FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Marzia Cutajar

RDF Data Querying in Multi-Model NoSQL Databases

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Martin Svoboda, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2020

Niddedika dan it-teżi lill-għażiża nanna tiegħi Lela, li ħalliet din id-dinja fit-8 ta' Diċembru 2017.

I would like to thank everyone who helped me in any way while writing this master's thesis.

Firstly, I thank my supervisor Martin Svoboda for his guidance and contribution to the writing process.

Secondly, I would like to thank my family and friends for supporting me during my studies. I especially thank my dear mother Gertrude for always inspiring me and believing in me. I would not be here without her.

Finally, I thank my boyfriend Dario for his patience and words of encouragement during the stressful times.

Title: RDF Data Querying in Multi-Model NoSQL Databases

Author: Marzia Cutajar

Department: Department of Software Engineering

Supervisor: RNDr. Martin Svoboda, Ph.D., Department of Software Engineering

Abstract: The RDF framework has gained popularity in recent years, as it makes semantic data easily accessible and queryable on the Web. However, RDF data management systems are facing certain challenges, particularly that of scalability. NoSQL databases, which are known for their ability to store large amounts of unstructured data, could be exploited for this purpose to gain more efficient systems. Even more interesting would be the use of Multi-Model NoSQL databases, which incorporate several different data models within a single database management system, and were created as a response to Polyglot Persistence and its challenges. As of yet, semantic data is not well supported in such database systems. In this thesis, we present approaches for transforming and storing RDF data within the Multi-Model database ArangoDB, as well as propose a query transformation algorithm that enables us to query RDF data in ArangoDB using SPARQL. The functionality of these data transformation and query transformation algorithms is experimentally evaluated on sample data and queries, using our prototype implementation.

Keywords: RDF, Multi-Model databases, NoSQL, ArangoDB, SPARQL

Contents

Introduction

Context

The adoption of the *Linked Data* paradigm and the *RDF* [33] format has grown significantly over the past decade. The Linked Data paradigm promotes the publishing of semantically enriched data on the Web through the use of self-describing data and interlinking based on associating globally unique identifiers of data.

Even though RDF data is gaining wider acceptance, there are still challenges that come with the practical use of RDF. RDF data management systems are facing two challenges, namely system scalability and dealing with the generality of the data, the former being a particularly pressing issue [72]. Working with RDF graphs, which are typically highly connected and distributed, results in matching and querying large volumes of data, thus highlighting the scalability problem.

Larger amounts of data have pushed the development of solutions for handling *Big Data*, such as Hadoop [6]. To further improve the handling of data that exists in a variety of formats, *NoSQL* database systems have emerged, offering flexibility due to not enforcing the type and structure of stored data. This is in contrast to traditional relational database systems.

Moreover, not only because of the variety of Big Data, choosing a suitable logical data model as well as a particular database system for a given project became difficult. This led to the idea of so-called *Polyglot Persistence* [68], where multiple databases can be used within a single application. This concept is further addressed by *Multi-Model* NoSQL databases, incorporating several different data models even within a single database system.

Motivation

With ever-growing amounts of data being stored, the need to store it as descriptively as possible has become even more important. By using semantic data formats like RDF to describe and store data, the data in itself contains the value definition and data type. The RDF description makes it easier to use data for analytical purposes as the data does not need to be transformed or standardized before use.

The increasing amount of data and the variety of data formats have been the reason for the increase in popularity of NoSQL database management systems and their adoption, since they are able to store large amounts of unstructured data. The availability of many NoSQL database management systems as open-source solutions is another factor that contributed to their popularity.

In particular, semantic data is one of the data types that are not yet well supported in NoSQL and Multi-Model database systems. While contemporary RDF stores are often based on the underlying relational model and are facing not just scalability challenges, the flexibility of NoSQL databases could be exploited for this purpose to gain more efficient systems. Thus, Multi-Model and NoSQL databases have the potential of handling massive amounts of RDF data.

Research Questions

The scope of this thesis is to investigate the possibility of using a recently emerged NoSQL Multi-Model database, *ArangoDB* [10], to store and represent RDF data, as well as the possibility of querying RDF data stored in ArangoDB using the standard RDF querying language called *SPARQL*. This requires analyzing existing approaches for the storage and retrieval of RDF data within Multi-Model or core NoSQL database systems.

The purpose of this thesis is to develop novel approaches to store and query RDF data within a Multi-Model NoSQL database management system, thus making use of a storage solution that has the potential of better handling the scalability challenges traditional RDF stores are facing.

The main goal is to develop an algorithm that translates a SPARQL query expression into a query expression in the native query language of ArangoDB, which is *AQL*. Its prototype implementation will then be experimentally evaluated in order to demonstrate its properties and usability.

Based on the above-described purpose of this thesis, the following research questions are explored:

- How can the RDF data model be represented within the multi-model, NoSQL database management system ArangoDB?
- How can RDF data that is stored within ArangoDB be queried using SPARQL?
- How well do the suggested solutions for storing and querying RDF data in ArangoDB perform compared to existing RDF stores?

Thesis Outline

The remaining parts of the thesis are structured into the following chapters.

Chapter 1 contains general background information required for the thesis. It describes RDF data as well as NoSQL and Multi-Model databases. This chapter also introduces the multi-model database ArangoDB, on which the research is based.

Chapter 2 contains preliminary formal definitions for components of RDF, JSON, and ArangoDB, required as a basis for other definitions in consecutive chapters.

Chapter 3 provides overviews of research papers and studies related to the topic of this thesis.

Chapter 4 gives a formal definition of the SPARQL algebra, also describing the constructs used in a SPARQL query. We also define the structure of a SPARQL algebra tree and its nodes.

Chapter 5 introduces the query language of ArangoDB, AQL, giving details of its constructs and syntax. We also formally define a tree structure for representing an AQL query expression, called an AQL query tree.

Chapter 6 defines and describes our two approaches for transforming and storing RDF data in ArangoDB, and provides reasons for the chosen approaches.

Chapter 7 describes our approach for transforming a SPARQL query into an AQL query. It gives a detailed explanation of each transformation step, together with sample transformations for better understanding.

Chapter 8 describes the prototype implementation of our proposed transformation algorithms.

Chapter 9 presents the evaluation results for our prototype implementation, experimentally evaluated on sample data and queries.

Finally, we conclude the thesis with a discussion of the main findings, limitations, and contributions of the study. Moreover, recommendations for future work are presented and discussed.

1. Background

This chapter provides some background information related to the topic of this thesis and the technologies used. Specifically, we introduce and describe important semantic technologies such as RDF and SPARQL. We also describe the components, features and use cases of NoSQL and Multi-model databases, focusing on our database of interest ArangoDB.

1.1 Semantic Web

The *Semantic Web* [46] is an extension of the World Wide Web (WWW) [48] as envisioned and standardised by the *World Wide Web Consortium (W3C)* [45], and is essentially considered to be a Web of Linked Data.

The term "Semantic Web" was coined by Tim Berners-Lee [49], the inventor of the WWW and director of the W3C. The idea behind it is to insert machine-readable metadata about web pages and how they are related to each other. This data can take the form of meta tags that are simply appended to the HTML of a website. However, HTML has its limitations, which led to the creation of new solutions and technologies specifically made with the Semantic Web in mind, for example the *Resource Description Framework (RDF)* [33]. Such technologies can be combined in order to provide data that supplements or replaces the content of Web documents.

Semantic Web standards promote common data formats and exchange protocols on the Web, thus providing a common framework that allows data to be shared and reused across applications, systems, and enterprises.

Semantic Web technologies enable people to create data stores on the Web and define vocabularies and rules for describing and handling different data and the relationships between them. These technologies allow a better and more automatic interchange of data.

1.1.1 Linked Data

Linked Data is structured data that is interlinked with other related data over the Web for useful semantic querying.

Linked data that is freely available for everyone to access and reuse is called *Linked Open Data (LOD)*, and enriches the *Linked Open Data Cloud (LOD Cloud)* [44]. A small part of the LOD Cloud is pictured in Figure 1.1. There are ever-increasing sources of LOD on the Web, together with data services that may be restricted to the suppliers and consumers of those services.

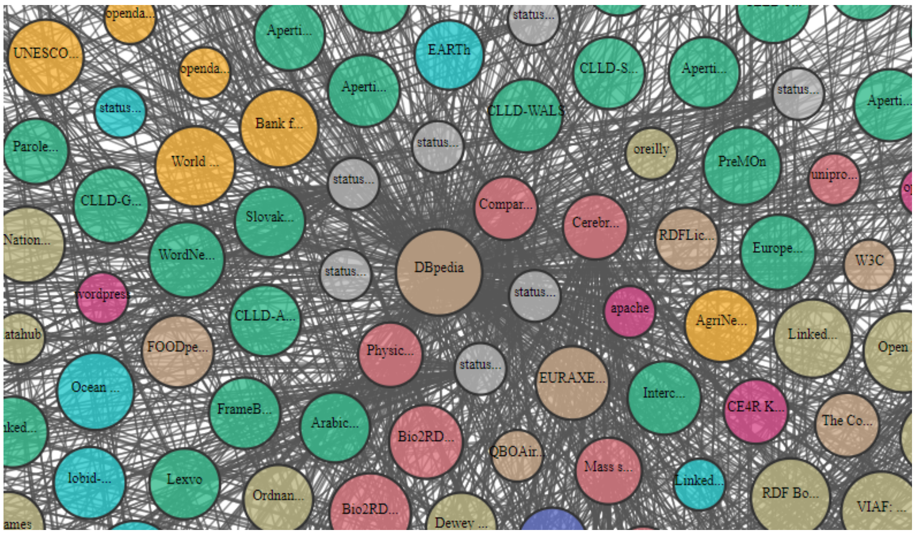


Figure 1.1: Part of the LOD Cloud [44]

Linked Data is about the use of URIs as names for things, the ability to dereference these URIs to get further information about them, and to include links to other URIs when publishing data on the Web. This creates an interconnected network of machine-processable, discoverable data.

1.1.2 RDF

RDF is a standard format for data interchange on the Web. It is particularly used for specifying relationships between resources on the Web, thus providing the foundation for linking and publishing data.

The RDF data model is based on the idea of creating statements about Internet resources in the *subject-predicate-object* form. In RDF terminology, these sentences are called *RDF triples*. The *subject* defines the described resource. The *predicate* describes a characteristic of the resource and expresses the relation between the subject and the object. The *object* stores the value of this relation. A graphical representation of an RDF triple is shown in Figure 1.2.

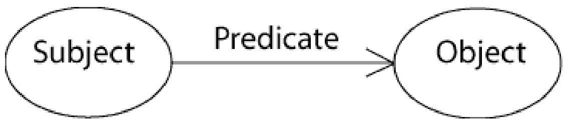


Figure 1.2: Graphical representation of an RDF triple

The basic components of the RDF model relate to the resources. The elements are composed of three disjoint subsets: Internationalized Resource Identifiers (IRIs), *blank nodes*, and *literals* [33].

An IRI is a global identifier that may be used to uniquely identify a resource.

Blank nodes are used for marking resources for which the IRI is not given. They represent resources which we are not able to, do not want to or simply cannot be referenced using a globally unique IRI.

Literals are used for representing values such as strings, numbers, and dates. A literal l is made up of two or three components:

- $value(l)$ = a lexical value in the form of a Unicode string
- $datatype(l)$ = a data type identified by an IRI, which determines how the lexical value can be mapped to the actual value
- $lang(l)$ = a language tag, if and only if $datatype(l)$ = `http://www.w3.org/1999/02/22-rdf-syntax-ns\#langString`, otherwise it is undefined.

The subject of an RDF triple can be either an IRI or a blank node, the predicate must be an IRI, and the object can be an IRI, blank node, or literal.

An *RDF graph* is made up of a set of RDF triples. In an RDF graph, the subject and object of a triple are represented as vertices, while the predicate of the triple is represented as a directed edge connecting the subject vertex to the object vertex.

An *RDF Dataset* is made up of one *Default Graph* and zero or more *Named Graphs*. A Default Graph is an RDF graph that does not have a name and can be empty, while a Named Graph is an RDF graph that is identified by a name, where the name is an IRI.

There are multiple different reasons why one might want to group RDF triples into a named graph instead of storing them in the default graph. For example, it can help to track the provenance of RDF data, or it can simply be a convenient way of sorting the triples.

Listing 1.1 shows a serialization of an RDF Dataset made up of the default graph containing two triples, and two named graphs identified by the IRIs `http://example.org/persons` and `http://example.org/places`. The syntax of this RDF data serialization is explained in 1.1.3.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <http://schema.org/> .
@prefix ex: <http://example.org/> .

# default graph
ex:persons rdfs:label "Graph of people" .
ex:places  rdfs:label "Graph of geographical places"

# named graph http://example.org/persons
ex:persons
{
  ex:Maria_Cassar rdf:type foaf:Person ;
        foaf:firstName "Maria" ;
        foaf:lastName "Cassar" ;
        schema:birthPlace ex:Malta ;
        schema:address _:maria_cassar_address ;
        foaf:knows ex:Jaroslav_Svoboda .

  _:maria_cassar_address rdf:type schema:postalAddress .

  ex:Jaroslav_Svoboda rdf:type foaf:Person ;
        foaf:firstName "Jaroslav" ;
        foaf:lastName "Svoboda" ;
        schema:birthPlace ex:Czech_Republic .
}

# named graph http://example.org/places
ex:places
{
  ex:Prague rdf:type schema:City ;
        rdfs:label "Prague"@en ;
        rdfs:label "Praha"@cs ;
        schema:containedInPlace ex:Czech_Republic .

  ex:Malta rdf:type schema:Country ;
        rdfs:label "Malta"@en .

  ex:Czech_Republic rdf:type schema:Country ;
        rdfs:label "Czech Republic"@en ;
        rdfs:label "Česká Republika"@cs ;
        ex:capitalCity ex:Prague .
}
```

Listing 1.1: Sample RDF Dataset serialized in TriG

1.1.3 RDF Formats

RDF data can be represented using different serialization formats. The most common format is *RDF/XML* [38] which represents data in XML syntax. Two other well-known formats are *N-Triples* [35] and *Turtle* [37], which are more human-readable formats than RDF/XML. These formats are semantically equivalent and can be converted into one another using RDF translation tools.

Turtle allows writing RDF in a compact and natural text form, with abbreviations for common usage patterns. In Turtle, IRIs have to be enclosed between angle brackets. In addition, they can also be written relatively, since Turtle allows defining bases and prefixes in order to refer to IRIs using a more human-readable shortened form. Blank nodes are expressed as _: followed by a blank node label, and repeated usage of the same blank node label identifies the same blank node. However, blank nodes can also appear in an unlabelled form by using square brackets in the subject or object position of a triple.

The lexical value of a literal is written between double quotation marks. To specify the data type of a literal, two circumflexes followed by an IRI are appended to the plain literal value, making it a typed literal. Moreover, a language tag can also be specified using the @ symbol.

Finally, Turtle provides ways to compact RDF triple statements. This is useful because the same subject is often referenced by several different predicates, and the same subject can also be linked to several different objects by the same predicate. Turtle allows making lists of predicate-object pairs for the same subject, by using a colon to repeat the subject of triples. Similarly, a comma can be used to repeat subject-predicate pairs for triples differing only in their objects. An example of RDF data serialized using Turtle is given in Listing 1.2.

Another useful syntax is *TriG* [36], an extension of the Turtle format allowing the serialization of named graphs and RDF datasets. An example of an RDF dataset serialized using TriG can be seen in Listing 1.1. This dataset contains the same triples serialized using Turtle in Listing 1.2, but split into different RDF graphs.

In TriG, the triple statements making up an RDF graph are enclosed within curly brackets, and a label identifying the graph is placed before the opening curly bracket. This is called a graph statement, where the label is the IRI of the named graph. The label of a graph statement may be omitted, in which case the graph is considered to be the default graph of the RDF Dataset. In a TriG document, the same graph IRI or blank node may be used as the label for more than one graph statement. In this case, the union of all the triples within these graph statements gives the whole set of triples within the graph identified by the given label.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <http://schema.org/> .
@prefix ex: <http://example.org/> .

ex:Maria_Cassar rdf:type foaf:Person ;
        foaf:firstName "Maria" ;
        foaf:lastName "Cassar" ;
        schema:birthPlace ex:Malta ;
        schema:address _:maria_cassar_address ;
        foaf:knows ex:Jaroslav_Svoboda .

ex:Jaroslav_Svoboda rdf:type foaf:Person ;
        foaf:firstName "Jaroslav" ;
        foaf:lastName "Svoboda" ;
        schema:birthPlace ex:Czech_Republic .

_:maria_cassar_address rdf:type schema:postalAddress .
```

```
ex:Prague rdf:type schema:City ;
        rdfs:label "Prague"@en ;
        rdfs:label "Praha"@cs ;
        schema:containedInPlace ex:Czech_Republic .

ex:Malta rdf:type schema:Country ;
        rdfs:label "Malta"@en .

ex:Czech_Republic rdf:type schema:Country ;
        rdfs:label "Czech Republic"@en ;
        rdfs:label "Česká Republika"@cs ;
        ex:capitalCity ex:Prague .
```

Listing 1.2: Sample RDF data serialized in Turtle

1.1.4 Ontologies

On the Semantic Web, *ontologies* [40], also referred to as *vocabularies*, define the terms used to describe and represent some domain. Vocabularies are used to classify the terms that can be used in a particular application, characterize possible relationships, and define possible constraints on using those terms.

Some of the most well-known vocabulary definition technologies are *RDFS* [39], *OWL* [30], and *SKOS* [41]. Using such technologies, it is possible to organize and enrich data with additional metadata, which allows people as well as machines to do more with it.

RDF Schema (RDFS) is a general-purpose language for representing simple RDF vocabularies on the Web. It is used for describing and adding classes, sub-classes, and properties to RDF resources.

Similar to RDFS, the Web Ontology Language (OWL) allows representing knowledge about things, groups of things, and relations between things. However, OWL builds on RDFS, providing a far larger and richer vocabulary so that more can be said about the data.

Simple Knowledge Organization System (SKOS) provides a standard way to represent knowledge organization systems, such as thesauri, glossaries and classification schemes, in the form of RDF data. SKOS is based on OWL and can thus be considered an OWL ontology.

1.1.5 SPARQL

The *SPARQL Protocol and RDF Query Language (SPARQL)* [42] is essentially a pattern matching query language for RDF graphs, and is the query language of the Semantic Web.

The language lets users query RDF data in data sources, whether the data is stored natively as RDF or viewed as RDF through the use of middleware. Generally, the user does not need to be aware if middleware is being used. SPARQL queries can be run on an RDF data source through a SPARQL endpoint, which acts as a sort of interface for query execution over the data.

The SPARQL language is mainly built around triple patterns, used to select triples from the RDF dataset being queried. Of course, the language is made up of other constructs as well, which are somewhat similar to constructs in the *Structured Query Language (SQL)* used in relational databases.

Although the SPARQL language offers the possibility to update data [43], only the data retrieval aspect is considered in this thesis.

SPARQL provides four query forms for querying RDF data, these being `SELECT`, `CONSTRUCT`, `DESCRIBE`, and `ASK`. All these query forms work with the same principle of sub-query pattern matching. They differ only in the type of result they return and the way the result is presented, that is the data format.

A `SELECT` query returns tabular data, a `CONSTRUCT` or `DESCRIBE` query returns an RDF graph, and an `ASK` query returns a boolean value. Moreover, many SPARQL endpoints allow selecting the data format used to serialize the result, such as XML, CSV, and so on.

The SPARQL components and algebra are explained and defined in Chapter 4.

1.2 Triplestores

A *triplestore* or *RDF store* is a database built purposely for the storage and retrieval of RDF triples through SPARQL queries.

Triplestores can be broadly classified into three categories. These are native triplestores, RDBMS-backed triplestores, and NoSQL-backed triplestores.

Native triplestores are those that are implemented from scratch for storing RDF data and exploit the RDF data model to efficiently store and access the data. Some examples of such triplestores are Apache Jena TDB [9], AllegroGraph [3], 4Store [1] and RDF4J [18].

RDBMS-backed triplestores are built by adding an RDF-specific layer to an existing Relational Database Management System (RDBMS). IBM DB2 [19] and Virtuoso [28] are examples of this type of triplestore.

NoSQL-backed triplestores are RDF stores built on existing NoSQL databases. For example, CumulusRDF [16] is built on top of Cassandra [5]. One can also find a large amount of research about using other NoSQL databases as RDF stores. Some of this research is presented and described in Chapter 3.

1.2.1 OpenLink Virtuoso Universal Server

OpenLink Virtuoso [28] is one of the most popular RDF stores, available in both open-source as well as commercial editions. It provides SQL, XML, as well as RDF data management. Triplestore access is available via SPARQL, ODBC [23], JDBC [21] and ADO.NET [2], among others.

DBPedia, one of the biggest and most well-known datasets forming part of the LOD Cloud, uses Virtuoso to store and provide user access to the data.

1.3 NoSQL Databases

NoSQL databases have been designed to address the problems of voluminous, multi-source and multi-format data processing in Big Data environments.

Although SQL systems are still used extensively and are ideal for certain use cases, such as maintaining transactional or legacy data, they are limited in their ability to store unstructured or semi-structured data. This is because data stored in relational databases must fit a predefined schema. This rigid structure also causes issues in terms of scalability. Due to the need to maintain the integrity of the data, SQL databases were designed to run on a single server, and are thus vertically scalable. On the other hand, NoSQL databases scale horizontally, which is a big advantage that NoSQL databases have over the relational databases, although it can cause complications implied by various aspects of the data distribution.

We shall now describe the data models used in NoSQL databases, as we refer to them later when we introduce multi-model NoSQL databases. We also briefly describe *JSON* [20] as it is a data format that is used in many NoSQL databases, including our database of interest ArangoDB.

1.3.1 NoSQL data models

NoSQL databases can be classified into four different types. These are the *key-value*, *document*, *graph*, and *column-oriented* models.

Key-value model

In this model, data is represented as simple key-value pairs. The key is a string whereas the value can be a string, number, an object, etc. Such databases only provide some simple operations such as *get*, *put* and *delete*.

Column-oriented model

In column-oriented NoSQL databases, data is stored in cells grouped in columns rather than as rows. A column consists of a column name and column value, thus a record is a collection of columns identified by a unique key. Columns are logically grouped into column families, such that each column family represents a group of similar data that is usually accessed together, somewhat like a table of data.

Document model

This model could be viewed as an extension of the key-value model, such that the value is a JSON or XML document. Each key is associated with a document whose structure remains free. The advantage of this model is the ability to retrieve a set of hierarchically structured information via a single key whereas the same operation on a relational database could involve several joins.

Graph model

This model of data representation is based on graph theory, where data is represented as nodes and edges between them representing some connection, with properties on those nodes and edges. A graph-oriented database can be considered an object-oriented database, facilitating the representation of real-world things.

1.3.2 JSON

JavaScript Object Notation (JSON) is a lightweight data-interchange format that is easy and quick for machines to process and generate. It is a text format that is language-independent but uses conventions that are familiar to programmers of the C-family of languages.

It originated as the data representation format in JavaScript, the programming language of the Web. Due to its simple yet expressive structure, JSON has quickly expanded beyond the Web into applications and services. Nowadays, JSON is displacing the more complex XML format as the serialization format for exchanging semi-structured data between applications.

Many NoSQL document-based database vendors have chosen JSON as their primary data representation format due to its advantages and widespread adoption.

JSON is mainly built on two structures:

- A collection of attribute-value pairs, called a *JSON object*.
- An ordered list of values, called a *JSON array*.

The attribute in an attribute-value pair of a JSON object must be a string. The value can be an atomic value such as a string, number, boolean value, or `null` value, or an embedded object or array. The value can also be a nested structure of such values.

1.4 Multi-Model Databases

A multi-model database is able to store and process structurally different data, that is data with distinct models, in a single data store. In the age of Big Data, having such databases is important due to the variety of the data.

Multi-model databases aim to address the limitations and issues of polyglot persistence, which supports multiple data models by using multiple data stores. With polyglot persistence, one may end up with multiple databases (both SQL and NoSQL), each with its own storage and operational requirements. This requires managing fault tolerance, scalability, and performance requirements for multiple data stores individually, which can become very complex. It can also make deployment more complicated and cause data consistency and duplication issues. With a multi-model database, these requirements are simplified due to the usage of a single data store. In addition, with multi-model databases, data integration can be easier when compared to polyglot persistence.

Two of the most popular open-source multi-model databases are *ArangoDB* [10] and *OrientDB* [29].

1.4.1 OrientDB

OrientDB is a multi-model NoSQL database that supports the key-value, document, and graph data models. It also supports the object model, that is data can be modeled in the form of objects, possibly using inheritance or polymorphism, as done in object-oriented programming.

In the document model, data is stored in documents made up of key-value pairs, and documents are grouped into classes or clusters. A graph is represented as a structure of vertices interconnected by edges, where each vertex or edge has some mandatory properties as well as user-defined properties. Interestingly, OrientDB allows using a document or graph element as the value of a key-value pair.

OrientDB uses SQL as its query language, however, it extends it to support graph querying. Another difference is that it does not use join operations, as OrientDB does not model relationships in the same way as relational databases. Instead, it uses what are called *LINKs*, where a LINK is a relationship managed by storing the target record identifier in the source record.

1.4.2 ArangoDB

ArangoDB is a multi-model NoSQL database that supports the key-value, document, and graph data models. One can use and freely combine all the supported data models even in a single query. All data in ArangoDB is stored as documents that closely follow the JSON format, making it possible to store complex and nested data.

ArangoDB can operate as a distributed and highly scalable database cluster for better performance as well as resilience through replication and automatic failover.

Data stored in ArangoDB can mainly be queried using the *ArangoDB Query Language (AQL)*, which is ArangoDB's own declarative query language. It shares some similarities with the SQL used in relational databases, making it easier to learn for users with an SQL background. AQL can be used to retrieve and modify data stored in ArangoDB. However, it does not support data definition operations, such as creating or dropping databases, collections, and indexes. The AQL constructs and syntax are described in detail in Chapter 5.

Another way of accessing data in ArangoDB is through the HTTP API [14], which provides endpoints for creating, accessing and manipulating the data. This API also lets a user validate and execute AQL queries, among other options.

ArangoDB also provides drivers for a number of programming languages, such as JavaScript, Java, and PHP. It also has a built-in JavaScript framework called *Foxx* [12]. This framework allows users to write data-centric HTTP microservices that run directly inside of ArangoDB.

For the purpose of this thesis, we chose ArangoDB over OrientDB as our multi-model database of interest. This is partially due to the lack of research on the usage of ArangoDB as an RDF store. To our knowledge, there has not been any attempt at SPARQL-to-AQL query translation in existing research papers, whereas there have been many studies on the translation of SPARQL queries to SQL. This made ArangoDB a more interesting choice. We also decided to use ArangoDB due to a number of performance benchmark tests that show ArangoDB performing better than OrientDB [73, 69, 27].

We shall now describe the way data is modeled in ArangoDB, the storage engines it offers, and the index structures it supports.

Data Modelling

In ArangoDB, data takes the form of JSON objects called documents that are stored in collections. Thus, arbitrarily nested data structures can be represented in a single document.

Each document must contain the special attributes `_id`, `_key` and `_rev`. The `_key` attribute stores the primary key value, which uniquely identifies a document within a collection. Furthermore, each document is uniquely identified by its document handle across all collections in the same database. This document handle is stored in the `_id` attribute. Different revisions of the same document can be distinguished by their document revision, which is stored in the `_rev` attribute. These three attributes are system attributes, as they are automatically created by ArangoDB for each document. However, if desired, users can also provide the `_key` value themselves when creating a document. The `_id` and `_key` values are immutable once the document has been created, while the `_rev` value is maintained automatically by ArangoDB.

Edge documents are special documents used in ArangoDB graphs which, in addition to the mandatory system attributes, must contain the attributes `_from` and `_to`. These two attributes contain document handles of the starting and ending documents respectively, that is `_from` contains the `_id` value of the start vertex and `_to` contains the `_id` value of the end vertex. Thus, edge documents are connection documents that reference other documents.

Documents are grouped into collections, such that each collection is uniquely identified by a collection name. Every document in a given collection can have arbitrary attribute-value pairs, that is documents in a single collection do not need to have the same structure, although in practice their structure is usually similar.

There are two types of collections, these being the *document collection*, and the *edge collection*. An edge collection is a special type of collection that stores only edge documents, whereas a document collection stores regular documents. Thus, normal documents and edge documents cannot be mixed in the same collection.

In the context of graphs, document collections are also referred to as *vertex collections*. Generally, two documents, i.e. vertices, stored in document collections are linked by a document, i.e. edge, stored in an edge collection. This is the graph data model provided by ArangoDB, which follows the mathematical concept of a directed, labeled graph. The only difference is that an edge does not simply have a label, but instead is a document containing multiple properties.

The vertices of an ArangoDB graph can be documents from multiple different collections, and its edges can be edge documents from multiple different edge collections. ArangoDB edge documents can also be used as vertices in an ArangoDB graph, however, this feature is not considered or used in this thesis.

ArangoDB allows users to work with named graphs or anonymous graphs. Named graphs are defined by the user by providing the name of the graph, and the vertex and edge collections that form the graph. ArangoDB ensures graph integrity for named graphs, both when inserting and when removing edges or vertices, thus using named graphs comes with an additional cost. On the other hand, users might not require the power of named graphs. In this case, anonymous graphs, i.e. loosely coupled collections, can be used in traversals. An anonymous graph is a graph that is not strictly defined, but rather is loosely specified by the user in a traversal query, by providing a list of edge collections to be used in the traversal. Which vertex collections are used in the traversal is determined by the edges in the specified edge collections.

Collections exist inside of databases. A single ArangoDB instance can contain multiple databases, each identified by a unique name. The default database in ArangoDB is named `_system` and cannot be removed. Database users are managed through this database, and their credentials are valid for all the databases of a server instance.

Storage Engines

ArangoDB offers two storage engines, these being *RocksDB* and *MMFiles*. The storage engine is responsible for persisting ArangoDB documents on disk, loading and managing copies in memory, as well as providing indexes and caching to improve query performance. The engine must be selected for the whole server or cluster when installing ArangoDB, thus it is not possible to mix engines as they both work very differently.

RocksDB is currently the default storage engine. It is optimized for large datasets, specifically datasets that do not fit in main memory. Using this engine, indexes are always stored on disk, however, caches are used for better performance. RocksDB uses document-level locking allowing for concurrent writes on different documents. Moreover, writes do not block reads, and vice versa.

The Memory-Mapped Files (MMFiles) storage engine is optimized for datasets that fit into main memory. Indexes are always stored in memory and are rebuilt on each startup. Although this gives better overall performance, the start-up time is longer when using this engine. MMFiles enables very fast concurrent reads, however, locking is implemented on the collection level, and writes block reads.

Indexes

ArangoDB automatically indexes the `_id`, `_key`, `_from`, and `_to` attributes of documents, but it also allows users to create additional indexes on other attributes of documents, including nested attributes.

It supports a range of indexes including the *hash index*, which is useful when searching for equality, and the *skiplist index*, most useful for range queries. Some index types, such as the *full-text index*, allow indexing just one attribute, whereas other index types allow indexing multiple attributes at the same time, i.e. a combined index. Using a combined index is very useful in case it is known that certain attributes are often present together in a given search condition in an AQL query.

A hash index can be used to quickly find documents having specific attribute values. It supports equality look-ups but not range queries or sorting due to it being an unsorted index. A hash index can be created as a combined index and is only used by a query if all the index attributes are present in the search condition, and are all compared using the equality operator. A hash index can be declared as unique in order to create a unique constraint on the values of the indexed attributes.

The hash index and skiplist index have recently been deprecated on the RocksDB engine, as they have been unified into what is called a *persistent index*, available only on this engine. This type of index is used for queries that perform equality look-ups on the index attributes, if either all the index attributes or a leftmost prefix of the index attributes, are covered by the query. In the latter case, the last used index attribute in the prefix can also be used for a range look-up.

