups) such that it can properly generate error message for late requests. Note that query IDs contain a global timestamp, on which a reasonable maximum timeout can be enforced, so the size of the hash table should remain limited.

Updating queries can generate XRPC requests to both normal (read-only) XQuery functions as well as *updating* functions as defined by the XQUF, and are processed as follows.

1. When an XRPC request is received:
   a) check the query ID *id* carried by the request to see if a connection C_{id} for this query has already been created, and if not, create a new one, starting a new transaction (as mentioned, an error is generated for expired IDs). Also, a new subdirectory D_{id} is created in the logging directory of the XRPC Wrapper;
   b) if the called function is a read-only function, execute it using the underlying XQuery engine and send its result back to the caller[1]. If the execution fails, add the query ID to the expired query log and remove D_{id};
   c) otherwise, save the XRPC request message to the logging subdirectory D_{id} and send a response message to the caller to indicate success *without* actually executing the updating function (this is possible, as updating XQuery functions do not return a result). The rationale is that in order to provide repeatable reads, we must execute all updates together, at the end of the transaction; otherwise their effects would be visible for subsequent requests belonging to the same transaction.
2. When a `Prepare` request with ID *id* is received, then:
   a) if ID is expired, send `Aborted` to the coordinator;
   b) otherwise, if there are no request messages saved in the logging directory D_{id}, send `ReadOnly` to the coordinator, and then remove the logging directory D_{id}[2];
   c) otherwise, construct a *single* query containing *all* updating requests that have been saved so far (by using XQuery sequence construction). Execute the query in connection C_{id}, *without* committing the transaction yet. If this update query fails, add the query ID to the expired query log and remove D_{id}. Finally, send the decision `Committed` or `Aborted` to the coordinator (depending on the update success).
3. When a `Rollback` or a `Commit` request with ID *id* is received:
   a) if the request is `Commit`, log a "committing message" to D_{id}, and commit the transaction in C_{id}; The XRPC Wrapper should cease operation if committing in C_{id} fails, and then try to restart the underlying XQuery engine and/or itself, entering recovery mode;
   b) add the query ID to the expired query log and remove D_{id}.

[1] Note that, to reduce possible communication time with the coordinators needed by the recover procedure, each message should be logged before it is sent.

[2] Upon receipt of a `ReadOnly` notification, the coordinator knows that the participant votes to commit the transaction and has forgotten the transaction.

Thus, the XRPC Wrapper plays the game of declaring a distributed transaction committed, before actually committing in the underlying XQuery engine, relying on its own logging to do so at the global commit point.

Recovery is done every time the XRPC Wrapper starts, before it accepting any new XRPC requests. During recovery, the logging directory is scanned for unfinished transactions, i.e., for subdirectories containing messages of unfinished transactions. For each subdirectory, if no final decision can be deduced from the logs (message logs and expired query ID log), it is requested from the coordinator. Transactions that should be committed are then re-executed (Step 3).

The worst possible case is finding a "committing" message. As it may happen that the underlying XQuery engine committed but the XRPC Wrapper crashed before removing the D_{id} directory, re-trying the commit runs the risk of executing its updates twice. This risk can be mitigated by inspecting the log of the underlying XQuery engine (if accessible).

4.6 MonetDB/XQuery*

MonetDB/XQuery provides generic XQuery functionality, and its distributed querying and update facilities can be used in widely varying environments. First, we show how the mechanism described so far, can be useful in LAN environments with a limited number of nodes. When considering WAN applications with potentially thousands or more participating peers (such as StreetTiVo), we propose to use Distributed Hash Table (DHT) data structures under the hood of the system.

In the following, we will show how these widely varying application areas can be addressed by the `fn:doc()` and `fn:put()` built-in functions plus our XRPC `execute at` language construct.

4.6.1 Simple Scenarios

Our XRPC extension for the XQuery language enables a query shipping model to query and manipulate remote XML documents. Given our choice for SOAP over HTTP as the network protocol for XRPC, it is interesting to note that the `execute at` construct, when combined with `fn:doc()` and `fn:put()`, provides an implementation of HTTP-based data shipping, as shown by the following rewriting rules:

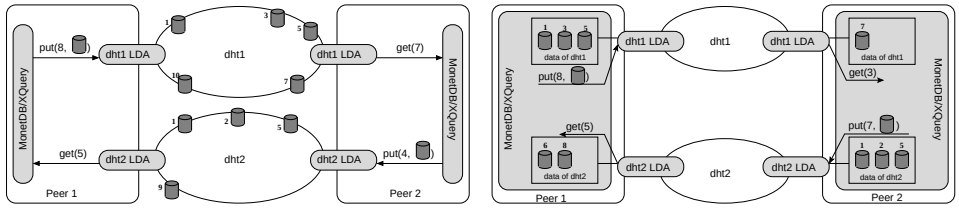
$$\frac{\begin{array}{c}StatEnv.baseURI \Leftarrow \emptyset \\ \texttt{execute at } \{\text{"xrpc://host"}\}\{\texttt{fn:put}(\$node, \text{"localname"})\}\end{array}}{\texttt{fn:put}(\$node, \text{"xrpc://host/localname"})} \quad (\mathcal{R}_{put_1})$$

$$\frac{\begin{array}{c}StatEnv.baseURI \Leftarrow \emptyset \\ \texttt{execute at } \{\text{"xrpc://host"}\}\{\texttt{fn:doc}(\text{"localname"})\}\end{array}}{\texttt{fn:doc}(\$node, \text{"xrpc://host/localname"})} \quad (\mathcal{R}_{doc_1})$$

Thus, an XQuery system with XRPC can implement the HTTP protocol in `fn:doc()`, `fn:put()` internally by using XRPC to execute those requests remotely with the local part of the URI (and an empty "base-URI", from the static environment [67]).

4.6.2 Loose DHT Coupling

A Distributed Hash Table [147, 2] provides *(i)* robust connectivity (i.e., tries to prevent network partitioning), *(ii)* high data availability (i.e., prevent data loss if a peer goes down by automatic replication), and *(iii)* a scalable (key,value) storage mechanism with $O(log(N))$



(a) Loose DHT/DBMS Coupling (b) Tight DHT/DBMS Coupling

Figure 4.6: MonetDB/XQuery with multiple DHT connections

cost complexity (where N is the amount of peers in the network). A number of P2P database prototypes have already used DHTs [42, 43, 100, 101, 108, 142]. An important design question is how a DHT should be exploited by an XQuery processor, and if and how the DHT functionality should surface in the query language.

We propose here to avoid any additional language extensions, but rather introduce a new `dht://` network protocol, accepted in the destination URI of `fn:doc()`, `fn:put()` and `execute at`. The generic form of such URIs is `dht://`*dht_id*`/`*key*. The `dht://` indicates the network protocol. The second part, *dht_id*, indicates the DHT network to be used. Such an ID is useful to allow a P2P XDBMS to participate in multiple (logical) DHTs simultaneously, as shown in Figure 4.6(a). The third part (*key*) is used to store and retrieve values in the DHT.

The simplest architecture to couple a DHT network with a DBMS is to just use the DHT API, the `put(key,value)` and `get(key):value` functions, to implement the XQuery data shipping functions `fn:put()` and `fn:doc()`, as shown in rules $\mathcal{R}_{put_2}$ and $\mathcal{R}_{doc_2}$:

$$\frac{\begin{array}{c} p_i = dht\_hash_{dht\_id}(key) \\ dht\_send_{p_0 \to p_i} request(\text{"}\texttt{put}\text{"}, (key, \$node)) \\ dht\_store@p_i \vdash \texttt{put}(\$node)@p_i \Rightarrow dht\_store'@p_i \\ dht\_send_{p_i \to p_0} response() \end{array}}{db@p_0 \vdash \texttt{fn:put}(\$node, \texttt{dht://}dht\_id/key) \Rightarrow db@p_0} \quad (\mathcal{R}_{put_2})$$

$$\frac{\begin{array}{c} p_i = dht\_hash_{dht\_id}(key) \\ dht\_send_{p_0 \to p_i} request(\text{"}\texttt{get}\text{"}, (key)) \\ dht\_store@p_i \vdash \texttt{get}(dht\_id/key) \Rightarrow \$node, dht\_store@p_i \\ dht\_send_{p_i \to p_0} response(\$node) \end{array}}{db@p_0 \vdash \texttt{fn:doc}(\texttt{dht://}dht\_id/key) \Rightarrow \$node, db@p_0} \quad (\mathcal{R}_{doc_2})$$

That is, we simply use the DHT to store XML documents as string values. The rules indicate that at the remote peer p_i, only the peer's DHT storage is involved, hence, peer p_i does not even have to have a running MonetDB/XQuery* instance. Note that the XQuery function `fn:doc` is a read-only function, since the document retrieved by using this function is stored as a transient document.

In this architecture, we can run the DHT as a separate process called the Local DHT Agent (LDA). Each LDA is a process that is connected to one DHT *dht_id* (see Figure 4.6(a)). This process runs separately from the database server, such that we can use the DHT software without any modifications.

The `execute at` can be "simulated" as follows:

$$\frac{\begin{array}{c} StatEnv.baseURI \Leftarrow \texttt{dht://}dht\_id/prefix \\ db@p_0 \vdash f_r(ParamList) \Rightarrow val, db@p_0 \end{array}}{db@p_0 \vdash f_r(ParamList)@\texttt{dht://}dht\_id/prefix \Rightarrow val, db@p_0} \quad (\mathcal{R}_{xrpc_2})$$

Rule $\mathcal{R}_{xrpc_2}$ in fact just evaluates the function locally, by getting all documents with a relative

URI name from the DHT. This is achieved by setting the baseURI in the static environment to `dht://`*dht_id*`/`*prefix*. If the function body thus contains any `fn:doc()`, `fn:put()` on some relative URI *localname*, the rules $\mathcal{R}_{put_2}$ and $\mathcal{R}_{doc_2}$ specify that the document should be stored/retrieved into/from `dht://`*dht_id*`/`*prefix*`/`*localname*. One should note that the *prefix* may be empty.

While this approach allows zero-effort coupling of DHT technology with DBMS technology, we consider it nothing more than a workaround. Rule $\mathcal{R}_{xrpc_2}$ substitutes function shipping by data shipping, defeating the purpose of XRPC. In case of updates, we would need to modify the rule to store the modified documents using `put` back in the DHT, but such a two-step update is hard to be made atomic.

4.6.3 Tight DHT Coupling

In a tight coupling scenario, rather than keep XML as string blobs inside the DHT (in RAM), each DHT peer actually uses its local XDBMS to store the documents (see Figure 4.6(b)). To realise this, we need to extend the DHT API with a single new method:

$$\texttt{xrpc(}key\texttt{, }q\texttt{, }m\texttt{, }f_r\texttt{(}ParamList\texttt{))}\ \texttt{:}\ \ \texttt{item()*}$$

This new method allows the *request* in the below rule to be routed through the DHT (*dht_send*), to achieve the following semantics for XRPC calls to a "`dht://`" URI:

$$\frac{\begin{array}{c} p_i = dht\_hash_{dht\_id}(key) \\ dht\_send_{p_0 \to p_i} request(q, m, f_r, ParamList) \\ db@_q p_i \ \vdash\ f_r(ParamList)@p_i \Rightarrow val,\ db_q@p_i \\ dht\_send_{p_i \to p_0} response(val) \end{array}}{db_q@p_0 \ \vdash\ f_r(ParamList)@\texttt{dht://}dht\_id/key \Rightarrow val,\ db_q@p_0} \quad (\mathcal{R}_{xrpc_3})$$

This rule states that the DHT *dht_id* routes an XRPC request using the normal DHT routing mechanism towards the peer p_i responsible for *key*. When the Local DHT Agent (LDA) in p_i receives such a request, it performs an XRPC to the MonetDB/XQuery instance on the same peer p_i. This XRPC executed at remote location p_i from the LDA into MonetDB/XQuery (it may use either semantic $\mathcal{R}_{\mathcal{F}_r}$ or $\mathcal{R}'_{\mathcal{F}_r}$). The response is then transported back via the DHT towards the query originator p_0.

In this scenario, we can support `fn:doc()` and `fn:put()` by combining rule $\mathcal{R}_{xrpc_3}$ with $\mathcal{R}_{doc_1}$ and $\mathcal{R}_{put_1}$. That is, use an XRPC request routed via the DHT to do a remote execution of `fn:doc()`, `fn:put()` on the relative URI *localname*.

In the tight coupling, we have to extend the DHT implementation. A positive side-effect of this is that the DBMS gets access to to information internal to the P2P network. This information (e.g. peer resources, connectivity) can be exploited in query optimisation. Also, bulk XRPC requests routed over the DHT may be optimised (similar to Bulk RPC), by combining requests that follow the same route as long as possible in single network messages.

4.7 Conclusion

In this chapter, we have discussed various aspects of using XRPC in distributed XQuery processing. First we show that XRPC can be easily adopted by different XQuery engines, such that complex P2P communication patterns can be programmed using XRPC. To enhance adoption of XRPC, we described a XRPC wrapper that allows any XQuery data source to handle

XRPC calls[3]. During our Saxon experiments, we also saw that Bulk RPC enables set-oriented optimisations, such that Bulk RPC execution of a selection function can be handled using a join strategy.

Then, to better match the transaction semantics in databased, we define a deterministic update semantics for XQUF queries, and showed how the SOAP XRPC can be extended to guarantee deterministic order in distributed update scenarios. To provide atomic distributed commit, we have chosen to use the SOAP-based 2PC industry standard Web Services Atomic Transaction [55].

Finally, we discussed work on MonetDB/XQuery* that aims to create powerful P2P XML database technology that preserves the full XQuery language (+XQUF), extending it only with a single new construct, i.e., XRPC. We described how Distributed Hash Tables (DHTs) can be integrated without further XQuery extensions, by adding support for a new `dht://` protocol in URIs. We discussed the semantics of two ways of coupling (loose and tight) a DHT with an XDBMS, of which the latter is more powerful. In Chapter 7, we will show how this functionality can be used in the StreetTiVo collaborative video indexing application. Our next step in this area is to implement these couplings in MonetDB/XQuery using the Bamboo DHT [147], and perform experiments in environments like PlanetLab. Especially the tight coupling will open up a playing field for a number of query optimisation techniques that exploit the P2P network characteristics.

[3]XRPC and the XRPC wrapper are available in the open-source XDBMS MonetDB/XQuery (`http://monetdb.cwi.nl`).

5 XQuery Decomposition

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

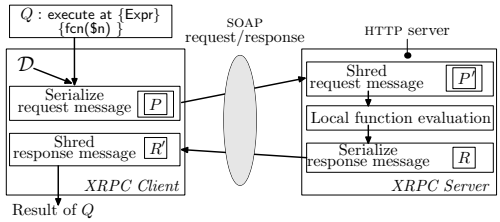


Figure 5.1: XQuery Remote Procedure Call under pass-by-value

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 `fcn()` with a single parameter (a node `$n` from some document $\mathcal{D}$). To make an XRPC call, the local

```
 1 declare function makenodes() as node()
 2 {⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩/b};                          ▷ node ⟨b⟩⟨c/⟩⟨/b⟩ has parent::a
 3 declare function overlap($l as node(), $r as node()) as boolean
 4 {not(empty($l//* intersect $r//*))};              ▷ are $l and $r related?
 5 declare function earlier($l as node(), $r as node()) as node()
 6 {if ($l≪$r) then $l else $r};
 7 let $bc := makenodes(), $abc:=$bc/parent::a       ▷ $bc has a parent $abc
 8 return (for $node in ($bc, $abc)
 9         let $first := earlier($bc, $abc)                  ▷ always $abc
10         where overlap($first, $node)                   ▷ always overlap
11         return $node)//c                            ▷ returns only one ⟨c/⟩
```

Table 5.1: Example query Q_1

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 `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, $Q(\mathcal{D})$ denotes the result of evaluating Q over a (possibly distributed) database $\mathcal{D}$. Two queries Q and Q' are *equivalent*, if $Q(\mathcal{D}) = Q'(\mathcal{D})$ for any given database $\mathcal{D}$ (under the XQuery *deep-equal* semantics).

We illustrate XQuery decomposition as follows:

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

the URL `xrpc://example.org/depts.xml` implies that the remote peer `example.org` supports XRPC, so the predicates could be pushed as:

```
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 `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:

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

here, `$abc` evaluates to the empty sequence, instead of the correct *a*-node ⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩.

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:

```
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:

```
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

```
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.

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[2]. Class 2 is dealt with by adding these properties as attributes in the XRPC nodes (such as `xrpc:element`) that enclose serialised 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.

[2]If `static-base-uri()` is not set, we ship the value `xrpc://`$\mathcal{P}$`/doc/`, so that `fn:doc()` calls with a relative document URI call back to the originating peer $\mathcal{P}$.

Expr	::=	*"()" | ExprSingle | ExprSeq*
ExprSeq	::=	*"(" ExprSingle ("," ExprSingle)∗ ")"*
ExprSingle	::=	*Literal | VarRef | ForExpr | LetExpr | IfExpr | Typeswitch | CompExpr | OrderExpr | NodeSetExpr | Constructor | StepExpr | FunCall | TransformExpr | UpdExpr*
VarRef	::=	*"$"Var*
Var	::=	*"$"QName*
ForExpr	::=	*"for" Var "in" Expr "return" Expr*
LetExpr	::=	*"let" Var ":=" Expr "return" Expr*
IfExpr	::=	*"if" "(" Expr ")" ThenElse*
ThenElse	::=	*"then" Expr "else" Expr*
Typeswitch	::=	*"typeswitch" "("Expr")" CaseClause+ "default" Var "return" Expr*
CaseClause	::=	*"case" Var "as" SequenceType "return" Expr*
CompExpr	::=	*Expr (ValueComp | NodeCmp) Expr*
ValueComp	::=	*"=" | "!=" | "<" | "<=" | ">" | ">="*
NodeCmp	::=	*"is" | "≪" | "≫"*
OrderExpr	::=	*Expr "order by" OrderSpecs*
OrderSpecs	::=	*Expr ("ascending" | "descending")(, OrderSpecs) ∗*
NodeSetExpr	::=	*Expr NodeSetOp Expr*
NodeSetOp	::=	*"union" | "intersect" | "except"*
Constructor	::=	*("document" | "text") "{" Expr "}" ("element"|"attribute") (QName |"{"Expr"}")"{"Expr"}"*
StepExpr	::=	*"/" | AxisStep "::" NodeTest*
AxisStep	::=	*RevAxis | FwdAxis | HorAxis*
RevAxis	::=	*"ancestor" | "ancestor-or-self" | "parent"*
FwdAxis	::=	*"self" | "child" | "attribute" | "descendant" | "descendant-or-self"*
HorAxis	::=	*"preceding" | "preceding-sibling" | "following" | "following-sibling"*
NodeTest	::=	*"node()" | "text()" | QName | "∗"*
FunCall	::=	*QName "(" (Expr ("," Expr) ∗)? ")"*
TransformExpr	::=	*"copy" Var ":=" SourceExpr ("," Var ":=" SourceExpr)* "modify" ExprSingle "return" ExprSingle*
UpdExpr	::=	*InsertExpr | DeleteExpr | RenameExpr | ReplaceExpr*
InsertExpr	::=	*"insert nodes" SourceExpr InsTgtChoice TargetExpr*
InsTgtChoice	::=	*(("as" ("first" | "last"))? "into") | "after" | "before"*
DeleteExpr	::=	*"delete nodes" TargetExpr*
RenameExpr	::=	*"rename node" TargetExpr "as" SourceExpr*
ReplaceExpr	::=	*ReplaceNode | ReplaceValue*
ReplaceNode	::=	*"replace node" TargetExpr "with" SourceExpr*
ReplaceValue	::=	*"replace value of node" TargetExpr "with" SourceExpr*
SourceExpr	::=	*ExprSingle*
TargetExpr	::=	*ExprSingle*

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]:

```
XRPCExpr   ::= "execute" "at" "{"ExprSingle"}" "function" XRPCParam "{"Expr"}"
XRPCParam  ::= "()" | "(" "$"Var ":=" VarRef ("," XRPCParam)?")"
```

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-

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                                                    Q2
```

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      Q2c
```

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      Q2n
```

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*:rule[val]*, where v_i is a unique vertex identifier, *rule* is the grammar rule represented by v_i, and *val* is an optional value indicating the right-hand-side of *rule*. There is a single v_{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 $v_{22},\cdots,v_{39}$ 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 $x \overset{p}{\rightsquigarrow} y$, if y is reachable from x via *only* parse edges; `(2)` *x "varref-depends on" y*, denoted as $x \overset{v}{\rightsquigarrow} y$, if y is reachable from x via at least one varref edge; and `(3)` *x "depends on" y*, denoted as $x \rightsquigarrow y$, if either $x \overset{p}{\rightsquigarrow} y$ or $x \overset{v}{\rightsquigarrow} y$ holds. The compositional nature of XQuery means that $x \rightsquigarrow y$ concisely captures all semantic dependencies between subexpressions.

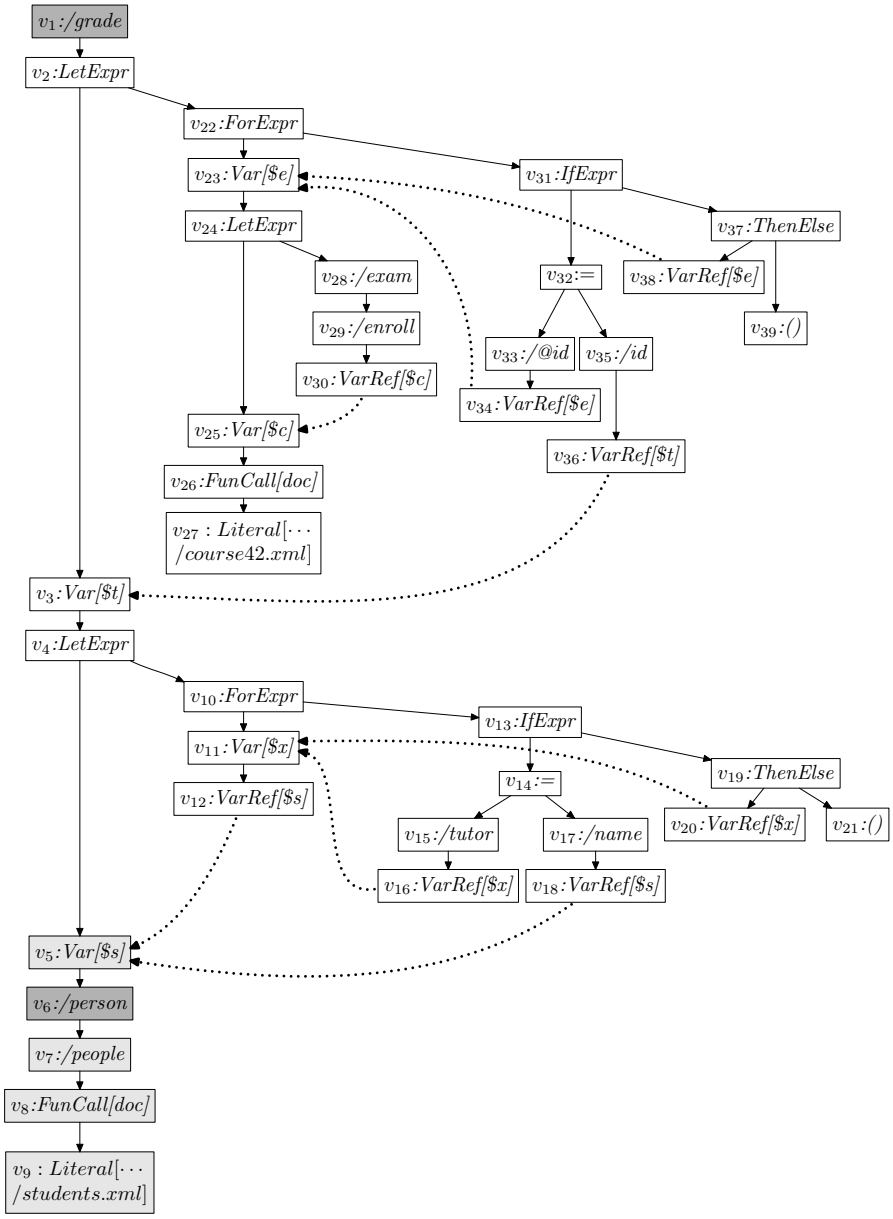


Figure 5.2: *d*-graph of the normalised XCore variant Q_2^n in Table 5.3

Consider Figure 5.2, $v_{15} \overset{p}{\rightsquigarrow} v_{16}$, since (v_{15}, v_{16}) is a parse edge; $v_{15} \overset{v}{\rightsquigarrow} v_{11}$, as v_{11} is reachable from v_{15} via $(v_{15}, v_{16}), (v_{16}, v_{11})$ and (v_{16}, v_{11}) is a varref edge.

For a d-graph G and a vertex $r_s \in V(G)$, 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 $r_s \overset{p}{\rightsquigarrow} u$; r_s is called the *root* of the subgraph. For instance, the subgraph rooted at vertex v_{22} contains vertices $v_{22}, \cdots v_{39}$, 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

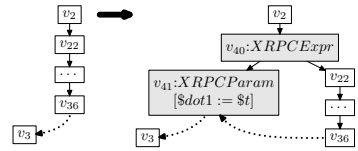


Figure 5.3: `XRPCExpr` insertion

We can decide to evaluate a certain subgraph G_{r_s} rooted at r_s remotely over XRPC, by inserting a v_x:`XRPCExpr` node above it. This may only be done if we can ensure that the result of the rewritten query is identical 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}[3] that resolve to variable bindings outside G_{r_s}:

1) Insert a vertex v_x:`XRPCExpr`, a parse edge (v_x, r_s), and replace each incoming edge (v_{in}, r_s) with a new edge (v_{in}, v_x)[4].
2) For each outgoing varref edge from vertex $v_i \in V(G_{r_s})$ to $v_j \in V(G) \backslash V(G_{r_s})$, where edge $(v_i, v_j) \in E_v(G)$ is a varref edge as (v_i:`VarRef`[$*qname*], v_j:`Var`[$*qname*]), we insert a new vertex v_k, a new parse edge (v_x, v_k) 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:`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:`XRPCParam`[()] (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:

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

[4] 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}.

i. $\nexists n \in V(G) : n.rule \in \{\texttt{RevAxis, HorAxis}\} \wedge (\texttt{useResult}(n, r_s) \vee \texttt{useParam}(r_s, n))$;

ii. $\nexists n \in V(G) : n.rule \in \{\texttt{NodeCmp, NodeSetExpr}\} \wedge ((r_s.rule \in \{\texttt{NodeCmp, NodeSetExpr}\} \wedge n \notin V(G_{r_s})) \vee \texttt{useResult}(n, r_s) \vee \texttt{useParam}(r_s, n))$;

iii. $\nexists n \in V(G), \exists m \in V(G) : n.rule = \texttt{AxisStep} \wedge m.rule \in \{\texttt{ForExpr, OrderExpr, ExprSeq, NodeSetExpr, AxisStep}\backslash\{\texttt{self, child, attribute}\}\} \wedge ((\texttt{useResult}(n, m) \wedge m \rightsquigarrow r_s) \vee (\texttt{useResult}(n, r_s) \wedge m \in V(G_{r_s})) \vee (m \notin V(G_{r_s}) \wedge r_s \overset{p}{\rightsquigarrow} n \overset{v}{\rightsquigarrow} m))$;

iv. $\nexists n \in V(G) : n.rule = \texttt{FunCall} \wedge n.val \in \{\texttt{fn:root(), fn:id(), fn:idref(), fn:lang()}\} \wedge (\texttt{useResult}(n, r_s) \vee \texttt{useParam}(r_s, n))$.

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:

$$\begin{aligned} \texttt{useResult}(n, r_s) &\Leftrightarrow n \rightsquigarrow r_s \\ \texttt{useParam}(r_s, n) &\Leftrightarrow n \in V(G_{r_s}), \exists v \in V(G) \backslash V(G_{r_s}) : n \rightsquigarrow v \end{aligned}$$

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 ii 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^n (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* `ForExpr`*s) 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 URIs that are used as parameters of `fn:doc()` in vertices that the vertex v_x can reach via parse edges:

$$\begin{aligned} D(v_x) = \{uri :: v_y | \{v_y, v_z\} \in E(G) : v_x \overset{p}{\rightsquigarrow} v_y \wedge v_y.rule = \texttt{FunApp} \wedge v_y.val = \text{“}\texttt{doc}\text{”} \wedge \\ ((v_z.rule = \texttt{Literal} \wedge uri = v_z.val) \vee (v_z.rule \neq \texttt{Literal} \wedge uri = \text{“}*\text{”}))\} \end{aligned}$$

We tag each *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

Q_2^v: decomposed Q_2^n under pass-by-value

```
declare function fcn1() as node()*
{doc("xrpc://A/students.xml")/child::people/child::person};

declare function fcn0() as node()*
{(let $t := let $s := execute at{'A'} {fcn1()}
            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};

execute at { ...} {fcn0()}
```

Q_2^f: decomposed Q_2^n under pass-by-fragment

```
declare function fcn1() as node()*
{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 ()};

declare function fcn2($para1 as node()) as node()*
{for $e in (let $c := doc("xrpc://B/course42.xml") return $c/child::enroll/child::exam)
 return if ($e/attribute::id = $para1/child::id) then $e else ()};

declare function fcn0() as node()*
{let $t := execute at {'A'} {fcn1()} return (execute at {'B'} {fcn2($t)}) / child::grade};

execute at { ...} {fcn0()}
```

Applying Distributed Code Motion in Q_2^f

```
declare function fcn2new($para2 as xs::string*) as node()*
{for $e in (let ...return ...)  return if ($e/attribute::id = $para2) then $e else ()};

declare function fcn0() as node()*
{let $t := execute at {'A'} {fcn1()}
 return let $l := $t return (execute at {'B'} {fcn2new($l/child::id)})/child::grade};
```

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(`v_i`::`v_i`)`.

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[5], *(b)* contain at least one `fn:doc()` and *(c)* execute at least one XPath step on the `fn:doc()` function:

$$I'(G) = \{v_x | v_x \in I(G) : \nexists v_y : v_y \overset{p}{\rightsquigarrow} v_x \wedge D(v_x) = D(v_y) \wedge \\ \exists v_z : v_x \overset{p}{\rightsquigarrow} v_z \wedge v_z.\mathit{rule} = \texttt{AxisStep} \wedge \exists \texttt{xrpc://}\mathit{uri} \in D(v_x)\}$$

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 v_5 and v_{25} correspond to two different equivalence classes $D(v_5) = \{$`xrpc://A/students.xml`$::v_9\}$ and $D(v_{25}) = \{$`xrpc://B/course42.xml`$::v_{27}\}$. However, v_{25} is not a valid insertion point. The vertices in $I'(G)$ (coloured dark grey) are v_6 (the highest non* `Var` *vertex in the subtree rooted at v_5) and the root v_1. Thus, $I'(G) = \{v_1, v_6\}$.*

[5] If 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_2^c (Table 5.3) can be normalised to Q_2^n (Table 5.3), which can thus be rewritten as Q_2^v in Table 5.4.

The main achievement of normalisation in the above case is to relate the call to `doc("../course42.xml")` through parse-edges (directly calling $\$c$ in Q_2^n), instead of varref edges (referencing $\$c$ in Q_2^c), 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_2^f (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* `$para1/child::id` *only depends on the function parameter* `$para1`. *Shipping full* `person` *nodes* `$para1` *from peer* `A` *to* `B`, *only to extract the string value of its* `id` *child at* `B`, *may waste bandwidth, especially if* `person` *carries much more data than just an* `id`. *Instead, it would be better to extract the string value of* `id` *at peer A and only ship the strings. This optimisation can be realised by adding a new parameter* `$para2` *to the function, and substituting* `$para1/child::id` *in the body with it. In the function* `fcn0()` *that calls* `fcn2new()`, *we save the original function parameter* `$t` *in a new* `let`*-binding* `$l`, *and pass* `$l` *instead of* `$t`. *The additional function parameter is passed as* `$l/child::id`. *Finally, the affected function parameter* `$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:[6]

```
$msg//fragment[$fragid]/descendant::node()[$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* $\langle b\rangle\langle c/\rangle\langle/b\rangle$ *is contained in the* `$abc` *fragment* $\langle a\rangle\langle b\rangle\langle c/\rangle\langle/b\rangle\langle/a\rangle$*. 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:*

```
$bc  := $msg:fragment[1]/descendant::node()[2],
$abc := $msg:fragment[1]/descendant::node()[1]
```

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.

[6] 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.

Excerpt from a request message with pass-by-value

```
⟨call⟩
  ⟨sequence⟩⟨element⟩⟨b⟩⟨c⟩⟨/b⟩⟨/element⟩⟨/sequence⟩
  ⟨sequence⟩⟨element⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/element⟩⟨/sequence⟩
⟨/call⟩
```

Excerpt from of pass-by-fragment message for `earlier($bc,$abc)`

```
⟨env:Envelope ...⟩
  ⟨env:Body⟩
    ⟨request⟩
      ⟨fragments⟩⟨fragment⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/fragment⟩⟨/fragments⟩
      ⟨call⟩
        ⟨sequence⟩⟨element fragid="1" nodeid="2"⟩⟨/sequence⟩
        ⟨sequence⟩⟨element fragid="1" nodeid="1"⟩⟨/sequence⟩
      ⟨/call⟩
    ⟨/request⟩
  ⟨/env:Body⟩
⟨/env:Envelope⟩
```

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.

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

I. $\nexists n \in V(G) : n.rule \in \{\texttt{RevAxis}, \texttt{HorAxis}\} \wedge (\texttt{useResult}(n, r_s) \vee \texttt{useParam}(r_s, n))$;

II. $\nexists n \in V(G) : n.rule \in \{\texttt{NodeCmp}, \texttt{NodeSetExpr}\} \wedge$
$((r_s.rule \in \{\texttt{NodeCmp}, \texttt{NodeSetExpr}\} \wedge n \notin V(G_{r_s}) \wedge \texttt{hasMatchingDoc}(n, r_s)) \vee$
$((\texttt{useResult}(n, r_s) \vee \texttt{useParam}(r_s, n)) \wedge \texttt{hasMatchingDoc}(n, n)))$;

III. $\nexists n \in V(G), \exists m \in V(G) : n.rule = \texttt{AxisStep} \wedge m.rule \in \{\texttt{ForExpr, ExprSeq, NodeSetExpr}\} \wedge$
$((\texttt{useResult}(n, m) \wedge m \rightsquigarrow r_s) \vee (\texttt{useResult}(n, r_s) \wedge m \in V(G_{r_s})) \vee$
$(m \in V(G) \backslash V(G_{r_s}) \wedge r_s \overset{p}{\rightsquigarrow} n \overset{v}{\rightsquigarrow} m)) \wedge \texttt{hasMatchingDoc}(m, m)$;

IV. $\nexists n \in V(G) : n.rule = \texttt{FunCall} \wedge n.val \in \{\texttt{fn:root(), fn:id(), fn:idref(), fn:lang()}\} \wedge$
$(\texttt{useResult}(n, r_s) \vee \texttt{useParam}(r_s, n))$.

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:

$$\texttt{hasMatchingDoc}(v_1, v_2) \Leftrightarrow \forall uri_l :: v_i \in D(v_1), \exists uri_r :: v_j \in D(v_2) : \\ v_i \neq v_j \wedge (uri_l = uri_r \vee uri_l = * \vee uri_r = *)$$

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

ProjectionPath	::=	**doc** "(" Literal "::" Literal ")" ("/" SimplePath)*
SimplePath	::=	*AxisStep* "::" *NodeTest* | *SimplePath* "/" *AxisStep* "::" *NodeTest*
AxisStep	::=	"self" | "child" | "attribute" | "descendant" | "descendant-or-self"
		| **"ancestor"** | **"ancestor-or-self"** | **"parent"** | **"prececding"** | **"prececding-sibling"**
		| **"following"** | **"following-sibling"** | **"root()"** | **"id()"** | **"idref()"**
NodeTest	::=	"node()" | "text()" | *QName* | "*"

Table 5.6: Grammar rule extension of *ProjectionPath* (bold)

`attribute`), 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 $\mathcal{D}$, to extract a minimal subdocument $\mathcal{D}'$ needed to execute Q such that $Q(\mathcal{D}) = Q(\mathcal{D}')$. 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 $\mathcal{P}$ of query Q from path analysis, a loading algorithm is applied to $\mathcal{P}$ and an XML document (from a file or a stream) $\mathcal{D}$. A projected XML document (or stream) $\mathcal{D}'$ 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()`,

Excerpt from of request message for `makenodes()`

```
⟨request⟩
  ⟨projection-paths⟩
    ⟨used-path/⟩
    ⟨returned-path⟩parent::a⟨/returned-path⟩
  ⟨/projection-paths⟩
  ⟨fragments/⟩
```

Excerpt from of response message for `makenodes()`

```
⟨env:Envelope ...⟩
  ⟨env:Body⟩
    ⟨response⟩
      ⟨fragments⟩⟨fragment⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/fragment⟩⟨/fragments⟩
      ⟨call⟩⟨sequence⟩⟨element fragid="1" nodeid="2"⟩⟨/sequence⟩⟨/call⟩
    ⟨/request⟩
  ⟨/env:Body⟩
⟨/env:Envelope⟩
```

Table 5.7: Pass-by-projection messages

`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, iff r_s satisfies all of the following conditions:

(a) $\nexists n \in V(G) : n.rule \in \{\texttt{NodeCmp}, \texttt{NodeSetExpr}\} \wedge$
$((r_s.rule \in \{\texttt{NodeCmp}, \texttt{NodeSetExpr}\} \wedge n \notin V(G_{r_s}) \wedge \texttt{hasMatchingDoc}(n, r_s)) \vee$
$((\texttt{useResult}(n, r_s) \vee \texttt{useParam}(r_s, n)) \wedge \texttt{hasMatchingDoc}(n, n)))$;

(b) $\nexists n \in V(G), \exists m \in V(G) : n.rule = \texttt{AxisStep} \wedge m.rule \in \{\texttt{ForExpr, ExprSeq, NodeSetExpr}\} \wedge$
$((\texttt{useResult}(n, m) \wedge m \rightsquigarrow r_s) \vee (\texttt{useResult}(n, r_s) \wedge m \in V(G_{r_s})) \vee$
$(m \in V(G) \backslash V(G_{r_s}) \wedge r_s \overset{p}{\rightsquigarrow} n \overset{v}{\rightsquigarrow} m)) \wedge \texttt{hasMatchingDoc}(m, m)$.

