Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS

RNDr. Jiří Dokulil

Methods for effective querying of RDF data

Department of Software Engineering

Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.

2010
I would like to thank all those who supported me in my doctoral studies and work on my thesis. In the first place I appreciate the help and advice received from my supervisor Jaroslav Pokorný. I would also like to thank the members of our department for their direct and indirect help; special thanks go to David Bednářek, Jakub Yaghob and Filip Zavoral. I am also grateful for the assistance provided by Jana Katreniaková.

Last but not least, I would like to thank my friends and family for their support.

My work was partially supported by the Grant Agency of Charles University, contract number 4271/2009.
Title: Methods for effective querying of RDF data
Author: RNDr. Jiří Dokulil
Department: Department of Software Engineering
       Faculty of Mathematics and Physics
       Charles University in Prague
Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.
Author’s e-mail address: dokulil@ksi.mff.cuni.cz
Supervisor’s e-mail address: pokorny@ksi.mff.cuni.cz

Abstract:
The RDF is one of the basic building blocks of the Semantic Web. It is a low-level data format intended to be used by software developers to create semantic-enabled applications. The ability to place and efficiently evaluate queries is key in this scenario. In this thesis, we approach the problem of RDF querying from three different angles. First, we present an RDF visualization tool, that the developer can use to get an idea about the structure and contents of the data. Second, we have designed extensions of the XQuery language that allow us to give it RDF handling capabilities. The main contribution is introduction of records into the language. Third, to cover query evaluation, we have designed the Bobox parallel framework, which can be used to simplify development of parallel data processing applications. It provides both task and data parallelism.

Keywords: RDF, querying, XQuery, visualization, parallel processing.
Název: Nástroje pro efektivní dotazování nad RDF daty
Author: RNDr. Jiří Dokulil
Pracoviště: Katedra softwarového inženýrství
Mathematickofyzikální fakulta
Univerzita Karlova v Praze
Školitel: Prof. RNDr. Jaroslav Pokorný, CSc.
E-mail autora: dokulil@ksi.mff.cuni.cz
E-mail školitele: pokorny@ksi.mff.cuni.cz

Abstrakt:

Klíčová slova: RDF, dotazování, XQuery, vizualizace, paralelní zpracování.
# Contents

1 Introduction .............................. 1
   1.1 Contribution .......................... 3
   1.2 Outline .............................. 4

2 RDF ..................................... 7
   2.1 Multigraph interpretation ............... 8
   2.2 Formal model .......................... 9
      2.2.1 Blank nodes ....................... 10
      2.2.2 Simple entailment between RDF graphs 11
      2.2.3 RDF vocabulary and RDF entailment 11
   2.3 “Special” cases ....................... 12
      2.3.1 Reifications ....................... 12
      2.3.2 Collections and containers .......... 12
   2.4 Related standards ..................... 13
      2.4.1 RDF Schema (RDFS) ................ 13
      2.4.2 Web Ontology Language (OWL) ....... 14
      2.4.3 RDF/XML .......................... 15

3 RDF visualization ......................... 17
   3.1 Visualization algorithm ................. 18
      3.1.1 Comparison ....................... 18
      3.1.2 Triangle layout algorithm ......... 22
      3.1.3 Vertical range distribution ...... 24
   3.2 Implementation ........................ 27
      3.2.1 Node merging ...................... 27
      3.2.2 Animation ......................... 28
      3.2.3 Examples ......................... 28
   3.3 Determining points of interest .......... 28
      3.3.1 Starting node ..................... 30
      3.3.2 Nodes with high degree .......... 33
   3.4 Big fat graphs ........................ 34
3.5 Multiple views ........................................... 36
3.6 Related work ........................................... 36

4 TriQuery ...................................................... 39
  4.1 XQuery essentials ...................................... 40
  4.2 Records in XQuery .................................... 42
    4.2.1 Definitions and examples ....................... 42
    4.2.2 Pattern matching operation ..................... 47
  4.3 RDF support .......................................... 49
  4.4 Relational data support ............................. 51
  4.5 SP²Bench examples .................................. 51
  4.6 Relation between TriQuery and XQuery .......... 54
  4.7 Implementation concerns ........................... 55
  4.8 Related work ........................................ 56

5 Bobox ....................................................... 59
  5.1 Main design concepts ................................ 60
    5.1.1 Task level parallelism .......................... 60
  5.2 Run-time architecture ............................... 61
    5.2.1 Basic elements and terminology ............... 62
    5.2.2 Execution of the model instance ............... 64
    5.2.3 Boxes ........................................... 64
    5.2.4 Parallel execution environment ............... 67
    5.2.5 Flow control ................................... 70
  5.3 Data level parallelism ............................... 72
    5.3.1 Poisoned pill ................................... 73
    5.3.2 Variable size columns ......................... 74
    5.3.3 Vias ............................................ 75
  5.4 Advanced concepts ................................... 75
    5.4.1 Dynamic pipeline ................................ 76
    5.4.2 Memory allocator ................................ 76
    5.4.3 Bobox server ................................... 77
    5.4.4 Implementation ................................ 78
    5.4.5 The ulibpp library .............................. 79
    5.4.6 Run analysis ................................... 79
  5.5 Experiments ........................................... 82
    5.5.1 SP²Bench ........................................ 82
    5.5.2 Envelope size ................................... 88
    5.5.3 Parallelism ..................................... 89
  5.6 Related work ......................................... 91
6 Conclusion

6.1 Future work ....................................................... 96

A Personal responsibilities ........................................ 107

B TriQ ................................................................. 109

B.1 Data model ......................................................... 109
B.1.1 Decoration of nodes and edges ............................. 110
B.1.2 Meaning of decoration ...................................... 111
B.2 Graph pattern operation ....................................... 112
B.2.1 Definition ...................................................... 112
B.3 Algebraic operations ............................................ 113
B.3.1 Selection .................................................... 113
B.3.2 Projection .................................................... 115
B.3.3 Distinct ..................................................... 115
B.3.4 Joins .......................................................... 115
B.3.5 Group by ..................................................... 116
B.3.6 Aggregation .................................................. 117
B.3.7 Set operations ................................................. 118

C TriQuery grammar .................................................. 119
# List of Figures

3.1 Layout algorithms .............................................. 19  
3.2 Layout algorithm – explanation ................................. 23  
3.3 Example of a tree that requires large area .................... 25  
3.4 Example of a two-layered tree. ................................. 29  
3.5 Example of a large tree. ........................................ 29  
3.6 Additional navigation ........................................... 35  

5.1 Run-time architecture ........................................... 61  
5.2 Pipeline example .............................................. 63  
5.3 Scheduling example .......................................... 69  
5.4 Congestion example ............................................ 71  
5.5 Different table representations ................................. 73  
5.6 Bobox server .................................................. 78  
5.7 Time line of execution .......................................... 81  
5.8 Model for query Q5a ........................................... 85  
5.9 Query performance for different number of threads ........ 90  

B.1 Meaning of decoration ........................................ 111  
B.2 Projection .................................................... 115  
B.3 Left join of names and e-mails ................................ 116  
B.4 Group by ....................................................... 117
Chapter 1

Introduction

The Semantic Web is one of the main trends of development of the World Wide Web. It aims at giving even machines the ability to understand data presented on the Web and thus provide better services to the users [11]. The Resource Description Framework (RDF) provides the means to store the information and metadata in a variety of serialization formats [3, 10]. It is one of the basic building blocks of the Semantic Web and other standards and technologies are built on top of the RDF, for instance the RDFS [15] and OWL [46, 60].

All of the aforementioned technologies have a very strong theoretical background with well defined formal models. However, from the software engineering point of view, there is still a lot of work to be done before they can become truly widespread. One important aspect that may have been somewhat overlooked is providing the software developers with easy to use and efficient tools for handling the data of the Semantic Web. If such tools are not available, the production cycle slows down (and the costs increase) which may hinder the adoption of the new standards by the industry.

Another issue is the performance of semantic tools and applications. Although there was a great improvement in the last years, there is still a lot of work to do. Great part of the software still exists only in the form of scientific prototypes written in Java with a very poor scalability. Also, they rarely follow current high-performance computing trends like parallel processing using SIMD (single instruction, multiple data) stream instructions or multi-core processors.

In this work, we tackle the issues of usability and performance. We focus on the lowest level of the Semantic Web technologies – the RDF. Furthermore, we do not cover the problem of acquiring the data. We assume, that the data was already extracted from its original source, which may be an RDF annotated web page, an export from a relational database or an output from
a semantic analyzer [43, 18, 19]. Our research is focused on querying of data available in some form of an RDF store.

We approach the RDF querying from several angles to cover the whole life cycle of a query. First, we consider the problem of visualization of RDF data, since knowing the overall structure of the queried data helps the user write the queries. The "user" in this context is not the end-user of the Semantic Web but a software developer trying to create an application that works with RDF data. Even though the data usually comes with a schema or an ontology, it is not sufficient for application development – it usually does not provide detailed description of the format of individual fields, whether or not a value of a field tends to be missing etc.

The main concern in RDF visualization is handling the large scale of the data and presenting only the most relevant data in a very compact way. This is complicated by the fact, that the RDF is a very general format, so very few assumptions about the typical structure of the data can be made. On the other hand, the RDF is naturally interpreted as a (multi)graph, which makes the idea of visualizing it very natural.

Only after the user gets a reasonably good idea of what the data look like, can he or she start to formulate queries using a query language. The query language is an interface between a database system (the RDF store) and the user. It defines the query apparatus available to the user and the language that lets him or her use it. The two main concerns in query language design are the expressive power and the ease of use. We believe, that when compared to the SQL, the current RDF query languages fail in combining these aspects. We decided to extend the XQuery language to handle RDF data, since the XQuery standard is already well established, the language is powerful, has a tidy design, and the XQuery community is open to new extensions of the language. Our extension is done in two steps: we extend the XQuery data model and operators with record (tuples) handling and then we use the records to allow the language to process RDF data.

Our modification of XQuery is not a major modification of the language, but it is severe enough to require either a complex modification of an existing evaluation engine or creation of a completely new engine in order to function efficiently. Correct and efficient evaluation of queries is an absolute necessity for any database system. Parallel processing is one of the main trends in speed-critical applications, but writing parallel code is very error prone and any mistakes are hard to discover and correct – for example various race conditions often disappear when the application is run inside a debugger since the debug mode may affect timing, scheduling and synchronization in such a way that the race condition no longer demonstrates. One way of reducing these issues is to use a library or a framework that hides most of
the parallel programming pitfalls. Such framework may be tested separately in a more controlled environment and then used to create the actual evaluation engine, making the whole process easier.

Since none of the libraries and frameworks available today suits our needs, we decided to design a new framework called Bobux. It was designed specifically for parallel data processing - to support development of parallel databases but also other data intensive parallel tasks like data stream processing. It leverages both task-level parallelism and data parallelism. However, it handles only parallelization issues; persistent data storage, query optimization, indexing, data manipulation, and other missing pieces have to be added by the developers that use the framework to create a parallel database engine.

1.1 Contribution

The following section briefly describes main contributions of the thesis, split into three sections that correspond to the main chapters of the text.

RDF visualization The visualization part of the thesis presents several novel ideas that we have introduced for RDF visualization. One of them is the triangle layout algorithm - an algorithm that can be used to draw a rooted tree when nodes are displayed as rectangles with variable size. The ability to handle the different sizes of nodes, especially a wide range of heights, is an improvement over most algorithms used in other RDF visualizers, because they usually handle fixed height equal to one row of text. The ability to display several rows of text allows us to use another new technique - node merging. It is a technique that can greatly reduce the amount of space required to present certain amount of information to the user. In this case, we can display node and its neighbors within one rectangle that is much smaller than the corresponding “full” drawing of such subgraph.

TriQuery The main goal of the query language part of this work is not to provide a definitive RDF query language. It explores alternative approach to RDF querying, since current languages are typically based on SQL (with some notable exceptions [47]). The addition of records to XQuery allows us to provide some level of static query checking, similar to the way that correct table and column names may be checked in SQL. It has also opened several interesting directions for further research, by providing a new way of adding new features to the language (for instance the ability to handle relational
data) and also by giving more information to the evaluation engine, possibly allowing new optimizations.

**Box** Another contribution of this work is design and implementation of the Box parallelization framework. Besides using it for processing of queries over RDF data, it can be used in a much wider set of scenarios where a large amount of data is processed and the whole computation can be divided into several smaller interconnected components. The advantage of the Box design is the fact, that the communication, synchronization and work scheduling is handled by the framework. The individual components that perform the actual computation can be relatively simple – their code, which is supplied by the users of the framework, basically consists only of the algorithm that has to be performed on the data.

We have a working implementation of the key components of the Box framework, including the run-time environment. Even with the most basic implementation, the systems shows a promising performance. This is demonstrated by a partial implementation of queries of one of the most popular SPARQL benchmarks – the SP²Bench [57].

## 1.2 Outline

First, basic principles of the RDF are briefly recollected in the Chapter 2. We describe the features of the RDF relevant to our work.

The Chapter 3 deals with RDF visualization. The Section 3.1 first compares different approaches to graph drawing from the point of view of RDF visualization. Then, we describe the triangle layout algorithm, which was designed based on these criteria. We provide a proof that the algorithm requires only quadratic area. The Section 3.2 provides further information about the visualizer, most notably the node-merging technique. It also describes the animation used in the visualizer and provides examples of the results provided by the visualizer. The Section 3.3 deals with visualization of large RDF collections. It provides alternatives for handling of two basic problems: selecting the starting node for the visualization and handling of nodes with very high degree. The Section 3.4 deals with situation, where the visualizer needs to display a graph that is too large to fit onto the display. The Section 3.5 shows a way in which the concept of tabbed browsing can be adapted from web browsers to RDF visualization. Finally, the Section 3.6 provides related work – our visualizer is compared to some of the other RDF visualizers.
The Chapter 4 covers TriQuery – the modification of XQuery we have created to handle RDF data. Since XQuery is not traditionally related to RDF, we present the basic features of XQuery in the Section 4.1. The way in which the records were added to the XQuery language is described in the Section 4.2. We show the definitions of the newly added operations, the extensions of the existing operations and simple examples. We also present the pattern matching operation which is not essential for the records to work, but we added it to XQuery to allow more user-friendly inclusion of the RDF. The RDF support built on top of the records is described in the Section 4.3. The Section 4.4 shows that the records also allow us to easily add support of relational data. To provide further demonstration of RDF querying, the Section 4.5 shows TriQuery equivalents of some SPARQL queries present in the SP2Bench benchmark. The Section 4.6 more closely explores the relation between TriQuery and XQuery and presents a way in which TriQuery can be translated to pure XQuery. The way in which the extensions affect the query evaluation is described in the Section 4.7. The last section – the Section 4.8 – compares the query capabilities of TriQuery to SPARQL [53] and SeRQL [16] query languages.

The Chapter 5 describes the Bobox parallelization framework. The Section 5.1 gives the main concepts that we followed when we designed the system, most notably the way in which we decided to leverage the task level parallelism. The Section 5.2 gives a detailed description of the design. It describes the way in which the users of the framework specify the components and the structure of the pipeline they want the system to execute, and the way in which the synchronization and scheduling is handled by the Bobox run-time. An important aspect to note is how the system handles the varying speed of execution achieved by various components of the pipeline – load balancing and flow control.

The Section 5.3 describes the way the data is represented during the computation and how it enables data level parallelism. It gives some technical details of the implementation and also addresses the issue of values with variable length. The Section 5.4 provides description of various implementation details and description of less critical parts of the framework, namely the support of a dynamic pipeline, the memory allocator for parallel environment, the architecture of the Bobox server for data processing, technical information about implementation, and debugging tools. The Section 5.5 shows the experiments we have performed with the system. Its first part provides results of some of the queries of the SP2Bench SPARQL benchmark and provides comparison to the in-memory store of Sesame [17]. The second part contains results of several synthetic benchmarks aimed at testing various aspects of the parallelization library itself. Finally, the Section 5.6 presents
related work – alternative parallelization libraries.

The Chapter 6 concludes the text and suggest several directions of further research. The thesis also includes two appendices. The environment in which this research was performed and the people involved are named in the Appendix A. The Appendix B shows the definition of the TriQ algebra – an older model for an RDF query language that later evolved into TriQuery. The Appendix C contains the grammar of the TriQuery language.
Chapter 2

RDF

The RDF – the Resource Description Framework – is a language for representing information about resources in the World Wide Web [45]. It is intended for situations where the information is interpreted by machines rather than people. The resources are identified by URIs (Uniform Resource Identifiers) and they are described in terms of simple properties and property values.

Each specification of a value of a property is a statement. Each RDF statement can be seen as a triple:

- subject – the resource
- predicate – the property being described
- object – the value of the property

The possible values in the RDF are URIs, (typed) literals and blank nodes. The subject is a URI or blank node, the predicate is always an URI, and the object may be any of the three. The RDF data are a set of statements (triples). For a general predicate (there are some cases of pre-defined predicates that are more constrained), there are no limitations on the combination or number of subjects and objects it is used with. For instance, one URI may be used as a subject and an object (or even a predicate) in any number of triples. At the moment, literals may only be used as objects, but the W3C are considering allowing literals to pose also as subjects.

The blank nodes are similar in function to URIs, however, they have no universal identifiers like the URI does. They are only assigned identifiers when they are serialized into some textual representation. These identifiers are unique for each distinct blank node, but they are only valid within one serialized document. Traditionally, the serialized identifiers are written like this: _:john or _:123.
Typed literals use type system introduced by the XSD standard [12],
but it can be extended by new data types as long as they conform to the
abstraction defined by the RDF standard.

The examples in the following text contain URIs. To conserve space and
improve readability, the URIs are shortened using prefixes. The prefixes
include:
ex for http://example.org/
rdf for http://www.w3.org/1999/02/22-rdf-syntax-ns#
dc for http://purl.org/dc/elements/1.1/

Then the term ex:test.html stands for http://example.org/test.html
and dc:creator for http://purl.org/dc/elements/1.1/creator. This
is common notation used for example by the W3C standard documents
and even by the technologies themselves (e.g., qualified names in XML or
SPARQL). The individual components of a triple are delimited either by
commas or just spaces. In the latter case, a period is written after each
triple.

For instance, the information “The web page http://example.org/test.html
was created by John Smith.” may be represented in RDF as the following
triple: ex:test.html, dc:creator, "John Smith".

Assuming that the person John Smith is represented by the URI ex:john,
the information may be represented as ex:test.html, dc:creator, ex:john.
The information about John Smith, for instance the actual name, is then
given as a set of statements about the URI ex:john like this:

ex:john ex:name "John Smith" .
ex:john ex:mail "john@smith.com" .

2.1 Multigraph interpretation

The RDF can be seen as a directed, labeled multigraph. The subjects and
objects are the nodes of the multigraph. Each statement is transformed into
an edge of the multigraph that connects the node that corresponds to the
subject with the node that corresponds to the object. The node is labeled
with the predicate.

This creates a general directed multigraph – it may contain multiple edges
between two nodes or cycles (even cycles on one node). In RDF, one URI
may even be used as a predicate and object – this situation is not directly
reflected by the multigraph interpretation of the RDF (no edge is added to
the multigraph).
The example with John Smith would look like this:

Despite being a multigraph, it is usually referred to as an RDF graph.

2.2 Formal model

The semantics of the RDF are defined as a separate standard [39]. The definition uses model theory to specify the semantics of the formal language: Model theory assumes that the language refers to a “world”, and describes the minimal conditions that a world must satisfy in order to assign an appropriate meaning for every expression in the language. The idea is to provide an abstract, mathematical account of the properties that any such interpretation must have, making as few assumptions as possible about its actual nature or intrinsic structure, thereby retaining as much generality as possible. The chief utility of a formal semantic theory is not to provide any deep analysis of the nature of the things being described by the language or to suggest any particular processing model, but rather to provide a technical way to determine when inference processes are valid, i.e. when they preserve truth.

The basic intuition of model-theoretic semantics is that asserting a sentence makes a claim about the world: it is another way of saying that the world is, in fact, so arranged as to be an interpretation which makes the sentence true. In other words, an assertion amounts to stating a constraint on the possible ways the world might be. In general, the larger an RDF graph is – the more it says about the world – then the smaller the set of interpretations that an assertion of the graph allows to be true - the fewer the ways the world could be, while making the asserted graph true of it.

The exact definition of an interpretation can be found in the appropriate standard, however it is very straightforward and follows the common practice in model theory.
2.2.1 Blank nodes

There is a special type of nodes in an RDF graph called *blank nodes*. Blank nodes are treated as indicating the existence of a thing, without using the name of that thing. They serve a purpose similar to existential variables (whose scope is the whole graph) and they have no meaning outside of the graph where they are used. For example, when merging two RDF graphs, no blank node from the first graph may be identified with a blank node in the second graph – they are not supposed to stray outside the scope of the graph where they were originally defined.

A modified example with John Smith may look like this:

```
ex:test.html dc:creator _:b123 .
_:b123 ex:name "John Smith" .
_:b123 ex:mail "john@smith.com" .
```

The main difference between the two versions is the fact, that one cannot directly reference the blank node that corresponds to John Smith in the second version. The identifier _:b123 used in the example has no meaning outside the example, so one cannot for instance make a query that finds the name connected to _:b123. The only way to reference a blank node is using the “context” of the node – the statements where it is used. For example, one could make a query asking for the mail of the person whose name is John Smith. The advantage is that one does not need to come up with a URI for every node in the RDF graph. In many cases, such name is not necessary, since some nodes are never need to be referenced by a name. A more reasonable example could look like this:

```
ex:john ex:name "John Smith" .
ex:john ex:mail "john@smith.com" .
ex:john ex:street-address _:b567 .
_:b567 ex:street "Long street" .
_:b567 ex:number 123 .
_:b567 ex:zip "XY1 85" .
_:b567 ex:city "Smithstown" .
```

In this case, there may be no need to assign an URI to the street address, since we may intend to always access it starting with John Smith or one of the literals. For example, to make a query that gets a street address for a given name or to get the name of a person that resides on a given address.
2.2.2 Simple entailment between RDF graphs

An important concept in RDF semantics is the entailment. The standard provides a definition under which a set of graphs $S$ (simply) entails another graph $E$ – a set $S$ of RDF graphs entails a graph $E$ if every interpretation which satisfies every member of $S$ also satisfies $E$.

Entailment is the key idea which connects model-theoretic semantics to real-world applications. Making an assertion amounts to claiming that the world is an interpretation which assigns the value true to the assertion. If $A$ entails $B$, then any interpretation that makes $A$ true also makes $B$ true, so that an assertion of $A$ already contains the same “meaning” as an assertion of $B$; one could say that the meaning of $B$ is somehow contained in, or subsumed by, that of $A$. If $A$ and $B$ entail each other, then they both “mean” the same thing, in the sense that asserting either of them makes the same claim about the world [39].

Since the scope of this work are only the low-level aspects of RDF processing, we stay on the level of simple entailment, which does not give any special meaning (semantic conditions) to a particular vocabulary (“keywords”). When querying RDF data, we are usually trying to find a result that is entailed by the queried data. This is similar to finding a subgraph [2], but for instance – we need to handle the blank nodes more carefully, to cope with the locality of their identifiers.

2.2.3 RDF vocabulary and RDF entailment

There are several “keywords” in RDF that have a special meaning, for example `rdf:type` which is used to specify types of resources. Formally, the standard defines an rdf-interpretation, which is more limited than a general interpretation, so as to only allow the intended interpretation of the specified RDF terms like `rdf:type` or `rdf:Property`.

For example, it states that a resource $x$ is a predicate if and only if it is stated to be so (using the triple $x,rdf\text{-}type,rdf\text{-}Property$). The standard also provides an infinite set of axiomatic triples, for example the triple `rdf:type,rdf:type,rdf:Property`.

