Abstract

As part of the Semantic Web initiative, the *Resource Description Framework* (RDF) is gaining momentum as a format for storing data, particularly metadata. The *SPARQL Protocol and RDF Query Language* is a SQL-like query language, recommended by W3C for querying RDF data.

Fast is exploring the possibilities of supporting storage and querying of RDF data in their Mars search engine. As part of this, a SPARQL parser has been created for the Microsoft .NET Framework.

This thesis proposes a solution for efficiently storing and retrieving RDF data in Mars, based on decomposition and B+ Tree indexing. Further, a method for transforming SPARQL queries into Mars algebra is described. Finally, the implementation of a prototype is discussed.

The prototype has been developed in collaboration with Fast and has required customized indexing in Mars. Some deviations from the proposed solution were made in order to create a working prototype within the available time frame. The focus has been on exploring possibilities, and performance has thus not been a priority, neither in indexing nor in evaluation.

The prototype is able to evaluate a wide range of typical SPARQL queries. Such queries include *SELECT* queries against a single, default data set, specifying multiple triple patterns in the *WHERE* clause. All solution modifiers are supported, including ordering, variable projection and requiring distinct solutions, with only minor nonconformities. No query optimizations are employed, however.

Preface

This thesis was written at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) during the spring semester of 2009. The assignment was given by and written for Fast and the iAD Research Centre.

We would like to thank our supervisors, professor Svein Erik Bratsberg at NTNU and Øystein Torbjørnsen at Fast, for feedback and guidance during the course of this project.

Trondheim, June 2009

Ole Petter Bang

Tormod Fjeldskår

Contents

List of Tables

List of Figures

List of Definitions

Chapter 1

Introduction

The *Resource Description Framework* (RDF) [1], originating from the *Semantic Web* initiative, has grown popular for representing various kinds of data, particularly metadata. The *SPARQL Protocol and RDF Query Language* (SPARQL) [2] is the query language proposed by the *RDF Data Access Working Group* for querying RDF data sets.

Information Access Disruptions (iAD) is a constellation between Fast Search & Transfer (Fast), two Norwegian enterprises, the Norwegian University of Science and Technology (NTNU) and several other universities. iAD "[...] targets core research for next generation precision, analytics and scale in the information access domain." [3] As part of this research, it is desirable to be able to query several different types of information from various data sources, including RDF using SPARQL.

Mars is the next generation search engine by Fast, combining database and search engine technology, targeting enterprises. Mars is a key component in the iAD search technology, focusing on creating "[...] schema agnostic indexing services fusing structured, unstructured and multimedia content in precision, analytics and scale optimized information access services." [3]

In an academic project [4] preceding this master thesis, a parser for the SPARQL query language was created, written in C# targeting Microsoft's .NET Framework. This parser was the first step towards being able to query RDF data sources in Mars using SPARQL.

The focus of this thesis is to explore the possibilities of extending the capabilities of Mars, in order to support storage and retrieval of RDF data using the SPARQL query language. This being the focus, analytical in-depth comparisons with similar systems is considered out of the scope. The tight integration between storage model and query engine achieved in this approach differs notably from the majority of similar solutions. Thus, establishing a common scale of measure allowing for useful, in-depth comparisons with other solutions is a challenge.

The goals of this work are to:

- **Explore possible storage alternatives for RDF data**.
- **Compile SPARQL queries into Mars query algebra**.
- **Implement a prototype Mars component**.

Besides this thesis, part of the delivery is a prototype Mars component for evaluating simple queries, utilizing the SPARQL parser originating from the aforementioned academic project.

Chapter 2 presents background information on RDF, SPARQL and the Mars search engine. This is not a complete presentation, but focuses on introducing essential concepts in order to establish the context of the thesis. Chapter 3 is an extract of the aforementioned parser project, introducing the concepts of parsing and abstract syntax trees as well as presenting the SPARQL parser used by the prototype. Chapter 4 discusses various approaches to storing RDF data and is concluded with a justified proposal for a storage model for Mars. In Chapter 5, the method of transforming SPARQL queries into operator graphs evaluable by Mars is explained while Chapter 6 discusses a selection of interesting topics with regards to implementation details. Finally, the thesis is concluded by Chapter 7 discussing the results and Chapter 8 which states the conclusion and outlines further work based on the prototype.

Throughout this thesis, a fictitious music metadatabase is used as the basis for most examples and figures. This metadatabase is loosely based on the MusicBrainz[1] project which contains a comprehensive collection of metadata about artists/bands and their releases.

Appendix A contains the *NodeBase* class, from which all classes used to represent the abstract syntax trees (ASTs) descend. This abstract class implements the *INode* interface and thus the *Accept* methods that realize the Visitor pattern used for traversing the ASTs. In addition, the class implements functionality for cloning nodes, used when resolving syntactic sugar.

Appendix B contains the *WhereClauseTransformer* class, which is responsible for transforming *WHERE* clause constructs from the abstract syntax trees into their corresponding SPARQL algebra representations.

Appendix C contains information about the enclosed ZIP archive which is also part of the delivery. This archive contains the produced source code as well as a digital copy of this thesis. A digital copy of the unpublished document "Developing a SPARQL parser for .NET" [4], cited in this thesis, is also included.

[1] Freely available from `http://musicbrainz.org/`

Chapter 2

Background

RDF has quite a versatile data model, suitable for representing data in a number of applications. Using schemas, application-specific vocabularies may be defined. SPARQL is a comprehensive query language designed specifically for querying RDF data. SPARQL includes several common query language features as well as introducing new ones specific to the RDF data model.

The Mars search engine combines database and search engine technology. While the engine does not follow the RDBMS paradigm, relational structures may still be emulated.

2.1 RDF

The Resource Description Framework (RDF) [1] is a specification for describing resources on the Web, usually in context of the Semantic Web [5]. Few restrictions apply to the data describing the resources, making RDF suitable for a number of applications.

RDF data constitutes a directed, labeled graph, usually materialized as triples, consisting of a subject, a predicate and an object. Triple elements may all be identified using URIs[1]. Whether or not a URI actually resolves to a retrievable Web resource, however, is of little or no interest as the sole purpose of the URI is to serve as a unique identifier.

2.1.1 Data Model

The RDF data model is based on triple entities, consisting of a subject, a predicate and an object. The subject and object each correspond to a node,

[1] Strictly speaking, in the context of RDF and SPARQL, the term URI always refers to Internationalized Resource Identifiers (IRIs), a generalization of the Uniform Resource Identifier (URI), which may contain characters form the Universal Character Set (Unicode/ISO 10646).

and the predicate corresponds to the directed arc from a subject to an object node, in the directed, labeled *RDF graph* formed by the data.

A subject node is identified by a URI, an object node either by a URI or by a literal, and a predicate arc by a URI as well. A subject or object node may, however, be used without any identification at all, in which case it is referred to as a *blank node*. Blank nodes serve as anonymous resources that are not uniquely identifiable. In order to distinguish blank nodes from one another, they are assigned labels which are only required to be unique within the context where the triples reside.

Using blank nodes, an RDF graph may express properties of a resource typically not identifiable by a URI. Blank nodes are consequently ideal for representing structured property values, that is, properties like dates and addresses that can be decomposed into smaller parts.

When representing structured property values, one could make each property component properties of the subject. A better solution, however, would be to introduce an intermediary node representing the structured property value and make each property component a property of that node. Such an intermediary node having its value represented by its properties typically has no need for any identification and could very well be a blank node.

Figure 2.1 shows a sample RDF graph illustrating the concepts of the RDF data model. Arrows represent predicates and point from a subject to an object. Note that a single resource can appear as subject of some predicates and object of other predicates. The figure depicts some properties of the rock band "U2". A blank node labeled *collection* is used to represent the list of albums created by the band, which is a structured property value. Further, this blank node has edges to every album, which consequently has their own properties.

Formally, an RDF triple may be defined as follows:

Definition 2.1 (RDF Triple). *Given a set of URI references $\mathcal{R}$, a set of blank nodes $\mathcal{B}$, and a set of literals $\mathcal{L}$, a triple $(s, p, o) \in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{R} \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L})$ is called an RDF triple.*

RDF has no built-in set of data types[2], making literals untyped unless otherwise specified. Literal data types may be specified using data type URIs. As with resource URIs, the data type URI only serves as a unique identifier for the data type. It is still entirely up to the application interpreting the RDF data to decide which semantics to employ.

[2]According to [1] RDF has indeed a built-in data type *http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral* for representing XML literals. It is still up to the application in question, however, to determine how to interpret the RDF data.

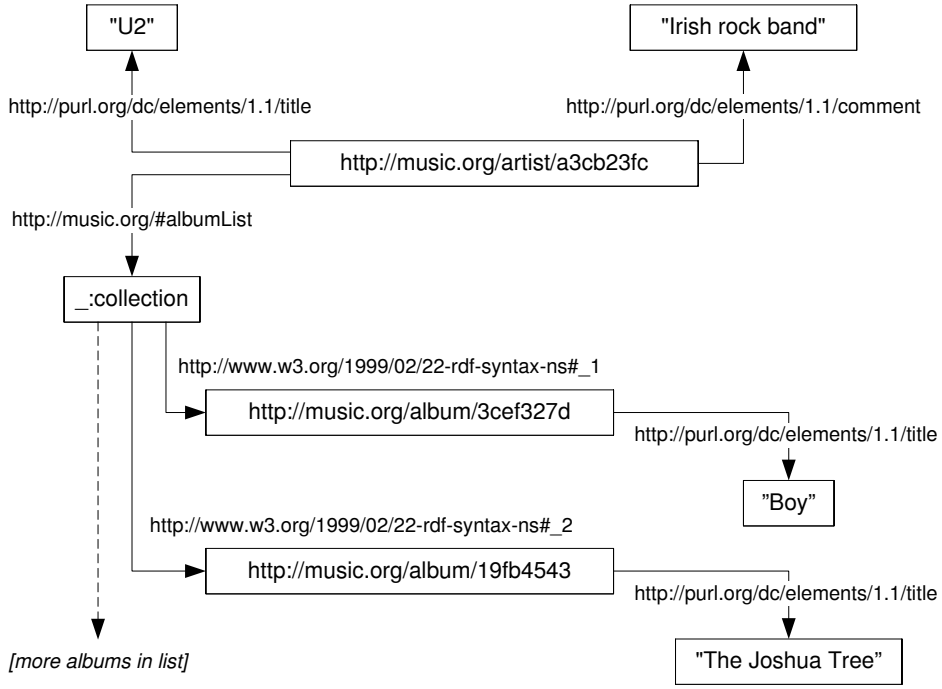


Figure 2.1: RDF Sample Graph

2.1.2 Schema

RDF is suitable for describing resources in a number of applications. Basically, every single resource that can be described by a set of properties and their corresponding property values may be represented using RDF.

Some resources have properties with several property values, like a book having several authors. Also, some properties may have a range of possible, predefined values. RDF defines a number of vocabularies for representing such properties.

In general, application-specific vocabularies may be defined using RDF Schema [6]. RDF Schema itself defines no vocabularies; it simply provides a way of defining them. The vocabularies are defined using RDF models, analogous to how XML Schema vocabularies are defined using XML.

Using RDF Schema, application-specific classes of resources and properties defining the resources may be defined. A resource may be defined as being a class, and further as being a subclass of some other class. A resource may also be defined as a property and its allowed values may be constrained by a set of classes or by a typed literal. As with classes, a property may also be a sub-property of another property. Finally, a property may be defined as being in the domain of one or more classes, meaning it is a property of those classes.

There are notable differences separating the class hierarchy represented

by an RDF Schema vocabulary and the class hierarchy of a typical object-oriented programming model. Properties are independent of class definitions and are, by default, defined in a global scope. Properties not defined as being in the domain of any classes may be used to describe instances of any class. The definition of such a property, however, is still independent of the class instances it is used to describe. Changes made to the property apply to all class instances it is used to describe rather than applying only for a specific class instance.

Figure 2.2 shows a sample UML model and the corresponding RDF Schema, serialized using Terse RDF Triple Language (Turtle) [7]. In the sample, each line represents an RDF triple. The sample utilizes four prefixes for reducing the triple element identifiers; an example prefix, *ex*, the RDF and RDF Schema prefixes, *rdf* and *rdfs*, and the XML Schema prefix, *xsd*.

The URI prefixes used in the sample schema and throughout this thesis are defined as follows:

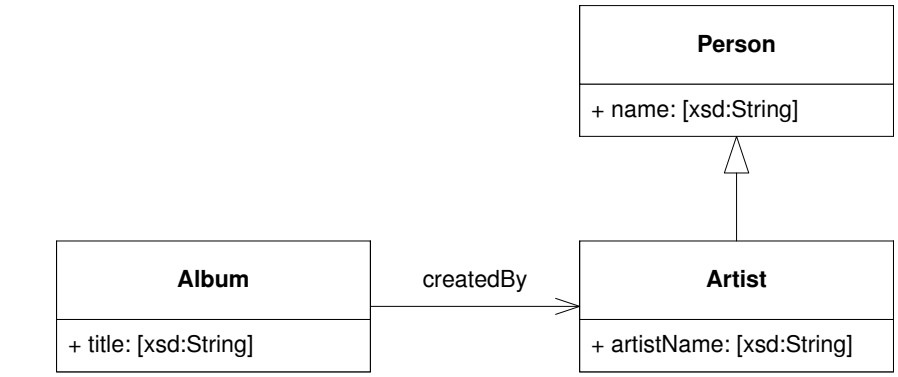
- *rdf*: http://www.w3.org/1999/02/22-rdf-syntax-ns#
- *rdfs*: http://www.w3.org/2000/01/rdf-schema#
- *xsd*: http://www.w3.org/2001/XMLSchema#

The schema starts off by defining three resources as being of the type *rdfs:Class*, which is the RDF Schema class type. Next, the *Artist* class is defined as being a subclass of the *Person* class, as depicted in the UML model. Further, the XML Schema data type *String* is defined as an RDF data type. All resources used as class properties are then defined as such, followed by data type definitions for each property using the *rdf:range* predicate. Finally, a domain is defined for each property, effectively attaching its usage to the specified class.

As stated earlier regarding resource and data type URIs, it is entirely up to the application to decide how to interpret the RDF Schema as a whole. The schema is, however, defining the semantics for the RDF data it is used to describe, enabling inference of relationships between entities. This is useful when interpreting the RDF data, especially when querying the data, using query languages like SPARQL.

2.2 SPARQL

The SPARQL Query Language for RDF [2] is designed to express queries against RDF data, based on the use cases and requirements identified by the RDF Data Access Working Group [8]. Queries may span several RDF



UML

RDF

```
ex:Person       rdf:Type          rdfs:Class .
ex:Artist       rdf:Type          rdfs:Class .
ex:Album        rdf:Type          rdfs:Class .

ex:Artist       rdfs:subClassOf   ex:Person .

xsd:String      rdf:Type          rdfs:Datatype .

ex:name         rdf:Type          rdf:Property .
ex:artistName   rdf:Type          rdf:Property .
ex:createdBy    rdf:Type          rdf:Property .
ex:title        rdf:Type          rdf:Property .

ex:name         rdfs:range        xsd:String .
ex:artistName   rdfs:range        xsd:String .
ex:createdBy    rdfs:range        ex:Artist .
ex:title        rdfs:range        xsd:String .

ex:name         rdfs:domain       ex:Person .
ex:artistName   rdfs:domain       ex:Artist .
ex:createdBy    rdfs:domain       ex:Album .
ex:title        rdfs:domain       ex:Album .
```

Figure 2.2: RDF Schema Sample

graphs and match both required and optional graph patterns using conjunctions and disjunctions as well as value testing and constraining. Syntactically, SPARQL is based on the Terse RDF Triple Language [7], and shares many of the same language elements.

2.2.1 Graph Pattern Matching

A SPARQL query specifies a graph pattern which is matched against one or more RDF graphs, forming an RDF data set. The data set always contains one default graph and zero or more named graphs. The *FROM* keyword is used to specify the RDF graphs to be merged into the default graph, and the *FROM NAMED* keyword is used to specify the named graphs. The *GRAPH* keyword is used to select the currently active graph used when matching a graph pattern. If no graph is specified, the default graph is used.

A *basic graph pattern* contains a set of triples forming a graph pattern, in which each element may be a variable instead of an URI or a literal. Determining whether a graph pattern matches some subgraph, involves first letting all pattern variables constitute the corresponding RDF triple elements from the subgraph. If the resultant graph is equivalent to the subgraph then the graph pattern matches the subgraph [2].

Graph patterns may be constructed as a combination of other graph patterns. Using *group graph patterns*, multiple graph patterns may be specified that all have to match. Also, *optional graph patterns* and *alternative graph patterns* may be used to specify optional graph patterns and a set of alternative patterns, respectively.

Graph patterns may contain blank nodes in the same way RDF triples do. Blank node labels are scoped within a basic graph pattern. Thus, reusing labels across basic graph patterns is prohibited.

Graph pattern matching, according to the SPARQL specification, only supports simple entailment. Thus, RDF Schema is not necessarily supported by implementations following the SPARQL specification, as described in Section 2.2.5.

2.2.2 Matching and Constraints

SPARQL is aware of the data types present in the RDF data. Integers, decimals, doubles and booleans may be specified in a shortened form which is translated into its full form during interpretation. The shortened form for integers is simply their digits, without enclosing quotation marks. The full form representation for an integer 42, however, would be `"42"^^xsd:integer`. Other data-typed literals have to be specified in their full form.

During graph pattern matching, literal values may be tested using specified *filters* in addition to simply comparing URIs and potentially data-

```
@prefix a:   <http://music.org/artist/> .
@prefix dc:  <http://purl.org/dc/elements/1.1/> .

a:a3cb23fc  dc:comment  "Irish rock band"@en .
a:a3cb23fc  dc:comment  "Irsk rockegruppe"@no .
a:a3cb23fc  dc:comment  "Irsk flokk av rockemusikantar"@no-NN .
```

Figure 2.3: RDF Sample Data

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?comment
 WHERE { ?s dc:comment "Irish rock band"@en .
         ?s dc:comment ?comment .
         FILTER langMatches( lang(?comment), "no" ) }
```

Figure 2.4: SPARQL Sample Query

typed literal values. Strings may be tested using regular expressions, and numeric values may be compared arithmetically. SPARQL offers filter usage for several XML Schema data types, including *xsd:boolean* and *xsd:dateTime*. SPARQL also defines a set of filter functions for testing whether a query variable is bound or whether triple elements are URIs, blank nodes, literals and so on.

Figures 2.3 and 2.4 show some sample RDF data and a sample SPARQL query. The RDF data simply consists of a set of comments for the rock band featured in Figure 2.1, using language tags for expressing different language versions of the comments. The SPARQL query specifies a single *group graph pattern* consisting of two *basic graph patterns* and a *filter*. The first *basic graph pattern* specifies that the comment of some variable node *?s* should be *"Irish rock band"@en*, whereas the second one simply defines the *?comment* variable as a mapping to the comment property values for the same variable node. Finally, the *filter* contains an expression specifying that the comment property value matching the *?comment* variable should have an *no* language tag.

When the sample query is evaluated, the first two basic graph patterns are equi-joined on the *?s* variable. If no common variables existed, the result would be the Cartesian product. Next, the results from the join are filtered according to the filter condition, and finally a projection is performed. Altogether, the query finds all Norwegian comments for resources having an English comment of "Irish rock band".

2.2.3 Query Results

A query may have zero, one or multiple solutions, depending on whether the query graph pattern matches the RDF graph(s). Also, the solutions may

not have any specific order. A single solution represents one way in which the query variables constitute the corresponding RDF triple elements. As all possible solutions are given, queries including several variables may potentially result in quite large result sets.

A result set may contain blank nodes, whose labels are scoped to the result set. Two blank nodes sharing the same label, are in fact the same blank node. A blank node may, however, be labeled differently in different result sets. Thus, one should not expect blank node labels in a result set to refer to particular blank nodes in the RDF data.

SPARQL defines a set of sequence modifiers, or operators, altering the solution sequence in some way, most of which are familiar from SQL-like query languages. The *order*, *projection*, *distinct*, *offset* and *limit* modifiers alter the order of solution elements, selects the variables to be returned, ensures element uniqueness, sets the offset for the solution sequence and limits the number of elements returned, respectively. In addition, there exists a *reduced* modifier that simply *allows for* element uniqueness without enforcing it.

Regarding the *order* modifier, SPARQL has a set of rules for how the ordering should be performed. The different classes of result values are ordered relative to each other as follows, from lowest to highest. [2]

1. No value assigned to the variable.
2. Blank nodes
3. Resource URIs
4. RDF literals
5. Data-typed RDF literals

This means that, in an ascending order, the resource URI *"http://z.com/"* appears before the RDF literal *"http://a.com/"*.

Values within the first four classes are sorted lexicographically. Data-typed RDF literals are sorted according to the semantics of the data type, if such semantics exist. Otherwise, lexicographic ordering is used.

2.2.4 Alternative Query Forms

In addition to regular *SELECT* queries described so far, SPARQL defines three additional query forms. In this thesis, however, the primary focus will be directed at *SELECT* queries.

CONSTRUCT queries construct an RDF graph by substituting variables in a set of triple templates forming a graph template. *ASK* queries simply return a boolean indicating whether or not a query graph pattern matched. Finally, *DESCRIBE* queries return an RDF graph describing the resources found.

2.2.5 Lack of Schema Support

The SPARQL specification describes how basic graph patterns should be matched using simple entailment, and simply states conditions that solutions supporting more elaborate forms of entailment should satisfy. The query language itself, however, is expressive enough as is, and the actual level of entailment supported only really depends on the specific implementation.

The SPARQL keyword *a* is syntactic sugar for the predicate *rdf:Type* from the RDF prefix *rdf*, and is used to describe the data type of a resource, as shown in Figure 2.2. In a standard implementation supporting only simple entailment, subjects and objects related through the *rdf:Type* predicate may be identified directly as they are defined in the very same RDF triple. Inferring relations in the class hierarchy, however, like that an instance of the *Artist* class is also an instance of the *Person* class in the schema sample, is not possible without implementing a more elaborate form of entailment, interpreting the RDF schema and evaluating the semantics it defines.

As iAD focuses on schema-agnostic indexing services, the matter of schema support in SPARQL will not be discussed any further [3].

2.2.6 Base and Prefixes

URIs are essential in RDF and SPARQL. URIs tend to be very verbose. To compensate for this, the SPARQL specification defines the *BASE* and *PREFIX* keywords. A query may declare at most one base URI and an arbitrary amount of named prefix URIs. The base URI, if specified, is used as a prefix for all relative URIs used in the query. Prefix URIs may be referenced by their names when specifying URIs in the query. This reduces verbosity and redundancy when the very same prefix is used for several URIs.

As an example, a music metadatabase may identify artists by the URI pattern *http://music.org/artist/artist-id*. All these identifiers share a common prefix. By specifying this prefix as the base URI in the prologue of the SPARQL query, any references to relative URIs will be treated as suffixes to this URI. Figure 2.5 shows two equivalent queries, illustrating the use of a base specification.

Similarly, the database may use *http://music.org/album/album-id* as the URI pattern to identify albums. Specifying two bases would introduce a conflict. Instead, the prefixes can be specified as named prefixes in the query prologue using the *PREFIX* keyword. Figure 2.6 shows two equivalent queries, illustrating the use of prefix specifications.

Base and prefix specifications may also be combined. This allows for relative prefixes, interpreted as suffixes to the base declaration. This is illustrated in the query in Figure 2.7, which is equivalent to the queries in Figure 2.6.