Every edge collection has an automatically created *edge index*, providing quick access to edge documents by their `_from` or `_to` attributes. Edge indexes are used by AQL when performing equality look-ups on these two attributes in an edge collection, however, an edge index cannot be utilized for range queries or sorting. The `_from` and `_to` attributes can also be used in user-defined indexes, in this case called *vertex centric indexes.* They are so-called because they can be considered localized indexes for an edge collection, which are stored at every single vertex. This is because this type of index indexes a combination of a vertex, the edge direction and any arbitrary set of other attributes on the edges attached to the vertex.

Vertex centric indexes are useful when running more specific queries involving other attributes of edge documents. They are used in graph traversals when the optimizer finds appropriate `FILTER` statements. However, it is not guaranteed that such an index is always used, as the optimizer may estimate that using the standard edge index is better. To make sure that the optimizer makes use of a vertex centric index, the user should, if possible, apply the required filter conditions on the path of the graph traversal, instead of the vertex or edge.

2. Preliminary Definitions

In this chapter, we provide some formal definitions that will be used in consecutive chapters in this thesis. Specifically, we provide definitions for components of RDF, JSON, and ArangoDB.

2.1 RDF

Definition 2.1 (Domains $\mathbb{I}$, $\mathbb{L}$, $\mathbb{B}$)**.** Let $\mathbb{I}$ be the domain of IRIs, $\mathbb{L}$ be the domain of literals, and $\mathbb{B}$ be the domain of blank nodes, such that $\mathbb{I}$, $\mathbb{L}$, and $\mathbb{B}$ are pairwise disjoint infinite sets.

Definition 2.2 (RDF terms)**.** The set of *RDF terms* is defined as $\mathbb{T} = \mathbb{I} \cup \mathbb{L} \cup \mathbb{B}$.

Definition 2.3 (RDF Triple)**.** Assume that $\mathbb{I}$ is the domain of IRIs, $\mathbb{L}$ is the domain of literals, and $\mathbb{B}$ is the domain of blank nodes.
Then $t = (s, p, o) \in (\mathbb{I} \cup \mathbb{B}) \times \mathbb{I} \times (\mathbb{I} \cup \mathbb{B} \cup \mathbb{L})$ is called an *RDF triple*, where s, p, and o are called the *subject*, *predicate*, and *object* of the triple respectively.

Definition 2.4 (RDF Graph)**.** An *RDF Graph* is a finite set of RDF triples $\{t_1, t_2, \ldots, t_n\}$ for some $n \in \mathbb{N}_0$.

A Named Graph is an RDF graph that is identified by an IRI.

Definition 2.5 (Named Graph)**.** Let G be an RDF Graph and $u \in \mathbb{I}$ be an IRI identifier. Then (u, G) is a *named graph*.

A Default Graph is an RDF graph that does not have a name and may be empty.

Definition 2.6 (Default Graph)**.** Let G be an RDF Graph and $\perp \notin \mathbb{I}$. Then $(\perp, G)$ is a special type of graph called the *default graph*.

An RDF Dataset is made up of one default graph and zero or more named graphs.

Definition 2.7 (RDF Dataset)**.** Let $(\perp, G_1)$ be the default graph and $\{(u_2, G_2), \ldots, (u_n, G_n)\}$ for some $n \in \mathbb{N}$ be a set of named graphs. Then $\mathcal{D} = \{(\perp, G_1), (u_2, G_2), \ldots, (u_n, G_n)\}$ is an *RDF Dataset*, such that $\forall i, j \in \mathbb{N}, i \neq j : u_i \neq u_j$, and $\forall i \in \mathbb{N}, i \geq 2 : u_i \neq \perp$.

2.2 JSON

Definition 2.8 (Domains $\mathbb{A}$, $\mathbb{H}$)**.** Let $\mathbb{A}$ be the domain of all JSON attributes, and $\mathbb{H}$ be the domain of all atomic JSON values, such that $\mathbb{A} \subset \mathbb{H}$.

Definition 2.9 (JSON object)**.** A *JSON object* $O = \{(a_1, h_1), (a_2, h_2), \ldots, (a_n, h_n)\}$ for some $n \in \mathbb{N}_0$ is an unordered set of attibute-value pairs, such that $\forall i \in \mathbb{N}_0, i \leq n : h_i \in \mathbb{H}$ or h_i is a nested JSON object or array. We say that h_i is the value associated with the attribute $a_i \in \mathbb{A}$, (a_i, h_i) is called a property, and $O.a_i = O[a_i] = h_i$. We assume that attributes are distinct, i.e. $\forall i, j \in \mathbb{N}_0, i, j \leq n, i \neq j : a_i \neq a_j$.

To access the value of the attribute of a JSON object, one can use the dot notation or the bracket notation, as in the above definition.

Definition 2.10 (JSON array)**.** A *JSON array* $A = \langle h_1, h_2, \ldots, h_n \rangle$ for some $n \in \mathbb{N}_0$ is a sequence of values, such that $\forall i \in \mathbb{N}_0, i \leq n : h_i \in \mathbb{H}$ or h_i is a nested JSON object or array.

2.3 ArangoDB

Definition 2.11 (ArangoDB Document)**.** An *ArangoDB document* is a JSON object $\{(a_1, h_1), (a_2, h_2), (a_3, h_3), \ldots, (a_n, h_n)\}$, for some $n \in \mathbb{N}, n \geq 3$, such that $a_1 = \texttt{\_id}$, $a_2 = \texttt{\_key}$, and $a_3 = \texttt{\_rev}$.

Definition 2.12 (ArangoDB Edge Document)**.** An *ArangoDB edge document* is an ArangoDB document $\{(a_1, h_1), (a_2, h_2), (a_3, h_3), (a_4, h_4), (a_5, h_5), \ldots, (a_n, h_n)\}$ for some $n \in \mathbb{N}, n \geq 5$, such that $a_4 = \texttt{\_from}$, and $a_5 = \texttt{\_to}$.

Definition 2.13 (Domain $\mathbb{X}$)**.** Let $\mathbb{X}$ be the domain of all ArangoDB collection names.

Definition 2.14 (ArangoDB Collection)**.** Let $D = \{d_1, d_2, \ldots, d_k\}$ be a finite set of ArangoDB documents for some $k \in \mathbb{N}_0$. Then an *ArangoDB collection* is a pair $C = (n, D)$, such that $n \in \mathbb{X}$ and $\forall i, j \in \mathbb{N}, i, j \leq k, i \neq j : d_i.\_key \neq d_i.\_key$, i.e. keys of documents in a collection are unique.

Definition 2.15 (ArangoDB Graph)**.** Let $D = \{d_1, d_2, \ldots, d_n\}$ be a finite set of ArangoDB documents for some $n \in \mathbb{N}_0$, and $C = \{c_1, c_2, \ldots, c_k\}$ be a finite set of ArangoDB edge documents for some $k \in \mathbb{N}_0$. Then an *ArangoDB graph* for D and C is defined as $G_{D,C} = (V, E)$, where:

- $V = \{v_i \mid i \in \mathbb{N}, i \leq n : v_i = d_i.\texttt{\_id}\}$ is a set of *vertices*
- $E = \{(v_i, v_j) \mid v_i, v_j \in V : l \in \mathbb{N}, l \leq k, v_i = c_l.\texttt{\_from}, v_j = c_l.\texttt{\_to}\}$ is a set of *edges*

Definition 2.16 (ArangoDB Database)**.** An *ArangoDB database* is a set of ArangoDB collections $\{(n_1, D_1), (n_2, D_2), \ldots, (n_k, D_k)\}$ for some $k \in \mathbb{N}_0$, such that $\forall i, j \in \mathbb{N}, i \neq j : n_i \neq n_j$, i.e. names of collections are unique. Moreover, $\forall d_l, d_m \in (D_1 \cup D_2 \cup \cdots \cup D_k), d_l \neq d_m : d_l.\texttt{\_id} \neq d_m.\texttt{\_id}$, i.e. all documents in a database have unique document handles.

3. Related Work

In this chapter, we will mention and describe existing research related to this thesis. This includes research on the usage of relational or NoSQL databases as RDF stores. Moreover, we present existing research on how RDF data can be stored in ArangoDB, which is particularly relevant to this thesis.

3.1 RDF Data Storage in ArangoDB

In [69], Samuelsen proposes different ways of representing and storing RDF data in ArangoDB. He presents three formats for storing the RDF data, all of them based on the graph data model of ArangoDB.

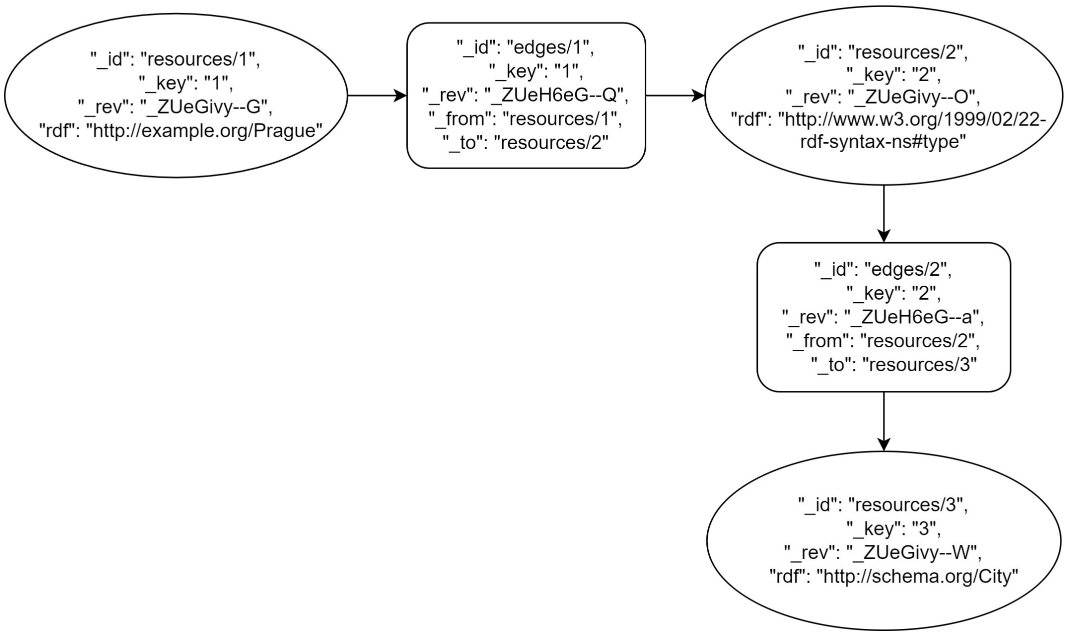
His first approach is to create three ArangoDB documents for each triple, each representing the subject, predicate, and object of the triple respectively. Two ArangoDB edge documents connecting the subject document to the predicate document, and then the predicate document to the object document are also created. The author calls this the *direct representation* and it gives the most expressive representation of RDF and can essentially be applied in any graph database. However, both multi-model as well as general graph databases are not optimized to work with a very large number of small objects. Thus, this approach gives the least satisfactory performance of the three approaches. An example of how RDF data is transformed using this approach is shown in Figure 3.1.

The second approach is similar to the first, however, in this case, two ArangoDB documents are created for each triple, representing the subject and object of the triple. An edge document connecting those two documents is then created, and the predicate IRI is stored as a property of that edge document. The advantage of using this approach over the first is that we gain a reduction in the size of the data, due to using fewer documents. This approach is called the *direct representation with edge values* and an example of this transformation approach is shown in Figure 3.2.

In the third approach, an ArangoDB document is created for each IRI in the RDF dataset. All the literal values linked to that IRI by some predicate are mapped to JSON properties in the document of that IRI, where the attribute is the predicate name, and the value is the literal value. Any `rdf:type` and `rdfs:label` values for an IRI are also mapped to properties in the document of the IRI for faster filtering by type, and for vertex labeling respectively. Edge documents are then created to represent predicates linking IRI nodes, such that the predicate IRI is stored as an attribute in the edge document as in the second approach. The author refers to this storage model as the *flattened representation* of RDF. The advantage of using this approach is that we significantly decrease the amount of ArangoDB documents used, and we have a lower number of small objects to join together in a query. An example transformation is given in Figure 3.3.

(⟨http://example.org/Prague⟩ ,
⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ,
⟨http://schema.org/City⟩)

(a) Sample RDF triple



"_id": "resources/1", "_key": "1", "_rev": "_ZUeGivy--G", "rdf": "http://example.org/Prague"
"_id": "edges/1", "_key": "1", "_rev": "_ZUeH6eG--Q", "_from": "resources/1", "_to": "resources/2"
"_id": "resources/2", "_key": "2", "_rev": "_ZUeGivy--O", "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
"_id": "edges/2", "_key": "2", "_rev": "_ZUeH6eG--a", "_from": "resources/2", "_to": "resources/3"
"_id": "resources/3", "_key": "3", "_rev": "_ZUeGivy--W", "rdf": "http://schema.org/City"

(b) ArangoDB graph representation

Figure 3.1: Samuelsen's direct representation of RDF

We believe that the problem with the third approach is that when querying data, the user has to be very familiar with the data to know which predicates usually have literal values, in order to look for those values in the ArangoDB document of a given IRI, and which predicates usually have IRI or blank node values, in which case a graph traversal must occur. This might not be an issue when working with RDFS and other well-known vocabularies, but might be if the RDF data uses other less-known vocabularies. This can also make queries more complicated to write for the user. Moreover, in the scope of SPARQL to AQL query translation, dealing with this model would require a mapping or configuration of which predicates are stored as attributes of the ArangoDB document of a resource, and which are represented as edge documents, to know how to translate SPARQL triple patterns into AQL query constructs. Thus, this would require more input from the user.

Choosing one of the three presented RDF storage models should be a decision based on how expressive or efficient the representation should be.

The author implemented an extension to the DataGraft [17] platform, which transforms tabular data into RDF using some user-provided schema mapping, and then into the flattened representation of RDF for storage in ArangoDB. In

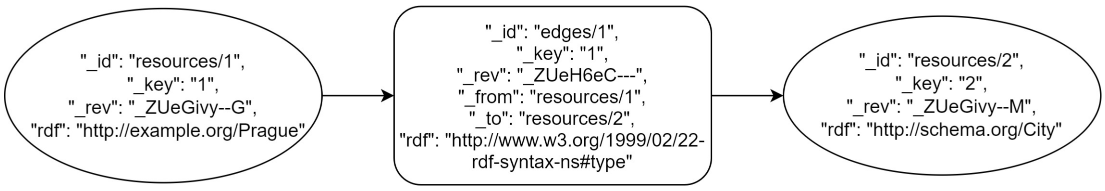


Figure 3.2: Samuelsen's direct with edges representation of RDF for the sample data in Figure 3.1a

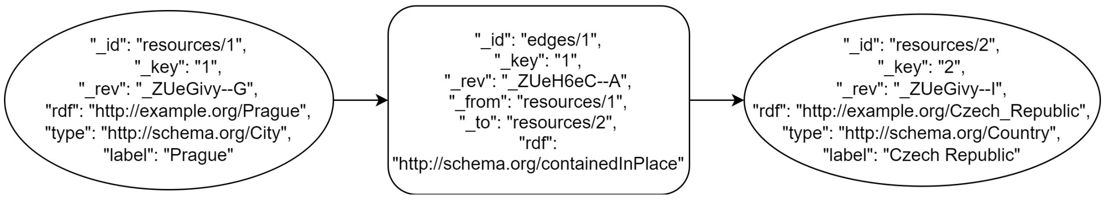


Figure 3.3: Samuelsen's flattened representation of RDF

our opinion, the limitation of this implementation is that it does not allow a user to simply provide data that is already in RDF form.

The author ran benchmark tests to compare the MMFiles and RocksDB engines of ArangoDB with OrientDB, Neo4j [26], MongoDB [24] and PostgreSQL [31]. The results from the benchmark tests showed that an ArangoDB solution performs just as good and in many cases even better than the other more traditional storage solutions. In most cases, ArangoDB performed significantly better than its multi-model counterpart OrientDB.

The author also ran benchmark tests to compare the performance of querying transformed RDF data in ArangoDB, stored using the RocksDB engine, to querying the same RDF data stored in the triplestore Apache Jena Fuseki. ArangoDB outperformed the triplestore in all test cases. Thus, this benchmark result supports the idea that using a multi-model database to store and query RDF data can provide better scalability and faster response times, meeting with the main challenges of current RDF stores.

The author states that for easier adoption of existing RDF users, there is a need for research on the possibility of translating SPARQL queries into AQL queries, to provide users with the ability to query data in ArangoDB using SPARQL. This would ease the possible transmission to the usage of ArangoDB as an RDF store. The need for such a service is not only to adopt users but also to be able to support the very essence of using semantic data and RDF, which is the possibility of combining datasets by using the same query language and data format. It would help eliminate the issue of interoperability with other RDF systems and make the approach applicable within the Semantic Web. This is a motivation for our work.

3.2 RDF Data Querying in Relational databases

Chaloupka [55] proposed a way of querying relational data using SPARQL and R2RML [32]. The motivation for it was to make it easier to map relational data into RDF for publishing and sharing. Although there are several tools that can be used to dump a relational database to an RDF file, this is not efficient, as one would need to create a large dump upon every change and load it into memory.

The R2RML language is a language used to express how to map a relational database to an RDF dataset, that is to present relational data as RDF. It describes how particular data in the relational database should be mapped into RDF triples. R2RML is a standard proposed by W3C for such data mapping.

The main idea behind his work was to create a virtual SPARQL endpoint which converts every SPARQL query into an SQL query, executes the SQL query on the relational database, and then transforms the result into the expected format based on the SPARQL query form used. The SPARQL query translation involves a number of transformation steps. The SPARQL query is first transformed into its algebraic representation, to which information from the R2RML mapping can then be added. The SPARQL algebra tree is then optimized, after which it is transformed directly into the SQL query expression.

In paper [61], Kiminki et al. presented a more generalized approach for SPARQL to SQL query translation using an intermediate abstract query language. Using this intermediate language, they were looking to bridge the differences between SPARQL and SQL, and to ensure that the translation is not fixed to a particular SQL schema or dialect.

They created a prototype implementation of their approach called Type-ARQuE. The query translator in Type-ARQuE first translates a SPARQL query into an intermediate query, after which the intermediate query goes through some transformation and optimization passes. Finally, the translator translates the intermediate query into an SQL query using a particular target dialect.

Their approach produces a single SQL query for a single SPARQL query and does not require result post-processing other than for result presentation. Moreover, the translator produces SQL queries that use native SQL data types where possible and avoids creating sub-selects in queries.

In [70], Sequeda and Miranker propose a system called Ultrawrap which creates a representation of a relational database as an RDF graph, called a Tripleview, using unmaterialized SQL views, and then translates SPARQL queries into SQL queries on those views. The approach enables real-time consistency between the relational data and its RDF representation.

Ultrawrap translates an SQL schema and its constraints to an OWL ontology, using an augmented direct mapping. Thus, a user does not need to have knowledge of the relational schema, learn a mapping language or manually create the mapping, although a custom R2RML mapping can be provided by the user if desired. The Tripleview is then created over the relational data in the form of one or more SQL views, based on the mapping.

Briefly, the transformation to OWL consists of representing tables as ontological classes, foreign key attributes of a table as object properties and all other attributes as datatype properties. Tables that represent a many-to-many relationship are translated to object properties. A Tripleview is an SQL view consisting of 5 attributes, these being subject, primary key of subject, predicate, object, and primary key of object, such that the query optimizer can exploit indexes on the primary keys. Instead of using IRIs to uniquely identify resources, the table name is concatenated with the primary key value or the attribute name.

Bizer and Seaborne [52] present a declarative language called D2RQ, used to describe mappings from application-specific relational database schemas to RDFS and OWL ontologies. D2RQ makes it possible for RDF applications to treat legacy relational databases as virtual RDF graphs, which can be queried using RDQL.

D2RQ is implemented using an Apache Jena graph, such that it wraps one or more local relational databases into a virtual, read-only RDF graph. The central object within the D2RQ mapping language is the ClassMap, which represents a class or a group of similar classes from the ontology, and how the relational data should be mapped to them.

D2RQ rewrites RDQL queries and Jena API calls into SQL queries that are application and data model specific. The result sets of these SQL queries are then transformed into RDF triples.

The authors performed benchmark tests comparing the performance of D2RQ to the performance of the Jena2 database backend. The results showed that D2RQ is very competitive when dealing with the most commonly used subject-predicate-object query patterns.

3.3 RDF Data Querying in NoSQL databases

In [71], Szeremeta and Tomaszu describe the idea of using a document-oriented, JSON-based NoSQL database as an RDF store. They propose an RDF/JSON [34] serialization which would allow storing RDF data in such a NoSQL database and to query it efficiently. Their approach is also cache-friendly.

In [54], Bouhali and Lauren investigate how RDF could be transformed for storage in NoSQL graph databases. Such databases provide very efficient tools for handling large volumes of complex data and are particularly powerful for retrieving relationships between objects. This makes them an interesting option for storing and manipulating RDF data. They propose two approaches for converting RDF data to fit the Property Graph model [66], in which data is organized as nodes, relationships, and properties on the nodes and relationships. Since RDF can be treated as a submodel of a property graph, any set of RDF triples can be translated into a property graph without any structural loss.

In the first approach, triples are transformed into a structure similar to the flattened representation described in [69]. Each RDF resource becomes a graph node. If the object of a triple is a literal, the predicate and object become a property name and value respectively, and this property is added to the existing or created node representing the subject resource. If the object is a resource, then both the subject and object become graph nodes, and a relationship is created between them, specifying the name of the predicate as the relationship type. If the predicate is `rdf:type`, the label of the subject node is set to the name of the object. An example is given in Figure 3.4. The main advantage of this method is that each triple can be processed one at a time, making it possible to handle a stream of input RDF triples and convert them on the fly. The main drawback of this approach is that it does not make full use of the property graph model, since no properties are defined over relationships.

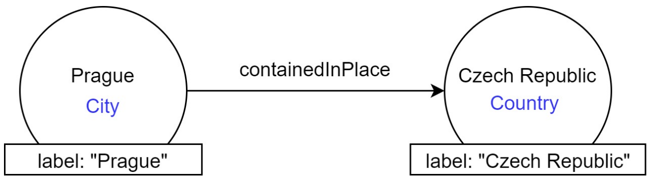


Figure 3.4: Property Graph representation of RDF

The second approach is an extension of the first, which uses a mapping table to transform RDF triples to a property graph. This mapping table helps with handling cases where many triples with the same predicate are defined for the same subject resource, in which case the different object values should be stored in a collection. It also helps in keeping track of which data type should be attributed to some literal, in cases where the type is not clearly specified in the RDF data. Another significant feature of this approach is that structural mappings can be used to refactor the generated property graph data by replacing certain relationships with others. This feature can be used to reduce the amounts of relationships or modify them as required for a particular scope. Such a structural mapping would be provided by the user based on his needs. However, the user would have to have good knowledge of his dataset as well as the final graph representation and graph database used, which is rarely the case.

The authors implemented an RDF conversion tool that utilizes the second approach, and imports the transformed data into a Neo4j graph database for experimentation and testing.

In paper [64], Michel et al. propose a method for querying legacy data in MongoDB using SPARQL. Their approach is to first translate a SPARQL query into a pivot abstract query using MongoDB-to-RDF mappings represented in the xR2RML [62] language, and then translate the pivot query into a particular MongoDB query. xR2RML is an extension of R2RML and RML [57] that also allows mapping non-relational data to RDF. Rewriting the SPARQL query into the pivot abstract query is independent of the target database. Thus, this method can be used also when translating SPARQL queries to other query languages other than that of MongoDB, as stated by the same authors in [63]. By using this intermediate pivot abstract query, one can handle and resolve any discrepancies between the expressiveness of SPARQL and the target database query language.

Similarly, Botoeva et al. [53] also propose a way of querying data in MongoDB using SPARQL. However, their approach was to extend the Ontology-Based Data Access (OBDA) paradigm for use with non-relational databases, particularly to support MongoDB. They create a relational view over the MongoDB database, based on its schema. A given SPARQL query can then be rewritten into a relational algebra query over the relational view, and then translated into MongoDB aggregate queries.

The authors of [67] propose storing RDF data in MongoDB using the JSON-LD [22] serialization. They compared the performance of querying such data in MongoDB using its query language, to the performance of querying the equivalent RDF and SQL datasets in Apache Jena TDB [9] and MySQL [25] respectively. MongoDB outperformed Apache Jena TDB significantly in the majority of cases. It also performed better than MySQL in many cases, however, MySQL performed better for queries with complex filters, joins, and sub-queries.

In [59], Haque and Perkins present the approach of using the column-oriented NoSQL database HBase [4] as an RDF data store and HiveQL as the query language. They proposed a way of modeling RDF triples in an HBase table and implemented a prototype translator which transforms a given SPARQL query into a HiveQL query. The latter query can then be run over the HBase database and return data.

Jena-HBase [60] is a distributed, scalable triple store that also uses HBase to store its data and can be used with the Jena framework. It uses several HBase tables with different schemas to store RDF triples. It provides a number of custom-built RDF data storage layouts for HBase, each having different trade-offs in terms of query performance and storage. It also supports SPARQL processing through the implementation of appropriate Jena interfaces.

In [56], Cudré-Mauroux et al. evaluated the usage of the document-oriented, NoSQL database Couchbase [15] as an RDF store. Since Couchbase has native support for JSON documents, they mapped RDF triples onto JSON documents, such that all triples sharing the same subject are in one document and the subject is used as the key of that document. Couchbase then provides MapReduce views on top of the stored JSON documents, which are used for querying the transformed RDF data.

Bikakis et al. [50] introduce the SPARQL2XQuery Framework, which creates an environment in which SPARQL queries are automatically translated to queries written in XQuery [47]. It was created to enable access to XML data across the Web, in the form of Linked Data.

The SPARQL2XQuery Framework contains a component that automatically generates an OWL ontology that captures the semantics of an XML Schema. Another component, called the Mapping Generator, takes both the XML schema and the generated ontology as input, and automatically generates and maintains the mappings between them. The framework supports both automatic as well as manual mapping specification between ontologies and XML Schemas. Thus, users can provide their own OWL or RDFS ontology mappings.

The SPARQL queries posed on the ontology are translated to XQuery expressions by a Query Translator component, and the query results are then transformed into the desired format by a Query Result Transformer component.

Finally, in [58], Fischer et al. present a complete translation of SPARQL to XQuery, which does not make assumptions about the database schema. They implemented a translator that takes any SPARQL query and turns it into an XQuery expression. Their performance test results show that even with limited optimizations, XQuery is typically as fast as and often faster than native SPARQL.

4. SPARQL Algebra

In this chapter we describe the SPARQL algebra and the properties of its parts. The following definitions are based on the algebra described in [42] and [65]. Only the query parts and operators required for this thesis are defined. We also introduce and define a SPARQL algebra tree and its nodes, for use in SPARQL query transformations.

4.1 Basic Definitions

The two basic elements of the SPARQL query language are the RDF term and the query variable. An RDF term can be an IRI, a literal or a blank node as described in section 1.1.2.

A query variable is a name which can be bound to some RDF term during the evaluation of the result of a SPARQL query. Query variables are distinguished by using '?' as a prefix to the name (e.g. `?var1`, `?var2`).

A blank node is indicated by either the label form, such as `_:bn1`, or, if it is used in only one place in the query syntax, by the abbreviated form `[]`. It is important to note that blank nodes in SPARQL queries do not reference specific blank nodes in the RDF data being queried, rather they act as undistinguished variables.

RDF terms and variables are used to compose triple patterns intended to match triples in the queried dataset.

SPARQL also supports what are called *simple literals*. These are literals that have no datatype and no language tag, and we consider them to be separate from the literals we call RDF terms. Simple literals are used in comparison conditions, arithmetic operations as well as function calls in a query.

In the following formal definitions, we will define the basic components of the SPARQL query language. The purpose of these definitions is not to describe the syntactic aspects of these components, but rather their logical representation.

Definition 4.1 (Domains $\mathbb{V}, \mathbb{S}$)**.** Let $\mathbb{V}$ be the domain of all query variables disjoint from the set $\mathbb{T}$ of RDF terms and $\mathbb{S}$ be the domain of simple literals disjoint from $\mathbb{T} \cup \mathbb{V}$.

Definition 4.2 (Triple pattern)**.** Assume that $\mathbb{V}$ is the set of all query variables disjoint from the set $\mathbb{T}$ of RDF terms.
Then $t = (s, p, o) \in (\mathbb{I} \cup \mathbb{B} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{L} \cup \mathbb{B} \cup \mathbb{V})$ is called a *triple pattern*.

A graph pattern is one of:

- Basic graph pattern

- Group graph pattern
- Filter graph pattern
- Optional graph pattern
- Union graph pattern
- Graph graph pattern
- Minus graph pattern

We now formally define the basic graph pattern and the group graph pattern, as they are the basis of other graph patterns.

Definition 4.3 (Basic graph pattern)**.** A *basic graph pattern* $P = \{t_1, t_2, \ldots, t_n\}$ is a set of triple patterns such that $n \in \mathbb{N}_0$. If $n = 0$, it is called the empty graph pattern.

Definition 4.4 (Group graph pattern)**.** A *group graph pattern* $P = \{P_1, P_2, \ldots, P_n\}$ is a set of graph patterns, such that $n \in \mathbb{N}_0$ and $\forall i \in \mathbb{N}_0, i \leq n : P_i$ is a graph pattern. If $n = 0$, P is called an empty group pattern.

Definition 4.5 (Variables in a graph pattern)**.** Let P be a graph pattern. Then $var(P) = \{v \mid v \in \mathbb{V}$ and $\exists(s, p, o) \in P$ such that $(s = v) \vee (p = v) \vee (o = v)\}$.

Definition 4.6 (Blank nodes in a graph pattern)**.** Let P be a graph pattern. Then $blank(P) = \{b \mid b \in \mathbb{B}$ and $\exists(s, p, o) \in P$ such that $(s = b) \vee (p = b) \vee (o = b)\}$.

When evaluated against an RDF dataset, graph patterns return solution mappings, that is data values bound to variables in the pattern during pattern matching. Solution mappings can also simply be called solutions. The empty group pattern matches any graph, including the empty graph, with one solution that does not bind data to any variables.

Definition 4.7 (Solution mapping)**.** A *solution mapping* is a partial function $\mu : \mathbb{V} \to \mathbb{T}$ mapping a set of query variables to a set of RDF terms. The domain of μ, denoted by $dom(\mu)$, is the subset of $\mathbb{V}$ where μ is defined.

A variable that is mapped to a value is called a bound variable. Unbound variables are those that are not mapped to a value by the solution mapping, that is they are not in the domain of the solution mapping. By substituting variables in a graph pattern by the values bound to them by a solution mapping, we obtain a single data result.

Definition 4.8 (Variable substitution)**.** Let P be a graph pattern and μ be a solution mapping such that $var(P) \subseteq dom(\mu)$. Then $\mu(P)$ is the result of replacing every variable $v \in var(P)$ by $\mu(v)$.
Thus $\forall t = (s, p, o) \in P : \mu(t) = (s', p', o')$, where:

- $s' = \begin{cases} \mu(s), & \text{if } s \in var(t) \\ s, & \text{otherwise} \end{cases}$

- $p' = \begin{cases} \mu(p), & \text{if } p \in var(t) \\ p, & \text{otherwise} \end{cases}$

- $o' = \begin{cases} \mu(o), & \text{if } o \in var(t) \\ o, & \text{otherwise} \end{cases}$

When performing set operations on two sets of solution mappings, we have to consider the compatibility of solution mappings in one set, to solution mappings in the other.

Definition 4.9 (Compatible solution mappings)**.** Two solution mappings μ_1 and μ_2 are *compatible* if and only if $\forall v \in dom(\mu_1) \cap dom(\mu_2)$: $\mu_1(v) = \mu_2(v)$, i.e. $\mu_1(v)$ and $\mu_2(v)$ are the same RDF term. Otherwise we say that μ_1 and μ_2 are incompatible when they are not compatible, i.e. if $\exists v \in dom(\mu_1) \cap dom(\mu_2)$: $\mu_1(v) \neq \mu_2(v)$.