Then, RDF entailment is defined using rdf-interpretation (“$S$ rdf-entails $E$ when every rdf-interpretation which satisfies every member of $S$ also satisfies $E$”). RDF querying languages often support just some form of simple entailment, so for example a query that lists all triples does not list the axiomatic triples.
2.3 "Special" cases

In a simplified view, an RDF graph may be related to a relational database. Resources of one type correspond to a table, their properties correspond to the columns of the table, each resource corresponds to a row of the table, and each triple is transformed to a cell of the table.

However, the RDF is more flexible and allows to specify more complex situations than the relational database, which we have just described. For example, one may specify more triples with the same subject and predicate but different objects, thus providing multiple values for the property of that object.

Other interesting cases are given in the following sections. They are specified directly in the RDF standard.

2.3.1 Reifications

A reification is a statement about a statement. It allows the users to (indirectly) use a triple as a subject in other triples. This can be used, for instance, to specify that a statement was made by a certain person, or that it originated from a certain source. The standard provides a reification vocabulary which can be used to describe a triple and make further statements about it. An example may look like this (_:xxx denotes a blank node not used elsewhere):

```
_:xxx rdf:type rdf:Statement .
_:xxx rdf:predicate ex:b .
_:xxx rdf:object ex:c .
_:xxx ex:asserted-by ex:John .
```

These statements first describe a triple ex:a,ex:b,ex:c and then make a statement about the triple. Note, that the triple itself may not necessarily be present in the RDF graph.

2.3.2 Collections and containers

The RDF also provides vocabulary for several types of containers and collections. The container may either be a bag, a sequence or a collection of alternatives. The elements of the container are specified using the predicates rdf:_1, rdf:_2 etc.
Collections are list structures connected by head-tail links (using the predicates rdf:first and rdf:rest) and terminated by rdf:nil. An example may look like this:

_:c1 rdf:first <ex:aaa> .
_:c1 rdf:rest _:c2 .
_:c2 rdf:first <ex:bbb> .
_:c2 rdf:rest rdf:nil .

This defines the blank node _:c1 to be the list (ex:aaa, ex:bbb). It also defines _:c2 to be the list (ex:bbb).

2.4 Related standards

The RDF is intended as the basic building block on which new, more advanced Semantic Web technologies can be built. They often use the RDF in two ways. First, they work with data stored as RDF and use the formal semantics provided by the RDF. Second, they use RDF as the format in which the information added by these new technologies is stored. For example, the RDFS schemas and OWL ontologies are stored as a set of RDF triples.

2.4.1 RDF Schema (RDFS)

The RDF does not define or enforce any type constraints on the statements, except for some limitations placed on pre-defined RDF vocabulary. This functionality is added by the RDFS standard (the RDF Schema [15]). It defines vocabulary that can be used to constraints (domain and range) on the predicates and a class hierarchy (parent-child relation) not only among resources, but also predicates. The schema is represented as a set of RDF statements. Consider the following example:

ex:MotorVehicle rdfs:type rdfs:Class .
ex:Van rdfs:subClassOf ex:MotorVehicle .
ex:MiniVan rdfs:subClassOf ex:Van .

The RDFS defines rdfs:subClassOf to be transitive, so ex:MiniVan is also a subclass of ex:MotorVehicle. Note that RDFS allows "multiple inheritance", i.e. the ex:MiniVan is a subclass of two classes.
Another example that demonstrates the domain and range constraints may look like this:

```ruby
ex:hasMother rdf:type rdf:Property .
ex:hasMother rdfs:range ex:Female .
ex:hasMother rdfs:range ex:Person .
ex:hasMother rdfs:domain ex:Person .
```

This defines the property `ex:hasMother` — it is a property of a person but the range must be a person AND a female. Usually, `ex:Female` would be a subclass of `ex:Person` making the requirement that the range must be a person redundant.

### 2.4.2 Web Ontology Language (OWL)

To allow computers to “understand” the data a more complex system was created. The OWL (Web Ontology Language [46, 60]) provides vocabulary and formal semantics for RDF to allow reasoning on the RDF data. It is used to define ontologies, that define meaning of the terms and relationships between the terms. The RDFS vocabulary and semantics is used by OWL, but it provides the users with more ways of specifying constraints, for example to specify cardinality of predicates or the fact than one predicate is inverse of another.

The following example shows a definition of a class `AllStarBaseballTeam`. It is defined as being equivalent to an intersection of a `AllStarTeam` and `BaseballTeam`. Note the use of `_:y` which is an RDF collection that contains `AllStarTeam` and `BaseballTeam`.

```owl
_:AllStarBaseballTeam rdf:type owl:Class .
_:AllStarBaseballTeam owl:equivalentClass _:x .
_:x rdf:type owl:Class .
_:x owl:intersectionOf _:y .
_:y rdf:first _:AllStarTeam .
_:y rdf:rest _:z .
_:z rdf:first _:BaseballTeam .
_:z rdf:rest rdf:nil .
```

In this work, we focus purely on the RDF data, without the semantics that may be added by OWL. We see it as the basic data format, that may be used to build more complex applications that add the semantics to the plain data, for instance the OWL reasoners.
2.4.3 RDF/XML

Another important standard related to the RDF is the RDF/XML [3]. It is a serialization format used to physically store RDF data and in exchange of information. A complete example of RDF document that gives properties of the web page with the RDF/XML standard looks like this:

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
    dc:title="RDF/XML Syntax Specification (Revised)">
    <ex:editor>
      <rdf:Description ex:full_name="Dave Beckett">
        <ex:homePage rdf:resource="http://purl.org/net/dajobe/" />
      </rdf:Description>
    </ex:editor>
  </rdf:Description>
</rdf:RDF>
```

An interesting and useful feature of RDF/XML is the fact, that it uses RDF-specific namespaces, so it may be used in combination with other XML schemas, especially XHTML. This way, the users may add semantic annotation to existing web pages without breaking the page – the browsers that do not understand RDF/XML will ignore it and display the page correctly.
Chapter 3

RDF visualization

The RDF data format is a general data format and since the very beginning it was intended to handle very diverse domains. To work with such data, the developer needs to get some idea of what the data look like.

Since the RDF data is supposed to always come with some kind of schema, usually RDFS, or even ontology (OWL, KAON, . . . ), the developer can use them to get some basic idea of the structure of the data. But, similar to the case of relational databases, seeing just the schema is often not enough, because a lot of useful information can only be obtained from the data or a more detailed documentation. Such information may include formats of the fields (not just the data type but the way in which real data is represented), which fields tend to be filled in and which are often missing, what is the usual length of certain values, etc. If the detailed documentation is missing, the actual data may be the only available source of such information.

If this is the case, the developer could start querying the data using the basic information that he or she acquired from the ontology. But if the database was a relational database, the preferred choice would probably be to display contents of the database tables in a form of a (graphical) table and browse through the data. An RDF visualization software provides similar options for RDF data.

The following parts of the text describe one such RDF visualizer. We have created it to present large RDF data to the user using several novel ideas and algorithms. But there is an important aspect of the visualizer that the reader should keep in mind and that affects the whole design. The target audience (the users) of the visualizer are software developers. Not the actual end users of the Semantic Web. The visualizers displays raw RDF data, which are of very limited use to the end users. Although the graph nature of RDF data makes the idea of visualization very appealing, it may not be what the users expect [58].
Visualization of RDF data brings up several issues. Most important of them is the size of the data. The data can be huge (millions of nodes and edges) and contain nodes with extremely high degree (thousands or even hundreds of thousands). This not only limits the possibilities of drawing the graph but also the acceptable complexity (both time and space) of the drawing algorithm. Traditional graph-based techniques work very well for small graphs. Unfortunately, the difficulty of finding readable layout significantly increases with the increasing size of the graph. We have therefore focused on finding an approach that is effective both from complexity and user point of view. One possibility to partially overcome the problem with large data is to use incremental navigation [40]. We decided to use the incremental navigation enhanced by our novel node merging technique so that we can draw even nodes with large degree. To make the drawing easily readable we proposed a triangle layout algorithm [24, 25].

The Section 3.1 compares different layout algorithm and gives detailed description of the triangle layout. The implementation – including the node merging – is described in the Section 3.2. Sections 3.3 and 3.4 discusses some of the problems with handling very large graphs and the Section 3.5 suggest adopting a “tabbed view” from web browsers to other visualization software. The Section 3.6 puts our proposals into a wider perspective.

3.1 Visualization algorithm

Since the RDF data can be extremely large, some kind of incremental exploration and visualization technique [40] is necessary. The user is given the possibility to explore the neighborhood of the displayed subgraph by extending the displayed part of the graph by one (or more) nodes. This way a navigation tree for the data is created. This tree consists of the nodes and edges that are currently displayed to the user. The tree is rooted at the first node that was used at the start of the navigation. We focus on drawing of the navigation tree. The non-tree edges can be drawn as lines between corresponding nodes [26].

3.1.1 Comparison

There are more approaches to drawing of rooted trees. One of the common techniques is layered drawing where nodes are placed on layers that contain nodes with the same depth (distance from the root). This layers can have different shapes (lines, circles, squares, ...). Examples include:
**Figure 3.1: Layout algorithms**

**Vertical Layered Drawing (Fig. 3.1(a)).** The layers are vertical lines. It is a very simple approach with good results. The paths in the tree are very easy to follow. This approach is used by Experimental RDF Visualizer created in HP labs [56], which is also interesting due to the fact that it avoids non-tree edges by duplicating parts of the graph and transforming it to a tree. Another example of the vertical layout is IsaViz [51].

**Horizontal Layered Drawing.** Is a variant of the vertical layered drawing. It is rarely used because unlike vertical drawing, which offers plenty space for node and edge labels, long node labels make this layout impractical.

**Radial Drawing (Fig. 3.1(b)).** The nodes are placed on concentric circles with increasing diameters. The root of the tree is placed in the center. The nodes are usually displayed as circles but as radial drawing is an extremely common technique, there are plenty of variants. Examples of uses of this technique include gnutella\textvisiblespace\textvisiblespace\textvisiblespace\textvisiblespacev ision [62] and GViz [61].
Square Layout (Fig. 3.1(c)). Square layout is a variant of the radial layout that uses concentric squares instead of circles. It is better suited for drawing rectangular nodes [24].

Triangle Layout (Fig. 3.1(d)). We have introduced the triangle layout as a modification of our older square layout. Unlike the square layout, the triangle layout uses only the first quadrant of the plane (with coordinate origin in the center of the squares). It is further described in the following parts of this chapter.

There are more approaches to drawing trees than just layered drawing, including:

Ferris-Wheel Layout (Fig. 3.1(e)). The Ferris-Wheel layout is inspired by the radial layout but only leaves that are direct neighbors of a node are displayed on a circle around the node. Other nodes are positioned in the drawing space without any sophisticated layout algorithm and positioning them to a ‘good’ position is left to the user. To handle nodes with high degree, the user is given the option to zoom in on one of the circles (called wheels) and gradually explore the nodes by rotating the wheel. This approach is used in PGV [21].

Spring Embedding. Spring Embedding does not specify an exact algorithm for positioning of the nodes. The nodes are connected by springs that either pull them together or push them apart. Then the effect of the springs is simulated until a stable position is reached. In the basic version, the connected nodes are connected by springs that pull them together and unconnected nodes with ones that push them apart. By changing power or direction of the springs, layouts with more complex characteristics can be achieved. This approach is used for instance in RDF Gravity [38].

Although many different techniques can be used for the visualization, it is difficult to find a precise way of evaluating them. We have set up several criteria to compare different layout techniques. Some of these criteria are requirements imposed on the layout algorithms by the nature of the RDF data while other criteria were set up to improve user-friendliness of the resulting application. Note that numbers in parentheses after the criteria definitions correspond to numbers of columns in the Table 3.1.

Data-imposed criteria Based on the experience with real RDF data we can assume that the data will contain nodes with high degree. Even such
A: The radial layout is best suited for drawing circular nodes. With rectangular nodes the available area can be used inefficiently if the nodes are placed onto the layer in a wrong order. If incremental navigation is used, the correct order cannot be maintained without reordering the nodes.

B: The node merging - can be used to handle nodes with high degree.

C: Although the path does not follow a direct route from the center, it generally follows a certain direction without significant deflections.

Table 3.1: Comparison of different layout techniques

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radial Layout</td>
<td>☺ part of annulus wedge</td>
<td>☺</td>
<td>☺</td>
<td>C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Vertical Layered</td>
<td>B whole layer ☺ ☺ ☺ to the right☺</td>
<td>☺</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Horizontal Layered</td>
<td>B whole layer ☺ ☺ downwards☺</td>
<td>☺</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Layout</td>
<td>☺ limited ☺ ☺☺/☺ ☺ ☺ ☺</td>
<td>☺</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Layout</td>
<td>☺ whole layer ☺ ☺ ☺ C ☺</td>
<td>☺</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ferris-Wheel</td>
<td>☺ not limited 0 ☺ ☺ ☺ ☺</td>
<td>☺</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spring Embedding</td>
<td>B not limited 0 ☺ ☺ ☺ ☺</td>
<td>☺</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

nodes should be displayable without making the visualization unreadable to the user (1). For the same reason the area that can be used to draw children of a node should not be too limited (2). Although it may not always be the case, there is a significant chance that number of nodes on each level will be much larger than on the previous one. Thus it may be useful if the size of the layers increases gradually (3).

User-imposed criteria The visualization should be well arranged. But there is no general understanding of what that means [41]. We have picked several criteria we believe are important when working with RDF data.

The user should be able to easily locate ancestor and descendants of a node (4). If the user follows a certain path through the tree, then the whole path should at least roughly maintain the same direction (5). Last but not least, the area required to draw the graph should not be too large (6).

We have evaluated the listed drawing techniques according to the selected criteria. The results are summarized in the Table 3.1. The presented results are either claimed by the authors of the individual algorithms or can be easily deduced by examining the algorithms. Although this is certainly not
a definitive comparison of existing tree drawing techniques, the results show that the idea of triangle layout is worth exploring. We have created an experimental implementation. There is currently no other implementation we are aware of.

The next part of the text gives a more detailed description of the triangle layout algorithm and its properties while Section 3.2 is focused on the implementation.

### 3.1.2 Triangle layout algorithm

The purpose of the algorithm is to determine positions of the nodes from part of the graph that is visible at the moment. The edges of the graph that the user used to reach the visible nodes form a navigation tree $T$ with root $r_T$. The children of a node $v$ are the nodes that were reached by exploring edges connecting them to the node $v$. The order of the children is the same as the order in which they were reached. All nodes with the same distance from the root form a layer. A node with distance $i$ from the root is placed in layer $l_i$ (by $L(h)$ we denote nodes on the level $h$ of the tree and $L(0) = \{r_T\}$).

Layers are represented as lines connecting $[r_i, 0]$ and $[0, r_i]$, where the value $r_i$ is called radius of layer $l_i$.

The nodes are drawn as rectangles $\Gamma(v)$ that are $H(v)$ pixels high and $W(v)$ pixels wide. They are labeled by URI or literal value of the node they represent and also display a list of edges that start or end in the node (for further details see Subsection 3.2.1). The rectangle $\Gamma(v)$ representing node $v \in L(i)$ is placed from the outside of the line representing $l_i$ (we place the lower left corner of the rectangle onto the line). The corner of the rectangle $\Gamma(v)$ that lies on the layer is denoted $\gamma_0(v)$ in the following text, while the opposite corner is denoted $\gamma_1(v)$.

The radius $r_i$ of each level is computed so that $r_{i+1} > r_i$ and to make sure there is enough space to place all nodes that belong to the layer $l_i$. This is influenced by the fact that we place descendants of a node $v$ into a so called angle of influence of the node $v$. The angle of influence is actually defined by two angles that define lower and upper boundary where all descendants (even indirect ones) must fit. This way each path from the root is given a certain direction to follow, which was one of the user-imposed criteria defined in the previous section. Having $\alpha_1(v), \alpha_2(v) \in (0, 90)$ and radius $r$ the vertical range (height of the available space in pixels) available to node $v$ is

$$D(v) = r \cdot \left( \frac{\sin \alpha_1(v)}{\sin \alpha_1(v) + \cos \alpha_1(v)} - \frac{\sin \alpha_2(v)}{\sin \alpha_2(v) + \cos \alpha_2(v)} \right)$$

We fit the successors of $v$ into this vertical range. Let $v_1 \ldots v_k$ be the children
of the node \( v \) and \( \text{let } v_i \) have a size of \( H(v_i) \times W(v_i) \). If the minimal distance between nodes is \( \delta \), then the minimal required vertical space for the children of \( v \) is \( \sum_{i=1}^{k} (H(v_i) + \delta) \). Hence the inequality \( D(v) > \sum_{i=1}^{k} (H(v_i) + \delta) \) should hold.

The layout algorithm first displays the root on the coordinate origin (i.e. \( r_0 = 0 \)). For each depth \( h \) of the tree (beginning with \( h = 1 \)) the algorithm works as follows (see also Fig. 3.2):

Let \( r_{\text{cont}} \) be such radius, that triangle \([r_{\text{cont}}, 0], [0, 0], [0, r_{\text{cont}}]\) completely contains all nodes in layers \( l_1 \ldots l_{h-1} \). For each node \( v \in L(h-1) \) the angle of influence has already been computed. Let \( v_1 \ldots v_k \) be the children of \( v \) and \( H(v_1) \ldots H(v_k) \) their heights. From the inequality \( D(v) > \sum_{i=1}^{k} (H(v_i) + \delta) \) we compute the minimal required radius \( r_{\text{min}} \) for the children of \( v \). Let \( r \) be the maximum of the minimal required radii and the radius \( r_{\text{cont}} \). The nodes from \( L(h) \) will be placed on layer with radius \( r \). Radius \( r \) of the square and the angle of influence of node \( v \) determine the vertical range \( D(v) \) for the sub-tree rooted in \( v \). The distance \( \delta(v) \) between the children of \( v \) has to be recomputed from the inequality \( D(v) > \sum_{i=1}^{k} (H(v_i) + \delta(v)) \). Now, having global parameter \( r \) - radius of the layer and for each node \( v \in L(h-1) \) the parameter \( \delta(v) \), we can compute the display coordinates of children of \( v \) and their angles of influence. More formally, for each \( v_i \) with height \( H(v_i) \) we determine the angle of influence of \( v_i \) and the coordinates of \( \gamma_0(v_i) \).

The angle of influence of the node \( v \) is divided among the children of \( v \) according to a function \( f : V \rightarrow \langle 0, 1 \rangle \) where \( \sum_{i=1}^{k} f(v_i) = 1 \).

**Layout Algorithm**

1. \( \gamma_0(r_T) \leftarrow [0, 0] \)  // Place the root \( r_T \) to the coordinates origin
\( \alpha_1(r_T) \leftarrow 0, \alpha_2(r_T) \leftarrow 90 \)

for each \( h \) in \( \{1, 2, \ldots\} \)

do

for each \( v \) in \( L(h-1) \)
do COUNT\( (r_{\min}(v)) \)

\( r \leftarrow \max\{r_{\text{cont}}, \max\{r_{\min}(v) \mid v \in L(h-1)\}\} \)

for each \( v \) in \( L(h-1) \)
do COUNT\( (\delta(v)) \)

\( D(v) \leftarrow r \cdot \left( \frac{\sin \alpha_2(v)}{\sin \alpha_2(v) + \cos \alpha_2(v)} - \frac{\sin \alpha_1(v)}{\sin \alpha_1(v) + \cos \alpha_1(v)} \right) \)

for each \( v \) in \( L(h-1) \)
do \( \alpha_1(v_0) \leftarrow \alpha_2(v) \), \( \gamma \leftarrow \alpha_2(v) \)

for \( i = 1 \) to \( k \)
do

\( y_{\text{aux}} \leftarrow r \cdot \frac{\sin \gamma}{\sin \gamma + \cos \gamma} - H(v_i) - \delta(v) \)

\( \gamma \leftarrow \arctg \frac{y_{\text{aux}}}{r-y_{\text{aux}}} \)

\( y(v_i) \leftarrow y_{\text{aux}} + \frac{\delta(v)}{2} \)

\( x(v_i) \leftarrow r - y(v_i) \)

\( \alpha_2(v_i) \leftarrow \alpha_1(v_{i-1}) \)

\( y_{\text{aux}} \leftarrow r \cdot \frac{\sin \alpha_2(v_i)}{\sin \alpha_2(v_i) + \cos \alpha_2(v_i)} - f(v_i) \cdot D(v) \)

\( \alpha_1(v_i) \leftarrow \arctg \frac{y_{\text{aux}}}{r-y_{\text{aux}}} \)

3.1.3 Vertical range distribution

The angle of influence of a node is divided among its children according to the function \( f \). Let \( v \) be a node and \( u_1 \ldots u_k \) the children of \( v \). The only constraint for the function \( f \) imposed by the algorithm is that \( \sum_{i=1}^{k} f(u_i) = 1 \). The choice of the function greatly affects the behavior of the visualization algorithm. In [24] we proposed the following definition of \( f \).

\[
\begin{align*}
  f(u_i) &= \frac{H(u_i) + \delta}{\sum_{j=1}^{k} (H(u_j) + \delta)}
\end{align*}
\]

In the following text we use \( r_i^{\text{req}}(v) \) to denote the minimal radius of layer \( l_i \) such that all children of node \( v \in L(i-1) \) fit into the angle of influence of the node \( v \). We also use \( r_i^{\text{req}} \) for \( \max\{r_i^{\text{req}}(v) \mid v \in L(i-1)\} \).

Consider a tree (see Figure 3.3) \( T_{k,p} = (V, E) \) where all nodes are of the same size ( \( H \) and \( W \) ) and

\[
V = \{v_{0,1}\} \cup \{v_{i,j} \mid i \in \{1 \ldots p\} \land j \in \{1 \ldots k\}\}
\]

24
\[ E = \{(v_{i,1}, v_{i+1,j}) | i \in \{0 \ldots p - 1\} \land j \in \{1 \ldots k\}\} \]

On every level of the tree, there is a critical node \( v \) such that \( r_{i+1}^{\text{req}}(v) = r_{i+1}^{\text{req}} \).

Clearly \( v_{0,1} \ldots v_{p-1,1} \) are critical nodes. We denote \( v_{i,1} \) as \( v_i \) in the following text.

For a critical node \( v_i \) the angle of influence covers \( \left( \prod_{j=0}^{i-1} k \right)^{-1} \) of the total vertical range of level \( l_{i+1} \). We need to place \( k \) children of \( v_i \) into this fraction of the vertical range. Thus \( r_{i+1}^{\text{req}} \) of the level \( l_{i+1} \) is \( r_{i+1}^{\text{req}}(v_i) = H \cdot \prod_{j=0}^{i} k \).

The number of nodes in the tree \( T_{k,p} \) is \( N = k \cdot p + 1 \), so the radius of \( l_p \) is

\[ r_p \geq r_p^{\text{req}} = H \cdot k^p = H \cdot \left( \frac{N - 1}{p} \right)^p \]

So the area required to draw the graph grows exponentially with the number of nodes. This is not a good result from the theoretical point of view and it was also confirmed by the implementation of the algorithm using real-world data.

A better choice seems to be such function \( f \) where the value of \( f(u_i) \) is the number of nodes of \( T(u_i) \) (the tree rooted in \( u_i \)) divided by the number of nodes of \( T(v) \). In the following text we will prove, that this function produces better drawings of the tree. The number of the nodes of \( T(v) \) is denoted \( N(v) \) while \( N \) denotes the number of nodes in the whole tree. First, we compute \( H = \max\{H(v) + \delta \mid v \in V\} \) and \( W = \max\{W(v) + \delta \mid v \in V\} \) and use them as the heights and widths of all nodes in the graph and use \( \delta = 0 \).

**Lemma 3.1.1** Every node \( v \) is assigned at least \( \frac{N(v)}{N} \) of the vertical range available to the whole layer that the \( v \) is positioned on.

**Proof 3.1.2** For the root \( r_T \) the statement holds \( \left( \frac{N(v)}{N} = 1 \right) \).
Let \( v \) be a child of \( r_T \). Then \( v \) is assigned \( \frac{N(v)}{N-1} \) of the vertical range and \( \frac{N(v)}{N-1} > \frac{N(v)}{N} \) so the statement holds.

Let \( v \) be a child of \( u \). We already know, that \( u \) was assigned at least \( \frac{N(u)}{N} \) of the vertical range. This vertical range is divided among the children of \( u \). The node \( v \) is assigned \( \frac{N(v)}{N(u)-1} \) of the range of \( u \), which totals to \( \frac{N(u)}{N} \frac{N(v)}{N(u)-1} = \frac{N(u) \cdot N(v)}{N(u)-1} \). Since \( \frac{N(u)}{N(u)-1} > \frac{N(v)}{N} \) the statement holds. \( \square \)

**Lemma 3.1.3** For every node \( v \in L(i-1) \) the required radius \( r_i^{\text{req}}(v) \) is at most \( N \cdot H \).

**Proof 3.1.4** The node \( v \) is assigned at least \( \frac{N(v)}{N} \) of the vertical range. The range has to be divided among the children of \( v \) which means at most \( N(v) - 1 \) nodes. Height of each child is \( H \) so the total height of the children of \( v \) is at most \( H \cdot (N(v) - 1) \). The \( \frac{N(v)}{N} \) fraction of the whole vertical range has to cover the height of the children and since the total vertical range is equal to the radius \( r_i \) the value of \( r_i \) must be big enough for \( r_i \frac{N(u)}{N} \geq H(N(v) - 1) \) to hold. This is equivalent to \( r_i \geq H \frac{N(v)-1}{N(v)} N \). The value \( r_i = N \cdot H \) fulfills this condition. The condition \( r_i \geq \max \{ r_i^{\text{req}}(v) \mid v \in L(i) \} \) implies \( r_i^{\text{req}}(v) \leq r_i = H \cdot N \). \( \square \)

For every layer \( l_i \) of the tree, the radius \( r_i^{\text{req}} \) required to fit all children is lesser than \( N \cdot H \). The actual radius of layer \( l_i \) is one of the following:

- \( r_i^{\text{req}} \) if \( r_{i-1} + (H + W) < r_i^{\text{req}} \)
- \( (H + W) \cdot i \) if the path to the root contains no layer \( j \) where \( r_j^{\text{req}} = r_j \)
- \( r_j^{\text{req}} + (i - j - 1) \cdot (H + W) \) where \( l_j \) is the first layer on the path to the root where \( r_j^{\text{req}} = r_j \)

The maximal number of layers is \( N - 1 \). For the last layer \( l_p \), the \( r_p \) is one of the values:

- \( r_p^{\text{req}} \leq N \cdot H \) (inequation holds due to Lemma 3.1.3)
- \( (H + W) \cdot p \leq (H + W) \cdot N \) since \( p \leq N - 1 \)
- \( r_j^{\text{req}} + (p - j - 1) \cdot (H + W) \leq r_1^{\text{req}} + (p - 2) \cdot (H + W) \leq N \cdot H + N \cdot (H + W) \) (Lemma 3.1.3 and \( p \leq N - 1 \)).
Thus $r_p \leq N \cdot H + N \cdot (H + W) = N(2 \cdot H + W)$. The area required to draw the graph grows quadratically with the number of nodes but also with the value of $H$ and $W$. Since for rooted trees the layered drawings have quadratic area requirement [55], the area is optimal. The widths of the nodes are limited by the length of the longest label in the data. The heights are limited by the highest node degree present in the data. Although both of these numbers could be potentially very large (causing $H$ and $W$ to be large), for practical reasons they can be limited by much lower threshold (only first part of the labels and some of the edges are displayed). The user may still be given another way of accessing the complete information. This approach is used in our implementation.

### 3.2 Implementation

We have created a stand-alone implementation of the visualizer that processes a local file with data. The layout algorithm is implemented independent of the data-source and the user interface. At the moment, there is a SDL-based user interface. This interface displays the drawing to the user and enables him or her to scroll through the whole drawing (it may not fit on the screen) and expand edges by clicking their label in a merged node. For implementation reasons, the drawing is turned upside down compared to the theoretical algorithm, so the origin of the coordinate system is in the upper left corner and the y-axis grows downwards.

#### 3.2.1 Node merging

We use a technique called node merging to help the user navigate the graph. Node does not contain only its label but also a list of incoming and outgoing edges. This allows us to present the neighbors of the node to the user without using too much space. Important advantage of this approach is the fact that the user picks only the neighbors he or she is interested in and the view is then extended only by these nodes. This way we eliminate problem that a RDF node can have thousands (or even hundreds of thousands) of neighbors. Without node merging we would either have to display all of the neighbors which would hardly create a well-arranged and readable drawing of graph or the algorithm would have to pick only a few of the neighbors to display. If node merging is used and the number of neighbors is small, the neighbors can be displayed directly in the node. If the number is higher, the list of neighbors is displayed in a separate window with the option to filter the displayed entries, which allows handling of even nodes with large number of
neighbors.

Node merging is also useful for displaying a certain type of nodes. RDF data usually contain nodes representing certain object with outgoing edges representing its properties, e.g. a person together with his or her name, date of birth, etc. Merged node for the person will contain the name and other information directly so the user can see them without expanding the neighbors. Furthermore a lot of drawing space is conserved since the user will probably be interested in these values and would otherwise expand all of the neighbors which may mean adding tens of nodes.

3.2.2 Animation

When the users expands an edge so that a new node is displayed a drawing of the new tree has to be computed and displayed. To improve the user’s experience the transition between the old drawing and the new drawing is animated in real-time. This not only ‘looks nice’ but more importantly it helps the user maintain connection between objects in the old and the new drawing. Using animation between time-slices to show how nodes and edges are moved to the new positions may also assist in preserving the mental map over time [54].

The animation is a simple linear transition of rectangles that represent nodes and lines that represent edges.

3.2.3 Examples

We have tested the application using the data described in [23]. Figures 3.4 and 3.5 are screenshots of some of the visualizations produced by the system.

Darker nodes represent URIs while white nodes are literals. In the list of incoming and outgoing edges, the black entries can be clicked and new node is displayed, the gray ones represent edges that are already expanded.

3.3 Determining points of interest

The great disproportion between the amount of information that can be stored in an RDF graph and the information that can fit onto a screen means that the user can only see a small portion of the graph. This section deals with several types of situation where the visualizer is forced to choose one or several of many items.
3.3.1 Starting node

Our visualizer starts by displaying one node (in the form of a merged node) and lets the user navigate to other parts of the graph. The problem is selecting the best node to start the visualization.

One important aspect to mind is the size of the displayed data – it is often not possible or viable to load and analyze the whole data when the visualizer starts. The size of the data may exceed the available memory or loading the whole data from the data source might take too long. Even if the visualizer uses a database as its data source, it may not be possible to compute the optimal starting node at the server side, since most such algorithms use graph operations which are poorly supported by most RDF databases [1].

On the other hand, we may leverage the fact that RDF databases can store arbitrary data and that namespaces can be used to store data from different domains without breaking other applications. This way, the visualizer can store some meta-information about the data directly in the database. For instance, one of the nodes can be marked as the best starting node and the visualizer can start by querying the database for such node.

This eliminates the need to compute the starting node every time the visualizer starts, but still it has to be computed at some point. The advantage is, that this process can occur at the server thus eliminating the huge data transfers. For example, application server modules of the Trisokla architecture [34] are ideal for this task, but some solution could be used with any centralized RDF store. This architecture allows us to use even complex analysis of the whole data to select the starting node.

Selected by authority  But let us start with some simpler possibilities. In some cases, the starting node can be specified manually. For example, consider RDF data containing organizational structure – in this case, the root of the structure is probably the best place to start a visualization and it can easily be defined by the database administrator or by the user during his or her previous session.

Random node  Another simple case is the selection of a random node. This would work for data that correspond to a graph with relatively small diameter. In that case, any node can be used to quickly navigate to any part of the graph. The question is determining the random node. One possibility is to use the server-side processor that selects a node and marks it as a root. Another, much simpler, is making a database query whose solution is the list of all triples (or nodes – depending on the query language used). The
visualizer can either choose a random node from the complete result set or simply select the first record returned by the server.

**Center of the RDF graph** A whole class of algorithms can be defined as *graph-based selection*. These focus on the structure of the graph defined by the RDF statements and usually require some server-side processing since they process the whole data. Obvious examples are finding the center of the graph or finding the node with the highest degree. The idea is that such nodes provide good access to the rest of the graph. Using the center of the graph produces undesirable result if the graph contains a long path – in that case the start node can easily be near the center of the path forcing the user to navigate to one end. This situation can occur if there is a big collection represented as a first-rest type of a linked list.

**Node with highest degree** Using a node with the highest degree is even more likely to select a bad starting point. The reason is, that e.g. node that represents the numeric literal 1 is very likely to have a high degree and it may be connected to many different types of nodes. In that case, we face a problem very similar to the starting node selection problem, only the number of possibilities is limited.

**PageRank** An interesting option is some variant of the PageRank [49] algorithm. But it is problematic, since in the visualization incoming and outgoing edges are equally important and the PageRank would likely select a node with high number of incoming edges, resulting in the same problem that was described for the highest degree selecting algorithm.

**Search dialog** A completely different approach is to start the visualizer with a search dialog that allows the user to select a node. The problem is that the user has to be at least partially familiar with the data. An alternative is to list all predicates used in the data. Their number should be much lower, usually tens or hundreds of different URIs. After that, some of the previous techniques can be used to select the starting node with the limitation that it must be an endpoint of an edge labeled by the selected predicate. For instance, one such node can be selected randomly.

**Visit history** Yet another variant is to strengthen the cooperation between the server and the visualizer. As the user navigates the data, the visualizer informs the server about the nodes the user visited and the server keeps the statistics. As was mentioned earlier, this information can be stored by the
RDF store alongside the actual data. The starting node is determined by randomly selecting one of the nodes that was visited most times. Note that displaying a node as one item in the merged node is not considered to by a visit to the node. The user must actually navigate to the node – this eliminates the problems caused by common literals (like the number 1).

**Unconnected graphs** So far we have assumed that the visualized graph is connected, i.e. there is a (non-directed) path between any two nodes. But the RDF data need not look that way, even though they usually do, since for example the whole data may share the same class hierarchy or two parts may be connected via a literal.

If the data is not connected and we cannot use server-side analysis, there is no way to handle the problem since we need (almost) the whole data to check whether it is connected or not.

With server side processing we can identify the components of the graph and then select starting node for each of them separately. Then we create a virtual node and connect it to all of the individual starting nodes. The virtual node and the new edges are sent to the user as the starting (merged) node. While we could present the user with some list of components, this solution should be just as useful and does not introduce new concepts to the visualizer application.

Based on these options, we have decided to use the following solution. If the data on the server contain information about preferred starting node, it is used. If no server side processing can be done and the data is read from a remote data source (a database), we can only use a random starting node. This means, that if the data is not connected, the user cannot access the whole data. If the data can be made available locally (e.g., if the visualizer is used to process one local RDF file or we know that the size of the data is comparable to the transfer rate from the server and locally available memory) we use the random node approach but extended with component handling described in the previous paragraph. If server-side processing is available we perform component analysis on the server and then return random node for each component.

A search dialog is an interesting option for any situation, but its design and capabilities have to be tailored to the actual data access layer used by the visualizer. Still, it should be present at least in some limited form in all situations. To at least handle the situation where the user is interested in one specific node (i.e., the URI or literal value is known).

It is tempting to include the statistics of users’ behavior. The problem with this approach is the use-case we are trying to handle, i.e. pure RDF
visualization for software developers. For this approach to work, we need to collect reasonably large sample. This means having large group of developers working on the same data set. When new developer joins the team, he or she may benefit from the usage data already collected, but it is questionable whether the relatively large effort is worth it. Despite all these doubts, we plan to test such system in the future.

3.3.2 Nodes with high degree

In order to create only reasonable drawings of the displayed graph, we cannot allow merged nodes to have arbitrary height. The maximal reasonable number of edges displayed in a merged node seems to be somewhere between 20 and 40. But the actual degree of a node can go to thousands or in extreme cases even millions (one of our test data sets contains a node with around 1.3 million neighbors, although it was a rather pathological situation).

As we have already mentioned, while the number of incident edges (i.e. statements that contain the value represented by the node in question) can be very large, the total number of different predicates is usually small. If this is the case, we can display only one row for each predicate either listing the predicate and total number of edges or predicate, total number of edges and one example. The choice depends on the data layer – whether it is capable of providing just the number in a much more efficient way (e.g., a variant of SQL’s `select count(*) ...group by ...`) than providing the number and the example (e.g., by listing all nodes). To help the user backtrack his or her way through the graph, the edge connecting a node to its parent (the node it was reached from) is always displayed as a separate line. This is also done with the edges that connect the node to any other node that is currently displayed, providing that the number of such edges is small.

There is one typical situation where the predicate based “compression” would not work. The RDF recommendation provides vocabulary for containers and members of a container are specified by sequentially numbered predicates. This usually means that there are as many different predicates as there are outgoing edges. Fortunately, this situation is easily distinguishable and can be solved by displaying only some elements stored in the container. The actual of displayed elements depends on the count of other incident edges – such edges have precedence over the elements in the container and they are handled as if there were no elements only with slightly lower limit for the number of rows – we always want to display at least a few elements stored in the container.

If even the reduced number of rows that should be displayed in the merged node is too high, we display only the most important (with highest cardinal-
ity) rows and use the last row to inform the user about the total number of edges and predicates that had to be omitted.

It is obvious that we need to provide the user with some way of accessing even the edges that have not been displayed in the merged node. Such situation is always represented graphically – either the number of edges with the same predicate is displayed or the rows-omitted notice is present. In such case the user can display a special dialog that can be used to search the complete list of incident edges. The user can specify a substring or regular expression (this once again depends on the capabilities of the underlying data layer – some RDF stores can handle regular expressions, some can not) for the value of subject, predicate or object and all edges that satisfy the condition are displayed.

Note that it does not make sense to specify both subject and object since one of them must be equal to the node whose neighbors we are exploring. The predicate can be selected from the list of all relevant predicates since – as we have already stated – their number is small in most cases.

The user can select any node displayed this way and expand the current view of the graph by addition of that node.

3.4 Big fat graphs

As we have already stated, we cannot display the whole data as their amount cannot even be processed in reasonable time, let alone display it within reasonable space. Of course there are visualizations, where this can be done (for example visualizations of web page relations or large social networks), yet their purpose is to only suggest the overall structure of the data, not display individual items.

Although our visualization – the triangle layout algorithm – produces asymptotically optimal area [28], the layout alone can not overcome the problem raised by the large size of the graph. Some navigation in the graph is necessary. There are three basic ways for navigation in large graphs [40]:

**Zoom** Zoom is traditional tool in visualization. It is well suited for graphs, since in most cases the zoom can be made easily by scaling the image. However, for the zoom we have to draw the whole image (whole RDF graph) first. This is often impossible since the size of the data may exceed the available memory or the load time may be way too long.

**Focus + context** Another well known problem with zooming is the loss of the context. This can be partially overcome by displaying a map of the whole

34
Figure 3.6: Additional navigation

In our RDF visualizer we use the incremental exploration, where the user navigates the data by adding connected nodes to the already visible part of the graph. However, even by the navigation the graph may grow more than we can display. Therefore we need additional navigation in the displayed view. For this purpose either a map (see Figures 3.6(a) and 3.6(b)) or the
context or a cartesian fisheye view (see Figure 3.6(c)) may be used. Since the current frame (displayed part of the whole data) is smaller than the data by several orders of magnitude, we can use the techniques without worrying about required memory and computational time.

3.5 Multiple views

One of the significant advances in web browsers was the addition of tabs, i.e. the ability to display more pages within one browser instance and quickly switch between them. This feature has quickly become very popular. Some people use even tens of tabs while they surf the Web, requiring additional improvements to the tabs support in browsers or third party plug-ins to better organize the tabs.

Two common usage scenarios are browsing pages like Wikipedia or e-shopping. In the first case many users tend to read one page and when they encounter an interesting link they open it as a new tab but continue reading the original page. The opened tabs contain topics the user intends to read in the future. When shopping, one common practice is to browse the categories in the shop and open pages for items that appear interesting at a first glance in new tabs. After browsing the categories, the user explores the selected items in greater detail and closes the tabs with items he or she is no longer interested in.

These are similar to some of the ways an RDF visualizer can be used. The user is either exploring the data and he or she may come to a point where two or more promising directions of further navigation are at hand. Or the user may be “shopping” for interesting data, i.e. searching for data items that are worth further analysis or processing.

Addition of tabs to an RDF visualizer is very straightforward – all possible drawings (the basic view, node neighbor explorer, special views like reification view [33] or neighbor view, ...) can be displayed as a tab. At any point where such view can be opened, it can be opened as a new tab. Furthermore, any node in any view can be used to open new tab with the visualization containing only the selected node because it is displayed as a merged node so the user can navigate (explore) the neighborhood of the node.

3.6 Related work

Since RDF data have been around for quite some time, there are already several tools that try to visualize it. Many of them display the whole graph,
which is not suitable for large data because it requires too much resources and the resulting view is not clear. The visual navigation is impossible in this case. However, there are still some other tools that also allow the visual navigation in the RDF data. The three most relevant to our case are the following:

**Node-centric RDF Graph Visualization**  The RDF visualization software by the HP company [56] is one of the few tools that do not try to display the RDF graph precisely. According to the user’s choice, it always displays tree of node’s ancestors and descendants. If any of them can be reached by more than one path (which would create a non-tree edge in our solution) the node is displayed multiple times (once for each path) which preserves the tree structure of descendants and ancestors. One disadvantage of this tool is that it only displays two levels of ancestors and descendants and does not try to handle nodes with high degree.

**Paged Graph Visualization (PGV)**  The PGV [21] is similar to our tool because it does not try to display the whole graph. Incremental algorithm is used to layout the explored sub-graphs. However, the expansion of the view is the only navigation operation in the PGV explorer. Moreover, the expansion extends the view by all neighbors of selected node. It can be problem mainly with nodes that have high degree. Even though authors claim that the Ferris-Wheel technique handles high-degree nodes (but only with nodes having at most hundreds of neighbors) the expansion of all neighbors of such node is too space consuming, especially if no reduction of the view is allowed.

**IsaViz**  IsaViz [51] is a visual environment for browsing and authoring RDF models, represented as directed graphs. The graph is visualized once and can be explored using cameras that can be moved and zoomed. The user can not reduce or modify the view and can only browse the whole graph. If the graph is too big even the zooming may not lead to easily readable view.

Despite significant diversity of available visualization tools for RDF data, only a few of them can be used to visualize large data. None of the tools uses technique similar to our node merging and all of them run into trouble when the data contain nodes with very high degree although such nodes can commonly be found in the real world data. In many of the systems, the navigation in the graph is not supported or is limited to browsing through the whole displayed graph.

Some of the approaches described in the Section 3.3 are dependent on the node merging technique, but most of them can either be used directly
in other visualizers, adapted for different drawings or the visualizers can be extended to support node merging (this would be possible in many of them). The starting node selection problem is common for all visualizers. Since most of the visualizers always display all neighbors of a node, the problem of picking the rows to display in a merged node do not apply to them directly.

But in order to display nodes with extremely high degree, some of them would have to adopt a similar approach since most drawings of the RDF graph cannot reasonably handle nodes with a million or more neighbors.

The node neighbor explorer could be used in any visualizer as well as tabbed browsing. Most of the visualizers already have their ways to handle large graphs.
Chapter 4

TriQuery

The ability to extract information (make queries) is very important for any data format. The RDF is a low-level data format, comparable to relational databases. This means that the queries will be written by software developers that are building a semantic-enabled application. The SQL provides developers with tools to access relational databases in an efficient, yet simple and well defined way. SPARQL – the most notable RDF query language – is not very powerful, yet from the point of view of an SQL developer, it has a fairly complex and often counter-intuitive syntax and formal model. This creates an opportunity for improvement and exploration of new directions in the area of RDF querying. We decided to take the more exploratory approach.

We have developed a new algebra for RDF querying, based on some of the fundamental ideas of relational algebra and RDF. The definition is given in the Appendix B. But it does not give any query language built on top of the algebra. Even though developing a new language would have been an interesting undertaking, in the end we decided not to go that way. There were two main reasons; first, there are already many different query languages for RDF [53, 16, 47, 37]. Second, the alternative solution we have created has opened up new interesting areas.

The alternative was to extend an existing, well established language with RDF support. In the end, we chose the XQuery language [13] for several reasons: even though it is a compact and well defined language, it is powerful (Turing complete [48]), it is also a popular language with both software developers and scientific community. But the most important reason is the fact, that it allows us to work with RDF and XML within one language. For instance, the RDF data can be transformed into an XML document before they are returned from the query, thus leaving the choice of the output format to the user.
Extending the XQuery language with RDF querying capabilities has already been tried [52]. But we decided to take a more invasive approach — to make more substantial changes to XQuery, for instance adding new rules to the grammar of the language or adding a completely new kind of value that a variable can be bound to. The result of these changes will be called TriQuery in the rest of the text, as it still derives many ideas from the original TriQ proposal. We were able to preserve many of the interesting capabilities provided by TriQ, but were forced to abandon the TriQ data model and create a new one, better suited for integration into the existing XQuery data model.

A direct advantage of TriQuery is that it can be used to transform RDF to XML and back. Although this could be performed over RDF/XML [3] by pure XQuery, our goal is to use a native RDF database and specially suited techniques for RDF query handling, thus achieving significantly better performance.

An obvious drawback is that TriQuery cannot easily be implemented on top of an existing XQuery processor. Currently, we are building an XQuery processor for the Bobox architecture (the architecture is described in detail in the Chapter 5; the XQuery processor is a separate work [4]) and since the architecture requires most of the processor to be written from scratch, we can modify it to handle TriQuery as well.

4.1 XQuery essentials

Let us first briefly recollect some basic principles of XQuery. The basic building block of XQuery is an expression, which may be constructed from keywords, symbols, and operands. XQuery allows expressions to be nested with full generality. An expression evaluates to an ordered sequence of items. An item is either an atomic value (string literal, numeric literal, etc.) or a node reference — a reference to a node in an XML tree.

An important part of the language are the XPath operators [20, 9]. In fact, any valid XPath 2.0 expression is also a valid XQuery expression. This allows the user to navigate XML trees. Given a context within a XML tree, the following example selects the fifth para child of the context node that have a type attribute with value warning:

para[@type="warning"] [5]

Besides that, XQuery includes the FLWOR expression which is an acronym that stands for FOR, LET, WHERE, ORDER BY, RETURN. FOR and LET is used to generate a sequence of bound variables, WHERE filters the sequence, ORDER BY reorders it, and the RETURN creates the result from the sequence of bound
variables. The `RETURN` is a mandatory (last) part, while `WHERE` and `ORDER BY` are optional. The expression may start with any number of `FOR` and `LET` expressions arbitrarily interleaved. The difference between `FOR` and `LET` is that `FOR` iterates the sequence (the second argument of the expression) while `LET` stores the whole sequence into the variable. Consider the following example:

```xml
for $d in fn:doc("depts.xml")/depts/deptno
let $e := fn:doc("emps.xml")/emps/emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return $d
```

The `$d` variable iterates the department numbers and for each value, the collection of employees of that department is stored in `$e`. This means that when the `WHERE` clause is evaluated, `$e` is the sequence and it may test the number of elements in the sequence. The same holds for the `ORDER BY` clause, so the sequence of variable bindings (the sequence of `$d$s in this case) may be sorted by the average salary. The `$e/salary` expression evaluates to a sequence of `salary` children of members of `$e`. Then, the `fn:avg` function computes the average of that sequence.

The last example used pre-define functions to access XML files (`fn:doc`) and process sequences (`fn:count` and `fn:avg`). Besides several built-in functions, the XQuery also supports user-defined functions. For example, the following function adds one to each element of the sequence passed as the argument:

```xml
declare function local:add-one($vals as xs:integer*)
    as xs:integer*
{
    for $val in $vals
    return $val + 1
};
```

The type of the argument is checked at runtime. The functions may freely use recursion, although their support is limited in some implementations.

Another important part of the language are the constructors. They allow the user to create new XML fragments. For instance, the following example the following function converts a sequence of numbers into a sequence of XML nodes:

```xml
declare function local:add-one($vals as xs:integer*)
    as element(value) *
```
{  
  for $val$ in $vals$
  return <value>$val</value>
};

The language also includes the usual mathematical operations, comparison operators and logical operators.

4.2 Records in XQuery

We extend the XQuery language to handle records – a structured piece of data in the form “identifier - value”. Each record has a set of identifiers (field names) and a value corresponds to each of these identifiers (field value). The set of identifiers either consists of qualified names to form a named record or a sequence of numbers 1..n to form an anonymous record. The field values are sequences of items.

Sequences of records can be used in TriQuery as values of expressions. This of course requires us to extend most of the XQuery constructs to handle such sequences. These definitions are presented in the following section, along with examples of their use.

The following text references several concepts defined by the XQuery or related standards:
- (basic) item – an atomic value or a node reference,
- $Val$ – the set of all items and sequences of such items [36],
- QName – the set of all qualified names [14].

4.2.1 Definitions and examples

Our first extension of XQuery is the addition of records to create an extended item. The extended items are used instead of the basic items in all situations, except for the situations where we explicitly specify otherwise. This is a significant modification of the language and its semantics, but it greatly simplifies the inclusion of RDF querying into XQuery and it also creates new possibilities even when RDF is not used.

(anonymous/named) record, signature A record is a partial function from QName ∪ N to Val. Note that this does not allow records as field values (nested records). There are three limitations on the domain $D$ of the function:

- $D$ is finite
• either $D \subset QName$ (named record) or $D \subset \mathbb{N}$ (anonymous record)

• if $D \subset \mathbb{N}$ then $D = \{1..n\}$.

An extended item is a basic item or a record. If any sequence contains a record, we require the sequence to contain nothing but records and also that all of the records have the same domain $D$. This domain is called the signature of the sequence. TriQuery enforces the rule that during statical analysis of the query, it has to be possible to determine whether any expression may return a sequence of records and if it does, identify the signature of the sequence. All definitions are created in a way that allows such statical analysis.

**record constructor**  A record can be created by a record constructor. It is a language construct similar to the XML constructors already present in XQuery. There are two versions – for anonymous and named records.

**anonymous record constructor**  \([8, \text{"Hello, world!"}, x]\) creates an anonymous record where 1 is mapped to the integer 8, 2 to the string "Hello, world!" and 3 to the value of the variable $x$. The individual expressions may be exactly the same as in the case of function call, except that they may not contain records – this is checked statically.

**named record constructor**  \([\text{eight := 8, hello := "Hello, world!"}, \text{var := $x$}]\) creates a record where the qualified name “eight” is mapped to the integer 8, etc. One field name can be used at most once. The constraint on the expression is the same as with anonymous records.

**concatenation of sequences**  The sequence concatenation – the comma operator – is extended naturally to handle records. However, if any of the sequences contains records, then both sequences must have the same signature. According to this definition, the following query is illegal:

```latex
let $x := [a := 1, b := 2 ]$
let $y := [a := 1, c := 3 ]$
return ($x, y$)
```

The expression ($x, y$) violates the rule, that it must be statically verifiable that $x$ and $y$ have the same signature – $\{a, b\} \neq \{a, c\}$. On the other hand, this is valid since $\{a, b\} = \{b, a\}$:
let $x:=[a:=1, b:=2]$  
let $y:=[b:=1, a:=3]$  
return ($x$, $y$)

Also, the following query is invalid:

let $x:=[1]$  
let $y:=2$  
return ($x$, $y$)

The concatenation ($x$, $y$) is not valid since the value of $x$ is an anonymous record and $y$ is not a record at all.

**asterisk operator in constructors** In anonymous record constructor, an item may be replaced by $x.*$ which is a short for $x.1, \ldots, x.n$, where $n$ is the number of fields in $x$.

The named record constructor allows the form $*:=$ $x.*$ which is expanded to $x.f_1, \ldots, x.f_n$ where $f_i$ are the fields of the record in $x$. The requirement, that one field name cannot be used more than once, must still be maintained, i.e., the expanded form must fulfill the requirement.

In both cases, the $x$ is used only to shorten the definition; it may be any expression that evaluates to a record. This is only a “syntactic sugar”, but it can significantly shorten the textual representation of some queries (see the Section 4.5, query Q2 for an example).

**dot – field access operator** Fields of a record can be accessed using the dot operator in a way that is similar to that of object oriented programming languages.

**single record field access** If $r$ is a record and $x$ a member of QName $\cup \mathbb{N}$ such that $x \in \text{dom}(r)$, then $r.x$ evaluates to $r(x)$. For example $a.2$ evaluates to $r(2)$ if the value of $a$ is the anonymous record $r$. The situation is analogous for named records – $a.hello$ evaluates to $f(hello)$.

**record sequence field access** If $A$ is a sequence $(a_1, \ldots, a_n)$ such that $\forall i \leq n : \text{dom}(a_i) = D$ (i.e., $D$ is the signature of the sequence) and $x$ identifies a field (i.e., $x \in D$), then $A.x$ evaluates to a sequence $(a_1(x), \ldots, a_n(x))$. Note that $a_i(x)$ may be a sequence in which case it is concatenated with other values to form one “flat” sequence.

The dot operator can be used to correct the example that demonstrated incompatibility of records and atomic values. The following query is valid:
let $x := [1]$
let $y := 2$
return ($x.1, y$)

**record equality** The definition of the operator $=$ is extended to handle records in the following way: if $r$ and $s$ are records and it can be statically determined that $\text{dom}(r) = \text{dom}(s)$ then

$$r = s \iff \forall f \in \text{dom}(r) : r(f) = s(f)$$

If the $\text{dom}(r) = \text{dom}(s)$ constraint is not met, the query is considered invalid. Note that since nested records are not allowed, $r(f) = s(f)$ never test records, thus the original XQuery operator is used.

**equality of sequences** To be consistent with the way XQuery handles sequence equality, we have to use an existential quantifier in the definition - two sequences are equal, if the first one contains an item equal to an item contained in the other sequence.

If $A$ and $B$ are sequences of records $(a_1, \ldots, a_u)$ and $(b_1, \ldots, b_v)$ and we can statically verify that signatures of $A$ and $B$ are the same, then

$$A = B \iff \exists (i \leq u, j \leq v) : a_i = b_j$$

If the constraint on signatures is not met, the query is considered invalid. For example, the following query is invalid:

let $x := [a := 1, b := 2]$
let $y := [a := 1, c := 3]$
where $x = y$
return $x$

**record inequality** We do not allow the operator $!=$ to be used on records, since the extension of the original operator from XQuery would be very unnatural and counterintuitive.

**node comparators** The operators $<<$ and $>>$ are not allowed for records, the operator is is extended for record sequences $A$ and $B$ with the following conditions:

- length of $A$ and $B$ is at most one
- $A$ and $B$ have the same signature
• all fields of all records in A and B contain sequences with length at most one

If either A or B is empty sequence or any field of any record in A or B is an empty sequence, then the result is an empty sequence. Otherwise, A contains one record a and B contains only one record b. Then the result is true iff $\forall f \in \text{dom}(a) : a(f) = b(f)$. Otherwise, the result is false.

value comparison The definition of eq and ne is the same as the definition of is, except that the condition for eq is $\forall f \in \text{dom}(a) : a(f) \equiv b(f)$. For ne, the condition is $\exists f \in \text{dom}(a) : a(f) \neq b(f)$.

union Union is not supported for sequences of records.

FLWOR The FLWOR operators are naturally extended to handle sequences of records.

Let us give a few examples where FLOWR is used to process records. The signature \{a, b\} of the sequence bound to $\$x$ can be modified to \{x, y\} like this:

```latex
for $r$ in $x$
return [ x := r.a, y := r.b ]
```

Even an anonymous record may be created:

```latex
for $r$ in $x$
return [ $r$.a, $r$.b ]
```

The following example demonstrates how a sequence of records can be created from XML data:

```latex
let $x=$(for $n$ in fn:doc("ex.xml")//item
return [ number := $n/number, name := $n/name ])
```

The reverse would be:

```latex
for $r$ in $x$ return
<item><number>{$r.number}</number><name>{$r.name}</name></item>
```

46
**type definition**  The construction used by XQuery to define types are extended to handle anonymous and named records. Both cases are illustrated by the following example:

```xquery
declare function convert ( $x as [ a as xs:string, b as xs:datetime ] *,
$y as xs:string )
returns [ xs:string, xs:string ] *
{
    for $u in $x
    return [ $u.a, $y ]
}
```

The parameter $x$ contains a sequence of named records with fields a and b and the function returns a sequence of anonymous records with two fields.

### 4.2.2 Pattern matching operation

An important operation on records is a selection of a specific pattern from a large collection of records. The user specifies a set of records where some values are replaced by variables. The system then tries to find all possible substitutions of the variables that transform the pattern to a subset of the input data. In other words, it tries to find all instances of such pattern in the input data. The result of the operation are the applicable variable bindings.

Although this pattern matching could be handled by the FLWOR operator, we decided to include a new operation that simplifies the usage of such selection and also adds new functionality. The complete syntax of the operator is given in the Appendix C. The basic form of the operator is either $x$ match { **pattern** } or $x$ with **uri** match { **pattern** }. The **pattern** is a sequence of tuples that may contain variables. Note that the variables may be written as either $var$ or ?$var$ for better “visual compatibility” with other languages.

Let $x$ be a variable that contains a record sequence with signature {1, 2, 3}. Then an example of pattern matching operation may look like this:

```xquery
$x$ match { "John" {1+2} ?a . ?a ?b ?c }
```

This expression searches the sequence of records (the value of $x$) for records that look like [ "John", 3, a] and for each such record tries to find a record [ a, b, c] where a is the same value in both cases, while b and c may be any value. Then, it returns a sequence of records [ a:= a, b:= b, c:= c ]. A more complete example could look like this:
for $r$ in $x$ match { "John" {1+2} ?a . ?a ?b ?c }
return <r><a>{$r.a}</a><b>{$r.b}</b><c>{$r.c}</c></r>

There are some constraints on the pattern matching operation. Let us assume it is in the form of $D$ match \{ $P$ \} or $D$ with $T$ match \{ $P$ \}. Then the following must conditions must be met:

- The expression $D$ must evaluate to a sequence of anonymous records.
- All patterns in $P$ must have the number of elements equal to the cardinality of the signature of $D$.
- Each XQuery expression used in any pattern must return a sequence of basic items with at most one item.
- Acceptable values of $T$ are implementation-defined.

Note that the first two conditions (and fourth) can be verified statically, while the third can generally only be tested at run-time. The operation then returns a sequence of named records, whose signature is equal to the set of all variable names used in $P$.

In general, if the $T$ argument is present, the system may perform any operation on the input data and pattern and produce any result, as long as it produces a sequence of records with the right signature. However, in situations where $T$ is missing or if the default behavior is sufficient, the following “basic” definition is used.

**Basic definition** If $T$ is specified, it must identify a transformation function that transforms a sequence of records into another sequence of records with the same signature.

Let $n$ be the cardinality of the signature $\text{sig}$ of the input data $D$ ($D$ is the sequence of records that $D$ evaluates to). We know that $\text{sig} = \{1..n\}$. Let $P = \{(p_1^1,\ldots,p_n^1),\ldots,(p_1^n,\ldots,p_n^n)\}$ be the patterns in $P$, where all XQuery expressions have been evaluated and replaced with their values. Let $T$ be the function defined by $T$ or an identity function if $T$ is not used.

Let $m$ be the number of distinct variable names $V = \{v_1,\ldots,v_m\}$ used in $P$. Then the result $R$ of the operation is a sequence of records $(r_1,\ldots,r_p)$ with the signature $V$.

First, the input data $D$ is flattened. Flattening replaces each record with one or more records. A record $d$ with values $(d^1,\ldots,d^n)$ is replaced by a set of records $\text{Flat}_d = \{d_1,\ldots,d_n\}$ where $\text{Flat}_d$ is the maximal set of records $d_i$ such that $\forall 1 \leq i \leq x : \forall f \in \text{dom}(d_i) : d_i(f) \in d(f) \lor (d_i(f) = d(f) = \text{empty})$
(empty denotes an empty sequence). Note that we assume \(d(f)\) to always be a sequence of items and \(d_i(f)\) to be one item or an empty sequence. The set \(Flat_d\) is then transformed into a sequence \(Flat'_d\) with implementation defined order. The result is a list of records \(D_f\) created by concatenation of all individual \(Flat'_d\) for each \(d_i\) in \(D\). Note that \(D_F\) has the same signature as \(D\).

Variable mapping is a partial function \(\mu\) from \(V\) to \(Item\) where \(Item\) is the set of all basic items. \(\hat{\mu}\) is derived from \(\mu\) to handle whole patterns. First, let \(\mu' = \mu \cup Ident_{Item}\), where \(Ident_{Item}\) is an identity function on basic items. For a pattern \((p_1, \ldots, p_n)\), we define \(\hat{\mu}(p_i) = \{1 \to \mu'(p_1^i), \ldots, n \to \mu'(p_n^i)\}\). Note that although \(dom(\mu) = Item\), the domain of \(\hat{\mu}\) are anonymous records. The function \(\hat{\mu}\) is further extended to handle set of patterns by applying the function to each member of the set.

Let \(D_T = T(D_F)\). The result set \(R_{set}\) is the smallest set that contains \(\hat{\mu}(P)\) for all possible \(\mu\) such that \(\hat{\mu}(P) \subseteq D_T\). Note that this definition requires the values to be exactly the same, unlike the existential definition of equivalence in XQuery, however this is not a problem since the data is flattened and does not contain sequences. The result \(R\) of the operation is a sequence that contains the members of \(R_{set}\) in an implementation defined order.

## 4.3 RDF support

With records, inclusion of RDF querying is much easier. The whole RDF dataset can be represented as a sequence of anonymous records with three fields. The pattern matching operation can be used in a way that is similar to a basic graph pattern in SPARQL. The optional with modifier can be used to specify entailment regimes, for instance RDFS entailment. Since we have a constructor for new anonymous records, we can also create new RDF datasets.

A direct advantage of TriQuery is that it can perform transformations between XML and RDF - data from records (RDF triples) can be extracted using the dot operator and used in XML constructors. And, of course, the other direction can be done using record constructors.

The usual scenario is that the data is extracted from an RDF database:

```
let $c=triq:doc("data1")
```

Then, pattern matching is performed as the very first step:

```
for $m in $c with triq:RDFS match
 ( $x ex:first-name $fn . $x ex:last-name $ln )
```
After that, the rest of the query is written in a manner similar to the original XQuery, except that records can still be used not only to represent RDF triples and results of pattern matching. A more complete example is a simple extraction of facts about people into XML:

```xquery
let $d := triq:doc("people.rdf")
for $x in $d match ( $x ex:id $id . $x ex:name $n . $x ex:mail $m )
return <person id="{$x.id}"<name>{$x.name}</name>
<email>{$x.mail}</email></person>
```

However, there is one other issue that has to be addressed in order to fully support RDF. The support of blank nodes. We need ways to represent existing values and create new ones. The latter can easily be achieved by addition of a `triq:bnode()` function that returns a new blank node. There are two ways in which the representation issue can be handled. First, the XQuery type system could be extended by a new type, BNode, that supports equality and inequality testing and raises an exception in other situations. Another solution is to reserve a namespace for blank nodes and store them as URIs. The first option is cleaner and as such more preferable, but the other option should work just as well.

We may also create new RDF data sets, for instance by extracting information from an XML document. Consider the following example:

```xquery
let $d := fn:doc("in.xml")
for $x in $d//item
return [ [ $x/id, ex:name, $x/name], [ $x/id, ex:price, $x/price]
```

This creates a sequence of anonymous records with three fields — an RDF data set. This way, the language could be used to transform XML into RDF. Of course, we could also transform RDF into RDF:

```xquery
let $d := triq:doc("people.rdf")
for $x in $d match ( $person ex:age $age )
where $x.age>13 and $x.age<=19
return [ [ $x/person, ex:age-group, ex:teenager ]
```

The with modifier of the `match` operator can be used for advanced entailment modes, for instance in the form `$d with triq:rdfs match (...)`. 

50
4.4 Relational data support

An interesting consequence of the addition of records to XQuery is the ability to easily extend it to handle data stored in relational databases. All we need is a function, e.g. rel:table, that takes the name of the table and returns a sequence of named records that correspond to the rows of that table. For instance, consider tables person(id,name) and contact(id,mail). Then, we can use the following query:

```xquery
for $p in rel:table("person")
for $m in rel:table("contact")
where $p.id=$m.id
return <contact name="{$p.name">{$m.mail}</contact>
```

Alternatively, we could use even the following version:

```xquery
for $p in rel:table("person")
let $m1 :=
  (for $m in rel:table("person") return [ $m.id, $m.mail ] )
for $m2 in $m1 match ( {$p.id} $mail )
return <contact name="{$p.name">{$m2.mail}</contact>
```

The second version is, of course, needlessly complicated in this situation, but it demonstrates the possibilities of this combination of relational data and XQuery processing with records.

This extension is easy to implement, providing there is a suitable datastore for the relational data. The rest could be either directly reused or adopted from TriQuery. The whole system could then be used to transform data in any direction between relational databases, XML and RDF.

4.5 SP²Bench examples

This section gives TriQuery variants of some of the SPARQL queries from the SP²Bench benchmark [57], which is becoming one of the most popular benchmarks for SPARQL engines. It provides a data generator that generates artificial DBLP-like data (database of scientific publications) and a set of queries aimed at testing some of the typical patterns that may be present in SPARQL queries. We decided to present some of the queries in their original form and the corresponding TriQuery equivalents, since the queries have easy-to-see meaning and real-life interpretation.
Q1: A very basic query with straightforward translation to TriQuery.

```
SELECT ?yr
WHERE {
  ?journal dcterms:issued ?yr
}

for $x in triq:doc("dblp") match {
  $journal rdf:type bench:Journal .
  $journal dc:title "Journal 1 (1940)" .
  $journal dcterms:issued $yr
  return $yr
}
```

Q2: A query with a large pattern and an optional part.

```
WHERE {
  ?inproc rdfs:seeAlso ?ee .
  ?inproc dcterms:issued ?yr
  OPTIONAL {
    ?inproc bench:abstract ?abstract
  }
}
```

ORDER BY ?yr