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 response 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):*

```
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.*

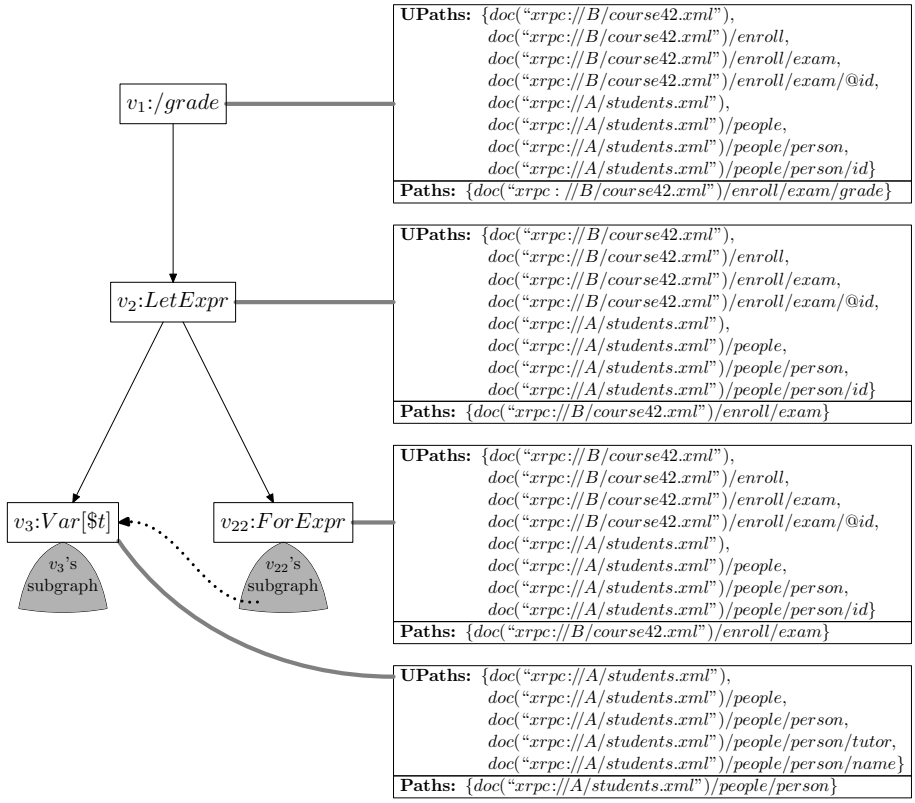


Figure 5.4: Path annotation example

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:

$$Env(v_i) \vdash Expr \Rightarrow Paths \texttt{ using } UPaths$$

The notation $Env(v_i)$ is used to identify the path annotation environment at a certain vertex v_i in the XQuery d-graph.

Path annotations are constructed bottom up by *path analysis rules* that derive the set of used (*UPaths*) and returned (*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 $v_i.UPaths$ and $v_i.Paths$ to refer to the path sets, with which the vertex v_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, $t, 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 `fn:doc(URI)`.

Path Analysis Rules We provide one rule for `fn:doc()` with a constant parameter and another for computed URIs:

$$\frac{()}{Env(v_i) \vdash \texttt{doc}(Literal_1) \Rightarrow \texttt{doc}(Literal_1 :: v_i) \texttt{ using } \emptyset} \quad (\text{DOC}_1)$$

$$\frac{Env(v_j) \vdash Expr_j \Rightarrow Paths_j \texttt{ using } UPaths_j}{\begin{array}{l} Env(v_i) \vdash \texttt{doc}(Expr_j) \Rightarrow \texttt{doc}(* :: v_i) \texttt{ using} \\ \qquad Paths_j \cup UPaths_j \cup Paths_j/\texttt{descendant::text()} \end{array}} \quad (\text{DOC}_2)$$

As mentioned in Section 5.4, in the definition of $D(v_x)$[7], we use a wildcard URI$*$ if the document name is an expression. Note that all paths start with `doc(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 `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 `fn:root()` is:

$$\frac{Env(v_j) \vdash Expr_j \Rightarrow Paths_j \texttt{ using } UPaths_j}{Env(v_i) \vdash fn{:}root(Expr_j) \Rightarrow \cup_{p \in Paths_j} p/\texttt{root() using } UPaths_j} \quad (\text{ROOT})$$

The built-in function `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 `fn:id()`/`fn:idref()`, are highly similar (only `fn:id()` provided):

$$\frac{\begin{array}{c} Env(v_j) \vdash Expr_j \Rightarrow Paths_j \texttt{ using } UPaths_j \\ Env(v_k) \vdash Expr_k \Rightarrow Paths_k \texttt{ using } UPaths_k \end{array}}{\begin{array}{l} Env(v_i) \vdash fn{:}id(Expr_j, Expr_k) \Rightarrow \cup_{p \in Paths_k} p/\texttt{id() using} \\ \qquad Paths_j \cup UPaths_j \cup Paths_k \cup UPaths_k \cup Paths_j/\texttt{descendant::text()} \end{array}} \quad (\text{ID})$$

The first parameter of `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 `fn:lang()` is:

[7] We use the `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.

$$\frac{\begin{array}{c} Env(v_j) \vdash Expr_j \Rightarrow Paths_j \texttt{ using } UPaths_j \\ Env(v_k) \vdash Expr_k \Rightarrow Paths_k \texttt{ using } UPaths_k \end{array}}{\begin{array}{l} Env(v_i) \vdash fn{:}lang(Expr_j, Expr_k) \Rightarrow () \texttt{ using } Paths_j \cup UPaths_j \cup Paths_k \cup \\ \quad UPaths_k \cup Paths_j\texttt{/descendant::text()} \cup Paths_k\texttt{/ancestor::*} \cup \\ \quad Paths_k\texttt{/ancestor-or-self::*/attribute::xml:lang} \end{array}} \quad \text{(LANG)}$$

The built-in function `fn:lang()` tests whether the language of its first parameter $Expr_k$, as specified by `xml:lang` attributes, is the same as (or is a sublanguage of) the language specified by its second parameter $Expr_j$. The language of $Expr_k$ is determined by the value of the XPath expression: `(ancestor-or-self::*/attribute::xml:lang)[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 `fn:root()`, `fn:id()`, `fn:idref()` and `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 `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 *cur* of the XML document is an ancestor of the current projection node *proj* (line 5), *cur* is added to output $\mathcal{D}'$ and moved to the *next node* in document order. If a *proj* is found (line 8), *proj* is added to $\mathcal{D}'$; if this *proj* is a returned node, all its descendants are also appended. Then *cur* is moved to its *next following node* in the document. Otherwise, if the current projection node *proj* is not a descendant of *cur*, the subtree of *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 $O(log(|\mathcal{D}|))$). 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 $\mathcal{D}'$.

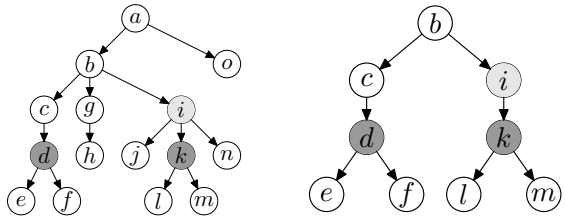
Algorithm 1: RUNTIMEXMLPROJECTION($U, R, \mathcal{D}$)

```
input  : U- used nodes
         R- returned nodes
         D- the original XML document
output : D'- the projection of U and R on D
1  projection nodes P ← sort(U∪R)          ▷ P is union of U and R sorted by document order
2  proj ← first node in P;
3  cur ← first node of D, i.e., root node;
4  while ¬P.end() do
5      if proj is a descendant of cur then
6          add cur to D';
7          cur ← next node in D;
8      else if proj = cur then
9          if proj is a returned node then
10             add cur and all descendants of cur to D';
11             cur ← next following node of cur in D;
12             while proj.next is a descendant of proj do
13                 proj ← proj.next                    ▷ prune projection nodes;
14             end
15         else
16             add cur to D';
17             cur ← next node in D;
18         end
19         proj ← proj.next                            ▷ next projection node;
20     else
21         cur ← next following node of cur in D;
22     end
23 end
24 cur ← root node of D';
25 while cur has only one child node ∧ cur ∉ {U ∪ R} do
26     cur ← first child of cur;
27 end
```

Example 5.6.3. *Consider an XML document $\mathcal{D}$ 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 $\mathcal{D}'$ of applying Algorithm 1 on U, R and $\mathcal{D}$.*

The algorithm starts with $P \leftarrow \{d,i,k\}$, $proj \leftarrow d$ and $cur \leftarrow a$. We traverse the tree using cur from a to d. Nodes a, b and c are added to $\mathcal{D}'$, since they are ancestors of the current context node d. Nodes d,e and f are also added to $\mathcal{D}'$, as d is a returned node. Then, 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 cur to i. Recall that i is a used node, thus only i is added to $\mathcal{D}'$. The last context node is k. Our current document node cur traverses from i to j, and then to k, where we can add nodes k, l and m to $\mathcal{D}'$. The traversal can be terminated, because there are no more context nodes to process. However, the intermediate result $\mathcal{D}'$ contains all *common ancestors of $\{d,i,k\}$. The post-processing removes node a from $\mathcal{D}'$, which produces the final projected document $\mathcal{D}'$ as shown in Figure 5.5(b).*

Relative Projection Paths At *compile time*, the XQuery compiler builds a query graph (*d*-graph) with root v_{root}, normalises it, and then does decomposition and code motion. For each inserted `XRPCExpr` v_{xrpc}, and for each `XRPCParam` parameter vertex v_{param}, it then extracts the relative paths:



(a) Original XML tree $\mathcal{D}$ (b) Projected tree $\mathcal{D}'$

Figure 5.5: Runtime XML projection example

$$
\begin{aligned}
U_{rel}(v_{xrpc}) &= \texttt{allSuffixes}(R(v_{xrpc}), U(v_{root})) \\
R_{rel}(v_{xrpc}) &= \texttt{allSuffixes}(R(v_{xrpc}), R(v_{root})) \\
U_{rel}(v_{param}) &= \texttt{allSuffixesVia}(R(v_{param}), U(v_{xrpc}), U(v_{root})) \\
R_{rel}(v_{param}) &= \texttt{allSuffixesVia}(R(v_{param}), R(v_{xrpc}), R(v_{root})), \text{ where:}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{allSuffixes}(Paths_i, Paths_j) = \{s_y | p_x/s_y \in Paths_j : \exists p_x \in Paths_i\} \\
&\texttt{allSuffixesVia}(Paths_i, Paths_j, Paths_k) = \\
&\qquad \{s_y/s_z \mid p_x/s_y/s_z \in Paths_k : \exists p_x/s_y \in Paths_j \wedge \exists p_x \in Paths_i\} \cup \\
&\qquad \{s_y \mid p_x/s_y \in Paths_k \wedge p_x/s_y \in Paths_j \wedge \exists p_x \in Paths_i\}
\end{aligned}
$$

At *runtime*, $\cup_{\forall v_{param}} U_{rel}(v_{param})$ and $\cup_{\forall v_{param}} R_{rel}(v_{param})$ are used to project the parameters in the outgoing XRPC request message. $U_{rel}(v_{xrpc})$ and $R_{rel}(v_{xrpc})$ 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_{param}, we need to take into account that (parts of) v_{param} could be returned by v_{xrpc}, and thus will be used by vertices depending on v_{xrpc}. Hence, in `allSuffixesVia()`, we not only find the relative paths that v_{xrpc} will apply on v_{param}, but also the relative paths that v_{root} will apply on v_{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 $\texttt{\$ctx}_{param}$ corresponding to v_{param} as context sequence into all suffix paths $s_i \in U_{rel}(v_{param})$ (resp. $R_{rel}(v_{param})$):

$$\texttt{union(\$ctx}_{param}\texttt{/}s_1\texttt{, union(\$ctx}_{param}\texttt{/}s_2\texttt{,...union(\$ctx}_{param}\texttt{/}s_{n-1}\texttt{, \$ctx}_{param}\texttt{/}s_n\texttt{)...))}$$

Paths $\texttt{\$ctx}/path_i/\texttt{root()}/path_j$ with function `root()` are executed as $\texttt{root(\$ctx)}/path_j$. Similarly, $\texttt{\$ctx}/path_i/\texttt{id()}/path_j$ is executed as $\texttt{root(\$ctx)}\,//\texttt{attribute()::}(a_1|..|a_n)/../path_j$, where $a_1,..,a_n$ 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 $U_{rel}(v_{xrpc})$ and $R_{rel}(v_{xrpc})$ using the result sequence of the function as $\texttt{\$ctx}_{xrpc}$ 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 a_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_{xrpc} is used as input to `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://`$\mathcal{P}/\mathcal{D}$ 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 `UpdExpr`s, denoted V_u, must be executed on the same peer that opened the document using `fn:doc()`. This, in turn, enforces that all expressions V_{a_i} (except `TransformExpr`) which depend on a $v_{u_i} \in V_u$, must be executed on the local peer. This is because V_{a_i} could only parse-depend on a v_{u_i}, as updating expressions are not allowed in a variable binding. Decomposing an expression in V_{a_i} would cause the v_{u_i} to be executed on a remote peer. To correctly identify the target nodes of an `UpdExpr`, all expressions V_{t_i} that produce target nodes for a v_{u_i}, must also be executed on the local peer. When decomposing an updating query, the vertices V_u, V_{t_i}, and V_{a_i} 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_{r_s} of G rooted at vertex r_s, under any semantics, r_s is in the set $I(G)$ of valid decomposition points, iff r_s *also* satisfies all of the following conditions:

(a) $r_s.rule \notin \{\texttt{UpdExpr}, \texttt{TargetExpr}\}$

(b) $\nexists v_u \in V(G) : v_u.rule \in \{\texttt{UpdExpr}\} \wedge r_s.rule \neq \texttt{TransformExpr} \wedge r_s \rightsquigarrow v_u \wedge$
$(\nexists v_m \in V(G) : v_m.rule = \texttt{TransformExpr} \wedge r_s \rightsquigarrow v_m \rightsquigarrow v_u)$

(c) $\nexists v_t \in V(G) : v_t.rule = \texttt{TargetExpr} \wedge v_t \rightsquigarrow r_s \wedge$
$(\exists p_t \in v_t.Paths \wedge \exists p_s \in r_s.Paths \wedge \texttt{starts-with}(p_t, p_s)$

Condition a avoids decomposing any `UpdExpr` and `Target-Expr`. Condition b states that if r_s is not a `TransformExpr`, r_s may not depend on an `UpdExpr`, unless the `UpdExpr` is a subexpression of a `TransformExpr`, on which r_s depends. Condition c states that r_s may not be decomposed, if r_s produces target nodes of an `UpdExpr`. We say r_s *produces target nodes*, iff a returned path p_s of r_s is a prefix of a returned path p_t of v_t, i.e., nodes returned by r_s 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 `UpdExpr`s. 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 $\mathcal{P}$, $\mathcal{P}$ becomes the "local peer" for all new nodes created by this `TransformExpr`. With condition a, we prevent `UpdExpr`s in the `modify` clause of a `TransformExpr` from being separated from the `TransformExpr` (i.e., executed on another peer than $\mathcal{P}$). Thus, the `UpdExpr`s in the `modify` clause will also be executed on $\mathcal{P}$, which is the local peer of their target nodes. This confirms the XQUF semantics that `UpdExpr`s may only be applied to local nodes. In the remainder of

this section, we continue our discussion on processing `UpdExpr`s 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:

$$\begin{aligned} D_u(Q_u) = \{(uri) | \exists v_t, v_y, v_z \in V(G_u) : v_t \rightsquigarrow v_y \wedge \{v_y, v_z\} \in E(G_u) \wedge \\ v_y.rule = \texttt{FunApp} \wedge v_y.val = \text{"}\texttt{doc}\text{"} \wedge v_z.rule = \texttt{Literal} \wedge uri = v_z.val \wedge \\ v_t.rule = \texttt{TargetExpr} \wedge \exists p \in v_t.Paths \wedge \texttt{fn:starts-with}(p, uri)\} \end{aligned}$$

$D^r_u(Q_u)$ is a subset of $D_u(Q_u)$, which contains the *affected remote documents*: $\forall d^r_i \in D^r_u(Q_u)$: $\texttt{starts-with}(d^r_i.uri, \text{"}\texttt{xrpc://}\text{"})$. The auxiliary functions `host()` and `path()` extract the peer identifier $\mathcal{P}$ and the document name $\mathcal{D}$ from an XRPC URI "`xrpc://`$\mathcal{P}/\mathcal{D}$", respectively. Each query operates in 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[9]:

$$\frac{\begin{array}{l} \forall d^r_x \in D^r_u(Q_u) : \texttt{fn:doc}(d^r_x.uri) \Rightarrow D^{r'}_u(Q_u) \\ \qquad db^{p_0}, D^{r'}_u(Q_u) \vdash Q_u \Rightarrow \Delta; \\ \qquad db^{p_0}, D^{r'}_u(Q_u) \vdash \texttt{upd:applyUpdates}(\Delta) \Rightarrow db'^{p_0}, D^{r''}_u(Q_u); \\ \forall d^{r''}_x \in D^{r''}_u(Q_u) : \texttt{fn:put}(d^{r''}_x.node, d^{r''}_x.uri) \Rightarrow (), db^{\texttt{host}(d^{r''}_x.uri)}; \end{array}}{db^{p_0} \vdash Q_u \Rightarrow (), db^{p_0}} \quad (R^u)$$

The rule 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 $D^r_u(Q_u)$, which could potentially be affected by Q_u, to p_0[10]. This yields a set of *local copies* $D^{r'}_u(Q_u)$ of $D^r_u(Q_u)$. Note that

[9] We use the ';' sign to suggest an order in the evaluation of the premises.

[10] 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 $D^r_u(Q_u)$ on the fly.

this step does not change db^{p_0}, as the documents in $D_u^{r'}(Q_u)$ are transient documents. Then, Q_u is executed in db^{p_0} with the additional documents $D_u^{r'}(Q_u)$ which first yields a PUL Δ. Subsequently, `upd:applyUpdates()` is executed to apply all update primitives in Δ to the affected documents. Updates in Δ that should be applied on remote documents $D_u^r(Q_u)$ are applied on their local copies $D_u^{r'}(Q_u)$ instead. This step produces a new current database state db'^{p_0}, which could differ from db^{p_0} (if Δ contains updates on really local documents), and a set of changed local copies $D_u^{r''}(Q_u)$. Finally, an additional step is executed, which calls `fn:put()` to store each $d_i^{r''} \in D_u^{r''}(Q_u)$ on its hosting peer and overwrite the existing $d_i^r \in D_u^r(Q_u)$. This step also creates a new current remote database state $db^{host(d_x^{r''}.uri)}$ on each hosting peer. As the rule R^u only applies Δ at the end of query execution, updates are not visible for the same query, which confirms the XQUF semantics. Hence, if Δ only contains updates on a single document, this rule already provides atomic updates.

Isolation Levels Note that the - potentially multiple - `fn:put("xrpc://..")` 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 `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 serialisability), 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_{repeat}:

$$
\begin{array}{l}
D'_u(Q_u) = \varnothing; \\
\forall d_x \in D_u(Q_u) : p_x = \texttt{host}(d_x.uri); \\
\qquad\qquad send^{p_0 \to p_x} request(q_u, \text{"}\texttt{fn:doc}\text{"}, d_x.uri);\ t_{q_u}^{p_x} \geq t_{q_u}^{p_0}; \\
\qquad\qquad db^{p_x}(t_{q_u}^{p_x}) \vdash d_x.node = \texttt{fn:doc}(d_x.uri) \Rightarrow d'_x.node; \\
\qquad\qquad send^{p_x \to p_0} reply(q_u, d_x.uri, d'_x.node); \\
\qquad\qquad db^{p_0}(t_{q_u}^{p_0}) \vdash D'_u(Q_u) = D'_u(Q_u) + (d_x.uri, d'_x.node); \\
db^{p_0}(t_{q_u}^{p_0}), D'_u(Q_u) \vdash Q_u \Rightarrow \Delta; \\
db^{p_0}(t_{q_u}^{p_0}), D'_u(Q_u) \vdash \texttt{upd:applyUpdates}(\Delta) \Rightarrow D''_u(Q_u); \\
\forall d''_x \in D''_u(Q_u) : p_x = \texttt{host}(d''_x.uri); \\
\qquad\qquad send^{p_0 \to p_x} request(q_u, \text{PREPARE}, \text{"}\texttt{fn:put}\text{"}, d''_x.node, d''_x.uri); \\
\qquad\qquad db^{p_x}(t_{q_u}^{p_x}) \vdash \texttt{log}(\text{"}\texttt{fn:put}\text{"}, d''_x.node, d''_x.uri) \Rightarrow r; \\
\qquad\qquad send^{p_x \to p_0} reply(q_n, r); \\
\hline
db^{p_0}(t_{q_u}^{p_0}) \vdash Q_u \Rightarrow ()
\end{array}
\qquad (R^u_{repeat})
$$

There are several differences between this rule R^u_{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 $db^{p_i}(t_{q_u}^{p_i})$ to handle requests originating from the same query. Usually, $db^{p_i}(t_{q_u}^{p_i})$ is the current state of peer p_i at the time $t_{q_u}^{p_i}$, 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, `fn:put()` is not executed immediately on a remote peer p_x, instead, it is sent as a PREPARE request. The execution of `fn:put()` is first "prepared", yielding a decision r, which could be COMMIT or ABORT. The execution of `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 `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^u_{repeat} does not modify the database state $db^{p_0}(t_{q_u}^{p_0})$.

XQUF Rewrites Rather than using `fn:doc("xrpc://`$\mathcal{P}/\mathcal{D}$`")` followed by an additional `fn:put ("xrpc://`$\mathcal{P}/\mathcal{D}$`")` after `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 `UpdExpr`s, except *homogeneous updating expressions*. An `UpdExpr` v_u^h is *homogeneous*, iff all returned paths of its `TargetExpr` $v_{t_u}^h$ start with the same "`xrpc://`$\mathcal{P}$", 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 $\mathcal{P}$, because executing v_u^h on another peer than $\mathcal{P}$ 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 `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 $V_{t_u}^h$ 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 $V_{t_u}^h$.

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

$$\forall v_i \in V_{t_u}^h : v_w^h \rightsquigarrow v_i \wedge (\forall p_t \in v_i.Paths, \forall p_w \in v_w^h.paths : \texttt{fn:starts-with}(p_w, p_t))$$

and define v_s as:

$$(v_s \rightsquigarrow v_u^h \vee v_s = v_u^h) \wedge \forall v_i \in V_{t_u}^h : v_s \rightsquigarrow v_i \wedge \nexists v_x \in V(G_u) : v_s \rightsquigarrow v_x \wedge v_x \rightsquigarrow v_u^h \wedge \forall v_i \in V_{t_u}^h : v_x \rightsquigarrow v_i$$

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 `fn:put("xrpc://`$\mathcal{P}/\mathcal{D}$`")` at the end of query execution). For updating queries containing both push-able and not push-able `UpdExpr`s, however, there is an additional issue to deal with: we can only push an `UpdExpr` v_{u_0} if we can guarantee that no other `UpdExpr`s elsewhere in the query update nodes from the same documents (a *clash*), *or*, if another `UpdExpr` does, it can also be pushed. This is because all `UpdExpr`s that are not pushed will generate an `fn:put()` in the end, which would potentially overwrite the pushed updating actions or other `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_s above:

$$\nexists v_x \in V(G_u) : v_x.rule = \texttt{UpdExpr} \wedge \neg \texttt{isHomogen}(v_x) \wedge \\ \exists p_x \in v_x.Paths, \exists p_u \in v^h_{t_u}.Paths : \texttt{docPeer}(p_x) = \texttt{docPeer}(p_u)$$

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

$$\texttt{isHomogen}(v_x) \Leftrightarrow \exists v_w \in V(G_u) \wedge (v_x, v_w) \in E(G_u) \wedge v_w.rule = \texttt{TargetExpr} \wedge \\ \forall p_i, p_j \in v_w.Paths : \texttt{docPeer}(p_i) = \texttt{docPeer}(p_j)$$

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^n_2 (in Table 5.3) so that it conforms to the XMark schema as the following:

```
(let $t:= let $s:=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 := 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

```
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 := ..."`) and the second `for`-loop (`"for $e in ..."`), and execute them on `peer1` and `peer2` respectively. The variable `$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 `child::author` by `parent::*`, then just applying code motion and no projection provides mediocre performance similar to by-value. It is the ability

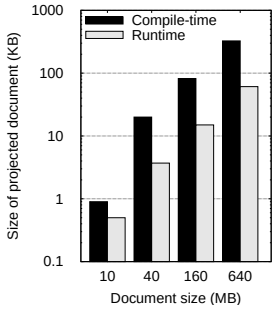


Figure 5.6: Selected nodes

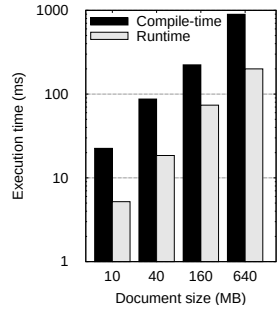


Figure 5.7: Execution time (ms)

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:

```
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:

```
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};
```

```
declare updating function fcn:doReplace($d as xs:string, $pid as xs:string, $n as node())
{do replace doc($d)//person[@id=$pid] with $n};
```

We tested all four kinds of updates, keeping the granularity of the updates constant, affecting 100 `person` nodes. For example, the insert query in XRPC looks as follows:

```
import module namespace fcn="foo" at "http://example.org/foo.xq";
fcn:doInsert("xrpc://p2/xmark200mb.xml", doc("Persons100.xml")/persons)
```

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.

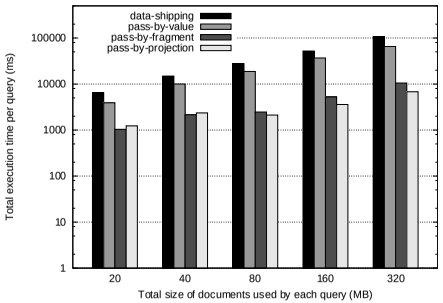


Figure 5.8: Execution time of read-only queries

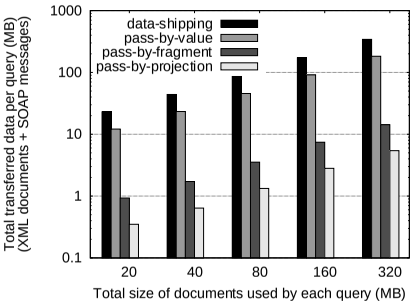


Figure 5.9: Bandwidth usage of read-only queries

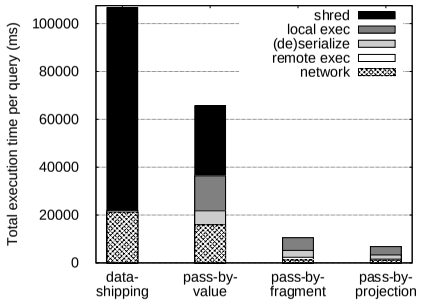


Figure 5.10: Time breakdown of read-only queries on 320MB data

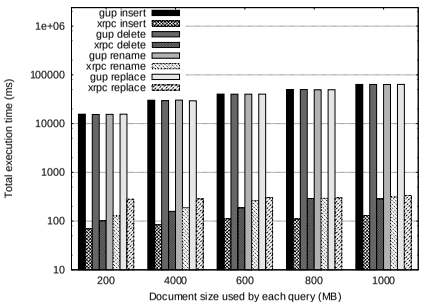


Figure 5.11: Execution time of updating queries

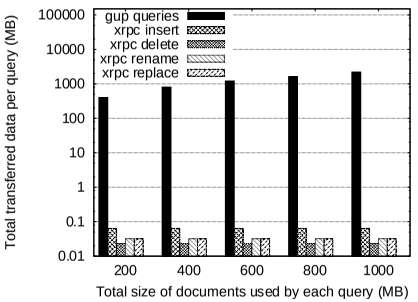


Figure 5.12: Bandwidth usage of updating queries

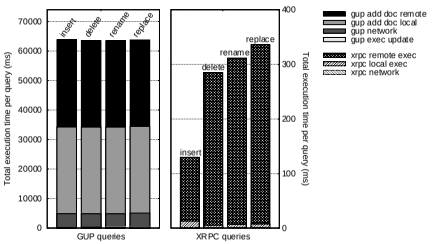


Figure 5.13: Time breakdown of updating queries on 1000MB data

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.

6

Correctness Proof of XQuery Decomposition

In this chapter, we formally prove the correctness of the decomposition algorithms presented in Chapter 5. For each algorithm, we prove that executing subexpressions of a query remotely over XRPC will produce results deep-equal to those of the original query, if this is allowed by a certain algorithm. We use a definition of deep-equal query results that also takes into account the freedoms an implementation has in processing some aspects of the language.

Roadmap Sections 6.1 and 6.2 contain auxiliary definitions, properties, lemmas and rules. Sections 6.3 - 6.7 are the main components of this chapter and contain theorems of the correctness of the techniques proposed in Chapter 5.

Since the goal of this chapter is to prove that a decomposed query produces deep-equal result to the original query, we start in Section 6.1 with definitions of different kinds of deep-equal for both sequences and queries. The contents of this section are important for the understanding of the proofs of the theorems in Sections 6.3 - 6.7.

Section 6.2 contains a complete list of judgement rules for all kinds of XQuery expressions. These rules specify how certain static properties of an expression are inferred. These rules are referred to by the proofs in Sections 6.3 - 6.7, because, to determine whether it is correct to decompose a certain expression, we need to know if the expression has the desired properties. This section can be used as a reference.

In Sections 6.3 - 6.5, we prove the correctness of each decomposition algorithm on read-only XCore queries, In Section 6.6, we prove the correctness of the decomposition algorithms on XCore queries containing XQUF expressions. Finally, we prove the correctness of the distributed code motion technique in Section 6.7.

6.1 Preliminaries

Notations. We use $\vec{a}$ to denote sequences $(a_1, ..., a_n)$ of length n (denoted as $|\vec{a}|$). We use $\vec{a}[i]$ to explicitly refer to the i-th item in a sequence $\vec{a}$. We permit $\vec{a}$ in set contexts to represent the set $\{a_1, ..., a_n\}$.

The symbol $\$x \mapsto E$ indicates that the variable $\$x$ is mapped to the value of the expression E, and $(Env + \$x \mapsto E)$ means that the environment Env is extended with the variable $\$x$ bound to (the value of) the expression E. We will use E interchangeably to represent an XCore expression and the result sequence of evaluating the expression.

We use $\vec{\tau}$ to denote the set of XML node types: {attribute, comment, document, element, processing-instruction, text}.

We abbreviate the XPath step "descendant-or-self" as "dos". Grammar rules (Table 5.2) concerning XPath step expressions are abbreviated as the following: ST = StepExpr, AS = AxisStep, NT = NodeTest.

6.1.1 Equality Relationships of Sequences

Equality relationships between sequences can be defined at different levels. The most strict equality relationship is the absolute equivalence between two sequences, which we define as the following:

Definition 6.1.1. Equivalent sequences: *Two sequences $\vec{s}_1$ and $\vec{s}_2$ are equivalent to each other, denoted $\vec{s}_1 \equiv \vec{s}_2$, iff they satisfy the following condition*[1]*:*

$$dEq(\vec{s}_1,\vec{s}_2) \wedge \forall i \in 1..|\vec{s}_1| : \texttt{dm:node-kind}(\vec{s}_1[i]) \in \vec{\tau} \mapsto \vec{s}_1[i] \ \texttt{is} \ \vec{s}_2[i]$$

However, in most circumstances, such as in XRPC, this strict equality relationship is not required, and deep-equal is sufficient. Using our notations, the XQuery deep-equal semantics for sequences can be expressed as follows:

Definition 6.1.2. Deep-equal sequences: *Two sequences $\vec{s}_1$ and $\vec{s}_2$ are deep-equal to each other, denoted $dEq(\vec{s}_1,\vec{s}_2)$, iff they satisfy the following conditions:*

$$\vec{s}_1 = \vec{s}_2 = (), or$$
$$|\vec{s}_1| = |\vec{s}_2| \wedge \forall i \in 1..|\vec{s}_1| : \begin{cases} \vec{s}_1[i] = \texttt{NaN} \wedge \vec{s}_2[i] = \texttt{NaN}, \\ \vec{s}_1[i] = \vec{s}_2[i], \textit{ if } \texttt{dm:node-kind}(\vec{s}_1[i]) \notin \vec{\tau} \wedge \texttt{dm:node-kind}(\vec{s}_2[i]) \notin \vec{\tau} \\ \texttt{fn:deep-equal}(\vec{s}_1[i], \vec{s}_2[i]), \textit{ if } \texttt{dm:node-kind}(\vec{s}_1[i]) \in \vec{\tau} \end{cases}$$

We use $dEq(E_1,E_2)$ to indicate that the result sequences of evaluating the expressions E_1 and E_2 are deep-equal to each other. For short, we say E_1 and E_2 are deep-equal.

In this section, the definitions of equality relationships of sequences are all based on the appearances of the sequences, i.e., two sequences could only possibly be equal (either equivalent or deep-equal), if their literal values appear to be equal. In the next section, we introduce a new kind of equality relationship for sequences, in which the equality relationship of two sequences is based on whether the results of applying a certain set of paths on these sequences are deep-equal.

6.1.2 Equality Relationships of Sequences with Projection

Let $\vec{\mathcal{P}}^{rel}$ denote a set of *relative projection paths*[2] which consists of a set of *relative used paths*, denoted $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{U}}$, and a set of *relative returned paths*, denoted $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{R}}$. Each path in $\vec{\mathcal{P}}^{rel}$ may contain XPath steps on all axes and the special built-in functions root(), id() and idref() (i.e., as defined by SimplePath in Table 5.6).

Given a set of relative projection paths $\vec{\mathcal{P}}^{rel}$, the results of applying $\vec{\mathcal{P}}^{rel}$ on two *non-deep-equal* sequences $\vec{s}_1$ and $\vec{s}_2$ could be *deep-equal*. Therefore, we introduce a lower level equality

[1] In XDM [71], the dm:node-kind accessor is only defined on the seven kinds of node. Here, we assume that dm:node-kind returns an error if $\vec{s}_1[i]$ is an atomic value.

[2] The projection paths are "relative" as they do not start from a document root, but rather from the node typed items in the sequences (Section 5.6.2).

relationship for sequences, with respect to a certain set of relative projection paths $\vec{\mathcal{P}}^{rel}$. We first give a formal definition of how the projection of $\vec{\mathcal{P}}^{rel}$ on a sequence $\vec{s}$ is computed at runtime.

Definition 6.1.3. Runtime XML projection operator $\mathfrak{P}$: *Let $\vec{s}$ be an XQuery node sequence, which may contain duplicates or overlapping nodes (i.e., nodes that have an ancestor-descendant relationship), and $\vec{\mathcal{P}}^{rel}$ a non-empty set of relative projection paths. The runtime XML projection operator $\mathfrak{P}$ creates a set of projected XML fragments $\vec{\mathcal{F}}$ by projecting $\vec{\mathcal{P}}^{rel}$ on $\vec{s}$. $\vec{\mathcal{F}}$ is computed as follows:*

1. *Apply $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{U}}$ and $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{R}}$ on $\vec{s}$ to produce the set of used nodes $\vec{\mathcal{N}}_u$ and the set of returned nodes $\vec{\mathcal{N}}_r$, respectively; and add $\vec{s}$ to $\vec{\mathcal{N}}_u$;*
2. *Let $\vec{\mathcal{D}}$ be the set of documents, from which nodes in $\vec{\mathcal{N}}_u$ and $\vec{\mathcal{N}}_r$ originate, sorted by document order;*
3. *Let $\vec{\mathcal{N}}_{u_i}$ and $\vec{\mathcal{N}}_{r_i}$ be respectively a subset of $\vec{\mathcal{N}}_u$ and $\vec{\mathcal{N}}_r$ which contain all nodes in $\vec{\mathcal{N}}_u$ and $\vec{\mathcal{N}}_r$ that originate from the same document $\vec{\mathcal{D}}[i]$. Then $\vec{\mathcal{F}}[i]$ is the projection of $\vec{\mathcal{N}}_{u_i} \cup \vec{\mathcal{N}}_{r_i}$ on $\vec{\mathcal{D}}[i]$, computed by the* RUNTIMEXMLPROJECTION *algorithm (Section 5.6.2, Algorithm 1), i.e.:*
   $|\vec{\mathcal{F}}| = |\vec{\mathcal{D}}| \wedge \forall i \in 1..|\vec{\mathcal{D}}| : \vec{\mathcal{F}}[i] = \text{RUNTIMEXMLPROJECTION}(\vec{\mathcal{N}}_{u_i}, \vec{\mathcal{N}}_{r_i}, \vec{\mathcal{D}}[i])$

Definition 6.1.4. By-projection equal sequences: *Let $\vec{s}_1$ and $\vec{s}_2$ be two XQuery sequences, and $\vec{\mathcal{P}}^{rel}$ a set of relative projection paths. If $\vec{\mathcal{P}}^{rel}$ is not an empty set, projections of $\vec{\mathcal{P}}^{rel}$ on $\vec{s}_1$ and $\vec{s}_2$, respectively, are computed as follows*[3]*: $\vec{\mathcal{F}}_1 = \mathfrak{P}(\vec{s}_1, \vec{\mathcal{P}}^{rel})$ and $\vec{\mathcal{F}}_2 = \mathfrak{P}(\vec{s}_2, \vec{\mathcal{P}}^{rel})$. We use $dEq^{\vec{\mathcal{P}}^{rel}}(\vec{s}_1, \vec{s}_2)$ to denote that $\vec{s}_1$ and $\vec{s}_2$ are by-projection equal to each other, with respect to $\vec{\mathcal{P}}^{rel}$. Whether $dEq^{\vec{\mathcal{P}}^{rel}}(\vec{s}_1, \vec{s}_2)$ holds is determined by the following rules:*

1. *If the set of relative projection paths is empty (i.e., no projection can be computed) s_1 or s_2 are by-projection equal,* iff *they are deep-equal to each other:*

$$\vec{\mathcal{P}}^{rel} = \emptyset : dEq^{\vec{\mathcal{P}}^{rel}}(\vec{s}_1, \vec{s}_2) \Leftrightarrow dEq(\vec{s}_1, \vec{s}_2)$$

2. *Otherwise, s_1 or s_2 are by-projection equal, if their projections $\vec{\mathcal{F}}_1$ and $\vec{\mathcal{F}}_2$ are deep-equal to each other:*

$$\vec{\mathcal{P}}^{rel} \neq \emptyset : dEq^{\vec{\mathcal{P}}^{rel}}(\vec{s}_1, \vec{s}_2) \Leftrightarrow dEq(\vec{\mathcal{F}}_1, \vec{\mathcal{F}}_2)$$

Example 6.1.5. *The leftmost column of Figure 6.1 shows two sequences $\vec{s}_1$ and $\vec{s}_2$ that each contains one different $\langle \texttt{a} \rangle \ldots \langle \texttt{/a} \rangle$ node. The right-most column of Figure 6.1 shows the resulting projected fragments $\vec{\mathcal{F}}_1$ and $\vec{\mathcal{F}}_2$, when the set of relative projection paths $\vec{\mathcal{P}}^{rel}$ is applied on $\vec{s}_1$ and $\vec{s}_2$, respectively. Here, $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{U}} = \{.,./b,./b/c\}$ and $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{R}} = \{./b/i\}$. Since $\vec{\mathcal{F}}_1$ and $\vec{\mathcal{F}}_2$ are deep-equal, we say that $\vec{s}_1$ and $\vec{s}_2$ are by-projection equal, with respect to $\vec{\mathcal{P}}^{rel}$. In Section 6.5, we explain in detail how $\vec{\mathcal{P}}^{rel}$ is computed.*

[3] If $\vec{s}_1$ or $\vec{s}_2$ contains a literal value, applying $\vec{\mathcal{P}}^{rel}$ on $\vec{s}_1$ or $\vec{s}_2$ results in a runtime error. However, in our compile time analysis, we can omit this check and assume that $\vec{s}_1$ and $\vec{s}_2$ contain correct nodes.

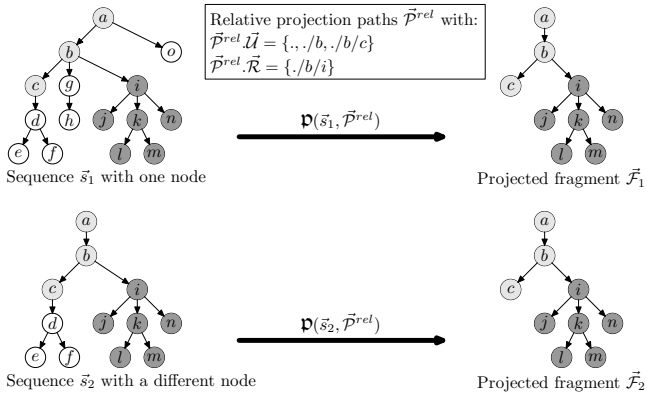


Figure 6.1: Two non-deep-equal sequences $\vec{s}_1$ and $\vec{s}_2$ that are by-projection deep-equal, with respect to $\vec{\mathcal{P}}^{rel}$, where $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{U}} = \{., ./b, ./b/c\}$ and $\vec{\mathcal{P}}^{rel}.\vec{\mathcal{R}} = \{./b/i\}$.

Remark In Definition 6.1.4, we require that $\vec{s}_1$ and $\vec{s}_2$ are by-projection equal, *if and only if* their projected fragments $\vec{\mathcal{F}}_1$ and $\vec{\mathcal{F}}_2$ are deep-equal, which implies that in the projected fragments, a used node may not contain (unused) descendant nodes. This requirement is more strict than what is exactly necessary, because, for instance, in Figure 6.1, if the $\langle \mathtt{c}/\rangle$ node in the projected fragment $\vec{\mathcal{F}}_2$ would have contained some descendants, it will not affect result of query evaluation (that is applied on $\vec{\mathcal{F}}_2$ instead of on s_2). In case of XRPC, it only causes unnecessary bandwidth usage in request/response messages, however, this defeats a major purpose of the by-projection semantics: minimise message sizes by pruning unused nodes. Thus, we regard it to be necessary to use this strict requirement in our definition of by-projection equal sequences, and the runtime XML projection operator $\mathfrak{P}$ guarantees that unused descendants of used nodes are pruned.

6.1.3 Equality Relationship of Read-Only Queries

Intuitively, one would say that two read-only queries are deep-equal to each other if their result sequences are deep-equal. Unfortunately, this simple comparison is not general enough to deal with the variations in the XQuery language. In the XQuery specifications[4], certain aspects of language processing are described as "*implementation defined*" or "*implementation dependent*". Implementation defined indicates an aspect that may differ between implementations, but must be specified by the implementer for each particular implementation. Implementation dependent indicates an aspect that may differ between implementations, is not specified by any W3C specification, and is not required to be specified by the implementer for any particular implementation. Since in our proofs, there is no need to differentiate whether a certain aspect of language processing is implementation defined or implementation dependent, we will refer to all such aspects as "*XQuery features with implementation freedom*". Because of these freedoms in the language, two queries containing the same expressions (or different executions of the same query) do not necessarily return literally the same results (i.e., XQuery deep-equal), while they both return *correct* results.

[4]This includes the following documents: XQuery 1.0 and XPath 2.0 Data Model [71], XQuery 1.0: An XML Query Language [38], XQuery 1.0 and XPath 2.0 Formal Semantics [67], XQuery 1.0 and XPath 2.0 Functions and Operators [124], XSLT 2.0 and XQuery 1.0 Serialization [39], and XQuery Update Facility 1.0 [58].

In this section, we first look into some examples of how the implementation freedom on the ordering of different XML documents affects query results. In case of differences due to nodes' document order, the XRPC rewriting framework guarantees consistent results by taking some extra precautions. For the differences, the basic approach of our rewriting framework is to re-define equality relationships of XQuery expressions to allow such differences.

Inter-Document Ordering *Document order* is defined in XQuery 1.0 and XPath 2.0 Data Model (XDM) [71]. The relative order of nodes in different XML documents (for short: *inter-document ordering*) is defined as: "*stable* but *implementation dependent*, subject to the following constraint: If any node in a given tree, `T1`, occurs before any node in a different tree, `T2`, then all nodes in `T1` are before all nodes in `T2`.", where *stable* means that "the relative order of two nodes will not change during the processing of a given query". That is, *any* ordering (including random order) of two different XML documents is *correct*, as long as the same ordering is used in one execution of a query. Under this definition, the query:

```
(doc("a")/a ≪ doc("b")/b) = (doc("a")/a ≪ doc("b")/b)
```

must always return `true`, since the ordering of `doc("a")` and `doc("b")` must be stable within this query. However, a query like

```
(doc("a")/a ≪ doc("b")/b) = (doc("d")/d ≪ doc("c")/c)
```

may return `true` or `false`, depending on how inter-document ordering is defined by the executing XQuery engine. In fact, it may even differ in subsequent executions using the same engine. Thus, when judging whether two queries are deep-equal, it is actually sufficient to check if they produce result sequences that are *"deep-equal with inter-document freedom"*:

Let Q and Q' be two XQuery queries containing one or more comparison(s) of inter-document ordering. Let $\vec{r}^{Q}$ and $\vec{r}^{Q'}$ be the result sequences of Q and Q', respectively. We say Q and Q' are deep-equal with inter-document freedom *to each other, if either $dEq(\vec{r}^{Q}, \vec{r}^{Q'})$ holds, or the differences between $\vec{r}^{Q}$ and $\vec{r}^{Q'}$ are* only *caused by that the executions of Q and Q' use different but stable inter-document ordering.*

If the query Q' above is a decomposition of Q, we regard Q' as a valid decomposition. Guaranteeing a stable ordering is trivial if Q opens each document only once, as *any* ordering is correct. Otherwise, a stable ordering is guaranteed by the decomposition algorithms by making sure that a subexpression comparing inter-document ordering is never decomposed, if there is another subexpression in the query that compares document order of the same document(s).

Deep-Equal Read-Only Queries with Implementation Freedom Besides inter-document ordering, the XQuery specifications have defined a number of other implementation defined or implementation dependent XQuery features. A complete list of these features, including those defined by XQUF, is given in Appendix B. Taking into account all XQuery features with implementation freedom, we say that a decomposed query Q' is "*deep-equal with implementation freedom*" to the original query Q, if the result sequence of Q' is deep-equal to *one of the result sequences* that *could* be returned by Q.