A multiset or sequence of solution mappings contain all the possible variable bindings for a graph pattern. We refer to a solution multiset when we do not care about the order of the solutions. When the order of the solutions is important, we consider solution mappings to be in a particular sequence.

Definition 4.10 (Solution multiset)**.** A *solution multiset* is a multiset of solution mappings $\Omega = \{\mu_1, \mu_2, ..., \mu_n\}$ for some $n \in \mathbb{N}_0$, such that $dom(\Omega) = \bigcup_{i=1}^{n} dom(\mu_i)$.

Definition 4.11 (Solution sequence)**.** A *solution sequence* is a sequence of solution mappings $\Psi = \langle \mu_1, \mu_2, ..., \mu_n \rangle$ for some $n \in \mathbb{N}_0$, such that $dom(\Psi) = \bigcup_{i=1}^{n} dom(\mu_i)$.

The result of any SPARQL query, executed over an RDF dataset, can be considered a sequence of solution mappings.

Definition 4.12 (Blank node mapping)**.** A *blank node mapping* σ is a partial function $\sigma : \mathbb{B} \rightarrow \mathbb{T}$ mapping a set of query blank nodes to a set of RDF terms. The domain of σ, denoted by $dom(\sigma)$, is the subset of $\mathbb{B}$ where σ is defined. Given a mapping σ and a graph pattern P such that $blank(P) \subseteq dom(\sigma)$, $\sigma(P)$ is the result of replacing every blank node $b \in blank(P)$ by $\sigma(b)$.
Thus $\forall t = (s, p, o) \in P : \sigma(t) = (s', p', o')$ where:

- $s' = \begin{cases} \sigma(s), & \text{if } s \in blank(t) \\ s, & \text{otherwise} \end{cases}$
- $p' = \begin{cases} \sigma(p), & \text{if } p \in blank(t) \\ p, & \text{otherwise} \end{cases}$
- $o' = \begin{cases} \sigma(o), & \text{if } o \in blank(t) \\ o, & \text{otherwise} \end{cases}$

Blank nodes in SPARQL graph patterns are treated as variables, however these cannot be projected and thus cannot be included in a solution mapping. The same blank node label cannot be used in two different basic graph patterns in the same query.

Definition 4.13 (Pattern instance mapping)**.** A *pattern instance mapping*, M, is the combination of a solution mapping and a blank node mapping. Let P be a basic graph pattern, μ be a solution mapping, and σ be a blank node mapping such that $var(P) \subseteq dom(\mu)$ and $blank(P) \subseteq dom(\sigma)$. Then $M(P) = \mu(\sigma(P))$ is the result of replacing every blank node $b \in blank(P)$ with $\sigma(b)$ and every $v \in var(P)$ with $\mu(v)$.

4.2 Algebra Operators

In this section, we define the operators that are used to evaluate a SPARQL query expression.

4.2.1 Basic Operators

We first define the basic SPARQL operators. For the following definitions, we only consider solution multisets, as the order of solutions is not important when working with these operators.

The *Bgp* operator is used to evaluate a basic graph pattern against a particular graph in a given RDF dataset.

Definition 4.14 (BGP operator)**.** Let $\mathcal{D}$ be an RDF dataset, P be a basic graph pattern, M be a pattern instance mapping, σ be a blank node mapping, and $u \in \mathbb{I} \cup \bot$ be an IRI or undefined. Then:

$$\begin{aligned} Bgp(P, \mathcal{D}, u) = \{\mu \mid dom(\mu) = var(P) \text{ and } \exists (u, G) \in \mathcal{D}, M :& \\ M(P) = \mu(\sigma(P)) \subseteq G\}& \end{aligned}$$

The *Join* operator is used to join the results of two graph patterns when they are compatible.

Definition 4.15 (Join operator)**.** Let Ω_1, and Ω_2 be solution multisets. Then:

$$Join(\Omega_1, \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$$

The *Difference* operator is used to discard solutions for a graph pattern that are compatible with some solution for another graph pattern.

Definition 4.16 (Difference operator)**.** Let Ω_1, and Ω_2 be solution multisets. Then:

$$Difference(\Omega_1, \Omega_2) = \{\mu \mid \mu \in \Omega_1 \text{ and } \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\}$$

The *LeftJoin* operator returns all the solutions for the first graph pattern even if they are not compatible with any of the solutions for the second graph pattern.

Definition 4.17 (LeftJoin operator)**.** Let Ω_1, and Ω_2 be solution multisets. Then:

$$\mathit{LeftJoin}(\Omega_1, \Omega_2) = \mathit{Join}(\Omega_1, \Omega_2) \cup \mathit{Difference}(\Omega_1, \Omega_2)$$

The *Union* operator is used to perform a set union of the results of two graph patterns, i.e. to combine two solution multisets into one.

Definition 4.18 (Union operator)**.** Let Ω_1, and Ω_2 be solution multisets. Then:

$$Union(\Omega_1, \Omega_2) = \{\mu \mid \mu \in \Omega_1 \cup \Omega_2\}$$

The *Filter* operator is used to discard solutions for a graph pattern that do not satisfy some condition.

Definition 4.19 (Filter operator)**.** Let Ω be a multiset of solution mappings, and E be a filter condition. Then:

$$Filter(\Omega, E) = \{\mu \mid \mu \in \Omega, \mu \vDash E\}$$

such that $\mu \vDash E$ means that μ satisfies E.

The *Minus* operator removes solutions for one graph pattern that are compatible with any solution for a second graph pattern, but only if the compatible solutions bind at least one common variable.

Definition 4.20 (Minus operator)**.** Let Ω_1, and Ω_2 be solution multisets. Then:

$$\begin{aligned} Minus(\Omega_1, \Omega_2) = \{\mu \mid \mu \in \Omega_1 \text{ such that } \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible} \\ \text{or } \mathit{dom}(\mu) \text{ and } \mathit{dom}(\mu') \text{ are disjoint}\} \end{aligned}$$

The *Extend* operator is used to add a variable mapping to a solution mapping, i.e. to extend the domain of a solution mapping, by binding the result of the evaluation of an expression to the given variable.

Definition 4.21 (Extend operator)**.** Let μ be a solution mapping, Ω be a multiset of solution mappings, $v \in \mathbb{V}$ be a variable and *expr* be an expression such that $\mathit{expr}(\mu)$ is the result of replacing each variable in *expr* with its mapping in μ. Then:

$$Extend(\mu, v, \mathit{expr}) = \mu \cup \{(v \to \mathit{value})\} \mid v \notin \mathit{dom}(\mu) \text{ and } \mathit{value} = \mathit{expr}(\mu)\}$$

$$Extend(\mu, v, \mathit{expr}) = \text{undefined if } v \in \mathit{dom}(\mu)$$

$$Extend(\mu, v, \mathit{expr}) = \mu \text{ if } \mathit{expr}(\mu) \text{ is an error}$$

$$Extend(\Omega, v, \mathit{expr}) = \{Extend(\mu, v, \mathit{expr}) \mid \mu \in \Omega\}$$

such that $\mathit{expr}(\mu)$ is the result of replacing every variable $v \in \mathit{var}(\mathit{expr})$ by $\mu(v)$

4.2.2 Solution Modifiers

Solution modifiers are used to modify the results of graph pattern matching and aggregation, if applicable. A solution sequence modifier is one of the below:

- **Order By** - put the solutions in some order
- **Project** - select certain variables
- **Distinct** - ensure solutions in the sequence are unique
- **Reduced** - eliminate some of the non-unique solutions
- **Offset** - control where the solutions start from in the overall sequence of solutions
- **Limit** - restrict the number of solutions

We now define the algebra operators used to evaluate the above solution modifiers. Due to there not being a well-defined behavior for the Reduced modifier, beyond that duplicates may or may not be eliminated, we do not formally define an operator for it. Ignoring this modifier and not eliminating any duplicates is still in accordance with the SPARQL specification.

For the following definitions, we only consider solution sequences, as the order of solutions is important when working with these operators.

Definition 4.22 (OrderBy operator)**.** Let Ψ be a solution sequence and C be an ordering condition. Then $OrderBy(\Psi, C) = \langle \mu_1, \mu_2, \ldots, \mu_n \rangle$ for some $n \in \mathbb{N}_0$ such that $\forall i \in \mathbb{N}, i \leq n : \mu_i \in \Psi$ and the sequence satisfies C.

All other solution modifiers must preserve any ordering given by the OrderBy operation.

Definition 4.23 (Project operator)**.** Let Ψ be a solution sequence, μ be a solution mapping, and PV be a set of variables. Then:

$$Project(\mu, PV) = \{\mu(v) \mid v \in dom(\mu) \text{ and } v \in PV\}$$

$$Project(\Psi, PV) = \langle Project(\mu_1, PV), Project(\mu_2, PV), \ldots, Project(\mu_n, PV) \rangle$$

such that $\forall i \in \mathbb{N}, i \leq n : \mu_i \in \Psi$.

Definition 4.24 (Distinct operator)**.** Let Ψ be a solution sequence.
Then $Distinct(\Psi) = \langle \mu_1, \mu_2, \ldots, \mu_n \rangle$ such that $\forall i \in \mathbb{N}, i \leq n : \mu_i \in \Psi$ and there must not exist two mappings $\mu_1, \mu_2 \in Distinct(\Psi)$ that are identical, i.e. $\forall i, j \in \mathbb{N}, \mu_i, \mu_j \in Distinct(\Psi) : \mu_i = \mu_j \implies i = j$.

The *Slice* operator below combines Offset and Limit together, such that *start* is the offset value and *length* is the limit value.

Definition 4.25 (Slice operator)**.** Let $\Psi = \langle \mu_1, \mu_2, ..., \mu_n \rangle$ be a solution sequence for some $n \in \mathbb{N}_0$, and $start, length \in \mathbb{N}_0$ be two integer numbers. Then

$$Slice(\Psi, start, length) = \langle \mu_{start+1}, \mu_{start+2}, \ldots, \mu_{MIN(start+length,n)} \rangle$$

such that $\forall i \in \mathbb{N}, (start + 1) \leq i \leq MIN(start + length, n) : \mu_i \in \Psi$.

4.3 Evaluation of SPARQL query parts

We now define how different parts of a SPARQL query expression are evaluated using the operators defined in the previous section.

For the below definitions, we assume that $\mathcal{D}$ is an RDF dataset, and $u \in \mathbb{I} \cup \{\perp\}$ is either an IRI or undefined, such that $\exists(u, G) \in \mathcal{D}$, i.e u indicates the active graph, which is either the default graph or a named graph in $\mathcal{D}$.

Definition 4.26 (Evaluation of a basic graph pattern)**.** Let P be a basic graph pattern. Then:

$$[\![P]\!]_{\mathcal{D},u} = Bgp(P, \mathcal{D}, u)$$

Although SPARQL does not provide a JOIN keyword, the group graph pattern combines graph patterns such that it joins every pair of compatible solution mappings for two graph patterns together.

Definition 4.27 (Evaluation of a group graph pattern)**.** Let $\{P_1, P_2\}$ be a group graph pattern. Then

$$[\![\{P_1, P_2\}]\!]_{\mathcal{D},u} = Join([\![(P_1)]\!]_{\mathcal{D},u}, [\![(P_2)]\!]_{\mathcal{D},u})$$

In a group graph pattern, there can be optional values due to the presence of an optional graph pattern. An optional graph pattern is defined using the `OPTIONAL` clause, and contains variables that do not necessarily have to be bound. This type of graph pattern either matches a graph, thus adding bindings to one or more solutions, or it keeps a solution unchanged without adding any additional bindings.

Definition 4.28 (Evaluation of an optional graph pattern)**.** Let (P_1 `OPTIONAL` P_2) be an optional graph pattern such that P_1 and P_2 are graph patterns. Then:

$$[\![(P_1 \texttt{ OPTIONAL } P_2)]\!]_{\mathcal{D},u} = LeftJoin([\![(P_1)]\!]_{\mathcal{D},u}, [\![(P_2)]\!]_{\mathcal{D},u})$$

SPARQL provides the `UNION` and `MINUS` keywords to perform set operations on the results of two graph patterns.

Definition 4.29 (Evaluation of a union graph pattern)**.** Let (P_1 `UNION` P_2) be a union graph pattern such that P_1 and P_2 are graph patterns. Then:

$$[\![(P_1 \texttt{ UNION } P_2)]\!]_{\mathcal{D},u} = Union([\![P_1]\!]_{\mathcal{D},u}, [\![P_2]\!]_{\mathcal{D},u})$$

Definition 4.30 (Evaluation of a minus graph pattern)**.** Let (P_1 `MINUS` P_2) be a minus graph pattern such that P_1 and P_2 are graph patterns. Then:

$$[\![(P_1 \texttt{ MINUS } P_2)]\!]_{\mathcal{D},u} = Minus([\![P_1]\!]_{\mathcal{D},u}, [\![P_2]\!]_{\mathcal{D},u})$$

The `FILTER` clause can be used to add constraints on graph patterns. It removes solutions that do not satisfy the filtering constraints.

Definition 4.31 (Evaluation of a filter graph pattern). Let $(P\ \texttt{FILTER}\ R)$ be a filter graph pattern such that P is a graph pattern and R is a filter condition. Then:

$$[\![(P\ \texttt{FILTER}\ R)]\!]_{\mathcal{D},u} = \mathit{Filter}([\![P]\!]_{\mathcal{D},u}, R)$$

The graph graph pattern provides the capability of accessing the named graphs in the RDF dataset. This type of graph pattern is identified using a `GRAPH` clause, which changes the current active graph, that will be used for matching the basic graph pattern(s) within the graph graph pattern.

Definition 4.32 (Evaluation of a graph graph pattern). Let $\mathcal{D} = \{(\bot, G_1), (u_2, G_2), \ldots, (u_n, G_n)\}$ for some $n \in \mathbb{N}$ be an RDF dataset, P be a graph pattern, $v \in \mathbb{V}$ be a variable, and $u_G \in \mathbb{I} \cup \{\bot\}$ be an IRI or undefined, such that $\exists i, i \in \mathbb{N}, i \leq n : (u_i, G_i) \in \mathcal{D}, u_G = u_i$. Then:

$$[\![(\texttt{GRAPH}\ u_G\ P)]\!]_{\mathcal{D},u} = [\![P]\!]_{\mathcal{D},u_G}$$

$$[\![(\texttt{GRAPH}\ v\ P)]\!]_{\mathcal{D},u} = \begin{cases} \bigcup_{i=1}^{n} \mathit{Extend}([\![P]\!]_{\mathcal{D},u_i}, v, u_i), & \text{if } v \notin \mathit{var}(P) \\ \bigcup_{i=1}^{n} \mathit{Join}([\![P]\!]_{\mathcal{D},u_i}, \{(v \to u_i)\}), & \text{if } v \in \mathit{var}(P) \end{cases}$$

.

The `BIND` clause can be used to bind the result of an expression, or even a simple constant value, to a variable.

Definition 4.33 (Evaluation of a Bind clause). Let $(P\ \texttt{BIND}\ X)$ be a bind clause such that X is a pair (v, e) where $v \in \mathbb{V}$ is a variable and e is an expression. Then:

$$[\![(P\ \texttt{BIND}\ X)]\!]_{\mathcal{D},u} = \mathit{Extend}([\![P]\!]_{\mathcal{D},u}, v, e)$$

Another way of assigning a value to a variable is by using the `VALUES` clause, which allows specifying multiple values for one or more variables. Using the `VALUES` clause, data can be written directly in a graph pattern or added to a query. `VALUES` provides inline data as a solution multiset, and that data can then be combined with the results of some query evaluation by a join operation.

Definition 4.34 (Evaluation of a Values clause). Let $(P\ \texttt{VALUES}\ \Omega)$ be a values clause such that P is a graph pattern and Ω is a solution multiset. Then:

$$[\![(P\ \texttt{VALUES}\ \Omega)]\!]_{\mathcal{D},u} = \mathit{Join}([\![P]\!]_{\mathcal{D},u}, \Omega)$$

If a variable has no value for a particular solution mapping in the `VALUES` clause, the keyword `UNDEF` is used instead of an RDF term, indicating that the binding for that variable should be omitted.

4.4 SPARQL Algebra Tree

For this thesis, we need to be able to transform a SPARQL query expression into a SPARQL algebra tree. For this reason, in this section, we define the algebra tree and all the nodes that can form part of a tree. Many of these nodes represent SPARQL algebra operators defined earlier in this chapter, or other functions offered by SPARQL.

In order to determine the type of a SPARQL tree node, we introduce an arbitrary function called *type* which takes a tuple representing a tree node and returns the first element in that tuple, which is always the identifier associated with the node type of the given node.

Definition 4.35 (SPARQL query algebra tree)**.** A *SPARQL query algebra tree* is a tuple containing three elements $(N, r, \mathit{children})$, where $N = \{n_1, n_2, \ldots, n_i\}$ for some $i \in \mathbb{N}$ is the set of nodes in the tree, $r \in N$ is the root node of the tree, and *children* is a function that maps every node in N to its sequence of child nodes.
Let n be a node in a SPARQL query algebra tree. Then:

$$\begin{aligned}\mathit{type}(n) \in \{&\mathtt{BGP}, \mathtt{GRAPH}, \mathtt{TABLE}, \mathtt{JOIN}, \mathtt{LEFTJOIN}, \mathtt{UNION}, \mathtt{MINUS},\\ &\mathtt{EXTEND}, \mathtt{FILTER}, \mathtt{PROJECT}, \mathtt{DISTINCT}, \mathtt{SLICE}, \mathtt{ORDER}\}\end{aligned}$$

and $\mathit{children}(n) = \langle c_1, c_2, \ldots, c_m\rangle$ for some $m \in \{0, 1, 2\}$ is a sequence of child nodes of node n.

A BGP node is used to represent a basic graph pattern and can also store an optional set of IRIs of named graphs against which it is meant to be evaluated.

Definition 4.36 (BGP node)**.** Let $P = \{t_1, t_2, \ldots, t_m\}$ be a set of triple patterns for some $m \in \mathbb{N}_0$, $v \in \mathbb{V} \cup \{\perp\}$ be a variable which can be undefined, and $G = \{u_1, u_2, \ldots, u_j\}$ be a set of graph IRIs for some $j \in \mathbb{N}_0$, which can be undefined. Then $n = (\mathtt{BGP}, P, G, v)$ is a *BGP node* such that $\mathit{type}(n) = \mathtt{BGP}$, $\mathit{children}(n) = \langle\rangle$ and $G \neq \perp$ if $v \in \mathbb{V}$.

A Graph node represents a `GRAPH` clause in the query and indicates that the active graph will need to change when evaluating BGP nodes in its subtree.

Definition 4.37 (Graph node)**.** Let $u \in \mathbb{I} \cup \{\perp\}$ be an IRI or undefined, and $v \in \mathbb{V} \cup \{\perp\}$ be a variable or undefined. Then $n = (\mathtt{GRAPH}, u, v)$ is a *Graph node* such that $\mathit{type}(n) = \mathtt{GRAPH}$, $\mathit{children}(n) = \langle c_1\rangle$, $u \neq \perp$ if $v = \perp$, and $u = \perp$ if $v \in \mathbb{V}$.

A SolutionTable node represents the solution multiset specified in a `VALUES` clause.

Definition 4.38 (SolutionTable node)**.** Let Ω be a solution multiset.
Then $n = (\mathtt{TABLE}, \Omega)$ is a *SolutionTable node* such that $\mathit{type}(n) = \mathtt{TABLE}$ and $\mathit{children}(n) = \langle\rangle$.

A Join node is used to represent a join operation between graph patterns.

Definition 4.39 (Join node)**.** A node $n = (\texttt{JOIN})$ is a *Join node* such that $type(n) = \texttt{JOIN}$, and $children(n) = \langle c_1, c_2 \rangle$.

A LeftJoin node is used to represent a left join operation between graph patterns, i.e. an `OPTIONAL` graph pattern.

Definition 4.40 (LeftJoin node)**.** A node $n = (\texttt{LEFTJOIN})$ is a *LeftJoin node* such that $type(n) = \texttt{LEFTJOIN}$, and $children(n) = \langle c_1, c_2 \rangle$.

A Union node is used to represent a set union operation on the solution multisets for two graph patterns.

Definition 4.41 (Union node)**.** A node $n = (\texttt{UNION})$ is a *Union node* such that $type(n) = \texttt{UNION}$, and $children(n) = \langle c_1, c_2 \rangle$.

A Minus node is used to represent a set minus operation on the solution multisets for two graph patternss.

Definition 4.42 (Minus node)**.** A node $n = (\texttt{MINUS})$ is a *Minus node* such that $type(n) = \texttt{MINUS}$, and $children(n) = \langle c_1, c_2 \rangle$.

An Extend node is used to represent an extend operation on a graph pattern, i.e. a `BIND` clause. In the definition of this node as well as some other following node definitions, we refer to *SPARQL expression trees*. A SPARQL expression tree is defined in Definition 4.49 and its nodes are defined in consecutive definitions.

Definition 4.43 (Extend node)**.** Let $X = \{(v_1, E_1), (v_2, E_2), \dots, (v_m, E_m)\}$ for some $m \in \mathbb{N}$ be a set of variable-expression pairs such that $\forall i \in \mathbb{N}, i \leq m : v_i \in \mathbb{V}$ and E_i is an expression tree. Then $n = (\texttt{EXTEND}, X)$ is an *Extend node* such that type(n) = `EXTEND`, and $children(n) = \langle c_1 \rangle$.

A Filter node is used to represent a filter condition on a graph pattern.

Definition 4.44 (Filter node)**.** Let E be a SPARQL expression tree. Then $n = (\texttt{FILTER}, E)$ is a *Filter node* such that $type(n) = \texttt{FILTER}$, and $children(n) = \langle c_1 \rangle$.

We now define the nodes used to represent solution modifiers in a SPARQL query.

Definition 4.45 (Project node)**.** Let $P = \{v_1, v_2, \dots, v_m\}$ be a set of variables for some $m \in \mathbb{N}$ such that $\forall i \in \mathbb{N}, i \leq m : v_i \in \mathbb{V}$. Then $n = (\texttt{PROJECT}, P)$ is a *Project node* such that $type(n) = \texttt{PROJECT}$, and $children(n) = \langle c_1 \rangle$.

Definition 4.46 (Distinct node)**.** A node $n = (\texttt{DISTINCT})$ is a *Distinct node* such that $type(n) = \texttt{DISTINCT}$, $children(n) = \langle c_1 \rangle$, and $type(c_1) = \texttt{PROJECT}$.

Definition 4.47 (Slice node)**.** Let $\mathit{offset}, \mathit{limit} \in \mathbb{N}_0$ be two integers.
Then $n = (\texttt{SLICE}, \mathit{offset}, \mathit{limit})$ is a *Slice node* such that $type(n) = \texttt{SLICE}$ and $children(n) = \langle c_1 \rangle$.

Definition 4.48 (Order node)**.** Let $C = \langle (E_1, d_1), (E_2, d_2), \ldots, (E_m, d_m) \rangle$ for some $m \in \mathbb{N}$ be a sequence of sort conditions such that $\forall i \in \mathbb{N}, i \leq m : E_i$ is an expression tree and $d_i \in \{\texttt{ASC}, \texttt{DESC}\}$. Then $n = (\texttt{ORDER}, C)$ is an *Order node* such that $\mathit{type}(n) = \texttt{ORDER}$ and $\mathit{children}(n) = \langle c_1 \rangle$.

We shall now define a SPARQL expression tree and its nodes. SPARQL expression trees are used to represent filtering conditions, arithmetic operations, function calls, and other expressions in a SPARQL query.

Definition 4.49 (SPARQL expression tree)**.** A *SPARQL expression tree* is a tuple containing three elements $(N, r, \mathit{children})$, where $N = \{n_1, n_2, \ldots, n_i\}$ for some $i \in \mathbb{N}$ is the set of nodes in the tree, r is the root node of the tree, and *children* is a function that maps every node in N to its sequence of child nodes. Let n be a node in a SPARQL expression tree. Then:

$$\begin{aligned}\mathit{type}(n) \in \{&\texttt{AND}, \texttt{OR}, \texttt{NOT}, \texttt{EQUALS}, \texttt{NOTEQUALS}, \texttt{VALUE}, \texttt{VARIABLE}, \texttt{BOUND}, \texttt{CONCAT},\\ &\texttt{LANG}, \texttt{LANGMATCHES}, \texttt{ADD}, \texttt{SUBTRACT}, \texttt{MULTIPLY}, \texttt{DIVIDE}, \texttt{GREATERTHAN},\\ &\texttt{GREATERTHANEQUAL}, \texttt{LESSTHAN}, \texttt{LESSTHANEQUAL}, \texttt{EXISTS}, \texttt{NOTEXISTS}\}\end{aligned}$$

and $\mathit{children}(n) = \langle c_1, c_2, \ldots, c_m \rangle$ for some $m \in \mathbb{N}_0$ is a sequence of child nodes of node n.

Definition 4.50 (LogicalAnd node)**.** A node $n = (\texttt{AND})$ is a *LogicalAnd node* such that $\mathit{type}(n) = \texttt{AND}$, and $\mathit{children}(n) = \langle c_1, c_2 \rangle$.

Definition 4.51 (LogicalOr node)**.** A node $n = (\texttt{OR})$ is a *LogicalOr node* such that $\mathit{type}(n) = \texttt{OR}$, and $\mathit{children}(n) = \langle c_1, c_2 \rangle$.

Definition 4.52 (LogicalNot node)**.** A node $n = (\texttt{NOT})$ is a *LogicalNot node* such that $\mathit{type}(n) = \texttt{NOT}$, and $\mathit{children}(n) = \langle c_1 \rangle$.

Definition 4.53 (Equals node)**.** A node $n = (\texttt{EQUALS})$ is an *Equals node* such that $\mathit{type}(n) = \texttt{EQUALS}$, and $\mathit{children}(n) = \langle c_1, c_2 \rangle$.

Definition 4.54 (NotEquals node)**.** A node $n = (\texttt{NOTEQUALS})$ is a *NotEquals node* such that $\mathit{type}(n) = \texttt{NOTEQUALS}$, and $\mathit{children}(n) = \langle c_1, c_2 \rangle$.

A *Value node* is used to represent an IRI or literal.

Definition 4.55 (Value node)**.** Let $v \in (\mathbb{I} \cup \mathbb{L} \cup \mathbb{S})$ be an IRI or literal. Then node $n = (\texttt{VALUE}, v)$ is a *Value node* such that $\mathit{type}(n) = \texttt{VALUE}$, and $\mathit{children}(n) = \langle\rangle$.

A *Variable node* is used to represent a SPARQL variable.

Definition 4.56 (Variable node)**.** Let $v \in \mathbb{V}$ be a variable.
Then node $n = (\texttt{VARIABLE}, v)$ is a *Variable node* such that $\mathit{type}(n) = \texttt{VARIABLE}$, and $\mathit{children}(n) = \langle\rangle$.

A *Bound node* is a node that represents the `BOUND` function, which checks whether a given variable has been bound or not.

Definition 4.57 (Bound node)**.** A node $n = (\texttt{BOUND})$ is a *Bound node* such that $type(n) = \texttt{BOUND}$, and $children(n) = \langle c_1 \rangle$.

A *Language node* is a node that represents the `LANG` function, which extracts the language tag of a given literal if it has one.

Definition 4.58 (Language node)**.** A node $n = (\texttt{LANG})$ is a *Language node* such that $type(n) = \texttt{LANG}$, and $children(n) = \langle c_1 \rangle$.

A *LanguageMatches node* is a node that represents the `LANGMATCHES` function, which checks whether the language tag of a given literal matches a particular language tag or is contained within a range of language tags.

Definition 4.59 (LanguageMatches node)**.** A node $n = (\texttt{LANGMATCHES})$ is a *LanguageMatches node* such that $type(n) = \texttt{LANGMATCHES}$, and $children(n) = \langle c_1, c_2 \rangle$.

A *Concat node* is a node that represents the `CONCAT` function, which concatenates a given sequence of string literals.

Definition 4.60 (Concat node)**.** A node $n = (\texttt{CONCAT})$ is a *Concat node* such that $type(n) = \texttt{CONCAT}$, and $children(n) = \langle c_1, c_2, \ldots, c_m \rangle$ for some $m \in \mathbb{N}$.

An *Arithmetic node* is a node that represents an arithmetic operation.

Definition 4.61 (Arithmetic node)**.** Let $t \in \{\texttt{ADD}, \texttt{SUBTRACT}, \texttt{MULTIPLY}, \texttt{DIVIDE}, \texttt{GREATERTHAN}, \texttt{GREATERTHANEQUAL}, \texttt{LESSTHAN}, \texttt{LESSTHANEQUAL}\}$ be a node type. Then node $n = (t)$ is an *Arithmetic node* such that $type(n) = t$, and $children(n) = \langle c_1, c_2 \rangle$.

An *Exists node* is used to represent an `EXISTS` condition in a `FILTER` clause, while a *NotExists node* represents a `NOT EXISTS` filtering condition.

Definition 4.62 (Exists node)**.** Let A be a SPARQL query algebra tree. Then a node $n = (\texttt{EXISTS}, A)$ is an *Exists node* such that $type(n) = \texttt{EXISTS}$, and $children(n) = \langle\rangle$.

Definition 4.63 (NotExists node)**.** Let A be a SPARQL query algebra tree. Then a node $n = (\texttt{NOTEXISTS}, A)$ is a *NotExists node* such that $type(n) = \texttt{NOTEXISTS}$, and $children(n) = \langle\rangle$.

5. ArangoDB AQL

In this chapter, we describe the constructs available in AQL and the syntax of an AQL query expression. We also introduce and define an AQL query tree and its nodes, for use in SPARQL-to-AQL query transformations.

5.1 Syntax and Semantics of AQL queries

For the purpose of this thesis, we consider the following main operations offered in AQL:

Keyword	Operation
FOR	Collection or JSON array iteration
RETURN	Results projection
FILTER	Results filtering
SORT	Result sorting
LIMIT	Result slicing
LET	Variable assignment
COLLECT	Result grouping

The result of an AQL query is an array of JSON values, that can be JSON objects and arrays, and items in the result do not necessarily have a homogeneous structure. The result is projected using the `RETURN` operator. An AQL query always includes a `RETURN` operation, in fact an AQL query can be a single `RETURN` statement projecting a static JSON value, as can be seen in figure 5.1. If duplicate results are unwanted, `RETURN DISTINCT` can be used to project unique items.

```
RETURN "Hello World"
```

Figure 5.1: Example AQL query expression with only a `RETURN` operation

The `FOR` operator is used to iterate over documents from a collection by specifying the collection name, and a variable used to store the current document during each iteration. This can be likened to the for loop used in many programming languages. A simple example is shown in figure 5.2, of a query that returns the name of each user in the `users` collection. During iteration, the order of the documents is undefined. To traverse the documents in a particular order, the `SORT` operation must be used to order the documents before performing other operations on them.

```
FOR user IN users
  RETURN user.name
```

Figure 5.2: Example AQL query expression with a simple for loop

It is normal to encounter documents that do not have all the attributes that are queried in an AQL query. In this case, the non-existing attributes in the document are treated as if they exist with a value of null.

The `FOR` operator can also be used to perform graph traversals, although the syntax is not the same as when we loop over a document collection. For the purpose of this thesis, we will consider the graph traversal syntax depicted in figure 5.3, and ignore other possible syntax components that we do not require.

```
FOR vertex[, edge[, path]]
  IN [min[..max]]
  OUTBOUND|INBOUND|ANY startVertex
  edgeCollection1, ..., edgeCollectionN
```

Figure 5.3: AQL graph traversal syntax

In the graph traversal version, the for loop emits one, two, or three variables depending on the user's needs. The `vertex` variable is mandatory and represents the current vertex in a traversal. The `edge` and `path` variables are optional, where `edge` represents the current edge in a traversal, and `path` represents the current graph path and is an object containing two attributes - `vertices` and `edges`. The `vertices` attribute is an array of all the vertices on the path, and `edges` is an array of all the edges on the path.