```
for $x in triq:doc("dblp") match {
  $inproc rdf:type bench:Inproceedings.
  $inproc dc:creator $author.
  $inproc bench:booktitle $booktitle.
  $inproc dc:title $title.
  $inproc dcterms:partOf $proc.
```
$inproc rdfs:seeAlso $ee.
$inproc swrc:pages $page.
$inproc foaf:homepage $url.
$inproc dcterms:issued $yr )
let $y := triq:doc("dblp") match
( { $x.inproc } bench:abstract $abstract )
order $x.yr
return if ($y) then
  (}
  for $a in $y return [ * := $x.*, abstract := $a ]
)
else
[ *:= $x.*, abstract := () ]

An OPTIONAL subquery in SPARQL is translated into for-let-if combination. The asterisk operator is used to shorten the textual representation of the query.

**Q9:** A query with union and distinct operators.

**SELECT DISTINCT ?predicate**
**WHERE {**
    ?subject ?predicate ?person
  } UNION {
    ?person ?predicate ?object
  }
}

return fn:distinct(
let $x := triq:doc("dblp") match ( $person rdf:type foaf:Person.
  $subject $predicate $person )
let $y := triq:doc("dblp") match ( $person rdf:type foaf:Person.
  $person $predicate $object )
return $x.predicate, $y.predicate
)
4.6 Relation between TriQuery and XQuery

Since XQuery is already Turing-complete, the TriQuery extensions certainly do not provide more power from the theoretical point of view. Furthermore, the original intention is that the fields of the records are used to store atomic values, rather than node references. If this is the case, the records have a very straightforward mapping to XML fragments.

For example, a named record $[a:=1, b:=2]$ can be represented by a XML fragment $<\textit{r}<a>1</a><b>2</b></r>$. Then $\textit{fx}/a$ would be used instead of $\textit{fx}\.a$. There are other differences, for instance the equivalence testing would have to be handled by a function, since the behavior of the equivalence operator is different. But overall the mapping is straightforward. The XML fragment representation even allows the user to mix atomic values and records with different signature within one sequence.

If node references are used as field values, the mapping cannot be done this way, since it is not possible to create references between XML tree fragments. But XQuery is powerful enough to handle even such queries. One possible transformation of sequence of records is to transform each records into a sequence and then concatenate those sequences. A record could be transformed into a sequence whose first member is a XML fragment that gives the number of values (the length of the sequence) for each field in the record. The rest of the sequence is created by concatenating the values of the fields. For instance, the record $[a:=1, b:=(), c:=(2,3)]$ can be translated to $<\textit{r}<a>1</a><b>0</b><c>2</c></r>, 1, 2, 3. Since most XQuery processors use some kind of relations for internal representation of intermediate results, it would most likely be easier to represent records by adding more columns to the tables rather than using such sequences.

The pattern matching operation could be rewritten as several nested FLWOR operators, although one has to be careful, that in order to exactly replicate the behavior of the operation, the input has to be flattened before searching the data for pattern instances. Again, the main difference of match operator is that the format of the input data is known in advance. Other than that, an advanced optimizer may be able to create just as good execution plan for the FLWOR operator as for the match operator.

There are three reasons for introducing records into XQuery, none of which is making it “stronger”. First, the record-based syntax allows us to handle RDF and relational data in a more convenient way – it is easier for the user to write such queries. Second, the requirement that each sequence of records must have a (statically identified) signature is very important for the way such queries are evaluated. Third, the signatures allow us to perform some level of static query checking. For instance, we are able to

54
verify, whether the expression $x.name$ is valid – we can determine that the variable $x$ does contain a sequence of named records and that the records have a field named name.

### 4.7 Implementation concerns

As we have already stated in the introduction, the modification of the XQuery is so extensive that a completely new implementation may be required. However, the limitations placed of the usage of records greatly simplify it. Especially the requirement that each sequence that contains records contains only records and that all of them have the same layout – the requirement for the existence of a signature.

Since this has to be statically identifiable, the evaluation engine may use a less general but more efficient representation when handling records compared to a situation where the records are transformed into XML fragments. A sequence of node references (XML fragments) is often represented as an array of references to memory where the fragments are stored. A corresponding sequence of records may be stored as a table, although the fact, that a field may contain a sequence still makes it more complicated than tables in SQL.

But, in many cases it may even be possible to determine whether or not a certain field of a record may contain more than one value (be a sequence). If no field may contain a sequence, the whole data stream is in fact a table, where columns correspond to fields and rows to records. Such analysis may be performed statically during the translation of the query.

The TriQuery queries that correspond to SPARQL queries contain the pattern for `triq:doc(...) match (...). This situation is interesting for RDF physical stores – it may be evaluated efficiently using indexes. On the other hand, TriQuery allows more general usage of the match operation, even on data that were computed by other parts of the query. In such cases, the indexes are not available which is likely to negatively impact evaluation performance. This is a new problem, not shared by e.g. SPARQL or SeRQL, since they support pattern matching only as the first step of the query, where it is evaluated over the underlying data-store. But it only ensues in cases, where the user explicitly specifies it in the query and the performance implications would be obvious to the (advanced) users.

If only records are used, the query evaluation is very similar to the TriQ algebra defined in the Appendix B. Although TriQuery allows more complex transformations and the high-level interpretation of the operations is different, the principles and implementation techniques can be transferred from
TriQ to TriQuery. In many situations (which can be statically identified from the query), the techniques that were developed for the implementation of TriQ can be directly used for TriQuery.

4.8 Related work

Today, the best known RDF query language is SPARQL [53]. It is a W3C recommendation (since 2008) and a new revision is being developed at this moment. Another important language is SeRQL [16] and although it is being suppressed by SPARQL, it is still interesting as the original query language used in the Sesame RDF server [17], which has played an important role in the Semantic Web research.

Both SeRQL and SPARQL are built as completely separate query systems, unlike the XQuery based TriQuery. Their syntax somewhat resembles SQL; more so in the case of SeRQL. Also, both of them use graph patterns to transform RDF data to a structure very close to tables and then perform operations on these "tables". Again, SeRQL is closer to SQL in this aspect, while SPARQL uses its own algebra to work on multisets of variable mappings, but it still closely resembles relational algebra, even though the exact definitions differ.

Since XQuery is Turing complete [48], a formal comparison with the other (not Turing complete [50]) languages is not very informative. A very specific way in which some of the SPARQL operations are defined (for instance, the behavior of the OPTIONAL pattern in border cases) makes creating exact counterparts in TriQuery difficult. And since it is not really useful for practical purposes, we decided not to go that way. Instead, we present the individual features and compare them from the user’s point of view.

Graph patterns All three languages provide very similar RDF pattern matching capabilities, with different levels of syntactic-sugar. SeRQL provides the most, while TriQuery provides none. But the underlying operation defined by the different syntax is almost identical in all cases. It always returns a set of all the ways the pattern can be matched to the queried data. No advanced graph operations are available [1].

Optional patterns SeRQL and SPARQL both support optional matching of parts of the pattern. In SeRQL parts of the pattern may be marked as optional, SPARQL uses the OPTIONAL keyword to provide operation similar to outer join. In TriQuery, optional matching is performed using two sepa-
rate pattern matching operations, sequences and the if statement (see the example in the Section 4.5).

**Selection**  All three languages support some form of WHERE clause, the main difference is the set of functions available in each of the languages. In all languages, the clause can be used even on a part of the query.

**Union**  SeRQL provides UNION and UNION ALL operators with the same meaning as in SQL. SPARQL provides UNION that does not eliminate duplicates. In TriQuery, sequence concatenation may be used on compatible (with the same signature) sequences to perform union without duplicate elimination. However, incompatible sequences have to be first made compatible – this is done implicitly in SeRQL and SPARQL.

**Intersection, minus**  Only SeRQL explicitly provides the INTERSECT and MINUS operators.

**Query nesting**  In SPARQL, queries cannot be nested. SERQL provides IN, ANY, ALL, and EXISTS operators very similar to SQL. In TriQuery, queries can be nested arbitrarily either directly or using functions. It is even possible to construct a new set of RDF statements and perform pattern matching on this set within one query, although the performance of such query is very likely to be poor.

**LIMIT and OFFSET**  SPARQL and SeRQL directly support query modifiers that force it to return only an interval (specified by position) of the result. The same result can be achieved in TriQuery using filters on the implicit numbering of sequences.

**Ordering**  SeRQL does not allow the user to specify ordering of the results. SPARQL and TriQuery both have ordering clauses.

**Multiple data sources**  SPARQL provides the NAMED clause that can be used to query several different data sources in one query. Similar behavior can be achieved in TriQuery by varying the identifier handed to the triq:doc function. SeRQL does not have any similar feature.
"Construct" queries  Both SPARQL and SeRQL have a special form of query that can create new RDF statements. In both cases, it is a kind of a modifier on the results of the query, i.e., it can only be performed as the last step of the query and no further operations can be performed on this data. TriQuery permits the use of constructors at any place, however, as we have already stated, there are performance considerations.

Aggregates  The SeRQL does not support aggregate functions. The same is true for SPARQL, but the inclusion of aggregate functions is planned for the next version. In TriQuery, aggregations can be performed (to some degree) using sequences and functions already provided by XQuery. The group by construct is not supported in the current version of XQuery, but it is planned for the next version.

On the whole, we believe that it is possible to use TriQuery for queries that can be written in SPARQL and SeRQL without the queries becoming significantly more complicated. There are aspects of TriQuery that are not available in the other languages, for instance the compositionality and the integration with XML.
Chapter 5

Bobox

Today, parallel processing is one of the most significant trends in designing high-performance applications. Most CPUs sold today are multi-core processors and while the performance of individual cores is increasing slowly it is possible to equip one blade server with four 8-core CPUs.

Unfortunately, design and implementation of parallel programs is a very error-prone task. It can be made much more manageable by using a more advanced parallel technology than just the threading and synchronization services provided by the operating system. Since the existing libraries provide only limited support for complex processing of data, we decided to create a new specialized library for such tasks. An advantage of such an approach is that the library can be programmed and debugged using simpler scenarios where tracking the exact behavior of the application is easier. Then, the library can be used in more complex scenarios with less trouble.

We expect that the library may often be used in an OLAP-like situations, where the users perform small amount of complex and computation-intensive operations (queries). Thus, it is not sufficient to allow parallelism on a query level. If we only allowed serial evaluation of a query to run in parallel with other queries, the performance of large queries that run alone most of the time would be poor. Since we want the parallelism to be handled by the library and not the user’s code, the computation has to be split into smaller components that can run in parallel. However, the system hides the parallelism (synchronization and scheduling) from the developers of the components. This limits the maximal parallelism that can be achieved by the system, but significantly simplifies the development of the components. This makes the system more usable for experiments with new ways of data processing, like the TriQuery language.

The following sections of the thesis describe the Bobox framework and then present the experiments we have performed using a pilot implementation
of the system.

5.1 Main design concepts

The main idea behind Bobox is to connect a set of relatively simple computational components into a (non-linear) pipeline. The data then flow through the pipeline, which controls the execution of the individual computational components. The components are executed in parallel, when they have data to process. But each computational component is guaranteed by the system not to be executed by more than one thread at the same time. Furthermore, the interface between the components and the rest of the system is created to eliminate any possible race conditions when reading inputs and writing outputs. So if the components do not communicate by other means than by sending messages over the pipeline, they need not handle problems of parallel programming at all.

5.1.1 Task level parallelism

The environment with many simple components and pipeline-based communication is very suitable for task level parallelization. In this paradigm, the program is not viewed as a process divided into several threads. Instead, it is seen as a set of small tasks [42]. A task is a piece of data together with the code that should be executed on the data. Their execution is handled by a task scheduler. The scheduler maintains a pool of tasks to be executed and a pool of execution threads and allocates the tasks to the threads. At any given time, a thread can either be executing a task or be idle. If it is idle, the task scheduler finds a suitable task in the task pool and starts the execution of the task on the idle thread.

There are several advantages to this approach. One of them may be the fact that in some cases such system may be easier to use for the software developers. But there are even more fundamental advantages. Task based parallelization may provide better utilization of system resources. First, the smaller granularity of tasks compared to threads means that the CPUs can be better utilized when there is a synchronization barrier in the execution – the finer granularity of work decreases idle time of the CPU cores. While in classical thread based parallelization a thread that finishes before the other threads has to idle (wait), in task based parallelization it can execute other tasks.

Second, a carefully designed scheduler can achieve a much better use of CPU caches. When a piece of data is transferred from one thread to another,
the data usually has to be loaded into the cache of the processor core where the receiving thread is currently running. In task parallelism, this situation is usually handled by creation of a new task with the data. This new task can be executed on the same CPU core as the task that created the data (after it finishes). This usually means that the data is still hot (stored) in the cache and the loading is avoided.

On the other hand, there is overhead associated with the work of the task scheduler, namely maintaining the task and thread pool and the execution of tasks – each task takes some time to set up before the actual code is executed and also some time to clean up after the execution. This time is proportional to the number of tasks, so its percentage in the total run-time decreases if the individual tasks take a long time to complete. But the smaller run time of tasks improves CPU utilization, since it reduces the time that the threads spend idling when there is not enough work available for all threads. So, some balance between these aspects has to be found when looking for the optimal task size.

5.2 Run-time architecture

One of the main differences between other parallelization frameworks and the Bobox architecture is the way the user’s code interacts with Bobox. OpenMP and TBB are used to invoke parts of the code in parallel; MPI provides means for communication between processes. Bobox is more similar to the first
two systems, but there are two key differences. First, it uses a declarative approach to describe the way in which elements of the computation are put together. Second, it provides more services to the user code but also imposes greater restrictions on the code.

The parallel execution environment displayed in the Figure 5.1 is somewhat similar to that of TBB, since it contains a task pool and several threads that execute tasks from that pool (the right half of the figure). However, the way in which the tasks are created and added to the pool is completely different. In TBB, this is controlled either directly by the user’s code or using a thin layer of parallel algorithms provided by the library.

In Bobox, the user first specifies a model. The model defines the way in which the individual computational components are connected. The model is then instantiated to produce a model instance. There can be multiple instances of one model. The instance then forms a basis for a request. The request contains only little information besides the model instance, mainly a unique request identifier. When the user code sends the request to the parallel execution environment, it no longer has control over the execution of the model instance and can only wait for the request to finish — the execution is fully handled by the execution environment based on the Bobox rules and the structure defined by the model.

The elements of the model instance are used as tasks. When they are ready, they are enqueued — added to the task pool. Later, a thread takes a task from the pool, perform the action (invokes the task) and then the model instance element is returned and can again be used as a new task and added to the pool. This conforms to the requirement that no box may be invoked more than once at the same time.

The Bobox system cannot be used for an arbitrary parallel computation. It is limited by the way in which the system decides what computational components should be executed. This is controlled by the flow of the data through the pipeline. The data must be passed in a way defined by the system. This way the system is aware of the fact that a component consumed or created (sent further down through the pipeline) some data.

This simplifies the design of the individual computational components. The do not have to be concerned with controlling the execution and data flow. Their code simply reads the input data, computes the results and writes them to the output.

5.2.1 Basic elements and terminology
As we have already stated, the overall computation is defined using a model which is then used to create the actual model instance used in the compu-
There are two types of elements that form the model instance. *Box* is the computational component mentioned in the previous text. It has zero or more inputs and zero or more outputs in the pipeline. *Via* represents one *link* in the pipeline. It connects one output of one box to one or more inputs of other boxes. The link only provides one-way data transfer.

*Envelope* is the smallest unit of data transferred through the pipeline. When a box sends an envelope to one of its outputs it is sent to the appropriate via. The via then makes a copy of the envelope for each box on the receiving end of the link and sends the copy to the appropriate box.

The model is an almost exact image of the structure of the instance. It is composed of *box models* and *via models* that define the types of boxes and the links to be used in the model instance. The main difference between a box and a box model is on an implementation level – the box does contain the actual code, buffers and state information required to run the computation while the box model does not. The same holds for vias and via models.

An example of a model (or model instance) can be seen in the Figure 5.2. It displays some of the basic situations like a via with more outputs, box with multiple inputs or different length paths.
5.2.2 Execution of the model instance

All model instances have one initialization box (see the Figure 5.2). This box is executed by the system at the beginning of the evaluation and its only purpose is to send one envelope to its only output. All other boxes are executed “when needed” depending on the data flow in the pipeline. For instance, a box or via is executed when an envelope is received on one of its inputs. This is the most significant case, although there are several other due to the needs of parallel execution and buffer management.

In combination, this means that the initialization box sends one envelope which is then received by all boxes linked to its output and these boxes are then executed by the system. These boxes can then send envelopes further down the pipeline which in turn results in other boxes being executed. One box can produce multiple envelopes so the whole execution is not just one “wave”.

It is important to interpret the term “executed” in the context of the task parallelism. This means that instead of directly performing the action a new task is created and added to the task pool to be executed later. We say that the box or via is enqueued (added to the pool) and then invoked (the code is actually executed by the CPU). This suits the run-time mechanism quite well, since one via can send an envelope to multiple boxes and all of the boxes should then be executed. Instead of sequential execution or launching a thread for each box a new task is created for each of the boxes (the boxes are enqueued). These tasks are then executed (invoked) on a fixed number of threads at an appropriate time determined by the task scheduler. This allows for a parallel execution without spawning too many threads, which would result in poor overall performance.

From the scheduling point of view, the boxes and vias behave the same way – they use the same enqueue-involve mechanism, they are enqueued using the same task pool, etc. But there is a significant difference in the way they interact with the rest of the system. It is due to the fact, that while boxes contain user-defined code, vias are an integral part of the framework and their behavior cannot be modified. The next section describes specific properties of boxes that provide support for the user-defined code and as such are not present in vias. A detailed description of vias will be presented later, since it requires deeper understanding of the internal structure of envelopes.

5.2.3 Boxes

The invocation of a box is performed in three steps: the prologue, the action and the epilogue. The actual code looks like this:
if (prologue())
    do_action();
epilogue();

Only the middle part contains any user code – it performs the actual computation. The rest is provided as part of the Bobox framework and it handles communication and synchronization. The user code interacts with the rest of the system only by accessing selected member variables and methods of the box class. This creates a controlled environment for the action – it provides an abstract, stable interface and eliminates possible race conditions.

For instance, the user code may use a method to send an envelope to one of the outputs. Since this action affects the rest of the system and may require synchronization, the envelope is only stored in an internal queue of the box and sent later, during the epilogue.

The following code shows the prologue:

```c
bool prologue()
{
    if (!output_buffers_are_empty())
        return false; // first, check for congestion
    lock.acquire(); // synchronization to prevent the box
        // from concurrently receiving more envelopes
    create_snapshot_of_inputs();
    lock.release();
    return true;
}
```

The epilogue is a bit more complex:

```c
void epilogue()
{
    bool run_again=run_again_request; // the variable is set
        // to true if the user’s code requested that the
        // box should be scheduled again
    clear_snapshot_of_inputs();
    lock.acquire(); // the following operations require
        // synchronization since they interfere with
        // the rest of the system
    if (box_rejected_input_from_via)
        signal_all_input_vias(); // end congestion
    remove_consumed_envelopes_from_input_buffers();
    if (any_input_was_consumed)
    {
```
if (!input_buffers_are_empty) run_again=true;
}
lock.release(); // no need to protect concurrent
// access to inputs any more
bool all_sent = send_output_buffers(); // send all data
// from the output buffers to the appropriate vias.
// Returns false, if any via rejected an envelope.
if (run_again & all_sent) schedule_box();
  // the box schedules itself
}

The prologue creates a snapshot of the inputs of the box and stores it in a member variable available to the user code. This requires synchronization with the rest of the system. The epilogue sends the enqueued output envelopes – this also requires synchronization – and may enqueue the box in the scheduler’s task queue. This may occur for two reasons: the user code may have requested this explicitly (by setting a flag) or the box may have some unprocessed data on the inputs. The second condition is applied only if any data has been processed by the user code during this invocation. Otherwise, the box is considered to be waiting for further input data and is not enqueued until such data arrive (the arrival of data automatically enqueues the recipient).

The ability to again enqueue the box during its invocation has one advantage over repeatedly executing the code of the box right away – the task is placed in the task pool with other tasks and the thread can be used to perform other tasks. The subsequent execution of the original task is thus delayed. This is useful for boxes that can produce large number of envelopes without requiring more envelopes on its inputs. Such box can produce a reasonable number of envelopes and then delay further computation using this “trick”. Without it, the box would have to produce all of the envelopes thus filling the buffers unnecessarily and using a lot of memory for the envelopes. The long run-time of this box would also limit parallelism.

Another notable aspect is handling of multiple inputs – boxes can have more than one input link. Furthermore, since a box can produce several envelopes and send them to an output link, the box on the receiving end of the link needs to handle this situation as well. To address both issues, each box has a small buffers for each of its inputs. When a box is invoked (not scheduled) a snapshot of the head of each buffer is created and made available to the box. It is then up to the box whether it actually processes the envelopes on the input or leaves them in the input buffer. If the box processed any of the envelopes and the buffers are still not empty after the
invocation ends, the box is enqueued again. This is the situation where the box has processed some of its inputs but there are more inputs available. On the other hand, if the box finishes the invocation without processing any of the inputs, it is not enqueued even if the buffers are not empty. This is usually the case of boxes that perform an operation similar to a database join where the data is available for just one of the inputs. Then the join can not do any computation and has to wait for the other input. Since a box is enqueued when a new input arrives, it does not have to be enqueued after the invocation.

The input buffers are also used to balance speed of different parts of the pipeline. They have a limited size and when the buffer is full, the invocation of boxes that send data to this input is suspended. The input buffers of the suspended boxes get eventually full as well. This way a large part of the pipeline can be suspended. This part contains the boxes that produced envelopes faster than the box whose buffer was first filled completely is able to process them. When this buffer is emptied, the invocation of the boxes that send data to this input is resumed. Note that the invocation is suspended or resumed not directly by suspending or resuming the execution of the thread but rather by controlling the process in which they are enqueued. This way the effect is delayed but in our case we do not need it to be immediate.

5.2.4 Parallel execution environment

Having one central task scheduler would create a bottleneck that would reduce parallelism and scalability (the speedup achieved when more CPUs are added). Instead, there is a separate scheduler for each thread in the thread pool. Each maintains a queue of tasks that have been enqueued as a result of an action performed on the thread that the manager is assigned to. When a task finishes, the scheduler starts the execution of the first task in the queue. If the queue is empty, the scheduler steals a task from another scheduler’s queue. The process of task stealing is following:

1. a victim is selected randomly from the other schedulers,

2. several of the tasks near the end of the victim’s queue are removed and placed in the stealing task’s queue,

3. the execution continues normally.

The scheduler steals more than one task to reduce the total number of task stealing, since it requires synchronization between the stealing task and the victim. The task are stolen from the end of the queue, since they have
the smallest chance that their data would still be hot in the cache, so moving them to another CPU (which is likely to result in data not being in cache at all) has the lowest impact on the total performance. The random selection of the victim may result in a suboptimal solution if some of the schedulers have empty queues, but this strategy eliminates the need for any central coordination thus providing better scalability.

One task can be in at most one queue at any time, so if there is a request to enqueue a task that is already in a queue, the request is ignored. However, if the request arrives while the task is being invoked, it is placed in the queue. This allows us to enqueue a box or via any time it receives data. Such event either enqueues the box (or via) or leaves them in the queue. If new data arrives during the invocation of the task (box or via), it is enqueued. The rationale for this is that each task is supposed to create a snapshot of the data (the input to be processed) at the start of the invocation and then process the snapshot. If new data arrive during the invocation, they are not “visible” to the task and become available the next time the task is invoked.

Another feature of the task scheduler is cancellation of requests. This means that request correspond to queries sent from the clients. Each task (it’s representation in the memory of the computer) contains the ID of the request it belongs to. If a task is canceled, all schedulers receive the ID of the canceled request and eventually (after they finish the current task invocation) remove all tasks that belong to the canceled request from the queues.

An example of the scheduling is displayed is displayed in the Figure 5.3 where the pipeline is shown along with the task queues of the four threads used in this case.

1. The initial box is enqueued (shown in the Figure 5.3(a)).

2. When the initial box gets invoked, it produces the poisoned pill, sends it to the first via and enqueues the via (5.3(b)).

3. The via duplicates the poisoned pill, sends it to the boxes and enqueues them (5.3(c)).

4. The thread no. 2 steals one of the tasks (5.3(d)).

5. Later, both tasks are invoked, the boxes produce the results and send them to the appropriate vias. Note that one of the boxes has two outputs and sends a result along both of them. Also note, that the newly created tasks are enqueued with the same thread that the creator was invoked on. This is shown in the Figure 5.3(e).
Figure 5.3: Scheduling example
6. In the last step shown in this example (5.3(f)), one of the task gets stolen from thread no. 1 by the thread no. 3.

5.2.5 Flow control

Consider a part of a pipeline that consist of just two boxes. The first box $P$ quickly produces data and sends it to the second box $C$ that performs a complex computation. At some time after the start of the computation, $P$ receives the poisoned pill, generates first envelope with data, sends the envelope to $P$ and enqueues itself. The via $V$ that connects $P$ to $C$ receives the first envelope ans is also enqueued. After invocation of $P$ ends, $V$ is immediately invoked by the scheduler (this is due to the way the tasks are placed into the scheduler’s queue). $V$ forwards the envelope to $C$, enqueues $C$ and ends the invocation. The box $C$ is then invoked and starts its long computation. At some point, the task corresponding to $P$ gets stolen by another scheduler and invoked. Another envelope is produced and sent to $C$ over $V$.

If this went on long enough, a great number of envelopes would be created by $P$ and remain waiting on the input of $C$. This could consume a lot of memory – up to the total size of data produced by $P$. If the data were produced and consumed at a constant rate, it could cope with (small) constant number of envelopes.

To prevent wasting the memory, the size of the input buffer of each box is limited. If the buffer is full (it is congested), further data is rejected. When – at a later time – data is removed from the buffer, the via that sends data to this input is notified of this fact and resends the data. This is implemented by enqueuing the via – trying to resend rejected envelopes is default behavior for vias.

The same mechanism is also implemented in vias – they have a limited input buffer, reject inputs and then “wake up” (enqueue) the source box if the buffer is no longer full. This way, the congestion can propagate in the direction opposite to the flow of data. As a result, even if the slow box is near the end of the pipeline, it can stall the whole pipeline and limit the number of memory used by the pipeline.

However, there is a downside to this mechanism. Consider a situation with one stream of numbers where the intention is to filter out those numbers that are above the average. The pipeline could split the data into two branches – this is done by a via $V$ with one input and two outputs. One branch does nothing with the data (a “pass-through” branch) and the other computes the average (implemented by a box $A$). These branches are then combined by a box $C$ with two inputs – the data stream and the average
value. This box cannot process the data stream until it receives the average. The box A must wait for the end of the data stream (the poisoned pill) before producing the result. But, if the number of envelopes in the data stream is larger than the total size of buffers on the pass-through branch, the box C would cause congestion on the pass-through branch which will propagate up to the via V and the via V will also start rejecting any further inputs. This is – in effect – a deadlock situation. The box A is waiting for data to be sent by the via V, V is waiting for C to end the congestion, and C is waiting for the average value computed by A.

A similar example for the buffer size of four is shown in the Figure 5.4 – the small squares represent envelopes stored in a buffer that corresponds to the input that the arrow on which it is drawn leads to. The execution proceeds like this:

1. The data flows without waiting in the buffers (the Figure 5.4(a)).

2. The box 4 starts rejecting inputs, which first start to queue at the input from via 3 to box 4 (the Figure 5.4(b)).

3. All buffers on one branch of the pipeline get full, thus stopping it completely (the Figure 5.4(c)).

4. The box 4 starts once again consuming envelopes on its input and the congestion disperses.

But if the box 4 needs to get all data from the box 2 (the situation described in the previous example), the fourth step might never occur and the computation freezes. The solution to this problem is left to the language
module that creates the model. It has to identify such situations and (in the above case) solve it by adding an extra box to the pass-through branch. The new box should accept all incoming envelopes and store them until the poisoned pill is received. This way, the C box does not get congested and the A box can get the complete data and compute the average value. It means that all of the data is kept in the memory, but this cannot be avoided in this scenario. However, there are alternative solutions. For instance, if the stream can be computed twice with reasonable cost, then both branches can have their own source and even if the pass-through branch gets congested, it does not create the deadlock cycle.

5.3 Data level parallelism

The specific way in which the parallel execution is performed by Bobox makes it more useful for data processing scenarios, like database query evaluation or stream processing. This is further supported by the way in which the envelopes and vias are designed.

So far, we considered the envelope to be the smallest piece of data. But it does have an internal structure to store the actual values. Each envelope contains a small table with data in a column oriented way [59]. That is, instead of storing data for one row together, each column is stored separately as a sequence of fields of that column. For an example, see the Figure 5.5. Each column contains the same number of rows as the others, but each column has its own data type. The list of column data types (type descriptors) is a schema of the envelope. We require all envelopes that pass through one via to have the same schema. This schema is shared among all envelopes of the same type to conserve space.

The data for one column are stored in one continuous block of memory where each field has a fixed size. This allows for easier data alignment and the use of SIMD (single instruction – multiple data) instructions like SSE. The columns are stored as a set of shared pointers. This allows new columns to be added easily and even allow one column to be used in several envelopes and ensures that it is deallocated when all envelopes that reference it cease to exist. This can be useful in situations where one envelope is sent along two branches of the pipeline and some of the columns are used in both branches but none of them modifies it. If the data is not copied but only shared among the branches, we do not only conserve space and the time required to copy the data, but we also increase the chance that the data is present in the CPU cache.
5.3.1 Poisoned pill

There is a special type of envelope called a *poisoned pill*. It contains no data (the number of rows is zero). The purpose of this envelope is to inform the boxes that no more data will be send over the link the pill was received from. This is the envelope sent by the initialization box to start the pipeline. Other boxes have to be programmed to handle and pass the pill correctly, although they can postpone passing it for a long time.

The poisoned pill also terminates the computation. All pipelines are required to contain one termination box and all paths in the pipeline are required to end with this box. When this box receives the poisoned pill on all of the inputs, the computation ends and the pipeline is deallocated.

The poisoned pill is (from the programmers point of view) the same as an envelope with data, except that it contains no columns, no schema and a flag marks it as a poisoned pill. This is useful for the BoBox architecture, since it requires no special handling (no added complexity) of poisoned pills. Although the poisoned pill is used to mark that no more data will pass through a certain path, this is not used by the execution environment. It is not necessary, since the proper clean-up and correct interpretation of poisoned pills is left to the boxes. The little resources that could be saved by deallocating the boxes as soon as possible (after the poisoned pill was sent to all outputs of the box) would not justify the added complexity and limitation imposed on the design of the boxes.

For instance, a box with one input and output can receive the poisoned
pill and immediately pass it to the output, but enqueue itself in the process. The poisoned pill is sent further down the pipeline and at a later time, the box is invoked again and it can do the clean-up at this time without delaying the rest of the pipeline until the clean-up is finished.

5.3.2 Variable size columns

Some of the columns store values whose binary representation has variable size. For instance, string literals with varying length and unlimited (or large) maximal length. Such strings cannot be stored in an array of same-sized fields. A similar problem exists in row oriented systems, but the whole row (with all the strings stored in the row) is stored in one block of memory. This may seem like a small difference, but when low-level data-intensive computations are concerned, such details become significant. The following sections discuss several approaches to storing variable size columns.

**Limited size** If the maximal size of the representation of all values is limited by a constant, the data can be stored by extending each value to the maximal size. The problem is, that such maximum may not exist or that the average size is much lower and this layout would waste too much space.

**Indirect storage** The actual value is not stored in the column. Instead, only a fixed size handle for the value is stored in the column and the actual value is stored in a dedicated storage. The handle may reflect some properties of the value or be totally independent. For instance, the storage may be set up to detect duplicate entries and create handles in such a way that handle(X) = handle(Y) iff X = Y.

**Overflow** A modification of the previous technique. The data is split into two columns. The first contains first N bytes of each value and the second stores the rest (if any) in an indirect storage.

**Direct storage** Usually, each column is stored as a continuous block of memory with a well known size equal to (field size) * (number of rows). However, if we relax this requirement, the variable sized columns could be stored sequentially and directly in the column data block. There are two possible storage strategies: the data can be stored in one large continuous block of memory or it can be split into several smaller blocks. The first option uses less resources, but the other is more flexible, especially when the total required memory is not known in advance or when the data can be modified.
Except for the first option, all storage strategies can handle any data, but each have their individual strengths and weaknesses. At such low level, effects of caches and even virtual memory paging have to be considered.

5.3.3 Vias

Unlike boxes, whole vias are integral part of the Bobox framework – they contain no user code. They do not interpret the data in any way and pass it from their input to the outputs.

We require all envelopes that pass through a certain via to have the same schema. This implies that all envelopes that a box sends to one of its outputs must have the same schema and also all envelopes that a box receives on an input have the same schema. Different inputs/outputs of the same box may handle different schemas. This is quite natural requirement. If – for instance – a box performs a table join, it receives parts of the two tables on its two inputs and sends parts of the joined result onto its (one) output.

The via is also the only component of the system that can create new envelopes. The new envelope automatically inherits the schema of the via that created it. If a box needs to create an envelope, it must specify the output that the envelope will be sent to. This allows the request to be forwarded to the appropriate via that creates the actual envelope. This mechanism is used to allow the system to use the same schema definition for all envelopes that share the same schema not only from the logical point of view, but also in terms of using the same area of memory instead of each envelope having its own copy of the schema.

If a via has more than one output, a copy of any enveloped received by the via must be created for each of the outputs. However, as we mentioned earlier, making a copy of each column may not be necessary. Instead, the model specifies a vector of boolean values $priv^o$ for each output $o$. The vector $priv^o$ has the same dimension as the number of columns in the schema of the envelopes handled by the via. If a value on the $N$-th position $priv^o_N$ is true, the envelope sent to the output $o$ gets a new copy of the $N$-th column data. Otherwise, the column is shared with the outputs $o'$ where $priv^o_{N'} = false$.

5.4 Advanced concepts

The following sections describe some of the more detailed information related to the Bobox system that is not essential to the understanding of how the computation is performed.
5.4.1 Dynamic pipeline

To facilitate execution of recursive XQuery [5], the model instance is allowed to grow (expand) during the execution. The exact way in which the instance can grow is defined by the model, but the actual decision whether a certain expansion should be performed or not is made during the execution of an instance of the model. In the current version, the expansion is triggered when first envelope reaches a specified box. In that case, the box is replace by several other (interconnected) boxes, some of which may be also expandable.

5.4.2 Memory allocator

Creation of envelopes involves memory allocation – the most important part being the memory for the data, but other, smaller block of memory are also needed by the system. Traditional memory allocators used by C++ runtime environment allocate memory from one global heap. In a multi-threaded environment, synchronization is required – usually in the form of one global lock that is acquired at the start of the allocation (or deallocation) and freed after the operation has been completed. This lock is a bottleneck for parallel execution and limits scalability – with more threads the chance that two threads try to allocate memory at the same time increases. If they do, one of them is forced to wait and cannot perform any useful work.

To avoid this bottleneck, each thread is assigned one memory allocator that allocates memory from a private heap. This way, no locking is required and memory allocation is fast and easy. On the other hand, memory deallocation is more complex, since there is a significant chance that a block of memory \( b \) is deallocated by a different thread \( T_2 \) than the thread \( T_1 \) that allocated it. In that case, the allocator of \( T_2 \) must pass the information about \( b \) to the allocator of \( T_1 \). This requires some synchronization between \( T_1 \) and \( T_2 \). But the situation is different than in the case of one centralized allocator – we only have to synchronize a simple information transfer, not the whole allocation and deallocation process. Such transfer can be performed even with a lock-free structure that uses atomic instructions of the CPU.

Another advantage is the fact, that we only have to consider multi-threaded environment during memory deallocation, not allocation. And while the caller need the allocation to complete before it can continue, deallocation can be easily delayed. So when deallocation of \( b \) is requested on \( T_2 \), the call can return immediately after passing the information about \( b \) to \( T_1 \) and the computation performed at that time by \( T_1 \) is unaffected. The actual deallocation is performed by \( T_1 \) at a later time, when the thread \( T_1 \) makes a call (an allocation/deallocation request unrelated to \( b \)) to its allocator.
Another issue with the memory allocator is the allocation strategy. Traditional allocators are general purpose and cannot make any assumptions about the size and number of memory blocks that will be requested. In Bobox, the most significant part of the memory is used for column data. Usually, the size of such blocks (see sections 5.3 and 5.3.2) is the size of one field times the number of fields (number of rows carried by the envelope). The field size is often a power of two – either due to the architecture (e.g., a 32-bit integer) or design (e.g., a 64-byte long string prefix). The same holds for the number of rows. With the boxes we have implemented so far, each envelope produced by a box (with the possible exception of the last envelope before the poisoned pill) has the same number of rows. Furthermore, the number of rows is usually relatively large to reduce the total number of envelopes used for the computation, since each envelope produces some overhead – memory is needed to store the envelope and CPU time is used to pass it around. As a result, the number of distinct block sizes is limited.

This observation can be used by the memory allocator speed up allocation and reduce memory fragmentation. On the other hand, the effects of CPU caches has to be considered even when memory allocation is concerned. When a block of memory is deallocated, it is usually still hot (stored in the CPU cache) and it would be useful to use it as the next block to be allocated. This effect is noticeable especially in the case of small, computation and memory intensive models that are able to fit most of their data into the CPU cache. In other cases, the bad cache effects of the normal data access (reads and writes to the data) dwarf effects of the cache-aware allocation.

5.4.3 Bobox server

So far, we only considered Bobox to be just a parallelization framework for data processing. But there is also a server part of the project, which aims at creating a standalone, multi-purpose database server. It has a modular architecture and can support different data models and query languages. The following parts of the text describe only a general architecture of such server independent on any data model, query language or data storage.

The Bobox server consists of several parts (see the Figure 5.6). The front-end handles communication with clients and is completely independent on the query language and data format. As we have already mentioned, it is a module for the Apache HTTP server. The back-end consists of the Bobox parallel execution environment and one or more language modules. The language modules translate queries received from the client to the corresponding Bobox model, which is then executed. One module can handle one or more query languages, for example one module may handle SQL language while
another module handles both SPARQL and SeRQL.

The language modules are responsible for both translation of the queries and their optimization. For instance, if they perform cost-based optimization and require some statistics, they have to maintain those statistics themselves. They may also cache execution plans (models) for previously translated queries.

All queries are executed by the same execution environment, even if they are initiated by a different client or using a different query language. This means that (among other things) they share the same thread pool. This is a common technique in task parallelism and it prevents the system from executing more logical threads than there are physical cores in the system, which would result in task switching and reduce performance.

To allow interoperability between the server and clients that use different languages and platforms, we decided to use the HTTP as an interface between the server and clients. It is a well defined protocol and it can easily pass various NAT or proxy servers. However, creating a complete and efficient implementation of the HTTP is not an easy task.

Instead of writing the front-end from scratch, we decided to use the Apache HTTP Server [63] and create our front-end as an Apache module. This way, the HTTP communication is handled outside our code and we only need to transfer requests and responses between the Apache module and the Bobox server.

5.4.4 Implementation

The C++ programming language was chosen for the Bobox system. Some of the main reasons for this choice were:

- generic programming (templates) – for instance the ability to handle different data types with one code without resorting to virtual functions,
• ability to precisely control memory management and even use custom memory allocation methods tailored for specific tasks,

• portability - with carefully written code, the Bobox can be run on different operating system and hardware platforms.

Since the interface between the Bobox server and clients is built on top of the standard HTTP protocol, the programming language and environment used for the server does not limit their choice on the client side.

5.4.5 The ulibpp library

While the C++ language itself is portable across different platforms and operating systems, different properties of the platforms (for instance the pointer size), systems and APIs can result in different behavior of the same code in different environments. The ulibpp library is designed to hide these differences and provide a universal interface for all systems and platforms.

For example, it defines integral data types that have the same size on all platforms and compilers. The C++ standard only defines some constraints on these sizes, but in order to create a working database system, exact values are required. Another example is an interface for multithreaded programming – while the pthread library and the Windows API provide very similar services, the programming interface is different. Again, the ulibpp library provides one interface that works the same way on both systems.

5.4.6 Run analysis

Debugging complex, multithreaded programs is a very complex task. To provide at least some information of what operations the system performed, debugging messages are collected at different points in the system. To minimize the impact of logging on the overall system performance, the logs have a very simple binary format and they are collected and stored at many different places in the system rather than creating a centralized log that would require synchronization. At the end of the computation the logs have to be aggregated from all storages.

The logs are then saved to a file to be viewed offline later. A special tool – the Bobox log viewer – was created to do this. Since it is a general Bobox tool, it mostly logs engine related events, especially related to the scheduler.

Some examples of logged events include:

• a box/via was enqueued,
• the invocation of a box has just begun,

• the invocation of a user specified box code is about to begin,

• an envelope was received by a box,

• a new envelope was created.

The number of logged events is very high. Since they are collected at different places in the system in different threads they must be marked with some kind of a time stamp so that they can be ordered into one linear stream of events. However, the frequency of these events may be higher than the frequency of the clock provided by the system. The system does provide high frequency clock but they can either work incorrectly on multiprocessor machines or they may require too much time to complete.

For these reasons, we decided not to use wall clock as the timestamp but rather a logical clock. It is a global 64 bit integer which is used as a timestamp for the event and incremented. This is done using atomic CPU instructions. Although it requires the variable to be shared among different threads, the effect on the overall performance is minimal.

This way, we can only measure time required by different operations only in number of logged events that occurred during their execution. To create a mapping of events to real time, we have introduced new event – from time to time, we log current “wall clock” time. This way, we get mapping between the logical clock and time in seconds. But we only have mapping for some of the events and have to interpolate the rest.

The tool is intended to be used for debugging and can provide some overall idea of what is going on during the computation. The interpolated time is not accurate enough for performance measurements of individual components, because the pace at which the timestamp increases is far from linear.

The logged events only contain limited information. Besides the timestamp, originator and event type, only two data fields sized 32 and 64 bits are available. This places some requirements on the log viewer since the events do not contain all the information that may be required to display enough usable information. The viewer must simulate the computation of the Bobox system and display information based also on the events that occurred before the displayed event.

For instance, when an envelope is received by a box, the system must check the type of the event that created the envelope. Only by checking this information can it determine whether it was a normal envelope (with data)
or a poisoned pill. And since the event that the box received a poisoned pill is very significant, we need to perform this check.

The user interface of the log viewer is rather simple. It can list the events and filter the list by the event originator and event type. It can also display a time-line of the computation. For an example of the time-line see the Figure 5.7. The four rows at the bottom represent utilization of the four CPU cores used in the execution. Black parts represent the overhead, the gray parts the actual computation. Since we used very small data blocks (1024 rows per envelope), the overhead is large. The top part displays the time periods at which the individual boxes and vias were active — one box or via is represented by one row. They are sorted according to the time of their first execution, which creates the stairs effect. It is easy to see that there were 5 envelopes sent through the system, with the fifth being only a poisoned pill. The box that produces data is highlighted by red color, which shows it is the third row in the top part and that it was executed by all of the four cores (the bottom part), each time producing one envelope with data. The fifth execution at later time means it forwarded the poisoned pill, but only when the first of the CPU cores finished processing the data. Also note that all steps related to one envelope were performed by one CPU core, which is desirable for better cache utilization.
5.5 Experiments

Even though the implementation process is in its early stages, we have prototype implementations of the key Bobox components, most notably the parallel execution environment and some of the boxes. This allows us to conduct experiments to partially assess performance of the system, even though it cannot yet work as a stand-alone server.

5.5.1 SP²Bench

The system is now able to execute some of the queries of the SP²Bench [57] benchmark in a scenario that closely simulates an in-memory RDF database. However, we do not have a SPARQL compiler at this moment, so the queries were translated by hand into a Bobox model.

These tests were performed on an Intel Core i7 860 processor (2.8GHz, 4 cores, hyper-threading, 4x256KB L1 cache, 8MB L2 cache) workstation with 4GB of memory (DDR3, dual channel, 667MHz) running the 64-bit version of Windows 7 operating system. The HDD performance is not significant for this (in-memory) test setup. We also do not need to “warm up” the system (run the query a few times to get data from HDD into memory cache) since no caches of this type are used in a in-memory database. The Bobox parallel execution environment was configured to use only four threads, to eliminate effects of hyper-threading since all Bobox threads do intensive computations. The Sesame benchmarks were performed on a default installation of Sesame 2.3.1 with a fully in-memory database.

The SP²Bench can be configured to produce data of different sizes. In the following text, we identify the data by the number of triples, e.g. 1M data contain one million triples.

The data is stored in memory and indexed with indexes PSO, POS, OPS, and SPO, where PSO denotes an index on predicate, subject and object is this order. All join operations were performed by merge joins.

Q1 This query is a simple selection of a pattern from a database.

```
SELECT ?yr
WHERE {
  ?journal dcterms:issued ?yr
}
```
The only question in making a query plan in this case is the ordering of the join operations implied by the graph pattern, but even if the triples are located and joined in the order in which they appear in the query, the plan works very well.

On 1M data and five runs, the average time was 23.2 milliseconds (standard deviation 4.38). This result only demonstrates that the total overhead of Bbox is small for simple queries. It would not be reasonable to directly compare it to other systems, since the overhead of e.g. network communication and execution plan generation would affect the other systems’ results too significantly. Just to demonstrate, the Sesame can perform the query in 53 milliseconds (std. dev. 8). However, the architecture of the benchmark is much more complicated (it involves Tomcat server and HTTP communication).

Q2  A simple query, but with a large graph pattern.

```sql
WHERE {
    ?inproc rdfs:seeAlso ?ee .
    ?inproc dcterms:issued ?yr
    OPTIONAL {
        ?inproc bench:abstract ?abstract
    }
}
ORDER BY ?yr
```

Again, the most straightforward evaluation plan works very well, but the amount of data processed by the query is much larger. This results in a much longer execution time. The average time for five runs and 1M data was 1101 milliseconds (standard deviation 164). Note the relatively high std. dev. of the times. This is caused by the complexity of task-parallel execution — the different timing of subsequent runs may result in a different order of task execution and different assignment of task to threads. This does not affect
the computed result, but it does affect performance. The Sesame performs
the query in 550 milliseconds (std. dev. 16) when measuring the time from
the start of the execution to the time first row arrives. The result is large,
so waiting for the last row would be too affected by the performance of the
transfer.

The effect of scheduling on the (in)stability of execution time is increased
by the fact, that the test was performed with small chunks of data – the
envelopes only contained 1024 rows each. By increasing the number of rows,
we reduce the overhead and impact created by scheduling. For instance, by
increasing the chunk size four times, the time is 562 milliseconds (std. dev.
12.1).

Q3 and Q4 These queries are aimed at testing the sophistication of the
query optimizer. Even though we did try the queries, the results are not
given here, since they are not very relevant for our setup.

Q5 Two variants of one query, important is the use of the DISTINCT key-
word. This is the Q5a version:

SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?person foaf:name ?name .
  ?person2 foaf:name ?name2
  FILTER (?name=\?name2)
}

The average time is 2284 milliseconds (std. dev. 75). The streamed data
and merge joins performed quite well in this case. An example of the Bobox
model that corresponds to the query is shown in the Figure 5.8. The Sesame
server was unable to complete the query. However, the Q5b version of the
query can be evaluated by the Sesame. The query looks like this:

SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?person foaf:name ?name
}
Figure 5.8: Model for query Q5a
The Sesame took 683 milliseconds (std. dev. 13) while the Bobox version takes 486 milliseconds (std. dev. 17).

Q6  This query is in fact a simple “not exists” query.

```sql
SELECT ?yr ?name ?document
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?document2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  }
  FILTER (!bound(?author2))
}
```

The average time is 7734 milliseconds (std. dev. 75). Unfortunately, the Sesame server was unable to evaluate this query.

Q7  Q7 is only a slightly more complex variant of the previous query.

Q8  Q8 combines graph patterns, filters and union. Unfortunately, we do not have a usable implementation of some of the operation at this time.

Q9  The Q9 query is aimed at testing the query optimizer and the way it handles the DISTINCT modifier.

Q10  The query Q10 is extremely simple and is aimed to test selection on objects of triples.