```
BASE <http://music.org/artist/>
SELECT * { <a3cb23fc> ?p "U2" }

    ⇳

SELECT * { <http://music.org/artist/a3cb23fc>
           ?p
           "U2" }
```

Figure 2.5: Specifying a Base to Decrease Verbosity

```
PREFIX artist: <http://music.org/artist/>
PREFIX album: <http://music.org/album/>
SELECT * { artist:a3cb23fc ?p album:3cef327d }

    ⇳

SELECT * { <http://music.org/artist/a3cb23fc>
           ?p
           <http://music.org/album/3cef327d> }
```

Figure 2.6: Specifying Named Prefixes to Decrease Verbosity

```
BASE <http://music.org/>
PREFIX artist: <artist/>
PREFIX album: <album/>
SELECT * { artist:a3cb23fc ?p album:3cef327d }
```

Figure 2.7: Combining Base URI With Named Prefixes

```
(:a :b :c)

    ⇓

_:b1 rdf:first :a .
_:b1 rdf:rest _:b2 .
_:b2 rdf:first :b .
_:b2 rdf:rest _:b3 .
_:b3 rdf:first :c .
_:b3 rdf:rest rdf:nil
```

Figure 2.8: SPARQL Syntactic Sugar for Lists

2.2.7 Syntactic Sugar

The SPARQL grammar defines several syntactic sugar constructs, being a special shorthand notation of expressing constructs that are typically somewhat cumbersome to express using the full notation. As mentioned, SPARQL supports a handful of RDF data types in shortened form, often referred to as syntactic sugar. For instance, numeric data types like integers, decimals and doubles may be expressed directly in their mathematical notation, having their full data types appended during interpretation.

Lists

In addition to syntactic sugar for data types, syntactic sugar for constructs like lists and triples sharing subjects also exists. Figure 2.8 shows a simple list consisting of three elements expressed in SPARQL using syntactic sugar and the corresponding core language representation. The core language representation is a linked list of blank nodes, one per list element, *_:b1*, *_:b2* and *_:b3*. Each blank node points to the corresponding list element and to the next element's blank node. The blank node representing the first element also represents the entire list. Finally, the list is terminated by the *rdf:nil* resource.

Shared subject triple lists

Triples sharing the same subject may be expressed using syntactic sugar. Figure 2.9 shows six triples all sharing the same subject and some even the same predicate, along with the corresponding core language representation. The expression always starts off with a complete triple followed by a series of partial triples. Partial triples starting with a comma share both the subject and predicate with the preceding triple, whereas those starting with a semi-colon share only the subject.

```
:a :b :c , :d , :e ; :f :g ; :h :i , :j

    ⇓

:a :b :c .
:a :b :d .
:a :b :e .
:a :f :g .
:a :h :i .
:a :h :j
```

Figure 2.9: SPARQL Syntactic Sugar for Triples Sharing the Same Subject.

```
[:b :c , :d , :e ; :f :g ; :h :i , :j]

    ⇓

_:b1 :b :c .
_:b1 :b :d .
_:b1 :b :e .
_:b1 :f :g .
_:b1 :h :i .
_:b1 :h :j
```

Figure 2.10: SPARQL Syntactic Sugar for Blank Node Property Lists

Blank node property lists

A variant of the triples sharing the same subject is blank node property lists. A blank node representing the property list itself becomes the shared subject for the list items, and the explicit shared subject declaration is omitted. If no item list is specified, however, only the blank node representing the property list is produced. Since a property list ends up being represented by a blank node, the list may be used as subject or object in triples, or as elements in lists and other constructs. Figure 2.10 shows a property list closely resembling the shared subject list in Figure 2.9, along with the core language representation. The shared subject is omitted from the list, however, and an induced blank node representing the blank node property list becomes the shared subject.

Combined Sample

Syntactic sugar constructs may of course be combined, and the different combinations quickly become rather complicated. For instance, lists may be used as a subject or object triple element, lists may contain other lists as well as blank node property lists.

Figure 2.11 shows a query combining syntactic sugar constructs, along with the core language representation. The order of the triples is insignifi-

```
:a :b :c ; :d [:e :f; :g (:h :i :j) , :k] , :l

    ⇓

_:b1 rdf:first :h .
_:b1 rdf:rest _:b2 .
_:b2 rdf:first :i .
_:b2 rdf:rest _:b3 .
_:b3 rdf:first :j .
_:b3 rdf:rest rdf:nil .

_:b4 :e :f .
_:b4 :g :_b1 .
_:b4 :g :k .

:a :b :c .
:a :d _:b4 .
:a :d :l
```

Figure 2.11: SPARQL Combined Syntactic Sugar

cant, and has been chosen to promote readability. The inner-most construct transformations are listed first, starting off with the list containing the elements *:h*, *:i* and *:j*, identified by the blank node *_:b1*. Following is the blank node property list, identified by the induced blank node *_:b4*, referencing the inner list. Finally is the shared subject triples list, referencing the blank node property list.

2.3 The Mars Search Engine

This section discusses the peculiarities of Mars, particularly its operators and index types. Graphs of Mars operators are a commonly used representation of Mars algebra.

2.3.1 Query Evaluation

Query evaluations in Mars operate on directed acyclic graphs, where each node in the graph is an operator. For example, searching for *"ntnu trondheim"* might produce the operator graph shown in Figure 2.12. The flow between nodes consists of records containing a document identifier and other fields, such as position within the document and the scope within the document where a term occurs.

A number of operators exist for fetching records from different indices. The *Lookup* operator examines an inverted index for records meeting a certain search criteria, while the *Scan* operator reads the records of an index sequentially. Yet another index access operator exists, called *Value*, which merges incoming records with records from a given index.

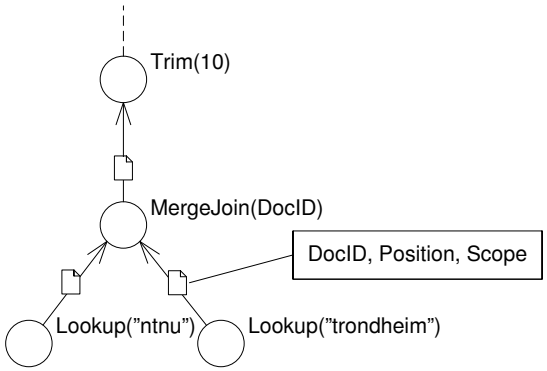


Figure 2.12: A Mars Operator Graph for the Search Term "ntnu trondheim"

2.3.2 Important Mars Operators

When creating operator graphs for Mars, a large amount of operators are available. The most important of these are described briefly below. The *input* for an operator describes the operator parameters and any incoming record sets, while the *output* for an operator describes the outgoing record set.

Lookup

The *Lookup* operator performs a direct lookup in an inverted index. By specifying an index, a search term and optionally a scope, records from the index meeting the criteria will flow to the operator's parent.

Input: Index name, search term and optionally scope parameters.
Output: Records from the index meeting the criteria of the input parameters.

Select

The *Select* operator investigates records flowing into it, and only relays records meeting specified criteria. This allows for filtering of records.

Input: Filter parameter, and records from the preceding operator(s).
Output: Records from the input meeting the filter criteria.

Map

The *Map* operator takes records from its input flow and maps their data to the output records based on a set of parameters. Record fields are redefined, with fields from the incoming records optionally retained. New

fields may be introduced as well, whose values are defined by expressions that may contain other fields.

Input: Mapping parameters, and records from a single, preceding operator.

Output: Records from the input, with fields reorganized according to the input parameters.

MergeJoin

The *MergeJoin* operator performs multiway equi-joins on all its inputs using the merge join algorithm. A *prefix* parameter specifies the number of initial record fields to be used as the join key. Note that when the prefix is set to 0, the output is the Cartesian product of its inputs.

Input: Join prefix parameter, and records from the preceding operators.

Output: Records from the inputs, joined based on the join prefix parameter.

Trim

The *Trim* operator limits the amount of records from its input to its output. An *offset* parameter specifies the amount of records to skip from the input before producing output, while a *hits* parameter specifies the maximum number of records to include in the output.

Input: Offset and hits parameters, and records from a single, preceding operator.

Output: After discarding the specified amount of records according to the offset parameter, the amount of records specified by the hits parameter are passed directly from input to output.

Sort

The *Sort* operator produces a sorted output from its input. The fields to sort on are specified explicitly. It is also possible to specify *trim* and *offset* parameters directly on the *Sort* operator, rather than relying on the Trim operator.

Input: A list of sort definitions, and records from a single, preceding operator.

Output: Records from the input ordered according to the sort definitions.

ProjectDistinct

The *ProjectDistinct* operator will, given a sorted input, produce only distinct records in its output. A *prefix* parameter specifies the number of initial record fields that have to be equal in order for a record to be considered a duplicate.

Input: A prefix parameter, and records sorted on the prefix from a single, preceding operator.
Output: Records from the input, with duplicates removed based on the prefix.

Union

The *Union* operator will produce the union of the records from its inputs, optionally eliminating any duplicates.

Input: A union type parameter, and records from the preceding operators.
Output: The union of the records from the inputs.

2.3.3 Indices in Mars

The *Lookup* and *Scan* operators operate on *inverted indices*. Each word of the lexicon translates into a list of records identifying the document and location within the document where the word appears. For structured documents, structure elements may also be part of the lexicon. Four general types of index records exist:

- DocID, Position, Scope
- DocID, Position
- DocID, Value, Scope
- DocID, Value

DocID uniquely identifies the document containing the word. *Position* is the location of the word within the document, counting whole words. *Scope* is used in structured documents to denote where in the structure the word appears, as shown in Figure 2.13. *Value* is used when the *Lookup* operator is used to search for structure elements rather than words. In this case, the *Value* part of the record denotes the content of the structure element.

Assuming a [*DocID, Position, Scope*] based index on the example XML document from Figure 2.13, some example entries in the inverted index, assuming DocID 1, are:

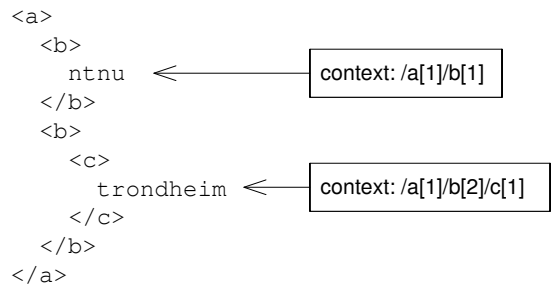


Figure 2.13: Context of Terms in a Structured Document

- ntnu → 1, 1, /a[1]/b[1]
- trondheim → 1, 2, /a[1]/b[2]/c[1]
- $a → 1, 1, /a[1]
- $b → 1, 1, /a[1]/b[1] , 1, 1, /a[1]/b[2]

$x denotes the structural element named *x*. Similarly, a [*DocID, Value*] based index on the same document would include these entries:

- $b → 1, “ntnu” , 1, “<c>trondheim</c>”
- $c → 1, “trondheim”

While Mars does not follow the RDBMS paradigm, relational structures can still be emulated. A table with *n* fields can be represented by a set of XML documents containing one row each, where each non-NULL field is represented by an XML element. Restoring the table is a matter of performing *n* lookup operations, fetching the elements corresponding to the fields, and joining the results from each lookup-operation on the *DocID* field. The result of this join operation is a set of [DocID, Field1, . . . , FieldN] records, one for each row.

Chapter 3

Parsing and Syntax Trees

The process of parsing involves analyzing tokens from some source lexically, syntactically and semantically based on a formally specified grammar. The result of parsing is typically a grammatical structure called an *abstract syntax tree* (AST). The AST may be translated into executable code, relational algebra and so on. Figure 3.1 shows the data flow of a typical parser.

3.1 Theory

The SharQL Parser was developed as a part of an academic project at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) in the autumn of 2008 [4]. In the parser project, the MPLex and MPPG scanner and parser generators that are part of the Microsoft Managed Babel package [9], were used to create a bottom-up, LALR parser based on the W3C SPARQL specification [2].

The output of the parser is an abstract syntax tree representing the structure of the parsed SPARQL query. This AST can then be interpreted or transformed into other representations. The nodes and structure of the AST closely resemble the structure of the SPARQL grammar as described in [2].

3.1.1 Context-Free Grammars

Grammars are used to specify the syntax of languages, like the SPARQL grammar specifies the syntax of the SPARQL query language. The grammar used to specify the SPARQL query language is called a *context-free grammar*, reflecting the characteristics of the language.

A context-free grammar consists of a set of *terminal* symbols, a set of *nonterminals*, a set of *productions* and a designation of one nonterminal as the *start* symbol. Terminal symbols are literal strings defining elementary symbols of the language, often referred to as *tokens*. Nonterminals each

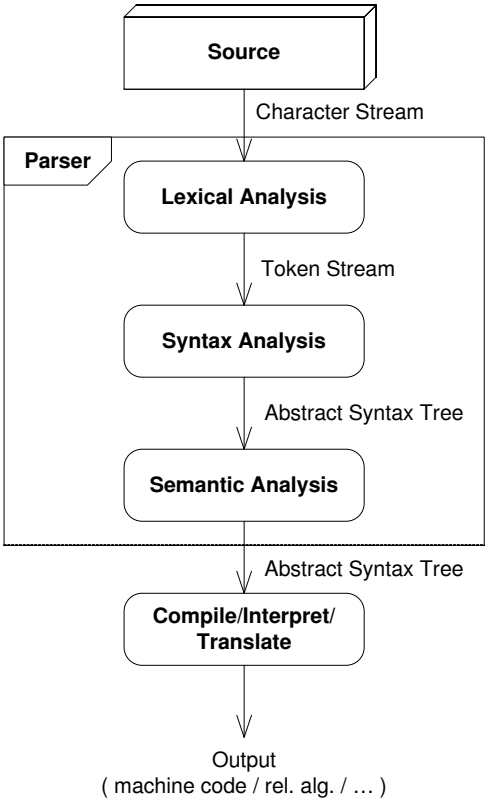


Figure 3.1: Parser Data Flow

```
list1          ::= list_item*
list2          ::= list_item+
list_item      ::= OPTIONAL_PART? REQUIRED_PART

  ⇓

list1          ::= list_tail
list2          ::= list_item list_tail
list_tail      ::= list_item list_tail | ϵ
list_item      ::= REQUIRED_PART | OPTIONAL_PART REQUIRED_PART

alternatively :
list_item      ::= optional_part REQUIRED_PART
optional_part  ::= OPTIONAL_PART | ϵ
```

Figure 3.2: EBNF Sample Rules and Corresponding BNF Representation

represent a set of strings of terminals. A production is on the form $V \rightarrow w$, where V is a nonterminal and w is a sequence of terminals and/or nonterminals. [10]

Backus-Naur Form (BNF) is a syntactic metalanguage for formally defining the syntax of formal definitions like programming language definitions. BNF is limited, however, to defining context-free grammars. *Extended Backus-Naur Form* (EBNF) is an extended version of BNF, introducing a set of extensions that ease writing and improve readability. [11]

The extensions introduced by EBNF most relevant to the SPARQL grammar are the modifiers **?** (zero-or-one), ***** (zero-or-more) and **+** (one-or-more). Consider the productions in Figure 3.2. Two types of lists are specified, one that can be empty, and one that can not be empty. Further, list items have an optional part and a required part. The `::=` operator corresponds to the $\rightarrow$ operator used to denote productions in context-free grammar definitions.

BNF lacks the aforementioned modifiers. Thus, to describe the same grammar using BNF, some translation is needed. The ***** and **+** modifiers from EBNF are achieved recursively in BNF, shown in the *list_tail* rule. The **?** modifier is achieved either by enumerating all legal combinations of required and optional elements, or by introducing a new non-terminal which may produce the optional part or the empty string (denoted by ϵ).

3.1.2 Common Parsing Techniques

There are two common types of parsing techniques; *top-down* and *bottom-up* parsing [10]. Top-down parsing consists of constructing a parse tree starting from the root and creating the subordinate nodes in a depth-first manner. Top-down parsing involves finding the left-most derivation of the input string, and the class of grammars represented by such parsers is often called *LL(k)*, where *k* is the number of lookahead symbols used. Two

common top-down parsing techniques are *recursive-descent parsing* and *non-recursive predictive parsing*.

Bottom-up parsing involves creating the parse tree by beginning at the leaves and working up towards the root. Bottom-up parsing finds the right-most derivation of the input string, and the class of grammars represented by such parsers is often called *LR(k)*, with some variants available, such as *lookahead-LR* (*LALR*). *Shift-reduce parsing* is a common bottom-up parsing technique, involving pushing input symbols onto a stack until the top-most sequence of symbols can be identified as a non-terminal production rule that may be reduced.

Further information on common parsing techniques may be found in [10].

3.1.3 Parser Generators

Crafting a parser by hand may turn out to be quite an extensive task for grammars defining an entire query language, like SPARQL. A number of *parser generators* exist, however, that, provided some specification, generate the source code for an entire parser.

The lexical analysis part of the parser work flow is often taken care of by a separate *scanner* that serves well-defined tokens to the other parser processes. A separate specification for the scanner usually has to be provided for a scanner generator. Token definitions are typically shared between the two specifications. Figure 3.3 shows the relationship between the scanner and parser specifications and generators.

Scanner and parser specifications are typically defined in a BNF-like representation. The generated scanner and parser code may be modified directly, but ideally the specification is tailored to produce the desired source code without having to modify the code itself.

A number of parser generators exist, covering multiple programming languages and parsing techniques. The parser generators are further discussed in [4].

3.1.4 Abstract Syntax Trees

An *abstract syntax tree* (AST) is a hierarchical representation of an input source. ASTs are constructed by parsers and often used as intermediate representations in the process of compiling or evaluating source code. [10]

ASTs can represent more than queries and computer source code. Consider the calculus expression $1 \times 2 \times 3 + 6 \div 2$. Basic calculus rules state that the $\times$ and $\div$ operators have precedence over the $+$ operator. Also, all these operators are left-associative. Factors like operator precedence and associativity affect the syntactic structure of the expression. While these factors

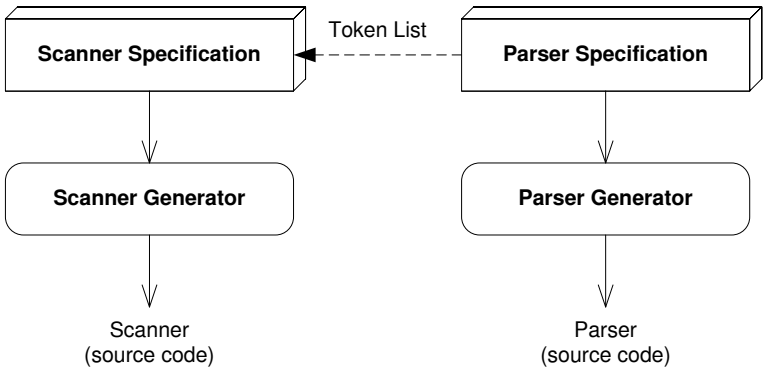


Figure 3.3: Parser Generation

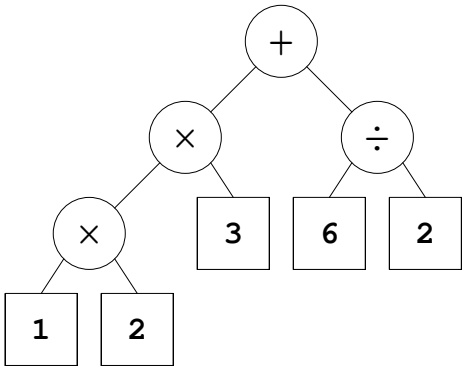


Figure 3.4: AST for the Calculus Expression $1 \times 2 \times 3 + 6 \div 2$

are impossible to deduce from the textual representation of the expression, they are easily identified in an AST.

A calculus expression parser in conformance with these basic rules would produce an AST similar to that in Figure 3.4 for this particular expression. This AST is equivalent to the explicit representation $(((1 \times 2) \times 3) + (6 \div 2))$.

For computer source code or complex SPARQL queries, ASTs tend to grow very large, easily beyond the point of comprehensibility when visualized. Thus, AST examples in this thesis will often show only the relevant parts of the tree, just enough to illustrate a point.

3.1.5 Tree Traversal using the Visitor Pattern

Tree traversal is an essential operation when it comes to handling ASTs. Most operations on the AST involve tree traversal, like translating the tree to another representation. Depending on how trees are represented in various programming languages, different techniques may be used to traverse

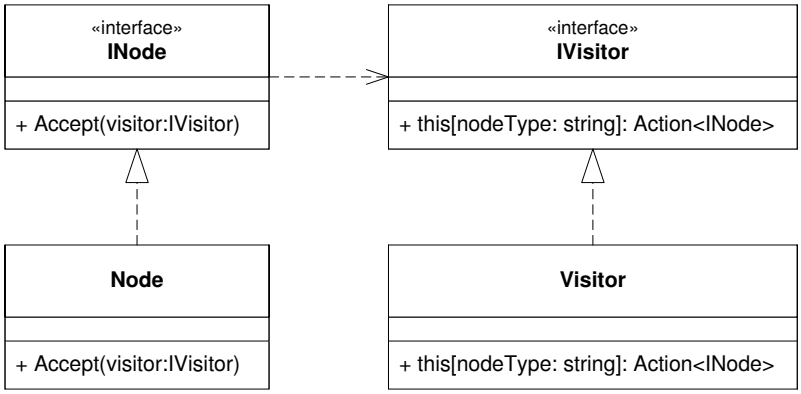


Figure 3.5: Visitor Pattern

them. Several methods exist, however, that are independent of how the trees are represented.

A common method for achieving reasonable tree traversal is the *Visitor pattern*, which separates the traversal algorithm from the data structure used to represent the tree. The pattern allows for defining new operations on the elements of an object structure without changing the classes of the elements on which it operates [12].

Figure 3.5 shows how the Visitor pattern is typically realized, represented by a C# class diagram. The `this[...]` statement represents a read-only indexer. An indexer basically corresponds to a single-parameter getter method in terms of conventional object oriented programming.

The *INode* interface offers an *Accept* method taking one argument, being a visitor implementing the *IVisitor* interface. The *IVisitor* interface defines an indexer property returning an *Action* delegate[1] given the type of the node supplied. The *Node* class implementation calls the Accept method on all child nodes, before finally calling the *Action* delegate returned by the supplied visitor with itself as the argument. Depending on the specific visitor implementation, a delegate performing some operation on the node is returned. If the visitor is only looking for some special node type, however, a delegate performing no operation may be returned. Figure 3.6 shows the initial part of the dispatch sequence.

3.2 Parsing SPARQL

W3C specifies the SPARQL grammar using an EBNF based notation [13]. Since the MPLex and MPPG scanner and parser generators are two separate entities, the grammar must be split into a scanner part and a parser

[1] A delegate is a reference to a method. For further information, see `http://msdn.microsoft.com/en-us/library/900fyy8e.aspx`.

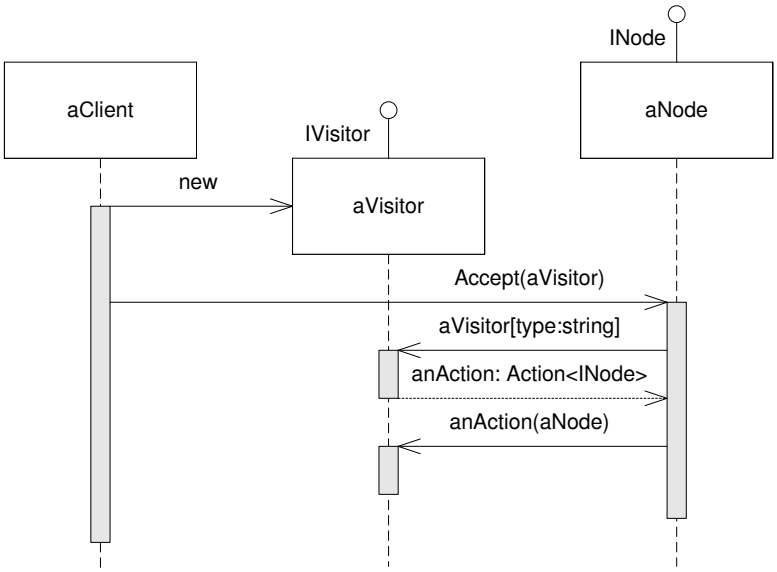


Figure 3.6: Visitor Pattern Delegate Dispatch

part. Further, the generators do not support specifying grammars directly using EBNF. Instead the grammar must be specified using specification languages closely resembling the BNF-based Lex and Yacc [14] specification languages.

These specification languages lack many of the extensions of EBNF, requiring several measures to be taken in the translation of the grammar, as described in Section 3.1.1.

3.2.1 The Scanner Specification

The grammar tokens are defined in the scanner specification using regular expressions. Each token that should be returned to the parser also defines a belonging snippet of code for returning the corresponding token enumeration item recognized by the parser. Additionally, helper tokens used to build other tokens may be defined, to avoid repeating regular expressions.

Order of declaration

Token declaration order is of significance and must be taken into account. During token matching, the scanner will attempt to match the longest token possible. If there is a tie between two or more tokens, the one defined first is returned to the parser.

For the SPARQL grammar, however, the declaration order is not an issue. All tokens sharing a common prefix differ in length, thus a tie will never occur.