Recall that XQuery queries are always executed in a dynamic context *dynEnv*, which we simplify to a database state *db*, i.e., the documents and their contents stored in an XML database. The semantic judgement $db \vdash E \Rightarrow \vec{s}^{E}$ specifies that in the database state *db*, a read-only expression E evaluates to a sequence $\vec{s}^{E}$, which is an instance of the XDM.

Definition 6.1.6. Deep-equal read-only queries (with implementation freedom): *Let Q be an XQuery query and $\vec{S}^Q = \{\vec{s}_1^Q, \ldots, \vec{s}_n^Q\}$ the set of all valid result sequences of Q. That is, assume that Q contains k XQuery features with implementation freedom, and for each of these features, the number of choices an implementation has is $\{m_1, m_2, \ldots, m_k\}$, respectively. Then, the total number of valid result sequences of Q is bound by the permutation of the sum of $\{m_1, m_2, \ldots, m_k\}$:*

$$n = |\vec{S}^Q| = (\sum_{i=1}^{k} m_i)!$$

Let Q' be a decomposed query of Q and $\vec{s}_j^{Q'}$ a result sequence of Q', then, we say that Q and Q' are "deep-equal with implementation freedom" to each other, iff there is one $\vec{s}_i^Q \in \vec{S}^Q$, such that $dEq(\vec{s}_i^Q, \vec{s}_j^{Q'})$. For simplicity, we just say that Q and Q' are deep-equal to each other, and denote it as $dEq(Q, Q')$. Hence:

$$\frac{\begin{array}{l}\forall db: \; db \vdash Q \Rightarrow \vec{s}_1^Q \quad | \; db \vdash Q \Rightarrow \vec{s}_2^Q \quad | \ldots | \; db \vdash Q \Rightarrow \vec{s}_n^Q \\ \qquad db \vdash Q' \Rightarrow \vec{s}_1^{Q'} \quad | \; db \vdash Q' \Rightarrow \vec{s}_2^{Q'} \quad | \ldots | \; db \vdash Q' \Rightarrow \vec{s}_n^{Q'} \\ \exists \vec{s}_i^Q \in \{\vec{s}_1^Q, \ldots, \vec{s}_n^Q\}. \forall \vec{s}_j^{Q'} \in \{\vec{s}_1^{Q'}, \ldots, \vec{s}_n^{Q'}\} : dEq(\vec{s}_i^Q, \vec{s}_j^{Q'})\end{array}}{dEq(Q, Q')}$$

6.1.4 Equality Relationship of Updating Queries

The results of updating queries, i.e., XCore queries containing XQUF expressions, are reflected on the documents affected by an updating query, after all updates in the query are made effective. Naturally, one would regard two updating queries distributed over XRPC to be deep-equal if they update the same documents (on the same peers), and the resulting documents (after updates have been made effective) are deep-equal. However, also due to the XQuery features with implementation freedom, this simple comparison is too strict to be useful. For instance, the XQUF specifies that "If multiple groups of nodes are inserted by multiple insert expressions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation dependent.". This implies that even in local executions of XQUF queries, after the execution of the query (assuming an XML document "a.xml" containing one node $\langle a/\rangle$):

```
(insert nodes (⟨b/⟩,⟨c/⟩) into doc("a.xml")/a, insert nodes (⟨d/⟩,⟨e/⟩) into doc("a.xml")/a )
```

the original document could be changed into ⟨a⟩⟨b/⟩⟨c/⟩⟨d/⟩⟨e/⟩⟨/a⟩ or ⟨a⟩⟨d/⟩⟨e/⟩⟨b/⟩ ⟨c/⟩⟨/a⟩.

A more general way to compare if two updating queries are deep-equal is to compare their pending update list (PUL), which is an unordered collection of all updates that should be applied after the evaluation of a certain updating query. The PUL contains sufficient information to compare two queries, yet, it provides space to take into account the XQuery features with implementation freedom that have been defined for updating queries.

For updating queries we use a slightly different semantic judgement. A successful execution of an updating query always yields an empty sequence and a Pending Update List (PUL) Δ[5]. Thus, the semantic judgement $db \vdash E \Rightarrow ((), \Delta)$ specifies that in the database state db, evaluating an updating expression E yields a tuple, which consists of the empty sequence '()' and a PUL Δ. Each update primitive δ in a Δ is a triple $(N, T, \vec{C})$, where:

[5] A PUL is an unordered list of possibly duplicate update primitives [58], thus, our notation $\vec{a}$ for sequences or sets is not applicable here.

- N is the name of this update primitive. It may be any one of the update primitive functions defined by XQUF [58].
- T is the node identifier of the target node of this update primitive.
- $\vec{C}$ contains the new contents, for instance, $\vec{C}$ contains a sequence of to be inserted nodes, if N="`upd:insertInto`"; $\vec{C}$ contains a string literal, if N="`upd:rename`"; and $\vec{C}$ is $\emptyset$, if N="`upd:delete`".

Definition 6.1.7. Deep-equal update primitives: *Let δ and δ' be two update primitives. We say that δ and δ' are deep-equal, denoted $dEq(\delta,\delta')$, iff δ and δ' both represent the same update action on the same node with the same deep-equal contents:*

$$\delta.N = \delta'.N \wedge \delta.T \ \ is \ \ \delta'.T \wedge dEq(\delta.\vec{C},\delta'.\vec{C}) \mapsto dEq(\delta,\delta')$$

Because the PULs are unordered lists, we cannot check if two PULs are deep-equal by comparing the corresponding update primitives at the same position in the lists. Moreover, two PULs do not necessarily contain the same number of update primitives to have the same effect on the affected documents, since multiple `upd:delete` operations *may* be applied to the same node during execution of a query. In XQUF, deleting the same node multiple times in a query has the same effect as deleting the node just once, thus, they could be treated as equal. Contrary to deletion, multiple insertions of the same node sequence is not equal to inserting the sequence only once. Finally, a PUL *may not* contain more than one `upd:rename` operation that has the same target node. The same condition holds for the operations `upd:replaceNode`, `upd:replaceValue` and `upd:replaceElementContent`.

Definition 6.1.8. Deep-equal pending update lists (with implementation freedom): *Let Δ and Δ' be two PULs. Let Δ_{del}, Δ_{re} and Δ_{ins} represent subgroups of Δ that contain certain kinds of update actions, defined as the following:*

- $\Delta_{del} = \{\delta | \delta \in \Delta \wedge \delta.N = \text{"}upd:delete\text{"}\}$
- $\Delta_{re} = \{\delta | \delta \in \Delta \wedge \delta.N \in \{\text{"}upd:rename\text{"}, \text{"}upd:replaceNode\text{"}, \text{"}upd:replaceValue\text{"}, \text{"}upd:replaceElementContent\text{"}\}\}$
- $\Delta_{ins} = \{\delta | \delta \in \Delta \wedge \delta.N \in \{\text{"}upd:insertBefore\text{"}, \text{"}upd:insertAfter\text{"}, \text{"}upd:insertInto\text{"}, \text{"}upd:insertIntoAsFirst\text{"}, \text{"}upd:insertIntoAsLast\text{"}, \text{"}upd:insertAttributes\text{"}\}\}$

Similarly, Δ'_{del}, Δ'_{re} and Δ'_{ins} represent the same subgroups of Δ'. We say that Δ and Δ' are deep-equal to each other with implementation freedom (for short: Δ and Δ' are deep-equal), denoted $dEq(\delta,\delta')$, iff Δ and Δ' satisfy the following conditions:

1. $(\forall \delta_i \in \Delta_{del}.\exists \delta'_j \in \Delta'_{del} : dEq(\delta_i,\delta'_j)) \wedge (\forall \delta'_i \in \Delta'_{del}.\exists \delta_j \in \Delta_{del} : dEq(\delta'_i,\delta_j))$
2. $(\forall \delta_i \in \Delta_{re}.\exists \delta'_j \in \Delta'_{re}.\nexists \delta'_k \in \Delta'_{re} : dEq(\delta_i,\delta'_j) \wedge j \neq k \wedge \delta'_j.N = \delta'_k.N \wedge \delta'_j.T \ \ is \ \ \delta'_k.T) \wedge (\forall \delta'_i \in \Delta'_{re}.\exists \delta_j \in \Delta_{re}.\nexists \delta_k \in \Delta_{re} : dEq(\delta'_i,\delta_j) \wedge j \neq k \wedge \delta_j.N = \delta_k.N \wedge \delta_j.T \ \ is \ \ \delta_k.T)$
3. $|\Delta_{ins}| = |\Delta'_{ins}| \wedge \forall \delta_i \in distinct\text{-}primitives(\Delta_{ins}).\vec{\delta}_i = \{\forall \delta_j \in \Delta_{ins} \wedge dEq(\delta_i,\delta_j)\}.\ \vec{\delta}'_i = \{\forall \delta'_x \in \Delta'_{ins} \wedge dEq(\delta'_x,\delta_i)\} : |\vec{\delta}_i| = |\vec{\delta}'_i| \wedge \forall n \in 1..|\vec{\delta}_i| : dEq(\vec{\delta}_i[n],\vec{\delta}'_i[n])$

In the above definition, condition 1 checks if for each update primitive $\delta_i \in \Delta_{del}$, Δ'_{del} contains at least one update primitive δ'_j that is deep-equal to δ_i; and vice versa. Condition 2 checks if for each update primitive $\delta_i \in \Delta_{re}$, Δ'_{re} contains exactly one update primitive δ'_j that is deep-equal to δ_i; and vice versa. Condition 3 checks if Δ_{ins} and Δ'_{ins} both contain the same number of deep-equal insertion primitives. It does this by first computing, for each distinct primitive δ_i in Δ_{ins}, a subgroup $\vec{\delta}_i$, which consists of all primitives in Δ_{ins} that are deep-equal to δ_i. Then, for each such subgroup $\vec{\delta}_i$, the corresponding group $\vec{\delta}'_i$ with primitives from Δ_{ins} is computed. Finally, $\vec{\delta}_i$ and $\vec{\delta}'_i$ are compared to see if they are deep-equal. Here, `distinct-primitives()` is defined as:

$$\texttt{distinct-primitives}(\Delta) = \{\Delta^{dist} |\ \forall \delta_i \in \Delta . \exists \delta_j^{dist} \in \Delta^{dist} : dEq(\delta_i, \delta_j^{dist}) \wedge \forall \delta_i^{dist} \in \Delta^{dist} . \exists \delta_j \in \Delta . \nexists \delta_j^{dist} \in \Delta^{dist} : dEq(\delta_i^{dist}, \delta_j) \wedge i \neq j \wedge dEq(\delta_i^{dist}, \delta_j^{dist})\}$$

Definition 6.1.9. Deep-equal updating queries (with implementation freedom): *Two updating XQuery queries Q and Q' are deep-equal with implementation freedom to each other (for short: deep-equal), denoted $dEq(Q,Q')$, iff, in* any *database state, the evaluations of Q and Q' yield the tuples $((),\Delta)$ and $((),\Delta')$, respectively, where Δ and Δ' are deep-equal:*

$$\forall db : db \vdash Q \Rightarrow ((),\Delta) \wedge db \vdash Q' \Rightarrow ((),\Delta') \wedge dEq(\vec{\delta},\vec{\delta}') \Mapsto dEq(Q,Q')$$

6.1.5 Sequence Properties

We define several properties concerning XML node typed items in XQuery sequences. A sequence $\vec{s}$ is *distinct*, denoted $\eta(\vec{s})$, if $\vec{s}$ does not contain duplicate XML nodes. *Disjunct* is a more strict property: $\vec{s}$ is disjunct, denoted $\mu(\vec{s})$, if none of the XML node typed items in $\vec{s}$ has an ancestor/descendant relationship with another node typed item in $\vec{s}$. Finally, $\vec{s}$ is *ordered*, denoted $\sigma(\vec{s})$, if all XML node typed items in $\vec{s}$ appear in document order. Formally:

Property 6.1.10. Sequence properties:

Distinct η: $\eta(\vec{s}) \Leftrightarrow \forall s_i \in \vec{s} . \nexists s_j \in \vec{s} . i \neq j \,.\, \texttt{type}(s_i) \in \vec{\tau} . \texttt{type}(s_j) \in \vec{\tau} : s_j = s_i$

Disjunct μ: $\mu(\vec{s}) \Leftrightarrow \forall s_i \in \vec{s} . \nexists s_j \in \vec{s} . i \neq j \,.\, \texttt{type}(s_i) \in \vec{\tau} . \texttt{type}(s_j) \in \vec{\tau} : s_j \in \{s_i/\texttt{d-o-s::node()}\}$

Ordered σ: $\sigma(\vec{s}) \Leftrightarrow \forall i,j \in 1..|\vec{s}| . i < j . \texttt{type}(\vec{s}[i]) \in \vec{\tau} . \texttt{type}(\vec{s}[j]) \in \vec{\tau} : \vec{s}[i] \ll \vec{s}[j] \Mapsto \texttt{true}$

The next three lemmas can be deduced directly from the above property definitions:

Lemma 6.1.11. Empty sequence property: *The empty sequence is always distinct, disjunct and ordered.*

Lemma 6.1.12. Single item property: *Sequences containing a single item are always distinct, disjunct and ordered.*

Lemma 6.1.13. Disjunct implies distinct: *If a sequence is disjunct, then it is also distinct.*

As we will see later, the combination of distinct and ordered properties is needed for determining if the resulting sequence of applying the function `fs:distinct-doc-order()` on a sequence is equivalent with the original sequence. The combination of disjunct and ordered properties is crucial in the conservative algorithm to determine if forward XPath steps on XML nodes from remote peers would return correct results.

6.1.6 XPath Steps and `distinct-doc-order`

The XQuery 1.0 and XPath 2.0 Formal Semantics [67] defines a function `fs:distinct-doc-order()` (`ddo()` for short), which sorts its input nodes sequence by document order and removes duplicates. It is trivial to see that if the input sequence of nodes $\vec{e}$ is distinct and ordered, the result of `ddo`$(\vec{e})$ is equivalent with $\vec{e}$:

Lemma 6.1.14. DDO **equivalence:** $\eta(\vec{e}) \wedge \sigma(\vec{e}) \Mapsto \vec{e} \equiv ddo(\vec{e})$

XQuery requires that XPath expressions return their resulting nodes sequences in document order with duplicates eliminated. This is ensured in the XQuery Formal Semantics [67] by passing the intermediate result of an XPath expression to the function `fs:distinct-doc-order()` to produce the final result.

Definition 6.1.15. Raw XPath result: *We use the notation:* $\vec{e}/$AS*::*NT $= ddo(\vec{e}/as::$NT$)$*, where* AS *and* `as` *are the same* `AxisStep`*, to differentiate the input (indicated by the lower-cased abbreviation* `as`*) and the output of* `ddo()`*. We call* $\vec{e}/as::$NT *the* ***raw result*** *of applying an XPath step* `as::`NT *on the node sequence* $\vec{e}$*.*

Let $n = |\vec{e}|$*, the semantics of* $\vec{e}/as::$NT *is defined as:*

$$\vec{e}/as::\text{NT} = (\vec{e}[1]/as::\text{NT}, ..., \vec{e}[n]/as::\text{NT}),$$

which is a literal concatenation of the resulting sequence of applying `as::`NT *on each node* $\vec{e}[i] \in \vec{e}$*, in the same order they appear in* $\vec{e}$ *(for short: sequence order).*

If $\vec{e}$ is a sequence consisting of only one node, i.e., $\vec{e} = (e)$, it is directly seen that $\vec{e}/$`as`::NT is equivalent to $\vec{e}/$AS::NT.

Lemma 6.1.16. Raw XPath result on single node: *The raw result of applying an XPath step on a single XML node e is distinct and ordered:* $\eta(e/as::$NT$)$ *and* $\sigma(e/as::$NT$)$.

Now we are ready to deduce the properties of the result of a `FwdAxis` step[6]:

Lemma 6.1.17. Distinct-and-ordered `FwdAxis`: *Let* $\vec{e}$ *be a sequence of XML nodes and* $\vec{e}/as::$NT*, where* `as` $\in FwdAxis$*, the raw result of applying a* `FwdAxis` *step on each node in* $\vec{e}$*. If* $\vec{e}$ *is disjunct, then* $\vec{e}/as::$NT *is distinct; if* $\vec{e}$ *is ordered, then* $\vec{e}/as::$NT *is also ordered:*

$$\forall as \in FwdAxis : \mu(\vec{e}) \Mapsto \eta(\vec{e}/as::\text{NT}),\ \sigma(\vec{e}) \Mapsto \sigma(\vec{e}/as::\text{NT})$$

Proof. The proof can be done by induction: the base case $|\vec{e}| = 0$ is trivial; the other case $|\vec{e}| = 1$ is proven by Lemma 6.1.16. We assume that the lemma holds when $|\vec{e}| = n$, and we need to prove that the lemma also holds when $|\vec{e}| = n+1$. Let $\vec{e} = (\vec{e}_n, e_{n+1})$ and $\vec{e}_n = (e_1, ..., e_n)$, we have the following hypotheses:

(h0) `as` $\in FwdAxis$
(h1-a) $\mu(\vec{e}_n) \Mapsto \eta(\vec{e}_n/$`as`::NT$)$
(h1-b) $\sigma(\vec{e}_n) \Mapsto \sigma(\vec{e}_n/$`as`::NT$)$
(h2) $\vec{e}/$`as`::NT $= (\vec{e}_n/$`as`::NT, $e_{n+1}/$`as`::NT$)$
(h3-a) $\mu(\vec{e})$

[6] Similar work has been done by Hidders et al. in [96] and Fernández et al. in [73] to avoid unnecessary ordering and duplicate elimination operations in XPath expressions. The authors present rules to infer the ordered and distinct properties of the results of XPath steps on all axes, except `self`.

(h3-b) $\eta(\vec{e})$
(h3-c) $\sigma(\vec{e})$

to prove that $\eta(\vec{e}/\texttt{as::NT})$ and $\sigma(\vec{e}/\texttt{as::NT})$ hold, it is equivalent to prove:

(t1) $\eta((\vec{e}_n/\texttt{as::NT}, e_{n+1}/\texttt{as::NT}))$
(t2) $\sigma((\vec{e}_n/\texttt{as::NT}, e_{n+1}/\texttt{as::NT}))$

Since e_{n+1} is a single node, by Lemma 6.1.16 we have:

(h4-a) $\eta(e_{n+1}/\texttt{as::NT})$
(h4-b) $\sigma(e_{n+1}/\texttt{as::NT})$

With (h1-ab), (h4-ab) and (h2), we only need to prove that the following statements hold:

(t1') $\forall e_i \in (\vec{e}_n/\texttt{as::NT}).\forall e_j \in (e_{n+1}/\texttt{as::NT}) : \neg(e_i \texttt{ is } e_j)$
(t2') $\forall e_i \in (\vec{e}_n/\texttt{as::NT}).\forall e_j \in (e_{n+1}/\texttt{as::NT}) : e_i \ll e_j$

We consider all possible values of `as`:

- `as` = `self`:

  $e_{n+1}/\texttt{self::NT}$ returns e_{n+1} (or "()"). (t1') holds, because e_{n+1} is distinct from $\vec{e}_n$ (h3-a). Similarly, (t2') holds, because $\forall e_i \in \vec{e}_n : e_i \ll e_{n+1}$ (h3-b).

- `as` $\in$ {`child`, `descendant`, `descendant-or-self`, `attribute`}

  First, let us consider the case `as=descendant`: $e_{n+1}/\texttt{descendant::NT}$ selects all descendants of e_{n+1} that satisfy the condition NT. Assume there exists a $e_x \in e_{n+1}/\texttt{descendant::NT}$ that is also a descendant of a node e_y in $\vec{e}_n$, which implies that e_y and e_{n+1} have the ancestor-descendant relationship. However, this conflicts with the hypothesis (h3-a) that e_{n+1} is disjunct with $\vec{e}_n$. Hence, the assumption does not hold, which proves the statement:
  (t1') $\forall e_i \in (\vec{e}_n/\texttt{descendant::NT}).\forall e_j \in (e_{n+1}/\texttt{descendant::NT}) : \neg(e_i \texttt{ is } e_j)$

  From the XQuery definition of *document order* [38], we have:
  (h5) $\forall e_j \in (e_{n+1}/\texttt{descendant::NT}) : e_{n+1} \ll e_j$
  with (h3-b) we have: $\vec{e}_n \ll e_{n+1}$, which implies
  (h6) $\forall e_i \in (\vec{e}_n/\texttt{descendant::NT}) : e_i \ll e_{n+1}$.
  From (h5) and (h6), we can deduce:
  (t2') $\forall e_i \in (\vec{e}_n/\texttt{descendant::NT}).\forall e_j \in (e_{n+1}/\texttt{descendant::NT}) : e_i \ll e_j$

  The other three cases are proven in a similar way. □

6.2 Static Properties Analysis

In Section 5.6, we have described a new runtime XML projection technique, which extends the basic compile-time XML projection technique [125] with new inference rules to handle a larger subset of expressions defined in the XQuery Core grammar. The compile-time projection technique [125] defines an inference rule for each expression in its grammar to calculate a set of *projection paths*, based on the subexpressions. The projection paths are an overestimation of the set of nodes that will be touched by an expression, which are divided into a set of *returned paths* $\vec{r}$ (specify nodes that are returned by the expression), and a set of *used paths* $\vec{u}$ (specify nodes that are used to compute the result of the expression, but are not part of the result).

To compute the distinct, disjunct and ordered properties for an expression, we extend the main judgement rule of the path analysis with the triple $\langle \eta, \mu, \sigma \rangle$. Thus, the judgement:

$Env \vdash E \Rightarrow \vec{r}, \vec{u}, \langle \eta, \mu, \sigma \rangle$ holds, iff, under the environment Env, the expression E returns the set of paths $\vec{r}$, uses the set of paths $\vec{u}$, and evaluating E produces a sequence that has the properties $\langle \eta, \mu, \sigma \rangle$. Each path in $\vec{r}$ and $\vec{u}$ is a `SimplePath` as defined in Table 5.6. If the result sequence of an expression does not have a certain property, the property is replaced by the symbol $\varnothing$ in the judgement.

In the remainder of this section, we redefine inference rules for those expressions whose results have at least one of the properties $\langle \eta, \mu, \sigma \rangle$. We omit redefining rules for expressions for which the judgement $Env \vdash E \Rightarrow \vec{r}, \vec{u}, \langle \varnothing, \varnothing, \varnothing \rangle$ always holds. In the inference rules, we use $\perp$ in premises to denote that the value of a property is not significant.

6.2.1 Literal Values

$$\frac{}{Env \vdash Literal \Rightarrow (), (), \langle \eta, \mu, \sigma \rangle} \quad \text{(LITERAL)}$$

Literal values do not access or return any paths. By Lemma 6.1.12, a literal value is always distinct, disjunct and ordered.

6.2.2 Variables

$$\frac{Env(\$x) = E_1 \qquad Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle}{Env \vdash \$x \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle} \quad \text{(VAR)}$$

If a variable $\$x$ is bound to the expression E, accessing the variable uses and returns the same set of paths as E.

The distinct, disjunct and ordered properties of a variable $\$x$ is determined by these properties of the expression E to which the variable is bound.

6.2.3 Sequences

$$\frac{}{Env \vdash () \Rightarrow (), (), ()\langle \eta, \mu, \sigma \rangle} \quad \text{(EMPTYSEQ)}$$

The empty sequence does not use or return any paths, and it is always distinct, disjunct and ordered (Lemma 6.1.11).

$$\frac{Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \vec{v}_1\langle \perp, \perp, \perp \rangle \qquad Env \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, \vec{v}_2\langle \perp, \perp, \perp \rangle}{Env \vdash (E_1,\ E_2) \Rightarrow \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, \vec{v}_3\langle \varnothing, \varnothing, \varnothing \rangle} \quad \text{(SEQ)}$$

The sequence expression concatenates two sequences into one sequence, without eliminating duplicate nodes or changing the order in which the items appear in the resulting sequence. The returned and used paths of a sequence expression is the union of the returned and used paths of its subexpressions. As it can not be statically determined, this rule deduces that the result of a (non-empty) sequence expression is not distinct, disjunct or ordered.

6.2.4 `for` Expressions

$$\frac{Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \vec{v}_1\langle \perp, \perp, \perp \rangle \qquad Env' = Env + (\$x \mapsto E_1) \qquad Env' \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, \vec{v}_2\langle \perp, \perp, \perp \rangle}{Env \vdash \texttt{for}\ \$x\ \texttt{in}\ E_1\ \texttt{return}\ E_2 \Rightarrow \vec{r}_2, \vec{r}_1 \cup \vec{u}_1 \cup \vec{u}_2, \vec{v}_3\langle \varnothing, \varnothing, \varnothing \rangle} \quad \text{(FOR)}$$

A `for` expression binds new variables in the environment, hence, the environment is first extended with the new variable and passed to the evaluation of E_2. A `for` expression returns the returned paths of its `return` clause. All other paths are used to calculate the result of a `for` expression.

Similar to sequence expression, a `for` expression returns a simple concatenation of the resulting sequences of all iterations, without eliminating duplicate nodes or sorting them in their document order, hence, this rule deduces that the result sequence of a `for` expression is not distinct, disjunct or ordered.

6.2.5 `let` Expressions

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env' = Env + (\$x \mapsto E_1) \\ Env' \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \eta_2, \mu_2, \sigma_2 \rangle \end{array}}{Env \vdash \texttt{let}\ \$x := E_1\ \texttt{return}\ E_2 \Mapsto \vec{r}_2, \vec{r}_1 \cup \vec{u}_1 \cup \vec{u}_2, \langle \eta_2, \mu_2, \sigma_2 \rangle} \quad \text{(LET)}$$

A `let` expression binds a new variable in the environment, hence, the environment is first extended with the new variable and passed to the evaluation of E_2. A `let` expression returns the returned paths of its `return` clause. All other paths are used to calculate the result of a `let` expression.

A `let` expression has the same distinct, disjunct and ordered properties as its return expression E_2.

6.2.6 Conditionals

$$\frac{\begin{array}{c} Env \vdash E_0 \Mapsto \vec{r}_0, \vec{u}_0, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle \\ Env \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \eta_2, \mu_2, \sigma_2 \rangle \end{array}}{Env \vdash \texttt{if}\ (E_0)\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \Mapsto \vec{r}_1 \cup \vec{r}_2, \vec{r}_0 \cup \vec{u}_0 \cup \vec{u}_1 \cup \vec{u}_2, \langle \eta_1 \& \eta_2, \mu_1 \& \mu_2, \sigma_1 \& \sigma_2 \rangle} \quad \text{(IF)}$$

An `if` expression returns either the expression in the `then` branch or the expression in the `else` branch. Thus, the returned paths of an `if` expression is the union of the returned paths of these two expressions. All other paths are used to calculate the result of the `if` expression.

At compile time, we can only conclude that the result of an `if` expression is distinct, disjunct and ordered, *iff* it can be statically determined that both its `then` and `else` branches are distinct, disjunct and ordered.

6.2.7 Typeswitch

$$\frac{\begin{array}{c} Env \vdash E_0 \Mapsto \vec{r}_0, \vec{u}_0, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle \\ \ldots \\ Env \vdash E_n \Mapsto \vec{r}_n, \vec{u}_n, \langle \eta_n, \mu_n, \sigma_n \rangle \end{array}}{\begin{array}{l} Env \vdash \texttt{typeswitch}\ (E_0)\texttt{case}\ \$x_1\ \texttt{as}\ \mathit{SequenceType}_1\ \texttt{return}\ E_1\ \ldots\texttt{default}\ \$x_n\ \texttt{return}\ E_n \\ \quad \Mapsto \vec{r}_1 \cup \ldots \cup \vec{r}_n, \vec{r}_0 \cup \vec{u}_0 \cup \ldots \cup \vec{u}_n, \langle \eta_1 \& \ldots \& \eta_n, \mu_1 \& \ldots \& \mu_n, \sigma_1 \& \ldots \& \sigma_n \rangle \end{array}} \quad \text{(TPSWTCH)}$$

The inference rule for `typeswitch` is very similar to the one for the conditionals, except that `typeswitch` needs to handle multiple branches.

If the `return` expression of all `case` clauses and the `default` clause are distinct, disjunct and ordered, the result of the `typeswitch` also has these properties.

6.2.8 Value and Node Comparisons

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash E_1 \odot E_2 \Mapsto (), \vec{r}_1 \cup \vec{u}_1 \cup \vec{r}_2 \cup \vec{u}_2, \langle \eta, \mu, \sigma \rangle} \quad \text{(COMP)}$$

The symbol $\odot$ represents the value and node comparison operators =, !=, <, <=, >, >=, `is`, $\ll$ and $\gg$. Value and node comparisons never return nodes, but a literal boolean value. Thus, all paths needed for a value comparison are used paths.

By Lemma 6.1.12, a single literal value is always distinct, disjunct and ordered, thus the result of a value comparison also has these properties, regardless of whether its subexpressions E_1 and E_2 have these properties or not.

6.2.9 Order Expressions

$$\frac{\begin{array}{c} Env \vdash E_0 \Rightarrow \vec{r}_0, \vec{u}_0, \langle \eta_0, \mu_0, \bot \rangle \\ Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ \ldots \\ Env \vdash E_n \Rightarrow \vec{r}_n, \vec{u}_n, \langle \bot, \bot, \bot \rangle \end{array}}{\begin{array}{l} Env \vdash (E_0)\ \texttt{order by}\ E_1\ \texttt{ascending|descending} \ldots E_n\ \texttt{ascending|descending} \\ \quad \Rightarrow \vec{r}_0, \vec{u}_0 \cup \vec{r}_1 \cup \ldots \cup \vec{r}_n \cup \vec{u}_1 \cup \ldots \cup \vec{u}_n, \langle \eta_0, \mu_0, \varnothing \rangle \end{array}} \quad (\textsc{Order})$$

An `order by` expression returns the expression E_0 reordered by the `OrderSpec` expressions $E_1, \ldots, E_n$. Thus, the returned paths of E_0 are also the returned paths of the `order by` expression, and all other paths are propagated as the used paths of the `order by` expression.

The distinct and disjunct properties of an `order by` expression are determined by its input expression E_0. However, the result of an `order by` expression is regarded as never ordered by document order, because it can not be determined at compile time how the result sequence will be ordered.

6.2.10 Node Set Expressions

$$\frac{\begin{array}{c} Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash E_1\ \texttt{union}\ E_2 \Rightarrow \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, \langle \eta, \varnothing, \sigma \rangle} \quad (\textsc{Union})$$

$$\frac{\begin{array}{c} Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \bot, \mu_1, \bot \rangle \\ Env \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash E_1 \boxdot E_2 \Rightarrow \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, \langle \eta, \mu_1, \sigma \rangle} \quad (\tfrac{\textsc{Intersect}}{\textsc{Except}})$$

The symbol $\boxdot$ represents the node set operators `intersect` and `except`. For all three kinds of node set expressions, it holds that *(i)* the returned and used paths are respectively the union of the returned and used paths of their subexpressions, and *(ii)* their results are always distinct and ordered as required by the XQuery language.

However, situations are different regarding the disjunct property. The result of a `union` expression is never disjunct, because it combines nodes from two sequences. Even if the two subexpressions E_1 and E_2 are disjunct themselves, statically, it can not be determined if all nodes in E_2 are disjunct with all nodes in E_1. The operators `intersect` and `except` *only* return nodes from their first subexpression E_1, hence, their disjunctness depends on that of E_1.

6.2.11 Constructors

$$\frac{\begin{array}{c} Env \vdash E_0 \Rightarrow \vec{r}_0, \vec{u}_0, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \end{array}}{\begin{array}{l} Env \vdash \texttt{element|attribute}\ \{E_0\}\{E_1\} \\ \quad \Rightarrow \texttt{doc}(v_i\texttt{::}v_i), \vec{r}_0 \cup \vec{u}_0 \cup \vec{r}_1 \cup (\vec{r}_1\texttt{/descendant::*}) \cup \vec{u}_1, \langle \eta, \mu, \sigma \rangle \end{array}} \quad (\textsc{ElemAttr})$$

$$\frac{Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{document|text}\ \{E_1\} \Rightarrow \texttt{doc}(v_i\texttt{::}v_i), \vec{r}_1 \cup (\vec{r}_1\texttt{/descendant::*}) \cup \vec{u}_1, \langle \eta, \mu, \sigma \rangle} \quad (\textsc{DocTxt})$$

Constructors make a deep copy of their operands. The newly constructed element is annotated with a synthetic (unique) URI $\texttt{doc}(v_i\texttt{::}v_i)$ to denote the *d*-graph vertex v_i, from which

the element originates (see Section 5.4). Evaluation of the name expression E_0 always yields a single literal string value. Constructors do not return any nodes from the original XML nodes or documents. However, as the whole subtree of nodes in the content expression E_2 are copied, we add all descendants of those nodes to the set of used paths.

As constructors always return a single fresh node, their result is always distinct, disjunct and ordered.

6.2.12 XPath Expressions

$$\frac{Env \vdash E_0 \Mapsto \vec{r}_0, \vec{u}_0, \langle \bot, \bot, \bot \rangle}{Env \vdash E_0/\texttt{attribute}::\text{NT} \Mapsto \vec{r}_0/\texttt{attribute}::\text{NT}, \vec{r}_0 \cup \vec{u}_0, \langle \eta, \mu, \sigma \rangle} \quad (\text{STEP}^a)$$

$$\frac{\begin{array}{c} Env \vdash E_0 \Mapsto \vec{r}_0, \vec{u}_0, \langle \bot, \mu_0, \bot \rangle \\ \text{AS} \in \{\texttt{self}, \texttt{child}\} \end{array}}{Env \vdash E_0/\text{AS}::\text{NT} \Mapsto \vec{r}_0/\text{AS}::\text{NT}, \vec{r}_0 \cup \vec{u}_0, \langle \eta, \mu_0, \sigma \rangle} \quad (\text{STEP}^{sc})$$

$$\frac{\begin{array}{c} Env \vdash E_0 \Mapsto \vec{r}_0, \vec{u}_0, \langle \bot, \bot, \bot \rangle \\ \text{AS} \in \texttt{AxisStep} \backslash \{\texttt{self}, \texttt{child}, \texttt{attribute}\} \end{array}}{Env \vdash E_0/\text{AS}::\text{NT} \Mapsto \vec{r}_0/\text{AS}::\text{NT}, \vec{r}_0 \cup \vec{u}_0, \langle \eta, \varnothing, \sigma \rangle} \quad (\text{STEP}^{\not sc})$$

An XPath step applies a `StepExpr` on each returned path of its subexpression E_0. The result of an XPath step is always distinct and ordered, as required by the XQuery language. To compute the disjunct property, however, the path steps need to be treated differently.

- STEP^a: attribute nodes of distinct nodes never overlap, thus, the result of an `attribute` step is always disjunct.

- STEP^{sc}: if E_0 is disjunct, the result of applying a `self`, `child`, `preceding-sibling` or `following-sibling` step on E_0 is also disjunct. The case $E_0/\texttt{self}::\text{NT}$ is trivial. The child nodes of a single node are disjunct, and all nodes in E_0 are disjunct, so $E_0/\texttt{child}::\text{NT}$ is disjunct as well.

- STEP$^{\not sc}$: for the remaining axis steps that include `ancestor`, `ancestor-or-self`, `parent`, `preceding`, `preceding-sibling`, `following`, `following-sibling`, `descendant` and `descendant-or-self`, it can not be statically determined if their results are disjunct, regardless of if the result of E_0 is disjunct or not.

6.2.13 Built-in Function Calls

$$\frac{\begin{array}{c} \forall i \in 1..k : Env \vdash E_i \Mapsto \vec{r}_i, \vec{u}_i, \langle \eta_i, \mu_i, \sigma_i \rangle \\ \mathcal{R}(\mathcal{F}(E_1, ..., E_k)) \Mapsto \vec{r}_\mathcal{F}, \vec{u}_\mathcal{F}, \langle \eta_\mathcal{F}, \mu_\mathcal{F}, \sigma_\mathcal{F} \rangle \end{array}}{Env \vdash \mathcal{F}(E_1, ..., E_k) \Mapsto \vec{r}_\mathcal{F}, (\bigcup_{i=1}^{k}(\vec{r}_i \cup \vec{u}_i)) \cup \vec{u}_\mathcal{F}, \langle \eta_\mathcal{F}, \mu_\mathcal{F}, \sigma_\mathcal{F} \rangle} \quad (\text{BLTIN})$$

For each built-in function, we assume that there is a corresponding helper rule $\mathcal{R}$ which specifies how the returned paths of the function results depend on the returned paths of the parameters and if the result of the function is distinct, disjunct or ordered. The helper rules for all built-in functions defined in [124] are listed in Appendix C.

6.2.14 Transform Expressions

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ \cdots \\ Env \vdash E_n \Mapsto \vec{r}_n, \vec{u}_n, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_m \Mapsto \vec{r}_m, \vec{u}_m, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_t \Mapsto \vec{r}_t, \vec{u}_t, \langle \eta_t, \mu_t, \sigma_t \rangle \end{array}}{\begin{array}{l} Env \vdash \texttt{copy } \$x_1 := E_1, \ldots, \$x_n := E_n \texttt{ modify } E_m \texttt{ return } E_t \\ \quad \Mapsto \vec{r}_t, (\bigcup_{i=1}^{n}(\vec{r}_i \cup \vec{u}_i \cup \vec{r}_i/\texttt{descendant::*})) \cup \vec{r}_m \cup \vec{u}_m \cup \vec{u}_t, \langle \eta_t, \mu_t, \sigma_t \rangle \end{array}} \quad \text{(TRANS)}$$

A `TransformExpr` expression returns the value of the expression in its `return` clause, thus the returned paths of a `TransformExpr` are the returned paths of E_t, all other paths are propagated as used paths. Since deep-copies of the results of $E_1, \ldots, E_n$ are made, it is necessary to add their descendants into used paths.

A `TransformExpr` makes deep copies of the subexpressions in its `copy` clause, and each newly created node gets a new node identity. The subexpression in the `modify` clause must be an updating expression (or a vacuous expression [58]), which does not return any value. Thus, the η, μ, σ properties of a `TransformExpr` is determined by the η, μ, σ properties of the `return` expression E_t.

6.3 Correctness Proof of the Conservative Decomposition Algorithm

The semantics of marshalling and unmarshalling function parameters (or results) in XRPC under the pass-by-value semantics is defined by the functions `s2n()` and `n2s()` (see Section 3.4). The effect of marshalling (`s2n()`) a sequence and then unmarshalling it (`n2s()`) is that a deep-copy of the sequence is created.

Definition 6.3.1. By-value copy operator C^v: *Let $\vec{s}$ be an XQuery sequence that may contain duplicates or overlapping nodes, we use $C^v(\vec{s})$ to indicate a by-value copy of $\vec{s}$. The semantics of the by-value copy operator C^v is defined as: $C^v(\vec{s}) = n2s(s2n(\vec{s}))$.*

Under the pass-by-value semantics, the semantics of executing an expression on a remote peer is to first replace each parameter of the expression with a by-value copy, then execute the expression (using the by-value copies on the local peer), and finally return a by-value copy of the result.

Property 6.3.2. C^v properties:

$$dEq(\vec{s}, C^v(\vec{s})) \qquad \eta(\vec{s}) \Mapsto \eta(C^v(\vec{s})) \qquad \mu(\vec{s}) \Mapsto \mu(C^v(\vec{s})) \qquad \sigma(\vec{s}) \Mapsto \sigma(C^v(\vec{s}))$$

Below we introduce a mapping function to denote the relationship between two corresponding nodes in the subtrees rooted at node typed items in $\vec{s}$ and $C^v(\vec{s})$, respectively.

Definition 6.3.3. By-value mapping function m_v: *Let $\vec{e}$ be a sequence of XML nodes and $C^v(\vec{e})$ its by-value copy. The by-value mapping function m_v maps each node $e_i \in \{\vec{e}/\texttt{d-o-s::node()}\}$ to exactly one node $e'_i \in \{C^v(\vec{e})/\texttt{d-o-s::node()}\}$, and vice versa. Formally, the result of m_v is defined as the following:*

$$\begin{array}{l} \forall i \in 1..|\vec{e}|. \forall j \in 1..|\vec{e}[i]/\texttt{d-o-s::node()}| : \\ \quad m_v(\vec{e}[i]/\texttt{d-o-s::node()}[j]) \ \ \texttt{is} \ \ C^v(\vec{e})[i]/\texttt{d-o-s::node()}[j]) \end{array}$$

For each e_i, $m_v(e_i)$ is called the by-value-mapping of e_i in $C^v(\vec{e})$. The reverse function m_v^{-1} maps an e_i' back to its corresponding e_i, such that, e_i `is` $m_v^{-1}(m_v(e_i))$.