```sql
SELECT ?subject ?predicate
WHERE {
  ?subject ?predicate person:Paul_Erdos
}
```
Since we have an index (two of them, in fact) that index predicate as the first "column", the implementation is trivial and evaluation takes very little time. In fact, most of the runs were completed in times shorter that the resolution of the system timer. Surprisingly, the Sesame took 75 milliseconds to evaluate the query (std. dev. 7), which is slower, even considering the overhead caused by the more complicated test setup.

**other queries** The remaining queries of the SP2Bench benchmark test variants of the basic SELECT query, for instance limit on the number of results or the ASK query that tests the non-emptiness of the result. They can either be evaluated in the traditional way and then interpreted differently or handled by the optimizer in a completely different way. But both cases are not relevant to our scenario.

These results of the Bobox implementation have to be interpreted carefully. First, they only represent time used by the query evaluation. In real-world deployment, the query has to be sent to the server, optimized, then it is executed and the results have to be sent back to the client. Second, we have to remember that the queries were translated and optimized by hand, although only using basic operations (merge join, sort, selection, projection, and indexed access) for evaluation.

The most important test in our case is the Q2 query of the benchmark. The reason is, that the query takes long enough to evaluate in both Bobox and Sesame to make the results relevant even considering the limited precision of the system clock. Furthermore, it is very simple so the quality of the query optimizer is much less significant and even simple evaluation techniques currently implemented in Bobox are sufficient to handle the query with decent performance. With the initial setup, the Bobox took almost exactly twice the time of the Sesame. The times achieved by the Bobox implementation were less stable than in the case of the Sesame. With some correction of the Bobox parameters, the time gets very close to Sesame. The stability of times is also similar.

We are still working on a SPARQL language module with comprehensive support of the language and a cost-based optimizer. Only when it is completed, tested and tuned to provide efficient plans, will we be able to do a fair performance test of the complete SP2Bench benchmark.

If we take all these considerations into account, the parallel evaluation engine of the Bobox system provides competitive results compared to the Sesame. RDF databases provide an interesting comparison, since they are relatively young compared to XML or relational databases. This way, the
<table>
<thead>
<tr>
<th>size</th>
<th>time (ms)</th>
<th>std. dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>1101</td>
<td>164.3</td>
</tr>
<tr>
<td>2K</td>
<td>536</td>
<td>21.7</td>
</tr>
<tr>
<td>4K</td>
<td>562</td>
<td>12.1</td>
</tr>
<tr>
<td>8K</td>
<td>606</td>
<td>6</td>
</tr>
<tr>
<td>16K</td>
<td>714</td>
<td>16.8</td>
</tr>
</tbody>
</table>

Table 5.1: Query performance of Q2 with different envelope sizes

<table>
<thead>
<tr>
<th>size</th>
<th>time (ms)</th>
<th>std. dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>2284</td>
<td>80.7</td>
</tr>
<tr>
<td>2K</td>
<td>1892</td>
<td>62.1</td>
</tr>
<tr>
<td>4K</td>
<td>1794</td>
<td>59.7</td>
</tr>
<tr>
<td>8K</td>
<td>2152</td>
<td>141.2</td>
</tr>
<tr>
<td>16K</td>
<td>3943</td>
<td>160.4</td>
</tr>
</tbody>
</table>

Table 5.2: Query performance of Q5a with different envelope sizes

The performance of the system is still very significant and is not mitigated by the quality of the query optimizer.

5.5.2 Envelope size

As we have already noted in the analysis of the Q2 query performance in the previous section, the envelope size (the number of rows stored in an envelope) may significantly affect the overall query performance. At this moment, we only consider situations where the envelope size is the same for all envelopes, although the architecture of the system does not require it.

Tables 5.1 and 5.2 show the times that were achieved by the system for queries Q2 and Q5a with different envelope sizes. Both tables show the optimum to be around 2K and 4K sizes. The optimum is affected by two main aspects – scheduling granularity and cache size.

The scheduling granularity works in two opposite directions. Larger envelope size means that tasks take longer to complete and their number is reduced. This reduces the scheduling overhead. On the other hand, longer tasks reduce the amount of time the system is able to run in parallel. For instance, if there is just one box connected to the output of the starting box and it produces a lot of data, no other box can be enqueued and invoked until the first box completes, since no other box has any input data.
<table>
<thead>
<tr>
<th># of threads</th>
<th>time (ms)</th>
<th>std. dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13416</td>
<td>8.4</td>
</tr>
<tr>
<td>2</td>
<td>6741</td>
<td>11.0</td>
</tr>
<tr>
<td>4</td>
<td>3708</td>
<td>188.3</td>
</tr>
<tr>
<td>8</td>
<td>3358</td>
<td>607.1</td>
</tr>
<tr>
<td>16</td>
<td>9391</td>
<td>2519.6</td>
</tr>
<tr>
<td>32</td>
<td>17104</td>
<td>3044.1</td>
</tr>
</tbody>
</table>

Table 5.3: Query performance for different number of threads

The main effect of the cache is, that if the data become larger that the available cache, the performance decreases. In simple setups, a single point where the data no longer fit into a cache can usually be easily identified as a sudden drop in performance. Since most CPUs have several levels of caches of increasing sizes (and decreasing speed), several drops are usually present.

In our scenario, the computations are not that memory access intensive, so the cache effect is insignificant when compared to the effect of scheduling. A lot of processing is performed between reads of consecutive row values, which allows the CPU to use its pre-fetch and speculative evaluation to mitigate the effect of cache misses.

We have also performed several tests that were focused on testing the impact of the CPU caches. These tests showed that the effects of L1 cache are negligible, but the highest level (L2 or L3) cache does affect the optimal size of the envelope. The problem is, that this size depends significantly on the task that is being executed, since the ideal situation is that all the data, that are actively being processed fit into the (highest level) cache. But the number of such envelopes may differ significantly from task to task and even the memory allocation strategy. At this moment, the cache effect is lower than the effect of the scheduler, so the size of the envelopes is chosen to create a reasonably sized work chunk without regard to the cache size.

5.5.3 Parallelism

The SP²Bench tests do not work well as tests for parallelization. The problem some of the operations (mainly sorts) take significant portion of time and in their current implementations cannot be parallelized. The tests demonstrate the performance of data flow through the pipeline or they could be used to demonstrate the performance of multiple concurrently executed queries.

To test speedup of a single query and limit the influence of other factors, we created an “ideal conditions” test. In this test, we use a linear pipeline
with 12 boxes (besides the initialization and termination box). First box produces a stream of data, then there are ten boxes that perform computation intensive operation on those data. The 12th box only receives the data and does no more processing. This is the setup that was used in the time-line example shown in the Figure 5.7 (in the Section 5.4.6). The Table 5.3 shows the results of the test performed on 256 of envelopes (with 4096 rows each) for different number of threads. All tests were performed under the same conditions (except the thread number) as SP²Bench tests.

The results are close to what was expected – for two and four threads, there is a significant speedup over one thread. When eight threads are used, the speedup is much lower, due to the fact, that the CPU has only four physical cores and the eight logical cores are provided by HyperThreading function. The actions performed by the individual threads are computation-intensive, thus reducing the effect of HyperThreading. Higher number of threads only increases overhead while providing almost zero increase in available computational power, since all physical cores are already completely utilized.

The times for number of threads from one to sixteen is shown in the Figure 5.9. A reasonable number of threads appears to be from 4 to 8. In this test, the best performance was achieved by 7 threads. However, the difference in the 4 to 8 range is too small to make any definitive conclusions, until a more representative set of test models is available. The optimal number of threads
can also be affected by using different scheduling strategies. Still, the range from 4 (the number of physical CPU cores) to 8 (number of logical threads of the CPU) appears to be a good option for task-based parallelism.

Also note the decreasing stability of the times with increased number of threads. With one thread, all five runs provided nearly identical times, but with 32 threads, times ranged from 14939 to 22481 milliseconds. This is also quite natural. With just one thread, the system always executes all tasks in the same order. For two or more threads, the order cannot be determined in advance due to, for instance, task stealing, which itself contains a random process (the “victim” is picked at random). Of course, exact timing of the concurrent actions starts to have an effect, as well as process scheduling performed by the operating system.

5.6 Related work

The most similar project to the Bobox run-time is the TBB library. The TBB was one of the first libraries that extensively focused on task level parallelism and it provides a powerful set of structures, algorithms and run-time environment. Compared to the Bobox library, it is low-level solution – it provides basic algorithms like parallel for cycle or linear pipeline and very efficient task scheduler. The developers are even able to directly create task for the scheduler and create their own parallel algorithms. But the task are designed in a way that makes it very hard to create a non-linear pipeline similar to the one Bobox provides. Such pipeline may be necessary for complex data processing [5]. Bobox also provides more services for data passing and flow control.

The newest version of OpenMP also provides a way to execute tasks in parallel, but it provides less features and less control than TBB. The OpenMP library is mainly focused on mathematical computations – it can execute simple loops in parallel really fast, it can also run blocks of code in parallel, but it is not well suited for parallel execution of a complex structure of blocks. Unlike TBB or Bobox, it is a language extension and not just a library. This means that the compiler is well aware of the parallelization and optimize the code better, but it also enables OpenMP to provide features that cannot be done with just a library, like defining the way variables are shared or private among threads with a simple declaration. In TBB such variable has to be explicitly passed to an appropriate algorithm by the programmer. In Bobox, it must either be explicitly passed to the model or sent using an envelope at run-time.

But there are also completely different approaches to parallelization. One
way would be to create a thread for each box and via in the model instance. This would also ensure that each box or via is running at most once at any given time. There are two main reasons for not using this architecture. First, it creates a large number of threads, usually much larger than the number of CPU cores. Although it forces the operating system to switch the threads running on a core, it may not impact the overall performance that badly, since it can be arranged that the idling threads (those assigned to a box or via that is not processing any data at the moment) are suspended and do not consume any CPU time. The second problem is, that when data (envelopes) are transferred from one box to another, there is very little chance that it would still be hot in the cache, since the thread that corresponds to the second box is likely to be scheduled to a different CPU, that does not share its cache with the original one. Both of these problems are avoided in TBB and Boxbox.

Another different approach is a monolithic architecture, where queries are evaluated by a query evaluator specialized for the task with an explicit parallelization. Although such systems may be able to achieve best possible performance, their design and implementation is extremely complex and they are tailored just for the specific problem they are dealing with. This approach is used by most parallel database engines. The Boxbox project aims to be more general, even at the cost of some performance.
Chapter 6

Conclusion

In this work, we covered a significant portion of a lifetime of an RDF query. We have provided the software developers with a visualizer that lets them familiarize themselves with the data that their software should be processing. This helps them formulate queries over the data more easily, increasing their productivity.

They could also benefit from the TriQuery language that we have created. It is based on XQuery, which many of the developers already know very well, and it does not change the basic principles of the language, only slightly extends it. The extension allowed us to provide XQuery with RDF and relational database querying capabilities, that are easy to use and integrate well with the XML processing available in XQuery. This allows the developers to use the language features and built-in functions already available in XQuery even when processing RDF data.

The last step in the lifetime of a query is evaluation of the query. This step is important both for the developers and the users of any semantic-enabled application, since all of them need the results to be correct and the evaluation of the query as efficient as possible. We have designed the Bobox parallel framework with both of these aspects in mind – on one hand, it allows us to create an efficient parallel evaluation engine that can provide good performance, on the other hand, it eliminates many of the pitfalls of parallel programming which created by concurrent evaluation of code, like race conditions in data access.

The following parts of this chapter describe the individual results presented in this thesis in greater detail, followed by a section with future work.

**Triangle layout algorithm and node merging** We have designed a new graph drawing algorithm – the triangle layout – for drawing of trees. The algorithm is tailored to fit the specific situation of RDF visualization with
the node merging technique. We have provided a proof that the area used by the algorithm is asymptotically optimal.

The node merging technique has dramatically decreased the area required to draw a neighborhood of a node and provides the user with clear overview of possible directions for future navigation. The technique can be adopted even by other RDF visualizers, providing they can handle visualization of rectangles with varying height.

The triangle layout algorithm could also be used in other applications. One of the parts of the visualization – the edge routing algorithm used for drawing of non-tree edges – can be used separately from the rest to draw edges in any situation, where the positions of nodes are fixed and the nodes do not overlap.

**Visualization of large RDF data** We have proposed several methods of dealing with the volume of the RDF data, since even basic data collections are way too large to be visualized all at one time. An obvious solution, which is also used in all other RDF visualizers is to use incremental exploration (navigation). However, choosing the right place to start and presenting the user with future directions of navigation is still a problem due to the large number of possibilities. The visualization section of the work provides some ideas for dealing with these problems.

**Records in XQuery** The addition of records to XQuery is an invasive operation, but only a minor one. The grammar of the language needs to be modified only at a few places. The impact on the language is twofold: it provides new language constructs that can be used by the user and it changes the way the queries are evaluated. An important aspect in both cases is the requirement, that if a sequence contains a record, it may contain nothing but records and all records must contain the same fields.

This “limitation” is in fact the core of the contribution – it can easily be enforced by statically checking a query, which is the advantage of records over using XML fragments. It allows the evaluation engine to use more aggressive optimizations since the possible inputs of operations are significantly limited.

The semantics and even the syntax of the `match` operation are tailored to allow an easy inclusion of the RDF support. Although it may be used outside RDF, its usability would probably be limited.

**Evaluation of TriQuery** The records and operations on records can be translated to pure XQuery. If the fields do not contain node references, a records can be transformed to a XML fragment in a very straightforward
way. If it contains the references, it has to be done in a more complicated way and the processing of the records would be complicated with a rather poor performance.

This means that some level of support of TriQuery is necessary to be provided by the query evaluation engine. Fortunately, most XQuery engines use some form of relations for internal data representation, which means they should be able to include more columns needed for records handling with relative ease.

**RDF and relational extensions** When records are used, the extension of XQuery to cover relational databases is very straightforward and natural. The RDF support makes use of the pattern matching operation that was introduced along with the records specifically for this purpose. It provides pattern matching on a level very similar to the current version of SPARQL, but the **with** modifier allows the language to be easily extended to cover RDFS or OWL entailment. Of course, this solves the problem only on the language level, an efficient implementation is a separate problem.

**Bobox parallel framework** To simplify development of various parallel data processing engines, we have created the Bobox parallel framework. It follows the design principle of task level parallelism, which is one of the main trends in parallel programming at this time. It is well suited for current multi-core processors and their cache architecture. The framework is designed to greatly simplify the implementation of data processing engines by hiding the difficulties of scheduling, flow control and synchronization from the programmer while still providing decent performance.

It can be used in applications that process large amount of data by the means of a non-linear pipeline that consists of large number of simple components. These components are provided by the user of the framework along with a definition of connections between the components. The framework provides means for defining these connections, the definition of the interface for the components, interconnecting components and the run-time environment necessary to run the pipeline.

**Bobox runtime environment** The Bobox runtime environment takes care of the parallel computation. It creates an interface between the user’s code and the operating system, which simplifies construction of platform independent software as a side-effect. The basic idea and design of the runtime is similar to Intel’s Threading Building Blocks, however, the rules for scheduling of tasks and especially the flow control rules are more complicated. This
is necessary to support the execution of a non-linear pipeline – the TBB only provides a simple, linear pipeline.

The runtime also defines the basic structure of the format used to store and send the data within the system. To enable data parallelism in the form of streaming instructions, the data is stored in a column-oriented manner, rather than the more traditional row-based representation. The users of the library are also encouraged to send the data in large block to reduce the overhead required for task scheduling.

A non-essential part of the Bobox framework is the Bobox server. It extends the runtime environment to create a stand-alone data processing server. However, the Bobox framework does not provide data persistence services, so they have to be provided by as part of the user’s code.

6.1 Future work

We have presented our work in three different areas related to RDF querying. We consider the RDF visualization topic to be already covered in sufficient detail including some collateral problems. There is even an experimental implementation of the algorithms that demonstrates the key features. However, a full implementation of all the ideas into a software that can be used with different RDF storages has yet to be done. Then, a detailed user study of the application should be performed to evaluate the application and the usability of the various algorithms.

TriQuery processor

There is an XQuery implementation created using the Bobox framework as part of another work [3]. We intend to extended it to handle even queries written using the TriQuery language. As we have already stated, extending the evaluation engine itself is not a hard task. A more challenging problem is to modify the query optimizer, since the join operations used to implement TriQuery are more complex – they are closer to SQL. XQuery engines built on top of an existing SQL engine will handle it easily, but our system was designed for the simpler XQuery joins and has to be modified.

Runtime modification of execution plans

One of the ways of further extending the Bobox framework is to allow runtime modifications to the structure of the pipeline. One reason for this is to allow evaluation of recursive XQuery functions. But even more important
reason is to allow experiments with runtime modification of query execution plans in order to achieve better performance (query optimization at runtime). Currently, it is believed to be the best way to handle XQuery, since static optimizations often fail to produce good plans [44].

**Bobox distributed runtime**

At the moment, the Bobox framework only supports parallel execution. But we intend to try even running it in a distributed environment. The architecture of boxes and vias allows us to use vias as the interface between the nodes of the distributed environment. But the problem with distributed computation is that not all resources are available at all nodes. This ranges from locally available physical data stores to the non-existence of shared memory – if the data in an envelope is represented as a reference to a different memory block, for instance an XML fragment, it is not sufficient to transfer just the data present in the envelope. Even the referenced information has to be available at the target node. There are several ways in which it can be achieved – the information may be sent along with the envelope, it may be transferred later upon a request or it may be stored in a distributed manner form the beginning, for instance using a distributed hash-table.

**Ongoing improvements of the implementation**

Currently, we have a working implementation of the Bobox runtime. But there is still much room for improvement. For instance, the task stealing used but the task scheduler is not very efficient at the moment. Also, any data-intensive parallel environment needs to handle memory allocation and deallocation in a more efficient way than the standard functions provided by the language runtime. These functions involve a global lock, which creates a bottleneck and prevents efficient parallelization. The other components of the system and various parameters should also be tweaked to minimize time that is spent outside the user’s code – the overhead of the system.

All things considered, we have designed a set of tools that support the whole process of querying RDF data and implemented most of the key components of the system. But a great deal of further research and development is required before they can be put into practice. We do intend to follow this line of research in the years to come.
Bibliography


[38] Sunil Goyal and Rupert Westenthaler. RDF gravity (RDF graph visualization tool).

102


103


Appendix A

Personal responsibilities

Any research is mostly a team effort. However, for the sake of this thesis, we would like to point out individual contributions of the author and other researchers in relation to the topics of the thesis. Even though no work was performed in total isolation, there are often cases where most of the research performed in certain area can be attributed to one or more individuals.

The research was supervised by Jaroslav Pokorný from the Charles University in Prague, where most of the research was conducted.

The RDF visualization research was performed by Jiří Dokulil in cooperation with Jana Katreniaková from the Comenius University in Bratislava. She designed the layout algorithms (triangle layout, several specialized views) and the edge routing algorithm. Jiří Dokulil dealt with the RDF-related aspects of the problems (e.g. the node merging technique and the algorithms for selection of the starting node), but also provided the proof of the area requirement of the triangle layout and created the pilot implementation of the RDF visualizer. We have presented the research at several conferences [24, 26, 27, 28, 29, 33].

The TriQuery language was designed by Jiří Dokulil in cooperation with David Bednářek from the Charles University in Prague. While Jiří Dokulil – based on his previous research of TriQ (see the Appendix B) – created the basic concepts of the language, David Bednářek helped with integrating the concepts into XQuery and define the grammar of TriQuery. Information related to RDF querying was also already published by the authors [22, 30, 35].

The Bobox system was designed by David Bednářek, Jiří Dokulil, and Jakub Yaghob (also from the Charles University in Prague). Most of the design was created in close cooperation of all of the people involved; however, Jiří Dokulil is responsible for most of the flow control algorithms, logging and log visualization. From the implementation point of view, Jakub
Yaghob created the ulibpp library and scheduler for the Bobox system, Jiří Dokulil implemented the flow control and data handling in the Bobox system, plus the log visualizer used for debugging, and David Bednářek created an experimental XQuery front-end, which is now being converted into an Tri-Query front-end. The benchmarks presented in this thesis were performed by Jiří Dokulil. We have already published less detailed information about the Bobox design [6, 7].

The research into RDF querying and evaluation was preceded by the research of semantic web infrastructure and interfaces called Trisolda, which was performed by Jiří Dokulil, Jaroslav Tykal, Jakub Yaghob, and Filip Zavoral (all from the Charles University in Prague) [31, 32, 35]. The design of the Bobox system bears some similarities to the T4 System developed by David Bednářek, Petr Merta, David Obdržálek, Jakub Yaghob and Filip Zavoral [8].
Appendix B

TriQ

Querying is one of the important issues for any data format. In the case of RDF, many query languages have been developed, including the SPARQL language [53], which is a W3C recommendation, or SeRQL [16], which is supported by the popular Sesame RDF framework [17].

Many are inspired by SQL (although not all of them, e.g., Lisp-like Versa [47]). But the inspiration is usually manifested (besides the fact that the queries syntax may look a bit like SQL) in the fact that the RDF graph is transformed by some graph pattern matching operation into some table-like form and these tables are then further processed.

This appendix deals with a proposal of an algebra that is inspired by SQL on a different level. An important feature of SQL (and its theoretical background – the relational algebra is the fact, that it is a closed system. Relations are transformed into relations. This way, a result of a query can be used as an input for another, more complex query, which is impossible in SPARQL and SeRQL.

The following text presents our proposal for TriQ – a SQL-inspired, closed RDF query system. To make the system closed, we couldn’t have used relations as our data model. We use the RDF. But the operations are inspired by relational algebra – we use selection, projection, (inner and outer) joins, etc. The semantics of these operations is not exactly the same (after all, we have a very different data model) but the ideas behind them are very similar.

B.1 Data model