```
WhereClause   ::=   'WHERE'? GroupGraphPattern

    ⇓

WhereClause
: WHERE GroupGraphPattern
{
  ...
}
| GroupGraphPattern
{
  ...
}
;
```

Figure 3.7: SPARQL Grammar Conditional Symbol Sample

3.2.2 The Parser Specification

The grammar rules are declared in the parser specification. A rule defines one or more belonging productions. A non-empty production defines a belonging snippet of code returning either a token value or an AST node to its superior production. Any empty production makes the rule optional.

Grammar Constructs

The modifiers *, + and ? are absent in the BNF-based parser specification language, and must be realized as discussed in Section 3.1.1. Figure 3.7 shows a production from the SPARQL grammar specification containing a conditional grammar symbol, namely the *WHERE* keyword. In this specific case an additional production not specifying the *WHERE* keyword is specified.

Declaration of lists of grammar symbols (one-or-more and zero-or-more instances) requires the introduction of a replacement rule, allowing for a recursive list of grammar symbols to be constructed during parsing. Figure 3.8 shows the realization of the *DataSetClause** zero-or-more instances list using the recursive list rule *DataSetClauseList*. The rule may result in an empty production (zero instances) or a production consisting of the non-terminals *DataSetClause* and *DataSetClauseList* (more instances).

Conflicts

MPPG generates shift-reduce parsers, consisting of a stack holding grammar symbols and an input buffer holding the rest of the input string to be parsed. The parser shifts input symbols onto the stack until it is ready to reduce a string of grammar symbols on the top of the stack into a superior grammar production. This process is repeated until an error is detected or

```
ConstructQuery    ::=   'CONSTRUCT' ConstructTemplate
                        DatasetClause* WhereClause
                        SolutionModifier
    ⇓

ConstructQuery
: CONSTRUCT ConstructTemplate DatasetClauseList WhereClause
    SolutionModifier
{
  ...
}
;

DatasetClauseList
: DatasetClause DatasetClauseList
{
  ...
}
| /* empty */
{
  ...
}
;
```

Figure 3.8: SPARQL Grammar List Symbol Sample

until the stack contains the predefined grammar start symbol and the input is empty. [10]

Two types of conflicts may occur during shift-reduce parsing: shift/reduce conflicts and reduce/reduce conflicts. Shift/reduce conflicts occur when the parser can not decide whether to shift input symbols onto the stack or to reduce grammar symbols on top of the stack. Reduce/reduce conflicts occur when the parser can not decide which superior production to reduce the grammar symbols on top of the stack into. The former conflict type typically occurs as a result of an ambiguous grammar, and is critical. The latter type easily occurs when creating the parser specification from a grammar specification like that of SPARQL, because of the required rewriting from an EBNF-based grammar to a BNF-based grammar.

Figure 3.9 addresses a reduce/reduce conflict encountered in translating the SPARQL grammar specification into parser grammar productions. In this specific case, both the *Prologue* and *PrefixDeclList* rules contain an empty production. In the context of a *Prologue* grammar rule, whenever the parser encounters an empty input string, it cannot decide whether to reduce the empty production to a *PrefixDeclList* rule or to a *Prologue* rule.

This reduce/reduce conflict may be prevented simply by removing the empty production belonging to the *Prologue* rule, since a *Prologue* rule may still produce an empty production via the *PrefixDeclList* rule.

```
Prologue
: BaseDecl
{
  ...
}
| BaseDecl PrefixDeclList
{
  ...
}
| PrefixDeclList
{
  ...
}
|
{ /* empty */ }
;

PrefixDeclList
: PrefixDecl PrefixDeclList
{
  ...
}
|
{ /* empty */ }
;
```

Figure 3.9: SPARQL Grammar Reduce/Reduce Sample

3.2.3 The Abstract Syntax Tree

Abstract syntax trees, described in 3.1.4, are constructed during parsing to represent the SPARQL queries. The productions defined in the parser specification contain code snippets for instantiating AST nodes of the appropriate type for representing the corresponding language construct.

The node class types used in the AST are defined in a class hierarchy closely resembling the syntactical constructs of the SPARQL grammar.

Class Hierarchy

An AST node is defined by a simple interface, *INode*, shown in Figure 3.10. This interface supports the construction of a tree structure, navigable in both directions through the *Parent* and *Children* properties. The *Accept* method is part of the *Visitor* interface, explained in Section 3.1.5.

Below the *INode* interface in the class hierarchy is the abstract class *NodeBase*, shown in Figure 3.11. The complete implementation of *NodeBase* is provided in Appendix A. The purpose of this class is to provide a basic implementation of the *INode* interface. All its members are virtual, allowing for subclasses to override them if necessary. Although this class has no abstract members, it is marked as abstract to prevent instantiation, and to allow for future abstract members.

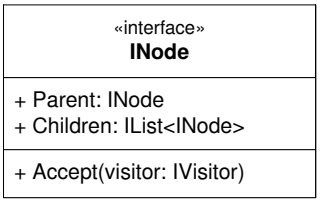


Figure 3.10: The *INode* Interface

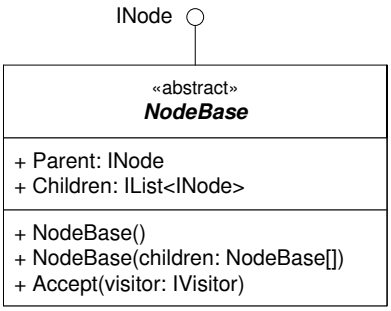


Figure 3.11: The *NodeBase* Abstract Class

The public *Parent* and *Children* properties expose the protected *parent* and *children* fields respectively.

NodeBase provides two constructors. The default parameterless constructor makes sure the *children* field is initialized as an empty collection, while a second constructor allows for initializing the *children* field with initial *NodeBase* objects.

The *Children* property of *NodeBase* is exposed as an *IList<INode>* interface, and implemented as a custom *NodeCollection* class. This class is marked as *internal* and is thus not exposed by the API. A custom *IList* implementation was chosen over the standard ones to allow for customization.

The concrete AST classes all reside in the *SharQL.Ast.Nodes* namespace. Ultimately, they are all descendants of *NodeBase* and thus implement the *INode* interface.

3.3 The Parser Facade Class

The MPLex and MPPG tools generate a scanner and a parser, respectively. In order for the end-user to perform any parsing, a scanner and a parser have to be instantiated, and the parser has to be made aware of the existence of the scanner instance in order to read its output. The scanner, in turn, needs to know what input to tokenize.

To encapsulate all this plumbing, a parser facade class, as shown in

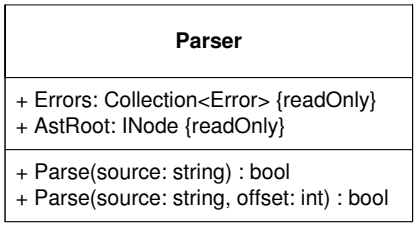


Figure 3.12: The *Parser* Facade Class

Figure 3.12, has been developed. The intention of this class is to let the end-user instantiate it once, and make successive calls to one of its *Parse* methods without needing to have a notion of a scanner at all. When a parsing succeeds, the resulting AST is made available through the *AstRoot* property. Additionally, the parser facade reports any errors that may occur, through its *Errors* property.

A UML sequence diagram showing parser facade class operation during parsing is shown in Figure 3.13.

Using the SharQL Parser is straight-forward. The *Parser* class resides in the *SharQL* namespace, and is instantiated and used as shown in Figure 3.14. The *Parse* method of the *Parser* class returns *true* only if the input query was successfully parsed. The abstract syntax tree can then be accessed from the parser's read-only property *AstRoot*. This property returns an *INode* object representing the root node of the AST. The *Parent* and *Children* properties can be used to traverse the AST manually, or the Visitor pattern can be employed by providing the Accept method with an *IVisitor* object.

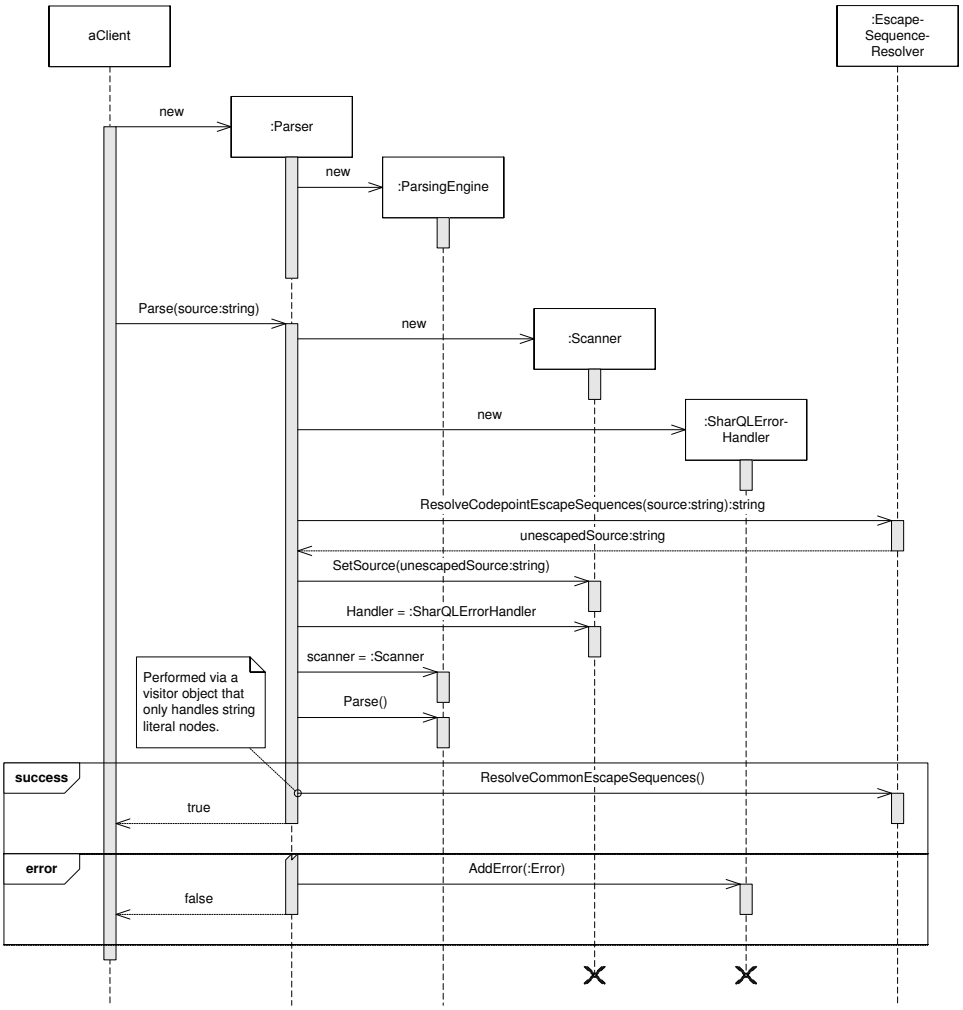


Figure 3.13: UML Sequence Diagram Showing Parser Facade Class Operation

```
string query = "SELECT ?var WHERE ...";
SharQL.Parser parser = new SharQL.Parser();
SharQL.Ast.INode astRoot = null;

try
{
  if (parser.Parse(query))
  {
    astRoot = parser.AstRoot;
  }
}
catch (Exception e)
{
  // Report error
}