Lemma 6.3.4. Mapped raw results of `FwdAxis` steps on $\vec{e}$ and $C^v(\vec{e})$: *Let $\vec{e}$ be a sequence of XML nodes and $C^v(\vec{e})$ its by-value copy. The raw results of applying multiple consecutive `FwdAxis` steps on $\vec{e}$ and $C^v(\vec{e})$ are the by-value-mapping of each other, denoted: $C^v(\vec{e})/\mathtt{as}_1::\mathtt{NT}_1/.../\mathtt{as}_n::\mathtt{NT}_n) \equiv m_v(\vec{e}/\mathtt{as}_1::\mathtt{NT}_1/.../\mathtt{as}_n::\mathtt{NT}_n)$, where $\forall \mathtt{as}_i \in FwdAxis$.*

Proof. If $\vec{e} = (e)$ and $n = 1$, it is trivial to see that the following holds:

(t1) $C^v(e)/\mathtt{as}_1::\mathtt{NT}_1 = m_v(e/\mathtt{as}_1::\mathtt{NT}_1)$

If $|\vec{e}| > 1$ and $n = 1$, with Definition 6.1.15, we know that the raw results of applying $\mathtt{as}_1::\mathtt{NT}_1$ on $\vec{e}$ is a literal concatenation of the intermediate raw result $\vec{e}[i]/\mathtt{as}_1::\mathtt{NT}_1$ in sequence order. The same holds for $C^v(\vec{e})$. Thus we have:

(t2) $C^v(\vec{e})/\mathtt{as}_1::\mathtt{NT}_1 = m_v(\vec{e}/\mathtt{as}_1::\mathtt{NT}_1)$

If $|\vec{e}| > 1$ and $n > 1$, the raw results is again a concatenation of each intermediate raw result (i.e., one step on one node) in sequence order, which implies:

(t3) $C^v(\vec{e})/\mathtt{as}_1::\mathtt{NT}_1/.../\mathtt{as}_n::\mathtt{NT}_n) = m_v(\vec{e}/\mathtt{as}_1::\mathtt{NT}_1/.../\mathtt{as}_n::\mathtt{NT}_n)$ □

Definition 6.3.5. Node relationship function R: *Let e_l and e_r be two XML nodes, the relationship function R takes e_l and e_r as its input and returns the relationship between e_l and e_r, as the following:*

$$R(e_l, e_r) = \begin{cases} \text{“}\ll\text{”}, \text{ if } e_l \ll e_r; \\ \text{“}\mathtt{is}\text{”}, \text{ if } e_l \ \mathtt{is}\ e_r; \\ \text{“}\gg\text{”}, \text{ if } e_l \gg e_r. \end{cases}$$

Note that R is exactly what is needed by `ddo()` to process its input sequence. The following lemma can be directly deduced from the definition of by-value copy:

Lemma 6.3.6. By-value node relationships: *Let e be a single XML node and $C^v(e)$ its by-value copy. Then, we have:*

$$\forall u, w \in \{e/\mathtt{d\text{-}o\text{-}s}::\mathtt{node}()\}.\forall u', w' \in \{C^v(e)/\mathtt{d\text{-}o\text{-}s}::\mathtt{node}()\} : u' = m_v(u) \wedge w' = m_v(w) \Mapsto R(u,w) = R(u',w')$$

That is, the relationship between any two nodes u and w in the subtree rooted at e is the same as the relationship between their corresponding nodes u' and w' in $C^v(e)$.

Lemma 6.3.7. By-value deep-equal `ddo()`: *Let e be a single XML node and $C^v(e)$ its by-value copy. Let $\vec{a}$ be a sequence containing XML nodes in the subtree rooted at e, i.e, $\forall a_i \in \{e/\mathtt{d\text{-}o\text{-}s}::\mathtt{node}()\}$, and $\vec{b}$ the by-value mapping of $\vec{a}$ in $C^v(e)$, then: $dEq(\mathtt{ddo}(\vec{a}), \mathtt{ddo}(\vec{b}))$.*

Proof. With $\vec{b} = m_v(\vec{a})$, we have:

(h1) $\forall i \in 1..|\vec{a}| : \vec{b}[i] = m_v(\vec{a}[i])$

With (h1) and Lemma 6.3.6, we have:

(h2) $\forall i, j \in 1..|\vec{a}| : R(\vec{a}[i], \vec{a}[j]) = R(\vec{b}[i], \vec{b}[j])$

That is, the relationship between any two nodes in $\vec{a}$ is the same as their corresponding nodes in $\vec{b}$. This directly leads to $dEq(\mathtt{ddo}(\vec{a}), \mathtt{ddo}(\vec{b}))$. □

Lemma 6.3.8. By-value deep-equal `FwdAxis`: *Let $\vec{e}$ be a sequence of XML nodes and $C^v(\vec{e})$ its by-value copy. Iff $\vec{e}$ is disjunct and ordered, the result sequences of applying any number of consecutive `FwdAxis` steps on $\vec{e}$ and $C^v(\vec{e})$ are deep equal to each other. Formally:*

$$\mu(\vec{e}), \sigma(\vec{e}) \Mapsto dEq(\vec{e}/\mathtt{AS}_1::\mathtt{NT}_1/.../\mathtt{AS}_n::\mathtt{NT}_n, C^v(\vec{e})/\mathtt{AS}_1::\mathtt{NT}_1/.../\mathtt{AS}_n::\mathtt{NT}_n) \wedge \forall i \in 1..n : \mathtt{AS}_i \in FwdAxis$$

Proof. We prove this lemma in two steps. First, we show that the lemma holds if $\vec{e}$ contains a single XML node. Then, we generalise the proof to the case that $\vec{e}$ contains multiple XML nodes. Note that computing the final results of applying XPath steps on a node sequence can be done by first computing the raw results (Definition 6.1.15), and then applying `ddo()` on the raw results to eliminate duplicates and sort nodes (Section 6.1.6). Lemma 6.3.4 has already proven that $\vec{e}/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n$ and $C^v(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n$ are by-value-mapping of each other. Thus, the crucial issue in this proof is to show that `ddo()` works correctly on $C^v(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n$.

Case 1: $\vec{e} = (e)$

With Lemma 6.3.4, we have:

(h1-1) $C^v(e)/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n = m_v(e/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n)$

For short, we use $\vec{u}$ and $\vec{w}$ to denote $e/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n$ and $C^v(e)/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n$, respectively. Since all $\texttt{as}_i$ are `FwdAxis` steps, we have:

(h1-2) $\forall x \in 1..|\vec{u}| : \vec{u}[x] \in \{e/\texttt{d-o-s::node()}\} \wedge \vec{w}[x] \in \{C^v(e)/\texttt{d-o-s::node()}\}$

With (h1-1) and (h1-2) Lemma 6.3.7, we have:

(h1-3) $dEq(\texttt{ddo}(\vec{u}), \texttt{ddo}(\vec{w}))$

With (h1-3) and Definition 6.1.15, we have:

(t1) $dEq(e/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, C^v(e)/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n)$

Thus, the lemma holds, when $\vec{e} = (e)$.

Case 2: $\vec{e} = (e_1, ..., e_x)$

With (t1), we have:

(h2-1) $\forall x \in 1..|\vec{e}| : dEq(\vec{e}[x]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, C^v(\vec{e})[x]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n)$

With Property 6.3.2, we have:

(h2-2) $\mu(\vec{e}) \Rightarrow \mu(C^v(\vec{e})), \sigma(\vec{e}) \Rightarrow \sigma(C^v(\vec{e}))$

Which implies that the intermediate results of applying $\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n$ on each node in $\vec{e}$ (or $C^v(\vec{e})$) in sequence order, are distinct and ordered, i.e., $\forall i, j \in 1..|\vec{e}| \wedge i < j$:

(h2-3) $\forall u \in \{\vec{e}[i]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n\}.\forall w \in \{\vec{e}[j]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n\}:$
$R(u,w) = \text{“} \ll \text{”}$

(h2-4) $\forall u' \in \{C^v(\vec{e})[i]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n\}.\forall w' \in \{C^v(\vec{e})[j]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n\}:$
$R(u',w') = \text{“} \ll \text{”}$

With (h2-3), (h2-4) and Lemma 6.1.14 we have:

(h2-5) $(\vec{e}[1]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, ..., \vec{e}[x]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n) \equiv$
$\texttt{ddo}(\vec{e}[1]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, ..., \vec{e}[x]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n)$

(h2-6) $(C^v(\vec{e})[1]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, ..., C^v(\vec{e})[x]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n) \equiv$
$\texttt{ddo}(C^v(\vec{e})[1]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, ..., C^v(\vec{e})[x]/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n)$

Which implies:

(t2) $dEq(\vec{e}/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, C^v(\vec{e})/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n)$

Thus, the lemma also holds, when $\vec{e} = (e_1, ..., e_x)$. □

Finally, the correctness of the conservative decomposition algorithm is defined as follows:

Theorem 6.3.9. Conservative Decomposition Correctness: *Let Q be a normal read-only XCore query (i.e., without any XRPC expressions) and G the corresponding d-graph. $I^v(G) \subset G$ is the non-empty set of decomposition points validated by the by-value insertion conditions. Let G' be the d-graph derived by doing an `XRPCExpr` insertion above each vertex in $I^v(G)$*

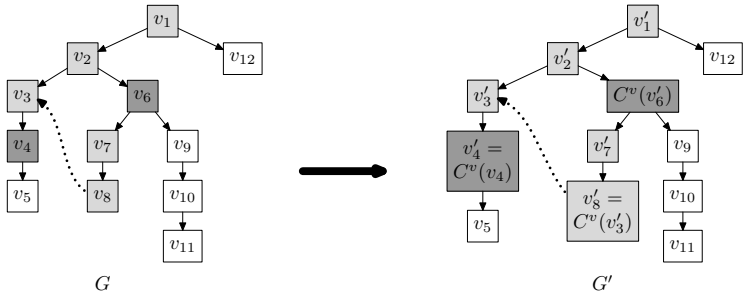


Figure 6.2: A d-graph G and its by-value decomposed d-graph G'

(see Section 5.3.2), and Q' is the corresponding query of G'. Then, $dEq(Q,Q')$ holds, under the definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6).

Proof. We prove this theorem by contradiction: assume $\neg dEq(Q,Q')$. Then there must exist one vertex $v_x \in G$, which depends on by-value copies of remote sequences[7], and its corresponding vertex in G' is v'_x such that v_x and v'_x are not deep-equal, *even if* each vertex v_z, on which v_x depends, is deep-equal to its corresponding vertex v'_z in G'. Formally, $\exists v_x \in G$ and $\exists v'_x \in G'$, such that:

$$(Cnd1)\ \neg dEq(v_x, v'_x)\ \wedge\ \forall v_z \in \{v_z | v_x \rightsquigarrow v_z\}.\forall v'_z \in \{v'_z | v'_x \rightsquigarrow v'_z\} : dEq(v_z, v'_z)$$
$$(Cnd2)\ (v_x \in I^v(G) \wedge \exists v_m \in V(G) \backslash V(G_{v_x}) \wedge v_x \overset{v}{\rightsquigarrow} v_m) \vee (\exists v_n \in I^v(G) \wedge v_x \rightsquigarrow v_n)$$

This can be illustrated more clearly by an example, as shown in Figure 6.2.

The figure shows a d-graph G with $I^v(G) = \{v_4, v_6\}$ at its left side. The varref-edge (v_8, v_3) determines that the vertices v_3 and v_8 represent a `Var` and a `VarRef` grammar rule, respectively. At the right side of the figure, it is the corresponding decomposed d-graph G', in which the effect of remote executions of v_4 and v_6 is indicated using by-value copy operators. As v_4 does not depend on any copied values, the effect of executing v_4 remotely is equivalent to executing v_4 locally, and return a by-value copy of the result. The effect of executing v_6 remotely is equivalent to executing v_6 locally on a by-value copy of its parameter v_8 and return a by-value copy of the result. Whether $dEq(v_6, v'_6)$ holds is the subject of this proof, as v'_6 depends on a copied parameter,

Thus, the set of vertices that could possibly return non-deep-equal results includes $\{v_1, v_2, v_3, v_6, v_7, v_8\}$. We need to check *(i)* if remote executions of v_6 would produce non-deep-equal results, because it depends on a by-value copied parameter; and *(ii)* if the vertices v_1, v_2, v_3, v_7 and v_8 return non-deep-equal results, because they depend on by-value copies of remote results. Note that, when checking, for instance, if $\neg dEq(v_6, v'_6)$ holds, we have the hypothesis: $\forall i \in 3,4,5,7..11 : dEq(v_i, v'_i)$. At this point, it is safe to assume, e.g., that $dEq(v_3, v'_3)$ holds, even if v_3 depends on v_4, because, otherwise, v_3 is already a v_x.

In the remainder of this proof, we examine each kind expression in the XCore grammar (Table 5.2) to see if it could be a v_x, i.e., expressions that produces non-deep-equal results, if any of their subexpressions are replaced by a deep-equal copy (either as parameters for remote executions, or as results of remote executions).

[7] Either v_x is in $I^v(G)$ and uses those sequences as its parameters; or those sequences are results of remote executions and are used by v_x.

6.3.9.1 Easy cases

The easy cases include `Literal`, variables, `ExprSeq`, `IfExpr`, `Typeswitch`, `OrderExpr`, value comparisons, and constructors. The results of these expressions are not determined by XML node identities or structural properties. Hence, it is easy to see that for an expression in this group, replacing any of its subexpressions by a deep-equal copy would not alter the result of this expression:

- Literals and variables are trivial cases.
- An `ExprSeq` simply concatenates the resulting sequences of its subexpressions, thus, replacing any of the subexpressions of the `ExprSeq` with a deep-equal copy will produce deep-equal result.
- The *boolean value* of the condition of an `IfExpr` determines the branch to be returned, and it could be replaced by a deep-equal copy without altering the decision. Since an `IfExpr` returns either its `then` or its `else` branch, the `IfExpr` would return deep-equal result if any of its branches is replaced with a deep-equal copy. Similar reasoning applies to `Typeswitch` and `OrderExpr`.
- Value comparisons compare the *literal* values of their operands.
- Constructors always produce fresh nodes by making a deep-copy of their operands.

Hence, v_x can not be an expression listed above.

6.3.9.2 `ForExpr` and `LetExpr`

Assume v_x is a `ForExpr`, according to condition $(Cnd1)$, the following statement must be true:

$$\frac{dEq(E_0, E_0') \qquad dEq(E_1(E_0), E_1'(E_0'))}{\neg dEq(\texttt{for \$}x\texttt{ in }E_0\texttt{ return }E_1,\ \texttt{for \$}x\texttt{ in }E_0'\texttt{ return }E_1')} \quad (t^v_{For})$$

Directly, the value of E_0 only determines the number of iterations of a `for`-loop, and it is trivial to see that E_0' leads to the same number of iterations. The result of a `for` expression is mainly determined by the result of its return clause E_1, which possibly has dependency on E_0 (indicated in the second premise by passing E_0 as a parameter to E_1). As we have pointed out earlier, at this point, it is safe to assume that $dEq(E_1(E_0), E_1'(E_0'))$ holds. Thus, replacing E_1 by E_1' implies that each iteration would produce a deep-equal sequence, and clearly, the concatenation of all those sequences is again deep-equal.

Hence, v_x can not be a `ForExpr`, and similarly, v_x could not be a `LetExpr`.

6.3.9.3 `NodeCmp` and `NodeSetExpr`

The by-value insertion condition ii (Section 5.4.1) states that there must not exist a valid decomposition point on which a `NodeCmp` (`is`, $\ll$, $\gg$) or a `NodeSetExpr` (`union`, `intersect`, `except`) depends. This prevents `NodeCmp` and `NodeSetExpr` expressions from using by-value copied results of remote executions. Condition ii also states that a decomposed `NodeCmp` or `NodeSetExpr` expression may not depends on any sequences outside the decomposed subgraph, preventing these expressions from using copied parameters. This contradicts with $(Cnd2)$ above, which state that a v_x must depend on at least one (copied) remote sequence.

Hence, v_x can not be a node comparison or a node set expression.

6.3.9.4 `StepExpr`

The by-value insertion condition i excludes `RevAxis` and `HorAxis` from depending on a (by-value) copied sequence, which contradicts with $(Cnd2)$ above. Thus, assume v_x is a `StepExpr`, according to $(Cnd1)$ above, the following statement must be true:

$$\frac{dEq(E_1, E_1') \quad \texttt{AS} \in \texttt{FwdAxis}}{\neg dEq(E_1/\texttt{AS::NT},\ E_1'/\texttt{AS::NT})} \qquad (t^v_{Step})$$

With Lemma 6.3.8 we have: $dEq(E_1/\texttt{AS::NT}, E_1'/\texttt{AS::NT})$, *iff* $\texttt{AS} \in FwdAxis$ and E_1 is disjunct and ordered. Thus, the statement t^v_{Step} is true if we can find an E_1 which is non-disjunct or unordered.

According to the static property analysis rules (Section 6.2), these expressions return non-disjunct results (regardless of the properties of their subexpressions): `ExprSeq`, `ForExpr`, `union`, `parent`, `ancestor`, `ancestor-or-self`, `descendant`, `descendant-or-self`, `preceding`, `preceding-sibling`, `following` and `following-sibling`, and `order by` expressions return unordered results. However, the by-value insertion condition iii forbids a `FwdAxis` step to depend on copies of any of these expressions. This implies that E_1 is always disjunct and ordered, thus, the statement t^v_{Step} is not true.

Hence, v_x can not be a `StepExpr`.

6.3.9.5 Function calls

Assume v_x is a `FunCall` to a built-in function $\mathcal{F}(r_1, ..., r_k)$[8], according to $(Cnd1)$ above, the following statement must be true:

$$\frac{\forall i \in 1..k : dEq(p_i, p_i')\ \mathcal{F} \in \texttt{built-in}}{\neg dEq(\mathcal{F}(p_1, ..., p_k),\quad \mathcal{F}(p_1', ..., p_k'))} \qquad (t^v_{BltIn})$$

A built-in function $\mathcal{F}$ would return non-deep-equal result if it needs to access values outside the subtrees of its parameters. Such built-in functions include `fn:id()`, `fn:idref()`, `fn:root()` and `fn:lang()`. However, the by-value insertion condition iv prevents any of these functions to depend on copied parameters.

Hence, v_x can not be a `FunCall` to a built-in function.

6.3.9.6 `TransformExpr`

Note that a `TransformExpr` is a read-only expression [58] and it is allowed to be decomposed by all three decomposition algorithms. Thus, we analyse here if v_x can be a `TransformExpr`.

Assume v_x is a `TransformExpr`, according to condition $(Cnd1)$, the following statement must be true:

$$\frac{\forall i \in 1..c : dEq(E_i, E_i') \qquad dEq(E_r, E_r')}{\begin{array}{r}\neg dEq(\texttt{copy}\ \$x_1 := E_1, ..., \$x_c := E_c\ \texttt{modify}\ E_m\ \texttt{return}\ E_r,\\ \texttt{copy}\ \$x_1 := E_1', ..., \$x_c := E_c'\ \texttt{modify}\ E_m\ \texttt{return}\ E_r')\end{array}} \qquad (t^v_{Trnsf})$$

A `TransformExpr` makes deep copies of its source expressions $E_1, ..., E_c$, which is equivalent to replacing these expressions with their by-value copies. The expression E_m in the `modify` clause must be an `UpdExpr` [58], which is not allowed to be decomposed by the XQUF insertion conditions (Section 5.7.1). The result of a `TransformExpr` is determined by the value of

[8] Our XCore grammar only allows calls to built-in functions (Section 5.3).

the `return` expression E_r. Clearly, replacing E_r with a by-value copy of it will not cause the `TransformExpr` to return non-deep-equal result. Thus, the statement t^v_{Trnsf} is not true.

Hence, v_x can not be a `TransformExpr`.

In summary, we were not able to find an v_x, which is not deep-equal to its corresponding vertex v'_x in G', while all vertices, on which v_x depends, are deep-equal to their corresponding vertices in G'. Thus, the assumption $\neg dEq(Q,Q')$ does not hold, which proves the correctness of the theorem. □

6.4 Correctness Proof of the By-Fragment Decomposition Algorithm

Definition 6.4.1. Canonical subsequence: *The canonical subsequence of a sequence $\vec{s}$, denoted $\boldsymbol{\zeta(\vec{s})}$, consists of a single occurrence of all node-typed items in $\vec{s}$ that are not a descendant of another node-typed item in $\vec{s}$, sorted by their document order. Formally, nodes in $\zeta(\vec{s})$ satisfy the following conditions:*

- *exist:* $\forall s_j \in \zeta(\vec{s}) : \exists s_i \in \vec{s} \wedge s_i \texttt{ is } s_j$
- *unique:* $\forall s_i \in \vec{s}: \exists s_j \in \{\zeta(\vec{s})/\texttt{d-o-s::node()}\} \wedge \texttt{exactly-one}(s_j) \wedge s_i \texttt{ is } s_j$
- *disjunct:* $\forall i,j \in 1..|\zeta(\vec{s})| : i \neq j \Rightarrow \zeta(\vec{s})[i] \notin \{\zeta(\vec{s})[j]/\texttt{d-o-s::node()}\}$
- *ordered:* $\forall k \in 2..|\zeta(\vec{s})| : \zeta(\vec{s})[k-1] \ll \zeta(\vec{s})[k]$

Definition 6.4.2. By-fragment copy operator C^f: *Let $\vec{s}$ be an XQuery sequence that may contain duplicates or overlapping nodes. A by-fragment-copy, denoted $C^f(\vec{s})$, of $\vec{s}$ is a pair $\langle \vec{\mathcal{S}}, \vec{\mathcal{F}} \rangle$, where*

- *$\vec{\mathcal{F}}$ is a set of fresh XML fragments, created by making a by-value copy (i.e., a deep-copy) of the canonical subsequence of $\vec{s}$, i.e., $\vec{\mathcal{F}} = C^v(\zeta(\vec{s}))$;*
- *$\vec{\mathcal{S}}$ is the return sequence of the by-fragment copy operator C^f. It is a one-to-one mapping of the items in $\vec{s}$ constructed according to the rules below:*

$$\forall i \in 1..|\vec{s}|, \begin{cases} \vec{\mathcal{S}}[i] = \vec{s}[i], \textit{ if } \texttt{type}(\vec{s}[i]) \notin \vec{\tau}; \\ \vec{\mathcal{S}}[i] \texttt{ is } \vec{\mathcal{F}}[j]/\texttt{d-o-s::node()}[k], \textit{where } \{j,k \mid \vec{s}[i] \texttt{ is } \zeta(\vec{s})[j]/\texttt{d-o-s::node()}[k]\}. \end{cases}$$

  That is, if $\vec{s}[i]$ is a literal value, $\vec{\mathcal{S}}[i]$ gets the value of $\vec{s}[i]$; otherwise, $\vec{\mathcal{S}}[i]$ is a reference to the node in $\vec{\mathcal{F}}$ that corresponds to $\vec{s}[i]$.

Under the pass-by-fragment semantics, the semantics of executing an expression on a remote peer is equal to first replacing each parameter of the expression with a by-fragment copy, then executing the expression (using the by-fragment copies on the local peer), and finally returning a by-fragment copy of the result. We use the notations $C^f(\vec{s}).\vec{\mathcal{S}}$ and $C^f(\vec{s}).\vec{\mathcal{F}}$ to refer to the sets $\vec{\mathcal{S}}$ and $\vec{\mathcal{F}}$ that belong to $C^f(\vec{s})$. However, since only $\vec{\mathcal{S}}$ is the return value of C^f, we use the shorthand $C^f(\vec{s})$ to refer to $C^f(\vec{s}).\vec{\mathcal{S}}$, if there is no ambiguity.

Definition 6.4.3. By-fragment mapping function m_f: *Let $\vec{e}$ be a sequence of XML nodes and $C^f(\vec{e})$ its by-fragment copy. The by-fragment mapping function m_f maps each node e_x in $\{\vec{e}/\texttt{d-o-s::node()}\}$ to exactly one node e'_x in $\{C^f(\vec{e}).\vec{\mathcal{F}}/\texttt{d-o-s::node()}\}$. The reverse*

function m_f^{-1} maps each node e'_x in $\{C^f(\vec{e}).\vec{\mathcal{S}}/\texttt{d-o-s::node()}\}$ to exactly one node e_x in $\{\zeta(\vec{e})/\texttt{d-o-s::node()}\}$. Formally, the results of m_f and m_f^{-1} are defined as the following:

$$\forall e_x \in \{\vec{e}/\texttt{d-o-s::node()}\}: e_x \texttt{ is } \zeta(\vec{e})[i]/\texttt{d-o-s::node()}[j] \mapsto m_f(e_x) \texttt{ is } C^f(\vec{e}).\vec{\mathcal{F}}[i]/\texttt{d-o-s::node()}[j]$$

$$\forall e'_x \in \{C^f(\vec{e})/\texttt{d-o-s::node()}\}: e'_x \texttt{ is } C^f(\vec{e}).\vec{\mathcal{F}}[i]/\texttt{d-o-s::node()}[j] \mapsto m_f^{-1}(e'_x) \texttt{ is } \zeta(\vec{e})[i]/\texttt{d-o-s::node()}[j]$$

We call $m_f(e_x)$ by-fragment-mapping of e_x in $C^f(\vec{e})$, and vice versa.

Lemma 6.4.4. Mapped raw results of `FwdAxis` steps on $\vec{e}$ and $C^f(\vec{e})$: *Let $\vec{e}$ be a sequence of XML nodes and $C^f(\vec{e})$ its by-fragment copy. The raw results of applying multiple consecutive `FwdAxis` steps on $\vec{e}$ and $C^f(\vec{e})$ are the by-fragment-mapping of each other, i.e.:*

$$C^f(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n) = m_f(\vec{e}/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n) \wedge \forall as_i \in FwdAxis$$

Proof. The proof is similar to the proof of Lemma 6.3.4. □

Lemma 6.4.5. By-fragment node relationships: *Let $\vec{e}$ be a sequence of XML nodes and $C^f(\vec{e})$ its by-fragment copy. Then, the relationship between any two nodes u and w in the subtrees rooted at the nodes in $\vec{e}$ is the same as the relationship between their corresponding nodes u' and w' in $C^f(\vec{e})$. Formally:*

$$\forall u,w \in \{\vec{e}/\texttt{d-o-s::node()}\}.\forall u',w' \in \{C^f(\vec{e})/\texttt{d-o-s::node()}\}: u' = m_f(u) \wedge w' = m_f(w) \mapsto R(u,w) = R(u',w')$$

Proof. Since u and w are both nodes in the subtrees rooted at nodes in $\vec{e}$, with Definition 6.4.1, we can assume:

(h1) u `is` $\zeta(\vec{e})[i]/\texttt{d-o-s::node()}[k]$, w `is` $\zeta(\vec{e})[j]/\texttt{d-o-s::node()}[l]$

where $i,j \in 1..|\zeta(\vec{e})|$ and $k,l \in 1..|\zeta(\vec{e})[i]/\texttt{d-o-s::node()}|$. With Definition 6.4.3 and $u' = m_f(u), w' = m_f(w)$, we have:

(h2) u' `is` $C^f(\vec{e}).\vec{\mathcal{F}}[i]/\texttt{d-o-s::node()}[k]$, w' `is` $C^f(\vec{e}).\vec{\mathcal{F}}[j]/\texttt{d-o-s::node()}[l]$

There are two possibilities: $i = j$ (i.e., u,w belong to the same subtree in $\zeta(\vec{e})$) or $i \neq j$ (i.e., u,w belong to different subtrees in $\zeta(\vec{e})$). Below we consider each case.

Case 1: i = j

With Definition 6.4.2 we have:

(h3) $dEq(C^f(\vec{e}).\vec{\mathcal{F}}[i], \zeta(\vec{e})[i])$

With (h1-3) we have:

(t1) $R(u,w) = R(u',w')$

Thus, the lemma holds when u and w belong to the same subtree in $\zeta(\vec{e})$.

Case 2: i ≠ j

With Definition 6.4.2 we have:

(h4) $dEq(C^f(\vec{e}).\vec{\mathcal{F}}, \zeta(\vec{e}))$

With Definition 6.4.1 we have:

(h5) $\mu(\zeta(\vec{e})), \sigma(\zeta(\vec{e}))$

With (h4,5) we have:

(h6) $i < j \mapsto \zeta(\vec{e})[i] \ll \zeta(\vec{e})[j] \wedge C^f(\vec{e}).\vec{\mathcal{F}}[i] \ll C^f(\vec{e}).\vec{\mathcal{F}}[i]$

(h7) $i > j \mapsto \zeta(\vec{e})[i] \gg \zeta(\vec{e})[j] \wedge C^f(\vec{e}).\vec{\mathcal{F}}[i] \gg C^f(\vec{e}).\vec{\mathcal{F}}[i]$

With (h1,2) and (h6,7), we have:

(t2) $R(u,w) = R(u',w')$

Thus, the lemma also holds when u and w belong to different subtrees in $\zeta(\vec{e})$. □

Lemma 6.4.6. By-fragment deep-equal `ddo()`: *Let $\vec{e}$ be a sequence of XML nodes, and $C^f(\vec{e})$ its by-fragment copy. Let $\vec{a}$ be a sequence containing XML nodes in the subtree rooted at $\vec{e}$, i.e., $\forall a_i \in \{\vec{e}/\texttt{d-o-s::node()}\}$, and $\vec{b}$ the by-fragment mapping of $\vec{a}$ in $C^f(\vec{e})$, then $dEq(ddo(\vec{a}), ddo(\vec{b}))$.*

Proof. With Lemma 6.4.5, this lemma can be proven using a similar reasoning as that of the proof of Lemma 6.3.6. □

Lemma 6.4.7. By-fragment deep-equal `FwdAxis`: *Let $\vec{e}$ be a single sequence of XML nodes and $C^f(\vec{e})$ its by-fragment copy. The results of applying any number of consecutive `FwdAxis` steps on $\vec{e}$ and $C^f(\vec{e})$ are deep equal to each other. Formally:*

$$dEq(\vec{e}/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n, C^f(\vec{e})/\texttt{AS}_1\texttt{::NT}_1/.../\texttt{AS}_n\texttt{::NT}_n) \wedge \forall i \in 1..n : \texttt{AS}_i \in FwdAxis$$

Proof. With Lemma 6.4.4, we have:

(h1) $C^f(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n) \equiv m_f(\vec{e}/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n)$

With (h1) and Lemma 6.4.6, we have:

(h2) $dEq(\texttt{ddo}(\vec{e}/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n), \texttt{ddo}(C^f(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n))$

With Definition 6.1.15, we have:

(h3) $\vec{e}/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n \equiv \texttt{ddo}(\vec{e}/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n)$

(h4) $C^f(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n \equiv \texttt{ddo}(C^f(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n)$

With (h2), (h3) and (h4), we have:

(t) $dEq(\texttt{ddo}(\vec{e}/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n), \texttt{ddo}(C^f(\vec{e})/\texttt{as}_1\texttt{::NT}_1/.../\texttt{as}_n\texttt{::NT}_n))$ □

The correctness of the by-fragment decomposition algorithm is proven as follows:

Theorem 6.4.8. By-Fragment Decomposition Correctness: *Let Q be a normal read-only XCore query (i.e., without any XRPC expressions) and G the corresponding d-graph. $I^f(G) \subset G$ is the (non-empty) set of decomposition points validated by the by-fragment insertion conditions. Let G' be the d-graph derived by doing an `XRPCExpr` insertion above each vertex in $I^f(G)$ (Section 5.3.2), and Q' be the corresponding query of G'. Then, $dEq(Q,Q')$ holds, under the definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6).*

Proof. We prove this theorem using a similar strategy as the proof of the correctness of the conservative decomposition (Theorem 6.3.9). Assume $\neg dEq(Q,Q')$, then we need to find a vertex $v_x \in G$ which depends on by-fragment copies of remote sequences, with its corresponding vertex $v'_x \in G'$, such that v_x and v'_x are not deep-equal, *even if* each vertex v_z, on which v_x depends, is deep-equal to its corresponding vertex v'_z in G'. Formally, $\exists v_x \in G$ and $\exists v'_x \in G'$, such that:

$$(Cnd1)\ \neg dEq(v_x, v'_x) \wedge \forall v_z \in \{v_z | v_x \rightsquigarrow v_z\}.\forall v'_z \in \{v'_z | v'_x \rightsquigarrow v'_z\} : dEq(v_z, v'_z)$$
$$(Cnd2)\ (v_x \in I^f(G) \wedge \exists v_m \in V(G) \backslash V(G_{v_x}) \wedge v_x \overset{v}{\rightsquigarrow} v_m) \vee (\exists v_n \in I^f(G) \wedge v_x \rightsquigarrow v_n)$$

Compared to the conservative algorithm, the by-fragment decomposition algorithm allows, *in addition*, the following kind of expressions to be decomposed.

- `NodeCmp` and `NodeSetExpr` expressions may depend on copied subexpressions, iff the `NodeCmp` and `NodeSetExpr` expressions do not depend on two different applications of `fn:doc()` with the same URI (condition II). This change is considered in 6.4.8.1 and 6.4.8.2.

- `FwdAxis` steps[9] may depend on copied subexpressions containing `OrderExpr`s or axis steps including `ancestor`, `ancestor-or-self`, `preceding`, `following`, `descendant` and `descendant-or-self`. `FwdAxis` steps may also depend on copied subexpressions containing `ForExpr`s, `ExprSeq`s or `NodeSetExpr`s, iff such subexpressions do not depend on two different applications of `fn:doc()` with the same URI (condition III). This change is considered in 6.4.8.3.

In Theorem 6.3.9, it has already been proven that v_x can not be one of the expressions which are allowed by the conservative algorithm to depend on (by-value) copied subexpressions. In this proof, we only consider if v_x can be one of the expressions which are *additionally* allowed by the by-fragment decomposition algorithm to depend on (by-fragment) copied subexpressions. Note that the analysis below is two-sided, i.e., a subexpression could be copied either because it is used as a parameter of a decomposed expression, or because the subexpression itself is a decomposed expression whose result is used further in the query.

6.4.8.1 `NodeCmp`

Assume v_x is a `NodeCmp`, according to $(Cnd1)$ above, the following statement must be true:

$$\frac{dEq(E_0,E_0') \quad dEq(E_1,E_1') \quad \neg\texttt{hasMatchingDoc}(E_0,E_1)}{\neg dEq(E_0 \boxdot E_1,\ \ E_0' \boxdot E_1')} \qquad (t^f_{Ncmp})$$

where the symbol $\boxdot$ represents the node comparison operators: `is`, $\ll$ and $\gg$.

The premise $\neg\texttt{hasMatchingDoc}(E_0,E_1)$ implies that E_0 and E_1 contain nodes from different documents. Thus, the query "E_0 `is` E_1" will always return `false`. The decomposed query "E_0' `is` E_1'" also always returns `false`, since E_0' and E_1' refer to nodes in different fragments. The result of "$E_0 \ll E_1$" is implementation dependent. It is either `true` or `false`. The query "$E_0' \ll E_1'$" also returns either `true` or `false`, as it depends on two distinct trees $C^v(\zeta(E_0))$ and $C^v(\zeta(E_1))$. Recall that the definition of deep-equal queries (Definition 6.1.6) takes into account the XQuery features with implementation freedom, and regards "$E_0' \ll E_1'$" to be deep-equal to "$E_0 \ll E_1$", if "$E_0' \ll E_1'$" returns a value that also could be returned by "$E_0 \ll E_1$". Thus, we have $dEq(E_0 \ll E_1, E_0' \ll E_1')$. Similar reasoning holds for "$E_0 \gg E_1$".

Hence, v_x can not be a `NodeCmp` whose subexpressions depend on calls to `fn:doc()` with different URIs.

6.4.8.2 `NodeSetExpr`

Assume v_x is a `NodeSetExpr`, according to $(Cnd1)$ above, the following statement must hold:

$$\frac{dEq(E_0,E_0') \quad dEq(E_1,E_1') \quad \neg\texttt{hasMatchingDoc}(E_0,E_1)}{\neg dEq(E_0 \boxdot E_1,\ \ E_0' \boxdot E_1')} \qquad (t^f_{Nset})$$

where the symbol $\boxdot$ represents the node set operators: `union`, `intersect` or `except`.

The premise $\neg\texttt{hasMatchingDoc}(E_0,E_1)$ implies that E_0 and E_1 contain nodes from different documents. The expression "E_0 `union` E_1" returns either ($\texttt{ddo}(E_0)$, $\texttt{ddo}(E_1)$) or ($\texttt{ddo}(E_1)$, $\texttt{ddo}(E_0)$). Similarly, the decomposed expression "E_0' `union` E_1'" returns either ($\texttt{ddo}(E_0')$, $\texttt{ddo}(E_1')$) or ($\texttt{ddo}(E_1')$, $\texttt{ddo}(E_0')$). By Lemma 6.4.6, we have $dEq(\texttt{ddo}(E_0),\texttt{ddo}(E_0'))$ and $dEq(\texttt{ddo}(E_1),\texttt{ddo}(E_1'))$, which implies:

[9] XPath steps on other axes are excluded by the by-fragment insertion condition I.

$$dEq((\texttt{ddo}(E_0),\texttt{ddo}(E_1)),(\texttt{ddo}(E'_0),\texttt{ddo}(E'_1))), \text{ and}$$
$$dEq((\texttt{ddo}(E_1),\texttt{ddo}(E_0)),(\texttt{ddo}(E'_1),\texttt{ddo}(E'_0)))$$

With Definition 6.1.6, we have $dEq(E_0 \texttt{ union } E_1, \quad E'_0 \texttt{ union } E'_1)$. Thus, the statement $\neg dEq(E_0 \boxdot E_1, E'_0 \boxdot E'_1)$ does not hold in this case.

If $\boxdot$ represents `intersect` or `except`, with $\neg\texttt{hasMatchingDoc}(E_0,E_1)$, the expression "$E_0$ `intersect` E_1" returns the empty sequence, and the expression "E_0 `except` E_1" returns $\texttt{ddo}(E_0)$. Similarly, the decomposed expression "E'_0 `intersect` E'_1" returns the empty sequence, and "E'_0 `except` E'_1" returns $\texttt{ddo}(E'_0)$. By Lemma 6.4.6, we have that $dEq(\texttt{ddo}(E_0), \texttt{ddo}(E'_0))$. Thus, the statement $\neg dEq(E_0 \boxdot E_1, E'_0 \boxdot E'_1)$ does not hold in these cases.

Hence, v_x can not be a `NodeSetExpr`, whose subexpressions depend on calls to `fn:doc()` with different URIs.

6.4.8.3 `FwdAxis` steps

Assume v_x is a `FwdAxis`, according to $(Cnd1)$ above, at least one of the following statements must be true:

$$\frac{\begin{array}{c} dEq(E_1,E'_1) \\ E_1 \in \{\texttt{AxisStep, OrderExpr}\} \\ \texttt{AS} \in \texttt{FwdAxis} \\ \nexists E_i : E_1 \rightsquigarrow E_i \wedge E_i \in \{\texttt{ForExpr, ExprSeq, NodeSetExpr}\} \end{array}}{\neg dEq(E_1/\texttt{AS::NT},\ E'_1/\texttt{AS::NT})} \quad (t^f_{Step_1})$$

$$\frac{\begin{array}{c} dEq(E_1,E'_1) \\ E_1 \in \{\texttt{ForExpr, ExprSeq, NodeSetExpr}\} \\ \texttt{AS} \in \texttt{FwdAxis} \\ \neg\texttt{hasMatchingDoc}(E_1,E_1) \end{array}}{\neg dEq(E_1/\texttt{AS::NT},\ E'_1/\texttt{AS::NT})} \quad (t^f_{Step_2})$$

In Lemma 6.4.7, it is proven that the results of applying a *FwdAxis* on a *single* sequence $\vec{v}$ (i.e., E_1 does not depend on any subexpressions E_i that combine two sequences in their results; this case is covered by the statement $t^f_{Step_2}$) and its by-fragment copy $C^f(\vec{v})$ are deep equal to each other. Note that Lemma 6.4.7 holds for any XML node sequence, regardless of their distinct, disjunct and ordered properties. Thus, the statement $t^f_{Step_1}$ is not true.

If the result of E_1 is a combination of two single subsequences (e_0,e_1), then the result of E'_1 is a combination of the by-fragment copies of the two subsequences (e'_0,e'_1). The statement $t^f_{Step_2}$ would be true, if we are not able to eliminate duplicate nodes that appear in both e'_0 and e'_1, or sort e'_0 and e'_1 in the same document order as e_0 and e_1. As described earlier (Section 5.2, *Problem 4*), this could only happen, if e_0 and e_1 contain nodes from the *same* document on a peer. There are three kinds of expressions that return combined subsequences: `ForExpr`, `ExprSeq` and `NodeSetExpr`. For these expressions, the problem with "mixed-call" is guarded by the predicate `hasMatchingDoc()` in the by-fragment insertion condition III, which simply forbids a *FwdAxis* to depends on a E_1 that contain multiple `fn:doc()` calls to access the same document. Thus, the statement $t^f_{Step_2}$ does not hold.

Hence, v_x can not be a `FwdAxis` that depends on an `OrderExpr` or an `AxisStep`, or on a `ForExpr`, an `ExprSeq` or a `NodeSetExpr` which do not access the same XML document with multiple `fn:doc()`.

In summary, we were not able to find a v_x which is not deep-equal to its corresponding vertex v'_x in G', while all vertices on which v_x depends are deep-equal to their corresponding vertices in G'. Thus, the assumption $\neg dEq(Q,Q')$ does not hold, which proves the correctness of the theorem. □

6.5 Correctness Proof of the By-Projection Decomposition Algorithm

When the by-projection decomposition algorithm is used, the set of *projection paths* $\vec{\mathcal{P}}_{v_i}$ are calculated for each vertex v_i in a d-graph which consists of a set of used paths $\vec{\mathcal{P}}_{v_i}.\vec{\mathcal{U}}$ and a set of returned paths $\vec{\mathcal{P}}_{v_i}.\vec{\mathcal{R}}$ (Section 5.6). Each path in $\vec{\mathcal{P}}_{v_i}$ is a `ProjectionPath` (Table 5.6) that can contain XPath steps on *any* axes and any of the special built-in functions `root()`, `id()` and `idref()`.

The concept *relative projection paths* (i.e., path suffixes) is always defined between two vertices. Assume v_i and v_j are vertices in a d-graph with $v_i \rightsquigarrow v_j$. We use $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}$ to denote the set of *relative projection paths between v_i and v_j* which consists of a set of *relative used paths* $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{U}}$ and a set of *relative returned paths* $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{U}}$, computed using the function `allSuffixes()` (Section 5.6.2):

$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{U}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{U}})$$
$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{R}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{R}})$$

However, beside the path $v_i \rightsquigarrow v_j$, v_i could also depend on v_j via other paths, say $v_i \rightsquigarrow v_k \rightsquigarrow v_j$. This happens if v_j is a variable declaration (or a subexpression of a variable declaration) and is referred to by v_k. For instance, as shown in the left part of Figure 6.4, v_1 depends on v_4 both via the path $v_1 \rightsquigarrow v_2 \rightsquigarrow v_4$ and via the path $v_1 \rightsquigarrow v_6 \rightsquigarrow v_4$. Thus, we use $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k \rightsquigarrow v_j}$ to denote the set of relative projection path that v_i will apply on v_j *via* its subexpression v_k. $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k \rightsquigarrow v_j}$ is computed using the function `allSuffixesVia()` (Section 5.6.2):

$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k \rightsquigarrow v_j}.\vec{\mathcal{U}} = \texttt{allSuffixesVia}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_k}.\vec{\mathcal{U}},\ \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{U}})$$
$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k \rightsquigarrow v_j}.\vec{\mathcal{R}} = \texttt{allSuffixesVia}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_k}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{R}})$$

The paths $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{U}}$ and $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{R}}$ *overestimate* the set of nodes, with respect to v_j, that will be used and respectively returned by v_i. Intuitively, if v_j represents an expression executed on a peer different than the peer, on which v_i is executed[10], shipping the nodes determined by $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{U}}$ and $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{R}}$ together with the result of v_j could enable correct evaluation of v_i. In other words, evaluating v_i on those shipped nodes produces deep-equal results to those of evaluating v_j in the original query. The main task of this section is to give a formal proof for this statement. But first, let us use an example to make the differences and relationships between projection paths and relative projection paths more clear.