Since we want a closed query system, we require every operation to take some RDF graphs (zero or more) as its input and produce an RDF graph as its output. But to make the operations simple to use, we need to add some
further information. We use the very RDF that contains the data for the task and add additional triples to the data – decorate it.

**Namespaces** To make decoration simple we define several namespaces. URIs starting with theses namespaces are prohibited in the queried data. The namespaces are

dec is http://ulita.ms.mff.cuni.cz/Trisolda/GQL/decoration
ptr is http://ulita.ms.mff.cuni.cz/Trisolda/GQL/pointer
graph is http://ulita.ms.mff.cuni.cz/Trisolda/GQL/graph

**B.1.1 Decoration of nodes and edges**

We can decorate either nodes of the RDF graph, in which case we add a new triple where the decorated node is the object of the triple, or edges, in which case we have to reify the edge (unless the triple is already reified) and then decorate the reification. To be more specific, to decorate the edge \(<S,P,O>\) with decoration triple \(<D_S,D_P,<?>>\) (the question mark is the decorated object) we add the following triples:

- \(<X,\text{dec:subject},S>,<X,\text{dec:predicate},P>,<X,\text{dec:object},O>\)
- \(<D_S,D_P,X>\)

The X denotes an anonymous node. Note that we do not use the standard reification defined by RDF, but rather use the dec namespace to avoid potential “collisions”. This way we can always distinguish statements added during decoration and statements that were present in the original data.

**Decoration options** There are two types of decoration edges. Let X be the decorated object (either a node or reification of an edge), G an URI from the namespace graph (each graph has a globally unique URI) and P an URI from the namespace ptr. The possible decoration triples are:

1. \(<G,\text{dec:contains},X>\)

2. \(<P,G,X>\)

There are no restrictions for the second type of triples. The only restriction for the first type is that every node and edge of the decorated graph is decorated by at least one such edge.
B.1.2 Meaning of decoration

The purpose of decoration is to help user define a structure in the queried graph and exploit it to define further operations. Furthermore, we would like the whole query system to resemble relational algebra, that works with relations – sets of tuples with a well defined schema.

The first type of decoration triples is used to make the (one) RDF graph appear as if it was a (multi)set of smaller graphs so that each of the smaller graphs resembles one tuple of a relation (row of a table). The triple \(<g, dec:contains,X>\) tells us that the graph G contains X. So if we take all such X for one G, we get one small RDF graph.

The second type of decoration triples were introduced as a parallel to schema of a relation (for an example see the Figure B.1). The set of all values from the namespace ptr used in the graph are schema of the graph. All values of X from all triples \(<p,g,x>\) for a graph G correspond to a value of column P in a row G of a table. But unlike SQL that allows zero (NULL) or one value, we allow zero or more values. Although we could restrict it to just one value, we believe it would unnecessarily limit the graph-handling capabilities of the query system. For instance, we would like to be able to create such set of graphs, where each graph contains information about one person (e.g., first and last name) and all of his or her e-mails. And to make handling of the data convenient, we would like to have a pointer ptr:fname point to the first name of the person, ptr:lname to the last name and ptr:email to all of the emails. That way, we could for example easily find people with more than one email or get the number of emails for each person.
B.2 Graph pattern operation

The graph pattern operation is in a certain sense the very basic operation of the query system. It is used to find patterns in the whole queried data. The very basic principle is that the operation specifies an RDF graph where some nodes or edges are replaced by variables. The evaluation is done by finding possible substitutions for these variables so that the we get a graph that is a subgraph of the queried data. This graph is then one member of the result set.

The operation gives the data a structure that helps us reference certain concepts in further query operations. For example, if the pattern looks like `<?x,ex:has-name,?y>` (where `?x` and `?y` are variables), we know that the actual nodes bound to `?x` are people and `?y` their respective names (provided we have reasonable data).

Each node and edge can of the pattern can have a pointer assigned to it. In that case, the corresponding node or edge of the result is then pointed to by that pointer.

Many RDF query languages (e.g., SPARQL [53], Trisolda query API [32], ...) use some kind of graph patterns to transform the RDF graph to a table (or something analogous like set of variable mappings in the case of SPARQL). In other words, they use it to transform the queried data into some other form suitable for further processing.

At the moment, we believe this is the only operation that should work with “raw” data and that all other operations can assume to be working with decorated data. This is not as important from the technical or formal point of view, but rather from the “average user’s” point of view. It would allow him or her to construct the query in two steps. First, well-structured pieces of data are defined by the pattern operation. Second, the structure is exploited to combine the pieces of data into the final result. The second phase should be as close as possible to writing a query in SQL.

B.2.1 Definition

The previous sections briefly and informally explained what capabilities the proposed pattern matching possesses. This section gives a more formal view of the operation.

Let $\text{Uri}$ be a set of all URIs, $\text{Lit}$ set of all RDF literals, $\text{Blank}$ set of all blank nodes, and $\text{Var}$ an infinite set of variables.

Let $V \subseteq \text{Uri} \cup \text{Lit} \cup \text{Var}$. The pattern $P$ is then defined as a non-empty set of triples $P \subseteq V \times (\text{Uri} \cup \text{Var}) \times V$. $V$ is the set of all nodes in the
pattern $P$, $\text{Var}(P)$ denotes set of all variables used in the pattern $P$. The pattern can be viewed as a directed, labeled multigraph. We require that each two nodes in $V$ are connected by an undirected path.

Each edge or node of the pattern can be assigned a pointer (i.e. URI from the namespace ptr). The same pointer may be assigned to more objects (edges and nodes).

Let $P_b$ be a pattern and $V_b$ nodes of $P_b$. Let $G$ be the queried data.

A variable mapping is a function $\mu : \text{Var} \to \text{Uri} \cup \text{Lit} \cup \text{Blank}$. We extend the variable to triples $t \in P_b$ and the whole pattern $P_b$ in the natural way (each variable $v$ used in $t$ or the whole $P_b$ are replaced by $\mu(v)$). We always use the variable mapping $\mu$ in conjunction with a pattern $P$, in which case we consider only minimal mapping, i.e. $\text{dom}(\mu) = \text{Var}(P)$.

We say, that an RDF graph $M_b$ is a match for $P_b$ iff there are mappings $\mu$ and $\eta$ such that the following statement holds: $M_b = \mu(P_b) \subseteq G$ where $\subseteq$ is a relation between RDF graphs. The basic version of TriQ assumes that $(A \subseteq B) \equiv (A \subseteq B)$.

Then we say that the tuple $\langle \mu, M_b \rangle$ is a result for the pattern $P_b$. Note that we only consider minimal $\mu$ mappings, i.e. mappings that only map variables used in $P_b$. The $M_b$ graph still has to be properly decorated according to the pointers that were assigned to the pattern $P_b$. We add decorating triples to the set $M_b$ in the way described in the Chapter B.1 – if a pointer was assigned to a node or edge of the pattern, we add the appropriate decoration to its image under $\mu(P_b)$.

### B.3 Algebraic operations

This section describes algebraic operations, that run on the decorated data and further filter and transform it. Although, strictly speaking, these operations could be run on undecorated data, but there is usually no reason to do so. In such case, each node and edge of the undecorated data would be decorated by the first type of decoration triples, which would assign the whole data to one graph.

We do not give formal definitions for these operation as they are quite straightforward and usually obvious. They would only add a few pages of not very interesting technicalities to the paper.

#### B.3.1 Selection

The selection operation has one argument and tests a condition for each graph $g$ in the argument multiset. If the condition is true, the graph is added to
the result. The basic idea is the same as in relational algebra, but there is some added complexity due to the fact that one pointer can have more than one value (within one graph) or no value at all. We use a language derived from the first-order predicate calculus to construct the expressions. The main difference from SQL is the addition of quantifiers. The quantifiers are always in the form $Q_{x \in X}$ where $x$ is a variable, $X$ a pointer from the schema of the operand and $Q$ either $\forall$ or $\exists$. The rest of the expression is formed from the variables, functions, predicates and logical operators. There are some limitations. One variable cannot be used in more than one quantifier and whole expression must be closed (meaning that there is no unquantified variable and no variable is used outside of the range of the quantifier for that variable). $\forall_{x \in X}$ denotes that the quantified condition must be true for each $x$ from $PtrVal(X, g)$ and $\exists_{x \in X}$ denotes that there must be at least one $x$ in $PtrVal(X, g)$ such that the quantified condition is true. $PtrVal(X, g)$ denotes values of all nodes pointed to by $X$ in the graph $g$ and predicates of all edges pointed to by $X$ in the graph $g$.

Example: $(\exists_{p \in \text{ptr:Payment}(p > 1000)}) \land (\forall_{r \in \text{ptr:Person}} \exists_{c \in \text{ptr:Customer}}(r = c))$. This means that the graph must have a value $p$ for variable $\text{Payment}$ that is more than 1000 and that for each value of $\text{Person}$ there is the same value for $\text{Customer}$.

The formal definition is very strict, but the actual query language can be more relaxed, allowing the user to write less verbose queries as long as there is a clearly defined transformation to the form defined here.

We do not attempt to list all functions and predicates. In general, we assume that there is always a (hidden) parameter that carries the currently processed graph as its value – in the strict definition of the model it is a pair containing the whole graph and subset identifier from the namespace $\text{graph}$.

Some of the functions and predicates we would like to include are:

- **PathLength**($x, y$) that return length of a path between nodes $x$ and $y$.
- **Sum**(A), **Max**(A), **Min**(A), **Count**(A) that return the sum, maximum, minimum or number of nodes that the pointer $A$ points to.
- **IsURI**($x$), **IsLiteral**($x$) that check, whether the value of $x$ is of the specified type.
- **TypeOf**($x$) that returns the data type of the literal $x$.

A detailed proposal of the language would include several basic functions and data type conversion rules as well as extension mechanism that would allow implementations to add further functions.
B.3.2 Projection

The purpose of the projection operation is to remove unneeded parts from the data. It has one argument and the operation is specified by a set $S$ of URIs from the namespace ptr. Any reasonable set $S$ should be a subset of the schema of the argument, but it is not required. The operation removes triples from its argument. There are two versions – induced and non-induced.

The non-induced version removes all non-decoration triples that are not pointed to by a member of $S$ and all decoration triples that represent pointers not in $S$. Then all decoration triples $<g,\text{dec:contains},o>$ (graph membership) are removed if there is no triple $<p,g,o>$ where $p \in S$. Note that if we remove a decoration triple that decorated an edge, we remove the three replication triples as well.

The induced works the same as non-induced except that it does not remove edges, where both endpoints are being pointed to by members of $S$.

The Figure B.2 gives an example where $S = \{Person, Mail\}$.

B.3.3 Distinct

So far, each operation generated a multiset of graphs. The distinct operation takes one argument and eliminates all duplicates. The equivalence of the graphs does consider decoration as well, i.e. for two graphs to be equal, even the pointers in both graph must point to equal nodes and edges.

B.3.4 Joins

Joins are an important part of relational algebra and SQL. As we are trying to get close to these languages, we also introduce join operations. Join is a binary operation that produces results by making a Cartesian product of the arguments and then filters the results according to a condition. There are
special variants of the operation – outer joins (left, right and full). The basic (inner) join could be defined as a combination of cross join (i.e. Cartesian product) and selection, but the outer joins are more complex so we have decided to include “whole” join operation.

The join can be seen as an operation that generates one small RDF graph for each pair of graphs where one is from the first argument and the other from the second argument. The graphs are union-ed together and if the produced graphs fulfills the join condition, it is added to the result.

The outer joins work just like in SQL. Consider for example left join. If there is a graph \( l \) in the left argument such that there is no graph \( r \) in the right argument that \( l \cup r \) fulfill the join condition, then \( l \) is added to the result.

An example of a left join is in the Figure B.3. The left and right operands are joined by a left join on a condition \( Person1 = Person2 \) (of course, the actual condition should contain the appropriate quantifiers).

### B.3.5 Group by

In SQL, the “group by” construct is almost exclusively used with aggregation. But as our data model allows more values per “row”, we can use “group by” as a standalone operation that groups related data together. To be more specific, it joins (makes a union) graph, that have the exactly the same values for a specified set of pointers.
The Figure B.4 shows an example where two graphs are grouped by the value of the “Person” pointer into one graph.

### B.3.6 Aggregation

A very important feature in SQL are aggregation functions. An important difference between our data model and SQL is that a pointer can point to more than one value within each graph. So there are two possibilities, where aggregation functions can be used. They can either aggregate values within one graph or make aggregations over whole data. We decided to allow both.

A local aggregation has the form $fnc(ptr) \rightarrow res$ where $fnc$ is an aggregation function (min, max, sum, count, avg and “distinct” variants of sum, count and avg), $ptr$ is a pointer and $res$ is a pointer. The aggregation function is evaluated for each graph $g$ and for a result $v$, the triples $<g,gql:contains,v>$ and $<res,g,v>$ are added to the data.

The global aggregation has the form $fnc_g(fnc_l((ptr))) \rightarrow res$ where $fnc_g$ and $fnc_l$ are from the same set of functions as in the local variant. The function $fnc_l$ is used to compute aggregation over each graph and then $fnc_g$ combines these results into one final value $v$. The result contains only one graph $g$ with triples $<g,gql:contains,v>$ and $<res,g,v>$. Because the data are “destroyed” more global aggregations can be specified in one aggregation operation.

An example that demonstrates why we decided to define global aggregation like this is the following. Consider the already familiar data about people and e-mails. We can use a global aggregation $\text{max}(\text{count}(\text{Mail})) \rightarrow \text{MaxMail}$ to get the maximal number of e-mails the people have. Then, on the same source data, we run a local aggregation $\text{count}(\text{Mail}) \rightarrow \text{MailCount}$. 

117
Then we join the data on $MaxMail = MailCount$ to get information about everyone with maximal number of e-mails. Note, that since we included the aggregation functions among the function that can be used in the selection operations, we could omit the local aggregation step in the example and use $MaxMail = \text{COUNT}(Mail)$ as the join condition.

### B.3.7 Set operations

Some set operations are also present in SQL – union, union all, intersect, and minus. The equivalent of union all is obvious, it simply returns union of the two graphs (since we require that graphs are identified by globally unique identifiers, there can be no “collision”). The is no such natural equivalent for the other three operations. The problem is that in our data model, “schema” is much more relaxed concept than in SQL – multiple values of a pointer, nodes and edges without pointers or with several different pointers, etc. Should the set operations be performed only with the data in graphs without regard to pointers (and what would be the pointers of the result) or with pointers? Perhaps it would be best to let the user specify a set of pointers and the operations only consider nodes and edges with these pointers.

But this creates a completely new problem. If we make an intersection of two sets and each of them contains a graph that is different, but the values pointed to by the specified set of pointers are the same. What should get into the result? The first or the second? That would make the intersection operation non-commutative.

We decided to use the following definition for the operations ($A$ and $B$ are operands, $S$ is a set of pointers, $g[S]$ denotes a projection of graph $g$ to the set of pointers $S$):

- $A \cup_{S} B$ is a shortcut for grouping operation with columns $S$ applied to $A \cup \text{all } B$

- $A \setminus B$ are all graphs $g$ of $A$ such that there is no graph $g'$ in $B$ for which $g[S] = g'[S]$ holds. Projection $g[S]$ is either induced or non-induced – the exact version is specified by the user.

- $A \cap_{S} B$ is not included as an operation. Although we came up with several possible semantics, none of them seemed more natural than the others. This and the fact that they could all be transformed into some combination of other operations led us to the decision not to include any of them as a build-in operation.
Appendix C

TriQuery grammar

This appendix presents the grammar of the TriQuery language. It is a modification of the official XQuery grammar, so the exactly same notation is used. The modified or added rules are highlighted.

    ("encoding" StringLiteral)? Separator
[4] LibraryModule ::= ModuleDecl Prolog
[5] ModuleDecl ::= "module" "namespace" NCName "=" URILiteral Separator
[7] Setter ::= BoundarySpaceDecl | DefaultCollationDecl | BaseURIDecl | ConstructionDecl | OrderingModeDecl | EmptyOrderDecl | CopyNamespacesDecl
[8] Import ::= SchemaImport | ModuleImport
[9] Separator ::= ";"
[10] NamespaceDecl ::= "declare" "namespace" NCName "=" URILiteral
[11] BoundarySpaceDecl ::= "declare" "boundary-space"
    ("preserve" | "strip")
[12] DefaultNamespaceDecl ::= "declare" "default" ("element" | "function") "namespace" URILiteral
[13] OptionDecl ::= "declare" "option" QName StringLiteral
[14] OrderingModeDecl ::= "declare" "ordering"
    ("ordered" | "unordered")
EmptyOrderDecl ::= "declare" "default" "order" "empty"
   ("greatest" | "least")
CopyNamespacesDecl ::= "declare" "copy-namespaces"
   PreserveMode "," InheritMode
PreserveMode ::= "preserve" | "no-preserve"
InheritMode ::= "inherirt" | "no-inherit"
DefaultCollationDecl ::= "declare" "default" "collation"
   URILiteral
BaseURIDecl ::= "declare" "base-uri" URILiteral
SchemaImport ::= "import" "schema" SchemaPrefix?
   URILiteral ("at" URILiteral ("," URILiteral)*)?
SchemaPrefix ::= ("namespace" NCName "=")
   | ("default" "element" "namespace")
ModuleImport ::= "import" "module"
   ("namespace" NCName ")") URILiteral
   ("at" URILiteral ("," URILiteral)*)?
VarDecl ::= "declare" "variable" 
   ("$" QName TypeDeclaration? ("=" ExprSingle) | "external")
ConstructionDecl ::= "declare" "construction"
   ("strip" | "preserve")
FunctionDecl ::= "declare" "function" QName
   ("(" ParamList?
   ")" ("as" SequenceType)?)
   (EnclosedExpr | "external")
ParamList ::= Param ("," Param)*
Param ::= "$" QName TypeDeclaration?
EnclosedExpr ::= "{" Expr "}"
[38] OrderByClause ::= ("order" "by") | ("stable" "order" "by") OrderSpecList
[39] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[40] OrderSpec ::= ExprSingle OrderModifier
[41] OrderModifier ::= ("ascending" | "descending")?
 ("empty" ("greatest" | "least")?)
 ("collation" URLLiteral)?
[42] QuantifiedExpr ::= ("some" | "every") "$" VarName TypeDeclaration? "in" ExprSingle 
("," "$" VarName TypeDeclaration? "in" ExprSingle)*
 "satisfies" ExprSingle
[43] TypeswitchExpr ::= "typeswitch" ("(" Expr ")") CaseClause+ 
"default" ("(" VarName ")" "return" ExprSingle
[44] CaseClause ::= "case" ("(" VarName "as")"? SequenceType 
 "return" ExprSingle
[45] IfExpr ::= "if" ("(" Expr ")" "then" ExprSingle 
 "else" ExprSingle
[46] OrExpr ::= AndExpr ( "or" AndExpr )*
[47] AndExpr ::= ComparisonExpr ( "and" ComparisonExpr )*
[48] ComparisonExpr ::= RangeExpr ( (ValueComp | GeneralComp | NodeComp) RangeExpr )?
[49] RangeExpr ::= AdditiveExpr ( "to" AdditiveExpr )?
[50] AdditiveExpr ::= MultiplicativeExpr ( ("+" | "-" )
 MultiplicativeExpr )*
[51] MultiplicativeExpr ::= UnionExpr ( 
 ("*" | "div" | "idiv" | "mod") UnionExpr )*
[52] UnionExpr ::= IntersectExceptExpr ( 
 ("union" | "|") IntersectExceptExpr )*
[53] IntersectExceptExpr ::= InstanceofExpr ( 
 ("intersect" | "except") InstanceofExpr )*
[54] InstanceofExpr ::= TreatExpr 
 ( "instance" "of" SequenceType )?
[55] TreatExpr ::= CastableExpr ( "treat" "as" SequenceType )?
[56] CastableExpr ::= CastExpr ( "castable" "as" SingleType )?
[57] CastExpr ::= UnaryExpr ( "cast" "as" SingleType )?
[58] UnaryExpr ::= ("-" | "+" )* ValueExpr
[59] ValueExpr ::= ValidateExpr | PathExpr | ExtensionExpr
[60] GeneralComp ::= ":=" | ":!=" | ":<" | ":<=" | ":>" | ":>="
[61] ValueComp ::= ":eq" | ":ne" | ":lt" | ":le" | ":gt" | ":ge"
[62] NodeComp ::= ":is" | ":<<" | ":>>"
[63] ValidateExpr ::= "validate" ValidationMode? 
{" Expr "}
ValidationMode ::= "lax" | "strict"
ExtensionExpr ::= Pragma + "{ Expr? "}"-
Pragma ::= "(# S? QName (S PragmaContents)? "#)"
PragmaContents ::= (Char* - (Char* '\#') Char*)
PathExpr ::= ("/" RelativePathExpr?) |
 | ("//" RelativePathExpr) | RelativePathExpr
RelativePathExpr ::= StepExpr ("/") StepExpr*)
StepExpr ::= FilterExpr | AxisStep
AxisStep ::= (ReverseStep | ForwardStep) PredicateList
ForwardStep ::= (ForwardAxis NodeTest) | AbbrevForwardStep
ForwardAxis ::= ("child" ":")
 | ("descendant" ":")
 | ("attribute" ":")
 | ("self" ":")
 | ("descendant-or-self" ":")
 | ("following-sibling" ":")
 | ("following" ":")
AbbrevForwardStep ::= "@"? NodeTest
ReverseStep ::= (ReverseAxis NodeTest) | AbbrevReverseStep
ReverseAxis ::= ("parent" ":")
 | ("ancestor" ":")
 | ("preceeding-sibling" ":")
 | ("preceeding" ":")
 | ("ancestor-or-self" ":")
AbbrevReverseStep ::= ":..
NodeTest ::= KindTest | NameTest
NameTest ::= QName | Wildcard
Wildcard ::= ":*" | (NCName ":" ":*") | (":* ":" NCName)
FilterExpr ::= PrimaryExpr PredicateList
PredicateList ::= Predicate*
Predicate ::= ":[" Expr "]"
PrimaryExpr ::= Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall | OrderedExpr |
 | UnorderedExpr | Constructor | RecordPattern | RecordConstructor
RecordConstructor ::= AnonymousRecordConstructor | NamedRecordConstructor
AnonymousRecordConstructor ::= ":[" ExprSingle |
 | "ExprSingle )* "]"
NamedRecordConstructor ::= ":[" QName ":=" ExprSingle |
 | "ExprSingle )* "]"
RecordPattern ::= ExprSingle ( "with" QName )? "match"
"(" OneRecordPattern ( "." OneRecordPattern )*)")

[84e] OneRecordPattern ::= PatternElement+
[84f] PatternElement ::= QName | PatternURI | Literal |
"{" ExprSingle "{"}" | VarRef | "?" VarName
[85] Literal ::= NumericLiteral | StringLiteral
[86] NumericLiteral ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
[87] VarRef ::= "$" VarName ( "." QName )?
[88] VarName ::= QName
[89] ParenthesizedExpr ::= "(" Expr? ")" ( "." QName )?
[90] ContextItemExpr ::= "."
[91] OrderedExpr ::= "ordered" "{" Expr "}"
[92] UnorderedExpr ::= "unordered" "{" Expr "}"
[93] FunctionCall ::= QName
"(" (ExprSingle ("," ExprSingle)*)? ")"
( "." QName )?
[94] Constructor ::= DirectConstructor | ComputedConstructor
[95] DirectConstructor ::= DirElemConstructor |
DirCommentConstructor | DirPIConstructor
[96] DirElemConstructor ::= "<" QName DirAttributeList |
( "/" | (">" DirElemContent* ">" QName S? ">"))
[97] DirAttributeList ::= |
(S QName S? ":=" S? DirAttributeValue)?*)
[98] DirAttributeValue ::= |
("" (EscapeQuot | QuotAttrValueContent)* ":") |
("" (EscapeApos | AposAttrValueContent)* ":")
[99] QuotAttrValueContent ::= QuotAttrContentChar |
CommonContent
[100] AposAttrValueContent ::= AposAttrContentChar |
CommonContent
[101] DirElemContent ::= DirectConstructor | CDATASection |
CommonContent | ElementContentChar
[102] CommonContent ::= PredefinedEntityRef | CharRef |
"{{" | ""])" | EnclosedExpr
[103] DirCommentConstructor ::= "<!--" DirCommentContents "-->"
[104] DirCommentContents ::= |
((Char - '-') | ('-' (Char - '?')))*)
[105] DirPIConstructor ::= |
"<![" Pitarget (S DirPIContents)? "]>"
[106] DirPIContents ::= (Char* - (Char* ">" Char*))
[107] CDATASection ::= "<![CDATA[ CDATASectionContents "]]>"
CDataSectionContents ::= (Char* - (Char* ']'])>'' Char*)

ComputedConstructor ::= CompDocConstructor |
CompElemConstructor | CompAttrConstructor |
CompTextConstructor | CompCommentConstructor |
CompPIConstructor

CompDocConstructor ::= "document" "{" Expr "}" |
CompElemConstructor ::= "element" (QName | ("{" Expr "}")) |
"{" ExprExpr? "}"

ContentExpr ::= Expr

CompAttrConstructor ::= "attribute" |
(QName | ("{" Expr "}")) "{" Expr? "}" |
CompTextConstructor ::= "text" "{" Expr "}" |
CompCommentConstructor ::= "comment" "{" Expr "}" |
CompPIConstructor ::= "processing-instruction" |
(NCName | ("{" Expr "}")) "{" Expr? "}" |

SingleType ::= AtomicType "?"|

TypeDeclaration ::= "as" SequenceType |
SequenceType ::= ("empty-sequence" "(" "))|
(ItemType OccurrenceIndicator?)

OccurrenceIndicator ::= "?" | "*" | "+

ItemType ::= KindTest | ("item" "(" ")) | AtomicType |
NamedRecordType | AnonymousRecordType

NamedRecordType ::= [" QName "as" SequenceType |
( "," QName "as" SequenceType )* "]|

AnonymousRecordType ::= [" SequenceType |
( "," SequenceType )* "]|

AtomicType ::= QName

KindTest ::= DocumentTest | ElementTest | AttributeTest |
SchemaElementTest | SchemaAttributeTest | PITest |
CommentTest | TextTest | AnyKindTest

AnyKindTest ::= "node" "(" ")"

DocumentTest ::= "document-node" |
(" (ElementTest | SchemaElementTest)? ")"

TextTest ::= "text" "(" ")"

CommentTest ::= "comment" "(" ")"

PITest ::= "processing-instruction" |
(" (NCName | StringLiteral)? ")"

AttributeTest ::= "attribute" |
(" (AttributeNameOrWildcard (" ," TypeName))? ")"

AttributeNameOrWildcard ::= AttributeName |
"*"

SchemaAttributeTest ::= "schema-attribute"
"(" AttributeDeclaration ")"
[132] AttributeDeclaration ::= AttributeName
[133] ElementTest ::= "element"
"(" (ElementNameOrWildcard ("," TypeName "?"?)?)? ")"
[134] ElementNameOrWildcard ::= ElementName | "+"?
[135] SchemaElementTest ::= "schema-element"
"(" ElementDeclaration ")"
[136] ElementDeclaration ::= ElementName
[137] AttributeName ::= QName
[138] ElementName ::= QName
[139] TypeName ::= QName
[140] URLLiteral ::= StringLiteral

The following rules define the terminals:

[141] IntegerLiteral ::= Digits
[142] DecimalLiteral ::= ("." Digits) | (Digits "." [0-9]*)
[143] DoubleLiteral ::= ("." Digits) |
(Digits ("." [0-9]*)?) [E] [+-]? Digits
[144] StringLiteral ::= ("" (PredefinedEntityRef | CharRef | 
EscapeQuot | ["&"])* "") | ("" (PredefinedEntityRef | 
CharRef | EscapeApos | [~&])* "")
[144a] PatternURLLiteral ::= "</" (PredefinedEntityRef | CharRef | 
EscapeQuot | [~&])* ">
[145] PredefinedEntityRef ::= 
"&" ("lt" | "gt" | "amp" | "quot" | "apos") ";"
[146] EscapeQuot ::= ";"
[147] EscapeApos ::= ";"
[148] ElementContentChar ::= Char - [{}<&]
[149] QuotAttrContentChar ::= Char - ["{}<&]
[150] AposAttrContentChar ::= Char - [‘{}<&]
[151] Comment ::= "(" (CommentContents | Comment)* ")"
[152] PITarget ::= [http://www.w3.org/TR/REC-xml#NT-PITarget]
[153] CharRef ::= [http://www.w3.org/TR/REC-xml#NT-CharRef]
[154] QName ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]
[155] NCName ::= [http://www.w3.org/TR/REC-xml-names/#NT-NCName]
[156] S ::= [http://www.w3.org/TR/REC-xml#NT-S] XML
[157] Char ::= [http://www.w3.org/TR/REC-xml#NT-Char] XML

Auxiliary (used in terminals definition):

[158] Digits ::= [0-9]+
[159] CommentContents ::= (Char* - (Char* ('(:' | ':') Char*))

125