The `startVertex` is the vertex from which the traversal will originate. This can be specified either in the form of an ID string or in the form of a document having the `_id` attribute. All other values will lead to an empty result, and the same applies in case the specified document does not exist.

The `min` and `max` numeric values are both optional and represent the minimal and maximal depth for the traversal, respectively. Thus, edges and vertices that are not within the min-max inclusive range are not returned by the query. The smallest possible value for each of them is 0. Moreover, `max` cannot be specified without specifying `min`. If not specified, `min` defaults to 1, and `max` defaults to the value of `min`.

One of the `OUTBOUND`, `INBOUND` and `ANY` keywords must be specified in order to follow outgoing, incoming, or edges pointing in either direction in the traversal, respectively.

Operations like `FILTER`, `SORT` and `LIMIT` can be added to the body of a `FOR` loop to narrow down and order the result. These operations do not have to occur in any fixed order, although the order can influence the result significantly. Moreover, it is allowed to specify multiple statements of the same operation type in a query, even in the same `FOR` loop.

The `SORT` operation sorts the array of already produced intermediate results in the current block. The `SORT` statement allows specifying one or more sort criteria, and an optional direction for each criteria by using the `ASC` or `DESC` keywords. If no direction is specified, ascending order is used by default. In the example given in figure 5.4, the collection of users is sorted by name in ascending order and the names are then returned in the sorted order.

```
FOR user IN users
  SORT user.name
  RETURN user.name
```

Figure 5.4: Example AQL query expression with `SORT`

The `LIMIT` operation slices the result array using an offset value and a count value. It reduces the number of elements in the result array to at most the specified count value. Specifying an offset value is optional, so a limit statement can be in the form `LIMIT` *offset*, *count* or simply `LIMIT` *count*. The example given in figure 5.5 sorts the collection of users as in the example in figure 5.4, however in this case, due to the `LIMIT` operation, only the first 10 user names from the sorted list are projected.

```
FOR user IN users
  SORT user.name
  LIMIT 10
  RETURN user.name
```

Figure 5.5: Example AQL query expression with `LIMIT`

AQL allows the user to assign values to variables in a query, using the `LET` operation, as shown in the sample query in Figure 5.6. The assigned variable is introduced in the scope the `LET` statement is placed in. All variables that are assigned a value must have a name that is unique within the context of the query and cannot have the same name as any collection used in the same query. Once a value is assigned to a variable, that variable cannot be re-assigned later in the query.

The `LET` operation can be used to store the results of a computation or subquery into a variable, like in the example given in figure 5.7. This is useful when we want to avoid repeating the same computations at multiple points in the query, or simply want to make the query more readable.

A join operation is expressed by nesting two `FOR` loops and applying `FILTER` conditions on the inner loop, which in this case can be considered join conditions,

```
FOR u IN users
  LET hobbies = u.hobbies
  RETURN { name : u.name, hobbies : hobbies }
```

Figure 5.6: Example AQL query expression with a simple `LET` operation

```
FOR u IN users
  LET friends = (
    FOR f IN friends
      FILTER u._id == f.userId
      RETURN f
  )
  RETURN { user : u, friends : friends }
```

Figure 5.7: Example AQL query expression with `LET` operation assigning subquery

to filter out the data that does not satisfy the join operation. In the sample query given in Figure 5.8, a join operation is used to find the city where a user lives.

```
FOR u IN users
  FOR c IN cities
    FILTER u.city == c._id
    RETURN { user: u, city: c }
```

Figure 5.8: Example AQL query expression with `JOIN` operation

The `COLLECT` operation can be used to group an array by one or multiple criteria. The `COLLECT` statement eliminates all local variables in the current scope, such that after the grouping operation, only the variables introduced by the COLLECT statement itself are available.

There are several different syntaxes for `COLLECT` operations, used to achieve different results. In the example given in figure 5.9, the `COLLECT` operation acts similar to the `DISTINCT` modifier, by finding and storing all the distinct values in `u.city` into the variable `city`.

To also preserve the list of users grouped by city, the `INTO` keyword needs to be used as in figure 5.10. It is also possible to group the data by more than one attribute simply by specifying multiple comma-separated grouping criteria after the `COLLECT` keyword.

The `COLLECT` operation also provides a `WITH COUNT` clause that can be used to efficiently count the number of group members. In its simplest form, it can be used as in figure 5.11, such that no grouping takes place and it simply counts the number of items.

```
FOR u IN users
  COLLECT city = u.city
  RETURN city
```

Figure 5.9: Example AQL query expression with a simple `COLLECT` operation

```
FOR u IN users
  COLLECT city = u.city INTO groupedUsers
  RETURN {
    "city" : city,
    "usersInCity" : groupedUsers
  }
```

Figure 5.10: Example AQL query expression with a `COLLECT INTO` operation

```
FOR u IN users
  COLLECT WITH COUNT INTO amountOfUsers
  RETURN amountOfUsers
```

Figure 5.11: Example AQL query expression with a `COLLECT WITH COUNT` operation

There are also other variants of the `COLLECT` syntax, however, these will not be mentioned here as they are not within the scope of this thesis.

5.2 AQL Query Tree

For the purpose of this thesis, we use an *AQL Query Tree* to syntactically represent an AQL query expression. Thus, by simply traversing and printing an AQL query tree, we obtain the AQL query expression it represents.

In an AQL query, all the variable names used must be different from the names of any collections used in the same query. For this reason, we define two domains below, ie. the domain of AQL variables, and the domain of collection names. As we define them here, these are not the infinite domains of all the possible variables and collection names in ArangoDB, as in that case, these domains would intersect. Instead, we consider both domains to be finite sets that are pairwise disjoint.

Definition 5.1 (Domains $\mathbb{W}$, $\mathbb{C}$)**.** Let $\mathbb{W}$ be the domain of AQL variables, and $\mathbb{C}$ be the domain of ArangoDB collection names, such that $\mathbb{W}$ and $\mathbb{C}$ are pairwise disjoint finite sets.

To determine the type of an AQL tree node, we use the same arbitrary function *type* which we introduced in Chapter 4, which takes a tuple representing a tree node and returns the first element in that tuple.

Definition 5.2 (AQL Query Tree)**.** An *AQL query tree* is a tuple containing three elements $(N, r, children)$, where $N = \{n_1, n_2, \ldots, n_i\}$ for some $i \in \mathbb{N}$ is the set of nodes in the tree, $r \in N$ is the root node of the tree, and *children* is a function that maps every node in N to its sequence of child nodes.
Let n be a node in an AQL query algebra tree. Then:

$$\begin{aligned} type(n) \in \{&\texttt{ITERATION}, \texttt{GRAPHITERATION}, \texttt{ASSIGNMENT}, \texttt{SEQUENCE}, \texttt{FILTER}, \texttt{NEST}, \\ &\texttt{SORT}, \texttt{PROJECT}, \texttt{LIMIT}, \texttt{COLLECT}\} \end{aligned}$$

and $children(n) = \langle c_1, c_2, \ldots, c_m \rangle$ for some $m \in \mathbb{N}_0$ is a sequence of child nodes of node n.

An Iteration node is used to represent a regular `FOR` loop over a collection or array. In the definition of this node as well as in other following node definitions, we refer to *AQL expression trees*. An AQL expression tree is defined in Definition 5.13 and its nodes are defined in consecutive definitions.

Definition 5.3 (Iteration node)**.** Let $v \in \mathbb{W}$ be a variable and E be an AQL expression tree. Then node $n = (\texttt{ITERATION}, v, E)$ is an *Iteration node* such that $type(n) = \texttt{ITERATION}$, and $children(n) = \langle\rangle$.

A Graph Iteration node is used to represent a graph traversal `FOR` loop.

Definition 5.4 (Graph Iteration node)**.** Let $v, e, p \in \mathbb{W}$ be variables, $min, max \in \mathbb{N}_0$ be two integers, s be an ArangoDB document, $d \in \{\texttt{OUTBOUND}, \texttt{INBOUND}, \texttt{ANY}\}$ be the traversal direction, and $E = \{c_1, c_2, \ldots, c_m\}$ for some $m \in \mathbb{N}$ be a set of names of edge collections.
Then node $n = (\texttt{GRAPHITERATION}, v, e, p, s, min, max, d, E)$ is a *Graph Iteration node* such that $type(n) = \texttt{GRAPHITERATION}$, and $children(n) = \langle\rangle$.

An Assignment node is used to represent a `LET` statement in a query.

Definition 5.5 (Assignment node)**.** Let $v \in \mathbb{W}$ be a variable and E be an AQL expression tree. Then node $n = (\texttt{ASSIGNMENT}, v, E)$ is an *Assignment node* such that $type(n) = \texttt{ASSIGNMENT}$, and $children(n) = \langle\rangle$.

A Sequence node is used when there is a sequence of `LET` statements followed by a `FOR` loop, on the same level of nesting.

Definition 5.6 (Sequence node)**.** A node $n = (\texttt{SEQUENCE})$ is a *Sequence node* such that $type(n) = \texttt{SEQUENCE}$, and $children(n) = \langle c_1, c_2, \ldots, c_m \rangle$ for some $m \in \mathbb{N}$.

A Filter node is used to represent a `FILTER` statement.

Definition 5.7 (Filter node)**.** Let E be an AQL expression tree. Then $n = (\texttt{FILTER}, E)$ is a *Filter node* such that $\mathit{type}(n) = \texttt{FILTER}$, and $\mathit{children}(n) = \langle c_1 \rangle$.

We use a Nest node to indicate when `LET` statements or `FOR` loops are nested within an external `FOR` loop.

Definition 5.8 (Nest node)**.** A node $n = (\texttt{NEST})$ is a *Nest node* such that $\mathit{type}(n) = \texttt{NEST}$, and $\mathit{children}(n) = \langle c_1, c_2 \rangle$.

A Sort node is used to represent a `SORT` condition.

Definition 5.9 (Sort node)**.** Let $C = \langle (E_1, d_1), (E_2, d_2), \ldots, (E_m, d_m) \rangle$ be a sequence of sort conditions such that $\forall i \in \mathbb{N}, i \leq m : E_i$ is an AQL expression tree and $d_i \in \{\texttt{ASC}, \texttt{DESC}\}$. Then $n = (\texttt{SORT}, C)$ is a *Sort node* such that $\mathit{type}(n) = \texttt{SORT}$ and $\mathit{children}(n) = \langle c_1 \rangle$.

A Project node is used to represent a `RETURN` or `RETURN DISTINCT` statement.

Definition 5.10 (Project node)**.** Let $V = \{(v_1, E_1), (v_2, E_2), \ldots, (v_m, E_m)\}$ be a set of variable-expression pairs for some $m \in \mathbb{N}$ such that $\forall i \in \mathbb{N}, i \leq m : v_i \in \mathbb{W}$ is a variable, E_i is an AQL expression tree, and $\forall i, j \in \mathbb{N}, i, j \leq m, i \neq j : v_i \neq v_j$. Moreover, let $d \in \{\texttt{TRUE}, \texttt{FALSE}\}$ be a boolean value.
Then $n = (\texttt{PROJECT}, V, d)$ is a *Project node* such that $\mathit{type}(n) = \texttt{PROJECT}$, and $\mathit{children}(n) = \langle c_1 \rangle$.

A Limit node is used to represent a `LIMIT` condition.

Definition 5.11 (Limit node)**.** Let $\mathit{offset}, \mathit{limit} \in \mathbb{N}_0$ be two integers.
Then $n = (\texttt{LIMIT}, \mathit{offset}, \mathit{limit})$ is a *Limit node* such that $\mathit{type}(n) = \texttt{LIMIT}$ and $\mathit{children}(n) = \langle c_1 \rangle$.

A Collect node is used to represent a `COLLECT` statement. In the below definition of this node, we use the boolean value b to identify whether there is a `WITH COUNT` statement, and if there is, the variable w is used to store the count result.

Definition 5.12 (Collect node)**.** Let $V = \langle (v_1, E_1), (v_2, E_2), \ldots, (v_m, E_m) \rangle$ for some $m \in \mathbb{N}_0$ be a sequence of variable-expression pairs such that $\forall i \in \mathbb{N}, i \leq m :$ $v_i \in \mathbb{W}$ is a variable, E_i is an AQL expression tree and $\forall i, j \in \mathbb{N}, i, j \leq m, i \neq j :$ $v_i \neq v_j$. Moreover, let $b \in \{\texttt{TRUE}, \texttt{FALSE}\}$ be a boolean value, and $w \in \mathbb{W} \cup \{\bot\}$ be a variable that can be undefined.
Then $n = (\texttt{COLLECT}, V, b, w)$ is a *Collect node* such that $\mathit{type}(n) = \texttt{COLLECT}$, $\mathit{children}(n) = \langle c_1 \rangle$ and $w = \bot$ if $b = \texttt{FALSE}$, $w \in \mathbb{W}$, otherwise.

We shall now define an AQL expression tree and its nodes. AQL expression trees are used to represent filtering conditions, subqueries, arithmetic operations, function calls, and other expressions in an AQL query.

Definition 5.13 (AQL expression tree)**.** An *AQL expression tree* is a tuple containing three elements $(N, r, \mathit{children})$, where $N = \{n_1, n_2, \ldots, n_i\}$ for some $i \in \mathbb{N}$ is the set of nodes in the tree, $r \in N$ is the root node of the tree, and *children* is a function that maps every node in N to its sequence of child nodes.
Let n be a node in an AQL expression tree. Then:

$$\begin{aligned} \mathit{type}(n) \in \{&\texttt{VARIABLE}, \texttt{COLLECTION}, \texttt{VALUE}, \texttt{EXPRQUERY}, \texttt{AND}, \texttt{OR}, \texttt{NOT}, \texttt{ADD},\\ &\texttt{SUBTRACT}, \texttt{MULTIPLY}, \texttt{DIVIDE}, \texttt{EQUALS}, \texttt{NOTEQUALS}, \texttt{GREATERTHAN},\\ &\texttt{GREATERTHANEQUAL}, \texttt{LESSTHAN}, \texttt{LESSTHANEQUAL}, \texttt{UNION}, \texttt{CONCAT},\\ &\texttt{CONDITIONAL}, \texttt{ISNULL}, \texttt{LENGTH}, \texttt{LOWER}, \texttt{NOTNULL}, \texttt{TOSTRING}, \texttt{POSITION}\} \end{aligned}$$

and $\mathit{children}(n) = \langle c_1, c_2, \ldots, c_m\rangle$ for some $m \in \mathbb{N}_0$ is a sequence of child nodes of node n.

A Variable node is used to represent an AQL variable.

Definition 5.14 (Variable node)**.** Let $v \in \mathbb{W}$ be a variable name.
Then node $n = (\texttt{VARIABLE}, v)$ is a *Variable node* such that $\mathit{type}(n) = \texttt{VARIABLE}$, and $\mathit{children}(n) = \langle\rangle$.

A Collection node is used to represent the name of an ArangoDB collection used in a query.

Definition 5.15 (Collection node)**.** Let $c \in \mathbb{C}$ be a collection name. Then node $n = (\texttt{COLLECTION}, c)$ is a *Collection node* such that $\mathit{type}(n) = \texttt{COLLECTION}$, and $\mathit{children}(n) = \langle\rangle$.

A Value node is used to represent an atomic value, that is a string, number, a boolean value, or a `null` value. It can also be used to represent a constant array or object.

Definition 5.16 (Value node)**.** Let v be a JSON value, array, or object. Then $n = (\texttt{VALUE}, v)$ is a *Value node* such that $\mathit{type}(n) = \texttt{VALUE}$, and $\mathit{children}(n) = \langle\rangle$.

An Expression Query node is used to represent a subquery within an assignment or function call.

Definition 5.17 (Expression Query node)**.** Let A be an AQL query tree. Then $n = (\texttt{EXPRQUERY}, A)$ is an *Expression Query node* such that $\mathit{type}(n) = \texttt{EXPRQUERY}$, and $\mathit{children}(n) = \langle\rangle$.

An Arithmetic node is a node that represents an arithmetic operation.

Definition 5.18 (Arithmetic node)**.** Let $t \in \{\texttt{ADD}, \texttt{SUBTRACT}, \texttt{MULTIPLY}, \texttt{DIVIDE}, \texttt{GREATERTHAN}, \texttt{GREATERTHANEQUAL}, \texttt{LESSTHAN}, \texttt{LESSTHANEQUAL}, \texttt{EQUALS}, \texttt{NOTEQUALS}\}$ be a node type. Then $n = (t)$ is an *Arithmetic node* such that $\mathit{type}(n) = t$, and $\mathit{children}(n) = \langle c_1, c_2\rangle$.

Definition 5.19 (LogicalAnd node)**.** A node $n =$ (AND) is a *LogicalAnd node* such that $type(n) =$ AND, and $children(n) = \langle c_1, c_2 \rangle$.

Definition 5.20 (LogicalOr node)**.** A node $n =$ (OR) is a *LogicalOr node* such that $type(n) =$ OR, and $children(n) = \langle c_1, c_2 \rangle$.

Definition 5.21 (LogicalNot node)**.** A node $n =$ (NOT) is a *LogicalNot node* such that $type(n) =$ NOT, and $children(n) = \langle c_1 \rangle$.

A Union node is a node that represents the UNION function, which combines the elements of multiple JSON arrays into one array.

Definition 5.22 (Union node)**.** A node $n =$ (UNION) is a *Union node* such that $type(n) =$ UNION, and $children(n) = \langle c_1, c_2, \ldots, c_m \rangle$ for some $m \in \mathbb{N}, m \geq 2$, such that $\forall i \in \mathbb{N}, i \leq m : c_i$ is a constant array or an expression that returns an array.

A Concat node is a node that represents the CONCAT function, which concatenates a given sequence of JSON values, objects, and/or arrays into a string.

Definition 5.23 (Concat node)**.** A node $n =$ (CONCAT) is a *Concat node* such that $type(n) =$ CONCAT, and $children(n) = \langle c_1, c_2, \ldots, c_m \rangle$ for some $m \in \mathbb{N}$.

A Conditional node is used to represent the ternary conditional operator that takes the form of *condition* ? *consequent* : *alternative*, where *condition* is an expression which, if evaluated to true, results in the evaluation of the *consequent* expression, of which the result is returned. Otherwise, the *alternative* expression is evaluated and its result is returned.

Definition 5.24 (Conditional node)**.** A node $n =$ (CONDITIONAL) is a *Conditional node* such that $type(n) =$ CONDITIONAL, and $children(n) = \langle c_1, c_2, c_3 \rangle$.

An IsNull node is used to represent the IS_NULL type checking function, which takes one parameter value and returns a boolean value TRUE if it is null, and FALSE, otherwise.

Definition 5.25 (IsNull node)**.** A node $n =$ (ISNULL) is an *IsNull node* such that $type(n) =$ ISNULL, and $children(n) = \langle c_1 \rangle$.

A Length node is used to represent the LENGTH function, which returns the number of elements in a given array.

Definition 5.26 (Length node)**.** A node $n =$ (LENGTH) is a *Length node* such that $type(n) =$ LENGTH, and $children(n) = \langle c_1 \rangle$.

A Lower node is used to represent the LOWER function which takes a string value and returns the same value but with all upper-case characters converted to lower-case.

Definition 5.27 (Lower node)**.** A node $n = (\texttt{LOWER})$ is a *Lower node* such that $type(n) = \texttt{LOWER}$, and $children(n) = \langle c_1 \rangle$.

A NotNull node is used to represent the `NOT_NULL` function, which takes a variable number of inputs in sequence and returns the first input that is not `null`, or `null` if all the inputs are `null`.

Definition 5.28 (NotNull node)**.** A node $n = (\texttt{NOTNULL})$ is a *NotNull node* such that $type(n) = \texttt{NOTNULL}$, and $children(n) = \langle c_1, c_2, \ldots, c_m \rangle$ for some $m \in \mathbb{N}$.

A ToString node is used to represent the `TO_STRING` type casting function, which takes an input value of any type and converts it into a string value.

Definition 5.29 (ToString node)**.** A node $n = (\texttt{TOSTRING})$ is a *ToString node* such that $type(n) = \texttt{TOSTRING}$, and $children(n) = \langle c_1 \rangle$.

A Position node is used to represent the `POSITION` function, which takes an array and a search element and returns a boolean value `TRUE` if the array contains the search element, and `FALSE`, otherwise. It also takes an optional boolean parameter, and if this has a value of `TRUE`, instead of a boolean value, the function returns the position of the search term in the array, if it contains it, and -1 otherwise.

Definition 5.30 (Position node)**.** A node $n = (\texttt{POSITION})$ is a *Position node* such that $type(n) = \texttt{POSITION}$, and $children(n) = \langle c_1, c_2, c_3 \rangle$.

6. Modelling RDF data in ArangoDB

In this chapter, we propose two approaches for transforming and storing RDF data in ArangoDB. The first approach, which we call the *Basic Approach*, utilizes the key-value and document data models of storage. The second approach, called the *Graph Approach*, makes use of the graph data model in addition to the other two models.

6.1 Basic Approach

Using this approach, we transform each RDF triple into an ArangoDB document such that the subject, predicate, and object are each transformed into a JSON object, and these three JSON objects are nested within the ArangoDB document. This is called the Basic Approach as it does not make use of the graph model of ArangoDB. This storage approach is a modification of the RDF/JSON serialization, based on [71].

In this approach, all the created documents are stored in a single ArangoDB collection. During import of the transformed data into ArangoDB, the system attributes `_id`, `_key` and `_rev` are automatically generated for each document.

In Definition 6.1 below, we define how a single RDF term in a triple is transformed into a JSON object. This is required for transforming each of the subject, predicate, and object during the transformation of a triple.

Definition 6.1 (RDF Term transformation to a JSON object)**.** Let $i \in (\mathbb{I} \cup \mathbb{L} \cup \mathbb{B})$ be an RDF term. Then

$$\mathit{transform}(i) = \{(\texttt{type}, v_{\texttt{type}}), (\texttt{value}, v_{\texttt{value}}), (\texttt{datatype}, v_{\texttt{datatype}}), (\texttt{lang}, v_{\texttt{lang}}\}$$

is a JSON object describing the original RDF term i such that:

- $v_{\texttt{type}} = \texttt{IRI}, v_{\texttt{value}} = i, v_{\texttt{datatype}} = \perp, v_{\texttt{lang}} = \perp$, if and only if $i \in \mathbb{I}$
- $v_{\texttt{type}} = \texttt{BNODE}, v_{\texttt{value}} = i, v_{\texttt{datatype}} = \perp, v_{\texttt{lang}} = \perp$, if and only if $i \in \mathbb{B}$
- $v_{\texttt{type}} = \texttt{LITERAL}, v_{\texttt{value}} = \mathit{value}(i)$, $v_{\texttt{datatype}} = \mathit{datetype}(i)$, $v_{\texttt{lang}} = \mathit{lang}(i)$, if and only if $i \in \mathbb{L}$

We now define the transformation of a whole RDF triple into an ArangoDB document in Definition 6.2 below. An example of this transformation is also given in Figure 6.1, in which we assume that the generated ArangoDB document is stored in a collection called `triples`.

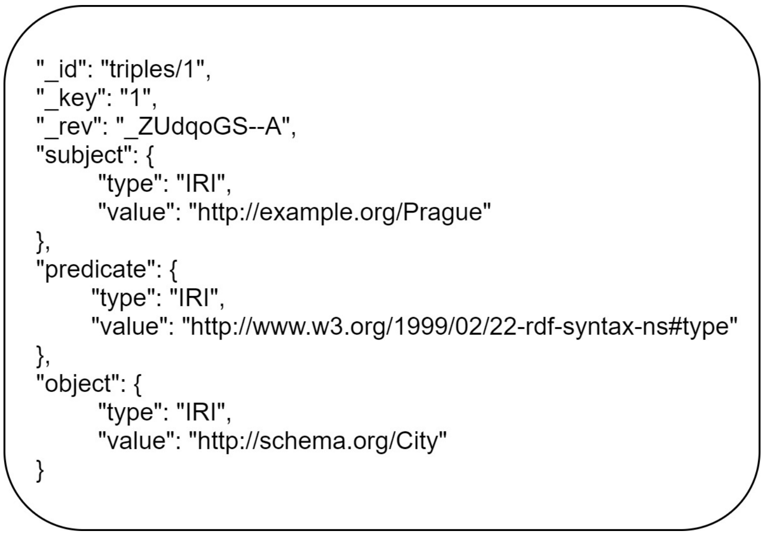
Definition 6.2 (RDF Triple transformation to single ArangoDB document)**.** Let (u, G) be an RDF graph and $t = (s, p, o) \in G$ be an RDF triple. Then t is

transformed into an ArangoDB document $\mathit{transform}(t) = O$ where
$O = \{(\texttt{\_id}, v_{id}), (\texttt{\_key}, v_{key}), (\texttt{\_rev}, v_{rev}), (\texttt{subject}, v_s), (\texttt{predicate}, v_p), (\texttt{object}, v_o), (\texttt{graph}, v_g)\}$ such that:

- v_{id}, v_{key}, v_{rev} are auto-generated by ArangoDB
- $v_s = \mathit{transform}(s)$ is a JSON object describing the original RDF subject s
- $v_p = \mathit{transform}(p)$ is a JSON object describing the original RDF predicate p
- $v_o = \mathit{transform}(o)$ is a JSON object describing the original RDF object o
- $v_g = \begin{cases} \mathit{transform}(u), & \text{if } u \in \mathbb{I} \\ \bot, & \text{otherwise} \end{cases}$

$(\langle \text{http://example.org/Prague} \rangle ,$
$\langle \text{http://www.w3.org/1999/02/22-rdf-syntax-ns\#type} \rangle ,$
$\langle \text{http://schema.org/City} \rangle)$

(a) Sample RDF triple



```
"_id": "triples/1",
"_key": "1",
"_rev": "_ZUdqoGS--A",
"subject": {
        "type": "IRI",
        "value": "http://example.org/Prague"
},
"predicate": {
       "type": "IRI",
       "value": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
},
"object": {
        "type": "IRI",
        "value": "http://schema.org/City"
}
```

(b) Generated ArangoDB document

Figure 6.1: Basic Approach - Example Transformation

We also define how to transform a whole RDF Graph and a whole RDF Dataset in Definitions 6.3 and 6.4, respectively.

Definition 6.3 (RDF Graph Transformation)**.** Let $\mathcal{G} = (u, G)$ be an RDF graph. Then $\mathit{transform}(\mathcal{G}) = \{t' \mid \exists t \in G : \mathit{transform}(t) = t'\}$ is the set of all ArangoDB documents generated for the triples in G, such that ($\texttt{triples}$, $\mathit{transform}(\mathcal{G})$) is an ArangoDB collection.

Definition 6.4 (RDF Dataset Transformation)**.** Let $\mathcal{D} = \{(\bot, G_1), (u_2, G_2), \ldots,$

$(u_n, G_n)\}$ for some $n \in \mathbb{N}$ be an *RDF Dataset.* Then

$$\mathit{transform}(\mathcal{D}) = \mathit{transform}((\bot, G_1)) \cup (\bigcup_{i=2}^{n} transform((\mu_i, G_i)))$$

is the set of all ArangoDB documents generated for the triples in $\mathcal{D}$.

In choosing this approach, we took into consideration some particular features and limitations of ArangoDB. One of the cases we wanted to cater to was that where a user wants to store and query RDF triples in different named graphs. An obvious approach would have been to have a separate ArangoDB collection for each named graph, and to store each triple in the collection that represents the named graph they form part of. The issue with this approach is that AQL does not allow you to query multiple collections at the same time. Thus, the user would have to either join the results of multiple duplicate queries run on different collections, or to use the `Union` function in AQL to first combine all the documents from the various collections into one array, and then query that single large array. Although the second option might sound ideal, one is presented with the problem that ArangoDB cannot make use of indexes on the results of a collection union. Thus, filtering the whole array of data would be particularly slow.

By using our method, we can store all triples in one collection and filter on the `graph` attribute to consider only triples in the desired named graph(s). By using a hash index on the `graph` attribute, this should be a relatively fast operation, although this depends on the number of documents in the collection. Moreover, if we want to consider just the triples in the default graph, we can simply filter and keep only the documents having `graph` undefined. However, since many RDF stores build the default graph of their default dataset by merging the default graph and all the named graphs, this structure also allows us to do the same by simply querying the whole collection.

Using this model, it is also easy to add more transformed RDF triples to the collection in the future, since it is just a matter of inserting a new document into the collection. This would not be the case if we nested the objects in the document differently.

6.2 Graph Approach

The Graph Approach is based on the second RDF storage approach for ArangoDB described in [69], i.e. the direct representation with edge values.

Using this approach, we transform each RDF triple into two vertices connected by a graph edge, i.e. we transform the subject and object into ArangoDB documents, and transform the predicate into a JSON object which we store as a property of an edge document connecting the subject and object documents.

Our goal is to have just a single ArangoDB document for each unique IRI or blank node in the RDF data. Thus, the IRI or blank node label is used to set the `_key` value for the document. Due to ArangoDB not allowing special characters in the `_key` attribute value, a simple hashing algorithm is needed to hash an IRI into a unique numeric value.

All the documents representing some resource $r \in (I \cup B)$ are stored in a single dedicated ArangoDB collection, whereas all the documents representing some literal $l \in L$ are stored in a single separate ArangoDB collection. All the edge documents representing predicate links between a subject and object of a triple are stored in a single ArangoDB edge collection.

Storing documents for IRIs and blank nodes separate from documents for literals is useful when querying the data, as well as for adding collection indexes. When we want to perform graph traversals, we can search for start vertices for a traversal in the collection containing resources, and ignore the collection containing literals, as the latter cannot be used as subjects of triples. This decreases query runtime as we iterate over fewer documents.

In Definition 6.5 below, we define how an RDF triple is transformed using the graph approach.