// Act on AST represented by astRoot
```

Figure 3.14: Using the SharQL Parser

Chapter 4

Storage Models

Efficiently storing and retrieving RDF data has been subject to a lot of research. While the RDF data model is simply a set of triple statements, efficient storage models are considerably more complex. Various approaches have been proposed, each with its strengths and weaknesses. In this chapter, the most relevant approaches are discussed, followed by a proposal for a storage model, given the confinements of Mars.

As iAD focuses on schema-agnostic indexing services, only schema-less approaches are considered [3].

4.1 Alternative Models

Various approaches to efficiently storing RDF data have been proposed. This section discusses relevant, schema-agnostic alternatives which meet different requirements in different scenarios.

4.1.1 General Triple Store

A *general triple store* is the basic naïve approach to storing RDF data. This method stores every statement in a single three-column table as shown in Table 4.1, cited from [15], usually in an RDBMS. Without further modifications, this approach is terribly slow, mainly because of the self-joins necessary for even relatively simple queries. Because RDF puts no restrictions on the size of URIs and literals, a straight-forward triple store is also very space consuming.

No field or combination of two fields is guaranteed to be unique throughout the triple store table. Hence, the primary key must consist of all three fields. Applying secondary non-unique indices on each separate field would improve efficiency when retrieving a triple through lookup on either subject, predicate or object.

Subject	Predicate	Object
ID1	type	BookType
ID1	title	“XYZ”
ID1	author	“Fox, Joe”
ID1	copyright	“2001”
ID2	type	CDType
ID2	title	“ABC”
ID2	artist	“Orr, Tim”
ID2	copyright	“1985”
ID2	language	“French”

ID3	type	BookType
ID3	title	“MNO”
ID3	language	“English”
ID4	type	DVDType
ID4	title	“DEF”
ID5	type	CDType
ID5	title	“GHI”
ID5	copyright	“1995”
ID6	type	BookType
ID6	copyright	“2004”

Figure 4.1: Triple Store Example

Storing potentially large URIs and literals directly in each row of the triple store has several negative implications; rows can not be of fixed size and the scalability degrades because of the space consumption. A common way to solve this issue is to *dictionary-encode* all URIs and literals. Every single URI and literal is stored in a separate table and assigned a unique fixed-size identifier, usually a 64-bit integer. These identifiers are easily computed through hashing or auto-incrementing.

By replacing all entries of a triple with fixed-size identifiers, the rows of the triple-store become fixed-size as well. This results in remarkably less overhead, because field length information no longer needs to be stored for each and every row. In fact, dictionary-encoding URIs and literals this way is so effective that every approach presented in this section, except the graph store, uses this technique. Dictionary encoding is explained further in Section 4.1.7.

4.1.2 Graph Store

Rather than having to reconstruct the relevant parts of an RDF graph in main memory from a compatible RDBMS schema, [16] proposes that the RDF graph itself is stored in an object-oriented DBMS. Their prototype, *OO-Store*, is based on the object-oriented DBMS *FastObjects* and translates queries from RQL to the query language OQL used by FastObjects.

OO-Store uses three types of objects, or classes, to represent the RDF data. A base class, *Values*, contain a single *outedges* property map. Two subclasses, *Resources* and *Literals*, represent URIs and literals respectively. Each key in the property map points to a list of Value objects, resulting in the OODBMS storing the actual graph structure, rather than storing the RDF data as triples. While the property map of URIs contains references to all nodes that further describe the resource identified by the URI, the property

Property Table

Subject	type	title	copyright
ID1	BookType	"XYZ"	"2001"
ID2	CDType	"ABC"	"1985"
ID3	BookType	"MNP"	NULL
ID4	DVDType	"DEF"	NULL
ID5	CDType	"GHI"	"1995"
ID6	BookType	NULL	"2004"

Left-Over Triples

Subject	Object	Property
ID1	author	"Fox, Joe"
ID2	artist	"Orr, Tim"
ID2	language	"French"
ID3	language	"English"

Figure 4.2: Property Table Example

map of literals are typically only used to store any data type information.

If object-oriented storage is available, storing RDF data as a graph in this way makes sense. Evaluating path queries is then a matter of following edges of the object graph, rather than requiring joins of relational tables. The performance gain is not significant, however. In fact, [16] states that the main advantage of its approach is the close relationship between the query languages OQL and RQL, simplifying the translation process.

4.1.3 Property Tables

RDF data often describes several properties[1] of the same subjects. If one can identify a common set of properties that are used to describe a set of subjects, one can construct a *property table* containing a subject identifier field in addition to fields for each of the properties identified. Additionally, data-typed literals can be stored directly in the property table using their corresponding native data types. A left-over triple store is used to store data that do not fit into any property table, as shown in Figure 4.2, cited from [15].

The major advantage of this approach is that expensive subject-subject self-joins are no longer needed to retrieve the common set of properties for one particular subject. Reading the single row containing information on the subject is sufficient. If a query requires information stored in another

[1] In this context, a property is equivalent to the predicate in an RDF triple. The value of the property is the object of the triple.

type

ID1	BookType
ID2	CDType
ID3	BookType
ID4	DVDType
ID5	CDType
ID6	BookType

title

ID1	"XYZ"
ID2	"ABC"
ID3	"MNO"
ID4	"DEF"
ID5	"GHI"

(etc.)

Figure 4.3: Vertical Partitioning Example

property table or in the triple store, however, potentially expensive self-joins still need to be applied.

A disadvantage of property tables is their lack of support for properties with multiple values. A book with several authors are easy to represent in a triple store, using one triple for each author, but can not be stored in a single Author field of a property table.

The main challenge with implementing property tables is to choose the set of properties to include. Including too few properties reduces the gain of implementing property tables in the first place. Including too many properties increases the chance of *NULL* values posing a significant storage overhead [15].

Designing property tables from schema-less RDF data sets is essentially a matter of deducing a schema from available data. [17] proposes an iterative method of deducing schemas from existing data sets. However, such an approach undermines the fundamental pay-as-you-go principle of schema-less RDF data by posing a significant overhead to the process of storing the data.

4.1.4 Vertical Partitioning

The authors of [15] propose using vertical partitioning, that is, rewriting the triples table into n two-column tables as shown in Figure 4.3, n being the number of unique predicates present among the data. The two columns of each table contain the subject and the object for each occurrence of the predicate represented by that table.

Predicates with multiple values are easily represented with this approach, by adding several rows with the same subject identifier to the table representing the predicate. *NULL* values are non-existent, as missing predicate values are simply omitted from the tables. While property tables force you to retrieve entire sets of properties at a time with potentially irrelevant fields, this approach has the benefit that only relevant fields has to be accessed on disk.

Another advantage is the possibility of materializing path expressions. In an example from [15], queries that fetch books where the author is born in a given year are identified as particularly common. By merging the tables *author* and *wasBorn* into a materialized *author:wasBorn* view, these queries are easily and efficiently resolved.

Accessing several properties in the same query still involves joining tables. However, these tables are substantially smaller than the single triple store used by other approaches.

Insert operations can be slower in vertically partitioned tables compared to property tables and triple stores, because up to n tables must be accessed to store n statements.

[15] also explores the possibility of storing the vertically partitioned tables in a column-oriented DBMS. By modifying the open source *C-Store* DBMS [18], a 32-fold performance gain compared to the state of the art triple stores is achieved, according to their benchmarks. However, [19] questions the benchmarks used, and argues that when proper clustered indices are used, triple stores perform better overall than vertical partitioning, especially for large data sets with many unique properties.

4.1.5 MAP Indexed Triple Store

[20] proposes extending a regular triples store with a set of access pattern indices. [21] refers to this approach as MAP (Multiple Access Patterns). The principle is that no matter which part(s) of a triple in a query that is unspecified, a B+ tree index [22] should exist to efficiently resolve all matching triples.

For a triple (s : p : o), there exist eight different access patterns where parts of a triple may be unspecified, as shown in Table 4.1 (? denoting an unspecified part). Note that the belonging context element of the triple, accounted for by [20], is considered irrelevant in this context and omitted in this example.

A naïve approach is to provide eight indices, one for each access pattern. A better approach is to use combined indices. In fact, every access pattern from Table 4.1 can be resolved using one of three combined indices. For example, the access pattern (s : ? : ?) can be resolved using a prefix query on an *spo* index. Necessary indices and the access patterns they support are shown in Table 4.2.

This simple and general, yet reasonably efficient approach to storing RDF data has made it popular and the method is used in systems like YARS, HPRD and RDF-3X [20, 23, 24].

#	Access Pattern
1	(? : ? : ?)
2	(s : ? : ?)
3	(s : p : ?)
4	(? : p : ?)
5	(? : p : o)
6	(? : ? : o)
7	(s : ? : o)
8	(s : p : o)

Table 4.1: Possible Access Patterns for an RDF Triple

spo	po	os
(? : ? : ?)	(? : p : ?)	(? : ? : o)
(s : ? : ?)	(? : p : o)	(s : ? : o)
(s : p : ?)		
(s : p : o)		

Table 4.2: Necessary Indices to Support All Access Patterns

4.1.6 TripleT

The *Three-way Triple Tree* (TripleT) approach [21] stores triples in a special type of clustered B+ tree index. Every single URI reference and literal (in effect, the set of all subjects, predicates and objects) are indexed in this single B+ tree. The payload of every leaf node k in the B+ tree is a record consisting of three tuple-lists, dubbed s, p and o, as shown in Figure 4.4. The s-list contains predicate/object pairs representing every statement where the k node appears as subject. Accordingly, the p- and o-lists contain subject/object pairs and subject/predicate pairs respectively.

TripleT can be considered a generalization of vertical partitioning. After all, the approach described in Section 4.1.4 is equivalent to a TripleT index where only predicates are indexed, and the payloads consist only of the p-list of subject/object pairs.

MAP's property of supporting all access patterns through indices are held also by TripleT. While the MAP approach requires multiple indices, however, TripleT requires only a single index.

A disadvantage of TripleT is the extra level of complexity added as a result of the customized payload. Each index lookup involves an extra step of accessing the appropriate of the three possible lists of tuples.

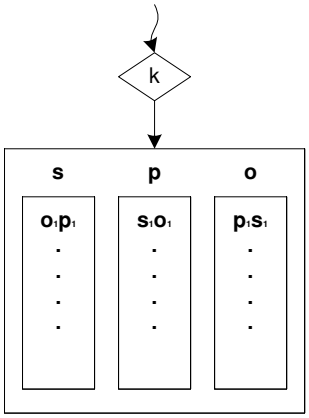


Figure 4.4: The Payload of a TripleT Index

4.1.7 Similarities Among Alternatives

Dictionary Encoding Every approach discussed in this section, except for the graph store, uses dictionary encoding on URIs and literals. This involves maintaining a separate repository, or dictionary, of mappings from fixed-size identifiers to the URIs and literals they represent.

The identifiers are typically integers and created in one of two ways; either by auto-incrementing or by hashing. Auto-incrementing starts by assigning the first entry the value zero, and then every new entry is assigned a value one higher than the previous. The benefit of this method is that the entire value space of the chosen integer type can be used without having to worry about collisions. However, to find the identifier of a URI or literal at a later point, a lookup in the dictionary is required.

Another way of creating dictionary identifiers is to use hashing. A hash function takes an arbitrary input and produces a fixed-size integer output. The MD5 hash function is a widely used universal[2] hash function, producing 128-bit integer keys. If 64-bit keys are used, the output of the MD5 hash can simply be truncated.

The benefit of using hashing is that it is easy to find the identifier of a URI or literal by just applying the hash function once more, removing the need to perform a dictionary lookup. There is a slight possibility that two input values yield identical output values, however, causing a key collision in the dictionary. This has to be asserted and accounted for in the implementation.

Regardless of whether auto-incrementing values or hashing is used, it is beneficial to store the dictionary in a B+ tree index, sorted on the key. This

[2] A universal hash function has the property that the probability of a collision between two different keys x and y are the same for all x and y [25].

makes it easier to perform prefix queries on URI and literal values [20].

By storing the RDF data using fixed-size identifiers rather than the actual URIs and literals, every triple becomes fixed-size as well. This decreases the space consumed and increases the speed of scans and lookups. It also significantly increases the speed of performing joins, as fixed-size integer values are much faster to compare than variable-size strings.

Auxiliary lookup structures MAP and TripleT in particular use indices as auxiliary structures, tailored for fast lookup of RDF triples meeting certain criteria. TripleT even specifies a custom format for the payload of its index, consisting of three separate lists of tuples. The rest of the approaches do not explicitly dictate the use of indices, but the underlying DBMSs will certainly use indices implicitly in order to improve performance.

4.2 Existing Implementations

This section discusses the approaches to RDF storage for the most commonly used implementations on the market. Most systems are based on variations of the general triple store, using a relational DBMS for persistence.

4.2.1 Sesame

Sesame is conceptually a generic architecture, independent of the choice of underlying DBMS. An abstraction layer handles the interfacing between the DBMS and the querying engine. An adapter for this abstraction layer has to be implemented for each DBMS to support. The adapter chosen for this abstraction layer dictates the structuring of the RDF graph in the DBMS. [26]

Initially, PostgreSQL was used as the underlying storage engine. Sesame supports RDF Schema, and PostgreSQL was chosen for its object-relational capabilities. Subclasses from the RDF Schema are represented in PostgreSQL by subtables. This resembles the graph store discussed in Section 4.1.2.

Sesame also has an adapter implemented for MySQL support. Unlike PostgreSQL, MySQL does not support subtables. Hence, the RDF Schemas are stored explicitly in separate tables which have to be consulted in order to answer RDFS queries. This prevents the database schema from changing when the RDF schema changes. The actual RDF triples are dictionary-encoded and stored in a general triple store as discussed in Section 4.1.1.

4.2.2 Jena

Jena is an open-source framework written in Java for storing and querying RDF data and RDFS schemas. [27]

The persistence functionality of Jena is supported through relational databases like MySQL, PostgreSQL, Oracle or Microsoft SQL Server. The DB schema is based on a triple store. Rather than strictly dictionary encoding every URI and literal, however, a hybrid solution is used where short[3] URIs and literals are stored directly in the triples table, while the remaining elements are stored as references to a resource dictionary. This way, the amount of joins between the triples table and the resource dictionary is decreased compared to a strict dictionary-encoded approach. The triples table has two non-unique indices, one on the key <*Subject, Property*> and another on the key <*Object*>. [28]

4.2.3 YARS

YARS (Yet Another RDF Storage) is an RDF storage system hosted as a web service. Storing and querying of RDF data is done via an HTTP-based interface. [29]

The developers of YARS proposed the MAP index structure as discussed in Section 4.1.5 [20]. A context node is stored with each RDF triple, effectively turning them into quadruples. Multiple indices are introduced in order to support all possible access patterns. By combining indices, fewer indices are needed than the number of possible access patterns.

Storage is implemented using the open-source persistence engine JDBM which supports B+ tree indices [30]. This relieved the authors of implementing B-tree structures themselves. An additional layer to support concatenated keys in the indices still had to be implemented.

4.2.4 Redland RDF Libraries

As the name implies, Redland is merely a set of libraries. These open-source libraries include functionality for manipulation, storage, querying and serialization of RDF graphs. It is written in C, but bindings are offered for the languages Perl, PHP, Python and Ruby. [31]

The storage engine is based on hashes. Each triple is stored using three hashes, each mapping two elements of the triple to the third. This method makes for efficient querying of triples where only one element is unknown. However, when more than one element is unknown, querying is potentially very slow when models grow large [32]. This approach is an improvement over the naïve triple store discussed in Section 4.1.1, but still

[3] By default, "short" means less than 256 characters, but this is configurable.

has major shortcomings with regards to the discrimination of some types of queries.

Persistence of the hash storage is supported via many alternative backends, among others Berkeley DB, MySQL, PostgreSQL and plain files. [31]

4.2.5 3store

3store is an RDF triple store written in C for POSIX compliant operating systems. Persistence is based on traditional relational database principles and it uses MySQL as its storage engine. A design principle of 3store has been to delegate as much as possible of the workload to the DBMS in order to benefit from optimizations performed by the query optimizer. This differs from the approaches of e.g. Sesame and Jena where queries are mostly evaluated by the RDF-engine on top of the DBMS. The approach taken by 3store is based on a dictionary-encoded triple store as discussed in Section 4.1.1. [25]

4.3 Choosing the Storage Model

Choosing the storage model is a matter of finding the most suitable solution given the confinements of the environment in which the solution will be implemented. As discussed in Section 2.3, Mars implies a different paradigm compared to standard relational DBMSs. This affects the choice of storage model.

4.3.1 Reasonable Alternatives

Several of the alternative storage models from Section 4.1 are possible to adapt to Fast's storage model. The general triple store is not considered in this section, due to its inferiority compared to the other approaches.

Property tables are possible to implement, e.g. by regarding each row of the property table as a document with scopes corresponding to the fields of the property table. However, the disadvantages discussed in Section 4.1.3 still apply, particularly the difficulty of choosing which properties to include in the table and which to leave for the left-over triples table.

The vertically partitioned approach from Section 4.1.4 can be adapted to Fast's storage model by regarding triples as documents. The subject/object pairs are stored in inverted lists, corresponding to the predicate keys of the index.

The MAP Indexed Triple Store approach from Section 4.1.5 is hard to implement in Fast's storage model, as having multiple indices pointing to a single data store is problematic, according to our supervisor from Fast.

The TripleT approach from Section 4.1.6 is also hard to implement in Fast's storage model, as having a payload list composed of three lists, one

per triple element, is problematic, also according to our supervisor from Fast.

4.3.2 Proposed Solution

Based on an evaluation of existing alternatives and feedback from our supervisor from Fast, a storage model based on TripleT was chosen. This approach, as the pure vertically partitioned approach, is highly suitable for efficient storage and retrieval in a column based storage system. Fast's storage engine supports column based storage, and is thus able to take advantage of the potential performance improvement in scenarios where vertical partitioning is beneficial.

Dictionary encoding of all URIs and literals should be used, with IDs of 64 bits or more, but no larger than the platform is able to handle natively[4]. Dictionary encoding is explained in Section 4.1.7.

Fast already supports mechanisms for efficiently resolving URLs from document IDs through their *Value* operator, used internally by the search engine. We propose that this mechanism can be altered to support the resolving of URIs and literals from IDs produced during dictionary encoding, reusing existing functionality where applicable.

Another benefit of using dictionary encoding is that it takes advantage of Fast's support for compressing index keys using delta coding. For a sequence of n values, delta coding only leaves the first value unaltered. Every subsequent value is encoded using a delta value, computed as the difference between the value of the current and previous element in the sequence. Delta coding is more efficient when the encoded values are within a 64-bit range than if they were of arbitrary size, because the sizes of delta values are upper-bound by the maximum size of the values they represent. [33]

Although based on TripleT, some modifications are proposed in order to adapt to Fast's storage model. Storing three lists of tuples in the payloads is problematic. Thus, rather than having a payload consisting of three different lists, this approach includes the role of the triple element as part of the key. A node element x will potentially appear three times in the index tree, as *s:x* if it appears as subject in the data set, as *p:x* if it appears as predicate or as *o:x* if it appears as object. As with TripleT, a B+ tree index is used for efficient indexing and lookup.

The payload of each index key is a single two-column list containing predicate/object pairs, subject/object pairs and subject/predicate pairs respectively. These lists are analogous to the predicate tables discussed in Section 4.1.4, and yield the same benefits with regards to column storage.

[4]Using larger values than the platform can handle natively introduces a significant overhead e.g. when performing value comparisons.

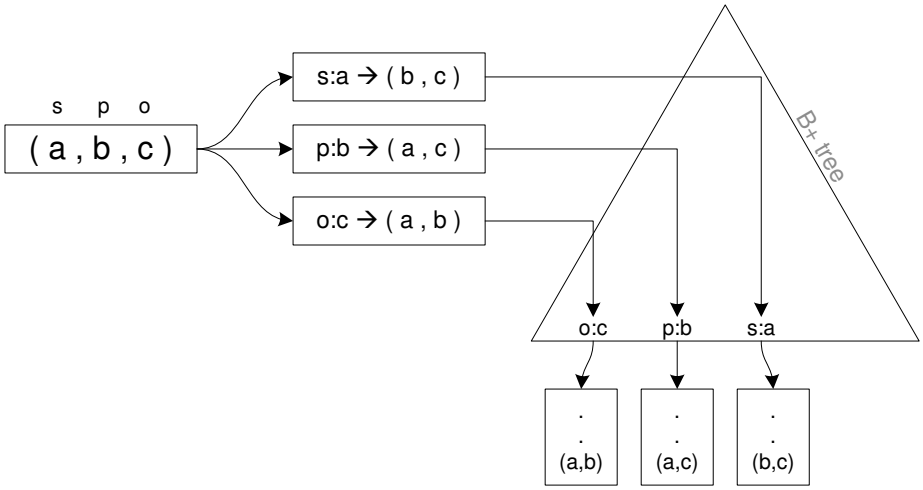


Figure 4.5: Inserting a Triple in the Chosen Storage Model

Using the proposed solution, the process of inserting a new triple is illustrated in Figure 4.5. The triple is decomposed into three tuples, each identified by the concatenation of the omitted triple element and a symbol identifying its role in the triple. Next, these three tuples are inserted into the B+ tree. If the key exists, the tuple is appended to the payload of the existing key. Otherwise, the key is inserted and a new list containing the tuple is set as the payload.

This proposed solution is based on results from state of the art research on RDF storage, but adapted to the environment in which Fast's systems operate. The solution is completely schema-agnostic, pursuant to the pay-as-you-go philosophy of the Semantic Web.

Chapter 5

Method

Initially, all SPARQL queries are parsed from their textual representations into abstract syntax trees by the SharQL Parser. An abstract syntax tree is little but an object graph representation of the corresponding query.

The SPARQL specification introduces the concept of SPARQL algebra. The semantics of SPARQL, in turn, is based on this algebraic representation of queries. The transformation from a textual SPARQL query to its algebraic representation is formally described in the specification [2].

Although Mars operates on directed acyclic graphs of operators, only graphs qualifying as trees will be used in the context of this thesis, and operator *trees* is consequently the term that will be used throughout the rest of this thesis.

When the query has been transformed into SPARQL algebra, the next step is to translate it into a tree of Mars operators. This step involves finding an equivalent set of Mars operators for each construct defined by the SPARQL algebra.

The different representations a query takes during evaluation are shown in Figure 5.1. In the end, the resulting tree of Mars operators is passed to Mars and evaluated. The results from this evaluation are then presented to the user.

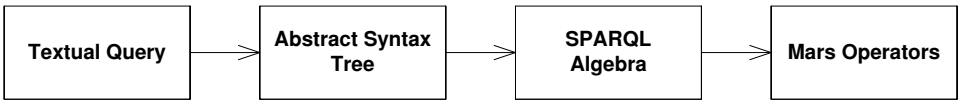


Figure 5.1: The Intermediate Representations Taken by a Query During Evaluation

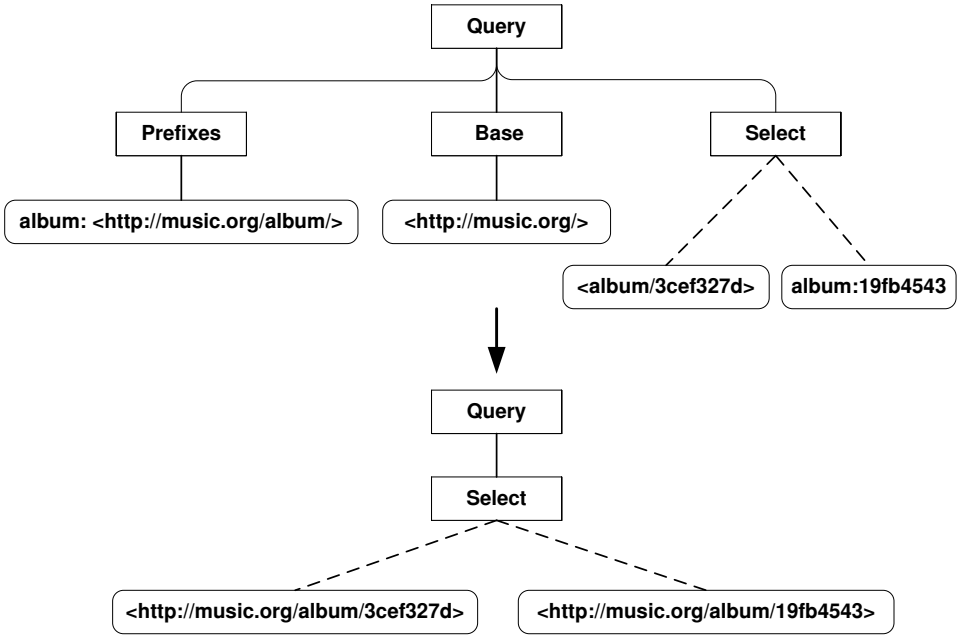


Figure 5.2: Applying Base and Prefix Declarations

5.1 Preparing the AST

The SPARQL language supports several shorthand notations to ease the writing of queries. In order to simplify the analysis of queries, such shorthand notations can be replaced by the equivalent explicit notation. This section presents the shorthand notations available in SPARQL and describes how to prepare the AST for transformation by replacing them with explicit notation.

The intention of preparing the AST before performing further analysis is to simplify the analysis process. When all nodes of the AST with certainty are represented in their explicit forms, the analysis will not need to consider different types of syntax for equivalent semantics.

Base and Prefix specifications URIs are essential in RDF and SPARQL. URIs tend to be very verbose, however. To compensate for this, SPARQL allows the specification of a base URI and a set of named prefixes, as described in Section 2.2.6. During AST preparations, all base and prefix declarations are applied to the URIs in the AST, as described in Section 6.4.5. Figure 5.2 shows a simple AST before and after applying base and prefix declarations.

Shorthand notation	Explicit notation
`123`	`"123"^^xsd:integer`
`123.4`	`"123.4"^^xsd:decimal`
`123e10`	`"123e10"^^xsd:double`
`true` (or `false`)	`"true"^^xsd:boolean`

Table 5.1: Implicitly Data-Typed Literals in SPARQL.

Implicitly Data-Typed Literals RDF allows for data-typed literals. The data-typed integer 42 is e.g. represented in Turtle notation by the literal `"42"^^xsd:integer`. SPARQL supports shorthand notations for some of the most common data types, as shown in Table 5.1. The preparation process traverses the AST and replaces any occurrence of these shorthand notations with the equivalent explicitly data-typed literal, as described in Section 6.4.1.

Other Syntactic Sugar Shared subject triple lists, blank node property lists and RDF lists can all be specified in SPARQL using shorthand notation as explained in Section 2.2.7. The preparation process expands all occurrences of these notations to their explicit notation, as described in Sections 6.4.2, 6.4.3 and 6.4.4 respectively.

5.2 From AST to Algebra

Parsing a SPARQL query yields an AST consisting of nodes closely resembling the structure of the SPARQL grammar, as described in Section 3.2. This tight coupling with the grammar makes it easy to recognize the query structure. Reasoning about its semantics becomes rather cumbersome, however, calling for transformation into an intermediate representation.

After preparing the AST as described in the previous section, the AST is transformed into a SPARQL algebra representation [2]. This eases future analysis of the query significantly, as needless metadata from the parsing process is eliminated as well as the fact that the semantic description of the SPARQL query language is based on such algebra.

5.2.1 Graph Patterns

Section 12.2.1 in the SPARQL specification [2] describes the transformation of SPARQL query patterns into SPARQL algebra patterns, originating from the *WHERE* clause. The specification defines a set of algebra patterns for evaluating any *WHERE* clause. These are:

- $\mathrm{Filter}(expression, pattern)$
- $\mathrm{Join}(pattern, pattern)$
- $\mathrm{LeftJoin}(pattern, pattern, expression)$
- $\mathrm{Union}(pattern, pattern)$
- $\mathrm{Graph}(uri$ or $variable, pattern)$

Additionally, to represent the basic graph patterns of the *WHERE* clause used for matching triples, a special algebra pattern is introduced:

- BGP(list of triple statements)

In order to discuss the semantics of SPARQL algebra, some terminology is introduced. A solution mapping is denoted by μ and defined as follows.

Definition 5.1 (Solution Mapping). *A solution mapping, μ, is a partial function $\mu : V \rightarrow T$, where V is the set of query variables and T is the set of RDF terms. The domain of μ, $\mathrm{dom}(\mu)$, is the subset of V where μ is defined.*

Definition 5.2 (Compatible Mappings). *Two solution mappings μ_1 and μ_2 are compatible if, for every variable v in $\mathrm{dom}(\mu_1)$ and in $\mathrm{dom}(\mu_2)$, $\mu_1(v) = \mu_2(v)$.*

Further, the RDF specification defines an instance mapping of blank nodes to RDF terms, denoted σ in the SPARQL algebra, as follows [34].

Definition 5.3 (Instance Mapping). *Suppose that M is a mapping from a set of blank nodes to some set of literals, blank nodes and URI references; then any graph obtained from a graph G by replacing some or all of the blank nodes N in G by $M(N)$ is an instance of G.*

From these definitions, a solution for a basic graph pattern is defined as follows [2].

Definition 5.4 (Basic Graph Pattern Matching). *Let BGP be a basic graph pattern and let G be an RDF graph.*

μ is a solution for BGP from G when there is a pattern instance mapping P such that $P(BGP) = \mu(\sigma(BGP))$ is a subgraph of G and μ is the restriction of P to the query variables in BGP.

With these basic definitions established, the graph patterns listed in the beginning of this section can be formally defined and described. In these definitions, Ω is used to denote multisets, or bags, which are a variation of unordered sets where elements may appear more than once. By default, SPARQL queries do not remove duplicates. Thus, it makes sense to represent intermediate results as multisets.

Definition 5.5 (Filter Pattern). *Let Ω be a multiset of solution mappings and $expr$ be an expression.*

$\text{Filter}(expr, \Omega) = \{\mu \mid \mu$ *in* Ω *and* $expr(\mu)$ *is an expression that has an effective boolean value of* $true\}$

For each solution in Ω, the *Filter* pattern evaluates $expr$. If the result of this evaluation yields a boolean value of $true$, the solution against which the expression was evaluated is included in the result set.

Definition 5.6 (Join Pattern). *Let Ω_1 and Ω_2 be multisets of solution mappings.*

$\text{Join}(\Omega_1, \Omega_2) = \{\ \mu_1 \cup \mu_2 \mid \mu_1$ *in* Ω_1 *and* μ_2 *in* Ω_2*, and* μ_1 *and* μ_2 *are compatible* $\}$

For each solution in Ω_1, the *Join* pattern checks for compatibility with every solution in Ω_2. For each compatible pair of solution mappings found, the two mappings are merged and included in the result set. From Definition 5.2, a *Join* pattern will result in the Cartesian product when Ω_1 does not contain solutions for any query variables included in the solutions of Ω_2, as such solutions are considered compatible.

Definition 5.7 (Diff Pattern). *Let Ω_1 and Ω_2 be multisets of solution mappings.*

$\text{Diff}(\Omega_1, \Omega_2, expr) = \{\ \mu \mid \mu$ *in* Ω_1 *such that for all* μ' *in* Ω_2*, either* μ *and* μ' *are not compatible or* μ *and* μ' *are compatible and* $expr(\mu \cup \mu')$ *has an effective boolean value of* $false\}$

The *Diff* pattern is used internally for defining the *LeftJoin* pattern, described below. The criteria for a solution in Ω_1 to be included in the result set is that it is not compatible with any solution in Ω_2 and the evaluation of $expr$ on the union of the two solution must yield an effective value of $false$.

Definition 5.8 (LeftJoin Pattern). *Let Ω_1 and Ω_2 be multisets of solution mappings and $expr$ be an expression.*

$\text{LeftJoin}(\Omega_1, \Omega_2, expr) = \text{Filter}(expr, \text{Join}(\Omega_1, \Omega_2)) \cup \text{Diff}(\Omega_1, \Omega_2, expr)$

The *LeftJoin* pattern is a combination of the *Filter*, *Join* and *Diff* patterns, as shown above. It differs from a regular *Join* pattern in that solutions from Ω_2 are optional. If no solutions from Ω_2 are compatible with a solution from Ω_1, the solution from Ω_1 is still included.

Definition 5.9 (Union Pattern). *Let Ω_1 and Ω_2 be multisets of solution mappings.*

$\text{Union}(\Omega_1, \Omega_2) = \{\ \mu \mid \mu$ *in* Ω_1 *or* μ *in* $\Omega_2\}$

The *Union* pattern simply includes a solution to the result set if the solution is part of either Ω_1 or Ω_2.

The *Graph* specifier pattern is used for specifying the currently active RDF data set to be queried.

5.2.2 The Transformation to Algebra

Several query forms exist, as described in Section 2.2.4. In the context of this thesis, however, only *SELECT* queries are considered. Such queries consist of a set of patterns and a set of modifiers, which are transformed into a set of algebra graph patterns and a set of solution modifiers, respectively. Transforming modifiers is trivial, whereas the transformation of the patterns is a more complicated process.

The set of patterns present in a *SELECT* query has its origin from the *WHERE* clause. This clause contains a *GroupGraphPattern* production which, in turn, is comprised of the following patterns: *TriplesBlock*, *Filter*, *OptionalGraphPattern*, *GroupOrUnionGraphPattern* and *GraphGraphPattern*. Section 12.2.1 in the SPARQL specification[2] describes a *Transform* function for transforming these patterns into SPARQL algebra, its pseudo code shown in Figure 5.3. This algorithm is initially invoked by passing a query's *GroupGraphPattern* as the only argument. The implementation of this function is discussed in Section 6.5.1.

A sample transformation from [2] is shown in Figure 5.4. Both the *UNION* and *OPTIONAL* keywords are left-associative. Hence, the Union pattern is a child of the LeftJoin pattern. The third argument to the LeftJoin operator is *true* because no filters are specified in any of the patterns belonging to the *OPTIONAL* keyword.

5.3 Evaluation Approaches

When it comes to evaluating queries against an RDF store, it is desirable to perform the majority of the query evaluations at the lowest possible layer of execution, typically inside the data store itself. The naïve approach on the other hand, involves potentially fetching all the data from the store and performing the entire evaluation independently, at a higher level of execution, typically outside the store.

5.3.1 Existing Solutions

A selection of commonly used RDF store implementations is presented in Section 4.2 with regards to the storage models used. The different implementations support querying using different query languages and different approaches.

The *Sesame* implementation is a generic architecture, independent of the choice of underlying data store [26]. In Sesame, query evaluation is mainly performed in the query engine itself, greatly reducing the dependency on the data store used. However, tailoring the query evaluation for specific data storages by utilizing their evaluation and optimization mechanisms, would probably offer better performance as it is reasonable to expect such

```
Function Transform(syntax form)

  If the form is TriplesBlock

    The result is BGP(list of triple statements)


  If the form is GroupOrUnionGraphPattern

    Let A := undefined

    For each element G in the GroupOrUnionGraphPattern
      If A is undefined
        A := Transform(G)
      Else
        A := Union(A, Transform(G))

    The result is A


  If the form is GraphGraphPattern

    If the form is GRAPH IRI GroupGraphPattern
      The result is Graph(IRI, Transform(GroupGraphPattern))
    If the form is GRAPH Var GroupGraphPattern
      The result is Graph(Var, Transform(GroupGraphPattern))


  If the form is GroupGraphPattern

    Let FS := the empty set
    Let G := the empty pattern, Z

    For each element E in the GroupGraphPattern
      If E is of the form FILTER(expr)
        FS := FS set-union {expr}
      If E is of the form OPTIONAL{P}
        Let A := Transform(P)
        If A is of the form Filter(F, A2)
          G := LeftJoin(G, A2, F)
        Else
          G := LeftJoin(G, A, true)
      If E is any other form:
        Let A := Transform(E)
        G := Join(G, A)

    If FS is not empty:
      Let X := Conjunction of expressions in FS
      G := Filter(X, G)