Example 6.5.1. *Figure 6.3 shows an abstract d-graph in which three vertices are shown explicitly. The root vertex of this d-graph is v_i. The vertex v_k depends on v_j via a varref varref edge (indicated using a dotted arrow). Thus, starting from v_i, there are two paths with which we can reach v_j, i.e., $v_i \rightsquigarrow v_j$ and $v_i \rightsquigarrow v_k \rightsquigarrow v_j$. The exact projection paths of all three vertices are also shown in Figure 6.3.*

[10] This includes both cases: *(i)* v_j is a parameter of the valid decomposition point v_i, or *(ii)* v_j is a valid decomposition point, whose result is used by v_i.

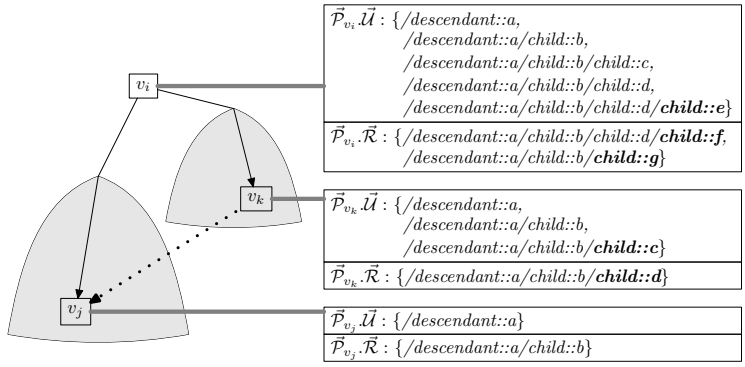


Figure 6.3: A d-graph with projection paths

It can be seen that v_k applies two XPath steps on (the result of) v_j: a `child::c` *step as a used path and a* `child::d` *step as a returned path. Then, v_i applies two XPath steps on v_k: a* `child::e` *step as a used path and a* `child::f` *step as a returned path. This implies that v_i applies, via v_k, the following XPath steps on v_j: a used path* `child::d/child::e` *and a returned path* `child::d/child::f`. *Since we also have $v_i \rightsquigarrow v_j$, v_i can also directly (i.e., not via v_k) apply XPath steps on v_j, which is, in this example, the* `child::g` *step. Thus, we can compute the following relative projection paths among these three vertices:*

$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{U}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_k}.\vec{\mathcal{U}}) = \{\texttt{child::c}\}$$
$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{R}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_k}.\vec{\mathcal{R}}) = \{\texttt{child::d}\}$$

$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k}.\vec{\mathcal{U}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_k}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{U}}) = \{\texttt{child::e}\}$$
$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k}.\vec{\mathcal{R}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_k}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{R}}) = \{\texttt{child::f}\}$$

$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{U}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{U}}) = \{\texttt{child::c}, \texttt{child::d}, \texttt{child::d/child::e}\}$$
$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{R}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{R}}) = \{\texttt{child::d/child::f}, \texttt{child::g}\}$$

$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k \rightsquigarrow v_j}.\vec{\mathcal{U}} = \texttt{allSuffixesVia}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_k}.\vec{\mathcal{U}}, \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{U}}) = \{\texttt{child::c}, \texttt{child::d}, \texttt{child::d/child::e}\}$$
$$\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k \rightsquigarrow v_j}.\vec{\mathcal{R}} = \texttt{allSuffixesVia}(\vec{\mathcal{P}}_{v_j}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_k}.\vec{\mathcal{R}}, \vec{\mathcal{P}}_{v_i}.\vec{\mathcal{R}}) = \{\texttt{child::d/child::f}\}$$

Note the difference between $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_j}.\vec{\mathcal{R}}$ and $\vec{\mathcal{P}}^{rel}_{v_i \rightsquigarrow v_k \rightsquigarrow v_j}.\vec{\mathcal{R}}$, which indicates that the step `child::g` *is not applied on the result of v_k. Thus, if v_k would be decomposed (v_j is then a parameter of this remote expression), we do not need to project the* `g` *child nodes of v_j for the request message.*

From the above example, the following property for the projection paths can be deduced[11]:

Property 6.5.2. Increasing projection paths: *Starting from leaf vertices[12], an expression always propagates all projection paths of all its subexpression(s). Thus, an existing projection path is never removed, only a new projection path is added, when there is an XPath step.*

The definition of the by-projection-copy operator C^p is based on the concept of relative projection paths.

[11] The property for the projection paths can also be deduced by examining the static properties analysis rules in Section 6.2

[12] By ignoring the varref edges, a d-graph has a tree shape. The leaf vertices are then those vertices without any outgoing edges.

Definition 6.5.3. By-projection-copy operator C^p: *The by-projection-copy operator takes as its input a pair $\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle$, where:*

- *$\vec{s}$ is an XQuery sequence that may contain duplicates or overlapping nodes,*
- *$\vec{\mathcal{P}}^{rel}$ is the set of relative projection paths that will be applied on $\vec{s}$;*

and produces as its output a pair $\langle \vec{\mathcal{S}}, \vec{\mathcal{F}} \rangle$, where:

- *$\langle \vec{\mathcal{S}}, \vec{\mathcal{F}} \rangle = C^f(\vec{s})$, if $\vec{\mathcal{P}}^{rel} = \emptyset$; otherwise*
- *$\vec{\mathcal{F}}$ is the set of projected XML fragments which is the projection of $\vec{\mathcal{P}}^{rel}$ on $\vec{s}$, i.e., $\vec{\mathcal{F}} = \mathfrak{P}(\vec{s}, \vec{\mathcal{P}}^{rel})$; $\vec{\mathcal{S}}$ is the return sequence of the by-projection-copy operator C^p. It is a one-to-one mapping of the items in $\vec{s}$ constructed as follows:*

$$\forall i \in 1..|\vec{s}| : \vec{\mathcal{S}}[i] \;\; \texttt{is} \;\; \vec{\mathcal{F}}[j]/\texttt{d-o-s::node()}[k], \text{ where } \{j,k \mid \vec{s}[i] \;\texttt{is}\; \vec{\mathcal{D}}[j]/\texttt{d-o-s::node()}[k]\}.$$

That is, $\vec{\mathcal{S}}[i]$ is a reference to the k-th node in the projected document $\vec{\mathcal{D}}[j]$[13] which corresponds to the node referred by $\vec{s}[i]$ in the original document.

We use $C^p(\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle)$ to denote a by-projection copy of $\vec{s}$ with relative projection paths $\vec{\mathcal{P}}^{rel}$. The outputs of C^p are referred to as $C^p(\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle).\vec{\mathcal{S}}$ and $C^p(\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle).\vec{\mathcal{F}}$. If there is no ambiguity, $C^p(\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle).\vec{\mathcal{S}}$ is abbreviated as $C^p(\vec{s})$. Since all node typed items in the return sequence $C^p(\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle).\vec{\mathcal{S}}$ refer to the projected XML fragments $\vec{\mathcal{F}}$ of $\vec{\mathcal{P}}^{rel}$ on $\vec{s}$, it is trivial to see that the results of applying $\vec{\mathcal{P}}^{rel}$ on both $\vec{s}$ and $C^p(\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle).\vec{\mathcal{S}}$ are deep-equal. Hence, we define the following property for the by-projection operator C^p:

Property 6.5.4. C^p properties: $dEq^{\vec{\mathcal{P}}^{rel}}(\vec{s}, C^p(\langle \vec{s}, \vec{\mathcal{P}}^{rel} \rangle).\vec{\mathcal{S}})$

Under the pass-by-projection semantics, the semantics of executing an expression v_i on a remote peer is to first replace each of its parameter v_{param_k} with a by-projection copy, with $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_i \rightsquigarrow v_{param_k}}$ (called: *the relative projection paths of the parameter v_{param_k}*), where v_{root} is the root vertex of the d-graph containing v_i. Then the expression v_i is executed on the local peer using the by-projection copies of its parameters, and finally a by-projection copy of the result is returned, with $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_i}$ (called: *the relative projection paths of the expression v_i*). Thus, when computing the projection for a remote expression, we always compare its returned paths with the projection paths of the root vertex v_{root} (either directly, or via a third vertex).

Example 6.5.5. *The left part of Figure 6.4 shows a d-graph G, in which each vertex is annotated with a set of projection paths. The right part of Figure 6.4 shows what the decomposed d-graph G′ looks like (the paths annotations are omitted), expressed using by-projection copies, if, for instance, v_4 and v_6 are pushed to remote peers. The vertex v_4 has no parameter, so only its result is projected using the relative projection paths $\vec{\mathcal{P}}^{rel}_{v_1 \rightsquigarrow v_4}$ of v_4, where:*

$$\vec{\mathcal{P}}^{rel}_{v_1 \rightsquigarrow v_4}.\vec{\mathcal{U}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_4}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_1}.\vec{\mathcal{U}})$$
$$\vec{\mathcal{P}}^{rel}_{v_1 \rightsquigarrow v_4}.\vec{\mathcal{R}} = \texttt{allSuffixes}(\vec{\mathcal{P}}_{v_4}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_1}.\vec{\mathcal{R}})$$

For vertex v_6, its parameter v_3 is first projected using the relative paths $\vec{\mathcal{P}}^{rel}_{v_1 \rightsquigarrow v_6 \rightsquigarrow v_3}$, where:

$$\vec{\mathcal{P}}^{rel}_{v_1 \rightsquigarrow v_6 \rightsquigarrow v_3}.\vec{\mathcal{U}} = \texttt{allSuffixesVia}(\vec{\mathcal{P}}_{v_3}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_6}.\vec{\mathcal{U}},\ \vec{\mathcal{P}}_{v_1}.\vec{\mathcal{U}})$$
$$\vec{\mathcal{P}}^{rel}_{v_1 \rightsquigarrow v_6 \rightsquigarrow v_3}.\vec{\mathcal{R}} = \texttt{allSuffixesVia}(\vec{\mathcal{P}}_{v_3}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_6}.\vec{\mathcal{R}},\ \vec{\mathcal{P}}_{v_1}.\vec{\mathcal{R}})$$

Then, the result of v_6 is projected using the relative paths $\vec{\mathcal{P}}^{rel}_{v_1 \rightsquigarrow v_6}$.

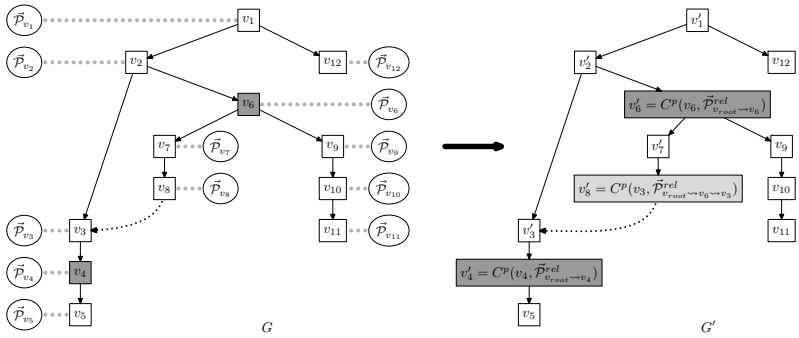


Figure 6.4: A d-graph G and its by-projection decomposed d-graph G'

The correctness of the by-projection decomposition algorithm is proven as follows:

Theorem 6.5.6. By-Projection Decomposition Correctness: *Let Q be a normal read-only XCore query (i.e., without any XRPC expressions) and G the corresponding d-graph. $I^p(G) \subset G$ is the (non-empty) set of decomposition points validated by the by-projection insertion conditions. Let G' be the d-graph derived by doing an* `XRPCExpr` *insertion above each vertex in $I^p(G)$ (Section 5.3.2), and let Q' be the corresponding query of G'. Then, $dEq(Q,Q')$ holds under the definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6).*

Proof. We prove this theorem by contradiction, similar to the proof of the correctness theorem of the conservative decomposition (Theorem 6.3.9). Since the relative projection paths of parameters of remote expressions are computed slightly differently than the relative projection paths of remote expressions, we split this proof into two parts. In Part I, we temporarily ignore this difference and assume that the relative projection paths of each vertex v_i in a d-graph are computed in the same way, i.e., $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_i}$. Under this assumption, we search for a vertex v_x, which could invalidate the statement $dEq(Q,Q')$. If we are not able to find such a v_x in Part I, we check in Part II if projecting parameters of a remote expression, using relative projection paths computed *via* the remote expression, could invalidate the statement $dEq(Q,Q')$.

Part I

Assume $\neg dEq(Q,Q')$. There must exist one vertex $v_x \in G$ which depends on by-projection copies of remote sequences[14] with its corresponding vertex $v'_x \in G'$, such that v_x and v'_x are not by-projection equal with respect to $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_x}$, *even if* each vertex v_z on which v_x depends is by-projection equal with respect to $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_x}$ to its corresponding vertex v'_z in G'. Formally, $\exists v_x \in G$ and $\exists v'_x \in G'$, such that:

$$(Cnd1)\ \neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_x}}(v_x, v'_x)\ \wedge\ \forall v_z \in \{v_z | v_x \rightsquigarrow v_z\}. \forall v'_z \in \{v'_z | v'_x \rightsquigarrow v'_z\} : dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_z}}(v_z, v'_z)$$
$$(Cnd2)\ (v_x \in I^p(G) \wedge \exists v_m \in V(G) \backslash V(G_{v_x}) \wedge v_x \overset{v}{\rightsquigarrow} v_m) \vee (\exists v_n \in I^p(G) \wedge v_x \rightsquigarrow v_n)$$

In the remainder of Part I, we examine each kind of expression[15] in the XCore grammar

[13] Since XPath steps can only be applied on XML nodes, we assume that $\vec{s}$ in this case only contain XML nodes.

[14] Either v_x is in $I^p(G)$ and uses those sequences as its parameters; or those sequences are results of remote execution and are used by v_x.

[15] But with focus on expressions whose result sequence can contain XML node-typed items, since it is trivial to see that copying a literal value (under any of our three semantics) does not alter query result.

(Table 5.2) to see if it could be a v_x that satisfies $(Cnd1)$ and $(Cnd2)$ above. For brevity, we call two corresponding vertices from G and G' to be by-projection equal without explicitly mentioning the relative projection paths with respect to which they are deep-equal, since all projection paths are computed in the same way in Part I. Similar with what we have explained in the proof of the correctness theorem of the conservative decomposition (Theorem 6.3.9), when checking if a vertex in G could be a v_x as defined above, it is safe to assume that all vertices on which this vertex depends are by-projection deep-equal to their corresponding vertices in G' (i.e., no v_x has been found among these vertices). Because, otherwise, we have already proven the assumption $\neg dEq(Q,Q')$.

6.5.6.1 Easy cases

Empty sequences, `Literal` values and variables are easy cases. `Literal`s are not projected, but rather copied literally. It is trivial to see that `Literal`s can always be replaced by a copy of them without altering the query result. A variable merely represents the value of the expression, to which the variable is bound. Thus, it is also trivial to see that if this expression is by-projection deep-equal to its corresponding expression in Q', the variable is also by-projection equal to its corresponding variable in Q'.

6.5.6.2 `LetExpr`, `IfExpr`, `Typeswitch`, `OrderExpr`, `Constructor` and `TransformExpr`

If v_x is a `LetExpr`, according to condition $(Cnd1)$, the following statement must be true:

$$\frac{dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_0}}(E_0,E'_0) \qquad dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}}(E_1,E'_1)}{\neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_{let}}}(E_{let},E'_{let}), \text{ where}} \quad (t^p_{let})$$
$$E_{let} = \texttt{let } \$x := E_0 \texttt{ return } E_1, \text{ and } E'_{let} = \texttt{let } \$x := E'_0 \texttt{ return } E'_1$$

A `LetExpr` expression merely returns its `return` clause E_1. Since it does not apply any XPath steps on E_1, it is clear that $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_{let}} = \vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}$. With the premise $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}}(E_1,E'_1)$, we thus have $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_{let}}}(E_{let},E'_{let})$.

Similar reasoning applies to the expressions `IfExpr`, `Typeswitch`, `OrderExpr`, `Constructor` and `TransformExpr` since they do not apply any XPath steps on their subexpressions or test node identities or structural properties of the nodes returned by their subexpressions. Hence, v_x can not be any one of these kinds of expressions.

6.5.6.3 `ExprSeq` and `ForExpr`

If v_x is a non-empty `ExprSeq`, according to condition $(Cnd1)$, the following statement must be true:

$$\frac{dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_0}}(E_0,E'_0) \qquad dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}}(E_1,E'_1)}{\neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_{seq}}}(E_{seq},E'_{seq}), \text{ where } E_{seq}=(E_0,E_1), \text{ and } E'_{seq}=(E'_0,E'_1)} \quad (t^p_{Seq})$$

A `ExprSeq` concatenates its two subexpressions into one expression. There are two cases.

If E_0 and E_1 contain nodes from the same XML documents, according to the by-projection insertion condition `(b)`, there is no `AxisStep`, `NodeCmp` or `NodeSetExpr` that depends on E_{seq}[16]. Hence, all three sets of relative projection paths $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_{seq}}$, $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_0}$ and $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}$ are

[16] If any of these kinds of expressions depend on E_{seq} in this case, we are not able to eliminate duplicates required by these expressions. Thus, the by-projection insertion condition `(b)` forbids decomposing *(i)* E_{seq}, *(ii)* all vertices on which E_{seq} depends, and *(iii)* all vertices depending on E_{seq} which could be reached from this `AxisStep`.

empty. With the third and fourth premises and Definition 6.1.4, we have: $dEq(E_0, E_0')$ and $dEq(E_1, E_1')$. With the first and second premises we have: $dEq(E_{seq}, E_{seq}')$. Since $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{seq}} = \emptyset$, with Definition 6.1.4 we have: $dEq(E_{seq}, E_{seq}') \Rightarrow dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{seq}}}(E_{seq}, E_{seq}')$.

If E_0 and E_1 only contain nodes from different XML documents, nodes in E_0 are disjunct with nodes in E_1. XPath steps on E_{seq} are allowed[17], thus the relative projection paths might not be empty (if they are empty, the reasoning is the same as that of the first case). Based on the static properties analysis rule SEQ (Section 6.2.3), it can be deduced that: $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{seq}} = \vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0} \cup \vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}$. The result of applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{seq}}$ on E_{seq} is equivalent to concatenating the results of applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0}$ on E_0 and applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}$ on E_1. The same holds for E_{seq}', E_1' and E_2'. With the third and fourth premises, we have $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{seq}}}(E_{seq}, E_{seq}')$.

If v_x is a `ForExpr`, according to condition $(Cnd1)$, the following statement must be true:

$$\frac{dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0}}(E_0, E_0') \qquad dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}}(E_1(E_0), E_1'(E_0'))}{\neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{for}}}(E_{for}, E_{for}'), \text{ where}} \quad (t^p_{for})$$
$$E_{for} = \texttt{for \$}x\texttt{ in }E_0\texttt{ return }E_1, \text{ and } E_{for}' = \texttt{for \$}x\texttt{ in }E_0'\texttt{ return }E_1'$$

The `ForExpr` expressions have similar behaviour to the `ExprSeq` expressions, i.e., they concatenate multiple subsequences into one sequence as their results. Using similar reasoning as above, we can deduce that the statement (t^p_{for}) is not true.

Hence, v_x can not be an `ExprSeq` or a `ForExpr`.

6.5.6.4 `CompExpr`

Assume v_x is a `CompExpr`, according to condition $(Cnd1)$, the following statement must be true:

$$\frac{dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0}}(E_0, E_0') \qquad dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}}(E_1, E_1')}{\neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{cmp}}}(E_{cmp}, E_{cmp}'), \text{ where } E_{cmp} = E_0 \odot E_1, \text{ and } E_{cmp}' = E_0' \odot E_1'} \quad (t^p_{cmp})$$

The symbol $\odot$ represents a value or a node comparison operator: =, !=, <, <=, >, >=, `is`, $\ll$ and $\gg$. A `CompExpr` does not apply any XPath steps on its subexpressions, and it returns a boolean value, on which no XPath steps can be applied, thus: $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0} = \emptyset$, $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1} = \emptyset$ and $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{cmp}} = \emptyset$.

If E_{cmp} is a value comparison expression, with $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0} = \emptyset$, $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1} = \emptyset$ and Definition 6.1.4 we have $dEq(E_0, E_0')$ and $dEq(E_1, E_1')$. It is then clear that $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{cmp}}}(E_{cmp}, E_{cmp}')$ holds.

If E_{cmp} is a node comparison expression, according to the by-projection insertion condition `(a)`, E_0 and E_1 only contain XML nodes from different XML documents[18]. If $\odot$ represents the `is` operator, both E_{cmp} and E_{cmp}' return `false`. If $\odot$ represents the $\ll$ or the $\gg$ operator, both E_{cmp} and E_{cmp}' return either `true` or `false`. Thus, E_{cmp} and E_{cmp}' are deep-equal, under the

[17] `NodeCmp` and `NodeSetExpr` are also allowed. The requirements that these expressions must eliminate duplicates and order nodes in their results can be treated as if they apply a `self` step on E_{seq}.

[18] If E_0 and E_1 contain XML nodes from the XML documents, the by-projection insertion condition `(a)` forbids these two expressions and all their subexpressions to be decomposed. This conflicts with condition $(Cnd2)$, which requires that E_{cmp} must depend on at least one remote sequence.

definition of deep-equal read-only queries with implementation freedom (Definition 6.1.6). Then, with $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{cmp}} = \emptyset$ and Definition 6.1.4, we have $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{cmp}}}(E_{cmp}, E'_{cmp})$.

Hence, v_x can not be a `CompExpr`.

6.5.6.5 `NodeSetExpr`

Assume v_x is a `NodeSetExpr`. According to condition $(Cnd1)$, the following statement must be true:

$$\frac{dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0}}(E_0, E'_0) \qquad dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}}(E_1, E'_1)}{\neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{nset}}}(E_{nset}, E'_{nset}), \text{ where } \; E_{nset} = E_0 \boxdot E_1, \; \text{ and } \; E'_{nset} = E'_0 \boxdot E'_1} \qquad (t^p_{nset})$$

The symbol $\boxdot$ represents a node set operator: `union`, `intersect` or `except`. Like the restriction on node comparison expressions, the by-projection insertion condition `(a)` enforces that E_0 and E_1 only contain XML nodes from different XML documents.

If $\boxdot$ represents an `intersect`, both E_{nset} and E'_{nset} return the empty sequence. It is trivial to see that $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{nset}}}(E_{nset}, E'_{nset})$ holds.

If $\boxdot$ represents a `union`, E_{nset} returns (E_0, E_1) or (E_1, E_0), and E'_{nset} returns (E'_0, E'_1) or (E'_1, E'_0). Thus, we have $dEq(E_{nset}, E'_{nset})$ (with implementation freedom). Based on the static properties analysis rule UNION (Section 6.2.10), it can be deduced that: $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{nset}} = \vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0} \cup \vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}$. The result of applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{nset}}$ on E_{nset} is equivalent to concatenating the results of applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0}$ on E_0 and applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}$ on E_1. The same holds for E'_{nset}, E'_1 and E'_2. Thus, we have $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{nset}}}(E_{nset}, E'_{nset})$.

If $\boxdot$ represents an `except`, E_{nset} and E'_{nset} return E_0 and E'_0, respectively. With the premise $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_0}}(E_0, E'_0)$, we have $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{nset}}}(E_{nset}, E'_{nset})$.

Hence, v_x can not be an `NodeSetExpr`.

6.5.6.6 `StepExpr`

Assume v_x is a `StepExpr` (shortened as: ST), according to $(Cnd1)$ above, the following statement must be true:

$$\frac{dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_1}}(E_1, E'_1)}{\neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{step}}}(E_{step}, E'_{step}), \text{ where } \; E_{step} = E_1/\text{ST}, \; \text{ and } \; E'_{step} = E'_1/\text{ST}} \qquad (t^p_{step})$$

Assume that the returned paths of E_1 are $\vec{\mathcal{P}}_{E_1}.\vec{\mathcal{R}} = \{p^{E_1}_1, \ldots, p^{E_1}_k\}$. Then the returned paths of E_{step} are [19] $\vec{\mathcal{P}}_{E_{step}}.\vec{\mathcal{R}} = \{p^{E_{step}}_1/\text{ST}, \ldots, p^{E_{step}}_k/\text{ST}\}$. Assume that expressions that depend on E_{step} apply a number of paths, with each path containing a number of XPath steps, on E_{step}, either as returned paths or as used paths. Then, the relative projection paths of E_{step} are:

$$\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{step}}.\vec{\mathcal{U}} = \{ \; \text{ST}^{u_1}_1/\text{ST}^{u_1}_2/\ldots/\text{ST}^{u_1}_u, \; \text{ST}^{u_2}_1/\text{ST}^{u_2}_2/\ldots/\text{ST}^{u_2}_v, \; \ldots, \; \text{ST}^{u_m}_1/\text{ST}^{u_m}_2/\ldots/\text{ST}^{u_m}_w \}$$

$$\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow E_{step}}.\vec{\mathcal{R}} = \{ \; \text{ST}^{r_1}_1/\text{ST}^{r_1}_2/\ldots/\text{ST}^{r_1}_x, \; \text{ST}^{r_2}_1/\text{ST}^{r_2}_2/\ldots/\text{ST}^{r_2}_y, \; \ldots, \; \text{ST}^{r_n}_1/\text{ST}^{r_n}_2/\ldots/\text{ST}^{r_n}_z \}$$

[19] See the rules STEP^a, STEP^{sc} and STEP$^{\mu}$ in Section 6.2.12.

The relative projection paths of E_1 are[20]:

$$\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}.\vec{\mathcal{U}} = \{\ \mathrm{ST}/\mathrm{ST}_1^{u_1}/\mathrm{ST}_2^{u_1}/\ldots/\mathrm{ST}_u^{u_1},\ \mathrm{ST}/\mathrm{ST}_1^{u_2}/\mathrm{ST}_2^{u_2}/\ldots/\mathrm{ST}_v^{u_2},\ \cdots,\ \mathrm{ST}/\mathrm{ST}_1^{u_m}/\mathrm{ST}_2^{u_m}/\ldots/\mathrm{ST}_w^{u_m}\}$$

$$\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}.\vec{\mathcal{R}} = \{\ \mathrm{ST}/\mathrm{ST}_1^{r_1}/\mathrm{ST}_2^{r_1}/\ldots/\mathrm{ST}_x^{r_1},\ \mathrm{ST}/\mathrm{ST}_1^{r_2}/\mathrm{ST}_2^{r_2}/\ldots/\mathrm{ST}_y^{r_2},\ \cdots,\ \mathrm{ST}/\mathrm{ST}_1^{r_n}/\mathrm{ST}_2^{r_n}/\ldots/\mathrm{ST}_z^{r_n}\}$$

The premise $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1}}(E_1, E_1')$ means that $\forall p_i \in \vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_1} : dEq(E_1/p_i, E_1'/p_i)$. Substitute E_1/ST with E_{step} and E_1'/ST with E'_{step}, we have $\forall p_j \in \vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow E_{step}} : dEq(E_{step}/p_j, E'_{step}/p_j)$. Hence, v_x can not be a `StepExpr`.

6.5.6.7 Built-in function calls

Under the by-projection semantics, the built-in functions `fn:root()`, `fn:id()`, `fn:idref()` and `fn:lang()` and their parameters can also be decomposed. A common property of these functions is that they need to access nodes outside the subtree(s) of their parameters, something not supported by the by-value and by-fragment semantics. In pass-by-projection, we have extended the projection path to allow the functions `root()`, `id()` and `idref()` to be part of a `SimplePath` (Table 5.6), while the function `fn:lang()` can be represented by `ancestor` and `attribute` steps[21].

For all `FunCalls` to built-in functions, we can use similar reasoning to that for `StepExpr` expressions to deduce that v_x can not be a `FunCall` to a built-in function.

In summary, we were not able to find a v_x among *all* vertices in a d-graph G (including the root vertex v_{root}) which is not deep-equal to its corresponding vertex v'_x in G', while all vertices on which v_x depends are deep-equal to their corresponding vertices in G'. This implies $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_{root}}}(v_{root}, v'_{root})$. It is trivial to see that $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_{root}} = \emptyset$, according to Definition 6.1.4, we have $dEq(v_{root}, v'_{root})$. Hence, the assumption $\neg dEq(Q, Q')$ of Part I does not hold.

Part II

In this part we drop the assumption used by Part I that the relative projection paths of *all* vertex $v_i \in G$ are $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_i}$. The relative projection paths for a parameter v_j of a remote expression v_i are actually $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_i\rightsquigarrow v_j}$. Since $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_i\rightsquigarrow v_j}$ is more selective than $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_i}$, we need to check if projecting v_j using $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_i\rightsquigarrow v_j}$ could cause the remote expression v_i to return an incorrect result.

Assume $\neg dEq(Q, Q')$, then there must exist one vertex $v_x \in G$, which is a parameter of a by-projection decomposition point $v_d \in G$, such that projecting v_x using the paths $\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_d\rightsquigarrow v_x}$ will cause the corresponding vertex v'_d of v_d (in the decomposed d-graph G') to be not by-projection equal to v_d. Formally, $\exists v_x, v_d \in G : v_d \overset{v}{\rightsquigarrow} v_x \wedge v_d \in I^p(G)$ and $\exists v'_x, v'_d \in G : v_d \overset{v}{\rightsquigarrow} v_x$, such that $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_d\rightsquigarrow v_x}}(v_x, v'_x) \Rightarrow \neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root}\rightsquigarrow v_d}}(v_d, v'_d)$ holds.

In XQuery, remote expressions can be seen as black boxes with one or more inputs[22] and one (possibly empty) output. Let v_p be a parameter of v_d. We examine all possible cases to

[20] As shown in Example 6.5.1, the relative projection paths of E_1 are longer than the relative projection paths of E_{step}, because E_1 is further away from v_{root} than E_{step}.

[21] See the rule (BLTIN^{lang}) in Section C.14.

[22] Remote expressions with zero input have already been handled in Part I.

see if v_p can be a v_x as defined above (if v_d has multiple parameters, the steps below can be repeated to check each parameter):

- v_p is not returned by v_d

  This is an easy case. Since v_p is only used by v_d to compute its result, the relative projection paths of v_p is equal to $\vec{\mathcal{P}}^{rel}_{v_d \rightsquigarrow v_p}$ (where $\vec{\mathcal{P}}^{rel}_{v_d \rightsquigarrow v_p}.\vec{\mathcal{R}} = \emptyset$). Thus $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d \rightsquigarrow v_x}}(v_x, v'_x)$ implies $dEq^{\vec{\mathcal{P}}^{rel}_{v_d \rightsquigarrow v_x}}(v_x, v'_x)$. The subgraph rooted at v_d can be seen as a separate query graph, with v_d as root vertex. In Part I, it has already been proven that $dEq^{\vec{\mathcal{P}}^{rel}_{v_d \rightsquigarrow v_p}}(v_p, v'_p) \Rightarrow dEq(v_d, v'_d)$ holds, if v_d is a root vertex. Thus, the assumption $\neg dEq(Q, Q')$ is not true.

- v_p is returned by v_d, but no XPath steps are applied on v_d

  In this case, the relative projection paths of v_p is also equal to $\vec{\mathcal{P}}^{rel}_{v_d \rightsquigarrow v_p}$. Using the same reasoning as that of the previous item, we again have $dEq^{\vec{\mathcal{P}}^{rel}_{v_d \rightsquigarrow v_p}}(v_p, v'_p) \Rightarrow dEq(v_d, v'_d)$. Thus, the assumption $\neg dEq(Q, Q')$ is not true.

- v_p is returned by v_d and a number of XPath steps are applied to v_d

  Although v_d could also return nodes originating from other expressions, those nodes are not relevant to our proof. Thus, it is safe to assume that v_d only returns nodes originating from v_p.

  Let $\vec{\mathcal{F}}_{v_p}$ be the projection of $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d \rightsquigarrow v_p}$ on v_p, i.e., $\vec{\mathcal{F}}_{v_p} = \mathfrak{P}(v_p, \vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d \rightsquigarrow v_p})$. The set $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d}$ contains all XPath steps that will by applied to v_d. In this case, if $\vec{\mathcal{F}}_{v_p}$ does not contain all nodes that should be returned, when applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d}$ on v_d, the assumption $\neg dEq(Q, Q')$ is proven.

  However, with the projection paths property (see Property 6.5.2) and the functions `allSuffixesVia()` and `allSuffixes()` with which $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d \rightsquigarrow v_p}$ and $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d}$ were computed, it is easy to see that each path in $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d}.\vec{\mathcal{U}}$ is a suffix of a path in $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d \rightsquigarrow v_p}.\vec{\mathcal{U}}$. The same holds for the returned paths. In other words, $\vec{\mathcal{F}}_{v_p}$ contains all nodes that are needed, when applying $\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d}$ to v_d. Thus, the assumption $\neg dEq(Q, Q')$ is not true.

In summary, we were not able to find a parameter v_p, which can satisfy the condition $dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d \rightsquigarrow v_x}}(v_x, v'_x) \Rightarrow \neg dEq^{\vec{\mathcal{P}}^{rel}_{v_{root} \rightsquigarrow v_d}}(v_d, v'_d)$. Hence, the assumption $\neg dEq(Q, Q')$ of Part II does not hold.

In conclusion, Part I and Part II together prove the correctness of the theorem. □

6.6 Correctness Proof of the XQUF Decomposition Algorithm

Theorem 6.6.1. XQUF Decomposition Correctness: *Let Q_u be a normal updating XCore query (i.e., without any XRPC expressions, but containing at least one updating expression which is not a subexpression of a `TransformExpr`) and G the corresponding d-graph. $I(G) \subset G$ is the (non-empty) set of decomposition points validated by one of the decomposition algorithms, and the vertices in $I(G)$ also satisfy the XQUF insertion conditions (Section 5.7.1).*

Let G' be the d-graph derived by doing an `XRPCExpr` *insertion above each vertex in* $I(G)$ *(Section 5.3.2), and* Q'_u *the corresponding query of* G'*. Then,* $dEq(Q_u, Q'_u)$ *holds under the definition of deep-equal updating queries with implementation freedom (Definition 6.1.9).*

Proof. Let Δ and Δ' be the PULs yielded by Q_u and Q'_u, respectively. By Definition 6.1.9, $dEq(Q_u, Q'_u)$ holds if we can prove that $dEq(\Delta, \Delta')$ holds according to the definition of deep-equal pending update lists (Definition 6.1.8). Definition 6.1.8 states that two PULs are deep-equal to each other if they both contain

(i) at least one `upd:delete` action on the same target nodes,

(ii) exactly one `upd:replaceNode`, `upd:replaceValue`, `upd:replaceElementContent` or `upd:rename` action on the same target nodes with deep-equal new contents, and

(iii) the same number of the same kind of insertion action on the same target nodes with deep-equal new contents.

Thus, there are three determining factors for two PULs to be deep-equal: *(1)* identifiers of the target nodes, *(2)* the new contents with which the target nodes will be updated, and *(3)* the number of the same update actions on the same target nodes.

The new contents for the target nodes and the number of times that a certain update action will be executed are computed by the read-only subexpressions that might be decomposed by the particular decomposition algorithm. In Theorem 6.3.9, 6.4.8 and 6.5.6, it has already been proven that a decomposed read-only subexpression produces deep-equal results to those of its corresponding subexpression in the original query. Thus, we only need to prove that Q_u and Q'_u update the same nodes, whose identifiers are computed by the `TargetExpr` subexpression of an `UpdExpr`.

Case 1: Q_u **contains only updates on local documents**

For this class of updating queries, the XQUF insertion conditions (Section 5.7.1) forces all `UpdExpr`s and their `TargetExpr` subexpressions in Q'_u to be executed on the local peer. This also prevents target nodes from being passed as function parameters or results, which would lose the original identities of the target nodes. Thus, it is easy to see that the target nodes of Q'_u are equivalent to those of Q_u, which proves $dEq(\Delta, \Delta')$. Hence, $dEq(Q_u, Q'_u)$ holds, in case Q_u only contains updates on local documents.

Case 2: Q_u **contains only updates on remote documents** In Section 5.7.2, we have extended the semantics of XQUF to allow updating expressions on remote documents. We specified that updates on a remote document are first applied on a local copy of the document (i.e., the remote document is retrieved to the local peer using `fn:doc()`), after which the updated local copy is sent to the peer hosting the remote document using `fn:put()` to overwrite the existing document.

When computing Q'_u, our XQUF rewrites allows those `UpdExpr`s (together with their `TargetExpr` subexpressions) which only contain updates on remote documents hosted by a single peer p_i to be decomposed and executed on p_i[23]. The constraint on a single peer is necessary to ensure that the target nodes are never passed as function parameters or results.

[23] As pointed out in Section 5.7.2, the updating functions could also be executed on other peers than the hosting peer, however, it implies the same semantics as executing those functions on the local peer, which makes remote execution not meaningful.

Let $\texttt{UpdExpr}_{loc}$ denote those UpdExprs in Q_u that will not be decomposed, and $\texttt{UpdExpr}_{h_i}$ denote the UpdExprs in Q_u that can be executed on peer h_i. Let Δ_{loc} and Δ_{h_i} denote the partial PULs that evaluating $\texttt{UpdExpr}_{loc}$ and $\texttt{UpdExpr}_{h_i}$ will yield, respectively. They correspond to Δ'_{loc} and Δ'_{h_i} in Q'_u, i.e., Δ'_{loc} is the result of evaluating $\texttt{UpdExpr}_{loc}$ on the local peer, while Δ'_{h_i} is the result of evaluating $\texttt{UpdExpr}_{h_i}$ on peer h_i. Hence,

$$\Delta = \Delta_{loc} \cup (\bigcup_{i=1}^{n} \Delta_{h_i}) \text{ and } \Delta' = \Delta'_{loc} \cup (\bigcup_{i=1}^{n} \Delta'_{h_i})$$

where n is the number of remote peers involved in the execution of Q'_u. Clearly, $dEq(\Delta_{loc}, \Delta'_{loc})$. For each Δ'_{h_i}, it is also easy to see that it identifies the same target nodes, since each Δ'_{h_i} is created by evaluating $\texttt{UpdExpr}_{h_i}$ on the peer hosting the documents. Together, we have $dEq(\Delta, \Delta')$, which implies that $dEq(Q_u, Q'_u)$ holds if Q_u only contains updates on remote documents.

Case 3: Q_u contains updates on both local and remote documents

For this case, $dEq(Q_u, Q'_u)$ is proven by combining Case 1 and 2. □

6.7 Correctness Proof of Distributed Code Motion

In Section 5.4.4, we have described that, in certain cases, subexpressions of a valid decomposition point r_s could be moved out of the subgraph rooted at r_s and be replaced by a new parameter presenting this subexpression. In this section, we prove that applying this so-called *distributed code motion* technique to the resulting decomposed d-graph of a certain decomposition algorithm will not cause a valid decomposition point r_s to produce non-deep-equal result.

Theorem 6.7.1. Distributed Code Motion Correctness: *Let G be a d-graph and G_{r_s} is the subgraph of G rooted at the vertex r_s, where r_s is a decomposition point validated by one of the decomposition algorithms. Let $\{p_1, ..., p_m\} \in V(G)$ be the set of vertices that are parameters of r_s. Let v_i be a vertex in the subgraph rooted at r_s which satisfies all of the following constraints:*

- *v_i is also a decomposition point validated by the same decomposition algorithm, i.e., $v_i \in V(G_{r_s}) \wedge v_i \in I(G)$;*
- *there is exactly one $p_x \in \{p_1, ..., p_m\}$ such that $v_i \overset{p}{\rightsquigarrow} p_x$;*
- *there is exactly one path $(v_i, v_{i+1}, ..., v_{i+n}, p_x)$ that can reach p_x starting from v_i.*

Then, moving v_i out of G_{r_s} by introducing a new parameter p_n to represent v_i, and replacing v_i with p_n will not cause remote executing of r_s to produce non-deep-equal result to the local execution of r_s.

Proof. Since r_s and v_i are both valid decomposition points, they could be executed on possibly different remote peers. With Theorem 6.3.9, Theorem 6.4.8 and Theorem 6.5.6, we have that, regardless of on which peer r_s is executed, it produces deep-equal results, possibly using the result of a remote execution of its subexpression v_i. Moving v_i out of G_{r_s} and passing its value as a parameter to r_s implies that v_i will be executed on a different peer than the peer, on which r_s will be executed. Therefore, this code motion will cause remote execution of r_s to produce a deep-equal result. □

7

StreetTiVo: Manage Multimedia Data Using A P2P XDBMS

StreetTiVo is a project that aims at bringing research results into the living room; in particular, a mix of current results in the areas of Peer-to-Peer XML Database Management System (P2P XDBMS), advanced multimedia analysis techniques, and advanced information retrieval techniques. The project develops a plug-in application for so-called Home Theatre PCs, such as set-top boxes with MythTV or Windows Media Center Edition installed, that can be considered as programmable digital video recorders. StreetTiVo distributes compute-intensive multimedia analysis tasks over multiple peers (i.e., StreetTiVo users) that have recorded the same TV program, such that a user can search in the content of a recorded TV program *shortly* after its broadcasting; i.e., it enables *near real-time* availability of the meta-data (e.g., speech recognition) required for searching the recorded content. StreetTiVo relies on our P2P XDBMS technology, which in turn is based on a DHT overlay network, for distributed collaborator discovery, work coordination and meta-data exchange in a volatile WAN environment. The technologies of video analysis and information retrieval are seamlessly integrated into the system as XQuery functions.

7.1 Motivation

Things are changing in the living room, under the TV set: TV is going digital and consumer electronics gets networked. The so-called "set-top" boxes that are needed for digital television have appeared in the houses of many; and, many of these set-top boxes are rather powerful computers connected to the Internet, running Windows Media Center or its open-source MythTV equivalent.