Definition 6.5 (RDF Triple transformation to two nodes linked by one edge). Let (u, G) be an RDF graph and $t = (s, p, o) \in G$ be an RDF triple. Then t is transformed into $transform(t) = \{S, O, P\}$ where:

- $S = \{(\texttt{\_id}, v_{id}), (\texttt{\_key}, v_{key}), (\texttt{\_rev}, v_{rev}), (\texttt{type}, v_{\texttt{type}}), (\texttt{value}, v_{\texttt{value}})\}$ is an ArangoDB document describing the original RDF subject s such that:
  - v_{id}, v_{rev} are auto-generated by ArangoDB
  - $v_{\texttt{key}} = hash(s), v_{\texttt{type}} = \texttt{IRI}, v_{\texttt{value}} = s$, if and only if $s \in \mathbb{I}$
  - $v_{\texttt{key}} = \textit{nextBNodeKey()}, v_{\texttt{type}} = \texttt{BNODE}, v_{\texttt{value}} = s$, if and only if $s \in \mathbb{B}$
- $O = \{(\texttt{\_id}, v_{id}), (\texttt{\_key}, v_{key}), (\texttt{\_rev}, v_{rev}), (\texttt{type}, v_{\texttt{type}}), (\texttt{value}, v_{\texttt{value}}), (\texttt{datatype}, v_{\texttt{datatype}}), (\texttt{lang}, v_{\texttt{lang}}\}$ is an ArangoDB document describing the original RDF object o such that:
  - v_{id}, v_{rev} are auto-generated by ArangoDB
  - $v_{\texttt{key}} = \textit{nextLiteralKey()}$
  - $v_{\texttt{type}} = \texttt{IRI}, v_{\texttt{value}} = o, v_{\texttt{datatype}} = \bot, v_{\texttt{lang}} = \bot$, if and only if $o \in \mathbb{I}$
  - $v_{\texttt{type}} = \texttt{BNODE}, v_{\texttt{value}} = o, v_{\texttt{datatype}} = \bot, v_{\texttt{lang}} = \bot$, if and only if $o \in \mathbb{B}$
  - $v_{\texttt{type}} = \texttt{LITERAL}, v_{\texttt{value}} = value(o)$, $v_{\texttt{datatype}} = datetype(o)$, $v_{\texttt{lang}} = \bot$, if and only if $o \in \mathbb{L}$ and o has no language tag
  - $v_{\texttt{type}} = \texttt{LITERAL}, v_{\texttt{value}} = value(o)$, $v_{\texttt{datatype}} = datetype(o)$, $v_{\texttt{lang}} = lang(o)$, if and only if $o \in \mathbb{L}$ and o is a language-tagged string
- $P = \{(\texttt{\_from}, S.\_id), (\texttt{\_to}, O.\_id), (\texttt{predicate}, v_p), (\texttt{graph}, v_g)\}$ is an ArangoDB edge document such that:
  - $v_p = transform(p)$ is a JSON object describing the original RDF pred-

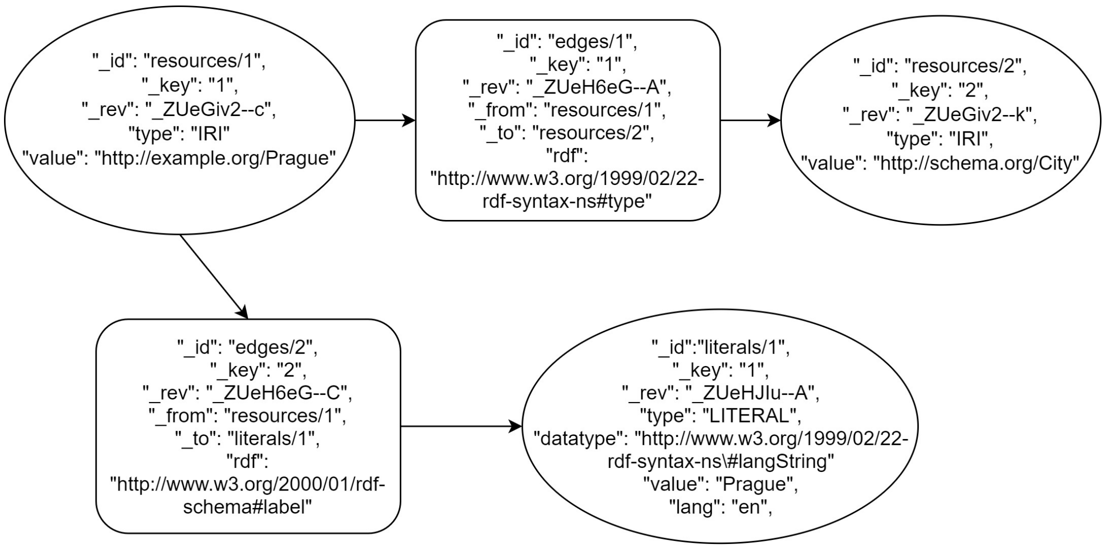
icate p

– $v_g = \begin{cases} transform(u), & \text{if } u \in \mathbb{I} \\ \bot, & \text{otherwise} \end{cases}$

where *hash* is a function that returns a unique key for an IRI, and *nextBNodeKey* and *nextLiteralKey* are functions that return a unique blank node key or unique literal key respectively, using separate incremented counters.

```
{(⟨http://example.org/Prague⟩,
        ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩,
              ⟨http://schema.org/City⟩),
 (⟨http://example.org/Prague⟩,
        ⟨http://www.w3.org/2000/01/rdf-schema#label⟩,
              "Prague"@en)}
```

(a) Sample RDF data


"_id": "resources/1", "_key": "1", "_rev": "_ZUeGiv2--c", "type": "IRI" "value": "http://example.org/Prague"
"_id": "edges/1", "_key": "1", "_rev": "_ZUeH6eG--A", "_from": "resources/1", "_to": "resources/2", "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
"_id": "resources/2", "_key": "2", "_rev": "_ZUeGiv2--k", "type": "IRI", "value": "http://schema.org/City"
"_id": "edges/2", "_key": "2", "_rev": "_ZUeH6eG--C", "_from": "resources/1", "_to": "literals/1", "rdf": "http://www.w3.org/2000/01/rdf-schema#label"
"_id":"literals/1", "_key": "1", "_rev": "_ZUeHJIu--A", "type": "LITERAL", "datatype": "http://www.w3.org/1999/02/22-rdf-syntax-ns\#langString" "value": "Prague", "lang": "en",

(b) Generated ArangoDB document

Figure 6.2: Graph Approach - Example Transformation

We also define how to transform a whole RDF Graph and a whole RDF Dataset, using the Graph Approach, in Definitions 6.6 and 6.7 respectively. A sample transformation of an RDF graph made up of two triples is also given in Figure 6.2.

Definition 6.6 (RDF Graph Transformation)**.** Let $\mathcal{G} = (u, G)$ be an RDF graph. Then $transform(\mathcal{G}) = \{\{S, O, P\} \mid \exists t \in G : transform(t) = \{S, O, P\}\}$ is the set of all ArangoDB documents generated for the triples in G, such that:

- (`vertices`, R) is an ArangoDB collection where $R = \{r \mid r \in transform(\mathcal{G}), r.\texttt{type} \in \{\texttt{IRI}, \texttt{BNODE}\}\}$

- (`literals`, L) is an ArangoDB collection where
  $L = \{l \mid l \in transform(\mathcal{G}), l.\texttt{type} = \texttt{LITERAL}\}$
- (`edges`, E) is an ArangoDB edge collection where
  $E = \{e \mid e \in transform(\mathcal{G})$ and e is an ArangoDB edge document$\}$

Definition 6.7 (RDF Dataset Transformation)**.** Let $\mathcal{D} = \{(\bot, G_1), (u_2, G_2), \ldots, (u_n, G_n)\}$ for some $n \in \mathbb{N}$ be an *RDF Dataset.* Then

$$transform(\mathcal{D}) = transform((\bot, G_1)) \cup (\bigcup_{i=2}^{n} transform((\mu_i, G_i)))$$

is the set of all ArangoDB documents generated for the triples in $\mathcal{D}$.

Other RDF storage options were also considered for the graph approach, such as using the named graph feature offered by ArangoDB to represent named graphs in RDF. However, AQL graph traversal queries cannot be performed on multiple named graphs at the same time, whereas in SPARQL, the ability to match different graph patterns in a query against different named graphs is an important feature. To simulate this in AQL, the results of multiple duplicate graph queries, executed on different named graphs, would have to be joined. This would be unintuitive and result in a large amount of duplicate code, especially if we want to query all the graphs at the same time, i.e. build our default dataset to include the default graph as well as all the named graphs.

7. Transforming SPARQL query to AQL query

In this chapter, the transformation of a SPARQL query into an AQL query is described. We present the phases making up this transformation, particularly focusing on the transformation of a SPARQL query algebra tree into an AQL query tree.

For this thesis, we only consider SPARQL queries in the `SELECT` query form, since it is the most commonly used of the four supported query forms. Nevertheless, our query transformation algorithm applies even for the other query forms, as these only differ in the way result data is presented to the user after query execution, which is not the focus of this study.

7.1 Transformation Phases

The query transformation can be divided into multiple phases. To be able to work with the SPARQL query, we first need to transform it into its algebra tree representation. This allows us to use the operations described in Chapter 4.

The next step is to transform the algebraic representation into a form from which it can be more easily converted into an AQL query expression. This will optimize the transformation process. Moreover, we will also apply other transformations to the SPARQL query algebra tree to optimize it and consequently optimize the generated AQL query expression.

At this point, we can transform the SPARQL algebra tree into the AQL query tree. The transformation goes through the algebra tree from the leaves up to the root using Postorder traversal, so we need to be able to create an AQL query subtree representing any node from the tree. During the generation of the AQL query tree, optimization options are taken into consideration and applied when appropriate. The final version of the AQL query tree is then serialized into the actual AQL query expression.

The created AQL query expression can then be executed against the data in ArangoDB. The final phase is to transform the query result into the format that is expected for the `SELECT` query form.

7.2 Modifying the SPARQL algebra tree

In this section, we propose several methods that can be used to optimize the query transformation process using some transformations on a given SPARQL query algebra tree. This type of optimization is used to simplify the algebra tree in preparation for its translation to AQL, thus making it easier to translate.

From this point onwards in the thesis, we assume that each tree node mentioned is unique, that is no two nodes are the same even if they seem identical, that is they are of the same type and store the same data. When actually implemented, each node would be associated with a unique identifier ensuring this uniqueness.

One optimization is that when a Graph node is encountered, we merge it into each of the BGP nodes found in its subtree. This is done by modifying each of these BGP nodes, after which the Graph node is completely removed from the tree. A BGP node is modified such that it stores the set of IRIs of named graphs, as well as the variable to which we have to bind a graph IRI, in case it is specified in the Graph node. Storing this information at each BGP node is useful for transforming the given SPARQL algebra tree into an AQL query tree. This is because it makes it easier to add filters on the graph attribute of our ArangoDB documents when matching triple patterns in the AQL query, as well as easier to bind the graph variable if given.

Definition 7.1 (Transformation of a Graph node)**.** Let $T = (N, r, \mathit{children})$ be a SPARQL query algebra tree, $n = (\texttt{GRAPH}, u, v) \in N$ be a Graph node in the tree, and $p \in N$ be a node in the tree such that n is a child node of p. Moreover, let $G = \{g_1, g_2, \ldots, g_j\}$ for some $j \in \mathbb{N}$ be a set of named graph IRIs such that $\forall l \in \mathbb{N}, l \leq j : g_l \in \mathbb{I}$.

We first find the set of all `BGP` nodes in the subtree of n as following:

$$\begin{aligned} Y = \{m \mid\ & i \in \mathbb{N}, \langle m_1, m_2, \ldots, m_i\rangle \text{ is a sequence such that} \\ & \forall w \in \mathbb{N}, w \leq i : m_w \in N, m = m_1, \mathit{type}(m_1) = \texttt{BGP}, m_i = n, \\ & \forall k \in \mathbb{N}, k < i : m_k \in \mathit{children}(m_{k+1}), \mathit{type}(m_k) \neq \texttt{GRAPH} \end{aligned}$$

We then update each of the `BGP` nodes in Y as following:

$$\forall x \in Y, x = (\texttt{BGP}, P_x, G_x, v_x) : x \text{ is modified to } (\texttt{BGP}, P_x, G'_x, v) \text{ such that}$$

$$G'_x = \begin{cases} G, & \text{if } u = \bot \\ \{u\}, & \text{otherwise} \end{cases}$$

If the subtree of n contains a `FILTER (NOT)EXISTS` condition, the same has to be done for the `BGP` nodes within the `EXISTS` or `NOT EXISTS` graph pattern tree as below:

$$\begin{aligned} Z = \{o \mid\ & i \in \mathbb{N}, \langle o_1, o_2, \ldots, o_i\rangle \text{ is a sequence such that} \\ & \forall w \in \mathbb{N}, w \leq i : o_w \in N, o = o_1, o_1 = (\texttt{FILTER}, (N_e, r_e, \mathit{children}_e)), \\ & \mathit{type}(r_e) \in \{\texttt{EXISTS}, \texttt{NOTEXISTS}\}, o_i = n, \text{ and} \\ & \forall k \in \mathbb{N}, k < i : o_k \in \mathit{children}(o_{k+1}), \mathit{type}(o_k) \neq \texttt{GRAPH} \end{aligned}$$

$$\begin{aligned} H = \{m \mid\ & i \in \mathbb{N}, \langle m_1, m_2, \ldots, m_i\rangle \text{ is a sequence such that} \\ & \forall w \in \mathbb{N}, w \leq i : m_w \in N : m = m_1, \mathit{type}(m_1) = \texttt{BGP}, m_i = r_e, \\ & \forall k \in \mathbb{N}, k < i : m_k \in \mathit{children}(m_{k+1}), \mathit{type}(m_k) \neq \texttt{GRAPH} \end{aligned}$$

$$\forall h \in H, h = (\texttt{BGP}, P_h, G_h, v_h) : h \text{ is modified to } (\texttt{BGP}, P_h, G'_h, v) \text{ such that}$$

$$G'_h = \begin{cases} G, & \text{if } u = \bot \\ \{u\}, & \text{otherwise} \end{cases}$$

Finally, we can say that $T \to T'$ such that $T' = (N', r, \mathit{children}')$ as following.

$$N' = N \setminus \{n\}$$

Assuming that $\mathit{children}(n) = \langle c_n \rangle$ and $\mathit{children}(p) = \langle c_1, c_2, \ldots, c_g, \ldots, c_q \rangle$ for some $q \in \mathbb{N}$ such that $\exists g \in \mathbb{N}, g \leq q : c_g = n$, then

$$\mathit{children}'(p) = \langle c_1, c_2, \ldots, c_u, \ldots, c_q \rangle$$

such that $u = g$ and $c_u = c_n$.

$$\mathit{children}' = (\mathit{children} \setminus \{(p, \mathit{children}(p))\}) \cup \{(p, \mathit{children}'(p))\}$$

Another optimization is that when a Slice node is encountered, given that its child node is a Project node, we move the Slice node over the child of the Project node, that is we swap the positions of the Slice and Project nodes so that the Project node is the node closest to the root. This is due to projection always being the final operation in AQL.

Definition 7.2 (Transformation of a Slice node with a Project child node)**.** Let $T = (N, r, \mathit{children})$ be a SPARQL query algebra tree, $n \in N$ be a node such that $\mathit{type}(n) = \texttt{SLICE}$ and $\mathit{children}(n) = \langle x \rangle$ and $\mathit{type}(x) = \texttt{PROJECT}$. Then $T \to T'$ such that $T' = (N, r', \mathit{children}')$ as following:

$$r' = \begin{cases} x, & \text{if } r = n \\ r, & \text{otherwise} \end{cases}$$

$$\mathit{children}'(n) = \mathit{children}(x)$$

$$\mathit{children}'(x) = \langle n \rangle$$

$$\text{and if } \exists y \in N, \mathit{children}(y) = \langle n \rangle : \mathit{children}'(y) = \langle x \rangle$$

$$\begin{aligned} \mathit{children}' = (&\mathit{children} \setminus \{(n, \mathit{children}(n)), (x, \mathit{children}(x)), (y, \mathit{children}(y))\}) \\ &\cup \{(n, \mathit{children}'(n)), (x, \mathit{children}'(x)), (y, \mathit{children}'(y))\} \end{aligned}$$

7.3 Variable Binders

Once we have obtained the optimized SPARQL algebra tree, the next step is to transform it into the corresponding AQL query tree and consequently obtain the AQL query expression. We shall first introduce the concept of *Variable Binders*, their structure, and their role in the transformation of a SPARQL algebra tree into an equivalent AQL query tree.

While generating each AQL query and subquery, we have to keep track of which AQL variable(s) represent which SPARQL variable. This way we are able to construct correct AQL expressions equivalent to SPARQL expressions, and project the correct bound values at the end of the AQL query. To store this information, we use what we call variable binders, which are only slightly similar to the value binders used in [55]. Our variable binders are used solely during the construction of the AQL query tree as required.

Definition 7.3 (Variable Binder). Let $v \in V$ be a SPARQL variable, $o \in \{\texttt{TRUE}, \texttt{FALSE}\}$ be a boolean value, and $A = \langle a_1, a_2, \dots, a_n \rangle$ for some $n \in \mathbb{N}$ be a sequence of AQL variable names. Then $b = (v, o, A)$ is a *variable binder.*

For each SPARQL algebra tree node that we process as we go from the leaves to the root, we keep a set of variable binders, containing one binder for each SPARQL variable in the scope of that node. A variable binder for some SPARQL variable can hold the names of multiple AQL variables or document (sub)attributes.

A variable binder only contains the name of one AQL variable in case we know the SPARQL variable it represents is definitely bound. However, a SPARQL variable could be unbound if it is contained in one or more `OPTIONAL` clauses, or if it is assigned an `UNDEF` value within a `VALUES` table. In order to be able to handle these cases, we keep a boolean value in each variable binder to indicate whether the SPARQL variable it represents can be unbound. If it is set to true, we can add `OR IS NULL` filter conditions on the AQL query variables of the variable binder during join operations. If a SPARQL variable can be bound by one of multiple `OPTIONAL` clauses and thus can possibly also remain unbound, the variable binder contains a sequence of AQL variables and the `NOT_NULL` function in AQL can be used to return the first non-null AQL variable from that list, or null if all the alternatives are null. The same applies when we are applying a join operation involving a `VALUES` table containing `UNDEF` values.

To create a set of variable binders for a particular tree node being processed, we have to make use of the sets of variable binders of its processed child nodes, unless it is a leaf node. For this reason, we need to be able to merge sets of variable binders as defined below.

Definition 7.4 (Merging sets of variable binders). Let $P = \langle B_1, B_2 \rangle$ be a sequence of two sets of variable binders such that $B_1 = \{b_1, b_2, \dots, b_i\}$ for some $i \in \mathbb{N}_0$ and $B_2 = \{d_1, d_2, \dots, d_j\}$ for some $j \in \mathbb{N}_0$. Then $\mathit{merge}(P) = B$ is a new set of variable binders such that:

$$
\begin{aligned}
B = \{g \mid k, l \in \mathbb{N}, k \leq i, l \leq j : b_k = (v_k, o_k, A_k) \in B_1, d_l = (v_l, o_l, A_l) \in B_2 \\
\text{such that } v_k = v_l \text{ and } g = \begin{cases} (v_k, o_k, A_k), & \text{if } o_k = \texttt{FALSE} \\ (v_k, o_l, A_k \cdot A_l), & \text{otherwise} \end{cases} \} \\
\cup \{b_k \mid b_k = (v_k, o_k, A_k) \in B_1, \nexists d_l = (v_l, o_l, A_l) \in B_2 : v_k = v_l\} \\
\cup \{d_l \mid d_l = (v_l, o_l, A_l) \in B_2, \nexists b_k = (v_k, o_k, A_k) \in B_1 : v_l = v_k\}
\end{aligned}
$$

where $A_k \cdot A_l$ means that the elements in sequence A_k are appended to the end of the sequence A_l.

Since we need to use the AQL variable name(s) stored in a variable binder within our AQL query expression, we need a way to transform each AQL variable name into an AQL query tree node of `VARIABLE` type. Moreover, if the variable binder contains more than one AQL variable name, then we need to add a NotNull node as a parent over all the created Variable nodes.

Definition 7.5 (Transformation of a variable binder to an AQL expression tree)**.** Let $b = (v, o, A)$ be a variable binder such that $A = \langle a_1, a_2, \ldots, a_k\rangle$ for some $k \in \mathbb{N}$.

Let us transform each variable name in sequence A into a `VARIABLE` node as following:

$$N_1 = \{n_i \mid i \in \mathbb{N}, i \leq k : n_i = (\texttt{VARIABLE}, a_i)\}$$

Then we can say that $\mathit{transform}(b) = (N, r, \mathit{children})$ is an AQL expression tree as following.

If k = 1:

$$N = N_1$$

$$r = n_1$$

$$\mathit{children} = \{(n_1, \langle\rangle)\}$$

Otherwise:

$$N = N_1 \cup \{n_r\}$$

$$n_r = (\texttt{NOTNULL}), \mathit{children}(n_r) = \langle n_1, n_2, \ldots, n_k\rangle$$

$$r = n_r$$

$$\mathit{children} = \{(n_r, \mathit{children}(n_r)), (n_1, \langle\rangle), (n_2, \langle\rangle), \ldots, (n_k, \langle\rangle)\}$$

Moreover, when we need to project results, we need to transform the variable binder of each SPARQL variable that has to be projected, into an AQL expression tree using the above definition. This gives us a variable-tree pair for each SPARQL variable. We define a function to obtain these pairs in the definition below.

Definition 7.6 (Generation of AQL project expressions from variable binders)**.** Let $P = \{s_1, s_2, \ldots, s_n\}$ for some $n \in \mathbb{N}$ be a set of SPARQL variables that can be undefined, and $B = \{b_1, b_2, \ldots, b_m\}$ for some $m \in \mathbb{N}$ be a set of variable binders. Then

$$\mathit{getProjectVarExprs}(P, B) = \begin{cases} \{(v_i, \mathit{transform}(b_i)) \mid & i \in \mathbb{N}, i \leq m : \\ & b_i = (v_i, o_i, A_i) \in B\}, \\ & \text{if } P = \bot \\ \{(v_k, \mathit{transform}(b_k)) \mid & j \in \mathbb{N}, j \leq n : s_j \in P \\ & \text{and } \exists k \in \mathbb{N}, k \leq m : \\ & b_k = (v_k, o_k, A_k) \in B, \\ & s_j = v_k\}, \\ & \text{otherwise} \end{cases}$$

To perform Join, LeftJoin, and Minus operations, we need to generate equality filters to match the bound value of variables that are present in both left and right operands. For this reason, we provide the below definition for generating the equality filters.

Definition 7.7 (Generation of Equality Filter conditions for common bound variables)**.** Let $B = \{b_1, b_2, \dots, b_i\}$ for some $i \in \mathbb{N}_0$ and $D = \{d_1, d_2, \dots, d_j\}$ for some $j \in \mathbb{N}_0$ be two sets of variable binders.

Firstly, let $C = \{c_1, c_2, \dots, c_m\}$ for some $m \in \mathbb{N}$ be the set of SPARQL variables bound in both B and D, such that $\forall o \in \mathbb{N}, o \leq m : \exists k, l \in \mathbb{N}, k \leq i, l \leq j : b_k = (c_k, o_k, A_k) \in B, d_l = (c_l, o_l, A_l) \in D, c_k = c_l = c_o$.

If C = {}, we can say that $filters(B, D) = T$ such that T = (N, r, children) as following:

$$N = \{n_b\}$$

$$n_b = (\texttt{VALUE}, \texttt{TRUE})$$

$$r = n_b$$

$$children = \{(n_b, \langle\rangle)\}$$

Otherwise, let $X = \{T_1^x, T_2^x, \dots, T_m^x\}$, $Y = \{T_1^y, T_2^y, \dots, T_m^y\}$ be two sets of AQL expression trees such that $\forall o \in \mathbb{N}, o \leq m$:

$$T_o^x = transform(b_k) = (N_o^x, r_o^x, children_o^x)$$

$$T_o^y = transform(d_l) = (N_o^y, r_o^y, children_o^y)$$

Then $Z = \{T_1^z, T_2^z, \dots, T_m^z\}$ is a set of expression trees such that $\forall o \in \mathbb{N}, o \leq m$: $T_o^z = (N_o^z, r_o^z, children_o^z)$ represents an equality condition for variable $c_o \in C$ as following:

$$N_o^z = N_o^x \cup N_o^y \cup \{n_o^z\}$$

$$n_o^z = (\texttt{EQUALS})$$

$$r_o^z = n_o^z$$

$$children_o^z = children_o^x \cup children_o^y \cup \{(n_o^z, \langle r_o^x, r_o^y \rangle)\}$$

Let $n_0 = r_1^z$. Then we can say that $filters(B, D) = T$ such that T = (N, r, children) as below:

$$\forall q \in \mathbb{N}, q < m : n_q = (\texttt{AND}), children(n_q) = \langle n_{q-1}, r_{q+1}^z \rangle$$

$$N = N_1^z \cup N_2^z \cup \dots \cup N_m^z \cup \{n_1, n_2, \dots, n_{m-1}\}$$

$$r = n_{m-1}$$

$$children = children_1^z \cup children_2^z \cup \dots \cup children_m^z \cup \{(n_1, children(n_1)), (n_2, children(n_2)), \dots, (n_{m-1}, children(n_{m-1}))\}$$

In Definition 7.7 above, we did not take into consideration two special cases that can be encountered when querying data stored using the graph approach. These are when a SPARQL query contains triple patterns that use a predicate variable in one triple pattern as the subject of another, or when a variable used to bind a graph name in a graph graph pattern is also used within one or more triple patterns. The problem with these cases is that in the graph approach, we store each RDF term as an ArangoDB document, except for an RDF term in the predicate position of a triple, and the IRI of the named graph that a triple is in, which are each represented as a JSON object nested in an edge document. In this case, we may end up generating a filter condition that checks that an ArangoDB document is equal to a JSON object. These two cannot be directly compared for equality because the ArangoDB document contains the extra `_id`, `_key`, and `_rev` attributes which a JSON object on an edge does not. Thus, they would be seen as unequal during query execution, even if they both represent the same RDF term. For this reason, in practice we have to separately compare the `type`, `value`, `datatype` and `lang` properties of the ArangoDB document to the corresponding property in the JSON object. This means that instead of one, we need to introduce four equality conditions that must all be satisfied.

7.4 Generating the AQL query expression

We will now describe and formally define the transformation of an optimized SPARQL query algebra tree into an AQL query tree. We define how each type of SPARQL tree node in a SPARQL query algebra tree or expression tree is transformed into an equivalent AQL node, AQL query tree, or AQL expression tree. As mentioned previously, every node in a given SPARQL algebra tree or SPARQL expression tree is transformed only after its child nodes have been transformed.

We also provide some sample SPARQL query expressions and equivalent AQL query expressions for them based on our defined transformations. For the majority of the examples, unless indicated otherwise, the AQL query expressions given are appropriate when querying RDF data transformed and stored using our basic approach, and we assume that `triples` is the name of the ArangoDB collection storing our transformed RDF data.

For use during the generation of the AQL query, we introduce an arbitrary function called *newVarName*, which returns a unique variable name $w \in \mathbb{W}$ ie. a variable that has not yet been used in the AQL query tree being built.

For the following definitions, we do not consider the possibility of having sub-selects within SPARQL queries, i.e. we assume there is only one `PROJECT` node in the tree. This is only because having nested `PROJECT` nodes would considerably complicate the following formal definitions as this would require extra checks and processing. Nevertheless, this does not mean that they cannot be supported. The definitions can be extended to allow sub-selects.

We first define the transformation of a given SPARQL query algebra tree and its nodes. In some of the definitions, we refer to the transformation of a SPARQL expression tree. The definition of this tree transformation and the transformation of its nodes are given later in Section 7.4.13.

Definition 7.8 (Transformation of a SPARQL query algebra tree)**.** Let $T = (N, r, \mathit{children})$ be a SPARQL query algebra tree.
Then $\mathit{transform}(T) = \mathit{transform}(r) = (T_r, B_r)$ such that T_r is the generated AQL query tree representing the final AQL query expression, and B_r is a set of variable binders.

7.4.1 BGP node

A BGP node is transformed differently depending on whether the RDF data we are querying was transformed and stored using the basic approach or the graph approach defined in Chapter 6. Thus, both approaches are described below.

Basic Approach

When working with the basic approach, for each triple pattern in a BGP, we create a `FOR` loop over the ArangoDB collection containing our transformed RDF data. Since the result of a BGP operation is essentially the result of joining the solutions of all given triple patterns, each `FOR` loop will be nested within that of the triple pattern before it. Appropriate `FILTER` clauses are added to each `FOR` loop to match the RDF terms in the triple pattern, as well as to simulate join conditions between the results of the current triple pattern and the results obtained from (joining) the results of previously processed triple patterns in the BGP.

We know that each triple pattern contains a subject, predicate, and object, and each of these can either be a variable, a resource (i.e. an IRI or literal), or a blank node. We will process the subject, the predicate and the object of a triple pattern separately, depending on their type, as described below:

- Variable - If the variable was not already present and bound in a previously processed triple pattern, a variable binder for it has to be added. Otherwise, a `FILTER` condition must be added over the `FOR` loop, to make sure that only triples containing the same value as the AQL variable in the existing variable binder, in the corresponding position, are kept.
- Resource - A `FILTER` condition must be added over the FOR loop to make sure that only triples containing the given IRI or literal value in the corresponding position are kept.
- Blank node - This is treated exactly the same way we treat a variable, because in the SPARQL language, blank nodes are quite similar to standard variables. The only two differences are that the same blank node label cannot be used in two different basic graph patterns in the same query, and a blank node cannot be projected. This is ensured when a SPARQL

query is loaded from file and validated, and thus does not affect the query transformation itself.

A BGP node can optionally also contain a set of named graph IRIs and/or a SPARQL variable to which the IRI of the named graph that triples matching the triple patterns form part of, must be bound. If both of these are not provided, then no further processing needs to be done.

If the SPARQL variable is present, then so is the set of named graph IRIs, since we have to make sure that the triple patterns only match RDF triples that are within one of the given named graphs. To ensure this, a `FILTER` condition has to be added to the `FOR` loop of the first processed triple pattern, the condition being that the value of the `graph` attribute of matched triples must be one of the given named graph IRIs. Thus, the variable will be bound by the first triple pattern and a variable binder is created for it. All consecutive triple patterns will match triples that are within the graph bound to that variable, which we ensure by adding an equality `FILTER` condition in the same way we do when we encounter a variable in the subject, predicate, or object position of a triple pattern.

If the SPARQL variable is not present, the node may still contain a set of one or more named graph IRIs. This can be due to having `FROM` clauses in the query or due to there being a `GRAPH` clause that contained an IRI, since we merge Graph nodes into BGP nodes during the optimization stage of query transformation. In this case, we again add a `FILTER` condition to the `FOR` loop of the first processed triple pattern, however, since there will not be a variable binder for the graph IRI, we must pass on the AQL variable name of that first `FOR` loop to consecutive processed triple patterns. This way, we will again add a `FILTER` condition on each `FOR` loop to match triples that are in the same graph. This is done by comparing the `graph` attribute of possibly matching documents with the `graph` attribute of the ArangoDB document represented by that AQL variable.

An example SPARQL query containing a simple basic graph pattern, together with its equivalent AQL query is given in Figure 7.1. Moreover, we formally define the transformation of a BGP node, a triple pattern in a BGP node, and a single triple pattern component in Definitions 7.9, 7.10, and 7.11 below.

Definition 7.9 (Transformation of a triple pattern component)**.** Let $x \in \mathbb{T} \cup \mathbb{V}$ be a triple pattern component. Moreover, let $B = \{b_1, b_2, \dots, b_i\}$ for some $i \in \mathbb{N}_0$ be a set of variable binders, $w \in \mathbb{W}$ be an AQL variable, and Z be a set of expression trees representing filter conditions.

Then $\textit{transform}(x, w, B, Z) = (B', Z')$ as below.

- If $x \in \mathbb{I} \cup \mathbb{L} : Z' = Z \cup \{T_x\}$ and $B' = B$
- If $x \in \mathbb{B} \cup \mathbb{V}$:
  - if $\exists k \in \mathbb{N}, k \leq i : b_k = (v_k, o_k, \langle a_k \rangle) \in B, v_k = x$ then $Z' = Z \cup \{T_x\}$ and $B' = B$
  - otherwise: $B' = B \cup \{(x, \texttt{FALSE}, \langle w \rangle)\}$ and $Z' = Z$

such that:

$$T_x = (N_x, r_x, \textit{children}_x)$$

$$N_x = \{n_e, n_v, n_t\}$$

$$n_e = (\texttt{EQUALS})$$

$$n_v = (\texttt{VARIABLE}, w)$$

$$n_t = \begin{cases} (\texttt{VALUE}, \mathit{transform}(x)), & \text{if } x \in \mathbb{I} \cup \mathbb{L} \\ (\texttt{VARIABLE}, a_k), & \text{if } x \in \mathbb{B} \cup \mathbb{V} \end{cases}$$

$$r_x = n_e$$

$$\mathit{children}_x = \{(n_e, \langle n_v, n_t \rangle), (n_v, \langle\rangle), (n_t, \langle\rangle)\}$$

Definition 7.10 (Transformation of a triple pattern)**.** Let $t = (s, p, o)$ be a triple pattern, $B = \{b_1, b_2, \ldots, b_i\}$ for some $i \in \mathbb{N}_0$ be a set of variable binders, $G = \{g_1, g_2, \ldots, g_j\}$ for some $j \in \mathbb{N}$ be a set of named graph IRIs which can be undefined, and $v \in V \cup \{\perp\}$ be a SPARQL variable which can be undefined. Moreover, let $a \in \mathbb{W} \cup \{\perp\}$ be an AQL variable which can be undefined.