    The result is G.
```

Figure 5.3: Pseudo Code for the Algebra Transformation

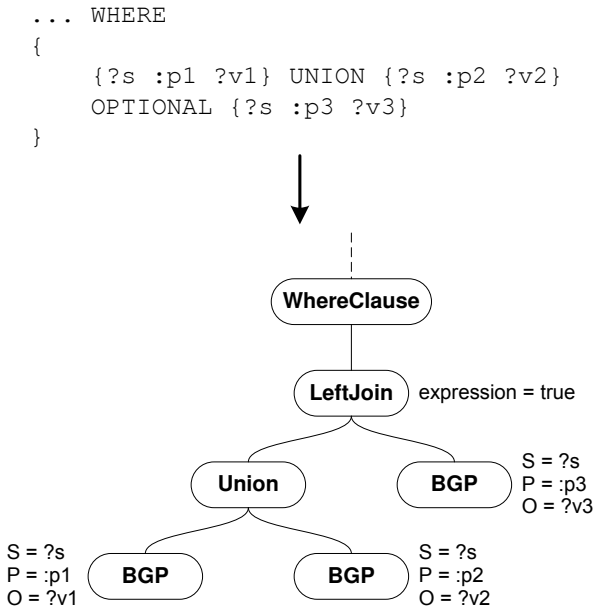


Figure 5.4: Sample Transformation of a *WHERE* Clause

mechanisms to perform best at the lowest possible level of execution, being inside the data store itself. Also, the query engine would only be a proxy for the data store's query engine, relieving it from performing the actual evaluation itself.

The *YARS* (Yet Another RDF Storage) implementation, on the other hand, approaches the query evaluation in a less generic manner [29]. A specific data store is introduced [20], and query evaluation is tailored for this specific store. As a result, YARS is completely dependent on the data store. This is quite advantageous with regards to performance, however, as the entire evaluation is tuned solely for a single data store, allowing for the evaluation to make use of all mechanisms available in the data store.

5.3.2 Approaching Mars

As the focus of this thesis is to explore the possibilities of extending the capabilities of Mars specifically, tight integration with Mars and the corresponding data store is a matter of course. The SPARQL algebra is consequently transformed into Mars operators, which is exclusively executed inside the search engine. Thus, RDF data or any metadata from the system catalog never leave the Mars query engine before the result is returned.

5.4 From Algebra to Mars Operators

When a complete representation of a query, containing solution modifiers and algebra patterns, is ready, the actual evaluation of the query can be performed. This process is handled by the Mars query engine.

As far as possible, evaluation of SPARQL queries should reuse existing Mars operators. A central task is thus to identify an equivalent set of Mars operators for each SPARQL algebra pattern.

Once the equivalent Mars operators are identified, the query object holding the SPARQL algebra and other solution modifiers can be transformed into a tree of Mars operators. This tree serves as a definition for which data needs to be fetched and the operations that have to be applied to this data before presenting the result to the user. The implementation of the transformation from query objects to trees of Mars operators and having Mars evaluate these operators is further discussed in Section 6.6.

5.4.1 Graph Patterns

The SPARQL algebra graph patterns include *BGP* (basic graph pattern), *Join*, *LeftJoin*, *Union* and *Graph*. The *LeftJoin* and *Graph* patterns are both ignored in the transformation procedure as the needed functionality is present in Mars yet.

The Mars operators used to achieve the behavior of the graph patterns include the *Map*, *Select*, *Lookup*, *Sort* and *MergeJoin* operators.

Basic Graph Patterns

To simplify the method of transforming a *BGP* containing multiple triple patterns, such *BGPs* are split into multiple new *BGPs*, each containing only a single triple pattern. These are further connected using *join* graph patterns in order to achieve the behavior specified for a *BGP* with multiple triple patterns.

A *BGP* graph pattern consisting only of variables should be transformed into a *Scan* operator returning all triples, followed by a *Map* operator for mapping the variables appropriately. As no *Scan* operator is currently available in Mars, queries containing such BGPs are not supported. BGPs consisting only of blank nodes and variables are not supported either, as the *Lookup* operator is only capable of looking up exact terms. Looking up a blank node would require looking up all terms prefixed as blank nodes.

A BGP consisting of other elements than just variables and blank nodes is transformed into a *Lookup* operator, possibly a *Select* operator, and finally a *Map* operator, as shown in Figure 5.5. The *Lookup* operator outputs records with document ID and predicate/object, subject/object or subject/predicate pair fields, depending on the kind of triple element used

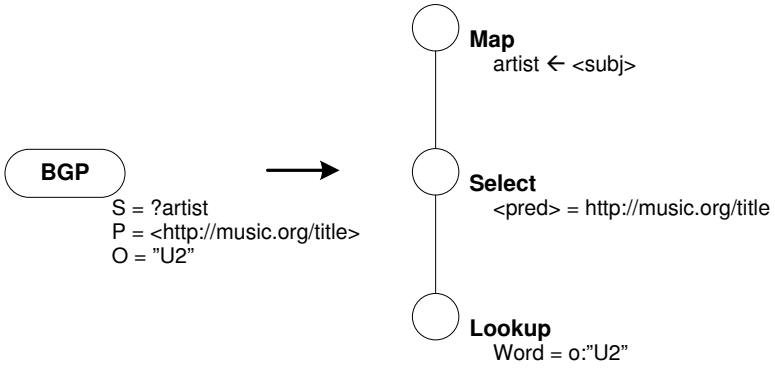


Figure 5.5: BGP Transformation

to perform the lookup. The sample lookup is performed on *o:"U2"*, that is, the object triple element, resulting in a subject/predicate pair.

If any of the two BGP triple elements not used to perform the lookup is either a literal or a URI, a *Select* operator is added to filter the records returned by the *Lookup* operator. As the sample BGP contains a URI predicate in addition to the object literal used in the lookup, a *Select* operator is used to filter records matching the predicate.

Finally, a *Map* operator is used to map any variables or blank nodes in the BGP to the corresponding triple field, based on the triple element type.

Join

A *Join* graph pattern is transformed either to an equi-join or to a Cartesian product depending on whether the two operands have any variables in common or not.

An equi-join is achieved using a *Map* operator, followed by a *Sort* operator and finally a *MergeJoin* operator, as shown in Figure 5.6.

The *Map* operator is needed to coordinate the order of the record fields for the two inputs to be joined. Common variable fields are identified and placed first, in the same order for both inputs.

As the *MergeJoin* operator requires the input to be sorted on common fields on which to perform the equi-join, a *Sort* operator is needed. Finally, the *MergeJoin* operator performs the join on the *join-prefix* first fields of the two inputs.

Union

A *union* graph pattern is transformed into a *Union* operator, as shown in Figure 5.7. As the SPARQL specification defines union as non-distinct, the

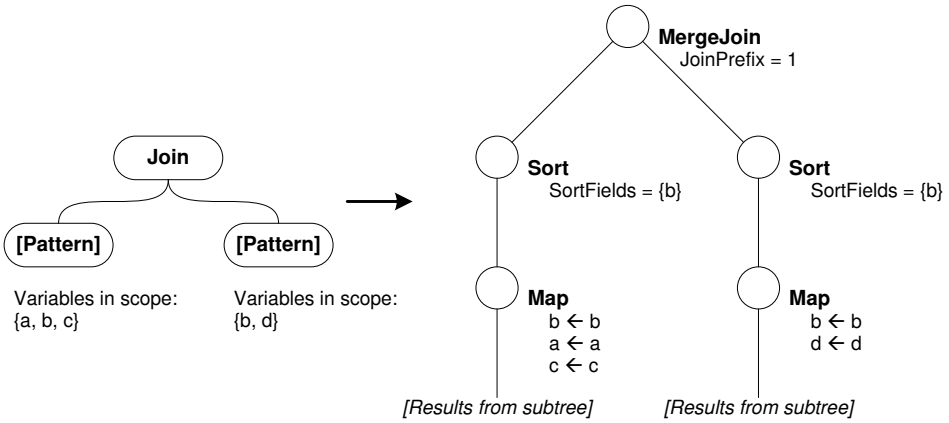


Figure 5.6: Join Transformation

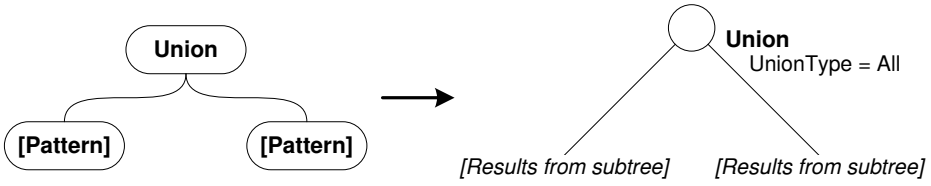


Figure 5.7: Union Transformation

operator is parameterized to include all records from both inputs, including any duplicates.

5.4.2 Solution Modifiers

The SPARQL algebra solution modifiers include *order*, *projection*, *distinct*, *reduced*, *offset* and *limit*, as described in Section 2.2.3. In accordance with the SPARQL specification, the modifiers are applied in the order shown in Figure 5.8.

As the *reduced* modifier simply *permits* duplicates to be eliminated from the solution set, it is completely ignored in the transformation procedure. No well-defined behavior exists beyond that duplicates may or may not be eliminated. Thus, ignoring this modifier and not eliminating any duplicates is still in accordance with the SPARQL specification.

The Mars operators used to achieve the behavior of the solution mod-

Figure 5.8: SPARQL Specification Solution Modifier Ordering

```
BASE <http://music.org/>
SELECT ?title
WHERE
{
  ?album <artist> "U2" .
  ?album <title> ?title
}
ORDER BY ?title
OFFSET 20
LIMIT 10
```

Figure 5.9: Composite SPARQL Query

ifiers include the *Sort*, *ProjectDistinct*, *Map* and *Trim* operators, which are described in Section 2.3.2.

The behavior of the *order* modifier is achieved using the *Sort* operator, as the names suggest. *Projection* is achieved using the *Map* operator, and the behavior of the *offset* and *limit* modifiers are achieved using the *Trim* operator. The behavior of the *distinct* modifier is achieved using the *Sort* operator followed by the *ProjectDistinct* operator, as the latter expects the input records to be sorted in advance.

Regarding the ordering rules described in Section 2.2.3, the *Sort* operator does not distinguish between different resource types, and will sort all resources lexicographically. This is a limitation that will be apparent in the resulting prototype, breaking with the SPARQL specification.

5.5 Example: Finding Album Titles

To illustrate the entire process of transforming a SPARQL query into a Mars operator tree, this section presents an example query and walks through the different transformation steps. Note that this example does not discuss the abstract syntax tree representation. For an elaborate discussion on abstract syntax trees, see Chapter 3.

The example query in question is shown in Figure 5.9. The first triple pattern matches all albums with an *artist* property of "U2", while the second pattern matches all album title properties for all albums. The result is the titles of all of U2's albums, sorted alphabetically by the album title. The *OFFSET* and *LIMIT* keywords specify typical values for a paged result, returning only ten results, skipping the first twenty.

First, the query is parsed and transformed into the intermediate representation as discussed in Section 5.2. The result of this process is the representation illustrated in Figure 5.10.

The root *Query* object holds values that are generic for any query type. *BASE* declarations are examples of such values. For this query, the root object thus keeps the URI *http://ex.org/music* in its *Base* property.

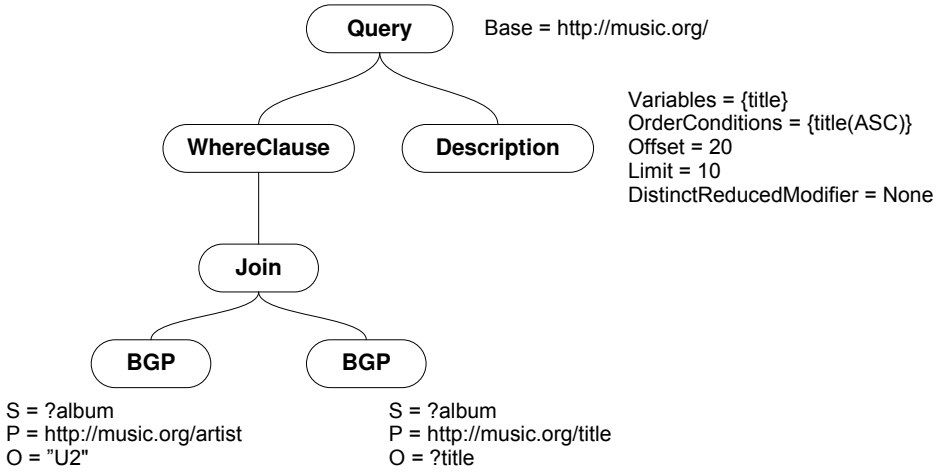


Figure 5.10: Intermediary Query Representation

Besides *BASE*, the remaining modifiers are specific to this being a *SELECT* query. Hence, these values are stored in corresponding properties of the root's *Description* child node.

The *WHERE* clause is transformed into SPARQL algebra and stored in the root's *WhereClause* node. Each of the two triple patterns is stored in a *BGP* pattern which contains both the value and the type of each triple node. For instance, the value of the subject node of the first triple is "album", while the type is *variable*. As these two triples describe required properties of the same results, a *Join* pattern is the parent of the *BGP* patterns.

This intermediate representation makes reasoning about the query much less complicated, compared to having to do the equivalent reasoning directly on the abstract syntax tree. In addition to evidently being a more compact data structure, this representation has a more intuitive location of the query properties.

The final transformation step is from the intermediate query representation to the full-fledged Mars operator tree as discussed in Section 5.4. For *SELECT* queries, the query solution modifiers are applied as operators in a strict serial order, directly at the root. Below these operators, the subtree constituting the transformed SPARQL algebra from the *WHERE* clause is added. The resulting operator tree for this example is shown in Figure 5.11.

The two leaf nodes are both *Lookup* operators performing index lookups on the value of their *Word* properties. The first triple has two known values, leading to an extra *Select* operator which is not present for the second triple. The following *Map* operator makes sure the correct values are stored in fields with names corresponding to the variable names. This concludes the transformation of the two *BGP* patterns from the SPARQL algebra.

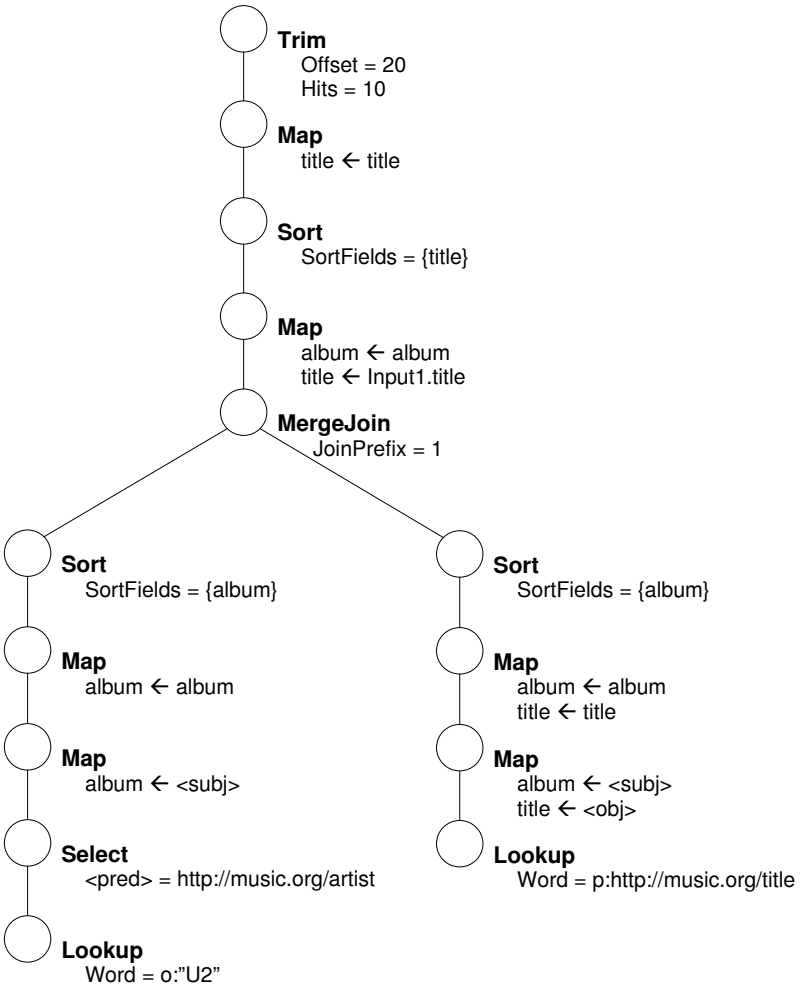


Figure 5.11: Complete Operator Tree

The *Join* pattern from the SPARQL algebra is mainly realized by the *MergeJoin* operator. Its *JoinPrefix* property is set to 1, indicating that the first of the fields is used as the join key. This operator also requires that all input is sorted on the join key. Thus, two *Sort* operators are added as immediate children, sorting on the *album* field. It is also required that the fields used as join key are the first fields of every record. The *Map* operators below the *Sort* operators ensure this. Another peculiarity of the *MergeJoin* operator is that fields which are not used as join key are prefixed in the output to ensure uniqueness. The *Map* parent operator simply removes this prefix. This concludes the transformation of the *WHERE* clause.

The remaining operators are added to achieve the various solution modifiers of the query. The *Sort* operator orders the output alphabetically by the *title* field. The *Map* operator performs a projection by specifying the only field of its output is a *title* field containing the value of the *title* field of its input records. Finally, the behavior specified by the *OFFSET* and *LIMIT* keywords is achieved by the *Trim* operator at the root, by setting appropriate values for its *Offset* and *Hits* properties.

Chapter 6

Implementation

This chapter discusses the implementation of the Mars component, from the transformation of abstract syntax trees, via the intermediate query representation, to the operator tree evaluable by the Mars query engine. Automated testing is also employed to a certain degree. This is discussed towards the end of the chapter.

6.1 Priorities for the Prototype

For this thesis, a prototype SPARQL query service for Mars is created. The top priority of this prototype is to support *SELECT* queries on a single RDF data set, consisting only of triple patterns in their *WHERE* clauses. This requires index lookups in triple patterns and merging of lookup results according to graph patterns.

Further, solution modifiers, such as ordering and requiring only distinct values, can easily be added in a pipe-and-filter fashion immediately before the outputting the results. The *Union* pattern is also realizable using built-in Mars operators. General filters are currently not supported in Mars, and is consequently not supported in the prototype. Left joins originating from the *OPTIONAL* keyword, typically containing filters, are not prioritized as well as not being realizable with the available Mars operators.

ASK, *CONSTRUCT* and *DESCRIBE* queries is not supported, nor will *SELECT* queries against data sets other than the default one.

Query optimization is not prioritized for the prototype. Rather, the focus is on implementing as much functionality as possible to facilitate demonstration purposes. When the basics are implemented, remaining pieces of the SPARQL specification can be implemented incrementally.

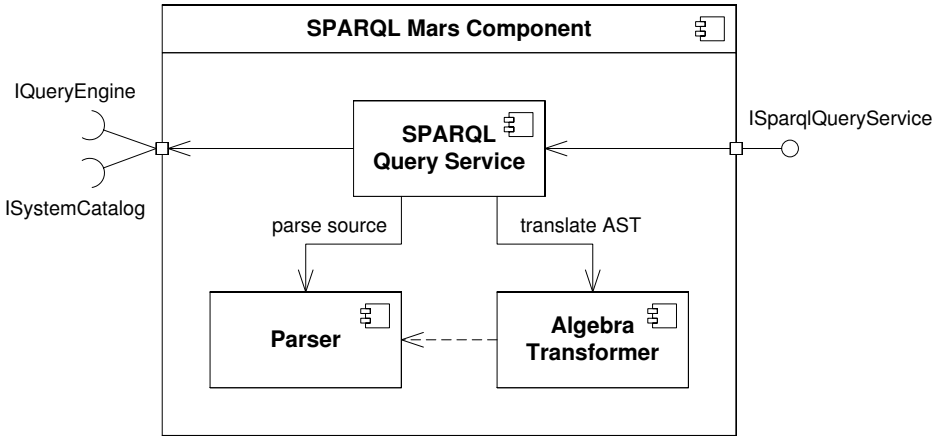


Figure 6.1: Overview of System Components

6.2 Overall System Description

The overall structure of the system is shown in Figure 6.1. The resulting Mars component consists of three major sub-components. The *Parser* component is responsible for converting SPARQL source code into abstract syntax trees. This component was developed prior to this thesis [4], and its principles are described in Section 3.

The *Algebra Transformer* component further transforms abstract syntax trees to an intermediary representation based on SPARQL algebra. This component does not use the parser component directly, but depends on the data types it defines. The principles of this transformation process are discussed in Section 5.2, while the implementation is described in Section 6.5.

The *SPARQL Query Service* component is responsible for the system's public API. When a query execution is requested, the parser and algebra transformer is used to construct SPARQL algebra. Based on this, a Mars operator tree is constructed as described in Sections 5.4 and 6.6. This tree of operators is then passed to the *Mars Query Engine*. The results from this evaluation process is formatted as XML and returned to the caller.

The Mars runtime environment is responsible for providing the prototype component with a reference to the Mars query engine. This is done using a technique called *Dependency Injection*[1]. By reading the attributes of the Mars component, the runtime will discover a dependency for an *IQueryEngine* object. Likewise, the runtime will discover that our component exposes an *ISparqlQueryService* object which can be injected into other components which depend on the query service.

[1] A supplementary introduction to DI can be found at `http://martinfowler.com/articles/injection.html`.

Details about the architecture of the SPARQL Query Service are further discussed in Section 6.7.

6.3 SharQL Parser Project Modifications

The SharQL parser project has been somewhat modified during the implementation of the Mars component. Modifications range from essential extensions to pure convenience patches to ease the processing of the abstract syntax tree.

6.3.1 Visitor Pattern Processing Order Option

The traversal of the abstract syntax tree was performed in a depth-first-manner, processing a node's children before processing the node itself. Processing the abstract syntax tree in this manner proved rather cumbersome while deducing different syntactic sugar constructs because of numerous upward dependencies in the syntax tree.

Depending on the specific syntactic sugar construct being resolved, the most reasonable approach varies quite a lot in regards to node processing order. In order to resolve different constructs in the best suitable way, being able to control the processing order is essential.

The Visitor pattern traversal algorithm has been modified to accept an optional processing order parameter to control whether a node is processed before or after its children. The latter is still the default option.

6.3.2 Visitor Pattern Reflection-Based Type Identification

The initial visitor implementation identified the element type based on a *NodeBase* property. Reflection is now used, resulting in significantly cleaner code. The overhead that is introduced with using reflection for this purpose will be negligible compared to the workload of executing the query.

Consequently, the *IVisitor* interface no longer uses a *string* parameter for its indexer. The signature of the indexer is now as shown in Figure 6.2, accepting *System.Type* objects as its parameter. Such objects can be obtained by calling the *typeof()* operator on any class or calling the *GetType()* method of any object. Hence, the *Accept* method will typically include the code `visitorObj[GetType()](this);` Figure 6.3 shows the initial part of the new dispatch sequence.

6.3.3 INode Interface Inheriting from ICloneable

When transforming syntactic sugar constructs in the abstract syntax tree, several nodes have to be duplicated and reused in the replacement con-

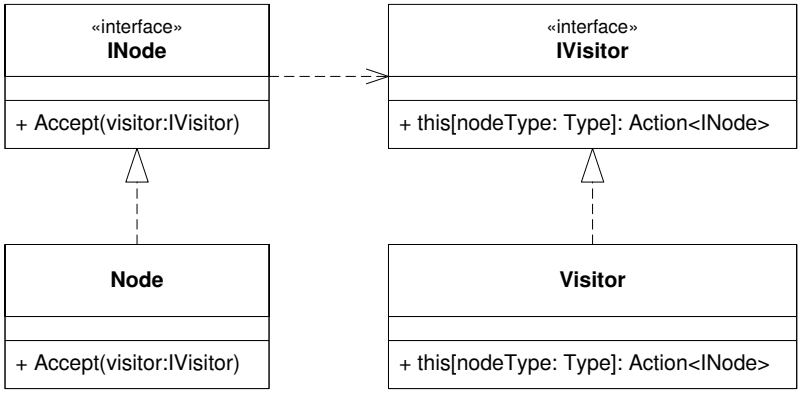


Figure 6.2: Visitor Pattern Delegate

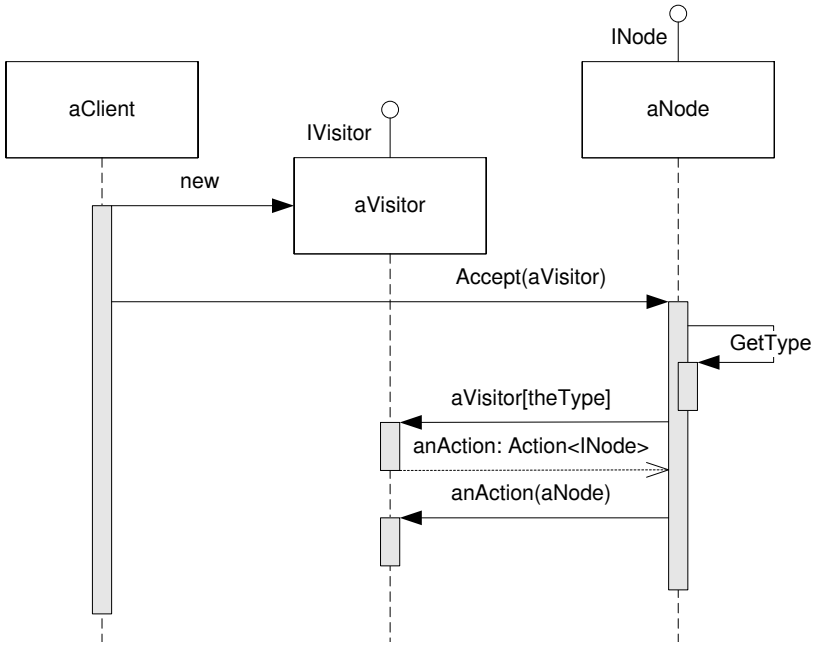


Figure 6.3: Visitor Pattern Delegate Dispatch Using Reflection

structs. For instance, when expanding a list specified using syntactic sugar, a blank node has to be introduced and used twice per list element.

The *INode* interface implemented by the *NodeBase* abstract class of which all AST nodes are descendants, has further been modified to inherit from the .NET framework *ICloneable* interface to support cloning. The latter interface's methods have been implemented in a general manner in the NodeBase abstract class, allowing for a node to be duplicated simply by calling its *Clone()* method, resulting in a shallow copy of the node and all its child nodes.

6.4 Syntactic Sugar

The implementation supports all syntactic sugar constructs defined in the SPARQL specification. The constructs are transformed into their core SPARQL language counterparts using syntactic sugar resolving visitors before the query is evaluated.

6.4.1 Implicitly Data-Typed Literals

Numeric and boolean literals may be specified as numbers and *true* or *false* string literals, respectively, in a query. This is by far the simplest kind of syntactic sugar, and it is rather easily transformed.

In the AST, these literals are represented as *NumericLiteral* and *BooleanLiteral* nodes, respectively, and are simply replaced by *RdfLiteral* nodes with the respective string value and a child *IriRef* node containing the data type URI. The *RdfLiteral-IriRef* node combination corresponds to the full form literal representation that general data-typed literals have in the abstract syntax tree.

6.4.2 Shared Subject Triple Lists

Triple lists sharing the same subject are probably the most complicated syntactic sugar in SPARQL. Figure 6.5 shows the extract of the abstract syntax tree for the query shown in Figure 6.4.

The shared subject is represented by the *PrefixedName* node directly below the *TriplesSameSubject* node. The predicates are directly below *PropertyList* nodes and the objects are placed directly below *ObjectList* nodes inside the *PropertyList* nodes. Transforming this construct in a children-before-parent order, starting off with the leaf object *:f*, requires knowledge of the above abstract syntax tree nodes. Traversing the tree all the way up to the subject *:a* along with the predicate *:d* is not as trivial as starting off from the top in a parent-before-children order.