A related trend is the increasing demand for multimedia information access. Presently, "ordinary" people own hundreds of gigabytes of multimedia data, resulting from their digital photo cameras, hard-disk video recorders, etc. However, searching multimedia files is usually restricted to simple look up in the meta-data of files, such as file names and (human-edited) descriptions. More advanced multimedia retrieval requires highly compute-intensive pre-processing of the data (e.g., speech recognition and image processing); as a rough estimation, it takes a moderate computer more than one order of magnitude more time to derive the auxiliary data that would enable better search facilities.

The idea of the *StreetTiVo* project is to unite the computing power of those Media Center devices (peers) in the living rooms. By distributed and parallel execution of compute-intensive

multimedia analysis tasks on multiple peers, near real-time indexing of the content can be provided using just the ordinary hardware available in the network. StreetTiVo divides the Media Center devices into groups and assigns each group a short time slice of a recording (e.g., ten seconds), to run multimedia analysis tools on those time slices only. Thus, the peers form virtual digital streets and are virtual neighbours of each other. StreetTiVo uses the P2P concept in a strictly legal way, as it is not used to distribute the video files themselves. Users can only watch the content they have recorded themselves. What is exchanged by StreetTiVo are only the results of multimedia analysis of those videos, i.e., generated meta-data.

Summarising, the goal of the StreetTiVo project is: *unite Media Center devices using P2P technologies to cooperatively run compute intensive multimedia analysis applications just in everybody's living room so that they could produce results in near real-time*.

Imagine for example, if only a tiny fraction of the millions of people recording the Champions League soccer competition would participate in StreetTiVo! Useful media analysis tools would include the transcription of the text spoken by the presenters during the match, but also the cross-media analysis to recognise goals and other exciting moments (e.g., from audio volume and/or camera motion patterns). After each group of media centers has exchanged its partial analysis results with the other groups (that recorded the same match), all StreetTiVo participants obtain the complete set of automatically derived annotations, such that meta-data could be used for direct entry to the most exciting moment(s) of the game, or, the automatic generation of a summary including all highlights.

Project Embedding StreetTiVo is a demo application of the Dutch national research project MultimediaN[1] that unites multimedia and database researchers in various academic and industrial research institutes to achieve high-quality multimedia solutions for the digital world of today and tomorrow. In this project, participating parties work together on new and existing multimedia applications, especially applications that involve audio and video analysis, for instance, finding back objects and people by shape. One of the goals of StreetTiVo is thus to ease the early adoption of research results in the practice, by using the existing hardware in everybody's living room.

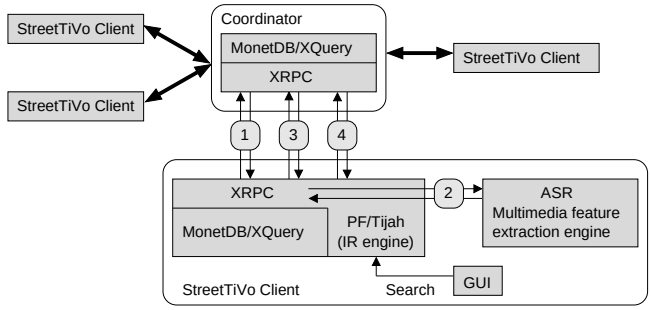
7.2 P2P Data Management

From a network communication perspective, DHT-based overlays have gained much popularity in both research projects and real-world P2P applications [2, 153, 143, 162, 101, 147, 142, 43, 148, 100, 108, 185]. DHT networks have proven to be efficient and scalable (guarantees $O(logN)$ scalability) in volatile WAN environments [147, 146]. From a data management perspective, XML has become the de facto standard for data exchange over the Internet, and XQuery the W3C standard for querying XML data. The infrastructure of StreetTiVo is therefore provided by *MonetDB/XQuery** [179], a P2P XML DBMS that supports distributed evaluation of XQuery[38] queries over DHT networks [2, 143, 162, 147].

Each peer runs an XDBMS, *MonetDB/XQuery* [41], to manage its local data. Communication among peers is done by remote execution of XQuery functions using *XRPC* [180, 181], a simple XQuery extension for Remote Procedure Calls (RPC) that enables *efficient* distributed querying of *heterogeneous* XQuery data sources.

The current implementation of StreetTiVo as XQuery expressions applies a Dutch Automatic Speech Recognition system (*ASR*) [102] to the recorded programmes, and provides

[1] See `www.multimedian.nl`



1. register−recording($channel, $startDT, $duration, $clntAddr): fragment
2. start−asr($progID, $fragID, $start, $end)
3. add−finished−fragment($progID, $fragID, $text, $clntAddr): fragment
4. get−speech−text($progID, $fragIDs)

Figure 7.1: StreetTiVo architecture: client-server model

a full-text retrieval service of the ASR output by employing PF/Tijah, a MonetDB/XQuery extension[97]. When a TV program is broadcast, all participants (peers that are recording this TV program) jointly extract the (Dutch) spoken text using the ASR component, and exchange local partial ASR results with each other. ASR produces XML documents containing speech texts and some meta-data, for example, start/end timestamp of a sentence in the video file. Each participant stores *all* resulting XML documents of the recording in its local MonetDB/XQuery that can then be queried using PF/Tijah. The meta-data of the retrieved sentences provide sufficient information for the StreetTiVo GUI to display only the desired video fragment.

7.3 StreetTiVo Architecture

The current version of StreetTiVo uses a simple client-server network model, as shown in Figure 7.1. In this setup, we assume that the coordinator is a reliable host, while the clients join and leave unpredictably (similar to the early work of [65]). While our next step will be to replace this model with a more sophisticated DHT-based P2P model, we first detail the current implementation.

Each peer runs a MonetDB/XQuery server and communicates with other peers via XRPC, by sending SOAP XRPC request/response messages. The central StreetTiVo coordinator is responsible for registration of recordings, and the generation and distribution of ASR tasks. For each `recording`, the coordinator maintains a list of participating peers and a list of tasks (called `fragments`). A recording is divided into short fragments (usually several seconds) to be analysed parallelly by the participants. For each `fragment`, the coordinator maintains if it is being processed by a peer (i.e., it has an `assignee`), or if its speech text is already available (i.e., it has an `owner`). All meta-data are in XML format and stored in MonetDB/XQuery.

Example 7.3.1. *The XML snippet in Table 7.1 shows the* `recording` *element for the TV program* `bbc1_20080425_200000`*. The progID is determined by channel+date+start time. The TV program is recorded by two peers and is divided into 4 fragments. The attributes* `start` *and* `end` *of a* `fragment` *indicate the relative start/end timestamps of the fragment in the video file. Fragments are assigned to the participants in the order they register, so initially fragments 1 and 2 are assigned to the hosts* `x.example.org` *and* `y.example.org`*, respectively. So far,*

```
⟨recording progID="bbc1_20080425_200000"⟩
  ⟨participants⟩
    ⟨participant host="x.example.org"/⟩
    ⟨participant host="y.example.org"/⟩
  ⟨/participants⟩
  ⟨fragments⟩
    ⟨fragment fragID="1" start="0"  end="10"⟩⟨owner host="x.example.org"/⟩⟨/fragment⟩
    ⟨fragment fragID="2" start="10" end="25"⟩⟨assignee host="y.example.org"/⟩⟨/fragment⟩
    ⟨fragment fragID="3" start="25" end="30"⟩⟨assignee host="x.example.org"/⟩⟨/fragment⟩
    ⟨fragment fragID="4" start="30" end="38"/⟩
  ⟨/fragments⟩
⟨/recording⟩
```

Table 7.1: Information maintained for each recording

`x.example.org` *has finished analyse* `fragment` *1 (indicated as the* `owner` *of the fragment) and has been assigned a new job* `fragment` *3. The host* `y.example.org` *is still processing* `fragment` *2. Since there are no more participants,* `fragment` *4 is waiting to be assigned.*

All interfaces between coordinator, clients and the local ASR engine have been defined as XQuery functions. A small Java program implements the XQuery function (`start-asr()` in Figure 7.1) that triggers the ASR engine. The interaction between one StreetTiVo client and the coordinator is shown in details. Collaborative speech recognition works as follows.

Step ① When a TV program is scheduled for recording, the StreetTiVo client sends an XRPC request to the coordinator to execute the function `register-recording()`. The coordinator responds with a `fragment`, which has not been processed by any participants, and inserts the client as an `assignee` into the `fragment` element. For reliability reasons, each `fragment` is assigned to multiple clients.

If the request is the first registration for a TV program, the coordinator first needs to generate ASR tasks. An easy way to do this is to divide the whole recording into equal sized fragments. To get high quality ASR result, the ASR segmenter should be used, which is able to filter out the audio that do not contain speech and generates fragments accordingly (see Section 7.3). Information provided by the ASR segmenter ensures that the ASR speech recogniser produces more accurate results. However, the better quality comes at a high cost of speed, because the coordinator must record the TV program itself and run ASR segmenter *afterwards*. So, there is a trade-off between speed and quality.

Step ② Upon receipt of the response from the coordinator (in *Step* ①), the StreetTiVo client starts its local ASR engine to analyse the fragment specified in the response message, by calling the interface function `start-asr()`.

Step ③ After the ASR engine has finished analysing a fragment, the StreetTiVo client reports this by calling `add-finished-fragment()` on the coordinator and passing among others the retrieved text as parameter.

Step ④ After having finished one task, the StreetTiVo client is expected to request a new task (`get-job()`), until the coordinator responds with an empty task, which might mean that there are sufficient number of assignees for each fragment, or that the coordinator has received the ASR results of all fragments.

Step ⑤ If there are no new tasks, the StreetTiVo client waits for some predefined time to give the other participants the opportunity to finish their ASR tasks, and then attempts to retrieve the speech text of the missing fragments. The StreetTiVo client can directly ask the coordinator

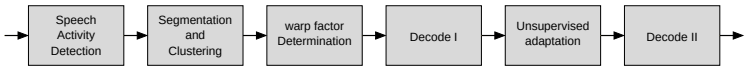


Figure 7.2: Overview of the ASR decoding system.

for the missing fragments (`get-speech-text()`), since all ASR results are also stored at the coordinator, but the preferred procedure is to just call the coordinator's `get-fragments()` function (not shown) to find out what participant owns which ASR result, and retrieve the fragments' texts from those nodes.

Once the ASR results are locally available, StreetTiVo users can search for video fragments by entering keywords in the GUI. The keywords are subsequently translated into Tijah queries. PF/Tijah returns matching sentences ranked by their estimated probability of relevance to the query. Each sentence is tagged with its relative start/end timestamps in the video file, this way, the desired video fragments can be retrieved. In summary, StreetTiVo has three major components: XRPC takes care of communication among peers, ASR provides video analysis functions, and PF/Tijah enables retrieval of video fragments using keywords. In the remainder of this section, we briefly give a overview of ASR and PF/Tijah.

ASR The Automatic Speech Recognition (ASR) supports the conceptual querying of video content and the synchronisation to any kind of textual resource that is accessible, including other annotations for audiovisual material such as subtitles. The potential of ASR-based indexing has been demonstrated most successfully in the broadcast news domain. Typically large vocabulary speaker independent continuous speech recognition (LVCSR) is deployed to this end.

The ASR system deployed in StreetTiVo was developed at the University of Twente and is part of the open-source SHoUT speech recognition toolkit[2]. Figure 7.2 gives an overview of the ASR decoding system. Each step provides the input for the following step. The whole process can be roughly divided into two stages. During the first stage, the *Speech Activity Detection* (SAD) is used to filter out the audio parts that do not contain speech. This step is crucial for the performance of the ASR system to avoid that it tries to recognise non-speech audio that is typically found in recorded TV programs such as music, sound effects or background noise with high volume (traffic, cheering audience, etc). After SAD, the system tries to figure out 'who spoke when', a procedure that is typically referred to as speaker diarisation. In this step, the speech fragments are split into segments that only contain speech from one single speaker. Each segment is labelled with its corresponding speaker ID. Next, for each segment the vocal tract length (VTLN) warping factor is determined for vocal tract length normalisation. Variation of vocal tract length between speakers makes it harder to train robust acoustic models. In the *SHoUT* system, normalisation of the feature vectors is obtained by shifting the Mel-scale windows by a certain warping factor during feature extraction for the first decoding step.

After having cleaned up the input sound and gained sufficient meta information, speech recognition can be started in the second stage. Decoding is done using the HMM-based Viterbi decoder. In the first decoding iteration, triphone VTLN acoustic models and trigram language models are used. For each speaker, a first best hypothesis aligned on a phone basis is created for unsupervised acoustic model adaptation. Optionally, for each file a topic specific language

[2]For information on the use of the SHoUT speech recognition toolkit see `http://wwwhome.cs.utwente.nl/~huijbreg/shout/index.html`

model can be generated based on the input of first recognition pass. The second decoding iteration uses the speaker adapted acoustic models and the topic specific language models to create the final first best hypothesis aligned on word basis. Also, for each segment, a word lattices is created. A more detailed description of each step can be found in [102].

PF/Tijah: XML Text Search PF/Tijah [97] is another research project run by the University of Twente with the goal to create a flexible environment for setting up search systems by integrating MonetDB/XQuery that uses the Pathfinder compiler [88] with the Tijah XML Information Retrieval (IR) system [121]. PF/Tijah includes out-of-the-box solutions for common tasks like index creation, stemming, result ranking (using several retrieval models), and relevance feedback, but it remains at the same time open to any adaptation or extension. The system aims to be *(i)* a general purpose tool for developing IR end user applications using XQuery statements with text search extensions, and *(ii)* a playground for the information retrieval scientist and advanced user to easily set up and test new search systems. The main features supported by PF/Tijah include the following:

- Retrieving arbitrary parts of textual data, unlike traditional IR systems for which the notion of a document needs to be defined up front by the application developers. For example, if the data consists of scientific journals one can query for complete journals, journal issues, single articles, sections from articles or paragraphs *without* adapting the index or any other part of the system configuration;
- Complex scoring and ranking of the retrieved results by means of so-called *Narrowed Extended XPath* (NEXI) [133] queries. NEXI is a query language similar to XPath that only supports the `descendant` and the `self` axis step, but that is extended with a special `about()` function that takes a sequence of nodes and ranks those by their estimated probability of relevance to the query;
- Ad hoc result presentation by means of its query language. For instance, when searching for a special issue of a journal, it is easy to print any information from that retrieved result on the screen in a declarative way (i.e., not by means of a general purpose programming language), such as its title, date, and the preface. This is simply done by means of XQuery element construction;
- PF/Tijah supports incremental indexing: when new ASR fragments are added to the database, their text will be automatically indexed by PF/Tijah, *without* the need to re-index the entire database from scratch;
- Search combined with traditional database querying, including for instance joins on values. As an example, one could search for employees from the financial department, who have also worked for the sales department and have sent an email about “tax refunds”.

StreetTiVo inserts fragments containing the transcripts of ASR whenever they are available. Therefore, they will not be nicely grouped per programme in the database, nor will they be in chronological order. The combination XQuery and NEXI text search enables StreetTiVo to search matching fragments, combine the fragments with the same programme identifier, combine their scores (or take the score of the best matching fragment), rerank programmes by the scores of their fragments, and display the matching programmes along with their best matching fragments: all of this is done in one query.

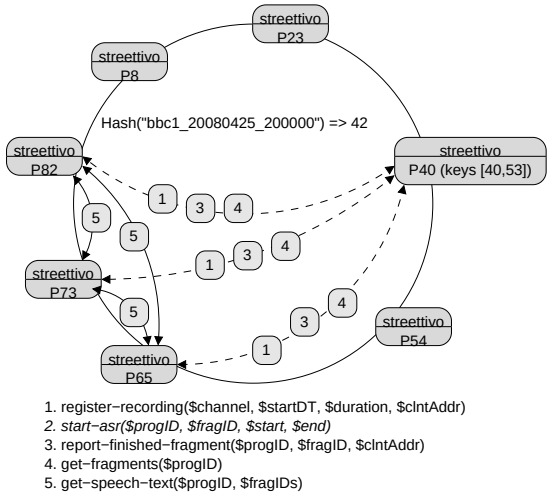


Figure 7.3: StreetTiVo architecture: DHT model

7.4 Next Steps

Our next step in the development of StreetTiVo is to replace the client-server model with MonetDB/XQuery⋆ [179], which integrate the P2P data structure Distributed Hash Tables (DHTs) into XQuery (see Figure 7.3). A DHT [2, 143, 162, 147] provides *(i)* robust connectivity (i.e., it tries to prevent network partitioning), *(ii)* high data availability (i.e., prevent data loss if a peer goes down by automatic replication), and *(iii)* a scalable $(key, value)$ storage mechanism with $O(log(N))$ cost complexity, where N is the number of peers in the network. A number of P2P database prototypes have already used DHTs [101, 142, 43, 100, 108].

In a StreetTiVo system using a DHT model, peers are managed by a DHT ring. There is no single StreetTiVo coordinator. All peers are unreliable and each can be both a coordinator and a client, thus, each peer must additionally support the functions provided by the coordinator, as discussed in Section 7.3. The process to collectively speech extraction using ASR is similar as in the client-server model, except several small changes:

Example 7.4.1. *Figure 7.3 shows an example scenario, in which three peers* `N65`*,* `N73` *and* `N82` *will record the TV program with progID=*"`bbc1_20080425_200000`"*. All participants use the same hash function to calculate the hash of the progID, which is 42 here. Since the peer* `P40` *is responsible for keys* $[40,53]$*, it is chosen as the coordinator for this TV program. The participating peers register the recording at* `P40` *(Step ①). Generating ASR tasks is still done by the (temporary) coordinator,* `P40`*. When a peer finished an ASR task, it reports this at the coordinator (Step ③), but* without *sending the extracted text. All participants will repeat steps ④, ② and ③ to process more fragments, until there are no more ASR tasks. Finally, the participants exchange the ASR results by first finding the owner of each fragments from the coordinator (Step ⑤), and then retrieving the missing ASR results from each other (Step ⑤).*

In the DHT model, the availability of the recording meta-data (i.e., the `recording` elements maintained by a coordinator as discussed in Section 7.3) is guaranteed thanks to the automatic replication facility provided by the underlying DHT network, that is, all data on a peer managed by the DHT network are replicated on the peer's predecessors and successors.

Thus, if peer P40 would fail in our example, all request with key 42 would be routed by the DHT network to P23 or P54. Also note that, the ASR results are not managed by the DHT network. Basically, all StreetTiVo peers can retrieve these data, but only the peers that have recorded the particular TV program can display the video. The availability of the ASR results is affected by the number of participants (more participants ⇒ higher availability). However, this is not a crucial issue, since every StreetTiVo peer is able to run ASR on a missing fragments.

Integrating XQuery and DHT The challenges in integration of XQuery and DHT are:

(i) How a DHT should be exploited by an XQuery processor?

(ii) If and how the DHT functionality should surface in the query language?

As described in Section 4.6, in MonetDB/XQuery*, we propose to avoid any additional language extensions, but rather introduce a new `dht://` network protocol, accepted in the destination URI of `fn:doc()`, `fn:put()` and `execute at`. The generic form of such URIs is `dht://`*dht_id*`/`*key*, where `dht://` is the network protocol, *dht_id* is the ID of the DHT network to be used. Such an ID is useful to allow a P2P XDBMS to participate in multiple (logical) DHTs simultaneously (see Figure 4.6(b)). The *key* is used to store and retrieve values in the DHT.

In the architecture that was shown in Figure 4.6(b), we run the DHT as a separate process called the Local DHT Agent (LDA). Each LDA is is connected to one DHT *dht_id*. In this *tight coupling* (Section 4.6.3) between the DHT network and the XDBMS [179], each DHT peer uses its local XDBMS to store the data (i.e., XML documents) and the local XDBMS uses its underlying DHT network to route XQuery queries to remote peers for execution (i.e., pass XRPC requests to the LDA). A positive side-effect of this tight coupling is that the DBMS gets access to the information internal to the P2P network. This information (e.g. peer resources, connectivity) can be exploited in query optimisation. To realise this coupling, we need to extend the DHT API (`put()` and `get()`) with one new method: `xrpc(`$key, q, m, f($*ParamList*$)$`):item()*`, where $f($*ParamList*$)$ is the XQuery function that is to be executed on a remote DHT peer determined by *key*. The parameters q and m specify XQuery module, in which the function f_r is defined and the location of the module file. With this method, an XRPC call on a peer p_0 to a `dht://`dht_x/key_y URI is handled as follows:

1. The XRPC `request`$(q,m,f,$*ParamList*$)$ is passed to the Local DHT Agent lda_0^x of p_0, which in turn passes the `request` to the DHT network dht_x.

2. The DHT dht_x routes the `request` using the normal DHT routing mechanism to the peer p_i responsible for key_y.

3. When the LDA lda_i^x on p_i receives such a request, it performs an XRPC call containing the same request to the MonetDB/XQuery instance on p_i.

4. When lda_i^x receives the response message, it transports the response back via dht_x to the query originator p_0.

Use Cases Below we show how two main StreetTiVo functions can be implemented as XQuery module functions, which then can be executed using XRPC and the tightly coupled DHT semantics.

(i) Collaborator Discovery. In StreetTiVo, every TV program has a unique identifier *progID*, and for each recorded TV program a `recording` element is maintained by the peer responsible for the key *hash*(*progID*) with lists of `participants` and `fragments`. If a peer is going to record the TV program "`bbc1_20080425_200000`", it should register the recording at the coordinator of this TV program. This can be done by the following XRPC call:

```
import module namespace stv = "streettivo" at "http://example.org/stv.xq";
let $key := hash("bbc1_20080425_200000"),
    $dst := fn:concat("dht://dht_x/", $key)
return execute at {$dst}
       {stv:register-recording(bbc1, "2008-04-25T20:00:00", "1H", "x.example.org")}
```

(ii) Distributed Keyword Retrieval. Assume a StreetTiVo user wants to search in today's newscast "`bbc1_20080425_20-0000`", he/she has recorded, for video fragments that were about the situation in Tibet, but the ASR results are not completely available (yet) on his/her local machine. Then the search request might be sent to other StreetTiVo peers that have recorded the same newscast. The following pseudo-code first retrieves the list of `fragments` from the coordinator, and then sends a `search` request to each peer that owns (`$frags//owner/@host`) the ASR results of a fragment:

```
import module namespace stv = "streettivo" at "http://example.org/stv.xq";
let $key := hash("bbc1_20080425_200000"),
    $dst := fn:concat("dht://dht_x/", $key)
    $frags := execute at {$dst} {stv:get-fragments(bbc1, "2008-04-25T20:00:00")
return for $p in $frags//owner/@host
       return execute at {$p} {stv:search("situation in Tibet")}
```

7.5 Conclusion

In this chapter, we have described StreetTiVo, a P2P Information Retrieval system that enables near real-time search in video contents by just using existing hardware in the living rooms to collectively run compute-intensive video analysis video content analysis tools.

Thanks to its implementation in a high-level declarative database language, it is straightforward to extend StreetTiVo with other types of functionality. We plan to complement the current media analysis with image processing techniques to automatically detect celebrities in news broadcasts or goals in soccer matches. Maybe even more interesting is that StreetTiVo users can also easily share their own human made annotations. For example, people usually schedule a recording several minutes before/after the start/end of the to be recorded TV program, to prevent missing part of the program. If just one StreetTiVo user has annotated the exact start/end timestamp of the TV program, the information can be shared in the platform with other StreetTiVo users who have recorded the same program, and the unnecessary parts of their recordings can be removed transparently.

8 Conclusion and Outlook

P2P content sharing applications have gained enormous popularity in less than a decade. But nowadays, the demand of more sophisticated P2P applications that go beyond simple keywords search is growing. However, developing P2P applications that provide non-trivial distributed data management facilities is a rather cumbersome task, since applications have to deal with information from huge collections of highly heterogeneous and volatile data sources. A P2P data management system which acts as a database middle-ware and offers a uniform database abstraction on top of a dynamic set of distributed data sources, should ease the development of data-intensive P2P applications. In this PhD work, we research which features such a database abstraction should offer and how it can be realised efficiently by extending and combining existing XDBMS with P2P technologies.

8.1 Research Summary

The main research question we try to answer in this thesis is:

> *How to support efficient processing of full-fledged XQuery queries – including those containing XQUF expressions – on large amounts of XML data served by heterogeneous XQuery engines in P2P settings?*

The main research question is divided into five more specific questions, as stated in Chapter 1. Below we summarise the contributions of this thesis as the answers to those specific research questions.

1. **How should we extend XQuery with a query shipping mechanism that is suitable for the targeted environments?**

   Before extending XQuery, we have first carefully analysed the characteristics of our target environments and defined a list of criteria which such extensions must satisfy (Section 3.1). The result is XRPC, a simple but powerful extension of XQuery that adds the RPC paradigm to XQuery. At the syntax level, the "`execute at`" statement of XRPC is inspired by that of XQueryD [144, 29]. However, XRPC has several properties that makes it particularly suitable for large scale P2P settings.

   XRPC is simple because *(i)* it adds RPC in the least invasive way to XQuery: adding a destination URI to an XQuery function application; *(ii)* it respects well-accepted (de factor) Web standards, e.g., RPC, SOAP, HTTP and Java; *(iii)* the SOAP XRPC protocol is a stateless protocol; and *(iv)* it is easy for other XQuery engines to adopt XRPC (we

will come back to this point in the next research question). As an orthogonal extension, XRPC preserves the expressiveness of XQuery, and the SOAP XRPC protocol supports all XDM data types, including user-defined XML Schema types with the ability to validate the SOAP messages.

XRPC is flexible because *(i)* an XRPC call can be made anywhere in a query where an XQuery function call is allowed; *(ii)* modules only need to be imported once and all the imported functions can be executed both locally and remotely on an arbitrary number of peers, avoiding any additionally effort to compile a module.

XRPC is efficient because *(i)* the Bulk RPC technique not only greatly reduces network latency, but also exploits the set-at-a-time infrastructure of DBMSs, e.g., turning bulk selections into a join between the SOAP message and the XML documents; *(ii)* with the by-projection decomposition algorithm, almost every XQuery expression can be decomposed, giving a large number of possibilities for distributed query optimisation; and *(iii)* the runtime projection algorithm can be much more accurate than the compile time techniques, which minimises the sizes of the SOAP messages. The efficiency of XRPC also implies that it is scalable with respect to the number of peers in the network (achieving minimal number of network round-trips thanks to the stateless protocol), the number of documents and the sizes of the documents.

However, XRPC is not *only* a language extension. The SOAP XRPC network protocol makes XRPC the only distributed XQuery proposal that is interoperable and full-fledged, at the time of this writing[1]. Within the scope of the XRPC project, we have also studied topics such as distributed transaction management and query optimisation. These topics are addressed by the remaining four research questions.

2. How can different XQuery engines be united to jointly evaluate a single query?

This problem calls for an interoperable and easy to support solution. Concerning interoperability, the basic building blocks of XRPC, i.e., XML, XQuery, SOAP and HTTP, are all well-defined and generally accepted Web standards. To enable communication among different XQuery engines, we propose a well-defined SOAP-based protocol, the SOAP XRPC protocol, in Section 3.3. Thus, network communication in XRPC uses XML messages over HTTP. Such a protocol is easy for existing XQuery engines to support, since they are already perfectly equipped to process XML messages, and there is ubiquitous support for URIs and HTTP.

The design of XRPC has been kept as simple as possible, which makes it easy to understand and support. RPC is an obvious and popular paradigm for implementing the client-server model of distributed computing[2]. Hence, to support XRPC, an XQuery engine merely needs to extend its grammar rules to support the "`execute at`" statement and implement the stub code and request handler. Serialisation and parsing of the SOAP messages are functionalities that already exist in every XQuery engine.

[1] XButler [134] is the only distributed XQuery proposal that *also* adopts an open communication protocol, i.e., SOAP RPC. However, due to the limitations in the supported data types by SOAP RPC (as discussed in Chapter 1), XButler is restricted to support functions, which only have atomic values as their parameters and results, like the WSDL services.

[2] The idea of RPC was first described in RFC707 [145] in 1976.

An XQuery engine can even participate in the evaluation of a distributed XQuery query *without* having XRPC integrated. The XRPC Wrapper, described in Section 4.2, is a SOAP service handler, implemented in Java and XQuery for reasons of interoperability. It can be run on top of *any* XQuery systems that is XQuery 1.0 compliant. The XRPC Wrapper *translates* a SOAP XRPC request message into a standard XQuery query, which is passed to the underlying XQuery engine for execution. Query results are wrapped in an XRPC response envelope and sent back to the caller. Extracting information from a request message (i.e., information of the called function and its parameter values) is done using XPath steps and generating the XRPC response message is done using element construction, again, features that are available in every XQuery engine.

3. How are distributed updating queries supported?

Since the XRPC extension is orthogonal to *all* XQuery features, it automatically supports remote execution of XQUF expressions by means of calling updating functions on remote peers. If updates are allowed in a system, dealing with transactional semantics is a matter of course. In this Ph.D. work, we have spent some effort on formally specifying the semantics of distributed updating XRPC queries and the support for atomically committing of distributed transactions. In Section 3.4, we have formally defined the semantics for both read-only and updating XRPC queries under different levels of isolation, and the necessary extensions to the SOAP XRPC message format. To support atomic committing of distributed transactions, we have chosen not to extend the SOAP XRPC protocol with any 2PC-like features, but rather to rely on the recent industry standard Web Services Atomic Transaction [55, 54], which provides a SOAP-based 2PC interface. Section 4.5 describes how updating queries are handled utilising this standard. By using the XRPC Wrapper, we have demonstrated processing of distributed updating XRPC queries involving Galax, MonetDB/XQuery, Saxon and X-Hive [181].

If multiple groups of XML nodes are inserted by multiple uses of the same kind of `insert` expression[3] with the same target node, XQUF leaves the ordering among these groups to be *implementation-dependent*, which implies an undeterministic ordering. However, node order plays an important role in XML and XQuery. We thus regard it worthwhile to define a deterministic update order that simply respect the order in which updates appear in `for`-loops and sequence constructions. In XRPC, a deterministic distributed updates order can be ensured by a simple extension to the SOAP XRPC protocol, as described in Section 4.4.

Finally, in Section 5.7.2 we extend the semantics of XQUF to allow updates on remote documents which are identified by an `xrpc://` URI scheme. In this section, we also propose rewriting rules to ensure the correctness of decomposed queries that contain updates on remote peers, that is, updating expressions may only be carried out on the peer owning the document to be updated.

4. How can we automatically decompose XQuery queries for distributed execution?

Decomposing queries to address multiple data sources is a well-known optimisation mechanism in distributed query processing. While many of the existing techniques can be carried over in distributed XQuery processing, there are challenges, introduced by the XML data model and the XQuery language, which do not exist in a value-based relational data

[3] This includes `insert into`, or `insert as first|last into`, or `insert before|after`.

model. That is, XML nodes have node identities and structural properties. In Chapter 5, we studied in detail automatic decomposition of standard XQuery queries with semantic guarantees. We have presented three decomposition algorithms for both read-only and updating queries. All three algorithms use a copy-based protocol (i.e., the SOAP XRPC protocol and its extensions) for (un)marshalling function parameters and results. In Chapter 1, we have argued our choice for a stateless protocol, which we believe is more suitable for our target large scale P2P environments than a stateful protocol. In Chapter 5, we show, especially with the by-projection algorithm, that such a simple stateless protocol is extremely efficient and gives sufficient freedom for distributed query optimisation. The correctness of all decomposition algorithms are formally proven in Chapter 6, which adds a strong theoretic foundation to the algorithms.

5. **How can we integrate existing DBMS with P2P overlay networks to provide non-trivial data management facilities to P2P applications?**

   During this Ph.D. work, we have taken some preliminary steps toward an integration of existing XDBMS and DHT network structures. This is described in Section 4.6. The basic idea is to avoid any additional extension to the XQuery language. Section 4.6 proposes two different ways of coupling an XDBMS with one or more DHT networks, i.e., loose coupling and tight coupling. Both couplings rely on a new `dht://` scheme to address resources in the underlying DHT network. In the tight coupling, we have to extend the DHT API with a new function so that a DHT peer can benefit from its local XDBMS to enable a more complex querying facility than the standard DHT API, with which only a complete document can be stored and retrieved by its name.

   The XRPC remote function execution mechanism and the ideas of MonetDB/XQuery* are applied in a P2P application called StreetTiVo [182]. StreetTiVo enables near real-time search in video contents by distributed and parallel execution of compute-intensive video analysis tasks on multiple peers. Our work on the StreetTiVo application confirms our assumption that a P2P middle-ware DBMS could ease the development of data-intensive P2P applications. With XRPC, the rather complex functionalities of StreetTiVo were quickly implemented using just a handful XQuery functions, which in turn are executed on the participating machines.

Theories proposed in this Ph.D. work have also been adopted in practice. XRPC has been included in the open-source XDBMS MonetDB/XQuery, which can be downloaded via `http://monetdb.cwi.nl/Download/`. The software includes the XRPC client and server for distributed XQuery processing, a Java package containing an XRPC Wrapper for cross-system XQuery processing, and an interface for directly making XRPC calls from within web pages. User experience has shown that XRPC is an easy to use mechanism for distributed XML querying.

8.2 Future Work

With this Ph.D. work, we have taken the first steps towards building a middle-ware PDBMS. However, there are still plenty of open areas for future research work. The following questions are particularly interesting.

Distributed Query Placement With the by-projection decomposition algorithm, we are able to decompose almost all XQuery expressions. A natural next step is to decide the placement

for each decomposed subexpression, i.e., to find the optimal peers on which a subexpression should be executed. Basically, this needs a cost-based optimisation that takes network, CPU and data distribution into account [68, 21, 80, 187, 188]. However, in P2P setting, it is unrealistic to assume the availability of these statistics. As a possible solution direction, one could contemplate using *runtime methods* to improve optimisation quality. One idea is to express peer selection criteria in XQuery expressions and attach such expressions to the destination URI of each `execute at` statement. At runtime, when the query (with the peer selection rules added) is evaluated, it will select the best remote peers at that moment, for each decomposed subquery to be executed. We could also borrow ideas from the execution model of Mutant Query Plans [136, 137], in which distributed query plans are evaluated incrementally (by trying to resolve as many as possible logical URNs into URLs at each peer). However, we will not consider exchanging query plans among peers for reasons of interoperability.

Scalable P2P Transaction Management So far, we have only used a strict 2PC protocol for transaction management. In P2P settings, we deem it important to define less strict but more scalable protocols to manage transactions. Distributed Snapshot Isolation (DSI) is especially interesting because it requires weaker locking protocols, which makes it likely to perform better than a classical two-phase locking protocol in high-latency WAN environments. To our knowledge, there has not been much previous work on Distributed Snapshot Isolation, while lately in commercial applications (centralised) snapshot isolation has found wide user acceptance. One idea here is to use Lamport Clocks [116] as the timestamp for DSI, providing the notion of *Lamport consistency*. The objectives of such work would contain a formal definition of this consistency criterion, as well as an analysis of the protocols needed. The StreetTiVo application could be used to validate this approach.

Query Optimisation in MonetDB/XQuery* Our initial work described in MonetDB/XQuery* needs more work to become mature. A first goal of using DHTs is to create "logical URL"s that specify XML data items without necessarily pinning down the actual URL (hostname, path). Such logical URLs may even be used as synonyms for XML data items that are spread over multiple locations. Another benefit of using DHTs is to allow $O(log(N))$ network cost equi-selections (N is the number of peers) and add self-managing properties to the distributed system under churn, i.e. maintaining connectivity, and providing an automatic replication mechanism that prevents data loss. Apart from these design aspects of the XDBMS-DHT integration, another interesting topic is to research which query optimisation possibilities the proposed tight DHT coupling can provide.

A

XML Schema Definition of the XRPC SOAP Messages

```
⟨?xml version="1.0"?⟩
⟨xs:schema version="0.1" xml:lang="EN" xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xrpc="http://monetdb.cwi.nl/XQuery" elementFormDefault="qualified"
           targetNamespace="http://monetdb.cwi.nl/XQuery"⟩
  ⟨xs:element name="atomic-value" type="xs:anySimpleType"/⟩          ⟨!-- simple elements --⟩
  ⟨xs:element name="attribute"/⟩
  ⟨xs:element name="comment"/⟩
  ⟨xs:element name="document"/⟩
  ⟨xs:element name="element"/⟩
  ⟨xs:element name="processing-instruction"/⟩
  ⟨xs:element name="text"/⟩
  ⟨xs:element name="udf-element" type="xrpc:udfElemType"/⟩
  ⟨xs:element name="request" type="xrpc:requestType"/⟩              ⟨!-- complex elements --⟩
  ⟨xs:element name="call" type="xrpc:callType"/⟩
  ⟨xs:element name="response" type="xrpc:responseType"/⟩
  ⟨xs:element name="sequence" type="xrpc:sequenceType"/⟩
  ⟨xs:element name="queryID" type="xrpc:qidType"/⟩
  ⟨xs:attribute name="host" type="xs:anyURI"/⟩                                ⟨!-- attributes --⟩
  ⟨xs:attribute name="timestamp" type="xs:dateTime"/⟩ ⟨!-- micro-seconds --⟩
  ⟨xs:attribute name="timeout" type="xs:double"/⟩ ⟨!-- mini-seconds --⟩
  ⟨xs:attribute name="module" type="xs:string"/⟩
  ⟨xs:attribute name="method" type="xs:string"/⟩
  ⟨xs:attribute name="location" type="xs:anyURI"/⟩
  ⟨xs:attribute name="arity" type="xs:integer"/⟩
  ⟨xs:attribute name="iter-count" type="xs:integer"/⟩
  ⟨xs:attribute name="updCall" type="xs:string"/⟩
  ⟨xs:attribute name="tag" type="xs:string"/⟩
  ⟨xs:attribute name="caller" type="xs:string"/⟩
  ⟨xs:complexType name="qidType"⟩                                          ⟨!-- complex types --⟩
    ⟨xs:attribute ref="xrpc:timestamp" use="required"/⟩
    ⟨xs:attribute ref="xrpc:host" use="required"/⟩
    ⟨xs:attribute ref="xrpc:timeout" use="required"/⟩
```

```
  ⟨/xs:complexType⟩
  ⟨xs:complexType name="udfElemType"⟩
    ⟨xs:sequence⟩⟨xs:any minOccurs="0" maxOccurs="1"/⟩⟨/xs:sequence⟩
  ⟨/xs:complexType⟩
  ⟨xs:complexType name="sequenceType"⟩
    ⟨xs:sequence minOccurs="0" maxOccurs="unbounded"⟩
      ⟨xs:choice⟩
        ⟨xs:element ref="xrpc:atomic-value"/⟩
        ⟨xs:element ref="xrpc:attribute"/⟩
        ⟨xs:element ref="xrpc:comment"/⟩
        ⟨xs:element ref="xrpc:document"/⟩
        ⟨xs:element ref="xrpc:element"/⟩
        ⟨xs:element ref="xrpc:processing-instruction"/⟩
        ⟨xs:element ref="xrpc:text"/⟩
        ⟨xs:element ref="xrpc:udf-element"/⟩
      ⟨/xs:choice⟩
    ⟨/xs:sequence⟩
  ⟨/xs:complexType⟩
  ⟨xs:complexType name="callType"⟩
    ⟨xs:sequence⟩
      ⟨xs:element ref="xrpc:sequence" minOccurs="0" maxOccurs="unbounded"/⟩
    ⟨/xs:sequence⟩
    ⟨xs:attribute ref="xrpc:tag"/⟩
  ⟨/xs:complexType⟩
  ⟨xs:complexType name="requestType"⟩
    ⟨xs:sequence⟩
      ⟨xs:element ref="xrpc:queryID" minOccurs="0" maxOccurs="1"/⟩
      ⟨xs:element ref="xrpc:call" minOccurs="1" maxOccurs="unbounded"/⟩
    ⟨/xs:sequence⟩
    ⟨xs:attribute ref="xrpc:module" use="required"/⟩
    ⟨xs:attribute ref="xrpc:method" use="required"/⟩
    ⟨xs:attribute ref="xrpc:location" use="required"/⟩
    ⟨xs:attribute ref="xrpc:arity" use="required"/⟩
    ⟨xs:attribute ref="xrpc:iter-count"/⟩
    ⟨xs:attribute ref="xrpc:updCall" use="required"/⟩
    ⟨xs:attribute ref="xrpc:caller"/⟩
  ⟨/xs:complexType⟩
  ⟨xs:complexType name="responseType"⟩
    ⟨xs:sequence⟩
      ⟨xs:element ref="xrpc:queryID" minOccurs="0" maxOccurs="1"/⟩
      ⟨xs:element ref="xrpc:sequence" minOccurs="1" maxOccurs="unbounded"/⟩
    ⟨/xs:sequence⟩
    ⟨xs:attribute ref="xrpc:module" use="required"/⟩
    ⟨xs:attribute ref="xrpc:method" use="required"/⟩
  ⟨/xs:complexType⟩
⟨/xs:schema⟩
```

B

XQuery Implementation Defined and Implementation Dependent Features

In this chapter, we give a complete list of *implementation defined* and *implementation dependent* features defined by the XQuery specifications, including [XQuery 1.0 and XPath 2.0 Data Model] [71], [XQuery 1.0: An XML Query Language] [38], [XQuery 1.0 and XPath 2.0 Functions and Operators] [124], [XSLT 2.0 and XQuery 1.0 Serialization] [39], and [XQuery Update Facility 1.0] [58].

B.1 XQuery 1.0 and XPath 2.0 Data Model

Implementation Defined Features

- Support for additional user-defined or implementation defined types.
- Some typed values in the data model are undefined. Attempting to access an undefined property is always an error. Behavior in these cases is implementation defined and the host language is responsible for determining the result.
- The internal structure of the values of the *unparsed-entities* property.
- When mapping a Document Node to a *document information item*, the *declaration base URI* property of the *unparsed entities* property.
- When constructing the *children* property for an Element Node from a PSVI, the relative order of Processing Instruction and Comment Nodes must be preserved, but the position of the Text Node, if it is present, among them is implementation defined.

Implementation Dependent Features

- The relative order of Namespace Nodes nodes is stable but implementation dependent.
- The relative order of Attribute Nodes nodes is stable but implementation dependent.
- The relative order of nodes in distinct trees is stable but implementation dependent.
- The names of anonymous types.
- The prefix associated with type names.