Firstly, we create an Iteration node that loops over our collection of transformed RDF data as below:

$$n_w = (\texttt{ITERATION}, w, T_1)$$

$$w = \mathit{newVarName}()$$

$$T_1 = (N_1, r_1, \mathit{children}_1)$$

$$N_1 = \{n_v\}$$

$$n_v = (\texttt{COLLECTION}, \texttt{triples})$$

$$r_1 = n_v$$

$$\mathit{children}_1 = \{(n_v, \langle\rangle)\}$$

We must then process s, p, o. Assuming $Z = \{\}$ is an empty set of expression trees, we perform the processing as following:

$$\mathit{transform}(s, w.\mathit{subject}, B, Z) = (B_1, Z_1)$$

$$\mathit{transform}(p, w.\mathit{predicate}, B_1, Z_1) = (B_2, Z_2)$$

$$\mathit{transform}(o, w.\mathit{object}, B_2, Z_2) = (B_3, Z_3)$$

We now take named graphs into consideration if applicable. We perform the below processing to obtain B_4, Z_4 and a'.

- If $a = \perp$:
  - if $v = \perp$:
    - if $G = \perp : B_4 = B_3, Z_4 = Z_3, a' = a$
    - if $G \neq \perp : B_4 = B_3, Z_4 = Z_3 \cup \{T_G\}$, $a' = w.\mathit{graph}$
  - otherwise:
    - if $\exists k \in \mathbb{N}, k \leq i : b_k = (v_k, o_k, \langle a_k \rangle) \in B, v_k = v$ then $B_4 = B_3$, $Z_4 = Z_3 \cup \{T_v\}, a' = a$
    - otherwise: $B_4 = B_3 \cup \{(v, \texttt{FALSE}, \langle w.\mathit{graph} \rangle)\}$, $Z_4 = Z_3 \cup \{T_G\}$, $a' = a$
- otherwise: $B_4 = B_3, Z_4 = Z_3 \cup \{T_a\}, a' = a$

such that T_G represents a $\texttt{POSITION}(G, w.graph.value, \texttt{FALSE}) = \texttt{TRUE}$ condition, T_v represents a $w.graph = a_k$ condition, and T_a represents a $w.graph = a$ condition.

Assume that $Z_4 = \{T_1^z, T_2^z, \dots T_l^z\}$ for some $l \in \mathbb{N}_0$ after processing, such that $\forall m \in \mathbb{N}, m \leq l : T_m^z = (N_m^z, r_m^z, children_m^z)$. Then we must join the expression trees in Z_4 into a single expression tree $E = (N_e, r_e, children_e)$ as follows.

If $l = 0$:

$$N_e = \{n_0\}$$

$$n_0 = (\texttt{VALUE}, \texttt{TRUE})$$

$$r_e = n_0$$

$$children_e = \{(n_0, \langle\rangle)\}$$

Otherwise, assuming $n_0 = r_1^z$:

$$\forall m \in \mathbb{N}, m < l : n_m = (\texttt{AND}), children_e(n_m) = \langle n_{m-1}, r_{m+1}^z \rangle$$

$$N_e = N_1^z \cup N_2^z \cup \dots \cup N_l^z \cup \{n_1, n_2, \dots, n_{l-1}\}$$

$$r_e = n_{l-1}$$

$$\begin{aligned} children_e = & children_1^z \cup children_2^z \cup \dots \cup children_l^z \cup \{(n_1, children_e(n_1)), \\ & (n_2, children_e(n_2)), \dots, (n_{l-1}, children_e(n_{l-1}))\} \end{aligned}$$

Then we can say that $transform(t, B, G, v, a) = (T, B', a')$ such that $T = (N, r, children)$ is an AQL query tree and B' is a set of variable binders as following:

$$N = \{n_w, n_f\}$$

$$n_f = (\texttt{FILTER}, E)$$

$$r = n_f$$

$$children = \{(n_f, \langle n_w \rangle), (n_w, \langle\rangle)\}$$

Definition 7.11 (Transformation of a BGP node)**.** Let $n = (\texttt{BGP}, P, G, v)$ be a BGP node in a SPARQL query algebra tree such that $P = \{t_1, t_2, \dots, t_k\}$ for some $k \in \mathbb{N}_0$. Moreover, let $B_0 = \{\}$ be an empty set of variable binders and $a_0 = \bot$ be an undefined AQL variable.

We first transform each triple pattern in P as follows:

$$\forall i \in \mathbb{N}, i \leq k : transform(t_i, B_{i-1}, G, v, a_{i-1}) = (T_i, B_i, a_i)$$

$$T_i = (N_i, r_i, children_i)$$

Let $n_0 = r_1$. Then we can say that $transform(n) = (T, B)$ such that $T = (N, r, children)$ is an AQL query tree and B is a set of variable binders as follows:

$$\forall j \in \mathbb{N}, j < k : n_j = (\texttt{NEST}),\ children(n_j) = \langle n_{j-1}, r_{j+1} \rangle$$

$$N = N_1 \cup N_2 \cup \dots \cup N_k \cup \{n_1, n_2, \dots, n_{k-1}\}$$

$$r = n_{k-1}$$

$$\begin{aligned} children = & children_1 \cup children_2 \cup \dots \cup children_k \\ & \cup \{(n_1, children(n_1)), (n_2, children(n_2)), \dots, (n_{k-1}, children(n_{k-1}))\} \end{aligned}$$

$$B = B_k$$

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

SELECT ?p ?firstName ?lastName
WHERE  {
  ?p rdf:type foaf:Person ;
     foaf:firstName ?firstName ;
     foaf:lastName ?lastName .
}
```

(a) SPARQL query

```
FOR forloop1item IN triples
  FILTER (forloop1item.predicate.value=="http://www.w3.org
     /1999/02/22-rdf-syntax-ns#type")
         && (forloop1item.object.type=="IRI")
         && (forloop1item.object.value=="foaf:Person")
  FOR forloop2item IN triples
    FILTER (forloop2item.s==forloop1item.s)
           && (forloop2item.p.value=="http://xmlns.com/foaf
              /0.1/firstName")
    FOR forloop3item IN triples
      FILTER (forloop3item.s==forloop1item.s)
             && (forloop3item.p.value=="http://xmlns.com/
                foaf/0.1/lastName")
        RETURN { p: forloop1item.s,
                 firstName: forloop2item.o,
                 lastName: forloop3item.o }
```

(b) Equivalent AQL query

Figure 7.1: Sample transformation of a SPARQL query with a single BGP

Graph Approach

In this case, for each triple pattern in a BGP we first have to find all the possible subjects for the pattern, and then perform a graph traversal for each subject, that is iterate over all the ArangoDB documents representing RDF resources, keeping all the ones that match the subject, and perform a graph traversal using the current subject document as the start vertex. A variable, resource, or blank node in the triple pattern is handled similar to the way described for the basic approach. However, in this case, `FILTER` conditions are not only applied over regular collection documents but also over paths found in the graph traversal.

In Figure 7.2, we give an equivalent AQL query expression for the same SPARQL query expression shown in Figure 7.1a, however this AQL query expression is for querying data stored using the graph approach. For this example, we assume that the ArangoDB documents representing transformed RDF resources are stored in a collection called `vertices_resources`, and edge documents are stored in a collection called `edges`.

```
FOR forloop1item IN vertices_resources
  FOR g_v1, g_e1, g_p1 IN 1..1 OUTBOUND forloop1item edges
    FILTER (g_p1.edges[0].p.value=="http://www.w3.org
        /1999/02/22-rdf-syntax-ns#type")
        && (g_p1.vertices[1].type=="IRI")
        && (g_p1.vertices[1].value=="http://xmlns.com/foaf
            /0.1/Person")
    FOR g_v2, g_e2, g_p2 IN 1..1 OUTBOUND forloop1item
        edges
      FILTER (g_p2.edges[0].p.value=="http://xmlns.com/foaf
          /0.1/firstName")
      FOR g_v3, g_e3, g_p3 IN 1..1 OUTBOUND forloop1item
          edges
        FILTER (g_p3.edges[0].p.value=="http://xmlns.com/
            foaf/0.1/lastName")
        RETURN { p: forloop1item,
                 firstName: g_p2.vertices[1],
                 lastName: g_p3.vertices[1]}
```

Figure 7.2: Equivalent AQL query for the SPARQL query given in Figure 7.1a, based on the graph approach of data storage

Definition 7.11, defining the transformation of a BGP node when dealing with the basic approach, applies for the graph approach as well. Definition 7.9 also applies for this approach. However, triple patterns in a BGP must be transformed differently. Thus, we provide Definition 7.12 to be used instead of Definition 7.10 only in case of the graph approach.

Definition 7.12 (Transformation of a triple pattern – Graph Approach)**.** Let $t = (s, p, o)$ be a triple pattern, $B = \{b_1, b_2, \dots, b_i\}$ for some $i \in \mathbb{N}_0$ be a set of variable binders, $G = \{g_1, g_2, \dots, g_j\}$ for some $j \in \mathbb{N}$ be a set of named graph IRIs which can be undefined, and $v \in V \cup \{\perp\}$ be a SPARQL variable which can be undefined. Moreover, let $a \in \mathbb{W} \cup \{\perp\}$ be an AQL variable which can be undefined.

Firstly, we create an Iteration node that loops over our collection of transformed RDF resources to find all possible resources matching the subject of the triple pattern.

$$n_s = (\texttt{ITERATION}, w, T_s)$$

$$w = newVarName()$$

$$T_s = (N_1, r_1, children_1)$$

$$N_1 = \{n_v\}$$

$$n_v = (\texttt{COLLECTION}, \texttt{vertices\_resources})$$

$$r_1 = n_v$$
$$\mathit{children}_1 = \{(n_v, \langle\rangle)\}$$

Assuming $Z_0 = \{\}$ is an empty set of expression trees, we then process s as following:

$$\mathit{transform}(s, w, B, Z_0) = (B', Z'_0)$$

Assume that $Z'_0 = \{Q_1, Q_2, \dots, Q_k\}$ for some $k \in \mathbb{N}_0$ such that $\forall m \in \mathbb{N}, m \le k$: $Q_m = (N^q_m, r^q_m, \mathit{children}^q_m)$.

Then we must join the expression trees in Z'_0 into a single expression tree $E_1 = (N^1_e, r^1_e, \mathit{children}^1_e)$ as follows.

If $k = 0$:

$$N^1_e = \{n^1_0\}$$
$$n^1_0 = (\texttt{VALUE}, \texttt{TRUE})$$
$$r^1_e = n^1_0$$
$$\mathit{children}^1_e = \{(n^1_0, \langle\rangle)\}$$

Otherwise, assuming $n_0 = r^q_1$:

$$\forall m \in \mathbb{N}, m < k : n_m = (\texttt{AND}), \mathit{children}^1_e(n_m) = \langle n_{m-1}, r^q_{m+1}\rangle$$

$$N^1_e = N^q_1 \cup N^q_2 \cup \dots \cup N^q_k \cup \{n_1, n_2, \dots, n_{k-1}\}$$
$$r^1_e = n_{k-1}$$

$$\begin{aligned}\mathit{children}^1_e = {} & \mathit{children}^q_1 \cup \mathit{children}^q_2 \cup \dots \cup \mathit{children}^q_k \cup \{(n_1, \mathit{children}^1_e(n_1)),\\ & (n_2, \mathit{children}^1_e(n_2)), \dots, (n_{k-1}, \mathit{children}^1_e(n_{k-1}))\}\end{aligned}$$

The next step is to create a GraphIteration node to perform a graph traversal of our transformed RDF data as below:

$$n_g = (\texttt{GRAPHITERATION}, \mathit{vertex}, \mathit{edge}, \mathit{path}, w, 1, 1, \texttt{OUTBOUND}, \{\texttt{edges}\})$$

such that $\mathit{vertex} = \mathit{newVarName}(), \mathit{edge} = \mathit{newVarName}()$ and $\mathit{path} = \mathit{newVarName}()$.

We must then process p and o as following:

$$\mathit{transform}(p, \mathit{edge.predicate}, B', Z_0) = (B_1, Z_1)$$

$$\mathit{transform}(o, \mathit{vertex}, B_1, Z_1) = (B_2, Z_2)$$

We now take named graphs into consideration if applicable. We perform the below processing to obtain B_3, Z_3 and a'.

- If $a = \perp$:
  - if $v = \perp$:
    - if $G = \perp : B_3 = B_2, Z_3 = Z_2, a' = a$
    - if $G \ne \perp : B_3 = B_2, Z_3 = Z_2 \cup \{T_G\}$, $a' = \mathit{edge.graph}$

  - otherwise:
    - if $\exists k \in \mathbb{N}, k \leq i : b_k = (v_k, o_k, \langle a_k \rangle) \in B, v_k = v$ then $B_3 = B_2, Z_3 = Z_2 \cup \{T_v\}, a' = a$
    - otherwise: $B_3 = B_2 \cup \{(v, \texttt{FALSE}, \langle \mathit{edge.graph} \rangle)\}$, $Z_3 = Z_2 \cup \{T_G\}$, $a' = a$
- otherwise: $B_3 = B_2, Z_3 = Z_2 \cup \{T_a\}, a' = a$

such that T_G represents a $\texttt{POSITION}(G, \mathit{edge.graph.value}, \texttt{FALSE}) = \texttt{TRUE}$ condition, T_v represents a $\mathit{edge.graph} = a_k$ condition, and T_a represents a $\mathit{edge.graph} = a$ condition.

Assume that $Z_3 = \{T_1, T_2, \dots T_l\}$ for some $l \in \mathbb{N}_0$ after processing, such that $\forall m \in \mathbb{N}, m \leq l : T_m = (N^z_m, r^z_m, \mathit{children}^z_m)$.

Then we must join the expression trees in Z_3 into a single expression tree $E_2 = (N^2_e, r^2_e, \mathit{children}^2_e)$ as follows.

If $l = 0$:

$$N^2_e = \{n^2_0\}$$
$$n^2_0 = (\texttt{VALUE}, \texttt{TRUE})$$
$$r^2_e = n^2_0$$
$$\mathit{children}^2_e = \{(n^2_0, \langle\rangle)\}$$

Otherwise, assuming $x_0 = r^z_1$:

$$\forall m \in \mathbb{N}, m < l : x_m = (\texttt{AND}), \mathit{children}^2_e(x_m) = \langle x_{m-1}, r^z_{m+1} \rangle$$
$$N^2_e = N^z_1 \cup N^z_2 \cup \dots \cup N^z_l \cup \{x_1, x_2, \dots, x_{l-1}\}$$
$$r^2_e = x_{l-1}$$

$$\begin{aligned} \mathit{children}^2_e = {} & \mathit{children}^z_1 \cup \mathit{children}^z_2 \cup \dots \cup \mathit{children}^z_l \cup \{(x_1, \mathit{children}^2_e(x_1)), \\ & (x_2, \mathit{children}^2_e(x_2)), \dots, (x_{l-1}, \mathit{children}^2_e(x_{l-1}))\} \end{aligned}$$

Then we can say that $\mathit{transform}(t, B, G, v, a) = (T, B', a')$ such that $T = (N, r, \mathit{children})$ is an AQL query tree and B' is a set of variable binders as following:

$$N = \{n_s, n^1_f, n_g, n^2_f, n_{nest}\}$$
$$n^1_f = (\texttt{FILTER}, E_1)$$
$$n^2_f = (\texttt{FILTER}, E_2)$$
$$n_{nest} = (\texttt{NEST})$$
$$r = n_{nest}$$
$$\mathit{children} = \{(n_{nest}, \langle n^1_f, n^2_f \rangle), (n^1_f, \langle n_s \rangle), (n_s, \langle\rangle), (n^2_f, \langle n_g \rangle), (n_g, \langle\rangle)\}$$

7.4.2 SolutionTable node

When a SolutionTable node is encountered, the multiset of solution mappings is transformed into an array of objects, and an AQL `FOR` loop is introduced over this array. We formally define the node transformation below.

Definition 7.13 (Transformation of a SolutionTable node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{TABLE}, \Omega) \in N$ be a SPARQL SolutionTable node in the tree such that $\Omega = \{\mu_1, \mu_2, \ldots, \mu_i\}$ for some $i \in \mathbb{N}_0$.

Firstly we transform the multiset of solution mappings into an array of JSON objects as below:

$$a = \begin{cases} \langle\{\}\rangle, & \text{if } i = 0 \text{ and } \exists x \in N : x = (\texttt{JOIN}), n_s \in \mathit{children}(x) \\ a_s, & \text{otherwise} \end{cases}$$

$$a_s = \langle v_1, v_2, \ldots, v_i \rangle \text{ is a JSON array such that } \forall j \in \mathbb{N}_0, j \leq i :$$
$$v_j = \{(c, f) \mid (c \to d) \in \mu_j : f = \mathit{transform}(d)\} \text{ is a JSON object}$$

Let $n_1 = (\texttt{ITERATION}, w, E)$ be an Iteration node such that $w = \mathit{newVarName}()$ and $E = (\texttt{VALUE}, a)$.

Then $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following:

$$N_T = \{n_1\}$$

$$r_T = n_1$$

$$\mathit{children}_T = \{(n_l, \langle\rangle)\}$$

$$B = \{(v, o, \langle w.v \rangle) \mid v \in \mathit{dom}(\Omega), o = \begin{cases} \mathit{true}, & \text{if } \exists \mu \in \Omega : \mu(v) \text{ is undefined} \\ \mathit{false}, & \text{otherwise} \end{cases}$$

7.4.3 Join node

To join the results of two AQL subqueries, one subquery is nested within the other, and a `FILTER` statement is added to the inner query, specifying equality conditions between the common variables in both subqueries. This is similar to how we join triple pattern results in a `BGP`. We take the variable binders for variables that are present in both subqueries, and for every pair of variable binders, we add a condition that the value is equal for both, or that the variable is unbound in at least one of the subqueries, in which case the value is null.

We then add and keep a new set of variable binders for this `JOIN` node, which is obtained by merging the two sets of subquery variable binders.

Definition 7.14 (Transformation of a Join node). Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{JOIN}) \in N$ be a SPARQL Join node in the tree such that $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$.

Firstly, we transform the child nodes n_s^1 and n_s^2 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B^1), \mathit{transform}(n_s^2) = (T_a^2, B^2)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1), T_a^2 = (N_2, r_2, \mathit{children}_2)$$

Then we modify T_a^2 by adding a `FILTER` node with equality conditions over it, obtaining $T_a^3 = (N_2', r_2', \mathit{children}_2')$ as follows:

$$N_2' = N_2 \cup \{n_f\}$$

$$n_f = (\texttt{FILTER}, filters(B_1, B_2))$$
$$r_2' = n_f$$
$$\mathit{children}_2' = \mathit{children}_2 \cup \{(n_f, \langle r_2 \rangle)\}$$

Then we can say that $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following:

$$N_T = N_1 \cup N_2' \cup \{n_1\}$$

$$n_1 = (\texttt{NEST})$$

$$r_T = n_1$$

$$\mathit{children}_T = \mathit{children}_1 \cup \mathit{children}_2' \cup \{(n_1, \langle r_1, r_2' \rangle)\}$$
$$B = \mathit{merge}(\langle B^1, B^2 \rangle)$$

7.4.4 Minus node

Although AQL provides a `MINUS` function, it does not work in the same way as the Minus operator in SPARQL. Thus, for this operation, we need to use a combination of other AQL constructs.

If there are no common SPARQL variables between the left and right graph patterns, the minus operation is ignored, since it would not remove any result. Thus, the AQL subquery representing the right graph pattern can be completely discarded.

Otherwise, we want to remove solutions for the left graph pattern that are compatible with some solution for the right graph pattern. The way we perform this in AQL is shown in the sample query transformation given in Figure 7.3. What we do is loop over the left-side results and within that loop we nest a `LET` statement which stores the count of right-side results that are compatible with the current left result. Thus, the `LET` statement assigns the result of an AQL subquery to a variable.

The subquery is a loop over the right-side results, with added equality `FILTER` conditions to make sure that the right-side value for a SPARQL variable present in both graph patterns is the same as the value for that variable in the left-side result. The subquery then projects the count of right-hand results found satisfying those conditions. In the outer loop, we then add a `FILTER` condition keeping only the left-hand results for which the count of compatible right-hand results computed within the `LET` statement is zero.

Definition 7.15 (Transformation of a Minus node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{MINUS}) \in N$ be a SPARQL Minus node in the tree such that $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$.

Firstly we transform the child nodes n_s^1 and n_s^2 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B^1), \mathit{transform}(n_s^2) = (T_a^2, B^2)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1), T_a^2 = (N_2, r_2, \mathit{children}_2)$$

such that $B^1 = \{b_1, b_2, \dots, b_i\}$ for some $i \in \mathbb{N}_0$ and $B^2 = \{d_1, d_2, \dots, d_j\}$ for some $j \in \mathbb{N}_0$. We then find the set of SPARQL variables bound in both B^1 and B^2 as below:

$$C = \{c_k \mid k, l \in \mathbb{N}, k \leq i, l \leq j : b_k = (c_k, o_k, A_k) \in B^1, (c_l, o_l, A_l) \in B^2, c_k = c_l\}$$

If $C = \{\}$, then $\mathit{transform}(n_s) = (T_a^1, B^1)$.

Otherwise, we must add a Filter node over T_a^2 with equality conditions for variables in C, upon which we then add a `COLLECT WITH COUNT` clause in node form to count the amount of solutions for T_a^2 matching solutions for T_a^1. We then add a Project node over the modified tree to obtain T_a^3 as follows:

$$T_a^3 = (N_3, r_3, \mathit{children}_3)$$

$$N_3 = N_2 \cup \{n_b, n_c, n_p\}$$
$$n_b = (\texttt{FILTER}, filters(B^1, B^2))$$
$$n_c = (\texttt{COLLECT}, \langle\rangle, \texttt{TRUE}, \texttt{length})$$
$$n_p = (\texttt{PROJECT}, \{(\texttt{length}, (\texttt{VARIABLE}, \texttt{length}))\}, \texttt{FALSE})$$
$$r_3 = n_p$$
$$\mathit{children}_3 = \mathit{children}_2 \cup \{(n_b, \langle r_2 \rangle), (n_c, \langle n_b \rangle), (n_p, \langle n_c \rangle)\}$$

We then add an Assignment node, such that the results of the evaluation of the AQL subquery represented by T_a^3, i.e. the count of matching solutions, are assigned into a variable as below:

$$n_a = (\texttt{ASSIGNMENT}, w, E)$$

$$w = \mathit{newVarName}()$$
$$E = (N_e, r_e, \mathit{children}_e)$$
$$N_e = \{n_q\}$$
$$n_q = (\texttt{EXPRQUERY}, T_a^3)$$

$$r_e = n_q$$

$$children_e = \{(n_q, \langle\rangle)\}$$

Then we have to filter out solutions for T_a^1 that match any solution for T_a^3 using the below Filter node n_f.

$$n_f = (\texttt{FILTER}, E_f)$$

$$E_f = (N_f, r_f, children_f)$$

$$N_f = \{n_e, n_v, n_z\}$$

$$n_e = (\texttt{EQUALS})$$

$$n_v = (\texttt{VARIABLE}, w)$$

$$n_z = (\texttt{VALUE}, 0)$$

$$r_f = n_e$$

$$children_f = \{(n_e, \langle n_v, n_z\rangle), (n_v, \langle\rangle), (n_z, \langle\rangle)\}$$

Then we can say that $transform(n_s) = (T, B)$ such that T is an AQL query tree and B is a set of variable binders as following:

$$T = (N_T, r_T, children_T)$$

$$N_T = N_1 \cup \{n_a, n_{nest}, n_f\}$$

$$n_{nest} = (\texttt{NEST})$$

$$r_T = n_f$$

$$children_T = children_1 \cup \{(n_f, \langle n_{nest}\rangle), (n_{nest}, \langle r_1, n_a\rangle), (n_a, \langle\rangle)\}$$

$$B = B^1$$

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX schema: <http://schema.org/>
PREFIX ex: <http://example.org/>

SELECT ?p ?f
WHERE  {
   ?p foaf:knows ?f
   MINUS
   {
      ?f schema:birthPlace ex:Czech_Republic
   }
}
```

(a) SPARQL query

Figure 7.3: Sample transformation of a SPARQL query with a Minus operation

```
FOR forloop1item IN triples
  FILTER (forloop1item.predicate.value=="http://xmlns.com/
     foaf/0.1/knows")
  LET assign1item = (
    FOR forloop2item IN triples
      FILTER (forloop2item.predicate.value=="http://schema.
         org/birthPlace")
         && (forloop2item.object.type=="IRI")
         && (forloop2item.object.value=="http://example.org
            /Czech_Republic")
      FILTER (forloop1item.object==forloop2item.subject)
      COLLECT WITH COUNT INTO length
      RETURN length
  )
  FILTER (assign1item==0)
  RETURN {p: forloop1item.subject, f: forloop1item.object}
```

(b) Equivalent AQL query

Figure 7.3: Sample transformation of a SPARQL query with a Minus operation (cont.)

7.4.5 LeftJoin node

Left joins are not directly supported in AQL and are thus more complicated to translate. A Left join operation can be represented in AQL using subqueries as shown in Figure 7.4. The node transformation is formally defined in the below definition.

Definition 7.16 (Transformation of a LeftJoin node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{JOIN}) \in N$ be a SPARQL Join node in the tree such that $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$.

Firstly, we transform the child nodes n_s^1 and n_s^2 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B_1), \mathit{transform}(n_s^2) = (T_a^2, B_2)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1), T_a^2 = (N_2, r_2, \mathit{children}_2)$$

We then need to modify T_a^2 by adding a `FILTER` node with conditions for matching variables bound in both B_1 and B_2. Finally we also add a `PROJECT` node over the modified tree, as following:

$$T_2' = (N_2', r_2', \mathit{children}_2')$$

$$N_2' = N_2 \cup \{n_f, n_p\}$$

$$n_f = (\texttt{FILTER}, \mathit{filters}(B_1, B_2))$$

$$n_p = (\texttt{PROJECT}, \mathit{getProjectVarExprs}(\bot, B_2), \texttt{FALSE})$$

$$r_2' = n_p$$
$$\mathit{children}_2' = \mathit{children}_2 \cup \{(n_f, \langle r_2 \rangle), (n_p, \langle n_f \rangle)\}$$

We then create an Assignment node so that the results of the AQL query represented by T_2' will be assigned to a variable as following:

$$l = (\texttt{ASSIGNMENT}, w, T_2'), w = \mathit{newVarName}()$$

Then $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following.

$$N_T = N_1 \cup \{l, n_{nest}^1, n_i, n_{nest}^2\}$$
$$n_{nest}^1 = (\texttt{NEST}), \mathit{children}_T(n_{nest}^1) = \langle T_a^1, l \rangle$$
$$n_i = (\texttt{ITERATION}, \mathit{newVarName}(), E)$$
$$E = (N_e, r_e, \mathit{children}_e)$$
$$N_e = \{n_c, n_g, n_l, n_v^1, n_v^2, n_z, n_a\}, r_e = n_c$$
$$n_c = (\texttt{CONDITIONAL}), \mathit{children}_e(n_c) = \langle n_g, n_v^2, n_a \rangle$$
$$n_g = (\texttt{GREATHERTHAN}), \mathit{children}_e(n_g) = \langle n_l, n_z \rangle$$
$$n_l = (\texttt{LENGTH}), \mathit{children}_e(n_l) = \langle n_v^1 \rangle$$
$$n_v^1 = (\texttt{VARIABLE}, w)$$
$$n_z = (\texttt{VALUE}, 0)$$
$$n_v^2 = (\texttt{VARIABLE}, w)$$
$$n_a = (\texttt{VALUE}, [\{\}])$$

$$\begin{aligned}\mathit{children}_e = \{&(n_c, \langle n_g, n_v^2, n_a \rangle), (n_g, \langle n_l, n_z \rangle), (n_l, \langle n_v^1 \rangle), (n_v^1, \langle\rangle), (n_z, \langle\rangle), (n_v^2, \langle\rangle),\\ &(n_a, \langle\rangle)\}\end{aligned}$$

$$n_{nest}^2 = (\texttt{NEST}), \mathit{children}_T(n_{nest}^2) = \langle n_{nest}^1, n_i \rangle$$
$$r_T = n_{nest}^2$$
$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_{nest}^1, \langle T_a^1, l \rangle), (l, \langle\rangle), (n_{nest}^2, \langle n_{nest}^1, n_i \rangle), (n_i, \langle\rangle)\}$$

Before we merge the set of variable binders B_1 and B_2 we need to update every variable binder in B_2 to indicate that each variable could possibly be unbound. Assuming that $B_2 = \{b_1, b_2, \ldots, b_i\}$ for some $i \in \mathbb{N}$, we obtain B_2' as following:

$$B_2' = \{b \mid j \in \mathbb{N}, j \leq i : b_j = (v_j, o_j, A_j) \in B_2, b = (v_j, \texttt{FALSE}, A_j)\}$$

Then we can say that $B = \mathit{merge}(\langle B_1, B_2' \rangle)$.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?x ?y ?label
WHERE
{
  ?x rdf:type ?y
  OPTIONAL { ?x rdfs:label ?label }
}
```

(a) SPARQL query

```
LET assign1item = (
  FOR forloop1item IN triples
  FILTER (forloop1item.predicate.value=="http://www.w3.org
     /1999/02/22-rdf-syntax-ns#type")
  RETURN {x: forloop1item.subject, y: forloop1item.object}
)
LET assign3item = (
  FOR forloop3item IN assign1item
  LET rightResults = (
    FOR forloop2item IN triples
    FILTER (forloop2item.predicate.value=="http://www.w3.
       org/2000/01/rdf-schema#label")
    FILTER (forloop3item.x==forloop2item.subject)
    RETURN { x: forloop2item.subject,
             label: forloop2item.object }
  )
  FOR optionalResult IN ((LENGTH(rightResults)>0.0) ?
     rightResults : [{ /* no match exists */ }])
  RETURN { x: forloop3item.x, y: forloop3item.y,
           label: optionalResult.label }
```

(b) Equivalent AQL query

Figure 7.4: Sample transformation of a SPARQL query with a LeftJoin operation

7.4.6 Union node

To combine two result sets in AQL, the `UNION` array function can be used by passing the results of two subqueries as function parameters. The result of the function call is a single array containing all the array elements combined.

In some cases, such as when transforming a Union node, we need to apply a Project node over an AQL query tree so that it represents a valid, complete query expression. This is required when we need to assign the results of an AQL subquery into a variable or pass the results to some function. We define a function for applying a Project node every an incomplete AQL query tree in Definition 7.17, followed by a definition for the transformation of a Union node. We also provide a sample transformation of a SPARQL query containing a Union operation in Figure 7.6.

Definition 7.17 (Applying a Project node over an incomplete AQL query tree)**.** Let $T = (N, r, \mathit{children})$ be an AQL query tree and $B = \{b_1, b_2, \ldots, b_i\}$ for some $i \in \mathbb{N}$ be a set of variable binders. Then $\mathit{ensureComplete}(T, B) = (T', B')$ such that:

$$T' = \begin{cases} T, & \text{if } \mathit{type}(r) = \texttt{PROJECT} \\ (N', r', \mathit{children}'), & \text{otherwise} \end{cases}$$

$$N' = N \cup \{n_p\}$$

$$n_p = (\texttt{PROJECT}, V, \texttt{FALSE})$$

$$V = \{(v_j, T_j) \mid j \in \mathbb{N}, j \leq i : b_j = (v_j, o_j, A_j) \in B, T_j = \mathit{transform}(b_j)\}$$

$$r' = n_p$$

$$\mathit{children}' = \mathit{children} \cup \{(n_p, \langle r \rangle)\}$$

$$B' = \{(v_j, \texttt{FALSE}, \langle v_j \rangle) \mid j \in \mathbb{N}, j \leq i : (v_j, o_j, A_j) \in B\}$$

Definition 7.18 (Transformation of a Union node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{Union}) \in N$ be a SPARQL Union node in the tree such that $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$.

Firstly, we transform the child nodes n_s^1 and n_s^2, and add a `PROJECT` node over each of the transformed trees as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B_a^1), \mathit{transform}(n_s^2) = (T_a^2, B_a^2)$$

$$(T_1, B_1) = \mathit{ensureComplete}(T_a^1, B_a^1)$$

$$(T_2, B_2) = \mathit{ensureComplete}(T_a^2, B_a^2)$$

We then want to create an `ITERATION` node to loop over the `UNION` of the results of both subquery trees as below:

$$n = (\texttt{ITERATION}, w, E), w = \mathit{newVarName}()$$

$$E = (\{n_u, n_l, n_r\}, n_u, \mathit{children}_e)$$

$$n_u = (\texttt{UNION})$$

$$n_l = (\texttt{EXPRQUERY}, T_1)$$
$$n_r = (\texttt{EXPRQUERY}, T_2)$$
$$\mathit{children}_e = \{(n_u, \langle n_l, n_r \rangle), (n_l, \langle\rangle), (n_r, \langle\rangle)\}$$

Then we can say that $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following:

$$N_T = \{n\}$$

$$r_T = n$$
$$\mathit{children}_T = \{(n, \langle\rangle)\}$$
$$B_m = \mathit{merge}(\langle B_1, B_2 \rangle) = \{b_1, b_2, \ldots, b_i\} \text{ for some } i \in \mathbb{N}$$
$$B = \{b \mid j \in \mathbb{N}, j \leq i, b_j = (v_i, o_i, A_i) \in B_m : b = (v_i, o_i, \langle w.v_i \rangle)\}$$