```
:a :b :c . :d :e , :f

  ⇓

:a :b :c .
:a :d :e .
:a :d :f
```

Figure 6.4: SPARQL Shared Subject Triple List Syntactic Sugar Sample

The shared subject triple construct is transformed by identifying the shared subject node before traversing the tree downwards looking for *PropertyList* nodes and *ObjectList* nodes. A *PropertyList* node below another *PropertyList* node represents a predicate and object sharing a common subject, while an *ObjectList* node below another *ObjectList* node represents an object sharing both a common subject and predicate from the ancestor *PropertyList* node. Figure 6.6 shows the abstract syntax tree for the transformed construct presented in Figure 2.9 on page 14.

6.4.3 Blank Node Property Lists

Blank node property lists are basically a variant of shared subject triple lists. The shared subject is omitted from the list, and an induced blank node representing the blank node property list itself is used as a shared subject. If no list is specified inside the `[]` blank node property list operator, the list itself is still replaced by a blank node.

The processing of blank property node lists uses the very same algorithm used for processing shared subject triple lists. The only difference is the initial identification of the shared subject, which is set to be a new, blank node for blank node property lists. Also, as blank node property lists is represented by a blank node, they may be used as subjects or objects in triples and as elements in lists and other constructs. Hence, the context of a blank node property list is of interest and affects the transformation of the construct.

6.4.4 Lists

Lists are a rather trivial syntactic sugar construct, especially because of the strictly recursive abstract syntax subtrees. The traversals of such trees are straight forward compared to those of shared subject triple lists. Figure 6.7 shows the abstract syntax tree for the query presented in Figure 2.8 on page 13.

Transforming a list construct simply involves tracking down *Collection* nodes and recursively iterate *GraphNodeList* nodes and extract their contents. Once all list elements have been extracted, a recursive triple block

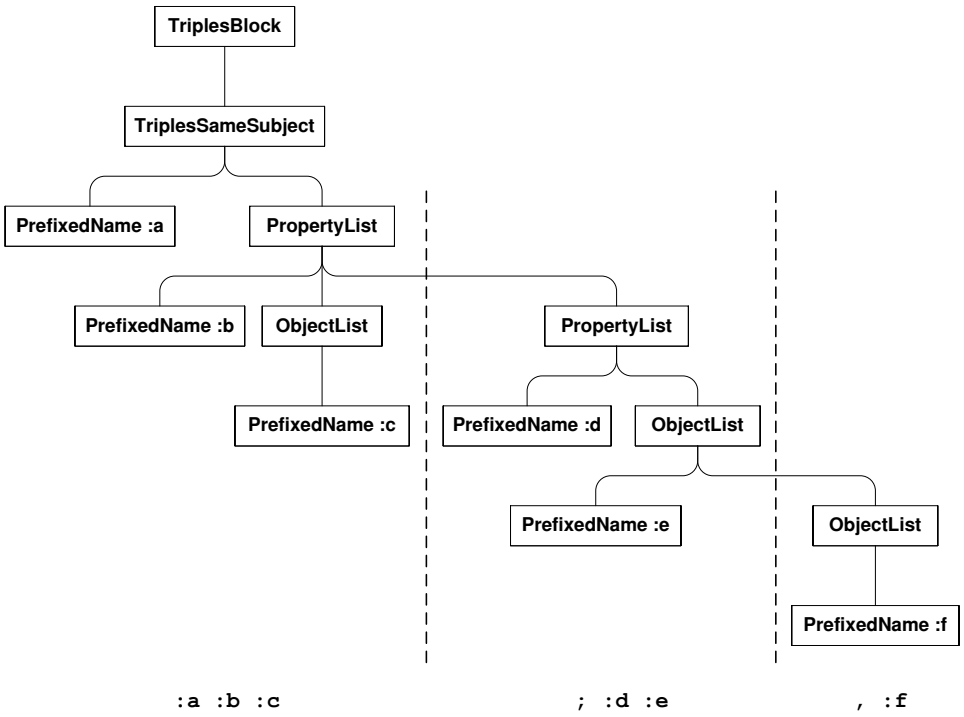


Figure 6.5: Shared Subject Triple List Syntactic Sugar AST Extract

sequence may be created. Figure 6.8 shows the abstract syntax tree for the result after applying the transformation to the AST in Figure 6.7.

6.4.5 Prefix Expansion

Prefix expansion is not syntactic sugar with regards to the SPARQL specification, but the expansion of prefixes is handled in the very same manner as syntactic sugar.

The abstract syntax tree is traversed looking for *IriRef* nodes to check if their URIs are relative and should be prefixed with the *BASE* URI. Also, *PrefixedName* nodes originating from using a prefixed URI are replaced by *IriRef* nodes having the prefixed URI expanded.

constructed from that of the prefix used appended and the corresponding identified from the *PrefixedName* node.

6.4.6 Identification of Unlabeled Blank Nodes

Blank nodes are usually labeled in the queries. Some blank nodes, however, are not explicitly specified and are consequently not labeled by the query writer. Such unlabeled blank nodes arise as a result of deducing syntactic sugar or by using specific operators.

Figure 6.6: Shared Subject Triple List AST Extract

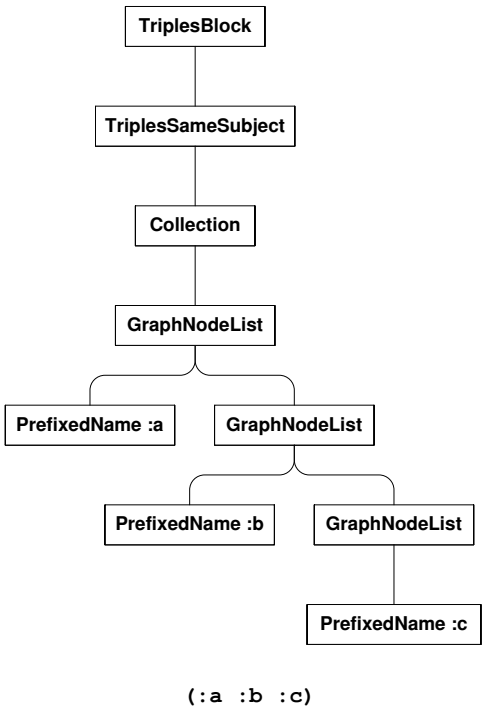


Figure 6.7: Lists Syntactic Sugar AST Extract

The blank node property list operator, `[]`, either empty or containing a predicate-object property list like shown in Figure 6.9, induces a new blank node. Also, syntactic sugar like lists, described in Section 2.2, introduces several placeholder blank nodes.

Such blank nodes induced during query evaluation are labeled internally using GUIDs to guarantee uniqueness [35].

6.4.7 Example: The Literal Explicator Visitor

To illustrate how the Visitor pattern is used to resolve syntactic sugar, this section will present the visitor in the implementation which is responsible for explicating the implicitly data-typed literals presented in Section 6.4.1.

This visitor class is called *LiteralExplicatorVisitor* and implements the *IVisitor* interface. As discussed earlier, the only member of this interface is an indexer which takes a *Type* as its only argument and returns a delegate which references the method to be called by each node. The code for this indexer is shown in Figure 6.10. Only *NumericLiteralNode* and *BooleanLiteralNode* objects are handled by this visitor; other nodes are ignored by letting the indexer return *null*.

The *explicateNode* method reference which is returned by the indexer is shown in Figure 6.11. This method casts the node reference to an appro-

Figure 6.8: Lists AST Extract

```
[] :a [:b :c ; :d :e , f]
```

Figure 6.9: SPARQL Blank Node Property List Syntactic Sugar Sample

```
public Action<INode> this[Type nodeType]
{
  get
  {
    if (nodeType == typeof(NumericLiteralNode) ||
        nodeType == typeof(BooleanLiteralNode))
    {
      return explicateNode;
    }
    else
    {
      return null;
    }
  }
}
```

Figure 6.10: Indexer of *LiteralExplicatorVisitor*

priate type and delegates the explication to other methods. Finally, the old AST node is replaced by the new and explicit form. This replacement is performed by a general helper method, transferring parent and children references.

The implementation of the *explicateBooleanLiteralNode* method is shown in Figure 6.12. This method simply returns what would have been the result if the explicit notation had been parsed, namely an *RdfLiteralNode* with two children: a *StringLiteralNode* node containing the value and an *IriRefNode* containing the data type identifier. The *explicateNumericLiteralNode* method works equivalently and its implementation will not be discussed here.

By applying this visitor to an AST, all numeric and boolean literal nodes will, in effect, replace themselves with their equivalent explicit representation.

6.5 Intermediate Query Representation

As discussed in Section 5.2, SPARQL queries are transformed into an intermediate representation based on SPARQL algebra. The base of this representation is the *Query* class, shown in Figure 6.13.

Each query may specify a base and any number of named prefixes. Also, depending on the type of query, a query description is stored as a *QueryDescriptionBase* object. *QueryDescriptionBase* is an abstract class, representing any of the concrete implementations. The details of these imple-

```
private void explicateNode(INode node)
{
  INode newNode = null;

  if (node is NumericLiteralNode)
  {
    newNode = explicateNumericLiteralNode((NumericLiteralNode)node);
  }
  else if (node is BooleanLiteralNode)
  {
    newNode = explicateBooleanLiteralNode((BooleanLiteralNode)node);
  }

  if (newNode != null)
  {
    Util.ReplaceNode(node, newNode);
  }
}
```

Figure 6.11: The *explicateNode* Method of *LiteralExplicatorVisitor*

```
private RdfLiteralNode explicateBooleanLiteralNode(BooleanLiteralNode node)
{
  return new RdfLiteralNode(
    new StringLiteralNode { Value = node.Value ? "true" : "false" },
    new IriRefNode { Value = "http://www.w3.org/2001/XMLSchema#boolean" }
  );
}
```

Figure 6.12: The *explicateBooleanLiteralNode* Method of *LiteralExplicatorVisitor*

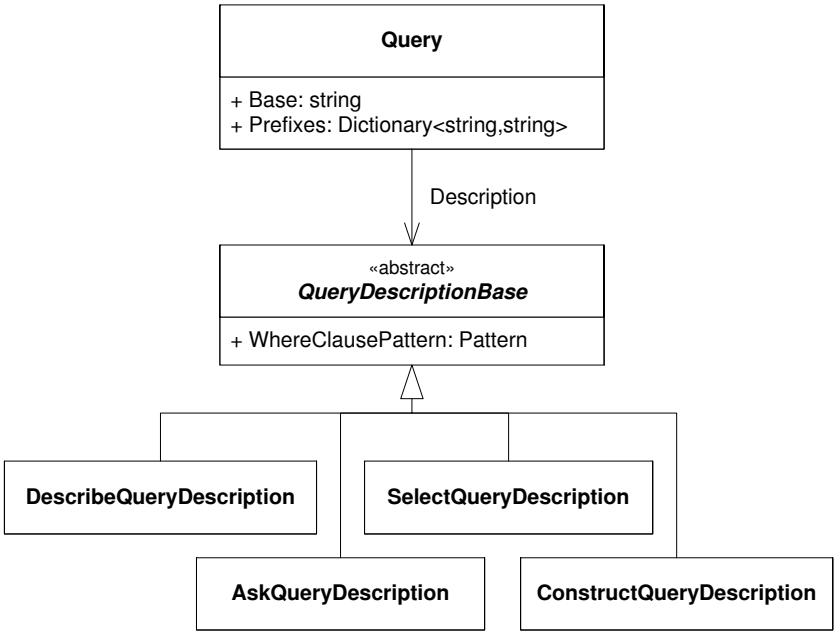


Figure 6.13: Overview of the *Query* Class

mentations are omitted from the figure.

The type of the query description dictates the type of the query. All description types contain *WHERE* clauses[2]. These are stored in the *WhereClausePattern* property as SPARQL algebra expressions.

SPARQL algebra expressions are built using classes that derive from the abstract *Pattern* class, as shown in Figure 6.14. These classes correspond to the SPARQL algebra operators presented in Section 5.2.1, except for the *EmptyPattern* class which is used as a placeholder where no pattern has yet been determined.

Note the *VariablesInScope* property of every pattern. This list of string objects contains the name of all the variables that are visible to the pattern. This collection is introduced at the pattern level to ease the reasoning when performing the transformation into Mars operator trees, as discussed in Section 6.6. For the *BasicGraphPattern*, the *VariablesInScope* property is populated with the names of variables occurring in the corresponding basic graph pattern of the query. For other patterns, it is assigned the set-union of the children's *VariablesInScope* properties.

The *BasicGraphPattern* class, corresponding to the *BGP* pattern of SPARQL algebra, is simply a list of *Triple* objects, as shown in Figure 6.15. Each *Triple* object holds three references to *Resource* objects, representing its subject, predicate and object. Each *Resource* object has a textual value as

[2]In *DESCRIBE* queries, the *WHERE* clause is optional

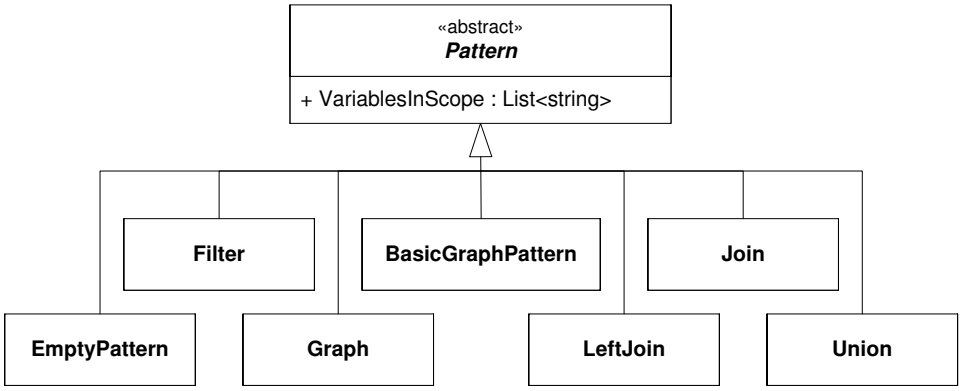


Figure 6.14: Classes Used to Build SPARQL Algebra Expressions

well as a *Type* indicating the role of this value.

6.5.1 Transforming the Abstract Syntax Tree

The process of transforming the abstract syntax tree from the parser into SPARQL algebra is done recursively. An overview of the method is presented in Section 5.2. The prototype implementation is based around a *transform* method which takes a single *INode* argument. The concrete type of this argument decides to which of the several *transform[...]* methods the actual transformation is delegated.

Each of the *transform[...]* methods is implemented as suggested by pseudo code in Figure 5.3 on page 53. For completeness, the entire source code for translating *WHERE* clauses into SPARQL algebra is provided in Appendix B.

The final result of this transformation is a SPARQL algebra expression represented by a *Pattern*-derived root node. As mentioned, this reference is stored in the *WhereClausePattern* property of the query's description object. However, there are other factors that affect the result of a query.

For instance, the result of a *SELECT* query can be affected by a distinct/reduced modifier or a specified ordering. All these factors are captured and stored in designated properties of the *SelectQueryDescription* class, as shown in Figure 6.16. If specified, these values are easily read directly from the AST.

Solution limits and offsets are specified as integers in queries. In C#, the *int* type is not nullable. Hence, the *Limit* and *Offset* attributes of the *SelectQueryDescription* class has to be of type *Nullable<int>*. The *Nullable<T>* type is a wrapper for value types, allowing *null* to be assigned if a value is not available.

Conducting the entire transformation from abstract syntax tree to the intermediary *Query* object representation is the responsibility of the static

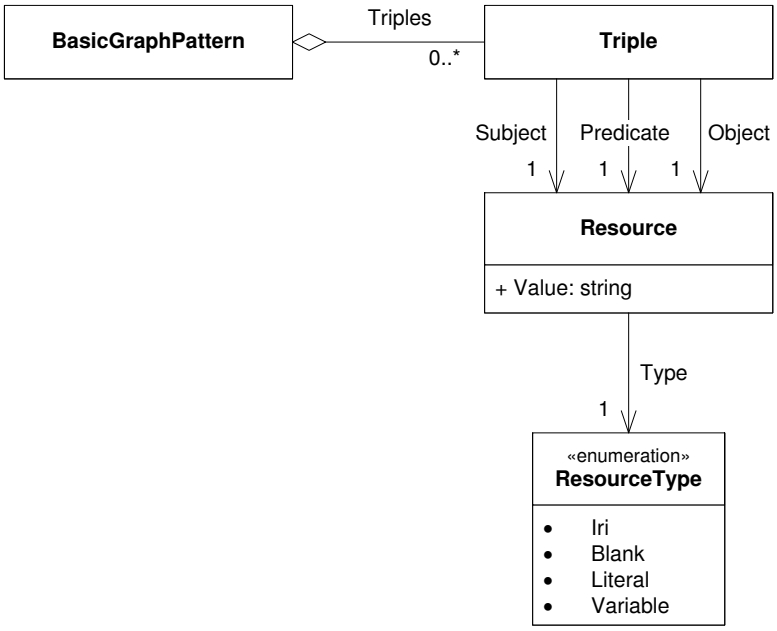


Figure 6.15: Overview of the *BasicGraphPattern* Class

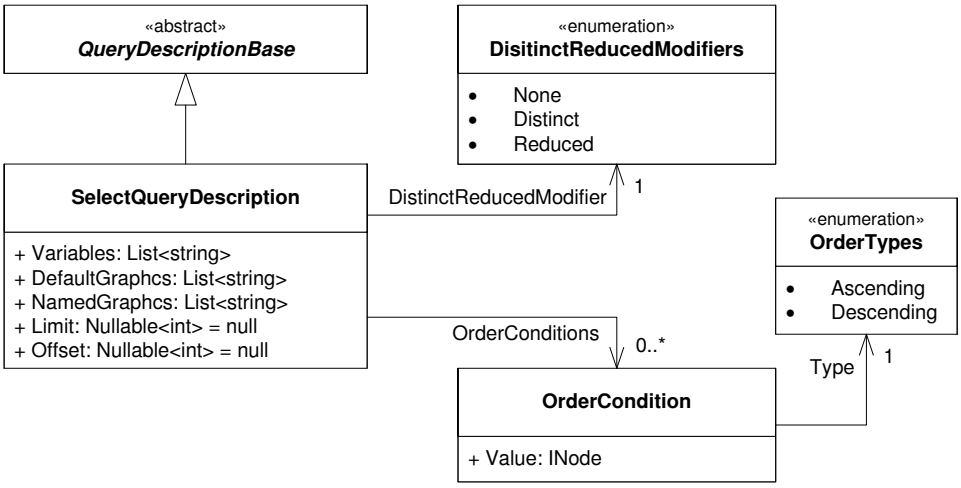


Figure 6.16: Overview of the *SelectQueryDescription* Class

```
public static Query Transform(QueryNode queryNode)
{
  Query query = CreateQueryFromPrologueNode(
      queryNode.Children[0] as PrologueNode);

  (···)

  if (queryNode.Children[1] is SelectQueryNode)
  {
    query.Description = SelectQueryTransformer
      .TransformSelectQueryNode(queryNode.Children[1] as SelectQueryNode);
  }

  (···)

  else
  {
    throw new ArgumentException(...);
  }

  return query;
}
```

Figure 6.17: Conducting the Transformation Process.

Transform method of the *QueryTransformer* class. This method inspects the syntax tree and decides what type of query is being processed. The transformation process is then delegated to other methods. This concept is shown in code in Figure 6.17.

6.5.2 Example: Transforming a part of the AST

While the source code of the *WHERE* clause transformer is provided as Appendix B, one of its methods will be discussed in this section to illustrate how this class operates. The method to be discussed is *transformGroupOrUnionGraphPattern* which corresponds to the second `If`-statement in the pseudo code in Figure 5.3 on page 53.

The source code for this method is listed in Figure 6.18. Apart from the population of the *VariablesInScope* collection, there is an apparent equivalence between the pseudo code and the actual C# code. The *Children* collection of the specified node is enumerated and each child is transformed separately. If more than one child is present, they are aggregated with *Union* constructs and the *VariablesInScope* property of each union is populated with the set-union of the *VariablesInScope* properties of its two children. The *Union* class is merely a data container, without any logic. Its implementation is shown in Figure 6.19.

The call to *Where* in the *foreach* statement is passed a lambda expression which ensures that only non-null values in the *Children* collection is

```
private static Pattern transformGroupOrUnionGraphPattern(
    GroupOrUnionGraphPatternNode node)
{
  Pattern pattern = null;
  foreach (INode child in node.Children.Where(c => c != null))
  {
    if (pattern == null)
    {
      pattern = transform(child);
    }
    else
    {
      pattern = new Union(pattern, transform(child));

      pattern.VariablesInScope.AddRange(
        ((Union)pattern).FirstPattern.VariablesInScope.Union(
        ((Union)pattern).SecondPattern.VariablesInScope));
    }
  }
  return pattern;
}
```

Figure 6.18: The *transformGroupOrUnionGraphPattern* of *WhereClauseTransformer*

```
public class Union : Pattern
{
  public Pattern FirstPattern { get; private set; }
  public Pattern SecondPattern { get; private set; }

  public Union(Pattern firstPattern, Pattern secondPattern)
  {
    FirstPattern = firstPattern;
    SecondPattern = secondPattern;
  }
}
```

Figure 6.19: The *Union* Class

included in the iteration. This is the LINQ[3]-construct for filtering a collection. Likewise, the call to *Union* inside *VariablesInScope.AddRange* is the LINQ-construct for performing set-unions.

By passing every child to the *transform* method, this implementation uses indirect recursion to iteratively transform the entire sub-tree into the form required by the intermediate query representation.

6.6 Operator Trees

The transformation of the intermediate query representation into Mars operators is discussed in Section 5.4. The implementation of the transformation is closely based on the transformation algorithm, although several methodical approaches are not directly transferable.

As described in the beginning of Chapter 5, Mars evaluates queries based on operator graphs, but in the context of this thesis only graphs qualifying as trees are used, and the term operator *tree* is consequently used.

The operator trees are constructed top-down, using the set of operators described in Section 2.3.2. In addition, a special *OutputOperator* is used as the root operator. The construction itself is a rather trivial procedure, consisting of appending operators to the desired position in the operator tree. Once the operator tree is fully constructed, it is validated. During validation, the ordering and parameterization of operators are checked. In addition, necessary metadata for nodes and edges in the operator tree is supplied, making the operator tree ready for execution.

6.6.1 Transforming Algebra Graph Patterns

The SPARQL algebra graph patterns are transformed into Mars operators as described in Section 5.4.1. The implementation corresponds closely to the described method. However, complementary implementation details for some of the graph patterns are worth mentioning.

Basic Graph Pattern

A BGP is transformed into a *LookupOperator*, possibly a *SelectOperator*, and finally a *MapOperator*. The lookup may only be performed on a literal or a URI, as variables or blank nodes may not be specified as a lookup term. If a triple pattern contains at least one literal or URI, however, the remainder of the triple pattern may contain both variables and blank nodes.

The *SelectOperator* does currently not support using filters containing escaped quotation marks, needed in order to specify literal values, which

[3]Language Integrated Query: `http://msdn.microsoft.com/en-us/data/cc299380.aspx`

are indexed on the form `o:"[value]"^^[data type]`, including the quotation marks. Thus, if such an element is present in a triple pattern, the lookup has to be performed on that element, and the filtering on the remainder of the triple elements. However, the object element is expected to be the most selective element in the RDF triple, which circumvents the *SelectOperator* limitation.

Consequently, lookups are preferentially performed on triple objects, subject or predicates, in that order, on the first element whose value are either a literal or an URI. Additional literal or URI value elements are filtered after the lookup.

Variables and blank nodes present in a triple pattern are mapped using the *MapOperator*. Variables are mapped using the names given in the query string, whereas blank node names are prefixed with a unique string in order to separate them from variables.

Join

A join is transformed either into an equi-join or into a Cartesian product. In order to determine the kind of join operation to be performed, the *VariablesInScope* property of every pattern, described in Section 6.5, is used to find any common variables.

Unique fields in the incoming records are automatically prefixed with *InputX* in the resulting record set, *X* being the sequence number of the input from which the field originated. Thus, unique fields from the two inputs in a two-way join are prefixed *Input0* and *Input1*, respectively. As a consequence, every join operation has to be succeeded by a remapping of the variable names back to the corresponding names before the join operator.

6.6.2 Example: Constructing an Operator Tree

To illustrate how Mars operators are parameterized and composed into an operator tree, the required code for manually constructing the operator tree for a sample query is shown in Figure 6.20. This code is not part of the implementation; it is simply an attempt to give a glimpse into the basics of operator tree construction.

As the transformation implementation is decomposed into several parts handling different parts of the SPARQL algebra, showing the course of the transformation would require quite a lot of code. Thus, this sample code simply brings together extracts of the pure operator tree construction code analogous to that produced by the transformation process.

The sample query is stated at the very beginning of the code sample. It is a simple query containing only a single basic graph pattern and a projection on the *?title* variable. Corresponding to the RDF sample graph in

Figure 2.1 on page 5, the *?title* variable should be bound to "U2" when the query is evaluated.

Initially the root *OutputOperator* object and the *OperatorFlow* object representing the operator tree are created. The latter contains methods for constructing the tree by adding operators at specific positions in the operator tree.

The basic graph pattern is transformed into three operators. First a *LookupOperator* is used to fetch records from the inverted index *Occurrence2*. Since no literal or URI object triple element is given in the query, the next preferred lookup element is the subject triple element, which qualifies for the lookup as it is a URI. Consequently, all RDF triples having the subject *http://music.org/artist/a3cb23fc*, prefixed by *s:* to match subject triple elements, are looked up.

In addition to the subject triple element, a URI predicate element is specified as well. This calls for additional filtering using the *SelectOperator*. The record set produced by the *LookupOperator* contains three fields; *DocID*, *Value1* and *Value2*, of which *Value1* and *Value2* represent the corresponding triple predicate and object elements, respectively, since the lookup was performed on the triple subject element. The filter is consequently set to match all records where the *Value1* field, the predicate triple element, equals the *http://purl.org/dc/elements/1.1/title* predicate URI.

Next, the *MapOperator* is used to map the *?title* variable to the *Value2* field, that is, the object triple element. Yet another *MapOperator* is used to perform the projection. As the *?title* variable is the only variable present in the record set at this stage, performing a mapping from the variable to itself is not necessary. The prototype's transformation process translates the SPARQL algebra naïvely, however, unconcerned with optimization issues like this one.

Finally, the operators are added to the tree. This is performed in a top-down manner, where the operators are added as predecessors to each other, in the appropriate order, starting off with the root *OutputOperator* and the projection *MapOperator*.

6.7 Component Architecture

This section describes the architecture of the SPARQL query service and its encapsulating Mars component. An overview of the classes involved is shown in Figure 6.21. Note the custom UML stereotypes *injected* and *exposed* which are used to indicate properties involved in dependence injection.

The central entity of the Mars component is the *SparqlQueryServiceComponent* class. This class encapsulates the entire SPARQL Query Service and exposes it to Mars through the *ISparqlQueryService* interface. By exposing

```
//  SELECT ?title WHERE {
//    <http://music.org/artist/a3cb23fc>
//    <http://purl.org/dc/elements/1.1/title>
//    ?title
//  }
OperatorBase root = new OutputOperator();
OperatorFlow tree = new OperatorFlow(root);

// Basic graph pattern
LookupOperator lookupOperator = new LookupOperator();
lookupOperator.IndexName = "Occurrence2";
lookupOperator.Word = "s:http://music.org/artist/a3cb23fc";

SelectOperator selectOperator = new SelectOperator();
selectOperator.Filter = "Value1 ==
  StringFunctions.ToSafeString(\"http://purl.org/dc/elements/1.1/title\")";

MapOperator mapOperator = new MapOperator();
mapOperator.ParameterMap =
  new Dictionary<IdentifierProperty, ExpressionProperty>()
mapOperator.ParameterMap.Add(
  new IdentifierProperty("title"), new ExpressionProperty("Value2"));

// Projection solution modifier
MapOperator projectionMapOperator = new MapOperator();
projectionMapOperator.ParameterMap =
  new Dictionary<IdentifierProperty, ExpressionProperty>();
projectionMapOperator.ParameterMap.Add(
  new IdentifierProperty("title"), new ExpressionProperty("title"));

// Add operators to operator tree
tree.AddPredecessor(root, projectionMapOperator);
tree.AddPredecessor(projectionMapOperator, mapOperator);
tree.AddPredecessor(mapOperator, selectOperator);
tree.AddPredecessor(selectOperator, lookupOperator);
```

Figure 6.20: Operator Tree Construction Sample

Figure 6.21: Architecture of the Mars Component

the service to the dependency injection framework, any other Mars component can request access to the SPARQL Query Service.

SparqlQueryServiceComponent is also responsible for providing the SPARQL Query Service with access to the Mars query engine and system catalog. This is achieved by registering a dependency for an *IQueryEngine* object and an *ISystemCatalog* object, respectively. Only when these dependencies are resolved is the actual SPARQL Query Service object created.

It is important to note that these dependencies are a consequence of the SPARQL Query Service not evaluating the Mars operator trees itself, but passing them to the query engine component. The system catalog is used as a parameter to the operator trees in order for the query engine to find the RDF data.

Most of the plumbing required by Mars components is provided by the *AbstractManageableCoreClient* class. By inheriting from this class, one gets both XML configuration capabilities as well as virtual methods to override in order to act on certain events. The *Activating* method, for instance, is called when all dependencies are resolved, and is overridden in this component to instantiate the SPARQL Query Service.

Also note that this component implements the *IInteractiveComponent* interface. The purpose of this is to provide an interactive shell to users who run Mars's command line interface.

The *ISparqlQueryService* interface is the interface providing access to the SPARQL Query Service. It consists only of two members, an *ExecuteQuery* method and an *Errors* property. The *ExecuteQuery* method accepts a SPARQL query string and returns a string representation of an XML document containing the results of the query. Errors that occur are enumerated through the *Errors* property.

The transformation from SPARQL algebra patterns to a Mars operator tree, discussed in the previous section, is the responsibility of the *OperatorGraphBuilder* class. Objects from this class require access to the configuration and the system catalog. These are provided by the *SparqlQueryService* constructor which is responsible for instantiating an operator graph builder.

The operator graph builder contains a single public method which transforms a *Query* object into a Mars operator tree. This tree is then validated and passed to the Mars query engine for evaluation.

As mentioned, inheriting from *AbstractManageableCoreClient* provides XML configuration capabilities. For the SPARQL Query Service, the default configuration is as shown in Figure 6.22. The configuration parameters are all related to how and where the index is stored.

At runtime, the configuration is automatically retrieved and stored in an *ISparqlQueryServiceConfiguration* field on the *SparqlQueryServiceComponent* object. The members of this interface exactly match the elements in the configuration XML, as shown in Figure 6.23. Ultimately, this configura-

```
<cc:Configuration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cc="http://www.fastsearch.com/Ceres/Config/Configuration/2008/11">
  <OccurrenceIndex>Occurrence2</OccurrenceIndex>
  <FirstValueField>Value1</FirstValueField>
  <SecondValueField>Value2</SecondValueField>
  <SubjectPrefix>s:</SubjectPrefix>
  <PredicatePrefix>p:</PredicatePrefix>
  <ObjectPrefix>o:</ObjectPrefix>
</cc:Configuration>
```

Figure 6.22: SPARQL Query Service Configuration

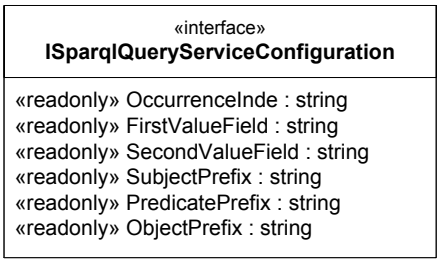


Figure 6.23: Interface of Configuration Object

tion object is passed to the constructor of the operator builder class.

6.8 Testing the Component

This section outlines the measures taken to test the correctness of the prototype component. The testing methods and degree of thoroughness vary among the sub-components. While some tests are fully automated, other tests are based on manual visual inspection. Some tests are even conducted completely by hand, without using any testing frameworks.

6.8.1 Parser Testing

The *Data Access Working Group* of W3C has released a set of test cases, including syntax tests for the SPARQL language [36]. All in all, the test suite contains five syntax test sets of 199 SPARQL queries in total which should either parse or fail to parse. While such tests do not fully test all aspects of a SPARQL parser, they constitute a solid foundation.

The test queries do not specify how a resulting AST should look like. After all, the structure of an AST depends on the application. Thus, it is not possible to verify the correctness of the ASTs produced by the parser, based on these tests alone. The tests neither provide any means of testing the scanner isolatedly. Such tests have to be authored by manually iden-

```
[TestMethod]
public void TestLiteralExplication()
{
  INode node = new NumericLiteralNode()
  {
    IntValue = 123,
    Type = NumericLiteralNode.LiteralType.Int
  };
  explicateLiterals(ref node);
  Assert.IsInstanceOfType(node, typeof(RdfLiteralNode));
  Assert.AreEqual("123",
    ((node as RdfLiteralNode).String as StringLiteralNode).Value);
  Assert.AreEqual("http://www.w3.org/2001/XMLSchema#integer",
    ((node as RdfLiteralNode).IriRef as IriRefNode).Value);
}
```

Figure 6.24: Fully Automated AST Preparation Test

tifying the token stream that should be produced. Once again, this is a natural constraint, given that the tokens chosen, like the AST, depend on the application.

Using the integrated support for automated testing in Visual Studio 2008, the W3C SPARQL Test Suite is completely automated. The running of the test suite is essential in determining the parser's level of conformance with the SPARQL grammar specification, and because of the great importance of running such tests, this should be as easy to carry out as possible.

As the parser used in this component is created and tested as a related but different project, the entire discussion on testing the parser is not recited here. For a complete presentation on the measures taken to test the parser, see [4].

6.8.2 Transformation Testing

Two aspects of the transformation from AST are tested using Visual Studio: the initial preparation of the AST and the following transformation into SPARQL algebra.

An example of a fully automated AST preparation test is shown in Figure 6.24. This test tests the *LiteralExplicatorVisitor* implementation discussed in Section 6.4.7. An AST node representing an implicitly data-typed numeric literal is created and applied the explication visitor as part of the call to the *explicateLiterals* method. Consequently, properties of the resulting node are inspected to make sure the explication was correct.

Other AST preparation tests are more advanced, creating ASTs which are subject to a visitor and then asserting properties on the entire trees. Some surrogate tests have also been created, which, when run in debug mode, will visualize the AST before and after the visitor is applied. These are not tests per se, but have been a helpful tool in debugging many of the

```
[TestMethod]
public void TestDistinctModifier()
{
  Parser parser = new Parser();
  parser.Parse("SELECT DISTINCT * {}");
  Query query = QueryTransformer.Transform(parser.AstRoot as QueryNode);

  SelectQueryDescription description =
    query.Description as SelectQueryDescription;
  Assert.AreEqual(DistinctReducedModifiers.Distinct,
    description.DistinctReducedModifier);
}
```

Figure 6.25: Fully Automated SPARQL Algebra Transformation Test

visitor classes.

A separate set of tests were created to verify the transformation from AST to the intermediate query representation. By parsing queries and passing the resulting ASTs to *QueryTransformer.Transform*, properties on the resulting *Query* object can be verified. An example of such an automated transformation test is shown in Figure 6.25. This particular test verifies the transformation of a query's *DISTINCT* modifier into a property value on the intermediate query description.

Transformation of *WHERE* clauses are tested in a similar fashion, but the resulting transformation is a tree of SPARQL algebra, rather than a simple property on a *Query* object. Hence, these tests traverse entire trees of SPARQL algebra patterns and verify both types and values on each element.

6.8.3 Evaluation Testing

The set of test cases released by the *Data Access Working Group* of W3C used for parser testing also contains numerous evaluation tests. Besides the syntax tests, the test cases contain a total of 241 SPARQL queries and accompanying data and result sets. Several of the evaluation tests do not qualify for testing the prototype, however, as the current implementation only supports a subset of the SPARQL specification, as discussed in Section 7.3.

As the actual evaluation of the operator trees is performed within Mars, the evaluation testing is performed in a manual fashion. The test procedure involves launching a Mars *node*, in which the test data set is indexed, before feeding the test query to the prototype component, all via a Mars interactive shell.

Chapter 7

Results and Discussion

This chapter starts by presenting the storage model that was provided by Fast for the SPARQL Query Service prototype, comparing it to the proposal from Section 4.3.2. Next, a brief introduction is given on how triples are being passed to Mars for indexing and which format those triples are assumed to have for the prototype to work. Following, the features of SPARQL supported by the prototype is discussed before finally presenting results from the tests conducted.

7.1 Prototype Storage Model

Section 4.3 discusses the reasonable storage model alternatives, from which a storage model based on TripleT has been proposed. The actual storage model provided by Fast for the prototype closely resembles the proposed storage model, shown in Figure 4.5 on page 46.

The most notable deviation from the proposal is the omission of dictionary encoding. Hence, all URIs are indexed and stored in their entirety. Further, if the payload lists contain two or less rows, they are stored directly in the B+ tree index.

As discussed in Section 2.3.3, Mars identifies every document by a unique document identifier. When the RDF data is indexed, every triple is considered a document and assigned a unique *DocID*. In addition to the two triple elements in the payload lists, the *DocID* is also included. As this value is never of any interest to the SPARQL query evaluation, it is simply ignored.

Payload lists are actually logical lists based on hash maps holding the status of every document. A linked list exists per document, holding the deltas for every state. Should SPARQL ever acquire constructs for specifying the generation of the RDF data to be queried, Mars has the infrastructure required to implement such a feature.

```
<doc>
  <s>{subject}</s>
  <p>{predicate}</p>
  <o>{object}</o>
</doc>
```

Figure 7.1: XML Format for RDF Triples

7.2 Triple Format

In order to let Mars index RDF triples, each triple is stored in a separate XML file, before being passed to Mars's Document Feeder component. Mars will further decompose each triple and index each triple element in a B+ tree, as discussed in Section 4.3.2. The XML must follow a specific format, shown in Figure 7.1

Also note that, in order to preserve whitespace and avoid having to replace special characters with entity references, CDATA nodes can be used for representing triple nodes in the XML, rather than regular text nodes.

The SPARQL Query Service component further assumes a certain format on the triple elements. This format is based on the Terse RDF Triple Language[7], with one notable difference: resource and data type URIs are stored as absolute URIs without enclosing angle brackets. This is primarily to make the XML representation more readable. For completeness, these assumed formats are listed below.

Resource URIs `http://example.org/ID`

Simple Literals `"Literal"`

Data-Typed Literals `"2"^^http://example.org/type`

Language Tagged Literals `"Copenhagen"@en`

Blank Nodes `_:BnodeLabel`

7.3 Supported SPARQL Features

The prototype implementation only supports a subset of the features defined in the SPARQL specification [2]. However, supporting the entire specification was never the intention. The top priority was to support *SELECT* queries, consisting only of triple patterns, evaluated against a single RDF data set.

```
Query           ::=  Prologue ( SelectQuery | ConstructQuery |
                     DescribeQuery | AskQuery )
Prologue        ::=  BaseDecl? PrefixDecl*
BaseDecl        ::=  'BASE' IRI_REF
PrefixDecl      ::=  'PREFIX' PNAME_NS IRI_REF
SelectQuery     ::=  'SELECT' ( 'DISTINCT' | 'REDUCED' )? ( Var+ | '*' )
                     DatasetClause* WhereClause SolutionModifier
ConstructQuery  ::=  'CONSTRUCT' ConstructTemplate DatasetClause*
                     WhereClause SolutionModifier
DescribeQuery   ::=  'DESCRIBE' ( VarOrIRIref+ | '*' ) DatasetClause*
                     WhereClause? SolutionModifier
AskQuery        ::=  'ASK' DatasetClause* WhereClause
```

Figure 7.2: SPARQL Grammar Extract

7.3.1 Query Forms

SPARQL defines four query forms, of which only *SELECT* queries are partly supported by the prototype. Figure 7.2 shows an extract of the EBNF SPARQL grammar defining the top level SPARQL constructs. All queries start off with an optional prologue construct, declaring base and prefixes, which is fully supported.

A *SELECT* query consists of an optional set of *solution modifiers*, an optional set of *DatasetClause*s and a mandatory *WhereClause* specifying one or more graph patterns.

Solution modifiers are almost fully supported, as discussed further in Section 7.3.2. The *dataset* clause is not supported, and the single RDF data set used for evaluating SPARQL queries is determined by the context in which the queries are executed, that is, the current Mars *node* hosting the prototype component. The *where* clause is partly supported, as discussed further in Section 7.3.3.

Of the other three query forms, the *CONSTRUCT* and *DESCRIBE* query forms also consist of *solution modifiers*, *dataset* clauses and *where* clauses; the *ASK* query form consists only of the latter two. Although the query forms themselves are not supported, the constructs shared with the *SELECT* query form are.

7.3.2 Solution Modifiers

All solution modifiers, including *order*, *projection*, *distinct*, *reduced*, *offset* and *limit*, are fully supported except for the *order* and *limit* solution modifiers, which are only partly supported.

The *order* solution modifier has a well-defined behavior, as described in Section 2.2.3. However, as resources, literals and blank nodes are all indexed and stored as string values, only lexicographical ordering can be achieved by the *Sort* operator, as discussed in Section 5.4.2.

The *limit* solution modifier behavior is achieved using the *Trim* operator, which also handles the *offset* solution modifier. The SPARQL specification defines a *limit* of `0` as no results being returned at all, while the *Trim* operator with the same limit would return all results, that is, no limit.

7.3.3 Graph Patterns

The graph patterns that may be specified in the where clause include *Filter*, *Join*, *LeftJoin*, *Union*, *Graph* and *Basic Graph Pattern*. The *Join*, *Union* and *Basic Graph Pattern* are fully supported, whereas the remaining graph patterns are not supported.

The *Filter* graph pattern is a complicated construct supporting complex expressions for testing values. Supporting this graph pattern has not been a priority, and it is consequently unsupported.

The *LeftJoin* graph pattern performs a conditional left join, typically combined with a *Filter* graph pattern. As no suitable operator is currently available in Mars, as well as this not being a priority, the *LeftJoin* graph pattern is not supported.

The *Graph* graph pattern is used to specify the currently active RDF data set to be queried. This graph pattern is not supported, and in the prototype only a single RDF data set is queried.

7.3.4 Data Types

RDF allows data types to be stored along with literals in order to attach semantics to the data. For instance, a numeric literal may be stored as an integer data type to indicate that it should be treated as a numeric value rather than a string of digits.

The prototype supports querying for literals of a specified data type. A query searching for `"12"^^xsd:Integer` will not return literals like `"12"` or `"12"^^xsd:Decimal`. This behavior is according to the specification.

It does not, however, consider any semantics affiliated with data types. While the SPARQL specification considers the two literals `"12"^^xsd:Decimal` and `"12.0"^^xsd:Decimal` to be equal, this is not the case with the prototype which relies solely on string comparisons. This also affects ordering. According to the specification, `"3"^^xsd:Integer` should appear before `"12"^^xsd:Integer` in ascending order. However, the prototype always uses lexicographical ordering, incorrectly ordering these two literals the other way around.

7.4 Test Results

Testing has been an important instrument in the development of the prototype, both as a measure of correctness and as a bug tracking mechanism. The W3C SPARQL Test Suite [36] has been a great resource for validating both the parsing and evaluation of SPARQL queries.

Testing has been employed at three different stages during the development of the prototype. First the parsing has been tested, followed by the transformation from abstract syntax trees to the intermediate representation, and finally the behavior of the evaluation of the generated Mars operator trees.

The test suite contains five sets of syntax tests, consisting of 199 SPARQL queries in total, all of which should either parse or fail to parse. The syntax tests have been used to test the parser's level of conformance with the SPARQL grammar. All syntax tests pass, indicating complete conformance with the SPARQL grammar covered by the test suite.

Besides the syntax test sets, the test suite contains 109 test data sets and 241 belonging SPARQL test queries and expected result sets. Of these, 45 data sets and 72 queries qualify for testing the SPARQL features supported by the prototype. As some SPARQL features are only partly supported, like value comparison for data-typed literals, 12 of the 72 queries fail.

Table 7.1 lists the failing tests, categorized by reason for failing. The first 5 tests fail because of limited support for data types. For instance, the indexed, data-typed literal `"5.0"^^xsd:Decimal` and the test query literal *5* interpreted as `"5"^^xsd:integer`, are not detected as being equal since the two are compared as strings instead of numbers. The next 5 tests fail because of limited support for international characters, most likely caused by an internal bug in Mars, according to feedback from our supervisor from Fast. The last 2 tests fail because of missing support for resource classification, required in order to correctly have URIs appear before literals when sorted in ascending order, as discussed in Section 5.4.2.

In addition to the tests from the W3C SPARQL Test Suite, custom tests tailored for the three stages, the intermediate representation in particular, have been employed to validate aspects not covered by the test suite.

Test	Reason for failing
Basic Term 6 Basic Term 7 Basic Term 8 sort-4 sort-7	Limited support for data types
kanji-01 kanji-02 normalization-01 normalization-02 normalization-03	Limited support for international characters
sort-6 sort-8	Missing support for resource classification

Table 7.1: Failing, Relevant Tests From the W3C SPARQL Test Suite

Chapter 8

Conclusion and Further Work

Throughout this project, research on how to efficiently store and index RDF data in the Mars search engine has been conducted. Further, the possibilities of evaluating SPARQL queries in Mars have been explored, prototyped and tested.

A storage model based on state of the art research within the Semantic Web initiative has been proposed. This model is based on dictionary encoding and decomposition of RDF triples in order to facilitate efficient retrieval of data and evaluation of queries. During the course of this project, Fast has provided a prototype storage model based on this proposal which has been used as the basis for evaluating SPARQL queries.

A prototype SPARQL Query Service component for Mars has been created, using a parser developed in a related project during fall 2008. Transformations from common SPARQL queries to SPARQL algebra and from SPARQL algebra to Mars operator trees have been described and implemented. Hence, the prototype is able to evaluate a wide range of typical SPARQL queries.

To quantify the degree of conformance with the specification, **testing of the prototype has been performed using W3C's own SPARQL test suite as well as custom tests**. All 199 syntax tests pass, indicating complete conformance with the SPARQL grammar. Further, 72 of 241 evaluation tests were considered relevant for the prototype. Of these, only 12 fail due to incomplete support for data types, international characters and resource classification.

8.1 Further Work

As discussed in Section 7.3, the prototype lacks support for parts of the SPARQL specification. Further work includes adding support for all query types as well as implementing Mars operators for handling the currently unsupported SPARQL operators of *SELECT* queries.

Correct interpretation of data-typed literals has to be implemented, particularly with regards to comparisons and ordering. The *ORDER BY* keyword from SPARQL has to be transformed into Mars operators that meet the special semantics of ordering results from SPARQL queries described in Section 2.2.3.

Optimizations regarding performance should also be considered. Implementing dictionary encoding, in particular, will decrease the size of the index, enforce constant field sizes and improve the efficiency of performing value comparisons. The introduction of dictionary encoding would, however, require changes to how the prototype operates. The index lookup values would have to be substituted by their corresponding keys in the dictionary, and the keys produced as query results would have to be substituted back to their original values before being returned to the user.

Further performance gains could be obtained by optimizing the Mars operator trees produced by the prototype. Using the general transformation algorithms described in this thesis will often produce sub-optimal operator trees. One example is the frequent occurrence of two Map operators in a row that could be merged into one single but semantically equivalent Map operator.

References

[1] W3C. RDF Primer. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, February 2004.

[2] W3C. SPARQL Query Language for RDF. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, January 2008.

[3] iAD. About the iAD Research Centre. http://www.iad-centre.no/about.html.

[4] Tormod Fjeldskår and Ole Petter Bang. Developing a SPARQL parser for .NET. Enclosed in ZIP archive, see Appendix C, December 2008.

[5] W3C. W3C Semantic Web Activity. http://www.w3.org/2001/sw/, January 2009.

[6] W3C. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/, February 2004.

[7] Turtle - Terse RDF Triple Language. http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/, January 2008.

[8] W3C. RDF Data Access Use Cases and Requirements. http://www.w3.org/TR/2005/WD-rdf-dawg-uc-20050325/, March 2005.

[9] Microsoft. Managed Babel. http://msdn.microsoft.com/en-us/library/bb165037(VS.90).aspx, November 2007.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[11] ISO/IEC 14977:1996(E). *Information technology - Syntactic metalanguage - Extended BNF*. ISO/IEC, Geneva, Switzerland.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[13] W3C. Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/2004/REC-xml11-20040204/#sec-notation, February 2004.

[14] The Lex & Yacc Page. http://dinosaur.compilertools.net/.

[15] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[16] Valerie Bönström, Annika Hinze, and Heinz Schweppe. Storing RDF as a Graph. In *LA-WEB '03: Proceedings of the First Conference on Latin American Web Congress*, page 27, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Luping Ding, Luping Ding, Kevin Wilkinson, Kevin Wilkinson, Craig Sayers, Craig Sayers, Harumi Kuno, and Harumi Kuno. Application-specific schema design for storing large RDF datasets. In *In First Intl Workshop on Practical and Scalable Semantic Systems*, 2003.

[18] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[19] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, 2008.

[20] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *LA-WEB '05: Proceedings of the Third Latin American Web Congress*, page 71, Washington, DC, USA, 2005. IEEE Computer Society.

[21] G. H. L. Fletcher and P. W. Beck. A role-free approach to indexing large RDF data sets in secondary memory for efficient SPARQL evaluation. *ArXiv e-prints*, November 2008.

[22] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, November 2002.

[23] Liu Baolin and Hu Bo. HPRD: A High Performance RDF Database. In *NPC*, pages 364–374, 2007.

[24] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.

[25] Stephen Harris and Nicholas Gibbins. 3store: Efficient Bulk RDF Storage, June 2003.

[26] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. pages 54–68. Springer, 2002.

[27] Jena - A Semantic Web Framework for Java. `http://jena.sourceforge.net/`, February 2009.

[28] Jena2 Database Interface - Database Layout. `http://jena.sourceforge.net/DB/layout.html`, February 2009.

[29] YARS: Yet Another RDF Store. `http://sw.deri.org/2004/06/yars/`, February 2009.

[30] The JDBM project. `http://jdbm.sourceforge.net/`, February 2009.

[31] Dave Beckett. Redland RDF Libraries. `http://librdf.org/`.

[32] David Beckett. The Design and Implementation of the Redland RDF Library. In *Proceedings of WWW10 conference*, 2001.

[33] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd revised edition, May 1999.

[34] W3C. RDF Semantics. `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`, February 2004.

[35] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[36] W3C. DAWG Testcases. `http://www.w3.org/2001/sw/DataAccess/tests/r2`.

Glossary

AST Abstract Syntax Tree, a tree representation of the syntax of some source code.

BNF Backus-Naur Form, a metasyntax used to express context-free grammars.

C# One of the programming languages supported by Microsoft for creating applications that target on the .NET Framework.

EBNF Extended Backus-Naur Form, an extension to BNF containing several shorthand notations.

iAD Information Access Disruptions, a constellation between Fast Search & Transfer (Fast), two Norwegian enterprises, the Norwegian University of Science and Technology (NTNU) and several other universities.

IRI Internationalized Resource Identifier, a generalization of the Uniform Resource Identifier (URI), allowing Unicode character rather than being restricted to a subset of ASCII characters.

Keyword When used in the context of a programming language, a keyword typically denotes a reserved identifier which specifies certain behavior.

LALR Parser Lookahead LR parser, produces the rightmost derivation, reading the input from left to right.

LL(k) Parser Produces the leftmost derivation, reading the input from left to right, using at most k tokens lookahead.

Mars The next generation search engine by Fast, combining database and search engine technology, targeting enterprises.

Mars Node An instance of Mars, typically part of a larger constellation of nodes, collectively constituting a search engine.

.NET Framework A software technology available from Microsoft which provides a library of pre-fabricated components and a virtual machine for managing application execution.

Parser A program that performs syntactical analysis on a sequence of tokens to determine the grammatical structure. A parser usually produces an Abstract Syntax Tree (AST) for further analysis.

Pattern A pattern can refer to a recognizable pattern in a data structure. It is also used to denote a semantically defined entity in SPARQL algebra.

RDF Resource Description Framework, a model for representing information about resources on the World Wide Web.

RDF Graph An RDF graph is a set of RDF triples, where subjects and objects constitute the nodes of the graph while predicates constitute the directed arcs between nodes.

RDF Triple An RDF triple consists of a subject and an object, as well as a predicate describing the subject's relationship with the object. A collection of RDF triples constitute an RDF graph.

Scanner A program that reads a sequence of characters and produce a sequence of tokens which represent one or more characters. A scanner performs the first step (the lexical analysis) when parsing input in a given language.

Semantic Web An extension of the World Wide Web in which the semantics of information and services on the web is defined, making it possible for the web to understand and satisfy the requests of people and machines to use the web content.

SharQL A SPARQL parser created in a related project.

SPARQL SPARQL Protocol And RDF Query Language, a query language for RDF data.

Turtle Terse RDF Triple Language, a serialization format for RDF, resulting in a less verbose output than the equivalent XML serialization.

URI Uniform Resource Identifier, a compact string of characters used to identify or name a resource on the Internet.

Visitor Pattern A way of separating an algorithm from the object structure upon which it operates.

W3C World Wide Web Consortium, the main international standards organization for the World Wide Web.

Appendix A

NodeBase Class

This appendix contains the *NodeBase* class, from which all classes used to represent the abstract syntax trees (ASTs) descend. This abstract class implements the *INode* interface and thus the *Accept* methods that realize the Visitor pattern used for traversing the ASTs. In addition, the class implements functionality for cloning nodes, used when resolving syntactic sugar.

The *NodeBase* class is discussed in Section 3.2.3.

```
1  using System;
2  using System.Collections.Generic;
3  using SharQL.Ast.Visitor;
4
5  namespace SharQL.Ast
6  {
7    /// <summary>
8    /// The base implementation of the <see cref="INode"/> interface.
9    /// </summary>
10   public abstract class NodeBase : INode
11   {
12     #region Protected Fields
13
14     /// <summary>The parent of the node.</summary>
15     protected INode parent;
16     /// <summary>The children of the node.</summary>
17     protected IList<INode> children;
18
19     #endregion
20
21     /// <summary>
22     /// Default constructor.
23     /// </summary>
24     public NodeBase()
25     {
26       children = new NodeCollection();
27     }
28
29     /// <summary>
30     /// Constructs the object and adds a set of children.
31     /// </summary>
32     /// <param name="children">The children to add</param>
33     public NodeBase(params INode[] children)
```

```
34        : this()
35      {
36        foreach (INode node in children)
37        {
38          if (node != null) node.Parent = this;
39          this.children.Add(node);
40        }
41      }
42
43      /// <summary>
44      /// Gets or sets the parent of the node. Should be set to
45      /// <c>null</c> if node is root.
46      /// </summary>
47      public virtual INode Parent
48      {
49        get { return parent; }
50        set { parent = value; }
51      }
52
53      /// <summary>
54      /// Gets the list of children of the node.
55      /// </summary>
56      public virtual IList<INode> Children
57      {
58        get { return children; }
59      }
60
61      /// <summary>
62      /// Accepts a visitor. The visitor pattern can be used for
63      /// e.g. optimizing the AST.
64      /// </summary>
65      /// <param name="visitor">The visitor</param>
66      public virtual void Accept(IVisitor visitor)
67      {
68        Accept(visitor, VisitorOrder.ChildrenBeforeParent);
69      }
70
71      /// <summary>
72      /// Accepts a visitor. The visitor pattern can be used for
73      /// e.g. optimizing the AST.
74      /// </summary>
75      /// <param name="visitor">The visitor</param>
76      /// <param name="order">The visitor order</param>
77      public virtual void Accept(IVisitor visitor, VisitorOrder order)
78      {
79        if (order == VisitorOrder.ChildrenBeforeParent)
80        {
81          for (int i = children.Count - 1; i >= 0; i--)
82          {
83            INode child = children[i];
84
85            if (child != null)
86              child.Accept(visitor, order);
87          }
88          Action<INode> action = visitor[GetType()];
89          if (action != null)
90          {
91            action(this);
92          }
93        }
94        else if (order == VisitorOrder.ParentBeforeChildren)
95        {
```

```
96          Action<INode> action = visitor[GetType()];
97          if (action != null)
98          {
99            action(this);
100         }
101         for (int i = children.Count - 1; i >= 0; i--)
102         {
103           INode child = children[i];
104
105           if (child != null)
106             child.Accept(visitor, order);
107         }
108       }
109     }
110
111     /// <summary>
112     /// Returns a string representation of the node.
113     /// </summary>
114     /// <returns>A string representation of the node</returns>
115     public override string ToString()
116     {
117       return this.GetType().FullName;
118     }
119
120     /// <summary>
121     /// Returns a prefixed string representation of the node.
122     /// </summary>
123     /// <param name="prefix">The string prefix to which the string
124     /// representation should be appended.</param>
125     /// <returns>A prefixed string representation of the node</returns>
126     public virtual string ToString(string prefix)
127     {
128       return prefix + ToString();
129     }
130
131     #region ICloneable Members
132
133     /// <summary>
134     /// Creates a new <see cref="NodeBase"/> object, with it memebers
135     /// initialized to the same values as the current object.
136     /// </summary>
137     /// <returns>a new <see cref="NodeBase"/> object, with it memebers
138     /// initialized to the same values as the current object</returns>
139     public virtual NodeBase Clone()
140     {
141       NodeBase newNode = (NodeBase)MemberwiseClone();
142       newNode.children = new NodeCollection();
143       foreach (INode node in children)
144       {
145         if (node != null)
146         {
147           newNode.children.Add(node.Clone() as INode);
148         }
149         else
150         {
151           newNode.children.Add(null);
152         }
153       }
154
155       return newNode;
156     }
157
```

```
158       object ICloneable.Clone()
159       {
160         return Clone();
161       }
162
163       #endregion
164     }
165   }
```

Appendix B

WhereClauseTransformer Class

This appendix contains the *WhereClauseTransformer* class, which is responsible for transforming *WHERE* clause constructs from the abstract syntax trees into their corresponding SPARQL algebra representations. The underlying transformation algorithm is shown in Figure 5.3.

The *WhereClauseTransformer* class is described in Section 6.5.1.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using SharQL.Algebra.QueryStructure;
5  using SharQL.Ast;
6  using SharQL.Ast.Nodes;
7
8  namespace SharQL.Algebra
9  {
10   /// <summary>
11   /// Contains functionality for transforming
12   /// <see cref="WhereClauseNode"/> objects into trees of
13   /// <see cref="Pattern"/>-derived objects.
14   /// </summary>
15   internal static class WhereClauseTransformer
16   {
17     /// <summary>
18     /// Transforms the specified where clause into SPARQL algebra.
19     /// </summary>
20     /// <param name="whereClause">The where clause to transform.</param>
21     /// <returns>A <see cref="Pattern"/> object representing the SPARQL
22     /// algebra expression.</returns>
23     public static Pattern Transform(WhereClauseNode whereClause)
24     {
25       if (whereClause.Children[0] == null)
26       {
27         throw new ArgumentException(
28           "First child of whereClause can not be null", "whereClause");
29       }
30
31       try
32       {
33         return transform(whereClause.Children[0]);
34       }
```

```
35          catch (ArgumentException e)
36          {
37            throw new FormatException(
38              "Unexpected node encountered in whereClause", e);
39          }
40        }
41
42        /// <summary>
43        /// Transforms the specified node.
44        /// </summary>
45        /// <param name="node">The node to transform.</param>
46        /// <returns>The resulting SPARQL algebra</returns>
47        private static Pattern transform(INode node)
48        {
49          if (node is TriplesBlockNode)
50          {
51            return transformTriplesBlock((TriplesBlockNode)node);
52          }
53          else if (node is GroupOrUnionGraphPatternNode)
54          {
55            return transformGroupOrUnionGraphPattern(
56              (GroupOrUnionGraphPatternNode)node);
57          }
58          else if (node is GraphGraphPatternNode)
59          {
60            return transformGraphGraphPattern((GraphGraphPatternNode)node);
61          }
62          else if (node is GroupGraphPatternNode)
63          {
64            PatternNormalizer.NormalizeGroupGraphPattern(
65              (GroupGraphPatternNode)node);
66            return transformGroupGraphPattern((GroupGraphPatternNode)node);
67          }
68          else
69          {
70            throw new ArgumentException(
71              "Unexpected node " + node.GetType().FullName, "node");
72          }
73        }
74
75        /// <summary>
76        /// Transforms the triples block.
77        /// </summary>
78        /// <param name="node">The <see cref="TriplesBlockNode"/>
79        /// node.</param>
80        /// <returns>The resulting SPARQL algebra</returns>
81        private static Pattern transformTriplesBlock(TriplesBlockNode node)
82        {
83          Triple[] triples = getListOfTriples(node);
84          Pattern prevPattern = null;
85          foreach (var triple in triples)
86          {
87            BasicGraphPattern bgp = new BasicGraphPattern();
88            bgp.Triples.Add(triple);
89            bgp.VariablesInScope.AddRange(getVariables(triple));
90
91            if (prevPattern != null)
92            {
93              Join join = new Join(prevPattern, bgp);
94              join.VariablesInScope.AddRange(
95                prevPattern.VariablesInScope.Union(bgp.VariablesInScope));
96              prevPattern = join;
```

```
97            }
98            else
99            {
100             prevPattern = bgp;
101           }
102         }
103
104         return prevPattern;
105       }
106
107       /// <summary>
108       /// Transforms the group or union graph pattern.
109       /// </summary>
110       /// <param name="node">The <see cref="GroupOrUnionGraphPatternNode"/>
111       /// node.</param>
112       /// <returns>The resulting SPARQL algebra</returns>
113       private static Pattern transformGroupOrUnionGraphPattern(
114         GroupOrUnionGraphPatternNode node)
115       {
116         Pattern pattern = null;
117         foreach (INode child in node.Children.Where(c => c != null))
118         {
119           if (pattern == null)
120           {
121             pattern = transform(child);
122           }
123           else
124           {
125             pattern = new Union(pattern, transform(child));
126
127             pattern.VariablesInScope.AddRange(
128               ((Union)pattern).FirstPattern.VariablesInScope.Union(
129               ((Union)pattern).SecondPattern.VariablesInScope));
130           }
131         }
132         return pattern;
133       }
134
135       /// <summary>
136       /// Transforms the graph graph pattern.
137       /// </summary>
138       /// <param name="node">The <see cref="GraphGraphPatternNode"/>
139       /// node.</param>
140       /// <returns>The resulting SPARQL algebra</returns>
141       private static Pattern transformGraphGraphPattern(
142         GraphGraphPatternNode node)
143       {
144         Graph graph = null;
145
146         if (node.Children[0] is IriRefNode)
147         {
148           graph = new Graph((IriRefNode)node.Children[0],
149             transform(node.Children[1]));
150         }
151         else if (node.Children[0] is VarNode)
152         {
153           graph = new Graph((VarNode)node.Children[0],
154             transform(node.Children[1]));
155         }
156         else
157         {
158           throw new ArgumentException(
```

```
159             "Expected IriRefNode or VarNode as first child of
                    GraphGraphPatternNode.", "node");
160         }
161
162         graph.VariablesInScope.AddRange(graph.Pattern.VariablesInScope);
163
164         return graph;
165       }
166
167       /// <summary>
168       /// Transforms the group graph pattern.
169       /// </summary>
170       /// <param name="node">The <see cref="GroupGraphPatternNode"/>
171       /// node.</param>
172       /// <returns>The resulting SPARQL algebra.</returns>
173       private static Pattern transformGroupGraphPattern(
174         GroupGraphPatternNode node)
175       {
176         List<INode> filterConstraintExpressions = new List<INode>();
177         Pattern pattern = new EmptyPattern();
178         foreach (INode child in node.Children)
179         {
180           if (child == null)
181           {
182             continue;
183           }
184           else if (child is FilterNode)
185           {
186             filterConstraintExpressions.Add(child.Children[0]);
187           }
188           else if (child is OptionalGraphPatternNode)
189           {
190             Pattern optionalPattern = transform(child.Children[0]);
191             if (optionalPattern is Filter)
192             {
193               pattern = new LeftJoin(pattern,
194                 ((Filter)optionalPattern).Pattern,
195                 ((Filter)optionalPattern).ConstraintExpression);
196             }
197             else
198             {
199               pattern = new LeftJoin(pattern,
200                 optionalPattern,
201                 new BooleanLiteralNode() { Value = true });
202             }
203             pattern.VariablesInScope.AddRange(
204               ((LeftJoin)pattern).LeftPattern.VariablesInScope.Union(
205               ((LeftJoin)pattern).RightPattern.VariablesInScope));
206           }
207           else
208           {
209             Pattern graphPattern = transform(child);
210             if (pattern is EmptyPattern)
211             {
212               pattern = graphPattern;
213             }
214             else
215             {
216               pattern = new Join(pattern, graphPattern);
217               pattern.VariablesInScope.AddRange(
218                 ((Join)pattern).FirstPattern.VariablesInScope.Union(
219                 ((Join)pattern).SecondPattern.VariablesInScope));
```

```
            }
          }
        }
        if (filterConstraintExpressions.Count > 0)
        {
          INode conjunctionOfConstraints = conjugateConstraints(
            filterConstraintExpressions);
          pattern = new Filter(conjunctionOfConstraints, pattern);
          pattern.VariablesInScope.AddRange(
            ((Filter)pattern).Pattern.VariablesInScope);
        }
        return pattern;
      }

      /// <summary>
      /// Conjugates the specified list of constraints.
      /// </summary>
      /// <param name="filterConstraintExpressions">The filter
      /// constraint expressions.</param>
      /// <returns>The conjugation of the specified constraints.</returns>
      private static INode conjugateConstraints(
        List<INode> filterConstraintExpressions)
      {
        INode conjunction = filterConstraintExpressions[0];

        for (int i = 1; i < filterConstraintExpressions.Count; i++)
        {
          conjunction = new ConditionalAndExpressionNode(
            conjunction as NodeBase,
            filterConstraintExpressions[i] as NodeBase);
        }

        return conjunction;
      }

      /// <summary>
      /// Gets the list of triples from the specified
      /// <see cref="TriplesBlockNode"/> object.
      /// </summary>
      /// <param name="triplesBlockNodeRoot">The triples block root
      /// node.</param>
      /// <returns>An array of <see cref="Triple"/> objects.</returns>
      private static Triple[] getListOfTriples(
        TriplesBlockNode triplesBlockNodeRoot)
      {
        List<Triple> triples = new List<Triple>();

        TriplesBlockNode current = triplesBlockNodeRoot;

        while (current != null)
        {
          triples.Add(translateTripleNode(current));
          current = current.Children[1] as TriplesBlockNode;
        }

        return triples.ToArray();
      }

      /// <summary>
      /// Translates a <see cref="TriplesBlockNode"/> object into a
      /// <see cref="Triple"/> object.
      /// </summary>
```

```
282       /// <param name="triplesBlock">The <see cref="TriplesBlockNode"/>
283       /// object to translate.</param>
284       /// <returns>The resulting <see cref="Triple"/> object</returns>
285       private static Triple translateTripleNode(TriplesBlockNode
              triplesBlock)
286       {
287         INode subject = null, predicate = null, @object = null;
288
289         TriplesSameSubjectNode tssn = triplesBlock.Children[0] as
                TriplesSameSubjectNode;
290         if (tssn != null)
291         {
292           subject = tssn.Children[0];
293
294           PropertyListNode pln = tssn.Children[2] as PropertyListNode;
295           if (pln != null)
296           {
297             predicate = pln.Children[0];
298
299             ObjectListNode oln = pln.Children[1] as ObjectListNode;
300             if (oln != null)
301             {
302               @object = oln.Children[0];
303             }
304           }
305         }
306
307         Triple triple = new Triple();
308
309         triple.setSubject(subject);
310         triple.setPredicate(predicate);
311         triple.setObject(@object);
312
313         return triple;
314       }
315
316       /// <summary>
317       /// Sets the triple's subject.
318       /// </summary>
319       /// <param name="triple">The target triple.</param>
320       /// <param name="subject">The subject to set.</param>
321       private static void setSubject(this Triple triple, INode subject)
322       {
323         if (subject is VarNode)
324         {
325           triple.Subject = new Resource(ResourceType.Variable,
                  ((VarNode)subject).Value);
326         }
327         else if (subject is IriRefNode)
328         {
329           triple.Subject = new Resource(ResourceType.Iri,
                  ((IriRefNode)subject).Value);
330         }
331         else if (subject is BlankNode)
332         {
333           triple.Subject = new Resource(ResourceType.Blank,
                  ((BlankNode)subject).Value);
334         }
335         else if (subject is RdfLiteralNode)
336         {
337           RdfLiteralNode node = (RdfLiteralNode)subject;
338
```

```
339           triple.Subject = new Resource(ResourceType.Literal, '"' +
                  ((StringLiteralNode)node.String).Value + '"');
340           if (node.IriRef != null)
341           {
342             triple.Subject = new Resource(triple.Subject.Type,
                    triple.Subject.Value + "^^" +
                    ((IriRefNode)node.IriRef).Value);
343           }
344           else if (!string.IsNullOrEmpty(node.LangTag))
345           {
346             triple.Subject = new Resource(triple.Subject.Type,
                    triple.Subject.Value + "@" + node.LangTag);
347           }
348         }
349       }
350
351       /// <summary>
352       /// Sets the triple's predicate.
353       /// </summary>
354       /// <param name="triple">The target triple.</param>
355       /// <param name="predicate">The predicate to set.</param>
356       private static void setPredicate(this Triple triple, INode predicate)
357       {
358         if (predicate is VarNode)
359         {
360           triple.Predicate = new Resource(ResourceType.Variable,
                  ((VarNode)predicate).Value);
361         }
362         else if (predicate is IriRefNode)
363         {
364           triple.Predicate = new Resource(ResourceType.Iri,
                  ((IriRefNode)predicate).Value);
365         }
366       }
367
368       /// <summary>
369       /// Sets the triple's object.
370       /// </summary>
371       /// <param name="triple">The target triple.</param>
372       /// <param name="object">The object to set.</param>
373       private static void setObject(this Triple triple, INode @object)
374       {
375         if (@object is VarNode)
376         {
377           triple.Object = new Resource(ResourceType.Variable,
                  ((VarNode)@object).Value);
378         }
379         else if (@object is IriRefNode)
380         {
381           triple.Object = new Resource(ResourceType.Iri,
                  ((IriRefNode)@object).Value);
382         }
383         else if (@object is BlankNode)
384         {
385           triple.Object = new Resource(ResourceType.Blank,
                  ((BlankNode)@object).Value);
386         }
387         else if (@object is RdfLiteralNode)
388         {
389           RdfLiteralNode node = (RdfLiteralNode)@object;
390
391           triple.Object = new Resource(ResourceType.Literal, '"' +
```

```
                ((StringLiteralNode)((RdfLiteralNode)@object).String).Value +
                '"');
392           if (node.IriRef != null)
393           {
394             triple.Object = new Resource(triple.Object.Type,
                    triple.Object.Value + "^^" +
                    ((IriRefNode)node.IriRef).Value);
395           }
396           else if (!string.IsNullOrEmpty(node.LangTag))
397           {
398             triple.Object = new Resource(triple.Object.Type,
                    triple.Object.Value + "@" + node.LangTag);
399           }
400         }
401       }
402
403       /// <summary>
404       /// Gets the variables present in the specified
405       /// <see cref="Triple"/>.
406       /// </summary>
407       /// <param name="triple">The triple to investigate</param>
408       /// <returns>The variables present in the specified
409       /// <see cref="Triple"/>.</returns>
410       private static IEnumerable<string> getVariables(Triple triple)
411       {
412         if (triple.Subject.Type == ResourceType.Variable)
413         {
414           yield return triple.Subject.Value;
415         }
416         else if (triple.Subject.Type == ResourceType.Blank)
417         {
418           yield return QueryTransformer.BlankNodeVariablePrefix +
                triple.Subject.Value.Substring(2);
419         }
420
421         if (triple.Predicate.Type == ResourceType.Variable)
422         {
423           yield return triple.Predicate.Value;
424         }
425         else if (triple.Predicate.Type == ResourceType.Blank)
426         {
427           yield return QueryTransformer.BlankNodeVariablePrefix +
                triple.Predicate.Value.Substring(2);
428         }
429
430         if (triple.Object.Type == ResourceType.Variable)
431         {
432           yield return triple.Object.Value;
433         }
434         else if (triple.Object.Type == ResourceType.Blank)
435         {
436           yield return QueryTransformer.BlankNodeVariablePrefix +
                triple.Object.Value.Substring(2);
437         }
438       }
439     }
440   }
```

Appendix C

Enclosed ZIP Archive

This appendix describes the contents of the ZIP archive enclosed with this thesis. This archive is available through the DAIM system at `http://daim.idi.ntnu.no/`.

The folder structure of the archive is as follows.

- `Documents`
  - `Developing a SPARQL parser for .NET`
    A digital copy of [4]
  - `Storing and Querying RDF in Mars`
    A digital copy of this document
- `Source`
  - `Component`
    The source code for the prototype presented in this thesis
  - `Parser`
    The source code for the SPARQL parser used in the prototype

Note that Fast's libraries have been removed from the prototype solution. Thus, the prototype source code will not build due to broken dependencies.