- The representation of the set of prefix/URI pairs returned by the `dm:namespace-bindings` accessor.
- The representation of namespaces, i.e. whether or not they are represented as nodes.
- When constructing the *children* property for an Element Node from a PSVI, where a fixed or default value for an element is defined in the schema, and the element takes this default value, a text node will be created to contain the value, even though there are no character information items representing the value in the PSVI. The position of this text node relative to any comment or processing instruction children is implementation dependent.

B.2 XQuery 1.0: An XML Query Language

Implementation Defined Features

- The *version of Unicode* that is used to construct expressions.
- The *statically-known collations*.
- The *implicit timezone*.
- The circumstances in which warnings are raised, and the ways in which warnings are handled.
- The method by which errors are reported to the external processing environment.
- Whether the implementation is based on the rules of [XML 1.0][49] and [XML Names][47] or the rules of [XML 1.1][50] and [XML Names 1.1][48] is implementation defined. One of these sets of rules must be applied consistently by all aspects of the implementation.
- Any components of the static context or dynamic context that are overwritten or augmented by the implementation.
- Which of the optional axes are supported by the implementation, if the Full-Axis Feature is not supported.
- The default handling of empty sequences returned by an ordering key (sortspec) in an `order by` clause (`empty least` or `empty greatest`).
- The names and semantics of any *extension expressions* (pragmas) recognised by the implementation.
- The names and semantics of any option declarations recognised by the implementation.
- Protocols (if any) by which parameters can be passed to an external function, and the result of the function can returned to the invoking query.
- The process by which the specific modules to be imported by a module import are identified, if the Module Feature is supported (includes processing of location hints, if any.).
- Each *module import* names a target namespace and imports an implementation defined set of modules that share this target namespace.
- In a module import, the `URILiteral`s that follow the `at` keyword are optional location hints, and can be interpreted or disregarded in an implementation defined way.
- Any static typing extensions supported by the implementation, if the Static Typing Feature is supported.

- The means by which serialisation is invoked, if the Serialization Feature is supported.
- The default values for the `byte-order-mark`, `media-type`, `normalization-form`, `encoding`, `omit-xml-declaration`, `standalone` and `version` parameters, if the Serialization Feature is supported.
- The result of an unsuccessful call to an external function (for example, if the function implementation cannot be found or does not return a value of the declared type).
- The following limits on ranges of values is implementation defined:
  - For the `xs:decimal` type, the maximum number of decimal digits (totalDigits facet) (must be at least 18).
  - For the types `xs:date`, `xs:time`, `xs:dateTime`, `xs:gYear`, and `xs:gYearMonth`: the maximum value of the year component and the maximum number of fractional second digits (must be at least 3).
  - For the `xs:duration` type: the maximum absolute values of the years, months, days, hours, minutes, and seconds components.
  - For the `xs:yearMonthDuration` type: the maximum absolute value, expressed as an integer number of months.
  - For the `xs:dayTimeDuration` type: the maximum absolute value, expressed as a decimal number of seconds.
  - For the types `xs:hexBinary`, `xs:base64Binary`, `xs:QName`, `xs:anyURI`, `xs:NOTATION`, `xs:string` and types derived from them: limitations (if any) imposed by the implementation on lengths of values.
- The *in-scope variables* may be augmented by implementation defined variables.

Implementation Dependent Features

- If an implementation does not support the Static Typing Feature, but can nevertheless determine during the static analysis phase that an expression, if evaluated, will necessarily raise a type error at run time, the implementation *may* raise that error during the static analysis phase. The choice of whether to raise such an error at analysis time is implementation dependent.
- Each schema type definition is identified either by an *expanded QName* (for a `named type`) or by an implementation dependent type identifier (for an `anonymous type`).
- Each element declaration is identified either by an expanded QName (for a top-level element declaration) or by an implementation dependent element identifier (for a local element declaration).
- Each attribute declaration is identified either by an expanded QName (for a top-level attribute declaration) or by an implementation dependent attribute identifier (for a local attribute declaration).
- The function implementation for a built-in function or external function,
- The *current dateTime* represents an implementation dependent point in time during the processing of a query, and includes an explicit timezone.

- During the static analysis phase, if the *Static Typing Feature* is not supported, the static types that are assigned to expressions are implementation dependent.
- In addition to the errors defined in this specification, an implementation may raise a dynamic error for a reason beyond the scope of this specification. For example, limitations may exist on the maximum numbers or sizes of various objects. Any such limitations, and the consequences of exceeding them, are implementation dependent.
- The `fn:collection()` function with zero arguments returns the default collection, an implementation dependent sequence of nodes.
- When an unknown schema type is encountered during the process of *SequenceType matching*, an implementation is allowed (but is not required) to provide an implementation dependent mechanism for determining whether the unknown schema type is derived from the expected schema type.
- When evaluating the argument expressions of a function call, the order of argument evaluation is implementation dependent and a function need not evaluate an argument if the function can evaluate its body without evaluating that argument.
- The order of the returned sequence of a path expression, if *ordering mode* is not `ordered`.
- If *ordering mode* is not `ordered`, the resulting sequence of the operators `union`, `intersect`, "|" and `except` is returned in implementation dependent order.
- The order, in which the operands of an arithmetic expression (i.e., `UnaryExpr`, `ValueExpr`, `AdditiveExpr` and `MultiplicativeExpr`) are evaluated.
- The order, in which the operands of a value comparison are evaluated.
- The order, in which the operands of a node comparison are evaluated.
- The order, in which the operands of a logical expression are evaluated.
- For each namespace used in the name of a constructed element or in the names of its attributes, a namespace binding must exist. If a namespace binding does not already exist for one of these namespaces, a new namespace binding is created for it. If the name of the node includes a prefix, that prefix is used in the namespace binding; if the name has no prefix, then a binding is created for the empty prefix. If this would result in a conflict, because it would require two different bindings of the same prefix, then the prefix used in the node name is changed to an arbitrary implementation dependent prefix that does not cause such a conflict, and a namespace binding is created for this new prefix.
- In a `for` clause, if the ordering mode is not `ordered`, the ordering of the variable bindings is implementation dependent.
- In the `order by` and `return` clauses, when two orderspec values are compared to determine their relative position in the ordering sequence, the order of two tuples `T1` and `T2` in the tuple stream is implementation dependent, if nether of the first pair of values encountered during the evaluation is "greater-than" the other, and if `stable` is not specified.
- The order, in which test expressions of a quantified expression (i.e., `QuantifiedExpr`) are evaluated for the various binding tuples, is implementation dependent.
- When evaluating a cast expression (i.e., `CastExpr`), an implementation may determine that one type is derived by restriction from another type either by examining the in-scope schema

definitions or by using an alternative, implementation dependent mechanism such as a data dictionary.

- The handling of an encoding declaration.
- If a version declaration is present, no XQuery `Comment` may occur before the end of the version declaration. If such a `Comment` is present, the result is implementation dependent.
- In a *schema import*, the `URILiteral`s that follow the `at` keyword are optional location hints, and can be interpreted or disregarded in an implementation dependent way.

B.3 XQuery 1.0 and XPath 2.0 Functions and Operators

Implementation Defined Features

- The destination of the trace output.
- For `xs:integer` operations, implementations that support limited-precision integer operations *must* either raise an error [`err:FOAR0002`] or provide an implementation defined mechanism that allows users to choose between raising an error and returning a result that is modulo the largest representable integer value.
- For `xs:decimal` values the number of digits of precision returned by the numeric operators is implementation defined.
- If the number of digits in the result of a numeric operation exceeds the number of digits that the implementation supports, the result is truncated or rounded in an implementation defined manner.
- It is implementation defined which version of Unicode is supported by the features defined in this specification, but it is recommended that the most recent version of Unicode be used.
- For `fn:normalize-unicode()`, conforming implementations *must* support normalisation form "NFC" and *may* support normalisation forms "NFD", "NFKC", "NFKD", "FULLY-NORMALIZED". They *may* also support other normalisation forms with implementation defined semantics.
- The ability to decompose strings into collation units suitable for substring matching is an implementation defined property of a collation.
- All minimally conforming processors *must* support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of 1 millisecond or three digits (i.e., s.sss). However, conforming processors *may* set larger implementation defined limits on the maximum number of digits they support in these two situations.
- The result of casting a string to `xs:decimal`, when the resulting value is not too large or too small but nevertheless has too many decimal digits to be accurately represented, is implementation defined.
- Various aspects of the processing provided by `fn:doc()` are implementation defined. Implementations may provide external configuration options that allow any aspect of the processing to be controlled by the user.
- The manner in which implementations provide options to weaken the *stable* characteristic of `fn:collection()` and `fn:doc()` are implementation defined.

Implementation Dependent Features

- For `fn:error()`, the method by which the `xs:anyURI` or `xs:QName` is returned to the external processing environment is implementation dependent.
- For `fn:error()`, if an invocation provides `$description` and `$error-object`, then these values may also be returned to the external processing environment. The method by which these values are provided to the external environment is implementation dependent.
- The format of the trace output is implementation dependent.
- The ordering of output from invocations of the `fn:trace()` function.
- For `fn:distinct-values()`, the order in which the distinct values are returned.
- For `fn:distinct-values()`, which value of a set of values that compare equal is returned.
- The function `fn:unordered()` returns the items of its parameter `$sourceSeq` in an implementation dependent order.
- The function `fn:max()` selects an item from the input sequence `$arg` whose value is greater than or equal to the value of every other item in the input sequence. If there are two or more such items, then the specific item whose value is returned is implementation dependent.
- The function `fn:min()` selects an item from the input sequence `$arg` whose value is less than or equal to the value of every other item in the input sequence. If there are two or more such items, then the specific item whose value is returned is implementation dependent.
- `fn:min((xs:float(0.0E0), xs:float(-0.0E0)))` can return either positive or negative zero. XML Schema Part 2: Datatypes Second Edition [35] does not distinguish between the values positive zero and negative zero. The result is implementation dependent.
- For `fn:doc()`, its result depends entirely on the run-time environment in which the expression is evaluated. This run-time environment includes not only an unpredictable collection of resources ("the web"), but configurable machinery for locating resources and turning their contents into document nodes within the XPath data model. Both the set of resources that are reachable, and the mechanisms by which those resources are parsed and validated, are implementation dependent.
- The precise instant during the query or transformation represented by the value of `fn:current-dateTime()`, `fn:current-time()` and `fn:current-time()`.
- When casting from `xs:string` and `xs:untypedAtomic`, for `xs:anyURI`, the extent to which an implementation validates the lexical form of `xs:anyURI`.
- When casting to `xs:string` and `xs:untypedAtomic`, for data types that do not have a canonical lexical representation defined, an implementation dependent canonical representation may be used.
- When casting an `xs:float` or `xs:double` to an `xs:string` or `xs:untypedAtomic`, if more than one representation of the same value are valid, it is implementation dependent which of these representations is chosen. For example, the `xs:float` value whose exact decimal representation is $1.26743223E15$ might be represented by any of the strings "1.26743223E15", "1.26743222E15" or "1.26743224E15" (inter alia).

B.4 XSLT 2.0 and XQuery 1.0 Serialization

Implementation Defined Features

- For any implementation defined output method, it is implementation defined whether sequence normalisation process takes place.
- If the namespace URI is non-null for the `method` serialisation parameter, then the parameter specifies an implementation defined output method.
- The effect of additional serialisation parameters on the output of the serialiser, where the name of such a parameter must be namespace-qualified, is implementation defined or implementation dependent. The extent of this effect on the output must not override the provisions of this specification.
- The effect of providing an option that allows the encoding phase to be skipped, so that the result of serialisation is a stream of Unicode characters, is implementation defined. The serialiser is not required to support such an option.
- An serialise may provide an implementation defined mechanism to place `CDATA` sections in the result tree.
- If the value of the `normalization-form` form parameter is not `NFC`, `NFD`, `NFKC`, `NFKD`, `fully-normalized`, or `none` then the meaning of the value and its effect is implementation defined.

Implementation Dependent Features

- The actual octet order used.
- In those cases where they have no important effect on the content of the serialised result, details of the output methods defined by this specification are left unspecified and are regarded as implementation-dependent. Whether a serialise uses apostrophes or quotation marks to delimit attribute values in the XML output method is an example of such a detail.
- If the serialisation method is one of the four methods `xml`, `html`, `xhtml`, or `text`, then the additional serialisation parameters *may* affect the output of the serialise to the extent (but only to the extent) that this specification leaves the output implementation defined or implementation dependent.
- For characters such as > where XML defines a built-in entity but does not require its use in all circumstances, it is implementation dependent whether the character is escaped.
- If the html element is generated by an XSLT literal result element of the form

  ```
  <html xmlns="http://www.w3.org/1999/xhtml">...</html>
  ```

  or by an XQuery direct element constructor of the same form, then the `html` element in the result document will have a node name whose prefix is "", which will satisfy the requirements of the DTD. In other cases the prefix assigned to the element is implementation dependent.

B.5 XQuery Update Facility 1.0

Implementation Defined Features

- The revalidation modes that are supported by this implementation.
- The default revalidation mode for this implementation.
- The mechanism (if any), by which an external function can return an XDM instance and/or a pending update list to the invoking query, is implementation defined.
- The semantics of `fn:put()`, including the kinds of nodes accepted as operands by this function.

Implementation Dependent Features

- If multiple groups of nodes are inserted by multiple insert expressions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation dependent.
- If an `insert into` expression is specified without `as first` or `as last`, the positions of the inserted nodes among the children of the target node are implementation dependent.
- When processing an `upd:applyUpdates`, if as a net result of making all update primitives other than `upd:put` effective, the *children* property of some node contains adjacent text nodes, these adjacent text nodes are merged into a single text node. The string-value of the resulting text node is the concatenated string-values of the adjacent text nodes, with no intervening space added. The node identity of the resulting text node is implementation dependent.
- When processing an `upd:revalidate($top as node(), $revalidation-mode as xs:string)`, if `$revalidation-mode` is `lax`, define `$topV` as the result of the XQuery expression `validate lax {$top}`. During computation of `$topV`, it is necessary to maintain a mapping between each node in `$topV` and the corresponding node (if any) in the subtree rooted at `$top` (this mapping is maintained in an implementation dependent way.)
- Marking of nodes is accomplished in an implementation dependent way – for example, an implementation might maintain a list of marked nodes.
- If an implementation does not support the Update Static Typing Feature, but can nevertheless determine during the static analysis phase that an expression, if evaluated, will necessarily raise a type error at run time, the implementation *may* raise that error during the static analysis phase. The choice of whether to raise such an error at analysis time is implementation dependent.

C

Static Property Analysis Rules for Built-in Functions

C.1 General Rules

Here we list analysis rules that are shared by multiple functions:

- The rule BLTIN^{none2atom} applies to some built-in functions with zero parameters and return a sequence of atomic values:

$$\frac{}{Env \vdash \mathcal{F}() \Rightarrow (), (), \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{none2atom})$$

- The rule BLTIN^{atom2atom} applies to some built-in functions, whose parameters are all atomic typed sequences, and return a sequence of atomic values:

$$\frac{\forall i \in 1..k : Env \vdash E_i \Rightarrow \vec{r}_i, \vec{u}_i, \langle \bot, \bot, \bot \rangle}{Env \vdash \mathcal{F}(E_1, ..., E_k) \Rightarrow (), \bigcup_{i=1}^{k} (\vec{r}_i \cup \vec{u}_i \cup \vec{r}_i/\texttt{descendant::text()}), \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{atom2atom})$$

  Note that XQuery implicitly converts the values of function arguments, thus, in all rules in this section, we add a `descendant::text()` step to each returned path of a parameter[1].
- The rule BLTIN^{item2atom} applies to some built-in functions, which have one or more sequences of `item()` as parameters, and return a sequence of atomic values. Such functions do not access the descendants of node typed items in their parameter sequences:

$$\frac{\forall i \in 1..k : Env \vdash E_i \Rightarrow \vec{r}_i, \vec{u}_i, \langle \bot, \bot, \bot \rangle}{Env \vdash \mathcal{F}(E_1, ..., E_k) \Rightarrow (), \bigcup_{i=1}^{k} (\vec{r}_i \cup \vec{u}_i), \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{item2atom})$$

C.2 Accessory

- `fn:node-name($arg as node()?)  as xs:QName?`

$$\frac{Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:node-name}(E_1) \Rightarrow (), \vec{r}_1 \cup \vec{u}_1, \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{nodename})$$

[1] In XQuery, the actual argument to a function is called an *argument* and the formal argument of a function is called a *parameter*. We use the same terminology here.

- `fn:nilled($arg as node()?)  as xs:boolean?`

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:nilled}(E_1) \mapsto (), \vec{r}_1 \cup \vec{u}_1, \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{nilled})$$

- `fn:string($arg as item()?)  as xs:string`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies.
- `fn:data($arg as item()*) as xs:anyAtomicType*`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies.
- `fn:base-uri($arg as node()?)  as xs:anyURI?`

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:base-uri}(E_1) \mapsto (), \vec{r}_1 \cup \vec{u}_1, \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{baseuri})$$

- `fn:document-uri($arg as node()?)  as xs:anyURI?`

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:document-uri}(E_1) \mapsto (), \vec{r}_1 \cup \vec{u}_1, \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{docuri})$$

C.3 The Error Function

- `fn:error($error as xs:QName?, $description as xs:string, $error-object as item()*) as none`

$$\frac{\forall i \in 1..3 : Env \vdash E_i \mapsto \vec{r}_i, \vec{u}_i, \langle \bot, \bot, \bot \rangle}{\begin{array}{r} Env \vdash \texttt{fn:error}(E_1, E_2, E_3) \mapsto (), \vec{r}_1 \cup \vec{r}_2 \cup \vec{r}_3 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{u}_3 \cup \vec{r}_1/\texttt{descendant::text()} \cup \\ \vec{r}_2/\texttt{descendant::text()} \cup \vec{r}_3/\texttt{descendant-or-self::*}, \langle \eta, \mu, \sigma \rangle \end{array}} \quad (\text{BLTIN}^{error})$$

C.4 The Trace Function

- `fn:trace($value as item()*, $label as xs:string) as item()*`

$$\frac{\begin{array}{c} Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle \\ Env \vdash E_2 \mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash \texttt{fn:trace}(E_1, E_2) \mapsto \vec{r}_1, \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_2/\texttt{descendant::text()}, \langle \eta_1, \mu_1, \sigma_1 \rangle} \quad (\text{BLTIN}^{trace})$$

C.5 Constructor Functions

To the following function, the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `fn:dateTime($arg1 as xs:date?, $arg2 as xs:time?)  as xs:dateTime?`

C.6 Functions and Operators on Numerics

To all functions and operators on numerics, the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `op:numeric-add($arg1 as numeric, $arg2 as numeric) as numeric`
- `op:numeric-subtract($arg1 as numeric, $arg2 as numeric) as numeric`

- `op:numeric-multiply($arg1 as numeric, $arg2 as numeric) as numeric`
- `op:numeric-divide($arg1 as numeric, $arg2 as numeric) as numeric`
- `op:numeric-integer-divide($arg1 as numeric,$arg2 as numeric) as xs:integer`
- `op:numeric-mod($arg1 as numeric, $arg2 as numeric) as numeric`
- `op:numeric-unary-plus($arg as numeric) as numeric`
- `op:numeric-unary-minus($arg as numeric) as numeric`
- `op:numeric-equal($arg1 as numeric, $arg2 as numeric) as xs:boolean`
- `op:numeric-less-than($arg1 as numeric, $arg2 as numeric) as xs:boolean`
- `op:numeric-greater-than($arg1 as numeric, $arg2 as numeric) as xs:boolean`
- `fn:abs($arg as numeric?)  as numeric?`
- `fn:ceiling($arg as numeric?)  as numeric?`
- `fn:floor($arg as numeric?)  as numeric?`
- `fn:round($arg as numeric?)  as numeric?`
- `fn:round-half-to-even($arg as numeric?, $precision as xs:integer) as numeric?`

C.7 Functions on Strings

To all functions on string, the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `fn:codepoints-to-string($arg as xs:integer*) as xs:string`
- `fn:string-to-codepoints($arg as xs:string?)  as xs:integer*`
- `fn:compare($comparand1 as xs:string?,$comparand2 as xs:string?,$collation as xs:string) as xs:integer?`
- `fn:codepoint-equal($comparand1 as xs:string?,$comparand2 as xs:string?)as xs:boolean?`
- `fn:concat($arg1 as xs:anyAtomicType?, $arg2 as xs:anyAtomicType?, ...)  as xs:string`
- `fn:string-join($arg1 as xs:string*, $arg2 as xs:string) as xs:string`
- `fn:substring($sourceString as xs:string?, $startingLoc as xs:double, $length as xs:double) as xs:string`
- `fn:string-length($arg as xs:string?)  as xs:integer`
- `fn:normalize-space($arg as xs:string?)  as xs:string`
- `fn:normalize-unicode($arg as xs:string?, $normalizationForm as xs:string) as xs:string`
- `fn:upper-case($arg as xs:string?)  as xs:string`
- `fn:lower-case($arg as xs:string?)  as xs:string`
- `fn:translate($arg as xs:string?, $mapString as xs:string, $transString as xs:string) as xs:string`
- `fn:encode-for-uri($uri-part as xs:string?)  as xs:string`
- `fn:iri-to-uri($iri as xs:string?)  as xs:string`
- `fn:escape-html-uri($uri as xs:string?)  as xs:string`
- `fn:contains($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:boolean`

- `fn:starts-with($arg1 as xs:string?,$arg2 as xs:string?,$collation as xs:string) as xs:boolean`
- `fn:ends-with($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:boolean`
- `fn:substring-before($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:string`
- `fn:substring-after($arg1 as xs:string?, $arg2 as xs:string?, $collation as xs:string) as xs:string`
- `fn:matches($input as xs:string?, $pattern as xs:string, $flags as xs:string) as xs:boolean`
- `fn:replace($input as xs:string?, $pattern as xs:string, $replacement as xs:string, $flags as xs:string) as xs:string`
- `fn:tokenize($input as xs:string?, $pattern as xs:string, $flags as xs:string) as xs:string*`

C.8 Functions on anyURI

To the following function the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `fn:resolve-uri($relative as xs:string?, $base as xs:string) as xs:anyURI?`

C.9 Functions and Operators on Boolean Values

To the following functions, the general rule BLTIN^{none2atom} (Section C.1) applies:

- `fn:true() as xs:boolean`
- `fn:false() as xs:boolean`

To the following functions, the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `op:boolean-equal($value1 as xs:boolean, $value2 as xs:boolean) as xs:boolean`
- `op:boolean-less-than($arg1 as xs:boolean, $arg2 as xs:boolean) as xs:boolean`
- `op:boolean-greater-than($arg1 as xs:boolean, $arg2 as xs:boolean) as xs:boolean`
- `fn:not($arg as item()*) as xs:boolean`

C.10 Functions and Operators on Durations, Dates and Times

The general rule BLTIN^{atom2atom} (Section C.1) applies to all functions and operators on durations, dates and times:

- `op:yearMonthDuration-less-than($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration) as xs:boolean`
- `op:yearMonthDuration-greater-than($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration) as xs:boolean`
- `op:dayTimeDuration-less-than($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration) as xs:boolean`
- `op:dayTimeDuration-greater-than($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration) as xs:boolean`

- op:duration-equal($arg1 as xs:duration, $arg2 as xs:duration) as xs:boolean
- op:dateTime-equal($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
- op:dateTime-less-than($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
- op:dateTime-greater-than($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:boolean
- op:date-equal($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
- op:date-less-than($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
- op:date-greater-than($arg1 as xs:date, $arg2 as xs:date) as xs:boolean
- op:time-equal($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
- op:time-less-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
- op:time-greater-than($arg1 as xs:time, $arg2 as xs:time) as xs:boolean
- op:gYearMonth-equal($arg1 as xs:gYearMonth, $arg2 as xs:gYearMonth) as xs:boolean
- op:gYear-equal($arg1 as xs:gYear, $arg2 as xs:gYear) as xs:boolean
- op:gMonthDay-equal($arg1 as xs:gMonthDay, $arg2 as xs:gMonthDay) as xs:boolean
- op:gMonth-equal($arg1 as xs:gMonth, $arg2 as xs:gMonth) as xs:boolean
- op:gDay-equal($arg1 as xs:gDay, $arg2 as xs:gDay) as xs:boolean
- fn:years-from-duration($arg as xs:duration?) as xs:integer?
- fn:months-from-duration($arg as xs:duration?) as xs:integer?
- fn:days-from-duration($arg as xs:duration?) as xs:integer?
- fn:hours-from-duration($arg as xs:duration?) as xs:integer?
- fn:minutes-from-duration($arg as xs:duration?) as xs:integer?
- fn:seconds-from-duration($arg as xs:duration?) as xs:decimal?
- fn:year-from-dateTime($arg as xs:dateTime?) as xs:integer?
- fn:month-from-dateTime($arg as xs:dateTime?) as xs:integer?
- fn:day-from-dateTime($arg as xs:dateTime?) as xs:integer?
- fn:hours-from-dateTime($arg as xs:dateTime?) as xs:integer?
- fn:minutes-from-dateTime($arg as xs:dateTime?) as xs:integer?
- fn:seconds-from-dateTime($arg as xs:dateTime?) as xs:decimal?
- fn:timezone-from-dateTime($arg as xs:dateTime?) as xs:dayTimeDuration?
- fn:year-from-date($arg as xs:date?) as xs:integer?
- fn:month-from-date($arg as xs:date?) as xs:integer?
- fn:day-from-date($arg as xs:date?) as xs:integer?
- fn:timezone-from-date($arg as xs:date?) as xs:dayTimeDuration?
- fn:hours-from-time($arg as xs:time?) as xs:integer?
- fn:minutes-from-time($arg as xs:time?) as xs:integer?
- fn:seconds-from-time($arg as xs:time?) as xs:decimal?
- fn:timezone-from-time($arg as xs:time?) as xs:dayTimeDuration?
- op:add-yearMonthDurations($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration) as xs:yearMonthDuration
- op:subtract-yearMonthDurations($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration) as xs:yearMonthDuration

- op:multiply-yearMonthDuration($arg1 as xs:yearMonthDuration, $arg2 as xs:double) as xs:yearMonthDuration
- op:divide-yearMonthDuration($arg1 as xs:yearMonthDuration, $arg2 as xs:double) as xs:yearMonthDuration
- op:divide-yearMonthDuration-by-yearMonthDuration($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration) as xs:decimal
- op:add-dayTimeDurations($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration) as xs:dayTimeDuration
- op:subtract-dayTimeDurations($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration) as xs:dayTimeDuration
- op:multiply-dayTimeDuration($arg1 as xs:dayTimeDuration, $arg2 as xs:double) as xs:dayTimeDuration
- op:divide-dayTimeDuration($arg1 as xs:dayTimeDuration, $arg2 as xs:double) as xs:dayTimeDuration
- op:divide-dayTimeDuration-by-dayTimeDuration($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration) as xs:decimal
- fn:adjust-dateTime-to-timezone($arg as xs:dateTime?) as xs:dateTime?
- fn:adjust-dateTime-to-timezone($arg as xs:dateTime?, $timezone as xs:dayTimeDuration?) as xs:dateTime?
- fn:adjust-date-to-timezone($arg as xs:date?) as xs:date?
- fn:adjust-date-to-timezone($arg as xs:date?, $timezone as xs:dayTimeDuration?) as xs:date?
- fn:adjust-time-to-timezone($arg as xs:time?) as xs:time?
- fn:adjust-time-to-timezone($arg as xs:time?, $timezone as xs:dayTimeDuration?) as xs:time?
- op:subtract-dateTimes($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:dayTimeDuration?
- op:subtract-dates($arg1 as xs:date, $arg2 as xs:date) as xs:dayTimeDuration?
- op:subtract-times($arg1 as xs:time, $arg2 as xs:time) as xs:dayTimeDuration
- op:add-yearMonthDuration-to-dateTime($arg1 as xs:dateTime, $arg2 as xs:yearMonthDuration) as xs:dateTime
- op:add-dayTimeDuration-to-dateTime($arg1 as xs:dateTime, $arg2 as xs:dayTimeDuration) as xs:dateTime
- op:subtract-yearMonthDuration-from-dateTime($arg1 as xs:dateTime, $arg2 as xs:yearMonthDuration) as xs:dateTime
- op:subtract-dayTimeDuration-from-dateTime($arg1 as xs:dateTime, $arg2 as xs:dayTimeDuration) as xs:dateTime
- op:add-yearMonthDuration-to-date($arg1 as xs:date, $arg2 as xs:yearMonthDuration) as xs:date
- op:add-dayTimeDuration-to-date($arg1 as xs:date, $arg2 as xs:dayTimeDuration) as xs:date
- op:subtract-yearMonthDuration-from-date($arg1 as xs:date, $arg2 as xs:yearMonthDuration) as xs:date

- `op:subtract-dayTimeDuration-from-date($arg1 as xs:date, $arg2 as xs:dayTimeDuration) as xs:date`
- `op:add-dayTimeDuration-to-time($arg1 as xs:time, $arg2 as xs:dayTimeDuration) as xs:time`
- `op:subtract-dayTimeDuration-from-time($arg1 as xs:time, $arg2 as xs:dayTimeDuration) as xs:time`

C.11 Functions Related to QNames

- `fn:resolve-QName($qname as xs:string?, $element as element()) as xs:QName?`

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{\begin{array}{l} Env \vdash \texttt{fn:resolve-QName}(E_1, E_2) \Mapsto \\ \qquad (), \vec{r}_1 \cup \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_1/\texttt{descendant::text()}, \langle \eta, \mu, \sigma \rangle \end{array}} \quad (\text{BLTIN}^{resolveqname})$$

- `fn:namespace-uri-for-prefix($prefix as xs:string?, $element as element()) as xs:anyURI?`

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{\begin{array}{l} Env \vdash \texttt{fn:namespace-uri-for-prefix}(E_1, E_2) \Mapsto \\ \qquad (), \vec{r}_1 \cup \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_1/\texttt{descendant::text()}, \langle \eta, \mu, \sigma \rangle \end{array}} \quad (\text{BLTIN}^{nsuri4prefix})$$

- `fn:in-scope-prefixes($element as element()) as xs:string*`

  To this function, the general rule $\text{BLTIN}^{item2atom}$ (Section C.1) applies.

To the remaining functions related to QNames, the general rule $\text{BLTIN}^{atom2atom}$ (Section C.1) applies:

- `fn:QName($paramURI as xs:string?, $paramQName as xs:string) as xs:QName`
- `op:QName-equal($arg1 as xs:QName, $arg2 as xs:QName) as xs:boolean`
- `fn:prefix-from-QName($arg as xs:QName?) as xs:NCName?`
- `fn:local-name-from-QName($arg as xs:QName?) as xs:NCName?`
- `fn:namespace-uri-from-QName($arg as xs:QName?) as xs:anyURI?`

C.12 Operators on base64Binary and hexBinary

To all operators on `base64Binary` and `hexBinary`, the general rule $\text{BLTIN}^{atom2atom}$ (Section C.1) applies:

- `op:hexBinary-equal($value1 as xs:hexBinary, $value2 as xs:hexBinary) as xs:boolean`
- `op:base64Binary-equal($value1 as xs:base64Binary, $value2 as xs:base64Binary) as xs:boolean`

C.13 Operators on NOTATION

To the following function, the general rule $\text{BLTIN}^{atom2atom}$ (Section C.1) applies:

- `op:NOTATION-equal($arg1 as xs:NOTATION, $arg2 as xs:NOTATION) as xs:boolean`

C.14 Functions and Operators on Nodes

- `fn:number($arg as xs:anyAtomicType?) as xs:double`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies.

- `fn:lang($testlang as xs:string?, $node as node()) as xs:boolean`

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{\begin{array}{l} Env \vdash \texttt{fn:lang}(E_1, E_2) \Mapsto (), \vec{r}_1 \cup \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_1/\texttt{descendant::text()} \cup \vec{r}_2/\texttt{ancestor::*} \cup \\ \qquad \vec{r}_2/\texttt{ancestor-or-self::*/attribute::xml:lang}, \langle \eta, \mu, \sigma \rangle \end{array}} \quad (\text{BLTIN}^{lang})$$

  For the function `fn:lang()`, we repeat the rule LANG defined in Section 5.6.1 and extend it with the analysis of the η, μ and σ properties. Since the function `fn:lang()` returns a single atomic value, by Lemma 6.1.12, we have that the result of `fn:lang()` has the properties η, μ and σ.

- `fn:root($arg as node()?) as node()?`

$$\frac{Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:root}(E_1) \Mapsto \bigcup_{i=1}^{|\vec{r}_1|} \vec{r}_1[i]/\texttt{root()}, \vec{u}_1, \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{root})$$

  We repeat the rule ROOT defined in Section 5.6.1 and extend it with the analysis of the η, μ and σ properties. As the function `fn:root()` returns a single node, by Lemma 6.1.12, we have that the result of `fn:root()` has the properties η, μ and σ.

To the remaining functions and operators on nodes, the general rule BLTIN^{item2atom} (Section C.1) applies:

- `fn:name($arg as node()?) as xs:string`
- `fn:local-name($arg as node()?) as xs:string`
- `fn:namespace-uri($arg as node()?) as xs:anyURI`
- `op:is-same-node($parameter1 as node(), $parameter2 as node()) as xs:boolean`
- `op:node-before($parameter1 as node(), $parameter2 as node()) as xs:boolean`
- `op:node-after($parameter1 as node(), $parameter2 as node()) as xs:boolean`

C.15 Functions and Operators on Sequences

C.15.1 General Functions and Operators on Sequences

- `fn:boolean($arg as item()*) as xs:boolean`

  To this function, the general rule BLTIN^{item2atom} (Section C.1) applies.

- `op:concatenate($seq1 as item()*, $seq2 as item()*) as item()*`

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash \texttt{op:concatenate}(E_1, E_2) \Mapsto \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, \langle \varnothing, \varnothing, \varnothing \rangle} \quad (\text{BLTIN}^{concat})$$

- `fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType) as xs:integer*`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies.
- `fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType, $collation as xs:string) as xs:integer*`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies.
- `fn:empty($arg as item()*) as xs:boolean`

  To this function, the general rule BLTIN^{item2atom} (Section C.1) applies.
- `fn:exists($arg as item()*) as xs:boolean`

  To this function, the general rule BLTIN^{item2atom} (Section C.1) applies.
- `fn:distinct-values($arg as xs:anyAtomicType*) as xs:anyAtomicType*`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies.
- `fn:distinct-values($arg as xs:anyAtomicType*, $collation as xs:string) as xs:anyAtomicType*`

  To this function, the general rule BLTIN^{item2atom} (Section C.1) applies.
- `fn:insert-before($target as item()*,$position as xs:integer,$inserts as item()*) as item()*`

$$\frac{\begin{array}{c} Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_3 \Rightarrow \vec{r}_3, \vec{u}_3, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash \texttt{fn:insert-before}(E_1, E_2, E_3) \Rightarrow \vec{r}_1 \cup \vec{r}_3, \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{u}_3, \langle \varnothing, \varnothing, \varnothing \rangle} \quad (\text{BLTIN}^{insertbefore})$$

- `fn:remove($target as item()*, $position as xs:integer) as item()*`

$$\frac{\begin{array}{c} Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle \\ Env \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash \texttt{fn:remove}(E_1, E_2) \Rightarrow \vec{r}_1, \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2, \langle \eta_1, \mu_1, \sigma_1 \rangle} \quad (\text{BLTIN}^{remove})$$

- `fn:reverse($arg as item()*) as item()*`

$$\frac{Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle}{Env \vdash \texttt{fn:reverse}(E_1) \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \neg\sigma_1 \rangle} \quad (\text{BLTIN}^{reverse})$$

  Since `fn:reverse()` returns the reverse of its input sequence, the σ property of the result of `fn:reverse()` is the negation of the σ property of its input sequence.
- `fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double, $length as xs:double) as item()*`

$$\frac{\begin{array}{c} Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle \\ Env \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_3 \Rightarrow \vec{r}_3, \vec{u}_3, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash \texttt{fn:subsequence}(E_1, E_2, E_3) \Rightarrow \vec{r}_1, \vec{r}_2 \cup \vec{r}_3 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{u}_3, \langle \eta_1, \mu_1, \sigma_1 \rangle} \quad (\text{BLTIN}^{subsequence})$$

- `fn:unordered($sourceSeq as item()*) as item()*`

$$\frac{Env \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \bot \rangle}{Env \vdash \texttt{fn:unordered}(E_1) \Rightarrow \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \varnothing \rangle} \quad (\text{BLTIN}^{unordered})$$

C.15.2 Functions that Test the Cardinality of Sequences

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle \quad \mathcal{F} \in \{\texttt{fn:zero-or-one}, \texttt{fn:one-or-more}, \texttt{fn:exactly-one}\}}{Env \vdash \mathcal{F}(E_1) \mapsto \vec{r}_1, \vec{u}_1, \langle \eta_1, \mu_1, \sigma_1 \rangle} \quad (\text{BLTIN}^{card})$$

The rule BLTIN^{card} applies to all functions that test the cardinality of sequences:

- `fn:zero-or-one($arg as item()*) as item()?`
- `fn:one-or-more($arg as item()*) as item()+`
- `fn:exactly-one($arg as item()*) as item()`

C.15.3 Equals, Union, Intersection and Except

- `fn:deep-equal($parameter1 as item()*, $parameter2 as item()*, $collation as xs:string) as xs:boolean`

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \quad Env \vdash E_2 \mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \quad Env \vdash E_3 \mapsto \vec{r}_3, \vec{u}_3, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:deep-equal}(E_1, E_2, E_3) \mapsto (), \bigcup_{i=1}^{3} (\vec{u}_i \cup \vec{r}_i / \texttt{descendant-or-self::}*), \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{deepequal})$$

- `op:union($parameter1 as node()*, $parameter2 as node()*) as node()*`

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \quad Env \vdash E_2 \mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{op:union}(E_1, E_2) \mapsto \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, \langle \eta, \varnothing, \sigma \rangle} \quad (\text{BLTIN}^{union})$$

- `op:intersect($parameter1 as node()*, $parameter2 as node()*) as node()*`

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \mu_1, \bot \rangle \quad Env \vdash E_2 \mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{op:intersect}(E_1, E_2) \mapsto \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, \langle \eta, \mu_1, \sigma \rangle} \quad (\text{BLTIN}^{intersect})$$

- `op:except($parameter1 as node()*, $parameter2 as node()*) as node()*`

$$\frac{Env \vdash E_1 \mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \mu_1, \bot \rangle \quad Env \vdash E_2 \mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{op:except}(E_1, E_2) \mapsto \vec{r}_1 \cup \vec{r}_2, \vec{u}_1 \cup \vec{u}_2, \langle \eta, \mu_1, \sigma \rangle} \quad (\text{BLTIN}^{except})$$

C.15.4 Aggregate Functions

To the following function `fn:count()`, the general rule BLTIN^{item2atom} (Section C.1) applies.

- `fn:count($arg as item()*) as xs:integer`

To all other aggregate functions, the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `fn:avg($arg as xs:anyAtomicType*) as xs:anyAtomicType?`
- `fn:max($arg as xs:anyAtomicType*, $collation as string) as xs:anyAtomicType?`
- `fn:min($arg as xs:anyAtomicType*, $collation as string) as xs:anyAtomicType?`
- `fn:sum($arg as xs:anyAtomicType*, $zero as xs:anyAtomicType?) as xs:anyAtomicType?`

C.15.5 Functions and Operators that Generate Sequences

- `op:to($firstval as xs:integer, $lastval as xs:integer) as xs:integer*`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `fn:id($arg as xs:string*, $node as node()) as element()*`
  `fn:idref($arg as xs:string*, $node as node()) as element()*`

$$\frac{\begin{array}{c} Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle \\ Env \vdash E_2 \Mapsto \vec{r}_2, \vec{u}_2, \langle \bot, \bot, \bot \rangle \end{array}}{Env \vdash \texttt{fn:id}(E_1, E_2) \Mapsto \bigcup_{i=1}^{|\vec{r}_2|} \vec{r}_2[i]/\texttt{id()}, \vec{r}_1 \cup \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_1/\texttt{descendant::text()}, \langle \eta, \varnothing, \sigma \rangle} \quad (\text{BLTIN}^{id})$$

  For the function `fn:id()`, we repeat the rule ID defined in Section 5.6.1 and extend it with the analysis of the η, μ and σ properties. We omit the rule for `fn:idref()`, as it is similar to the rule BLTIN^{id}.

- `fn:doc($uri as xs:string?)  as document-node()?`

$$\frac{Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:doc}(E_1) \Mapsto \texttt{doc}(E_1), \vec{r}_1 \cup \vec{u}_1 \cup \vec{r}_1/\texttt{descendant::text()}, \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{doc})$$

  For the function `fn:doc()`, we repeat the rule DOC defined in Section 5.6.1 and extend it with the analysis of the η, μ and σ properties.

- `fn:doc-available($uri as xs:string?)  as xs:boolean`

  To this function, the general rule BLTIN^{atom2atom} (Section C.1) applies:

- `fn:collection($arg as xs:string?)  as node()*`

$$\frac{Env \vdash E_1 \Mapsto \vec{r}_1, \vec{u}_1, \langle \bot, \bot, \bot \rangle}{Env \vdash \texttt{fn:collection}(E_1) \Mapsto \texttt{doc(*)}, \vec{r}_1 \cup \vec{u}_1 \cup \vec{r}_1/\texttt{descendant::text()}, \langle \eta, \mu, \sigma \rangle} \quad (\text{BLTIN}^{collection})$$

C.16 Context Functions

To all context functions, the general rule BLTIN^{none2atom} (Section C.1) applies:

- `fn:position() as xs:integer`
- `fn:last() as xs:integer`
- `fn:current-dateTime() as xs:dateTime`
- `fn:current-date() as xs:date`
- `fn:current-time() as xs:time`
- `fn:implicit-timezone() as xs:dayTimeDuration`
- `fn:default-collation() as xs:string`
- `fn:static-base-uri() as xs:anyURI?`

References

[1] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CoopIS*, pages 179–194, London, UK, 2001. Springer-Verlag.

[2] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *SIGMOD Rec.*, 32(3):29–33, 2003.

[3] S. Abiteboul. Managing an XML Warehouse in a P2P Context. In *CAiSE*, volume 2681 of *Lecture Notes in Computer Science*, pages 4–13. Springer, June 2003.

[4] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of Asynchronous Discrete Event Systems: Datalog to the Rescue! In *PODS*, pages 358–367, New York, NY, USA, 2005. ACM.