```
PREFIX schema: <http://schema.org/>
PREFIX ex: <http://example.org/>

SELECT DISTINCT ?p
WHERE {
 {
   ?p schema:birthPlace ex:Malta .
 }
 UNION {
   ?p schema:birthPlace ex:Czech_Republic .
 }
}
```

(a) SPARQL query

Figure 7.5: Sample transformation of a SPARQL query with a Union operation

```
LET resultSet1 = (
  FOR forloop1item IN triples
  FILTER (forloop1item.predicate.value=="http://schema.org/
     birthPlace")
        && (forloop1item.object.type=="IRI")
        && (forloop1item.object.value=="http://example.org/
           Malta")
  RETURN {p: forloop1item.subject}
)
LET resultSet2 = (
  FOR forloop2item IN triples
  FILTER (forloop2item.predicate.value=="http://schema.org/
     birthPlace")
        && (forloop2item.object.type=="IRI")
        && (forloop2item.object.value=="http://example.org/
           Czech_Republic")
  RETURN {p: forloop2item.subject}
)
FOR forloop3item IN UNION(resultSet1,resultSet2)
  RETURN DISTINCT {p: forloop3item.p}
```

(a) Equivalent AQL query

Figure 7.6: Sample transformation of a SPARQL query with a Union operation (cont.)

7.4.7 Filter node

A SPARQL Filter operation is represented in AQL using its own `FILTER` statement. This requires translating the SPARQL filtering conditions into equivalent filter expressions in AQL, as defined in Definition 7.19 and shown in the sample query transformation in Figure **??**.

Definition 7.19 (Transformation of a Filter node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{FILTER}, E) \in N$ be a SPARQL Filter node in the tree such that $\mathit{children}(n_s) = \langle n_s^1 \rangle$.

Firstly we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1)$$

Then we can say that $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree as following:

$$N_T = N_1 \cup \{n_f\}$$

$$n_f = (\texttt{FILTER}, \mathit{transform}(E, B))$$

$$r_T = n_f$$

$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_f, \langle r_1 \rangle)\}$$

7.4.8 Extend node

The SPARQL Extend operator assigns the result of an expression into a variable and adds that variable mapping into the current solution mapping.

To extend each solution mapping in the multiset of solutions for a graph pattern, i.e. the array of current results in AQL, we nest a `LET` assignment within the current `FOR` loop, which assigns the result of the extend expression into an AQL variable with the same name as the SPARQL variable in the extend operation. We then keep a variable binder that tells us that the SPARQL variable is bound to the value of the assigned AQL variable. We give the transformation of a sample SPARQL query containing an extend operation in Figure 7.7.

For the following definition, we assume that an Extend node cannot try to bind a variable that has already been bound in its subtree.

Definition 7.20 (Transformation of an Extend node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{EXTEND}, E) \in N$ be a SPARQL Extend node in the tree such that $E = \{(v_1, e_1), (v_2, e_2), \ldots, (v_m, e_m)\}$ for some $m \in \mathbb{N}$ and $\mathit{children}(n_s) = \langle n_s^1 \rangle$.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B^1)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1)$$

We then create an `ASSIGNMENT` node for each pair in E to compute and store the result of an expression tree to be bound to a variable. We also create a set of variable binders for these newly bound variables, as below:

$$N_e = \{n_i \mid i \in \mathbb{N}, i \leq m : n_i = (\texttt{ASSIGNMENT}, v_i, \mathit{transform}(e_i, B^1)\}$$

$$B_e = \{(v_i, \texttt{FALSE}, \langle v_i \rangle) \mid i \in \mathbb{N}, i \leq m : (v_i, e_i) \in E\}$$

Then we can say that $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following:

$$N_T = N_1 \cup N_e \cup \{n_f, n_l\}$$

$$n_f = (\texttt{NEST}), \mathit{children}_T(n_f) = \langle r_1, n_l \rangle$$

$$n_l = (\texttt{SEQUENCE}), \mathit{children}_T(n_l) = \langle n_1, n_2, \ldots, n_m \rangle$$

$$r_T = n_f$$

$$\begin{aligned}\mathit{children}_T = \mathit{children}_1 \cup \{&(n_f, \langle r_1, n_l \rangle), (n_l, \langle n_1, n_2, \ldots, n_m \rangle),\\ &(n_1, \langle\rangle), (n_2, \langle\rangle), \ldots, (n_m, \langle\rangle)\}\end{aligned}$$

$$B = B^1 \cup B_e$$

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?fullName
WHERE  {
   ?p foaf:firstName ?f ;
      foaf:lastName ?s
   BIND(CONCAT(?f, " ", ?s) AS ?fullName)
}
```

(a) SPARQL query

```
FOR forloop1item IN triples
FILTER (forloop1item.predicate.value=="http://xmlns.com/
   foaf/0.1/firstName")
FOR forloop2item IN triples
  FILTER (forloop2item.subject==forloop1item.subject)&&(
     forloop2item.predicate.value=="http://xmlns.com/foaf
     /0.1/lastName")
  LET fullName = CONCAT(forloop1item.object.value," ",
     forloop2item.object.value)
  RETURN fullName
```

(b) Equivalent AQL query

Figure 7.7: Sample transformation of a SPARQL query with an Extend operation

7.4.9 Order node

The SPARQL `ORDER BY` operator works in the same manner as the `SORT` operator in AQL, although in the latter we need to specify the document or object (sub)properties to be used to perform the ordering.

Moreover, we have to make sure that the ordering is performed correctly in the situation where we have a variable with more than one possible type. We need to follow the order introduced in the definition of the SPARQL language [42], that is the following order:

1. Unbound variables
2. Blank nodes
3. IRIs
4. Literals

In order to ensure this, in AQL we need to sort by the two document subattributes `type` and `value`. Since *null* < `BNODE` < `IRI` < `LITERAL`, sorting by the `type` attribute first ensures that types appear in the order defined in SPARQL. We then sort by the `value` attribute to sort result values of the same type alphabetically. This logic is not reflected in Definition 7.21 given below, however it is assumed that when the AQL query tree is being serialised into the actual AQL query expression, it is taken into consideration.

Although the order of literals can also be affected by their datatype and possible language tag, this ordering is not well-defined within the SPARQL specification, and thus we do not consider it.

Definition 7.21 (Transformation of an Order node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\mathtt{ORDER}, C) \in N$ be a SPARQL Order node in the tree, such that $C = \langle (E_1, d_1), (E_2, d_2), \ldots, (E_m, d_m) \rangle$ for some $m \in \mathbb{N}$, and $\mathit{children}(n_s) = \langle n_s^1 \rangle$.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1)$$

We then transform the sequence of sort conditions C as following:

$$C_a = \langle (E'_1, d_1), (E'_2, d_2), \ldots, (E'_m, d_m) \rangle$$

such that $\forall i \in \mathbb{N}, i \leq m : E'_i = \mathit{transform}(E_i, B)$.

Then we can say that $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree as following:

$$N_T = N_1 \cup \{n_o\}$$

$$n_o = (\mathtt{SORT}, C_a)$$

$$r_T = n_o$$

$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_o, \langle r_1 \rangle)\}$$

7.4.10 Project node

The Project operation is represented by looping over the current array of results and placing a `RETURN` statement within the `FOR` loop, in which we specify which bound variables are to be projected. We formally define the transformation of a Project node below.

Definition 7.22 (Transformation of a Project node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\mathtt{PROJECT}, P) \in N$ be a SPARQL Project node in the tree such that $P = \{v_1, v_2, \ldots, v_i\}$ for some $i \in \mathbb{N}$, and $\mathit{children}(n_s) = \langle n_s^1 \rangle$.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B^1)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1)$$

Then $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following:

$$N_T = N_1 \cup \{n_p\}$$

$$n_p = (\texttt{PROJECT}, \mathit{getProjectVarExprs}(P, B^1), \texttt{FALSE})$$
$$r_T = n_p$$
$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_p, \langle r_1 \rangle)\}$$
$$B = \{(v_j, \texttt{FALSE}, \langle v_j \rangle) \mid j \in \mathbb{N}, j \leq i : v_j \in P\}$$

7.4.11 Distinct node

The Distinct operation can be represented in AQL using a `RETURN DISTINCT` statement.

Definition 7.23 (Transformation of a Distinct node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{DISTINCT}) \in N$ be a SPARQL Distinct node in the tree such that $\mathit{children}(n_s) = \langle n_s^1 \rangle$.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1)$$

such that $r_1 = (\texttt{PROJECT}, V, d)$. We then modify r_1 to ensure a distinct projection, as following:

$$r_1' = (\texttt{PROJECT}, V, \texttt{TRUE})$$

Then we can say that $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following:

$$N_T = (N_1 \backslash \{r_1\}) \cup \{r_1'\}$$

$$r_T = r_1'$$
$$\mathit{children}_T = (\mathit{children}_1 \backslash \{(r_1, \mathit{children}_1(r_1))) \cup \{(r_1', \mathit{children}_1(r_1))\}\}$$

7.4.12 Slice node

A Slice operation can be represented using the `LIMIT` statement in AQL.

Definition 7.24 (Transformation of a Slice node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL query algebra tree and $n_s = (\texttt{SLICE}, \mathit{offset}, \mathit{limit}) \in N$ be a SPARQL Slice node in the tree such that $\mathit{children}(n_s) = \langle n_s^1 \rangle$.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1) = (T_a^1, B)$$

$$T_a^1 = (N_1, r_1, \mathit{children}_1)$$

Then $\mathit{transform}(n_s) = (T, B)$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL query tree and B is a set of variable binders as following:

$$N_T = N_1 \cup \{n_l\}$$

$$n_l = (\texttt{LIMIT}, \mathit{offset}, \mathit{limit})$$

$$r_T = n_l$$

$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_l, \langle r_1 \rangle)\}$$

7.4.13 SPARQL expression tree

We now define how a SPARQL expression tree within a SPARQL query tree node is transformed. To transform a SPARQL expression tree, we require the set of variable binders for the current scope of the SPARQL query tree being transformed.

Definition 7.25 (Transformation of a SPARQL expression tree)**.** Let $T = (N, r, \mathit{children})$ be a SPARQL expression tree and $B = \{b_1, b_2, \ldots, b_i\}$ for some $i \in \mathbb{N}_0$ be a set of variable binders. Then $\mathit{transform}(T, B) = \mathit{transform}(r, B)$.

In some cases, we need to access the sub-attribute of a document for comparisons, arithmetic operations, and function calls. For this reason, we need to be able to update Variable nodes that represent documents, such that instead they then represent sub-attributes of documents. We provide Definition 7.26 for this purpose.

Definition 7.26 (Transformation of a document variable to a document sub--attribute variable)**.** Let $n = (\texttt{VARIABLE}, v)$ be an AQL Variable node and $s \in \{\texttt{type}, \texttt{value}, \texttt{datatype}, \texttt{lang}\}$ be the name of a sub-attribute of a document. Then $\mathit{update}(n, s) \to (\texttt{VARIABLE}, v.s)$.

We now define the transformation of each possible type of node in a SPARQL expression tree.

Definition 7.27 (Transformation of a LogicalAnd or LogicalOr node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s \in N$ be a node in the tree such that $\mathit{type}(n_s) \in \{\texttt{AND}, \texttt{OR}\}$, and $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$. Moreover, let B be a set of variable binders.

Firstly, we transform the child nodes n_s^1 and n_s^2 as following:

$$\mathit{transform}(n_s^1, B) = T_a^1 = (N_1, r_1, \mathit{children}_1)$$

$$\mathit{transform}(n_s^2, B) = T_a^2 = (N_2, r_2, \mathit{children}_2)$$

Then $\mathit{transform}(n_s, B) = T$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree as following:

$$N_T = N_1 \cup N_2 \cup \{n_l\}$$

$$n_l = \begin{cases} (\texttt{AND}), & \text{if } \mathit{type}(n_s) = \texttt{AND} \\ (\texttt{OR}), & \text{otherwise} \end{cases}$$
$$r_T = n_l$$
$$\mathit{children}_T = \mathit{children}_1 \cup \mathit{children}_2 \cup \{(n_l, \langle r_1, r_2\rangle)\}$$

Definition 7.28 (Transformation of a LogicalNot node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (\texttt{NOT}) \in N$ be a LogicalNot node in the tree such that $\mathit{children}(n_s) = \langle n_s^1\rangle$. Moreover, let B be a set of variable binders.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1, B) = T_a^1 = (N_1, r_1, \mathit{children}_1)$$

Then $\mathit{transform}(n_s, B) = T$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree as following:

$$N_T = N_1 \cup \{n_l\}$$
$$n_l = (\texttt{NOT})$$
$$r_T = n_l$$
$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_l, \langle r_1\rangle)\}$$

Definition 7.29 (Transformation of a Bound node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (\texttt{BOUND}) \in N$ be a Bound node in the tree such that $\mathit{children}(n_s) = \langle n_s^1\rangle$. Moreover, let B be a set of variable binders.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1, B) = T_a^1 = (N_1, r_1, \mathit{children}_1)$$

Then $\mathit{transform}(n_s, B) = T$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree as following:

$$N_T = N_1 \cup \{n_e, n_l\}$$
$$n_e = (\texttt{NOTEQUALS})$$
$$n_l = (\texttt{VALUE}, \texttt{null})$$
$$r_T = n_e$$
$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_e, \langle r_1, n_l\rangle), (n_l, \langle\rangle)\}$$

Definition 7.30 (Transformation of a Language node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (\texttt{LANG}) \in N$ be a Language node in the tree such that $\mathit{children}(n_s) = \langle n_s^1\rangle$. Moreover, let B be a set of variable binders.

Firstly, we transform the child node n_s^1 as following:

$$\mathit{transform}(n_s^1, B) = (N_1, r_1, \mathit{children}_1)$$

such that $\mathit{type}(r_1) = \texttt{VARIABLE}$. Then we update r_1 so that we access the language sub-attribute of the document referenced by the variable, using $\mathit{update}(r_1, \texttt{lang})$. Then we can say that $\mathit{transform}(n_s, B) = T$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree as following:

$$N_T = \{r_1\}$$
$$r_T = r_1$$
$$\mathit{children}_T = \mathit{children}_1$$

Definition 7.31 (Transformation of an Equals or NotEquals node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (\texttt{EQUALS}) \in N$ be an Equals node in the tree such that $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$. Moreover, let B be a set of variable binders.

Firstly, we transform the child nodes n_s^1 and n_s^2 as following:

$$\mathit{transform}(n_s^1, B) = T_a^1 = (N_1, r_1, \mathit{children}_1)$$

$$\mathit{transform}(n_s^2, B) = T_a^2 = (N_2, r_2, \mathit{children}_2)$$

If and only if exactly one of the nodes r_1, r_2 is a variable, and the other is not a literal or IRI value, then we have to compare the `value` attribute of the JSON object represented by the variable. Thus,

- r_1 is modified using $\mathit{update}(r_1, \texttt{value})$ if and only if
  $(\mathit{type}(r_1) = (\texttt{VARIABLE}) \wedge \mathit{type}(r_2) \neq (\texttt{VARIABLE}) \wedge$
  $(\mathit{type}(r_2) \neq (\texttt{VALUE}) \vee (r_2 = (\texttt{VALUE}, v) \wedge v$ is not an IRI or literal$)))$
- r_2 is modified using $\mathit{update}(r_2, \texttt{value})$ if and only if
  $(\mathit{type}(r_1) \neq (\texttt{VARIABLE}) \wedge \mathit{type}(r_2) = (\texttt{VARIABLE})) \wedge$
  $(\mathit{type}(r_1) \neq (\texttt{VALUE}) \vee (r_1 = (\texttt{VALUE}, v) \wedge v$ is not an IRI or literal$)))$

Then $\mathit{transform}(n_s, B) = T$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree as following:

$$N_T = N_1 \cup N_2 \cup \{n_e\}$$

$$n_e = \begin{cases} (\texttt{EQUALS}), & \text{if } \mathit{type}(n_s) = \texttt{EQUALS} \\ (\texttt{NOTEQUALS}), & \text{otherwise} \end{cases}$$

$$r_T = n_e$$

$$\mathit{children}_T = \mathit{children}_1 \cup \mathit{children}_2 \cup \{(n_e, \langle r_1, r_2 \rangle)\}$$

In Definition 7.31 above, we did not specify that when checking for the equality of r_1 and r_2, if they both represent an ArangoDB document or JSON object for an RDF term, we have to separately check that the `type`, `value`, `datatype` and `lang` properties of r_1 are all equal to the corresponding property of r_2. As explained previously, this is done so that if only one of them contains the `_id`, `_key`, and `_rev` attributes, these will not affect their equality.

Definition 7.32 (Transformation of a LangMatches node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (\texttt{LANGMATCHES}) \in N$ be a LangMatches node in the tree such that $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$. Moreover, let B be a set of variable binders.

Firstly, we transform the child nodes n_s^1 and n_s^2 as following:

$$\mathit{transform}(n_s^1, B) = T_a^1 = (N_1, r_1, \mathit{children}_1)$$

$$\mathit{transform}(n_s^2, B) = T_a^2 = (N_2, r_2, \mathit{children}_2)$$

Finally, we can say that $\mathit{transform}(n_s, B) = T$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree as following.

If $n_s^2 = (\texttt{VALUE}, *)$:

$$N_T = N_1 \cup \{n_d, n_l\}$$

$$n_d = (\texttt{NOTEQUALS})$$

$$n_l = (\texttt{VALUE}, \texttt{null})$$

$$r_T = n_d$$

$$\mathit{children}_T = \mathit{children}_1 \cup \{(n_d, \langle r_1, n_l\rangle), (n_l, \langle\rangle)\}$$

Otherwise:

$$N_T = N_1 \cup N_2 \cup \{n_e\}$$

$$n_e = (\texttt{EQUALS})$$

$$r_T = n_e$$

$$\mathit{children}_T = \mathit{children}_1 \cup \mathit{children}_2 \cup \{(n_e, \langle r_1, r_2\rangle)\}$$

For the transformation of a Concat node given in Definition 7.33 below, we ignore the `language` and `datatype` attributes of literals and simply concatenate their string `value` properties, i.e. we treat them as simple literals.

Definition 7.33 (Transformation of a Concat node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (\texttt{CONCAT}) \in N$ be a Concat node in the tree such that $\mathit{children}(n_s) = \langle c_1, c_2, \ldots, c_m\rangle$ for some $m \in \mathbb{N}$. Moreover, let B be a set of variable binders.

Firstly we transform the children of n_s into a sequence of AQL expression trees as following:

$$X = \langle T_1, T_2, \ldots, T_m\rangle$$

such that $\forall i \in \mathbb{N}, i \leq m : T_i = \mathit{transform}(c_i, B) = (N_i, r_i, \mathit{children}_i)$ and if $\mathit{type}(r_i) = \texttt{VARIABLE}$, we update r_i using $\mathit{update}(r_i, \texttt{value})$ since we want to concatenate simple literals. Then we can say that $\mathit{transform}(n_s, B) = T$ such that $T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree as following:

$$N_T = N_1 \cup N_2 \cup \cdots \cup N_m \cup \{n_c\}$$

$$n_c = (\texttt{CONCAT})$$

$$r_T = n_c$$

$$\mathit{children}_T = \mathit{children}_1 \cup \mathit{children}_2 \cup \cdots \cup \mathit{children}_m \cup \{(n_c, \langle r_1, r_2, \ldots, r_m\rangle)\}$$

Definition 7.34 (Transformation of a Value node)**.** Let $n_s = (\texttt{VALUE}, v)$ be a SPARQL Value node. Then

$$\mathit{transform}(n_s, B) = \begin{cases} (\texttt{VALUE}, v), & \text{if } v \in \mathbb{S} \\ (\texttt{VALUE}, \mathit{transform}(v)), & \text{otherwise} \end{cases}$$

Definition 7.35 (Transformation of a Variable node)**.** Let $n_s = (\texttt{VARIABLE}, v)$ be a SPARQL Variable node and $B = \{b_1, b_2, \ldots, b_m\}$ for some $m \in \mathbb{N}$ be a set of variable binders. Then $\mathit{transform}(n_s, B) = \mathit{transform}(b_i)$ such that
$\exists i \in \mathbb{N}, i \leq m : b_i = (v_i, o_i, A_i) \in B, v_i = v$.

Definition 7.36 (Transformation of an Arithmetic node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (t) \in N$ be an Arithmetic node in the tree such that $\mathit{children}(n_s) = \langle n_s^1, n_s^2 \rangle$. Moreover, let B be a set of variable binders.

Firstly we transform the child nodes n_s^1 and n_s^2 as following:

$$\mathit{transform}(n_s^1, B) = (N_1, r_1, \mathit{children}_1)$$

$$\mathit{transform}(n_s^2, B) = (N_2, r_2, \mathit{children}_2)$$

If and only if $r_1 = (\texttt{VARIABLE}, v_1)$ such that $v_1 \in \mathbb{W}$ is a variable that represents a JSON object for an RDF literal, we update r_1 using $\mathit{update}(r_1, \texttt{value})$. The same applies to r_2, i.e. if and only if $r_2 = (\texttt{VARIABLE}, v_2)$ such that $v_2 \in \mathbb{W}$ is a variable that represents a JSON object for an RDF literal, we update r_2 using $\mathit{update}(r_2, \texttt{value})$.

Then we can say that $\mathit{transform}(n_s, B) = T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree such that:

$$N_T = N_1 \cup N_2 \cup \{n_r\}$$

$$n_r = (t)$$

$$r_T = n_r$$

$$\mathit{children}_T = \mathit{children}_1 \cup \mathit{children}_2 \cup \{(n_r, \langle r_1, r_2 \rangle)\}$$

For the following definition for the transformation of a SPARQL Exists or NotExists node, we assume that the graph pattern in the node can only contain basic graph patterns and simple joins. Other constructs are not considered for this particular case as they would significantly impact and complicate the other formal definitions. Nevertheless, the definitions can be modified and extended to allow further constructs.

Definition 7.37 (Transformation of an Exists or NotExists node)**.** Let $S = (N, r, \mathit{children})$ be a SPARQL expression tree and $n_s = (t, A) \in N$ be a SPARQL node in the tree such that $t \in \{\texttt{EXISTS}, \texttt{NOTEXISTS}\}$, and B be a set of variable binders.

Firstly we transform A as following:

$$\mathit{transform}(A) = (T_1, B_1)$$

$$T_1 = (N_1, r_1, \mathit{children}_1)$$

We then modify T_1 by adding a Filter node over it with equality conditions, to only keep solutions that match solutions in the parent scope of n_s, i.e. variables bound in both B_1 and B must have the same value. We also add a `PROJECT` node over the tree, obtaining T_1' as below:

$$T_1' = (N_1 \cup \{n_f, n_p\}, r_1', \mathit{children}_1')$$

$$n_f = (\texttt{FILTER}, filters(B, B_1))$$

$$n_p = (\texttt{PROJECT}, \{\}, \texttt{FALSE})$$

$$r'_1 = n_p$$

$$\mathit{children}'_1 = \mathit{children}_1 \cup \{(n_f, \langle r_1 \rangle), (n_p, \langle n_f \rangle)\}$$

We then use T'_1 as a subquery in a conditional filter expression, such that if n_s is an Exists node, the subquery must have returned at least one result, i.e. length of results is more than zero. Otherwise if n_s is a NotExists node, the subquery must have returned no result, i.e. length of results is zero. For this, we need to create the below nodes:

$$n_s = (\texttt{EXPRQUERY}, T'_1)$$

$$n_l = (\texttt{LENGTH})$$

$$n_o = \begin{cases} (\texttt{GREATERTHAN}), & \text{if } t = \texttt{EXISTS} \\ (\texttt{EQUALS}), & \text{otherwise} \end{cases}$$

$$n_z = (\texttt{VALUE}, 0)$$

Then we can say that $\mathit{transform}(n_s, B) = T = (N_T, r_T, \mathit{children}_T)$ is an AQL expression tree such that:

$$N_T = \{n_o, n_l, n_s, n_z\}$$

$$r_T = n_o$$

$$\mathit{children}_T = \{(n_o, \langle n_l, n_z \rangle), (n_l, \langle n_s \rangle), (n_s, \langle\rangle), (n_z, \langle\rangle)\}$$

7.5 Transforming the AQL query result

The last step after retrieving the results of the AQL query execution on the ArangoDB dataset is to return the data to the user in the format expected based for SPARQL `SELECT` query form.

Due to our transformation algorithm, each column in the result set is already labeled according to the SPARQL variable it bounds. Thus, these do not need to be changed and can simply be printed.

Each column value is either a `null` value, a document or JSON object representing some RDF IRI, blank node, or literal, or a string value representing a simple literal. In the case of a `null` value, we simply print out nothing, while in the case of an atomic string value, we simply print out that value as a quoted string. Otherwise, the nested `type` property of the JSON object must be read to check what type of RDF value it represents and format it accordingly.

If the object represents an IRI, then we can simply print out the IRI present in the `value` attribute. If the object represents a blank node, then we print it out in the form "_:" followed by the value of the `value` attribute of the object, that is the blank node label or identifier.

If the object represents a literal, then the format depends on the literal's datatype or language tag. If the literal has a language tag, then we print it in the format "⟨value_here⟩"@⟨lang_here⟩ such that ⟨value_here⟩ is replaced by the value of the `value` attribute of the object, and ⟨lang_here⟩ is replaced by the value of the `lang` attribute of the object. If the literal does not have a language tag but has a datatype, we print it in the format "⟨value_here⟩"^^⟨datatype_here⟩ such that ⟨value_here⟩ is replaced by the value of the `value` attribute of the object, and ⟨datatype_here⟩ is replaced by the value of the `datatype` attribute of the object.

7.6 Other Optimization Considerations

While transforming a SPARQL algebra tree into an AQL query tree, we consider other optimizations which are not mentioned in the transformations and formal definitions given previously.

One optimization is that before processing the triple patterns in a `BGP` node, we first modify the set of triple patterns into a sequence, such that we can first process the triple patterns which are most likely to match less data, thus improving the performance of the created nested `FOR` loops. For example, a triple pattern of which the subject, predicate, and object are all IRIs, will only match one triple. On the other hand, a triple pattern made up of three variables will match all the triples in the queried dataset if none of the variables were bound by previously processed triple patterns.

Another optimization considered is that when creating a graph traversal iteration to match triples to a triple pattern, we decide whether to perform an inbound or an outbound traversal based on the components of the particular triple pattern, as well as the set of currently bound variables. For example, if the subject of the triple pattern is a variable that has already been bound, we introduce an outbound traversal with the vertex representing the subject as the start vertex. We similarly use an outbound traversal when the subject is an RDF term, by first iterating over the vertex collection to find the document matching the subject, and then using it as the start vertex for the traversal. However, if the subject is a variable that has not yet been bound, we look at the object of the triple pattern to check if it is an RDF term or a variable that has already been bound. If it is, then we can use an inbound traversal with the document representing the object as the start vertex. This way, we avoid high-cost iterations to match the subject when it is possible. If both the subject and object are variables that have not yet been bound, then the high-cost iteration to match the subject cannot be avoided.

Unrelated to the query transformation itself, using indexes on the ArangoDB collections significantly improves the performance of generated AQL queries. However, it is important to consider that over-indexing can result in a worse performance. Moreover, the proper index selection depends on the storage approach used, as well as the typical queries used. Thus, the user must keep this in mind when defining indexes.

8. Implementation

In this chapter, we describe our implemented tools and the technologies used to implement them. We present two implementations, these being a tool for transforming RDF into a JSON structure for ArangoDB, and a prototype implementation of a tool for transforming SPARQL queries into AQL queries.

8.1 Technologies

The following main technologies were used to implement our two command-line tools.

8.1.1 Java

Java was used as the programming language for both the RDF-to-ArangoDB data transformation tool as well as the SPARQL-to-AQL query translator. It is a modern, object-oriented language that is particularly popular due to it being platform-independent. It was chosen due to its portability as well as the stability of its RDF libraries.

8.1.2 Apache Jena

Apache Jena [7] is a free and open-source Java framework composed of different APIs that interact together to process RDF data. It allows us to load RDF data or SPARQL query expressions from file and transform them into a structure of Java class instances for manipulation.

ARQ Query Engine

ARQ [8] is a query engine for Apache Jena that supports the SPARQL query language. ARQ is used to transform a SPARQL query into a SPARQL algebra expression. It also provides out-of-the-box optimization processes for the algebra expressions, in the form of Transformer classes which one can pick and choose from and apply as seen fit. Users can also add their own optimization processes by building on top of the existing Transformer classes in ARQ.

8.1.3 ArangoDB Java Driver

We use the official ArangoDB Java Driver [13] to connect to an ArangoDB database from within our SPARQL-to-AQL query translation tool, in order to execute our generated AQL queries and retrieve and process the query results.

8.2 RDF data transformation tool

A command-line tool [1] that loads RDF data from file and transforms it into a JSON format that can be stored in ArangoDB was implemented. When running the tool, the user must choose whether to transform the RDF data using the basic approach or using the graph approach, as defined in Chapter 6.

The RDF data is read from file and into a Jena Dataset instance so that it can be processed and transformed more easily and efficiently. Moreover, if there is any syntax error in the data provided, this is caught immediately by the Jena library while attempting to load the data into the dataset, and the user is notified accordingly.

Based on the chosen transformation approach, the tool makes use of an instance of the `RdfToDocumentModelBuilder` class or the `RdfToGraphModelBuilder` class to transform the RDF data into the appropriate JSON data, which can then be imported into ArangoDB.

The generated JSON data is saved to one or more files, depending on the transformation approach. If the basic approach was used, all the JSON objects are saved to a single file. If the graph approach was used, the tool saves four files of JSON data. One file contains the JSON objects representing all the RDF resources, that is IRIs and blank nodes. Another file contains the JSON objects representing all RDF literals. Another file contains all the JSON objects representing ArangoDB edges between two resources. The last file contains all the JSON objects representing ArangoDB edges between a resource and a literal. The ArangoDB edges have to be separated into these two files due to the way data is imported into ArangoDB.

The data in these files then has to be manually imported into ArangoDB, using the *arangoimport* command-line tool utility [11] that comes with ArangoDB.

[1] Available at https://github.com/Ponsietta/RdfToArangoDBJson

8.3 SPARQL-to-AQL query transformation tool

Another command-line tool [2] that loads a SPARQL query expression from file, transforms it into an equivalent ArangoDB AQL query expression, and runs the latter query on the ArangoDB database was implemented. The tool does not implement full support for the SPARQL language and its algebra. It offers support for the `SELECT` query form, and all the SPARQL query constructs described and defined in Chapter 4, with the following limitations:

- limited support for nested `OPTIONAL` patterns
- `VALUES` clauses can only be present within a `SELECT WHERE` clause
- the graph pattern within an `EXISTS` or `NOT EXISTS` statement can only contain basic graph patterns

One difference between the prototype implementation and the SPARQL query algebra tree we define in Chapter 4 is that we support another node called a *QuadPattern node*. This type of node is used instead of a BGP node when a basic graph pattern needs to be evaluated against one or more named graphs, and not the default graph. In our implementation, a BGP node simply contains the basic graph pattern. A QuadPattern node stores a basic graph pattern as well as a named graph IRI or a SPARQL variable to which a named graph IRI needs to be bound.

The project structure of our tool is made up of the following main folders:

- `com.aql.querytree` contains classes that represent nodes in an AQL query tree, and other classes used to serialize an AQL query tree into a concrete AQL query expression
- `com.sparql_to_aql` contains classes used to transform a SPARQL query into an AQL query

8.3.1 Processing the query

The SPARQL query is initially read from file, using the file path passed by the user as a command-line argument. The `QueryFactory` class in Jena ARQ is then used to parse the string query into a `Query` object. If the SPARQL query expression is not syntactically correct, Jena will catch the error at this stage, and the user can be notified that the query is invalid.

The Algebra class in ARQ is then utilized to build the SPARQL query algebra tree for the query expression. At this point, some transformations are applied to the SPARQL algebra tree to optimize it and make it easier to convert into our AQL query tree, as described and defined in Sections 7.2 and 7.6. We used the `TransformReorder` transformer provided in ARQ to reorder the triple patterns in every BGP and QuadPattern node in the algebra tree so that we process the

[2] Available at https://github.com/Ponsietta/SparqlToArangoDbAql

triple patterns that filter out the most data first. This is an optimization that improves query runtime.

We also implemented some custom transformers for modifying the SPARQL algebra tree. Our `OpProjectOverSliceTransformer` is used to swap the positions of any encountered Slice node and its child Project node, since in AQL, the data first has to be sliced and then projected. Our `OpGraphTransformer` class is used to merge any encountered Graph node into each BGP node in its subtree, by transforming each BGP node into a QuadPattern node, and then removing the Graph node from the tree.

The next step is to convert the SPARQL query algebra tree into an AQL query tree. Depending on which approach was used to transform and store the RDF data we are querying, the `ArqToAqlTreeVisitor_BasicApproach` class or the `ArqToAqlTreeVisitor_GraphApproach` class is used to carry out this transformation. These classes extend the `ArqToAqlTreeVisitor` class, which contains transformation logic common between both approaches. These classes make use of the Visitor design pattern to perform a postorder traversal of the SPARQL algebra tree and generate corresponding AQL query tree nodes for each node traversed. During traversal, if any unsupported operator is found, an error is returned to the user. Moreover, when expressions are encountered in operators, for example filter expressions, the `RewritingExprVisitor` is used to translate each SPARQL expression into an equivalent expression in AQL.

The created AQL query tree is then converted to the actual AQL query expression using our `AqlQuerySerializer` class. The ArangoDB Java driver is used to execute the query on the ArangoDB database storing the transformed RDF data. The database name and names of the database collections containing the data to be queried are specified in the `config.properties` configuration file within the project, which can be updated by the user as required.

The last step is the conversion of the AQL result data into RDF form, which is performed as explained in Section 7.5 in the previous chapter. The correctly formatted result data is then saved to a CSV file for the user to view.

9. Evaluation

In this chapter, the implemented SPARQL-to-AQL query transformation tool, as well as the two RDF storage approaches we presented for ArangoDB, will be evaluated. The Berlin SPARQL Benchmark (BSBM) [51] was used to perform tests and performance comparisons. The RDF data used in this evaluation was generated using the data generator presented in the BSBM, and we used SPARQL queries from the query mix of the benchmark.

9.1 Experiments

The BSBM is built around an e-commerce use case, where we have a set of products offered by different vendors, as well as reviews of the products, posted by different consumers. As a basis for our experiments, we used the BSBM data generator to generate an RDF dataset based around 1000 products, resulting in a dataset of about 375,000 triples. We then used our RDF transformation tool to transform the generated RDF data into a JSON format appropriate for ArangoDB, after which the JSON data was imported into an ArangoDB database. Both the basic and the graph approach for transforming and storing the data were used.

We also uploaded the original RDF data into a Virtuoso database so that we could compare the results of querying RDF data stored in a popular RDF store, to the results of querying the same transformed RDF data in ArangoDB. By making these comparisons, we could ensure that our SPARQL-to-AQL query transformation tool works correctly as expected.

We chose to use the SPARQL queries described in Table 9.1 to evaluate our SPARQL-to-AQL tool and its performance. The actual SPARQL query expressions are included in Appendix A. We used our tool to transform each of the eight SPARQL query expressions into two equivalent AQL query expressions, one for querying data stored using the basic approach, and one for querying data stored using the graph approach.

We also used our tool to execute each of the AQL query expressions against the ArangoDB database containing our transformed RDF data. Since our tool saves the result data of each query to file, we could then execute the original SPARQL query over the Virtuoso database and compare the returned results to those obtained from our AQL query execution.

The result data obtained from each of our executed AQL queries matched the execution results of the original corresponding SPARQL query on Virtuoso. This shows that our tool correctly transforms the given SPARQL query expressions into equivalent AQL query expressions, and that the results returned by ArangoDB are the same as the results returned by Virtuoso, even in terms of sort order.

Query	Description
Q1	Find products for a given set of generic features – touches a large amount of data and uses `ORDER BY` and `LIMIT`
Q2	Retrieve basic information about a specific product – touches only a small amount of data, contains many triple patterns, and uses `OPTIONAL` graph patterns
Q3	Find products having some specific features and not having one feature – uses negation, `ORDER BY` and `LIMIT`
Q4	Find products matching two different sets of features – uses `UNION`, `ORDER BY`, `LIMIT` and `OFFSET`
Q5	Find products that are similar to a given product – touches a large amount of data, uses complex `FILTER` conditions involving arithmetic operations, uses `LIMIT`
Q6	Retrieve in-depth information about a specific product including offers and reviews – touches a large amount of data including products, offers, vendors, reviews and reviewers, uses `OPTIONAL` graph patterns
Q7	Get recent reviews in English for a specific product – uses `langMatches` function, `ORDER BY` and `LIMIT`
Q8	Get all information about an offer – contains triple patterns with unbound predicates and uses `UNION`

Table 9.1: Table of queries used for evaluation

9.2 Performance

In this section, the performance of our SPARQL-to-AQL tool and its generated AQL queries is evaluated using the SPARQL queries above.

The performance is tested on a Windows 10 64-bit laptop having 16GB of RAM and a 1.8GHz Intel Core i7-4500u CPU.

To evaluate the performance, we measure the time it takes for an AQL query generated by our tool to execute over the ArangoDB database and return the correctly formatted results. ArangoDB's default storage engine RocksDB is used. We also measure the time it takes for the original SPARQL query to execute over the Virtuoso database and return the result data. The execution times of both scenarios are then compared.

Every query was executed 15 times against the database. The two best and two worst execution times were removed and an average of the remaining execution times was taken. Table 9.2 shows the resulting average query execution times, and the same results are also plotted in the form of a bar chart in Figure 9.1.

Query	**ArangoDB** Basic Approach	**ArangoDB** Graph Approach	**Virtuoso**
1	17.9 ms	15.4 ms	1 ms
2	34.4 ms	59.3 ms	2.3 ms
3	21.5 ms	46.3 ms	0.6 ms
4	102.4 ms	124.4 ms	1 ms
5	100 ms	179.6 ms	0.9 ms
6	29.8 ms	50.1 ms	1.1 ms
7	20.4 ms	33.9 ms	1 ms
8	2.8 ms	6 ms	1.1 ms

Table 9.2: Query execution times

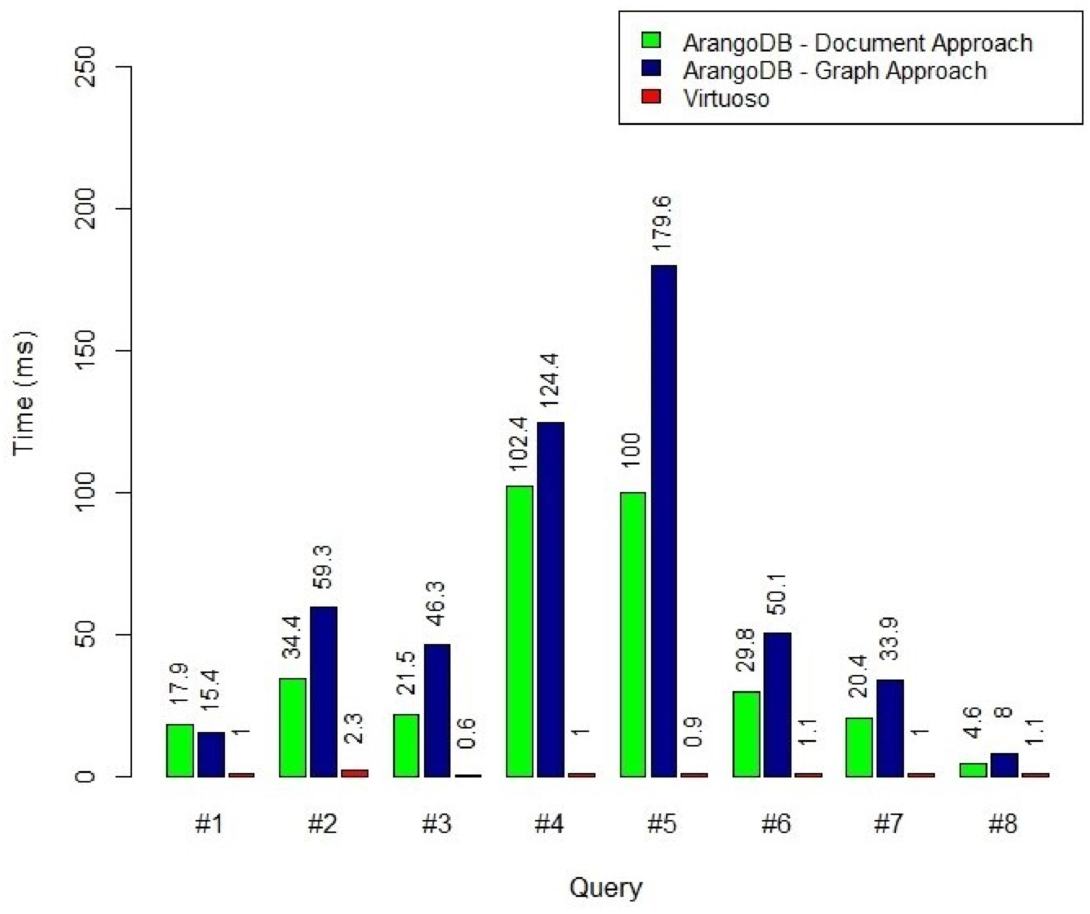


Figure 9.1: Plotted query execution times

It is important to mention that we defined a number of persistent indexes and vertex-centric indexes on our ArangoDB collections, which greatly improved our query runtimes.

Unfortunately, ArangoDB did not outperform Virtuoso for any of the queries used in our evaluation tests. Nevertheless, the query runtimes recorded were very reasonable both when querying data stored using the Basic Approach, as well as when using the Graph Approach. Query 4 and 5 stand out for having the longest average runtimes of all the queries for both basic and graph approaches.

For query 5, the longer runtime is due to the multiple arithmetic operations and comparisons in the query. The query optimizer does not use the available persistent indexes for improving the filter conditions involving these operations, as the indexes can only be used for the equality comparisons in the query and not the range comparisons. This is due to ArangoDB's indexing limitations.

In the case of query 4, the runtime is largely affected by the union operation. Once we union the results of two subqueries, the optimizer cannot make use of indexes for any consecutive filtering, sorting, and slicing of data.

Generally, an AQL query run on data stored using the graph approach, had a longer runtime than the equivalent AQL query run on data stored using the basic approach. This is due to the graph traversals performed in the former, which are expected to be slower than a regular iteration over a collection due to the traverser having to walk the graph edges and emit each vertex, edge, and path. Moreover, before performing a traversal, we often need to use a regular collection iteration to find all the possible start vertices for the traversal, depending on the subject or object of the triple pattern being processed. This adds to the cost of a query.

Although Virtuoso performed better than ArangoDB, this could be due to differences in their general system settings and configuration. For example, Virtuoso may be using a larger part of system memory than ArangoDB, or it may be caching more data and query results. Moreover, Virtuoso could be making greater utilization of the CPU than ArangoDB. Thus, comparing and adjusting their configurations could result in significant differences in query runtimes.

Furthermore, although Virtuoso outperformed ArangoDB in querying the dataset used for this thesis, it might not be the case if a much larger dataset is used. This is because ArangoDB is based on the horizontal scaling architecture and thus, its real advantage may only appear when working with truly big data.

Conclusion

In this master thesis, we presented two approaches that can be used to transform and store RDF data in ArangoDB, together with our main contribution, which is an algorithm for transforming a given SPARQL query into an AQL query, taking into consideration the chosen storage approach. A prototype implementation of our algorithm was also implemented, tested, and evaluated.

Although our query transformation algorithm does not fully cover the syntax and semantics of the SPARQL language, we have shown that it is possible to query RDF data in ArangoDB using SPARQL and that the performance of our evaluated queries is acceptable and relatively comparable to the popular RDF store Virtuoso. We also provided possible valid reasons for the slower performance of certain AQL queries and compared the performance of querying data stored using our basic approach against that of querying data stored using our graph approach.

We considered and handled several non-trivial issues concerning the query transformations, due to the differences between the SPARQL and AQL languages, as well as the different way data is stored in ArangoDB versus in a traditional RDF store.

Future Work

While this thesis presents a solution for storing and querying RDF data within a multi-model database, there are still ways to improve the solution.

Currently, the SPARQL-to-AQL query transformation tool provides limited support for the SPARQL algebra. The first task would be to expand and finalize the support for certain parts of the algebra which are currently supported with restrictions. The next task would be to add support for more components of the SPARQL language which were not considered in the thesis, until the whole language and its algebra is supported. This would most likely require the implementation of additional query optimizations.

Another task would be to extend our SPARQL-to-AQL query transformation algorithm to make it appropriate for querying RDF data stored in ArangoDB using the flattened representation described by Samuelsen [69], which we discussed in Chapter 3. It would be interesting to compare the performance of querying data stored using this approach against our two proposed approaches, as using the flattened representation could significantly improve query runtime.

One could also investigate the possible usage of xR2RML mappings for translating SPARQL queries into AQL queries. Using such mappings, although requiring more user input, could make translation more flexible and not restrict the user to using a single specific storage model.

Another future task is to implement a SPARQL endpoint that takes a SPARQL query provided by the user, transparently translates it into an AQL query using our algorithm and executes it on an ArangoDB database, and returns the result data to the user in RDF format. This would enable interoperability with other RDF backend applications.

Bibliography

[1] 4store. Last accessed 27 September 2019. URL: https://github.com/4store/4store.

[2] ADO.NET. Last accessed 3 December 2019. URL: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/.

[3] AllegroGraph. Last accessed 27 September 2019. URL: https://allegrograph.com/.

[4] Apachde HBase. Last accessed 29 September 2019. URL: https://hbase.apache.org/.

[5] Apache Cassandra. Last accessed 27 September 2019. URL: http://cassandra.apache.org/.

[6] Apache Hadoop. Last accessed 9 December 2019. URL: http://hadoop.apache.org.

[7] Apache Jena. Last accessed 16 July 2019. URL: https://jena.apache.org/.

[8] Apache Jena ARQ. Last accessed 16 July 2019. URL: https://jena.apache.org/documentation/query/.

[9] Apache Jena TDB. Last accessed 27 September 2019. URL: https://jena.apache.org/documentation/tdb/.

[10] ArangoDB. Last accessed 26 December 2019. URL: https://www.arangodb.com/.

[11] ArangoDB - Arangoimport. Last accessed 24 December 2019. URL: https://www.arangodb.com/docs/stable/programs-arangoimport.html.

[12] ArangoDB - Foxx Microservices. Last accessed 4 December 2019. URL: https://www.arangodb.com/docs/stable/foxx.html.

[13] ArangoDB Java Driver. Last accessed 24 December 2019. URL: https://www.arangodb.com/docs/stable/drivers/java.html.

[14] ArangoDB v3.5.3 HTTP API Documentation. Last accessed 4 December 2019. URL: https://www.arangodb.com/docs/stable/http/index.html.

[15] Couchbase. Last accessed 14 October 2019. URL: https://www.couchbase.com/.

[16] Cumulus RDF. Last accessed 27 September 2019. URL: https://github.com/cumulusrdf/cumulusrdf.

[17] DataGraft. Last accessed 2 October 2019. URL: https://datagraft.io/.

[18] Eclipse RDF4J. Last accessed 27 September 2019. URL: https://rdf4j.eclipse.org/.

[19] IBM Db2. Last accessed 27 September 2019. URL: https://www.ibm.com/cz-en/analytics/db2.

[20] Introducing JSON. Last accessed 17 June 2019. URL: https://www.json.org.

[21] Java JDBC API. Last accessed 3 December 2019. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

[22] JSON-LD. Last accessed 5 October 2019. URL: https://json-ld.org/.

[23] Microsoft Open Database Connectivity (ODBC). Last accessed 3 December 2019. URL: https://docs.microsoft.com/en-us/sql/odbc/microsoft-open-database-connectivity-odbc.

[24] MongoDB. Last accessed 19 September 2019. URL: https://www.mongodb.com/.

[25] MySQL. Last accessed 5 October 2019. URL: https://www.mysql.com/.

[26] Neo4j. Last accessed 5 October 2019. URL: https://neo4j.com/.

[27] NoSQL Performance Benchmark 2018. Last accessed 12 September 2019. URL: https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/.

[28] OpenLink Virtuoso Universal Server. Last accessed 15 September 2019. URL: https://virtuoso.openlinksw.com/.

[29] OrientDB. Last accessed 28 September 2019. URL: https://orientdb.com/.

[30] OWL 2 Web Ontology Language - Document Overview. Last accessed 23 November 2019. URL: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

[31] PostgreSQL. Last accessed 29 September 2019. URL: https://www.postgresql.org/.

[32] R2RML: RDB to RDF Mapping Language. Last accessed 19 September 2019. URL: https://www.w3.org/TR/r2rml/.

[33] RDF 1.1 Concepts and Abstract Syntax. Last accessed 27 December 2019.

[34] RDF 1.1 JSON Alternate Serialization (RDF/JSON). Last accessed 15 September 2019. URL: https://www.w3.org/TR/rdf-json/.

[35] RDF 1.1 N-Triples. Last accessed 30 July 2019. URL: https://www.w3.org/TR/n-triples/.

[36] RDF 1.1 TriG. Last accessed 23 November 2019. URL: https://www.w3.org/TR/trig/.

[37] RDF 1.1 Turtle. Last accessed 30 July 2019. URL: https://www.w3.org/TR/turtle/.

[38] RDF 1.1 XML Syntax. Last accessed 30 July 2019. URL: `https://www.w3.org/TR/rdf-syntax-grammar/`.

[39] RDF Schema 1.1. Last accessed 23 November 2019. URL: `https://www.w3.org/TR/rdf-schema/`.

[40] Semantic Web Ontologies. Last accessed 27 September 2019. URL: `https://www.w3.org/standards/semanticweb/ontology`.

[41] SKOS Simple Knowledge Organization System Reference. Last accessed 23 November 2019. URL: `https://www.w3.org/TR/skos-reference/`.

[42] SPARQL 1.1 Query Language. Last accessed 27 December 2019.

[43] SPARQL 1.1 Update. Last accessed 27 December 2019.

[44] The Linked Open Data Cloud. Last accessed 24 October 2019. URL: `https://lod-cloud.net/`.

[45] The World Wide Web Consortium (W3C). Last accessed 23 December 2019. URL: `https://www.w3.org/`.

[46] W3C Standards - Semantic Web. Last accessed 28 September 2019. URL: `https://www.w3.org/standards/semanticweb/`.

[47] XML Query. Last accessed 26 September 2019. URL: `https://www.w3.org/XML/Query/`.

[48] T. Berners-Lee and M. Fischetti. *Weaving the Web; The Original Design and Ultimate Destiny of the World Wide Web*. Harper, 1999.

[49] T. Berners-Lee, J. Hendler, and O Lissila. The Semantic Web. *Scientific American*, May 2001. Last accessed 29 September 2019. URL: `https://www.scientificamerican.com/article/the-semantic-web/`.

[50] N. Bikakis, C. Tsinaraki, I. Stavrakantonakis, N. Gioldasis, and S. Christodoulakis. The SPARQL2XQuery Interoperability Framework. Utilizing Schema Mapping, Schema Transformation and Query Translation to Integrate XML and the Semantic Web. *World Wide Web*, 18:403–490, January 2013.

[51] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5:1–24, 2009.

[52] C. Bizer and A. Seaborne. D2RQ - treating non-RDF databases as virtual RDF graphs. In *ISWC 2004 (posters)*, November 2004.

[53] E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk, and G. Xiao. OBDA beyond relational DBs: A study for MongoDB. In *Proc. 29th Int. Workshop on Description Logics*, volume 1577, 2016.

[54] R. Bouhali and A. Laurent. Exploiting RDF Open Data Using NoSQL Graph Databases. In R. Chbeir, Y. Manolopoulos, I. Maglogiannis, and R. Alhajj, editors, *Artificial Intelligence Applications and Innovations*, pages 177–190. Springer International Publishing, 2015.

[55] M. Chaloupka. Querying RDF graphs stored in a relational database using SPARQL and R2RML. Master's thesis, Charles University in Prague, 2014.

[56] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot. NoSQL Databases for RDF: An Empirical Evaluation. In *The Semantic Web – ISWC 2013*, pages 310–325. Springer Berlin Heidelberg, 2013.

[57] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proc. 7th Workshop on Linked Data on the Web*, April 2014. URL: `http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_01.pdf`.

[58] P. Fischer, D. Florescu, M. Kaufmann, and D. Kossmann. Translating SPARQL and SQL to XQuery. In *XML Prague*, pages 81–98, January 2011.

[59] A. Haque and L. Perkins. Distributed RDF Triple Store Using HBase and Hive. December 2012.

[60] V. Khadilkar, M. Kantarcioglu, B. M. Thuraisingham, and P. Castagna. Jena-HBase: A Distributed, Scalable and Effcient RDF Triple Store. In *International Semantic Web Conference*, 2012.

[61] S. Kiminki, J. Knuuttila, and V. Hirvisalo. SPARQL to SQL Translation Based on an Intermediate Query Language. 2010.

[62] F. Michel, L. Djimenou, C. Faron-Zucker, and J. Montagnat. xR2RML: Relational and Non-Relational Databases to RDF Mapping Language. Technical report, September 2015.

[63] F. Michel, C. Faron-Zucker, and J. Montagnat. A Generic Mapping-based Query Translation from SPARQL to Various Target Database Query Languages. pages 147–158, January 2016.

[64] F. Michel, C. Faron-Zucker, and J. Montagnat. A Mapping-Based Method to Query MongoDB Documents with SPARQL. volume 9828, pages 52–67, September 2016.

[65] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *The Semantic Web - ISWC 2006*, pages 30–43. Springer Berlin Heidelberg, 2006.

[66] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.

[67] K. Runapongsa Saikaew, C. Aswamenakul, and M. Buranarach. Design and evaluation of a NoSQL database for storing and querying RDF data. *KKU Engineering Journal*, 41:537–545, December 2014.

[68] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, chapter 13. Addison-Wesley, August 2012.

[69] S. D. Samuelsen. Representing and Storing Semantic Data in a Multi-Model Database. Master's thesis, University of Oslo, 2018.

[70] J. Sequeda and D. Miranker. Ultrawrap: Sparql execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 22:19–39, October 2013.

[71] L. Szeremeta and D. Tomaszuk. Document-oriented RDF graph store. *Studia Informatica*, 38:31–43, May 2017.

[72] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. *Proc. VLDB Endow.*, 6(4):265–276, February 2013.

[73] C. Zhang, J. Lu, P. Xu, and Y. Chen. Unibench: A benchmark for multi-model database management systems. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence*, pages 7–23. Springer International Publishing, 2019.

List of Figures

List of Tables

List of Abbreviations

AQL	ArangoDB Query Language.
BGP	Basic Graph Pattern.
BSBM	Berlin SPARQL Benchmark.
HTML	Hypertext Markup Language.
IRI	Internationalized Resource Identifier.
JSON	JavaScript Object Notation.
LOD	Linked Open Data.
LOD Cloud	Linked Open Data Cloud.
MMFiles	Memory-Mapped Files.
OBDA	Ontology-Based Data Access.
OWL	Web Ontology Language.
RDBMS	Relational Database Management System.
RDF	Resource Description Framework.
RDFS	RDF Schema.
SKOS	Simple Knowledge Organization System.
SPARQL	SPARQL Protocol and RDF Query Language.
SQL	Structured Query Language.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
W3C	World Wide Web Consortium.
WWW	World Wide Web.
XML	Extensible Markup Language.

A. Queries used for evaluation

```
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/
    bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
    v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?product ?label
WHERE {
 ?product rdfs:label ?label .
 ?product a bsbm-inst:ProductType15 .
 ?product bsbm:productFeature bsbm-inst:ProductFeature2113
     .
 ?product bsbm:productFeature bsbm-inst:ProductFeature10 .
 ?product bsbm:productPropertyNumeric1 ?value1 .
 FILTER (?value1 > 200)
}
ORDER BY ?label
LIMIT 10
```

Listing A.1: SPARQL query #1

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
    v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dataFromProducer3: <http://www4.wiwiss.fu-berlin.de/
    bizer/bsbm/v01/instances/dataFromProducer3/>
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/
    bsbm/v01/instances/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?label ?comment ?producer ?productFeature
  ?propertyTextual1 ?propertyTextual2 ?propertyTextual3
  ?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4
  ?propertyTextual5 ?propertyNumeric4
WHERE
  { dataFromProducer3:Product118
        rdfs:label ?label ;
        rdfs:comment ?comment ;
        bsbm:producer ?p .
    ?p  rdfs:label ?producer .
    dataFromProducer3:Product118
        dc:publisher ?p ;
        bsbm:productFeature ?f .
    ?f  rdfs:label ?productFeature .
    dataFromProducer3:Product118
        bsbm:productPropertyTextual1 ?propertyTextual1 ;
        bsbm:productPropertyTextual2 ?propertyTextual2 ;
```

```
          bsbm:productPropertyTextual3 ?propertyTextual3 ;
          bsbm:productPropertyNumeric1 ?propertyNumeric1 ;
          bsbm:productPropertyNumeric2 ?propertyNumeric2
      OPTIONAL
        {
          dataFromProducer3:Product118
              bsbm:productPropertyTextual4 ?propertyTextual4
        }
      OPTIONAL
        {
          dataFromProducer3:Product118
              bsbm:productPropertyTextual5 ?propertyTextual5
        }
      OPTIONAL
        {
          dataFromProducer3:Product118
              bsbm:productPropertyNumeric4 ?propertyNumeric4
        }
  }
```

Listing A.2: SPARQL query #2

```
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/
   bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?product ?label
WHERE {
 ?product rdfs:label ?label .
 ?product a bsbm-inst:ProductType2 .
 ?product bsbm:productFeature bsbm-inst:ProductFeature1717
    .
 ?product bsbm:productPropertyNumeric1 ?p1 .
 FILTER (?p1 > 400)
 ?product bsbm:productPropertyNumeric3 ?p3 .
 FILTER (?p3 < 1100)
 OPTIONAL
 {
    ?product bsbm:productFeature bsbm-inst:ProductFeature12
        .
    ?product rdfs:label ?testVar
 }
 FILTER (!bound(?testVar))
}
ORDER BY ?label
LIMIT 10
```

Listing A.3: SPARQL query #3

```
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/
   bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?product ?label ?propertyTextual
WHERE {
 {
  ?product rdfs:label ?label .
  ?product rdf:type bsbm-inst:ProductType3 .
  ?product bsbm:productFeature bsbm-inst:ProductFeature25 .
  ?product bsbm:productFeature bsbm-inst:ProductFeature2460
     .
  ?product bsbm:productPropertyTextual1 ?propertyTextual .
  ?product bsbm:productPropertyNumeric1 ?p1 .
  FILTER ( ?p1 > 100 )
 }
 UNION
 {
  ?product rdfs:label ?label .
  ?product rdf:type bsbm-inst:ProductType3 .
  ?product bsbm:productFeature bsbm-inst:ProductFeature25
    .
  ?product bsbm:productFeature bsbm-inst:ProductFeature34 .
  ?product bsbm:productPropertyTextual1 ?propertyTextual .
  ?product bsbm:productPropertyNumeric2 ?p2 .
  FILTER ( ?p2 > 750 )
 }
}
ORDER BY ?label
OFFSET 5
LIMIT 10
```

Listing A.4: SPARQL query #4

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>
PREFIX dataFromProducer4: <http://www4.wiwiss.fu-berlin.de/
   bizer/bsbm/v01/instances/dataFromProducer4/>

SELECT DISTINCT ?product ?productLabel
WHERE {
 ?product rdfs:label ?productLabel .
 FILTER (dataFromProducer4:Product159 != ?product)
 dataFromProducer4:Product159 bsbm:productFeature
    ?prodFeature .
 ?product bsbm:productFeature ?prodFeature .
 dataFromProducer4:Product159 bsbm:productPropertyNumeric1
    ?origProperty1 .
 ?product bsbm:productPropertyNumeric1 ?simProperty1 .
 FILTER (?simProperty1 < (?origProperty1 + 120) &&
    ?simProperty1 > (?origProperty1 - 120))
 dataFromProducer4:Product159 bsbm:productPropertyNumeric2
    ?origProperty2 .
 ?product bsbm:productPropertyNumeric2 ?simProperty2 .
 FILTER (?simProperty2 < (?origProperty2 + 170) &&
    ?simProperty2 > (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

Listing A.5: SPARQL query #5

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dataFromProducer6: <http://www4.wiwiss.fu-berlin.de/
   bizer/bsbm/v01/instances/dataFromProducer6/>

SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle
  ?review ?revTitle ?reviewer ?revName ?rating1 ?rating2
WHERE {
 dataFromProducer6:Product236 rdfs:label ?productLabel .
 OPTIONAL {
  ?offer bsbm:product dataFromProducer6:Product236 .
  ?offer bsbm:price ?price .
  ?offer bsbm:vendor ?vendor .
  ?vendor rdfs:label ?vendorTitle .
  ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/
     countries#DE> .
  ?offer dc:publisher ?vendor .
  ?offer bsbm:validTo ?date .
  FILTER (?date > "2008-07-10")
 }
 OPTIONAL {
  ?review bsbm:reviewFor dataFromProducer6:Product236 .
  ?review rev:reviewer ?reviewer .
  ?reviewer foaf:name ?revName .
  ?review dc:title ?revTitle .
  OPTIONAL { ?review bsbm:rating1 ?rating1 . }
  OPTIONAL { ?review bsbm:rating2 ?rating2 . }
 }
}
```

Listing A.6: SPARQL query #6

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dataFromProducer6: <http://www4.wiwiss.fu-berlin.de/
   bizer/bsbm/v01/instances/dataFromProducer6/>

SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?
   rating1 ?rating2 ?rating3 ?rating4
WHERE {
 ?review bsbm:reviewFor dataFromProducer6:Product250 .
 ?review dc:title ?title .
 ?review rev:text ?text .
 FILTER langMatches(lang(?text), "EN")
 ?review bsbm:reviewDate ?reviewDate .
 ?review rev:reviewer ?reviewer .
 ?reviewer foaf:name ?reviewerName .
 OPTIONAL { ?review bsbm:rating1 ?rating1 . }
 OPTIONAL { ?review bsbm:rating2 ?rating2 . }
 OPTIONAL { ?review bsbm:rating3 ?rating3 . }
 OPTIONAL { ?review bsbm:rating4 ?rating4 . }
}
ORDER BY DESC(?reviewDate)
LIMIT 20
```

Listing A.7: SPARQL query #7

```
SELECT ?property ?hasValue ?isValueOf
WHERE {
 {
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances
     /dataFromVendor1/Offer10> ?property ?hasValue
 }
 UNION
 {
  ?isValueOf ?property <http://www4.wiwiss.fu-berlin.de/
     bizer/bsbm/v01/instances/dataFromVendor1/Offer10>
 }
}
```

Listing A.8: SPARQL query #8