[5] S. Abiteboul, T. Allard, P. Chatalic, G. Gardarin, A. Ghitescu, F. Goasdoué, I. Manolescu, B. Nguyen, M. Ouazara, A. Somani, N. Travers, G. Vasile, and S. Zoupanos. WebContent: efficient P2P Warehousing of web data. *Proceedings of the VLDB Endowment*, 1(2):1428–1431, 2008.

[6] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In *SIGMOD*, pages 227–238, New York, NY, USA, 2004. ACM.

[7] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration. In *VLDB*, pages 1087–1090, February 2002.

[8] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *PODS*, pages 35–45, New York, NY, USA, 2004. ACM.

[9] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML Project: an Overview. *VLDB Journal*, 17(5):1019–1040, August 2008.

[10] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *SIGMOD*, pages 527–538, 2003.

[11] S. Abiteboul, I. Dar, R. Pop, G. Vasile, D. Vodislav, and N. Preda. Large scale P2P distribution of open-source software. In *VLDB*, pages 1390–1393. VLDB Endowment, 2007.

[12] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, pages 606–615, Washington, DC, USA, 2008. IEEE Computer Society.

[13] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and Querying Peer-to-Peer Warehouses of XML Resources. In *SWDB*, pages 219–225, August 2004.

[14] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and Querying Peer-to-Peer Warehouses of XML Resources. In *ICDE*, pages 1122–1123, Washington, DC, USA, April 2005. IEEE Computer Society.

[15] S. Abiteboul, I. Manolescu, and E. Taropa. A Framework for Distributed XML Data Management. In *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 1049–1058. Springer, March 2006.

[16] S. Abiteboul and B. Marinoiu. Distributed monitoring of peer to peer systems. In *WIDM*, pages 41–48, New York, NY, USA, 2007. ACM.

[17] S. Abiteboul, R. Pop, et al. EDOS: Environment for the Development and Distribution of Open Source Software. In *OSS*, July 2005. `http://www.edos-project.org`.

[18] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of active XML systems. In *PODS*, pages 221–230, New York, NY, USA, 2008. ACM.

[19] S. Abiteboul, L. Segoufin, and V. Vianu. Modeling and Verifying Active XML Artifacts. *IEEE Data Engineering Bulletin*, 32(3):10–15, 2009.

[20] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of active xml systems. *TODS*, 34(4):1–44, 2009.

[21] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *SIGMOD*, pages 137–146, New York, NY, USA, June 1996. ACM.

[22] V. Aguilera. XOQL home page - Active XML. `http://www.activexml.net/xoql/`.

[23] R. Akbarinia and V. Martins. Data Management in the APPA System. *Journal of Grid Computing*, 5(3):303–317, September 2007.

[24] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB*, pages 53–64, San Francisco, CA, USA, September 2000. Morgan Kaufmann Publishers Inc.

[25] P. Apers, A. Hevner, and S. Yao. Optimization Algorithms for Distributed Queries. *IEEE TSE*, 9(1):57–68, 1983.

[26] P. M. G. Apers. Data Allocation in Distributed Database Systems. *TODS*, 13(3):263–304, 1988.

[27] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion Project: from Data Integration to Data Coordination. *SIGMOD Record*, 32(3):53–58, 2003.

[28] The ActiveXML Project. `http://activexml.net`.

[29] N. Bales, J. Brinkley, E. S. Lee, S. Mathur, C. Re, and D. Suciu. A Framework for XML-Based Integration of Data, Visualization and Analysis in a Biomedical Domain. In *XSym*, pages 207–221, 2005.

[30] O. Benjelloun. *Active XML: A data-centric perspective on Web services*. PhD thesis, Universite Paris Sud XI, September 2004.

[31] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-Based XML Projection. In *VLDB*, pages 271–282. VLDB Endowment, September 2006.

[32] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, New York, NY, USA, 1995. ACM.

[33] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and llya Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *WebDB*, pages 89–94, June 2002.

[34] E. Bertino. Query decomposition in an object-oriented database system distributed on a local area network. In *RIDE-DOM*, page 2, Washington, DC, USA, 1995. IEEE Computer Society.

[35] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. W3C Recommendation 28 October 2004, October 2004.

[36] D. Biswas. Active XML Replication and Recovery. In *CISIS*, pages 263–269, Washington, DC, USA, 2008. IEEE Computer Society.

[37] D. Biswas and I.-G. Kim. Atomicity for P2P based XML Repositories. In *ICDEW*, pages 363–370, Washington, DC, USA, 2007. IEEE Computer Society.

[38] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation 23 January 2007, January 2007.

[39] S. Boag, M. Kay, J. Tong, N. Walsh, and H. Zongaro. XSLT 2.0 and XQuery 1.0 Serialization. W3C Recommendation 23 January 2007, January 2007.

[40] F. Bonchi, C. Castillo, D. Donato, and A. Gionis. Topical Query Decomposition. In *KDD*, pages 52–60, New York, NY, USA, 2008. ACM.

[41] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, pages 479–490, New York, USA, June 2006. ACM.

[42] A. Bonifati, E. Q. Chang, T. Ho, and L. V. S. Lakshmanan. HePToX: Heterogeneous Peer to Peer XML Databases. *CoRR*, abs/cs/0506002(UBC TR-2005-15), 2005.

[43] A. Bonifati, E. Q. Chang, A. V. S. Lakshmanan, T. Ho, and R. Pottinger. HePToX: marrying XML and heterogeneity in your P2P databases. In *VLDB*, 2005.

[44] A. Bonifati, P. K. Chrysanthis, A. M. Ouksel, and K.-U. Sattler. Distributed Databases and Peer-to-Peer Databases: Past and Present. *SIGMOD Rec.*, 37(1):5–11, 2008.

[45] A. Bonifati and A. Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data & Knowledge Engineering*, 59(2):247–269, 2006.

[46] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, pages 48–55, New York, NY, USA, 2004. ACM.

[47] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.0. W3C Recommendation 16 August 2006, August 2006.

[48] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.1. W3C Recommendation 16 August 2006, August 2006.

[49] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. W3C Recommendation 16 August 2006, August 2006.

[50] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1. W3C Recommendation 16 August 2006, August 2006.

[51] J.-M. Bremer and M. Gertz. On Distributing XML Repositories. In *WebDB*, pages 73–78, June 2003.

[52] S. Bressan, B. Catania, Z. Lacroix, Y. G. Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2):211–240, 2005.

[53] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using Partial Evaluation in Distributed Query Evaluation. In *VLDB*, pages 211–222. VLDB Endowment, September 2006.

[54] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey. Web Services Coordination (WS-Coordination), August 2005.

[55] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte. Web Services Atomic Transaction (WS-AtomicTransaction), August 2005.

[56] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter: Adaptable XML Filtering with Prefix-Caching Suffix-Clustering. In *VLDB*, pages 559–570. VLDB Endowment, 2006.

[57] S. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems*. McGraw-Hill, Inc., Singapore, 1984.

[58] D. Chamberlin, M. Dyck, D. Florescu, J. Melton, J. Robie, and J. Siméon. XQuery Update Facility 1.0. W3C Candidate Recommendation 09 June 2009, June 2009.

[59] D. D. Chamberlin, M. J. Carey, D. Florescu, D. Kossmann, and J. Robie. XQueryP: Programming with XQuery. In *XIME-P*, June 2006.

[60] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB Journal*, 11(4):354–379, 2002.

[61] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Scalable Filtering of Multiple Generalized-Tree-Pattern Queries over XML Streams. *TKDE*, 20(12):1627–1640, 2008.

[62] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Description Language (WSDL) 1.1. W3C Note 15 March 2001, March 2001.

[63] G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *SIGMOD*, pages 509–520, New York, NY, USA, 2007. ACM.

[64] DataDirect XQuery. `http://www.datadirect.com`.

[65] A. de Vries, B. Eberman, and D. Kovalcin. The design and implementation of an infrastructure for multimedia digital libraries. In *IDEAS*, pages 103–110, Cardiff, UK, July 1998.

[66] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, 28(4):467–516, 2003.

[67] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation 23 January 2007, January 2007.

[68] W. Du, R. Krishnamurthy, and M.-C. Shan. Query Optimization in a Heterogeneous DBMS. In *VLDB*, pages 277–291, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[69] eXist Open Source Native XML Database. `http://exist.sourceforge.net`.

[70] M. Fernández, T. Jim, K. Morton, N. Onose, and J. Siméon. DXQ: a distributed XQuery scripting language. In *XIME-P*, pages 1–6, New York, NY, USA, 2007. ACM.

[71] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation 23 January 2007, January 2007.

[72] M. Fernández, N. Onose, and J. Siméon. Yoo-Hoo!: building a presence service with XQuery and WSDL. In *SIGMOD*, pages 911–912, New York, NY, USA, June 2004. ACM.

[73] M. F. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Optimizing sorting and duplicate elimination in xquery path expressions. In *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 554–563. Springer, August 2005.

[74] M. F. Fernàndez, T. Jim, K. Morton, N. Onose, and J. Siméon. Highly distributed XQuery with DXQ. In *SIGMOD*, pages 1159–1161, New York, NY, USA, 2007. ACM.

[75] D. Florescu, A. Grünhagen, and D. Kossmann. XL: an XML programming language for web service specification and composition. In *WWW*, pages 65–76, May 2002.

[76] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *JACM*, 49(6):716–752, 2002.

[77] G. Fourny, D. Kossmann, T. Kraska, M. Pilman, and D. Florescu. XQuery in the browser. In *SIGMOD*, pages 1337–1340, New York, NY, USA, 2008. ACM.

[78] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, and D. McBeath. XQuery in the browser. In *WWW*, pages 1011–1020. ACM, April 2009.

[79] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. Dewitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.

[80] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *VLDB*, pages 378–389, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[81] G. Ghelli, K. Rose, and J. Siméon. Commutativity analysis for XML updates. *TODS*, 33(4):1–47, 2008.

[82] G. Gottlob, Z. Miklos, and T. Schwentick. Generalized hypertree decompositions: np-hardness and tractable variants. In *PODS*, pages 13–22, New York, NY, USA, 2007. ACM.

[83] G. Gou and R. Chirkova. Efficient Algorithms for Evaluating XPath over Streams. In *SIGMOD*, pages 269–280, New York, NY, USA, 2007. ACM.

[84] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. van Engelen, and M. J. Lewis. Toward Characterizing the Performance of SOAP Toolkits. In *GRID*, pages 365–372, Washington, DC, USA, 2004. IEEE Computer Society.

[85] J. Gray and L. Lamport. Consensus on transaction commit. *TODS*, 31(1):133–160, 2006.

[86] S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What Can Peer-to-Peer Do For Databases, and Vice Versa? In *WebDB*, pages 31–36, May 2001.

[87] P. Grosso and D. Weillard. XML Fragment Interchange. W3C Candidate Recommendation 12 February 2001, February 2001.

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

[89] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation 27 April 2007, April 2007.

[90] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation 27 APril 2007, April 2007.

[91] N. Gupta, J. R. Haritsa, and M. Ramanath. Distributed Query Processing on the Web. Technical Report TR-1999-01, Database Systems Lab, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India, 1999.

[92] N. Gupta, J. R. Haritsa, and M. Ramanath. Distributed Query Processing on the Web. In *ICDE*, page 84, Washington, DC, USA, February 2000. IEEE Computer Society.

[93] A. Y. Halevy, O. Etzioni, A. Doan, Z. G. Ives, J. Madhavan, L. Mcdowell, and I. Tatarinov. Crossing the Structure Chasm. In *CIDR*, January 2003.

[94] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW*, 2003.

[95] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management Systems. In *ICDE*, pages 505–516. IEEE, March 2003.

[96] J. Hidders and P. Michiels. Avoiding Unnecessary Ordering Operations in XPath. In *DBPL*, volume 2921/2004 of *Lecture Notes in Computer Science*, pages 113–114. Springer Berlin / Heidelberg, September 2003.

[97] D. Hiemstra, H. Rode, R. van Os, and J. Flokstra. PFTijah: text search in an XML database system. In *OSIR*, pages 12–17. Ecole Nationale Supérieure des Mines de Saint-Etienne, August 2006.

[98] B. D. Homayoun. *Query decomposition in a distributed database system*. PhD thesis, University of Connecticut, Storrs, CT, USA, 1988.

[99] K. Hose, A. Roth, A. Zeitz, K.-U. Sattler, and F. Naumann. A research agenda for query processing in large-scale peer data management systems. *Inf. Syst.*, 33(7-8):597–610, 2008.

[100] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR*, pages 28–43, January 2005.

[101] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, pages 321–332. VLDB Endowment, September 2003.

[102] M. Huijbregts, R. Ordelman, , and F. de Jong. Annotation of Heterogeneous Multimedia Content Using Automatic Speech Recognition. In *SAMT*, volume 4816 of *Lecture Notes in Computer Science*, pages 78–90, Berlin, December 2007. Springer Verlag.

[103] A. H. Igor Tatarinov. Efficient Query Reformulation in Peer-Data Management Systems. In *SIGMOD*, pages 539–550. ACM, June 2004.

[104] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *FPCA*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

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

[106] Y. Kambayashi and M. Yoshikawa. Query processing utilizing dependencies and horizontal decomposition. In *SIGMOD*, pages 55–67, New York, NY, USA, 1983. ACM.

[107] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John. UniStore: Querying a DHT-based Universal Storage. In *ICDE*. IEEE, April 2007.

[108] M. Karnstedt, K.-U. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John. UniStore: Querying a DHT-based Universal Storage. Technical report, LSIR, 2006.

[109] M. Kay. SAXON The XSLT and XQuery Processor. `http://saxon.sourceforge.net`.

[110] J. Knoop and B. Steffen. Code Motion for Explicitly Parallel Programs. *ACM SIGPLAN Nottices*, 34(8):13–24, 1999.

[111] C. Koch, S. Scherzinger, and M. Schmidt. XML Prefiltering as a String Matching Problem. In *ICDE*, April 2008.

[112] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *EDBT*, volume 2992 of *Lecture Notes in Computer Science*, pages 29–47. Springer, March 2004.

[113] G. Koloniari and E. Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges. *SIGMOD Rec.*, 34(2):6–17, 2005.

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

[115] H. Kozankiewicz, K. Stencel, and K. Subieta. Distributed Query Optimization in the Stack-Based Approach. In *HPCC*, pages 904–909. Springer Berlin / Heidelberg, October 2005.

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

[117] J. C. Lavariega and S. D. Urban. An Object Algebra Approach to Multidatabase Query Decomposition in Donají. *Distributed and Parallel Databases*, 12(1):27–71, 2002.

[118] T. T. T. Le, D. D. Doan, V. C. Bhavsar, and H. Boley. A Bottom-up Strategy for Query Decomposition. *International Journal of Innovative Computing and Applications*, 1(3):185–193, 2008.

[119] E. Leclercq, M. Savonnet, M.-N. Terrasse, and K. Tétongnon. Objekt Clustering Methods and a Query Decomposition Strategy for Distributed Objekt-Based Information Systems. In *DEXA*, pages 781–790, London, UK, 1999. Springer-Verlag.

[120] Y. Li, T. Özsu, and K.-L. Tan. XCube: Processing XPath Queries in a Hypercube Overlay Network. *Peer-to-Peer Networking and Applications*, 2(2):128–145, June 2009.

[121] J. A. List, V. Mihajlović, G. Ramírez, A. P. de Vries, D. Hiemstra, and H. E. Blok. Tijah: Embracing information retrieval methods in XML databases. *Information Retrieval Journal*, 8(4):547–570, 2005.

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

[123] L. M. Mackinnon, D. H. Marwick, and M. H. Williams. A Model for Query Decomposition and Answer Construction inHeterogeneous Distributed Database Systems. *JIIS*, 11(1):69–87, 1998.

[124] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation 23 January 2007, January 2007.

[125] A. Marian and J. Siméon. Projecting XML Documents. In *VLDB*, pages 213–224. VLDB Endowment, September 2003.

[126] V. Martins, R. Akbarinia, E. Pacitti, and P. Valduriez. Reconciliation in the APPA P2P System. In *ICPADS*, pages 401–410, Washington, DC, USA, 2006. IEEE Computer Society.

[127] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging Intensional XML Data. In *SIGMOD*, pages 289–300. ACM, June 2003.

[128] N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer. W3C Recommendation 27 April 2007, April 2007.

[129] MonetDB/XQuery. `http://monetdb.cwi.nl`.

[130] A. Ng, S. Chen, and P. Greenfield. An Evaluation of Contemporary Commercial SOAP Implementation. In *AWSA*, pages 64–71, April 2004.

[131] W. S. Ng, B. C. Ooi, and K.-L. Tan. BestPeer: A Self-Configurable Peer-to-Peer System. In *ICDE*, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[132] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: a P2P-based System for Distributed Data Sharing. In *ICDE*, 2003.

[133] R. A. O'Keefe and A. Trotman. The Simplest Query Language That Could Possibly Work. In *INEX Workshop*, pages 167–174, 2004.

[134] N. Onose and J. Siméon. XQuery at Your Web Service. In *WWW*, pages 603–611, New York, NY, USA, May 2004. ACM.

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

[136] V. Papadimos and D. Maier. Distributed Queries without Distributed State. In *WebDB*, pages 95–100, June 2002.

[137] V. Papadimos, D. Maier, and K. Tufte. Distributed Query Processing and Catalogs for Peer-to-Peer Systems. In *CIDR*, January 2003.

[138] E. Pitoura, S. Abiteboul, D. Pfoser, G. Samaras, and M. Vazirgiannis. DBGlobe: a service-oriented P2P system for global computing. *SIGMOD Rec.*, 32(3), 2003.

[139] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation 15 January 2008, January 2008.

[140] IBM DB2 pureXML. `http://www-01.ibm.com/software/data/db2/xml/`.

[141] Qizx. `http://www.qizx.com`.

[142] W. Rao, H. Song, and F. Ma. Querying XML Data over DHT System Using XPeer. In *GCC*, pages 559–566. Springer Berlin / Heidelberg, October 2004.

[143] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *SIGCOMM*, pages 161–172, New York, NY, USA, August 2001. ACM.

[144] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *IIWeb*, pages 116–121. VLDB Endowment, August 2004.

[145] A High-Level Framework for Network-Based Resource Sharing. RFC 707, January 1976.

[146] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *WORLDS*, pages 25–30, Berkeley, CA, USA, December 2005. USENIX Association.

[147] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *ATEC*, pages 10–10, Berkeley, CA, USA, April 2004. USENIX Association.

[148] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: a public DHT service and its uses. In *SIGCOMM*, 2005.

[149] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 1.1: An XML Query Language. W3C Working Draft 15 December 2009, December 2009.

[150] P. Rodríguez-Gianolli, A. Kementsietsidis, M. Garzetti, I. Kiringa, L. Jiang, M. Masud, R. J. Miller, and J. Mylopoulos. Data sharing in the Hyperion peer database system. In *VLDB*, pages 1291–1294. VLDB Endowment, 2005.

[151] A. Roth and F. Naumann. System P: Completeness-driven Query Answering in Peer Data Management Systems. In *BTW*, March 2007.

[152] A. Roth, F. Naumann, T. Hubner, and M. Schweigert. System P: Query Answering in PDMS under Limited Resources. In *IIWeb*, May 2006.

[153] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware*, pages 329–350, London, UK, November 2001. Springer-Verlag.

[154] G. Ruberg and M. Mattoso. XCraft: boosting the performance of active XML materialization. In *EDBT*, pages 299–310, New York, NY, USA, 2008. ACM.

[155] N. Ruberg, G. Ruberg, and I. Manolescu. Towards Cost-based Optimization for Data-intensive Web Service Computations. In *SBBD*, pages 283–297. UnB, 2004.

[156] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self-Organizing XML P2P Database System. In *P2P&DB*, pages 456–465. Springer Berlin / Heidelberg, March 2004.

[157] F. Scarcello, G. Greco, and N. Leone. Weighted hypertree decompositions and optimal query plans. In *PODS*, pages 210–221, New York, NY, USA, 2004. ACM.

[158] R. Schenkel, G. Weikum, N. Weißenberg, and X. Wu. Federated Transaction Management with Snapshot Isolation. In *Proceedings of the 8th International Workshop on Foundations of Models and Languages for Data and Objects - Transactions and Database Dynamics '99*, volume 1773, Dagstuhl Castle, Germany, 1999. Springer.

[159] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985. VLDB Endowment, August 2002.

[160] F. V. Silveira and C. A. Heuser. A Two Layered Approach for Querying Integrated XML Sources. In *IDEAS*, pages 3–11, Washington, DC, USA, 2007. IEEE Computer Society.

[161] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient Processing of XPath Queries with Structured Overlay Networks. In *ODBASE*, volume 3761 of *Lecture Notes in Computer Science*, pages 1243–1260. Springer Berlin / Heidelberg, October 2005.

[162] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, pages 149–160, New York, NY, USA, August 2001. ACM.

[163] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *VLDB Journal*, 5(1), 1996.

[164] S. Subramanian and G. Sindre. An Optimization Rule for ActiveXML Workflows. In *ICWE*, pages 410–418, Berlin, Heidelberg, 2009. Springer-Verlag.

[165] S. Subramanian and G. Sindre. Improving the Performance of ActiveXML Workflows: The Formal Descriptions. In *SCC*, pages 308–315, Washington, DC, USA, 2009. IEEE Computer Society.

[166] D. Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In *VLDB*, pages 227–238, San Francisco, CA, USA, September 1996. Morgan Kaufmann Publishers Inc.

[167] D. Suciu. Distributed query evaluation on semistructured data. *TODS*, 27(1):1–62, 2002.

[168] L. G. A. Sung, N. Ahmed, R. Blankco, H. Li, M. A. Soliman, and D. Hadaller. A Survey of Data Management in Peer-to-Peer Systems. *Web Data Management*, CVS856(Winter 2005):1–50, 2005.

[169] M. Sydow, F. Bonchi, C. Castillo, and D. Donato. Optimising Topical Query Decomposition. In *WSCD*, pages 43–47, New York, NY, USA, 2009. ACM.

[170] K. Tajima and Y. Fukui. Answering XPath Queries over Networks by Sending Minimal Views. In *VLDB*, pages 48–59. VLDB Endowment, August 2004.

[171] I. Tatarinov, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. L. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza peer data management project. *SIGMOD Rec.*, 32(3):47–52, 2003.

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

[173] K. Triantis and C. J. Egyhazy. A Framework for the Study of Query Decomposition for Heterogeneous Distributed Database Management Systems. Technical report, Virginia Polytechnic Institute && State University, Blacksburg, VA, USA, 1987.

[174] P. Valduriez and E. Pacitti. Data Management in Large-Scale P2P Systems. In *VECPAR*, volume 3402 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2005.

[175] E. Wong and K. Youssefi. Decomposition - A Strategy for Query Processing. *TODS*, 1(3):223–241, 1976.

[176] XQilla. `http://xqilla.sourceforge.net/`.

[177] X. Xu, H. Xiang, and J. Chen. Query decomposition based on ontology mapping in data integration system. In *ICNC*, pages 265–269, Washington, DC, USA, 2007. IEEE Computer Society.

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

[179] Y. Zhang and P. Boncz. Integrating XQuery and P2P in MonetDB/XQuery*. In *EROW*, volume 229 of *CEUR Workshop Proceedings*. CEUR-WS.org, January 2007.

[180] Y. Zhang and P. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *VLDB*, pages 99–110. VLDB Endowment, September 2007.

[181] Y. Zhang and P. Boncz. XRPC: Distributed XQuery and Update Processing with Heterogeneous XQuery Engines. In *SIGMOD*, pages 1331–1336, New York, NY, USA, 2008. ACM.

[182] Y. Zhang, A. P. de Vries, P. A. Boncz, D. Hiemstra, and R. Ordelman. StreetTiVo: Using a P2P XML Database System to Manage Multimedia Data in Your Living Room. In *APWeb/WAIM*, volume 5446 of *Lecture Notes in Computer Science*, pages 404–415. Springer, April 2009.

[183] Y. Zhang, N. Tang, and P. A. Boncz. Efficient Distribution of Full-Fledged XQuery. In *ICDE*, pages 565–576. IEEE, March 2009.

[184] Y. Zhang, N. Tang, and P. A. Boncz. Projective Distribution of XQuery with Updates. *TKDE*, 2010. To appear.

[185] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[186] D. Zhao, J. Mylopoulos, I. Kiringa, and V. Kantere. An ECA Rule Rewriting Mechanism for Peer Data Management Systems. In *EDBT*, pages 1069–1078. Springer Berlin / Heidelberg, March 2006.

[187] Q. Zhu and P.-A. Larson. A Query Sampling Method of Estimating Local Cost Parameters in a Multidatabase System. In *ICDE*, pages 144–153, Washington, DC, USA, 1994. IEEE Computer Society.

[188] Q. Zhu and P.-å. Larson. Solving Local Cost Estimation Problem for Global Query Optimization in Multidatabase Systems. *Distrib. Parallel Databases*, 6(4):373–421, 1998.

Summary

While P2P applications that provide trivial keywords search and file sharing features (such as Kazaa, eDonkey) have gained enormous popularity in a short time, the development of P2P applications that provide complex distributed data management and querying facilities advances only slowly. This is because that the development of such applications is still a highly cumbersome task, as applications have to deal with information from different data sources. In P2P settings this set of data sources is extremely dynamic and has an enormously large scale, thus foreseeing all possible combinations of available data sources is impractical. This puts a high adaptivity burden on the shoulders of the application programmers.

To ease the development of data-intensive P2P applications, we envision a P2P XDBMS that acts as a database middle-ware system. It manages dynamic collections of heterogeneous XML data sources (i.e., peers with different software installed) and provides a uniform database abstraction to the application. The ultimate goal is *to research which features such a database abstraction should offer, and how it can be realized efficiently by extending and combining existing XDBMS systems with P2P technologies*.

In our quest for creating P2P XDBMS technology, we first focus on Distributed XDBMS technology, as this area also was unexplored, with an extra requirement that the to be developed technology will serve as a building block for P2P XDBMS technology. The distinction between Distributed and P2P technology is that in the former, users (i.e., application programmers) are aware of on which sites (i.e., peers) data are located. Distributed queries typically involve specific and explicit locations where data are to be queried from. In P2P systems that mainly target large environments where users cannot keep track which data is on which peer and where the group membership is highly volatile (peers enter and leave continuously and unpredictably), users are typically shielded from explicit knowledge where data is located.

In this thesis we have looked into different aspects of Distributed XDBMS including query execution, query optimisation and transaction management. The result of this work is XRPC, a minimal but orthogonal XQuery extension that enables efficient distributed querying of heterogeneous XQuery data sources. XRPC allows any XQuery expressions including the XQUF expressions to be included in a function body and executed on an arbitrary number of (remote) peers using an RPC mechanism. The main design and implementation criteria of XRPC are imposed by the targeted P2P environments: *interoperability*, *efficiency* and *scalability*.

First, the thesis gives a formal definition of the syntax and the semantics of XRPC including the semantics of distributed updates that follow from the use of XQUF updating functions over XRPC. This includes the definitions of two isolation levels for read-only and updating XRPC queries. The experiences in MonetDB/XQuery suggest that adding XRPC to existing XDBMS is easy, as shredding, serialisation and HTTP functionality are usually already present. The work is limited to a small parser extension and stub code generation. Since interoperability is a major goal, XRPC also comprises a SOAP-based network communication protocol SOAP XRPC. Such a SOAP protocol has the additional advantage of seamless integration with web services and AJAX-based GUIs. The SOAP XRPC protocol supports the concept of *Bulk RPC*, i.e., the execution of multiple function calls can be handled in a

single message exchange. This amortises network and parsing latencies and can make XRPC a quite efficient communication mechanism. This thesis shows that the loop-lifting technique, which is pervasively applied in the MonetDB/XQuery system for the translation of XQuery expressions to relational algebra, can easily generate such Bulk RPC requests.

Then, the thesis discusses various aspects of using XRPC in distributed XQuery processing in Chapter 4, First, it shows that XRPC can be easily adopted by *different* XQuery engines, such that complex P2P communication patterns can be programmed using XRPC. To enhance adoption of XRPC, an XRPC Wrapper is described that allows any XQuery data source to handle XRPC calls. The experiments on Saxon show that Bulk RPC enables set-oriented optimisations, such that Bulk RPC execution of a selection function can be handled using a join strategy. To better match the transaction semantics in databases, a deterministic update semantics for XQUF queries is defined and the SOAP XRPC protocol is extended to guarantee a deterministic order in distributed update scenarios. Atomic distributed commit is supported by using a SOAP-based 2PC protocol defined by the industry standard Web Services Atomic Transaction.

Decomposing queries to address multiple data sources is a well-studied optimisation technique in relational, object-oriented and semi-structured databases. While 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. In Chapter 5, the thesis elaborates a framework for distributed execution of full-fledged XQuery (i.e., including XQUF), focusing on the issue of providing *deep-equal query decompositions*, in the face of semantic differences when (parts of) nodes are shipped across the network in XML messages. The thesis proposes a series of decomposition techniques such as pass-by-projection and the use of a novel runtime XML projection method for serialising XML messages, that remove virtually all semantic problems and strongly improve performance. The thesis also defines 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. The correctness of all proposed algorithms is formally proven in Chapter 6.

In this thesis we have also taken a first step towards creating powerful P2P XDBMS technology that preserves the full XQuery language (+XQUF) by extending it only with a single new construct, i.e., XRPC. The thesis proposes MonetDB/XQuery* in which DHTs can be integrated with XDBMS by adding support for a new `dht://` protocol in URIs. Thus no further XQuery extensions are required. The thesis discusses the semantics of two ways of coupling (loose and tight) a DHT with an XDBMS, of which the latter is more complex but powerful.

The XRPC remote function execution mechanism and the ideas of MonetDB/XQuery* are applied in a P2P Information Retrieval application called StreetTiVo. StreetTiVo enables near real-time search in video contents by distributed and parallel execution of compute-intensive video analysis tasks on multiple peers. Our work on the StreetTiVo application confirms the assumption that a P2P middle-ware DBMS could ease the development of data-intensive P2P applications. With XRPC the rather complex functionalities of StreetTiVo were quickly implemented using just a handful of XQuery functions which in turn are executed on the participating machines.

Samenvatting

XRPC: Efficiënte Gespreide Query Verwerking op Heterogene XQuery Systemen

Terwijl Peer-to-Peer (P2P) applicaties (bijvoorbeeld Kazaa en eDonkey) die triviale functies aanbieden zoals het zoeken op trefwoorden en het delen van bestanden, enorm aan populariteit hebben gewonnen in een korte tijd, lijkt de ontwikkeling van P2P applicaties die complexe functies aanbieden, zoals het beheren en opvragen van gedistribueerde data, meer op iets van de lange adem. De reden is dat de ontwikkeling van zulke data-intensieve applicaties nog steeds een zeer moeilijke taak is, omdat de applicaties om moeten gaan met data die uit veel verschillende bronnen vandaan komt. In P2P omgevingen is de complete set van data bronnen uiterst dynamisch en heeft een enorme omvang. Het is onuitvoerbaar om alle mogelijk combinaties van de beschikbare data-bronnen te voorspellen. Deze situatie zet een zware last op de schouders van applicatie-ontwikkelaars.

Om het ontwikkelen van data-intensieve P2P applicaties te vereenvoudigen, voorzien we een P2P XDBMS die de rol van een database middle-ware systeem speelt. Het P2P XDBMS is de tussenpersoon tussen de (bovenliggende) applicaties en de (onderliggende) heterogene XML data bronnen (d.w.z. computers waarop verschillende software programma's zijn geïnstalleerd). Aan de ene kant, beheert het P2P XDBMS de dynamische verzamelingen van de heterogene XML data-bronnen. Aan de andere kant, verbergt het P2P XDBMS de verschillen tussen de heterogene XML data-bronnen en voorziet de (bovenliggende) applicaties van één abstractie van de databases. Ons ultiem doel is om te onderzoeken voor welke functies deze gegevensbanken abstracties moeten aanbieden en hoe deze functies op een efficiënte manier gerealiseerd kunnen worden door de bestaande XDBMS systemen uit te breiden en te combineren met P2P technologiën.

Tijdens onze zoektocht naar het creëren van P2P XDBMS technologiën hebben we ons eerst geconcentreerd op het ontwikkelen van Gedistribueerde XDBMS technologiën, die ook een onontgonnen gebied is. Tegelijkertijd houden we er rekening mee dat de technologiën die we voor de Gedistribueerd XDBMSs bedenken as bouwstenen van de P2P technologiën zouden dienen. De verschillen tussen de Gedistribueerde en P2P technologiën zijn dat, in het eerst geval, de omvang van een systeem klein is en de samenstelling van de deelnemers van het systeem (d.w.z. de participerende computers) heel stabiel is. De gebruikers (d.w.z. de applicatie programmeurs) hebben het overzicht over het hele systeem en weten op welke computers de gegevens zijn opgeslagen. In gedistribueerde systemen bevatten de queries specifieke informatie van locaties waar de data vandaan moet worden gehaald. P2P systemen daarentegen hebben meestal zo'n enorme omvang dat het onmogelijk is voor de gebruikers om precies bij te houden welke data waar is opgeslagen. De samenstelling van de deelnemeners is zeer vluchtig, omdat de computers op elk willekeurig moment zich kunnen aanmelden of afmelden. In P2P systemen worden de gebruikers meestal afgeschermd van de exacte samenstelling van de ondeliggende systemen.

In dit proefschrift hebben we gekeken naar verschillende aspecten in Gedistribueerde XDBMS, inclusief het uitvoeren van queries, de optimalisatie van queries en het beheren van transacties. Het resultaat van dit onderzoekswerk is XRPC, een minimale maar ortho-

gonale uitbreiding van XQuery die het efficiënt en gedistribueerd opvragen van heterogene XQuery data-bronnen mogelijk maakt. Met XRPC kan elke XQuery expressie, inclusief de XQUF expressies, gebruikt worden in de body van een functie die vervolgens uitgevoerd kan worden op een willekeurig aantal computers op willekeurige locaties. De meest belangrijke ontwerp- en implementatie-voorwaarden voor XRPC zijn opgelegd door de eigenschappen van de P2P omgevingen waarvoor XRPC uiteindelijk bedoeld is: *interoperabiliteit*, *efficiëntie* en *schaalbaarheid*.

Het proefschrift begint met een formele definitie van de syntaxis en de semantiek van XRPC, inclusief de semantiek van de gedistribueerde bijwerking van data die uit het gebruik van XQUF updating functies over XRPC volgt. Dit omvat de definities van twee isolatie niveaus voor read-only en updating XRPC queries. De ervaringen met MonetDB/XQuery suggereren dat het toevoegen van XRPC aan de bestaande XDBMSs heel makkelijk is, omdat de functionaliteiten om XML documenten te genereren, te verwerken en te versturen (via HTTP) gewoonlijk al beschikbaar zijn. Alleen een kleine uitbreiding van de parser en het schrijven van de stub code was vereist. Omdat interoperabiliteit een belangrijk doel is van XRPC, omvat XRPC ook een op SOAP gebaseerd netwerk communicatie-protocol SOAP XRPC. Zo een op SOAP gebaseerd protocol heeft als extra voordeel dat het naadloos geïntegreerd kan worden met Web services en op AJAX gebaseerde GUIs. Het SOAP XRPC protocol ondersteunt het zogenaamde *Bulk RPC* concept, wat will zeggen dat het uitvoeren van meerdere functie-aanroepen in één uitwisseling van berichten afgehandeld kan worden. Dit amortiseert de vertragingen die zijn veroorzaakt door de netwerk communicatie en het parseren van berichten, waardoor XRPC een behoorlijk efficiënt communicatie mechanisme kan zijn. Het proefschrift laat zien dat met de loop-lifting techniek – welke overal is toegepast – in MonetDB/XQuery voor het vertalen van XQuery expressies naar de relationele algebra het heel eenvoudig is om Bulk RPC request berichten te genereren.

Vervolgens behandelt het proefschrift verschillende aspecten van het gebruik van XRPC in gedistribueerde verwerking van XQuery in Hoofdstuk 4. Eerst toon het aan dat XRPC makkelijk aangenomen kan worden door *verschillende* XQuery systemen, zodat ingewikkelde P2P communicatie patronen geprogrammeerd kunnen worden met XRPC. Om de toepassing van XRPC te verhogen, bieden we een XRPC Wrapper aan die het mogelijk maakt om voor elke XQuery data bron, XRPC aanroepen te verwerken. De experimenten op Saxon tonen aan dat Bulk RPC set-georiënteerde optimalisaties mogelijk maakt, zodat met Bulk RPC een `select` functie uitgevoerd kan worden met een join strategie die veel efficiënter is. Om de transactie-semantiek in gegevensbanken te behouden, hebben we een deterministische update semantiek voor XQUF queries gedefinieerd. Het SOAP XRPC protocol is uitgebreid om een deterministische volgorde in gedistribueerde update scenario's te garanderen. Atomaire gedistribueerde commit is ondersteund door gebruik te maken van een SOAP gebaseerd 2PC protocol die gedefinieerd is door de industrie standaard Web Service Atomic Transaction.

Query decompositie om meerdere data bronnen te adresseren is een goed bestudeerde optimalisatie techniek in relationele, object-georiënteerde en semi-gestructureerde gegevensbanken. Tewijl veel van de bestaande technieken hergebruikt kunnen worden, hebben het XML datamodel en de XQuery taal een aantal nieuwe uitdagingen geïntroduceerd. Deze uitdagingen draaien rondom de XML node identiteiten en structurele (in plaats van waardegebaseerde) relaties tussen nodes. In Hoofdstuk 5, werkt het proefschrift een framework uit voor het gedistribueerd uitvoeren van volwaardige XQuery (d.w.z. inclusief XQUF), gericht op de kwestie van het verstrekken van *deep-equal query decomposities*, in het aanzicht van

semantische verschillen wanneer (delen van) XML nodes verscheept worden over de netwerken in XML berichten. Het proefschrift stelt een reeks decompositie-technieken zoals pass-by-projection en een nieuwe runtime XML projectie-methode voor voor het serialiseren van XML berichten, die samen bijna alle semantische problemen oplossen en de prestatie van query verwerking sterk verbeteren. Het proefschrift definieert ook de semantiek van het bijwerken van zowel locale as remote XML documenten gebruikmakend van XQUF expressies en de extra beperkingen die toegevoegd moeten worden aan de voorgestelde decompositie technieken om de semantische gelijkwaardigheid van de gedecomposeerde queries te garanderen. De juistheid van alle voorgestelde algoritmes worden formeel bewezen in Hoofdstuk 6.

In dit proefschrift hebben we ook een eerste stap genomen in de richting van het creëren van krachtige P2P XDBMS technologieën die de volwaardige XQuery taal (+XQUF) behouden. Het proefschrift stelt MonetDB/XQuery* voor, waarin DHTs geïntegreerd kunnen worden met XDBMS door het toevoegen van een nieuw `dht://` protocol in URIs. Geen verdere XQuery uitbreidingen zijn nodig. Het proefschrift bediscussieert de semantieken van twee technieken ("loose" of "tight") om een DHT aan een XDBMS te koppelen, waarvan de laatste ingewikkelder maar krachtiger is.

Het mechanisme om functies uit te voeren op computers op afstand en de ideeën van MonetDB/XQuery* zijn toegepast in een P2P Information Retrieval applicatie genaamd StreetTiVo. StreetTiVo maakt bijna real-time zoeken in video mogelijk door gebruik te maken van het gedistribueerd en parallel uitvoeren van reken-intensieve video-analyse-taken op meerdere computers. Het werk dat we gedaan hebben voor de StreetTiVo applicatie bevestigt de aanname dat een P2P middle-ware DBMS de ontwikkeling van data-intensieve P2P applicaties zal vereenvoudigen. Met XRPC waren de tamelijk complexe functionaliteiten van StreetTiVo snel geïmplementeerd gebruikmakend van alleen een handvol XQuery functies die op hun beurt uitgevoerd worden op de deelnemende computers.

SIKS Dissertation Series

1998

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UvA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coupé (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UvA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petković (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UvA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UvA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM)
Politie en de Nieuwe Internationale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UvA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 Floor Verdenius (UvA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM))
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UvA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UvA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)
Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UvA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)
Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UvA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UvA)
Focused Information Access using XML Element Retrieval

2007

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UvA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanović (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramírez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

2008

2008-01 Katalin Boer-Sorbán (EUR)
Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UvA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UvA)
Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU)
Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM)
Adaptive Active Vision

2008-19 Henning Rode (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UvA)
Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.

2008-21 Krisztian Balog (UvA)
People Search in the Enterprise

2008-22 Henk Koning (UU)
Communication of IT-Architecture

2008-23 Stefan Visscher (UU)
Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU)
Using background knowledge in ontology matching

2008-25 Geert Jonker (UU)
Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT)
Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU)
Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN)
On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT)
Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU)
Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM)
Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT)
Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UvA)
Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU)
Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT)
Dendritic morphologies: function shapes structure

2009

2009-01 Rasa Jurgelenaite (RUN)
Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU)
Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT)
A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU)
Understanding Classification

2009-07 Ronald Poppe (UT)
Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU)
Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN)
Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UvA)
Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UvA)
Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin)
Operating Guidelines for Services

2009-13 Steven de Jong (UM)
Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU)
From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 Rinke Hoekstra (UvA)
Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT)
New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT)
Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI)
Armada, an Evolving Database System

2009-19 Valentin Robu (CWI)
Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU)
Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM)
Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT)
Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU)
Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA)
Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI)
RAM: Array Database Management through Relational Mapping

2009-26 Fernando Koch (UU)
An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU)
Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT)
Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT)
Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Żukowski (CWI)
Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UVA)
A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU)
Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT)
How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU)
Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL)
Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN)
Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN)
Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU)
Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin)
Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT)
Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT)
Digital Analysis of Paintings

2009-42 Toine Bogers
Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT)
Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT)
Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU)
Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA)
Querying XML: Benchmarks and Recursion

2010

2010-01 Matthijs van Leeuwen (UU)
Patterns that Matter

2010-02 Ingo Wassink (UT)
Work flows in Life Science

2010-03 Joost Geurts (CWI)
A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT)
Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT)
Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT)
Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT)
A Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL)
Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL)
A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL)
Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD)
The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU)
Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN)
High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU)
Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT)
Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD)
Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU)
Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU)
Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA)
People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT)
Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT)
Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI)
End-user Support for Access to Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU)
The Logical Structure of Emotions

2010-24 Dmytro Tykhonov
Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU)
Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI)
XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines