Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Queries4TOSCA: Concept and Development of a Query Language for TOSCA

Justin Kißling

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Miles Stötzner, M.Sc.
Commenced:	June 1, 2022
Completed:	December 1, 2022

Abstract

Cloud computing plays an increasingly important role in today's IT world. It lets enterprises access a virtually unlimited resource pool with a pay-per-use system similar to utilities like electricity and water. This led to its widespread adoption for large, scalable applications. These cloud applications are often composite systems made up of multiple heterogeneous components that interact with each other, making it necessary to deploy and manage them in an automated way. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard introduced to fulfill this task. It allows users to describe the topology of an application in a portable, platform-independent fashion in so-called service templates. It also allows the definition of tasks used to manage applications.

Service templates can become very complex and include a multitude of files, making it difficult to manually search through them. However, this might be necessary in some cases, for example when identifying components that need to be updated. This work aims to improve this situation by introducing a query language that can be used on single TOSCA templates or entire repositories. Through path expressions and filters, it allows users to specify the exact data they are looking for. Additionally, they can search for patterns within the topology of a service, such as traversing the entire hosting stack of a component.

This work also includes a prototypical implementation of the described query language as part of OpenTOSCA Vintner, which can query TOSCA templates from a variety of sources and pull instance data from orchestrators.

Kurzfassung

Cloud Computing spielt in der heutigen IT-Welt eine zunehmend wichtige Rolle. Es ermöglicht Unternehmen den Zugriff auf einen scheinbar unbegrenzten Ressourcenpool mit einem Pay-per-Use Modell, ähnlich wie Strom und Wasser. Durch diese Vorteile wird es inzwischen für viele große, skalierbare Anwendungen benutzt. Bei Cloud-Anwendungen handelt es sich oft um verteilte Systeme, bestehend aus mehreren heterogenen, miteinander interagierenden Komponenten, wodurch es notwendig ist, das Deployment und Management zu automatisieren. Eine Lösung dafür ist OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA), welche es Nutzern erlaubt, die Topologie ihrer Anwendung in einem portablen, plattformunabhängigen Format zu beschreiben. Außerdem lassen sich Management Tasks definieren, mit denen die Anwendung verwaltet werden kann. Dies geschieht mit Hilfe sogenannter Service Templates.

Da diese Service Templates sehr komplex sein können und oft aus mehreren Dateien bestehen, ist es schwierig, diese manuell zu durchsuchen, zum Beispiel wenn man eine Komponente identifizieren möchte, die ein Update braucht. Diese Arbeit setzt sich zum Ziel, eine Abfragesprache zu entwickeln, mit der man einzelne TOSCA Templates oder ganze Repositories durchsuchen kann. Mithilfe von Pfadausdrücken und Filtern kann ein Nutzer seine gewünschten Daten präzise definieren. Außerdem können Patterns in der Topology eines Services gefunden werden, zum Beispiel das Traversieren des gesamten Hosting Stacks einer Komponente.

Diese Arbeit präsentiert ebenfalls eine prototypische Implementierung der vorgestellten Abfragesprache als Teil von OpenTOSCA Vintner. Diese kann Abfragen auf TOSCA Templates aus verschiedenen Quellen ausführen, sowie Instanzdaten von einem Orchestrator beziehen.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API application programming interface. 37, 65

AST abstract syntax tree. 26, 59, 61, 62

BFS breadth-first search. 64

BNF Backus–Naur form. 29

BPEL Business Process Execution Language. 24

BPMN Business Process Model and Notation. 24

CFG context-free grammar. 26

CLI command-line interface. 59, 60, 65

CRUD create, read, update and delete. 33, 69

DBMS database management system. 17, 43

JSON JavaScript Object Notation. 31, 61, 64

NIST National Institute of Standards and Technology. 21, 22

OASIS Organization for the Advancement of Structured Information Standards. 3, 5, 17, 22, 24, 69

PEG parsing expression grammar. 18, 21, 26, 27, 49, 62

PGQL Property Graph Query Language. 18, 29, 37, 56

Q4T Queries4TOSCA. 14, 18, 19, 21, 29, 32, 41, 49, 57, 59, 61, 69, 70, 75

REST representational state transfer. 37, 59, 60, 61, 65

SQL Structured Query Language. 18, 29, 33, 34, 35, 57, 69

TOSCA Topology and Orchestration Specification for Cloud Applications. 3, 5, 14, 17, 18, 21, 22, 23, 24, 29, 31, 35, 37, 39, 41, 51, 55, 56, 63, 69

XML Extensible Markup Language. 22, 24, 29, 35, 69

YAML YAML Ain't Markup Language. 18, 21, 22, 24, 25, 26, 31, 43, 69

1 Introduction

This chapter explains the motivation behind this thesis in Section 1.1. Then, Section 1.2 gives an overview of the structure of this work and describes the contents of the following chapters.

1.1 Motivation

Cloud computing has become an increasingly ubiquitous phenomenon in the IT industry. According to a survey conducted by Eurostat in 2022, 72% of large enterprises in Europe use cloud computing services, an increase of seven percentage points compared to the previous year [Eur21]. This trend is not surprising, given the numerous benefits that cloud computing offers. Its pay-per-use model allows enterprises to access resources on demand, in a similar way to utilities like electricity or water [LF09]. This means that companies no longer need to make a large upfront investment into computing resources, the buildings to house them, or staff to manage them [Kim09]. Additionally, companies can react to an increase or decrease of demand without wasting unused resources.

Enterprise applications are usually large, composite systems, made up of multiple smaller components that provide various functionalities [BBKL14]. These components are often interconnected, for example, an application might connect to a database, which relies on a database management system (DBMS), which in turn runs on a virtual machine hosted on a physical server. The complex nature of these applications makes it necessary to automate at least some part of their operation and management, since doing these tasks manually is time-consuming and error-prone [BBKL14].

To enable the automated deployment and management of such cloud applications, the Organization for the Advancement of Structured Information Standards (OASIS) consortium introduced the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard. It can be used to describe web applications in a portable, platform independent format. The structure of an application is defined in a so-called topology template, which describes it in the form of nodes and relationships. Additionally, recurring tasks can be captured in management plans.

TOSCA definitions are often organized and managed in the form of a repository. These repositories can be very complex, containing a lot of different node types, making it impractical to manually search through them. However, in some cases it might be beneficial or even necessary to able to identify patterns in repositories. For example, imagine a scenario where a critical security update becomes available for a component that is used in multiple deployment models. With a query language, one would be able to quickly identify which models, or even currently running instances, contain this component. These instances could then be updated and redeployed.

A query language would also make it possible to project a large, complex service into smaller views, making it easier to understand. For example, one could run a query to retrieve a view that only contains software components that communicate with a specific other component, or a view that only consists of the hosting stack of a specific component.

Another potential use case is the determination of the IP address of a web application. TOSCA templates often contain complex hosting stacks, spanning over multiple components, for example including a web server and a virtual machine in between the application and the actual, physical hosting machine. Therefore, it would be desirable to have the ability to traverse this entire hosting stack to access information about the underlying host.

While there are existing solutions for path expressions in TOSCA, they are mostly limited to querying singular values within a template and cannot be run from an external program. Using one of the many established query languages is also not a viable solution in most cases. Languages like SQL do not lend themselves well to traverse the topology graph of TOSCA templates since they are optimized to work on flat, predefined database schemas. And while graph query languages like Cypher and PGQL offer the ability to traverse node graphs, they are not well suited to access the potentially hierarchical values contained in node templates, since their underlying graph databases only allow nodes with flat properties, leading to needlessly complex queries. Path-based query languages like XQuery allow a user to access values within the document that is the basis of a template, but lack the ability to find patterns inside the topology.

Therefore, to assist users when they need to access information about TOSCA templates and instances, this work introduces a query language called Queries4TOSCA (Q4T). It can be used to extract any kind of information from a service template using a path syntax enhanced with filters. The user also has full control over the shape of the output data, with the ability to define exact key-value pairs to include, making it easy to further process the output data.

In addition to the specification of this language, this work also presents a prototypical implementation of an interpreter with the ability to query attributes, which are only known at runtime. This is accomplished by gathering this information from an orchestrator and merging it into the service template, allowing the user to query the instance model of their application at the current point in time. It also allows users to put queries inside of a template and resolve them before deploying it. This gives them more flexibility and reusability when authoring templates.

1.2 Structure

This thesis begins with **Chapter 2**, which explains the fundamental background information necessary to understand the topic of this thesis. This includes cloud computing, TOSCA, and YAML, the file format on which TOSCA is based on. It also describes lexing, parsing, and parsing expression grammars (PEGs), which are important concepts for the execution of a query.

Afterwards, **Chapter 3** highlights important related work that deals with querying data, explains their approaches to constructing queries and relates them to the structure of the data they work on. This includes academic work related to querying TOSCA templates and cloud applications, but also other query languages working on various data formats.

Chapter 4 introduces the concept and syntax of Queries4TOSCA (Q4T), a query language for TOSCA. It starts out by presenting the running example and provides some introductory examples to the structure of queries. Afterwards, the full syntax of the language is explained, including all elements of the language with examples of their usage.

Chapter 5 explains the implementation of the prototype application to resolve Q4T queries. It first describes the overall philosophy and architecture behind the implementation. Afterwards, it goes into detail regarding multiple aspects, including parsing and resolving queries. Lastly, the limitations of the current prototype are pointed out.

Chapter 6 summarizes the previous chapters and presents the conclusion of this thesis. It also discusses several potential avenues for future research to build upon this work.

2 Fundamentals

This chapter introduces various concepts and technologies that help the reader understand the contents of this work. Section 2.1 introduces the concept of cloud computing. Section 2.2 then explains TOSCA, a specification for deploying and managing cloud applications. Section 2.3 gives a short introduction to YAML, which is the language used in the current TOSCA standard, making it vital to understand the decisions behind the concept and implementation of Queries4TOSCA (Q4T), the query language presented in this work. Section 2.4 describes the process of parsing an input, which is an important step in the execution of a query. Section 2.5 gives an overview of parsing expression grammars (PEGs), since one is used to describe the syntax of Q4T in the implementation.

2.1 Cloud Computing

One of the most commonly accepted definitions of cloud computing comes from the National Institute of Standards and Technology (NIST), which defines it as ”a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” [MG+11]. They define the following five essential characteristics: On-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.

On-demand self-service. The customer can unilaterally provision resources without any manual intervention by the provider.

Broad network access. Capabilities are available over the network.

Resource pooling. The resources of the provider are pooled together and possibly shared by multiple tenants, with the customer having only limited control over the location of the resources.

Rapid elasticity. Capabilities can be provisioned elastically to keep up with demand.

Measured service. Resource usage by the customer is monitored and logged by the provider, for example for billing purposes.

The NIST also defines three different services models for cloud computing, namely software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS) [MG+11].

Software as a service (SaaS) allows the consumer to use the provider’s application. The consumer has no control or influence over the infrastructure or the application’s capabilities, aside from being able to change the configuration.

Platform as a service (PaaS) lets the consumer deploy applications on the provider's infrastructure. The consumer has control over the applications and their hosting environment, but not the underlying infrastructure.

Infrastructure as a service (IaaS) offers the greatest level of control to the consumer. Here they have the ability to provision the provider's infrastructure and deploy any software, including operating systems.

The NIST also differentiates between four different deployment models of the cloud. Deployment models define the location of the infrastructure, and the party that controls it [MG+11].

Private cloud. The infrastructure is used exclusively by one organization. It may be managed by the organization itself, or a third party, and may be located on or off premise.

Community cloud. The infrastructure is provisioned for exclusive use by a specific community.

Public cloud. The infrastructure can be used by the general public, and can be owned by a business, academic, or government organization.

Hybrid cloud. The infrastructure is composed of a mixture between two or more of the aforementioned deployment models.

2.2 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard introduced by the Organization for the Advancement of Structured Information Standards (OASIS) to describe cloud applications and their management. TOSCA was created to address several problems in the then emerging field of cloud computing, including "(1) automated application deployment and management, (2) portability of applications and their management, and (3) interoperability and reusability of components"[BBKL14].

Over the years, several different versions have been released. The first version was released on the 25th of November 2013, with a format based on XML [OAS13]. At the time of writing, the current version is TOSCA 1.3, last updated on the 26th of February 2020. Instead of XML, this version uses YAML [OAS]. The next major version, 2.0, is currently in the drafting phase [OAS20].

In this chapter, we will introduce the key concepts of TOSCA and describe its various configuration files, the latter of which will be examined by the query language introduced in this work.

2.2.1 Service Template

A service template can include a topology template, management plans, and type definitions. The topology template describes the structure of the application, while management plans define operations related to the service, for example deploying, terminating, and managing it [BBLS12]. Type definitions are reusable blueprints for nodes and relationships.

Figure 2.1 shows an illustration of the various TOSCA components and how they relate to each other. A topology template consists of nodes connected by relationships, each of them based on a type. Types define properties and interfaces. What follows is a detailed explanation of each of the components.

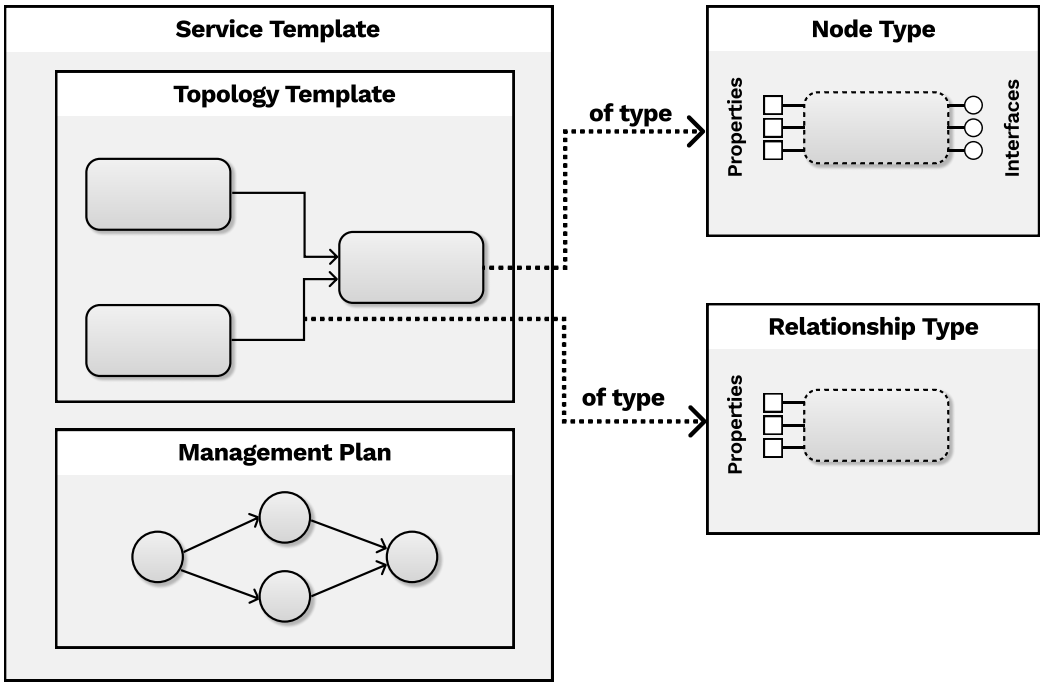

Figure 2.1: Illustration of TOSCA concepts (adapted from [BEK+16])

2.2.2 Topology Template

TOSCA describes the structure of an application as a directed, potentially disconnected graph. It consists of a set of node templates, which describe various application components, and relationship templates, which define the way in which they interact. For example, a database might have a `hostedOn` relationship with a server.

2.2.3 Node and Relationship Type

Each node template is based on a node type, which can be defined separately in order to facilitate reusability [OAS]. Node types define the properties, requirements, capabilities and supported interfaces of a node. Properties are named and typed values that can be used to provide input values to components that indicate their desired state. The actual state of a component, once instantiated, is described by attributes. Requirements are dependencies that need to be fulfilled by a capability definition of another node. Capabilities can include things like being able to host other components.

Similarly, relationship templates are based on relationship types, which define properties and interfaces of a relationship. Each capability and requirement needs to specify a relationship template.

2.2.4 Group

In order to facilitate the management of several nodes at once, TOSCA provides the ability to put multiple node templates together in a named group . For example, a policy can be attached to an entire group at once.

2.2.5 Policy

TOSCA allows attaching policies to relationships and nodes. Policies are declarative, they do not specify how and when they are evaluated [BBKL14]. Instead, the task of enforcing a policy is delegated to the orchestrator. Common use cases for policies include access control, placement, and quality-of-service [OAS]. For example, a policy may specify in what region a cloud provider should deploy the application, or how autoscaling should be managed.

2.2.6 Management Plan

Management plans can be used to formalize and automate recurring management tasks. They can involve multiple nodes, relationships, and even external services [BBKL14]. This reduces workload on developers and means that not every user has to have detailed knowledge on how to operate the application. TOSCA has a built-in language to describe workflows, but also supports other established languages like Business Process Execution Language (BPEL) and Business Process Model and Notation (BPMN) [OAS13].

2.3 YAML

YAML is a data serialization language that is designed to be easily human-readable [BEI09]. It is commonly used for storing data of applications, for example configuration files. It is also the language of choice for the latest TOSCA OASIS Standard, due to the fact that its syntax is ”much easier to read and edit than XML” [OAS]. According to the creators of YAML [BEI09], its goals are as follows:

- YAML is easily readable by humans
- YAML data is portable between programming languages
- YAML matches the native data structures of agile languages
- YAML has a consistent model to support generic tools
- YAML supports one-pass processing
- YAML is expressive and extensible

- YAML is easy to implement and use

Three dashes are used to denote the beginning of a new document, while three dots mark the end. Comments begin with a pound sign. YAML uses indentation to define the scope of elements. The most basic data type is a scalar, which describes any non-composite value like strings, integers or Booleans. Sequences, also called lists, consist of one or more entries at the same indentation level. Each entry starts with a dash and a space. Listing 2.1 shows a simple list in YAML, consisting of three scalars, in this case strings.

Listing 2.1 YAML Sequence

```
- Stuttgart
- Munich
- Frankfurt
```

It is also possible to map scalars to sequences, as seen in Listing 2.2.

Listing 2.2 YAML Scalar Mapped to Sequence

```
Cities:
    - Stuttgart
    - Munich
    - Frankfurt
```

Maps can be created by specifying a key-value pair separated by a colon, as shown in Listing 2.3.

Listing 2.3 YAML Map

```
population: 634830
country: Germany
```

YAML automatically detects data types, such as string, integer, Boolean or float. Listing 2.4 shows various scalars and their inferred types.

Listing 2.4 YAML Types

```
first:  100     # integer
second: text    # string
third:  NO      # Boolean (YAML 1.1)
fourth: 99.9    # float
```

While this feature makes files simpler to read, it also introduces a potential source of errors. The developer of `StrictYAML`, which does not have automatic type detection, recalls an incident in which the value `NO`, intended as a country code, was interpreted as a Boolean, leading to their website breaking down [OCo]. While this detection of `YES` and `NO` as Booleans has been removed in YAML

1.2, it does highlight the error potential of automatic type detection. To avoid this problem, YAML allows explicit typing by inserting two exclamation marks, followed by the name of the type, a space and the value. Listing 2.5 shows an example of this.

Listing 2.5 Explicit Typing in YAML

```
first:  !!string NO     # string
second: !!float  100    # float
```

2.4 Lexing and Parsing

One of the most important steps when executing a query is the translation of the query string into commands that a program can understand. This is accomplished by employing a lexer and a parser. *Lexing*, also called *tokenization*, describes the process of turning a sequence of input characters into tokens according to defined rules, usually based on regular expressions. For example, a lexer may find the input sequence `10 + 15` and produce the tokens `10`, `+`, and `15`, discarding the whitespace. The output of a lexer only consists of tokens, each having a value and a type, which are then passed on to the parser. The *parser* uses the tokens produced by the lexer as input for its grammar rules. The output of a parser usually consists of some kind of structural representation of the input, for example in the form of an abstract syntax tree (AST) or parse tree. It also checks if such a tree even exists, or in other words, if the input is valid and conforms to the syntactical rules defined by a grammar. The last step is semantic parsing, which converts the input to a logical form that an application can understand.

Figure 2.2 illustrates the process of lexing and parsing using a simple arithmetic term as an example. First the input is split into tokens by the lexer, then grouped by the parser.

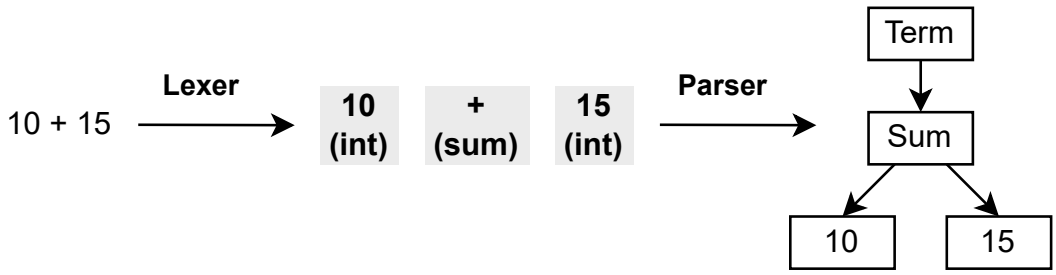

Figure 2.2: Lexing and Parsing Illustrated

2.5 Parsing Expression Grammar (PEG)

A parsing expression grammar (PEG) can be used to define a formal language by a set of rules. They were introduced by Bryan Ford in 2004 [For04]. One of the main reasons Ford cites for developing PEGs is the fact that existing solutions like context-free grammars (CFGs) contain ambiguity, which makes them suitable to express natural languages, but ”makes it unnecessarily difficult both to

express and to parse machine-oriented languages" [For04]. In contrast, PEGs use *prioritized choice* when multiple alternatives are available. This means that the first alternative that can be matched will always be chosen, removing any ambiguity.

This behavior can sometimes lead to unexpected results, since PEG parsers "greedily" match as much of the input as possible [For04]. Consider the short example grammar in Listing 2.6. The input `ab` will **not** be recognized by the grammar, instead, the parser will terminate after the first character. This happens because the parser will immediately match `a` using the first alternative of the rule and proceed to consume it from the input. In fact, the second alternative will never succeed. After the `a` is matched, it is consumed from the input, leaving only `b`, with no rule available to match it. This is the reason why generally, the most specified alternative should be listed first. If the order of the alternatives in the rule was swapped, the input would be matched as expected.

Listing 2.6 PEG Error Caused by Wrong Order of Alternatives

```
Input:
'a b'
  ^ Error! Expected 'a' or 'ab'

Grammar:
word <- 'a' / 'ab'
```

Listing 2.7 shows another example of a PEG that parses mathematical formulas that add or subtract any number of integers. The `number` rule uses a regular expression to match one or more digits. The `Term` rule either matches a term plus another term, a term minus another term, or simply a number. A list of valid sample inputs can be seen at the top.

Listing 2.7 A PEG for Simple Math Equations

```
Sample Inputs:
1 + 3 - 5
5 + 2
42

Grammar:
Exp     <- Term
Term    <- Term '+' Term / Term '-' Term / number
number  <- [0-9]+
```

3 Related Work

Query languages are specialized languages that can be used to retrieve and modify data in a database or information system [Tec16]. The syntax and semantic of a query language highly depend on the data model of the underlying system. For example, a flat relational database with a predefined schema requires a different query language than an XML file, which are often hierarchical and less predictable in their structure.

This chapter introduces query languages related to TOSCA, as well as some popular query languages used in other fields. It starts off with TOSCA_PATH in Section 3.1, followed by Unfurl expressions in Section 3.2. BiznetQL, a query language for modifying cloud application instances, is described in Section 3.3. Afterwards, Section 3.4 gives an overview of SQL, and Section 3.5 discusses XQuery. Section 3.6 introduces Cypher, followed by PGQL in Section 3.7 and GraphQL in Section 3.8. The three aforementioned languages are all graph data based. Since TOSCA topologies usually consist of multiple nodes connected by relationships, query languages that deal with graph databases are of special interest for this work. Graph databases were developed for use cases where traditional, relational databases proved insufficient. Examples include chemistry, biology, web mining, and semantic web, where the most important information are the relationships between the entities [Ang12]. A discussion of the different approaches between Queries4TOSCA (Q4T) and some of the languages presented in this chapter follows later in Section 4.7.

3.1 TOSCA Path

The specification for TOSCA version 2.0 contains a path syntax called TOSCA_PATH, which allows one to traverse nodes and relationships in order to retrieve values from other components [OAS20]. It is also possible to traverse the node graph over multiple steps. These path expressions can be used within a template to retrieve values from other parts of the document, using functions like `get_property` and `get_attribute`.

Listing 3.1 shows the grammar definition of TOSCA_PATH. It is written in Backus–Naur form (BNF), which means that vertical bars separate different choices, while sequences of rules are separated by commas. A TOSCA traversal path starts with an initial context. This can be the symbolic name of a node or relationship, or the keyword `SELF` to refer to the element that contains the path definition. Afterwards, the node graph can optionally be traversed over a variable number of steps. Starting from a node context, one can access the node's requirements and capabilities by name and select a target using their index. From a relationship context, either the source or target can be selected. The final part of a TOSCA_PATH expression is the name of a property or attribute, depending on which function it is used in.

Listing 3.1 TOSCA_PATH Grammar in BNF

```
<tosca_traversal_path> ::= <initial_context>, <node_context> |
                           <initial_context>, <rel_context>

<initial_context> ::= <node_symbolic_name> |
                      <relationship_symbolic_name> |
                      SELF

<rel_context> ::= SOURCE, <node_context> |
                  TARGET, <node_context> |
                  CAPABILITY |
                  <empty>

<node_context> ::= RELATIONSHIP, <requirement_name>, <idx_of_out_rel_in_req>, <rel_context> |
                   CAPABILITY, <capability_name>, RELATIONSHIP,
                   <idx_of_incoming_rel>, <rel_context> |
                   CAPABILITY, <capability_name> |
                   <empty>

<idx_of_out_rel_in_req> ::= <integer_index> |
                            ALL |
                            <empty>

<idx_of_incoming_rel> ::= <integer_index> |
                            ALL |
                            <empty>
```

Listing 3.2 shows a simple path expression that accesses a property directly by name. The expression can be found on the last line, and it accesses the property `name` of the node `mysql_database`.

Listing 3.2 Accessing Nodes by Name in TOSCA_PATH

```
node_templates:
    mysql_database:
        type: Database
        properties:
            name: my_database
    webapp:
        type: WebApplication
        properties:
            db_name: get_property: [ mysql_database, name ]
```

Listing 3.3 shows an example adapted from the specification [OAS20] that demonstrates how to traverse a relationship. The path expression can be found on the last line. It first selects the current relationship as the initial context, then traverses the node graph to the target and selects the `value` property.

Listing 3.3 Traversing Relationships in TOSCA_PATH

```
relationship_templates:
  my_connection:
    type: ConnectsTo
      interfaces:
        Configure:
          inputs:
            targets_value: { get_property: [ SELF, TARGET, value ] }
```

The TOSCA_PATH syntax was first proposed on the eighth of June 2022, after the start date of this thesis. However, the query language presented in this work provides additional functionality that sets it apart. For one, TOSCA_PATH can only be used from inside templates to access properties and attributes from other parts of the template. It cannot be used by a third-party application or user to access information about the template, and it is limited to returning fixed values. In contrast, the query language presented in this work can return arbitrary information that can also be reshaped, and it provides a more powerful path expression syntax with pattern matching and filters.

3.2 Unfurl Expressions

Unfurl is an orchestrator for TOSCA that comes with its own expression syntax that can be used in templates. Listing 3.4 shows the grammar according to the official documentation [One22]. It uses a path-based syntax, with each segment specifying a key of a resource or an object in a JavaScript Object Notation (JSON) or YAML document. Segments are separated by two colons in order to allow keys that contain slashes or dots.

Each segment can optionally define one or more filters. Filters can contain predicates that are applied to each value, and they can also test for the existence of a value. Additionally, they can be preceded by a negation operator in the form of an exclamation mark. At the end of a segment, a question mark operator can be used to only select the first result, to ensure that the result always consists of only a single element.

Listing 3.4 Grammar of Unfurl Expressions

```
expr    ::=  segment? ("::" segment)*
segment ::=  [key] ("[" filter "]")* ["?"]
key     ::=  name | integer | var | "*"
filter  ::=  ['!'] [expr] [("!=" | "=") test]
test    ::=  var | ([^$[]:?])+
var     ::=  "$" name
```

In addition to these grammar rules, Unfurl also includes a list of special keys that start with a dot. Table 3.1 shows a few of those keys along with their semantics.

name	country
.	Self
..	Parent
.status	Status of the instance
.parents	List of parents
.root	Root ancestor
.targets	Map of requirement names and target instances

Table 3.1: Special Keys in Unfurl Expressions

Listing 3.5 shows a snippet of a template that uses an Unfurl expression, denoted by the keyword `eval`. It first uses the special key `root` to access the root element, in this case the topology template, then retrieves the value `db_password` from the inputs.

Listing 3.5 Example of an Unfurl Expression

```
topology_template:
    inputs:
        db_password:
            type: string
    node_templates:
        my-app:
            type: WebApplication
            properties:
                db_username:
                    eval: ::root::inputs::db_password
# rest omitted for brevity
```

Like TOSCA_PATH, Unfurl Expressions can be used within templates, but are also able to be executed by other applications by calling an API with an `eval` function. Due to the fact that they are integrated into the Unfurl orchestrator, they are able to access some intrinsic properties like the status of the instance, which is not possible with our implementation. They also have some special keys that are not available in Q4T, for example `targets`, which allows one to get a map of requirements with their target instances. On the other hand, Q4T allows some operations that are not possible with Unfurl Expressions, including the ability to combine filters with a Boolean `OR`. Multiple filters can be chained in Unfurl, but they all need to be true at the same time, equivalent to a Boolean `AND`. Furthermore, Q4T has the ability to define custom key-value pairs to include in the return object, while Unfurl Expressions only allow the specification of an array index or the aforementioned question mark operator to return the first object. In Q4T, one can use an array access to index zero as an equivalent to Unfurl's question mark operator when only a single result is desired.

3.3 BiznetQL

BiznetQL is a proposed query language to manage cloud computing infrastructure resources [IBHS18]. Since the original work is available in Indonesian only, the analysis presented here is based on a machine translation, which may not be wholly accurate. BiznetQL aims to support create, read, update and delete (CRUD) operations. Additional goals include being reusable, scalable, maintainable, and testable. Syntactically, it is very similar to SQL, sharing the same basic structure and keywords.

What follows are two examples from the original work [IBHS18]. Listing 3.6 shows a `SELECT` statement in BiznetQL, using the wildcard operator to select all resources, then specifying the name of the instance in a `FROM` statement.

Listing 3.6 SELECT Statement in BiznetQL

```
SELECT * FROM neo.vm
```

Listing 3.7 shows an `UPDATE` statement to modify a virtual machine. Similar to SQL, the point at which the updated value should be inserted is chosen using a `WHERE` statement.

Listing 3.7 UPDATE Statement in BiznetQL

```
UPDATE neo.vm SET 'flavor'='SM4.4'
WHERE 'name'='vm-test'
```

Compared to our language, BiznetQL contains no pattern matching ability, but it has the advantage of supporting all CRUD operations. It is also unclear whether some form of path expressions are included, for example to access the properties of a certain node or list its requirements.

3.4 SQL

Structured Query Language (SQL) is the most commonly used database query language, and it is being supported by a huge number of database applications [GWO02]. SQL allows the data management of relational databases. It supports queries, data manipulation in the form of insertion, deletion and updating, as well as defining and modifying the schema of a database. This section will introduce the basic concepts of SQL queries. Table 3.2 shows some sample data. The table is called `customers` and contains four columns and five records.

A SQL statement always starts with a keyword. Some of the most common keywords include `SELECT` for querying data, `UPDATE` for modifying an entry, `WHERE` to add filters to a query, or `INSERT` for adding new data to a table. Of special interest to us is the `SELECT` statement that is used to run queries on a database. It always starts with the `SELECT` keyword, follow by a list of columns to retrieve, or an asterisk that returns all columns. Afterwards, the keyword `FROM` is used to select the table from

id	name	city	country
1	Max	Stuttgart	Germany
2	David	Munich	Germany
3	John	New York	United States
4	Rafael	Madrid	Spain
5	James	London	UK

Table 3.2: SQL Example Database

which the data should be retrieved. Listing 3.8 shows a simple query that retrieves all values in the `name` and `country` columns from our example table. The result is a new table that only consists of the columns that were specified, as can be seen in Table 3.3.

Listing 3.8 Simple Query Using SQL

```
SELECT name, country
FROM customers;
```

name	country
Max	Germany
David	Germany
John	United States
Rafael	Spain
James	UK

Table 3.3: Result of the Above Query

`WHERE` statements can be used to filter results. Only entries that satisfy the specified condition are returned. `WHERE` statements are not limited to `SELECT` statements, they can also be used to identify data following `DELETE` or `UPDATE` statements. In addition to the standard comparison operators (=, >, <, >=, <=, <>), SQL also supports `BETWEEN`, `LIKE`, and `IN`. `BETWEEN` can be used to check if a value is within a range. `LIKE` matches patterns in strings, and `IN` is used to check for equality to multiple possible values. `WHERE` statements can include the Boolean operators `AND`, `OR`, and `NOT` to filter records based on multiple conditions.

Listing 3.9 shows a query that returns all columns from our `customer` table, but only records where the value of `country` is equal to `Germany`. The result can be seen in Table 3.4.

Listing 3.9 WHERE Statement in SQL

```
SELECT *
FROM customers
WHERE country='Germany';
```

id	name	city	country
1	Max	Stuttgart	Germany
2	David	Munich	Germany

Table 3.4: SQL WHERE Statement Result

SQL is a powerful query language for relational databases. It includes a wealth of functionality that our query language does not have, for example subqueries, functions like `avg` or `count`, joins, and many more. However, SQL is intended to be used in relational databases that have a flat data model, whereas TOSCA templates have a hierarchical structure. It would be possible to spread the various parts of a TOSCA template into several tables, however, this would often lead to complex queries involving multiple joins.

3.5 XQuery

XQuery is a query language for Extensible Markup Language (XML) files. XML is a markup language for storing and transporting any data. XML files differ from relational databases in some key aspects, which means that using SQL would not be viable and a specialized query language was required, which lead to the development of XQuery in 2003 [CFR+03]. One of those differences is the fact that XML, unlike relational databases, allows data to be arbitrarily nested, so a query language needs to be able to navigate this hierarchy. Another important difference is the fact that relational databases have a value in every field, which may be null, while XML files do not have a fixed structure, which means that an element that is being searched for may not even exist. XQuery is a superset of XPath and makes use of its selection capabilities [RCDS11]. In addition to that, it also offers operations like joins and grouping. The query in Listing 3.10 shows a simple XPath expression that returns all `employee` elements anywhere in a document that have a `salary` element with a value greater than 50,000.

Listing 3.10 XPath Example

```
//employee[salary>50000]
```

Listing 3.11 shows a FLWOR expression that will return the same result as the previous XPath query. FLWOR stands for ”FOR, LET, WHERE, ORDER BY, RETURN”, describing the operations that are possible.

Listing 3.11 XQuery FLWOR Example

```
for $x in //employee
where $x/salary>50000
return $x
```

XQuery provides a good way to access hierarchical documents in the form of XPath expressions, which is why we are basing our path syntax on it. However, it provides no pattern matching ability, necessitating the addition of a pattern matching syntax in our query language.

3.6 Cypher Query Language

Cypher was first developed as the query language for Neo4j[1]. Neo4j is an open-source graph database that started out as being purely Java-based, but now supports a wide range of programming languages and frameworks [VWA+15]. However, Cypher is not only limited to Neo4j. Starting in 2015, the openCypher project is working on establishing Cypher as a standardized language for querying property graph databases [FGG+18].

Since Cypher is used for graph databases, representing nodes and their relationships plays an important role. Nodes are represented with parentheses that optionally contain a variable, a label, or both. Variables can be used to refer to the same node in other parts of the query. Labels are used to group similar nodes together. Listing 3.12 shows a simple query that finds a single node. The `MATCH` statement matches all nodes of type `Person` that have a property called `name` with a value `Max Mustermann`. Nodes that match this pattern are then bound to a variable `p`, which is used in the `RETURN` statement to output the address variable of that node.

Listing 3.12 A Simple Query in Cypher

```
MATCH (p:Person {name: "Max Mustermann"})
RETURN p.address
```

Relationships are represented by arrows between nodes. If the direction of the relationship is of no concern, two dashes can be used instead. Additionally, the type of relationship can be placed in square brackets. Similar to nodes, relationships can also be given variables and filters on labels. Listing 3.13 shows an example of a pattern involving a relationship. The `MATCH` statement finds all persons in the database that live in Stuttgart.

Listing 3.13 Query with Relationship in Cypher

```
MATCH (p:Person) -[:LIVES_IN]-> (:city {name: "Stuttgart"})
RETURN p.name
```

Cypher has a list of several reserved keywords that have special meanings. What follows is a list of some of the most important ones.

- `MATCH` is used to search for an existing node using a pattern. A pattern describes one or more nodes connected by relationships.
- `WHERE` can be used as part of a `MATCH` statement to impose further constraints on the result.

[1] https://neo4j.com/

- `RETURN` specifies the contents of the result set. It can return nodes, relationships, properties, and patterns.

Cypher provides a clear and intuitive syntax to express relationships between nodes, including ones with multiple steps. For this reason, we are adapting its syntax to express patterns in topology templates, with some adjustments. Cypher does not allow maps or heterogeneous lists as properties [Neo22], however, TOSCA nodes have complex elements, for example properties are maps and requirements are lists. Therefore, we are allowing paths instead of values inside `MATCH` statements, so we can for example access a specific property of a node inside a filter without having to add additional nodes for each step.

3.7 PGQL

PGQL is a query language that "combines graph pattern matching with SQL-like syntax"[RHK+16]. When describing patterns, vertices are surrounded by parentheses and can be optionally given a variable and a label. Labels are preceded by a colon. If a label is defined, only vertices with that label will match the query. Edges are surrounded by square brackets. Like vertices, they can be given a variable and a label. Left and right of the brackets, dashes or arrows denote directed or undirected edges.

Listing 3.14 shows a sample query in PGQL that selects the names of all friends of a person named `Peter`. The first vertex named `p` needs to have a `PERSON` label, and it needs to be connected by an edge with the label `FRIENDS_WITH` in the direction of vertex `friend`, which also needs to have a `PERSON` label. The following `WHERE` statement filters the starting node to only include persons named `Peter`.

Listing 3.14 Query with Filters in PGQL

```
SELECT friend.name
FROM MATCH (p:PERSON) -[:FRIENDS_WITH]-> (friend:PERSON)
WHERE p.name = 'Peter'
```

In addition to the fixed-length paths introduced above, PGQL can also match paths with variable lengths. Path finding goals can be defined to match paths with different properties, like finding the shortest, cheapest, or any path.

Similar to Cypher, PGQL does not allow complex values as node properties, which is why we made adjustments to the node syntax for our query language.

3.8 GraphQL

GraphQL is a query language for APIs that can be used to retrieve and manipulate data. Internal development at Facebook (now Meta) started in 2012, before publicly releasing in 2015 [Byr15]. GraphQL provides a way to access web-based data similar to REST. As the name implies, it works

on a graph data model, where users define the schema of their data in the form of different connected nodes. GraphQL queries request specific fields from objects. Listing 3.15 shows a simple POST request that asks for the `name` field of all students in a database.

Listing 3.15 Simple Query in GraphQL

```
{
  student {
    name
  }
}
```

As can be seen in Listing 3.16, we get back the names of all student objects in the database.

Listing 3.16 Result of the Previous GraphQL Query

```
{
  "data": {
    "student": {
      "name": "Max Mustermann"
    },
    {
    "student": {
      "name": "Albert Einstein"
    },
  }
}
```

GraphQL also allows passing arguments to fields, providing a way to filter the desired data. The following Listing 3.17 shows a request for all grades of a student with an `id` of `123654`. We also make a sub-selection for the `grades` objects of this student and request the fields `course` and `grade` of each one.

Listing 3.17 Query with Filter in GraphQL

```
{
  student(id: "123654") {
    name
    grades {
        course
        grade
    }
  }
}
```

As can be seen in Listing 3.18, the name of the student is returned, as well as all sub-objects of the `grades` object along with the two fields we requested.

Listing 3.18 Result of Query with Filter

```
{
  "data": {
    "student": {
      "name": "Albert Einstein",
      "grades": [
        {
          "course": "Math",
          "grade": "1.0"
        },
        {
          "course": "Physics",
          "grade": "1.3"
        }
      ]
    }
  }
}
```

GraphQL allows a user to formulate queries in an intuitive way and works on data graphs like topology templates in TOSCA. However, due to the use of line breaks and indentation to denote structure, queries take up a lot of space and span multiple lines. Part of our motivation is including queries as part of TOSCA templates, where GraphQL queries would severely impact the readability of the document due to their space requirements.

4 Queries4TOSCA Language Specification

This chapter introduces Queries4TOSCA (Q4T), a query language that can be used to retrieve data from one or more TOSCA service templates based on user-defined conditions, with the possibility to include attributes from running instances. Additionally, query expressions can be used inside of a template to access values from other parts of it.

Section 4.1 lists all functional and non-functional requirements. Section 4.2 then introduces the running example that will be used in various parts of this chapter to show queries and their results. Afterwards, a quick introduction based on various examples that highlight different functionalities is presented in Section 4.3, followed by an explanation of all statements in Section 4.4 and the Paths4TOSCA syntax in Section 4.5. Section 4.6 describes the process of matching patterns in the topology. Section 4.7 presents various alternative languages that were considered during the creation of this work and discusses their strengths and weaknesses.

4.1 Requirements

This section lists both functional and non-functional requirements that should be fulfilled by the TOSCA query language presented in this work and its implementation.

Human-Readable Queries (R1): Queries should be easy to understand by a human.

Declarative Language (R2): The query language should be declarative, meaning that the user only needs to specify what data they want to retrieve, while the underlying implementation decides how the query is executed.

Compact Syntax for Inline Queries (R3): The query language should include a path-based syntax to access individual elements. This syntax should be able to be written in a single line.

The following requirements relate to the static analysis of TOSCA template files. The query language must be able to return any of these values and offer the ability to filter templates based on their contents.

Querying Service Templates (R4): The user should be able to query service templates based on certain variables. These include:

1. TOSCA definitions version
2. Metadata
3. Description

The query language should be able to query service templates based on properties of their topology templates. It must be able to return any of these values, and apply filters based on their contents. Additionally, it must be possible to retrieve information about node templates contained in a group or targeted by a policy.

Querying Topology Templates (R5): It should be possible to query topology templates according to the following criteria:

1. Description
2. Groups
3. Policies

The query language should be able to query topology templates based on the node and relationship templates contained within. It must be possible to retrieve any of the values below, and run filters based on their contents. Additionally, it must be possible to traverse capabilities and requirements of nodes, meaning that information about their source, targets, or the relationship itself can be retrieved.

Querying Node Templates (R6): The user should have the ability to query node templates based on certain criteria. This includes the following information:

1. Type
2. Name
3. Description
4. Metadata
5. Properties
6. Attributes
7. Interfaces
8. Requirements
9. Capabilities

Querying Relationship Templates (R7): The user should have the ability to query relationship templates based on the following criteria:

1. Type
2. Name
3. Description
4. Metadata
5. Properties
6. Attributes
7. Interfaces

4.2 Running Example

All sample queries presented in this chapter will use the following running example, which represents a web application connected to a database. The name of the template is `my-app`. Listing 4.1 shows a textual representation written in YAML, while Figure 4.1 shows a visual interpretation of the node graph contained within the service template. The service consists of a web application using a Tomcat web server, which runs on a virtual machine hosted on an OpenStack instance. The web application also connects to a database hosted on a DBMS, which runs on a separate virtual machine. The second virtual machine runs on the same OpenStack instance as the web application.

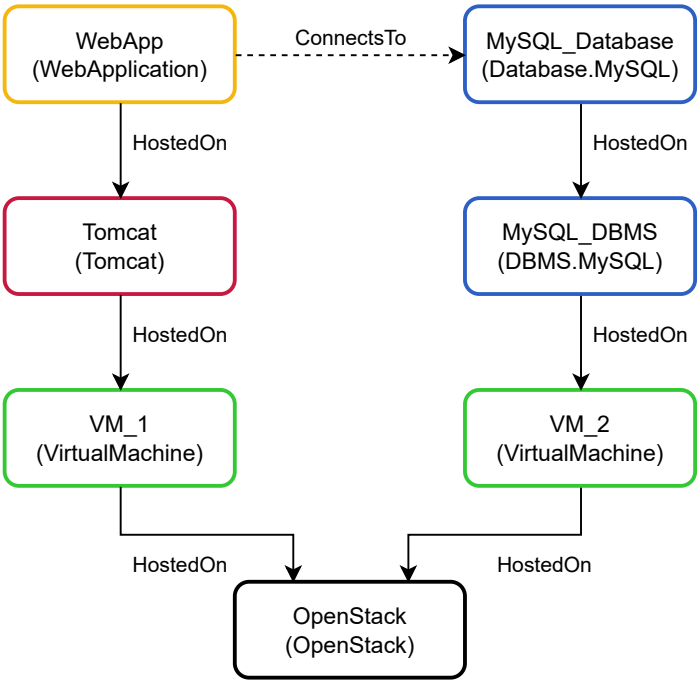

Figure 4.1: Visualization of the Running Example

Listing 4.1 Topology of the Running Example in YAML

```
tosca_definitions_version: tosca_simple_yaml_1_3

topology_template:
    node_templates:
        webapp:
            type: WebApplication
            properties:
                db_username: { get_property: [ mysql_database, username ] }
                db_password: { get_property: [ mysql_database, password ] }
                port: 3306
            requirements:
                - database_endpoint: mysql_database
                - host: tomcat
        tomcat:
            type: Tomcat
            requirements:
                - host: vm_1
        mysql_database:
            type: Database.MySQL
            properties:
                username: dbuser
                password: dbpwd
            requirements:
                - host: dbms
        mysql_dbms:
            type: DBMS.MySQL
            requirements:
                - host: vm_2
        vm_1:
            type: VirtualMachine
            properties:
                num_cpus: 2
                mem_size: 4 GB
                operating_system: Ubuntu 22.10
            attributes: # inherited from node type, not defined as part of template
                ip_address: 127.0.0.1
            requirements:
                - host: openstack
        vm_2:
            type: VirtualMachine
            properties:
                num_cpus: 2
                mem_size: 4 GB
                operating_system: Ubuntu 22.10
            attributes: # inherited from node type, not defined as part of template
                ip_address: 127.0.0.1
            requirements:
                - host: openstack
        openstack:
            type: OpenStack
            properties:
                ip_address: 127.0.0.1
```

4.3 Getting Started

This section provides a quick introduction to the basic structure of a query by showing various examples. A full description of the syntax follows in the next section.

4.3.1 Basic Examples

The simplest way to query a service template is to directly access an element. We specify the name of our template with a `FROM` statement, then use a `SELECT` statement to access the node template called `webapp` directly by its path. The query in Listing 4.2 will return all values contained within `webapp`. The result can be seen in Listing 4.3.

Listing 4.2 Selecting a Node with Q4T

```
FROM templates.my-app
SELECT node_templates.webapp
```

Listing 4.3 Result of Selecting a Node

```
type: WebApplication
properties:
    db_username: { get_property: [ mysql_database, username ] }
    db_password: { get_property: [ mysql_database, password ] }
    port: 3306
requirements:
    - database_endpoint: mysql_database
    - host: tomcat
```

Instead of searching elements of the template by their name, we can also filter them by their contents. In the example presented in Listing 4.4, we search all node templates of type `VirtualMachine`, then include their name in the result set, which can be seen in Listing 4.5. Since the result consists of multiple elements, an array is returned.

Listing 4.4 Using Filters in Q4T

```
FROM templates.my-app
SELECT node_templates.*[type='VirtualMachine'].name
```

Listing 4.5 Result of Using Filters in Q4T

```
- vm_1
- vm_2
```

4.3.2 Working with Return Structures

As can be seen in the last few examples, queries return all values at the current context. If we want to have more control over the shape of our output data, we can use curly braces to define key-value pairs to return. Key-value pairs are separated by a colon, and each part can consist of either a literal or an expression. In the example in Listing 4.6, we select the `openstack` node template, then return an object consisting of its name and IP address, defining our own key names. The result can be seen in Listing 4.7.

Listing 4.6 Using a Return Structure in Q4T

```
FROM instances.my-app
SELECT node_templates.openstack{'Host Name': name, 'IP Address': properties.ip_address}
```

Listing 4.7 Result of Using a Return Structure

```
Host Name: openstack
IP Address: 127.0.0.1
```

Listing 4.8 shows an example of a return structure specifying an array that lists the name of all node templates as well as their type. In this case, both key and value are expressions due to the lack of quotes, so they will both be evaluated for each individual node. The result of this query can be seen in Listing 4.9.

Listing 4.8 Return Structure with Pair of Values

```
FROM templates.my-app
SELECT node_templates.*{name: type}
```

Listing 4.9 Result of Return Structure with Pair of Values

```
- webapp: WebApplication
- tomcat: Tomcat
- mysql_database: Database.MySQL
- dbms: DBMS.MySQL
- vm_1: VirtualMachine
- vm_2: VirtualMachine
- openstack: OpenStack
```

4.3.3 Working with Pattern Matching

Not only can we query individual nodes, but we can also search for nodes based on their relationships to other nodes. The query presented in Listing 4.10 uses a `MATCH` statement to find a pattern in the topology template, then returns all nodes that match it. The pattern we are looking for consists of

all node templates that are hosted on `openstack`. Therefore, we first define an anonymous node with a filter that only includes nodes named `openstack`. Note that in `MATCH` statements, any names that we give to a node are only used as aliases, so we need to use filters to select a specific node instead of writing out its path. Next, we draw an incoming arrow originating from a node that we give the alias `node`. Inside the relationship, we add a filter in curly braces that only allows relationships named `host`. After matching the pattern, we add a `SELECT` statement to return all nodes that match our alias `node`, which gives us the result in Listing 4.11. We get back the nodes `vm_1` and `vm_2`, since they are both hosted on `openstack`.

Listing 4.10 Using Pattern Matching to Find All Nodes Hosted on Specific Node

```
FROM templates.my-app
MATCH ([name='openstack'])<-{[name='host']}-(node)
SELECT node
```

Listing 4.11 Result of Finding All Hosted Nodes

```
vm_1:
    type: VirtualMachine
    properties:
        num_cpus: 2
        mem_size: 4 GB
        operating_system: Ubuntu 22.10
    requirements:
        - host: openstack
vm_2:
    type: VirtualMachine
    properties:
        num_cpus: 2
        mem_size: 4 GB
        operating_system: Ubuntu 22.10
    requirements:
        - host: openstack
```

When examining relationships over multiple hops, instead of writing out the full path, we can also specify the length of a path in the relationship. Listing 4.12 shows an example of a query that traverses the entire hosting stack of `webapp`. Using an asterisk in the relationship declaration means that paths of any length are traversed, but it is also possible to specify an exact length, a minimum, maximum, or a range after the asterisk. The result seen in Listing 4.13 consists of the names of all nodes that match the `target` node in the pattern.

Listing 4.12 Using Variable-Length Pattern Matching

```
FROM templates.my-app
MATCH (webapp[name='webapp']-{[name='host']*}->(target)
SELECT target.*.name
```

Listing 4.13 Result of Using Variable Length Pattern Matching

```
- tomcat
- vm_1
- openstack
```

4.3.4 Advanced Example

In this scenario, a user is trying to find the IP address of the component `webapp`. Since it is an application that is hosted on a server, it does not contain its IP address directly. Instead, it is an attribute of the `VM_1` component, which the web application is connected to via a `hostedOn` relationship. This means that in order to retrieve the IP address, a query needs to traverse this relationship, then return the IP address from the target component. Figure 4.2 shows a visual representation of the desired information, as well as the actual location of the information within the template.

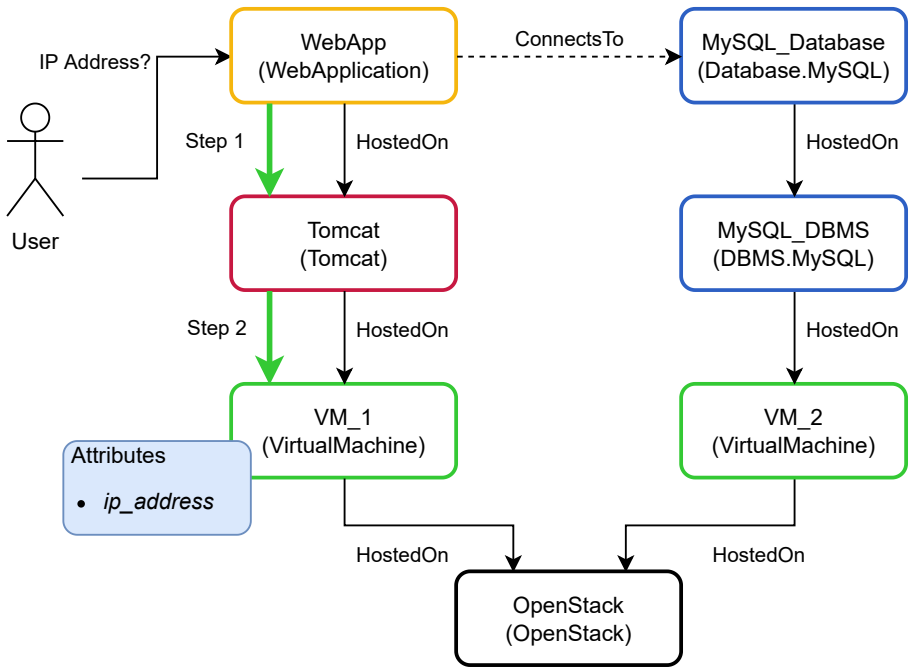

Figure 4.2: Visualization of This Example

To solve this task, we first use a `FROM` statement to choose our desired instance, in this case `my-instance`. Next, we use a `MATCH` statement to find a pattern in our template. We start out with an anonymous node that filters the name `webapp` as our starting point. From there, we define an outgoing *host* relationship with an asterisk, meaning that multiple hops will be matched. We name the node on the other end `host` so we can use it as an output later, and we add a filter for the type `VirtualMachine`, to make sure we only access the virtual machine component, not any components in between. Lastly, in a `SELECT` statement, we chose the public address attribute of our matched `host` node.

Listing 4.14 shows the full query to solve this scenario. As can be seen in Figure 4.2, the query first identifies the webapp node, then traverses the hostedOn relationship, marked step 1. Once it arrives at tomcat, it compares its type against VirtualMachine. This comparison fails, so the next hostedOn relationship in the outgoing direction is traversed, marked step 2. This yields the node vm_1 where the comparison succeeds, which means that the value of attributes.ip_address is retrieved and returned.

Listing 4.14 Finding the IP Address of a Virtual Machine with Q4T

```
FROM instances.my-instance
MATCH ([name: 'webapp'])-{[name='host']*}->(host[type='VirtualMachine'])
SELECT host.*.attributes.ip_address
```

Listing 4.15 shows the result of this query, containing value of the IP address.

Listing 4.15 Result of Finding the IP Address of the Host of Webapp

```
127.0.0.1
```

4.4 Statements

Q4T queries consist of various statements beginning with a keyword. What follows is an explanation of all statements and their possible contents.

4.4.1 FROM

A FROM statement is used to denote which templates or instances the query should be executed on. It starts with the literal FROM, followed by a space and the word templates, which is an abbreviation for service templates, or instances. Then, either a file path needs to be provided, or an asterisk can be used to select all service templates or instances, respectively. Listing 4.16 shows the PEG of a FROM statement.

Listing 4.16 Grammar of a FROM Statement

```
From = "FROM" ("instances" | "templates") ("/" | ".") ("*" | filePath)
```

4.4.2 SELECT

SELECT statements are used to select elements from a template. They are denoted by the keyword SELECT, followed by one or more path expressions separated by a comma. Path expressions are described in detail in Section 4.5. They can start with the special keywords Group or Policy, the

name of an element, or a dot to select everything. Afterwards, a series of mapping steps, filters, or array accesses can be used. Lastly, there is an optional return structure (see Section 4.5.5). Listing 4.17 shows the grammar of a `SELECT` statement.

Listing 4.17 Grammar of a SELECT Statement

```
Select = "SELECT" Path ("," Path)*
Path   = (Group | Policy | Step | ".") (ArrayAccess | Map | Filter)* ReturnClause?
```

4.4.3 MATCH

`MATCH` statements are used to search for patterns inside the topology of a service template. A pattern consists of at least one node, along with any number of additional nodes and relationships, as seen in Listing 4.18. Section 4.6 describes the syntax for denoting nodes and relationships.

Listing 4.18 Grammar of a MATCH Statement

```
Match = "MATCH" Node (Relationship Node)*
```

A `MATCH` statement needs to be followed up by a `SELECT` statement to specify which variables defined in the pattern should be part of the output.

4.4.4 Comments

It is possible to insert comments into queries. All comments are discarded at the parsing stage, they are only intended to be read by other developers and have no influence on the execution of the query. Single-line comments begin with two forward slashes and extend to the end of the current line. Multi-line comments begin with a forward slash and an asterisk and end with another asterisk and slash and can be inserted anywhere. Listing 4.19 shows an example for each type of comment.

Listing 4.19 Using Comments

```
// single-line comment
/* multi-line
comment */
```

4.5 Paths4TOSCA

Since elements of service templates can be nested arbitrarily deep, this query language needs a way to express paths to access them. Therefore, a path expression syntax is introduced that allows navigating through parts of a service template. All parts of a path are explained below.

4.5.1 Path Expressions

Path expressions can be used to select elements inside a template. They consist of the names of elements separated by a dot, which acts as the path separator. Elements that are part of the topology template, for example node templates, can be accessed directly, without specifying `topology_template` in the path. Aside from accessing elements directly by name, several other operators are available. The wildcard operator is written with an asterisk and selects all children of the preceding element. The special functions `GROUP()` and `POLICY()` can be used at the start of a path expression and return all node templates that belong to a TOSCA group or are targeted by a policy, respectively. The keyword `SELF` can only be used in queries contained inside templates. It selects the current element which contains the query.

Paths can also use various shortcuts to make it easier to access certain elements. Table 4.1 gives an overview of available shortcuts and their meanings.

Shortcut	Meaning
@	attributes
#	properties
$	requirements
%	capabilities

Table 4.1: Shortcut Expressions

Listing 4.20 shows various examples of path expressions using the aforementioned elements.

Listing 4.20 Examples of Path Expressions

```
node_templates.webapp    // Selecting a node directly by name
node_templates.webapp.@  // Selecting attributes of webapp
node_templates.*         // Selecting all nodes
GROUP(my-group)          // Selecting all nodes in group 'my-group'
POLICY(my-policy)        // Selecting all nodes targeted by policy 'my-policy'
```

4.5.2 Predicates

Predicates are used to filter results based on user-defined conditions. Only elements for which the expression inside the brackets evaluates to `true` are included in the result set. Strings need to be surrounded by single or double quotes and may optionally use regular expressions to find multiple possible matches. An exclamation mark in front of an expression can be used to negate its result. Table 4.2 shows all available comparison operators along with their semantics.

Additionally, if a filter only consists of a single variable with no comparison operator, it will return `true` if the current element has a child element with the same name. Listing 4.21 shows various examples of filtering nodes using different conditions, with the corresponding semantics explained in a line comment.

Operator	Meaning
=	Equality
!=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
=~	Matches regular expression
!	Negation

Table 4.2: Comparison Operators

Listing 4.21 Examples of Path Expressions with Filters

```
node_templates.*[type='VirtualMachine']   // equality
node_templates.*[name!='vm_1']            // inequality
node_templates.*[name=~'^vm_']            // regular expression
node_templates.*[properties]              // existence of field properties
```

4.5.3 Array Access

Some elements of service templates are not maps, but arrays. This means that their values are not accessed by key names, but zero-based indices, which is why an additional syntax is introduced. Accessing an element of an array can be accomplished by putting an integer inside square brackets after the path of the array. If the value inside the brackets is an integer, it is interpreted as an array access, otherwise, it is interpreted as a filter. Attempting to access non-existent array indices will return an empty result. Listing 4.22 shows two examples of selecting array elements.

Listing 4.22 Examples of Accessing Arrays

```
node_templates.*[0]                    // Selecting the first node template
node_templates.webapp.requirements[1]  // Selecting the second requirement of webapp
```

4.5.4 Boolean Operators

Boolean operators can be used in predicates to link two or more conditions. Boolean AND returns `true` if both predicates evaluate to `true`, Boolean OR returns `true` if at least one of the predicates evaluates to `true`. Listing 4.23 shows an example for each Boolean operator.

Listing 4.23 Examples of Using Boolean Operators

```
node_templates.*[type='VirtualMachine' AND name='vm_1']
node_templates.*[name='vm_1' OR name='vm_2']
```

4.5.5 Return Structures

By default, the output that is returned will always consist of the value at the current context specified by the path expression. However, it is possible to define the content of the returned data. This can be done by putting curly braces with comma-separated key-value pairs at the end of a path expression. Both key and value can either be a literal or a variable. When a variable is used as a key, it needs to evaluate to a string. Instead of a key-value pair, it is also possible to only specify the name of a value, in which case it will automatically be used as the name for the key. Listing 4.24 shows various examples of using return structures. The expression in the first line returns a list of objects comprised of the keys `Node Name` and `Node Type` mapped to the names and types of individual nodes. The second expression also returns a list of objects, but their keys are named directly after the values, namely `name` and `type`. The expression in the last line returns a list of objects consisting of only a single key-value pair, with the name of each individual node as the key, and the corresponding type as the value.

Listing 4.24 Examples of Using Return Structures

```
node_templates.*{'Node Name': name, 'Node Type': type}  // Using custom key names
node_templates.*{name, type}                            // Short form
node_templates.*{name: type}                            // List of node names mapped to type
```

4.6 Patterns

Another important aspect of this query language is the ability to query nodes based on their relationships, for example traversing the entire hosting stack of an application to find the IP address of its underlying host. Therefore, a way to match patterns in the topology of a template is introduced. A pattern matching statement begins with the keyword `MATCH`, followed by any number of nodes connected by relationships.

Nodes are denoted by a pair of parentheses. Inside those parentheses, a variable name can be given to the node. Otherwise they are anonymous and cannot be referenced in a `SELECT` statement. Listing 4.25 shows an example of an anonymous node and one that is assigned a variable.

Listing 4.25 Simple Node Declarations

```
()   // anonymous node
(n)  // node with variable n
```

Nodes can optionally contain a filter in square brackets that allows the same predicate syntax described in Section 4.5. Selecting a node template can be seen as the equivalent of a `SELECT` statement that implicitly starts at the path `node_templates.*`. Listing 4.26 shows examples of nodes with filters.

Listing 4.26 Adding Filters to Nodes

```
([type='VirtualMachine'])    // anonymous node with filter
(n [type='VirtualMachine'])  // node with variable n and filter
```

Between nodes, relationships can be specified. They are connected to nodes via dashes or arrows to denote undirected or directed relationships, respectively. An incoming relationship means that the requirement of another node is fulfilled by a capability of the current node, while an outgoing relationship means that a requirement of the current node is fulfilled by the capability of the other node. An undirected relationship applies to both scenarios. Listing 4.27 shows all possible relationship directions.

Listing 4.27 Simple Relationship Declarations

```
(a)-->(b) // a has requirement fulfilled by capability of b
(a)<--(b) // a has capability that fulfills requirement of b
(a)--(b)  // a and b have any relationship
```

Like node templates, relationships can be given a variable name, and their types can be specified using the same filter syntax. To achieve this, they need to be surrounded by curly braces inserted in the middle of the arrow, as can be seen in Listing 4.28.

Listing 4.28 Relationships with Variables and Filters

```
(a)-{r}->(b)                // relationship with variable r
(a)-{r [name='host']}->(b)  // relationship with symbolic name host and variable r
```

It is also possible to search for node templates connected over multiple relationships by specifying a cardinality. This can be accomplished by putting an asterisk at the end of a relationship, followed optionally by a number or a range. If both are omitted, relationships of any length will be matched. Listing 4.29 shows examples of variable length patterns.

Listing 4.29 Declaring Variable-Length Patterns

```
(a)-{*2}->(b)     // exactly two hops between a and b
(a)-{*2..5}->(b)  // between two and five hops
(a)-{*2..}->(b)   // at least two hops
(a)-{*..5}->(b)   // at most five hops
(a)-{*}->(b)      // any amount of hops
```

`MATCH` statements return a map of variable names that each contain all node templates that match that particular variable. They need to be followed up by a `SELECT` statement that specifies which values to return. The following is a complete query involving a `MATCH` statement.

Listing 4.30 Selecting Return Values when Using Pattern Matching

```
FROM templates.my-app
MATCH (a)-{*}->([type='VirtualMachine'])
SELECT a.*
```

4.7 Discussion

Developing a query language for TOSCA proved to be a unique challenge, because we are interested in matching patterns in the node graph, but also have a path-based syntax to access elements in a more concise way, for example when using queries within a template. This section highlights some of the alternatives that were considered during the development of this work, describes their strengths and weaknesses, and explains the reasoning behind choosing the final syntax.

4.7.1 GraphQL

Using GraphQL to formulate queries was one of the considered alternatives. Listing 4.31 shows an example of what a GraphQL request for the data of a node would look like, while Listing 4.32 shows the response. In this query, we retrieve the `type` and `requirements` fields of a specific node template, by specifying a filter that only includes nodes named `webapp`.

Listing 4.31 A Potential Query with A GraphQL-Based Language

```
{
  node_template(name: "webapp") {
    type
    requirements
  }
}
```

Listing 4.32 A Possible Response to the Above GraphQL Query

```
{
  "data": {
    "webapp": {
        "type": "WebApplication"
        "requirements": [
            {"database_endpoint": "mysql_database"},
            {"host": "tomcat"}
        ]
    }
  }
}
```

One of the advantages of GraphQL is the relative simplicity of the query syntax. The above example can be readily understood, even by someone with no prior knowledge. It also provides a convenient way of specifying the contents of the response object. Additionally, since GraphQL is designed around data graphs, it can express connections between nodes. A disadvantage of using GraphQL, at least in our case, is the fact that it uses line breaks and indentation to give structure to queries. This is problematic since part of our motivation is the ability to include queries within service templates, where GraphQL's large, multi-line queries would severely impact the readability of the template, contradicting requirement **R3**. Writing a GraphQL query in a single line would not be a viable solution either, since this would make queries difficult to comprehend, violating requirement **R1**.

4.7.2 Cypher

Topology templates in TOSCA can be thought of as graphs, with nodes as vertices and relationships as edges. This begs the question as to whether it would be possible to simply use Cypher (Section 3.6) or PGQL (Section 3.7). Listing 4.33 shows what a query on a service template in Cypher could look like, with Listing 4.34 showing the matching response. As can be seen in this example, this approach would be well suited to query the topology template. Due to the ASCII-like syntax, relationships between nodes are visualized in a simple and easily understandable manner.

Listing 4.33 A Query on a TOSCA Service Template in Cypher

```
MATCH (:Node_Template {name: 'webapp'})-[:HOST]->(node:Node_Template)
RETURN node.name, node.type, node.requirements
```

Listing 4.34 The Response to the Above Cypher Query

```
node.name: webapp
node.type: WebApplication
node.requirements: [
    database_endpoint: mysql_database
    host: tomcat
]
```

The downside of using Cypher is the fact that while a graph structure is suitable to represent the topology template, it might be difficult to represent other parts of a service template without a path syntax. In Cypher, complex data structures like maps and lists cannot be stored as properties of nodes [Neo22]. This means that parts of a node template like its properties would need to be translated to other nodes, and nested properties would also need to have a separate node for each nesting level. Listing 4.35 exemplifies this problem by showing a query that accesses a single property of a node template. In cases like this, a simple path-based syntax would suffice, without the need to write everything out as nodes.

Listing 4.35 Accessing a Node Property in Cypher

```
MATCH (:Node_Template {name: 'vm_1'})-->(cpus:Property {name: 'num_cpus'})
RETURN property.value
```

Listing 4.36 shows an equivalent query in Q4T, which is much shorter.

Listing 4.36 Accessing the Same Node Property in Q4T

```
SELECT node_templates.vm_1.properties.num_cpus
```

Additionally, Cypher does not have `FROM` statements to specify a data source, so a way to choose the desired template or instance would need to be added.

4.7.3 Path-Based Syntax with Additional Pattern Matching (Chosen Alternative)

The syntax that was ultimately chosen for Q4T draws inspiration from various query languages to address all requirements. A path-based syntax that is strongly inspired by XPath enables users to retrieve any information about a service template. This satisfies requirements R1 to R3, since it is a declarative language that is easily understandable and can be written in a single line. Requirements R4 and R5 can also be fulfilled by directly accessing elements of a template by their path. To satisfy requirements R6 and R7, a pattern matching functionality that allows the user to query relationships between nodes is added, using a syntax based on Cypher that connects nodes with directional arrows. Adjustments include the ability to specify nested paths inside nodes to make expressions more concise, and the change from square brackets around relationships to curly braces, since we already reserve square brackets for filters. Finally, the various elements of the language are denoted by keywords like `FROM` and `SELECT`, similar to SQL.

One minor disadvantage of this solution is the switch between accessing elements in the document by their hierarchy in `SELECT` statements, versus working on the topology in `MATCH` statements, which might be confusing at first since these are two different ways of retrieving data.

5 Implementation

This section describes the prototypical implementation of Q4T. It first gives an overview of the entire system in Section 5.1. Afterwards, it details each step of the execution, starting with receiving queries in Section 5.2, then the parsing and lexical analysis stage in Section 5.3. Section 5.4 explains how the abstract syntax tree (AST) generated by the parser is resolved, and Section 5.5 goes into detail about pattern matching. Section 5.6 describes the process of resolving queries inside a template. Lastly, Section 5.7 points out the limitations of the current implementation.

5.1 Overview

Q4T is implemented as part of OpenTOSCA Vintner, an open-source application that allows one to specify different deployment variants of an application in a single model [1]. It is implemented in TypeScript and uses the Node.js runtime environment. Figure 5.1 shows a schematic overview of the query subsystem. The user formulates their query and transmits it to the application. This can happen locally by directly calling Vintner using the command-line interface (CLI), or by submitting a REST call over the network to a server instance. After the query is received, it is interpreted and translated to an AST by the parser, which gets handed over to the resolver. The resolver then fetches one or more service templates from a repository plugin, and optionally merges instance data from an orchestrator plugin into the template if the query specifies it. Afterwards, it traverses the AST and resolves each statement, producing a result object. After the evaluation is finished, this result is returned to the user, either as a response to a REST request, or a file as well as a console output when using the CLI.

[1] https://github.com/OpenTOSCA/opentosca-vintner

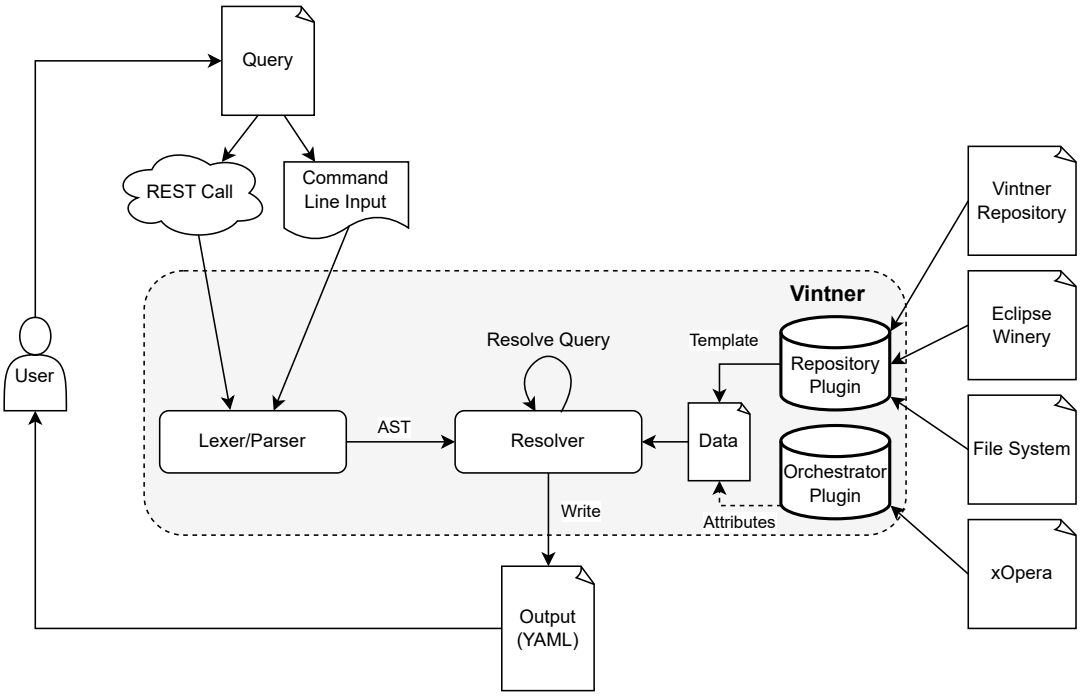

Figure 5.1: Architecture of the Implementation

5.2 Receiving a Query

This section describes the process of receiving a query, either by a CLI or via a REST call.

5.2.1 Command Line Interface (CLI)

Vintner uses the `Commander` framework [2] to implement a command-line interface (CLI) that can be used to execute commands, which was extended to support the execution of queries. All commands are defined in `index.ts`.

The command to run a query is `vintner query run`, followed by the options listed in Table 5.1. `query` is a required option that contains the query expression as a string. `source` defines where service templates should be searched, defaulting to the Vintner repository. Lastly, `output` can be used to define an output file to which the result of the query is saved.

Listing 5.1 shows a simple command to execute a query on a template from the Vintner repository, the result of which gets written into an output file called `output.yaml`.

[2] https://github.com/tj/commander.js

Flag	Meaning	Required
query	Query string (see Section 4.5)	yes
source	Place to search for template(s) in query	no
output	Name of a YAML file to dump the output of the query	no

Table 5.1: Vintner Query Command Flags

Listing 5.1 CLI Command Example

```
vintner query run --query 'FROM templates.my-app SELECT node_templates.*[type="VirtualMachine
"]'
--output output.yaml --source vintner
```

5.2.2 REST API

It is also possible to submit a REST request to a Vintner server and receive the result of the query as a response. The body of the request should include a JSON with the parameters described in Table 5.1. Listing 5.2 shows the same query as the previous example in the form of a REST request.

Listing 5.2 REST Request Example

```
POST /query/run HTTP/1.1
Host: http://127.0.0.1:3000
Content-Type: application/json

{
    "query": "FROM templates.my-app SELECT node_templates.*[type='VirtualMachine']"
    "output": "output.yaml"
    "source": "vintner"
}
```

5.3 Lexing and Parsing

The first step in processing a query is the lexical analysis and parsing of the input string. The class responsible for lexing and parsing is `parser.ts`, which has a `getAST()` method that takes a query string as input and returns a representation of the query as an AST.

This process is realized using Ohm [3], an open-source parsing toolkit. The Queries4TOSCA (Q4T) language is described in the `queryGrammar.ohm file`, which gets loaded by the parser class at runtime. This file describes the Q4T grammar in Ohm's domain-specific language, which is largely based on

[3] https://ohmjs.org/

parsing expression grammars (PEGs) (see Section 2.5). This grammar definition can also be found in the appendix (Listing A.1). While this file describes the syntax of the language, the semantics are handled separately by the `parser.ts` class. This class loads the aforementioned grammar file and uses it to instantiate an `Ohm` grammar object, which can then be used to match the query against it. If there are any syntactical errors in the query string, `Ohm` will return an error message that then gets passed on to the user, providing information about where the match failed and what kind of input was expected. If the match succeeds, an AST will be created to provide a structured representation of the query. This is accomplished by defining a so-called action dictionary and passing it to the `Ohm` library, along with the match object. The action dictionary consists of a collection of methods named after grammar rules. Each time the corresponding rule is matched, the method, also called a semantic action, gets executed. Each method creates a part of the AST.

Figure 5.2 shows an example of a query and its corresponding AST. The whole query, called an `Expression`, consists of a `FromExpression` and `SelectExpression`, which themselves are made up of various child elements. The `resolver` class can then use the information contained in this tree to produce the result.

FROM templates.my-app SELECT node_templates.*[type="VirtualMachine"]

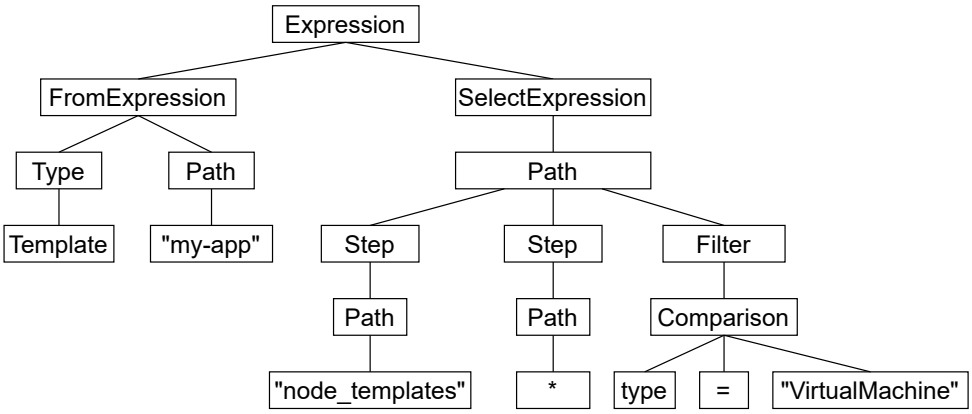

Figure 5.2: A Query and its Abstract Syntax Tree

5.4 Resolving Queries

Queries are resolved by calling the `resolve()` method in `resolver.ts`, which takes a query string as an input. It creates a new parser instance and calls the `getAST()` method to get a representation of the query as an AST. Upon successfully receiving the AST, it traverses its nodes to resolve all parts of the query. When this process is completed, the resulting matches are returned.

5.4.1 Loading Templates

The task of loading a template depends on two parameters: the path specified in the `FROM` statement, and the contents of the `source` option. The different ways of finding the desired template depending on this option are outlined below.

Vintner

This option directs the resolver to load a template or instance from the Vintner repository, identified by its name as written in the `FROM` statement. If the `FROM` statement specifies that templates should be used, the template of the instance with the specified name will be loaded. If instances are specified, the template will be enhanced with instance data by calling the method `getTemplateWithAttributes()`. This method fetches the service template of the specified instance, then uses the method `getAttributes()` of the currently active orchestrator to retrieve all instance data. The process of fetching instance data is described in detail in Section 5.4.2.

File

If this option is set, the service template is loaded directly from the file system, using the path provided in the `FROM` statement. This can be an absolute path, or a relative path starting from the directory from which Vintner is executed. If the file is not found, an error is returned.

Winery

When this option is used, Vintner will try to determine the location of the Eclipse Winery[4] repository on the current machine, then retrieve the service template specified by the path in the `FROM` statement. The location of the repository is determined by examining the configuration file located in `home/.winery/winery.yml` and looking for the key `repository.repositoryRoot`. Once this value is found, the specified file path is accessed, and the template is retrieved, using the winery repository as the root directory and appending the path in the `FROM` statement.

5.4.2 Fetching Instance Data

Attributes represent the current state of a TOSCA entity like a node or relationship. They are only known after an application has been deployed and instantiated. This means that they cannot be gathered from the template alone, instead, they need to be retrieved by looking at the data files that the orchestrator provides. The implementation prototype currently provides support for loading instance data from xOpera [5], with the potential to add other orchestrators later by implementing an interface. To fetch instance data from xOpera, the directory that the instance is running in is determined and searched. xOpera has a rather simple format for instance data. For each node

[4] https://github.com/eclipse/winery
[5] https://github.com/xlab-si/xopera-opera

and relationship, a corresponding JSON file is created in a folder called `data`. The files consist of a series of objects, named after the individual attributes. Each of these objects contains two properties: `is_set`, a Boolean which specifies whether the variable has been instantiated, and `data`, which contains the actual value of the attribute at the current point in time. Listing 5.3 shows a small snippet of an instance data file for a single node template. After all files are loaded by Vintner, they are compiled into a single object that contains the attributes for each node, which then gets merged into the template object that was loaded previously. This allows the user to query the instance model at the current point in time.

Listing 5.3 xOpera Instance Data File of a Single Node Template

```
{
  "tosca_name": {
    "is_set": true,
    "data": "first"
  },
  "tosca_id": {
    "is_set": true,
    "data": "first_0"
  },
  "state": {
    "is_set": true,
    "data": "started"
  }
}
```

5.5 Pattern Matching

To match patterns in a template, an abstract representation of the node graph is created in `node-graph.ts`. This representation contains a list of all nodes as well as their relationships. Capabilities are translated into incoming relationships, while requirements are converted to outgoing relationships.

Resolving a pattern starts from an initial node, commonly referred to as an anchor in graph query languages. We select the first node in the query as our anchor, without looking for more optimal candidates. At first, any conditions in the query are evaluated to find a set of nodes that fulfill them. Then, the method `limitedBFS()` in `node-graph.ts` is executed for each potential starting node. This method implements a modified breadth-first search (BFS) algorithm that can handle variable length patterns. It takes five input parameters: the starting node, the minimum number of desired hops, the maximum number of hops, the direction of the relation, and any conditions that the relation needs to fulfill. It maintains all nodes that still need to be visited in a queue, which first gets populated with the initial nodes. Afterwards, nodes that can be reached from the current nodes via relationships that fulfill all conditions are added to the queue. When the maximum number of hops is reached or the queue is empty, the algorithm terminates and returns the names of all potential target nodes as a result. Afterwards, the filters specified in the next node of the query are applied to all of them to filter out any nodes that do not fulfill the condition. All paths are saved, and if there are still unresolved relationships left, the same steps are executed, starting from the last node of each path.

After all nodes and relationships are resolved, the paths that were found are translated into a result map. This map contains an object for all specified variables in the query, along with all node templates that match that variable. Listing 5.4 shows an example query, while Listing 5.5 shows the result. Since `vm_1` and `vm_2` are both connected to `openstack`, they appear under the `vm` variable in the result.

Listing 5.4 Example of a Match Query

```
FROM templates.my-app MATCH (host[name='openstack'])<--(vm) SELECT .
```

Listing 5.5 Result of the Above Match Query

```
host:
    openstack:
        type: OpenStack
        properties:
            ip_address: 127.0.0.1
vm:
    vm_1:
        type: VirtualMachine
        properties:
            num_cpus: 2
            mem_size: 4 GB
            operating_system: Ubuntu 22.10
        requirements:
            - host: openstack
    vm_2:
        type: VirtualMachine
        properties:
            num_cpus: 2
            mem_size: 4 GB
            operating_system: Ubuntu 22.10
        requirements:
            - host: openstack
```

5.6 Resolving Queries in Templates

This implementation also includes the ability to resolve all queries contained inside a template. Similar to running a query, this command can be called from either the CLI or the REST API. The potential options are described in Table 5.2. One can define the template to resolve, the source, and the output file for the resolved template.

Listing 5.6 shows an example of a CLI command that resolves a template called *my-template* and saves it to *my-template-resolved.yaml*.

Flag	Meaning	Required
template	Name of the template to resolve queries in	yes
source	Place to search for specified template	no
output	Name of the resolved template	yes

Table 5.2: Vintner Template Query Resolve Command Flags

Listing 5.6 Example of a CLI Command to Resolve Queries in a Template

```
vintner query resolve --template my-template --output my-template-resolved.yaml
```

Listing 5.7 shows a small snippet of a service template that contains two queries, while Listing 5.8 shows the resulting file after resolving is finished. The queries are replaced with their actual values.

Listing 5.7 Snippet of a Template with Queries

```
topology_template:
    node_templates:
        webapp:
            properties:
                db_username: executeQuery(SELECT node_templates.mysql_database.properties.
username)
                db_password: executeQuery(SELECT node_templates.mysql_database.properties.
password)
        mysql_database:
            properties:
                username: my_user
                password: my_password
        # rest omitted
```

Listing 5.8 The Same Template After Resolving

```
topology_template:
    node_templates:
        webapp:
            properties:
                db_username: my_user
                db_password: my_password
        mysql_database:
            properties:
                username: my_user
                password: my_password
        # rest omitted
```

The algorithm starts out by initializing a variable tracking the number of queries to zero. Then, it recursively goes through the entire service template and tries to match each value it finds against a pattern that describes query commands. Each time it encounters a query, it increases the number of found queries by one, then it tries to execute the query contained in the command. If the result of the query is itself a query, it does not change the value, if it is an atomic value, it replaces the query command with the result. This loop is repeated until either no more queries are found, or the number of queries found in the loop is equal to the number of queries found in the previous loop. This means that there is a cyclic dependency in the template, meaning that one query is the result of another query, and vice versa. In this case, an error is thrown, and the process of resolving the template is aborted, as to prevent an infinite loop.

5.7 Limitations

This section discusses limitations of the current implementation and how they could be addressed in the future.

5.7.1 Limited Support for Instance Data

Currently, the implementation can only gather instance data from the orchestrator `xOpera`. However, the main `Vintner` application also supports `Unfurl`[6], and potentially other orchestrators in the future. Since every orchestrator has its own unique format for their instance data, a custom method needs to be written for each of them to be able to include their data in queries. An interface to implement the necessary methods is provided, making it easy to add support for more orchestrators.

5.7.2 No Support for Imports

The current implementation only considers elements directly contained in the template file. However, service templates may declare imports from other files. In a future version, those could be made accessible in queries by loading them in before executing a query.

[6] https://github.com/onecommons/unfurl

6 Conclusion and Future Work

This chapter presents a conclusion of this work in Section 6.1 and discusses potential future development and research in Section 6.2.

6.1 Conclusion

This thesis presents a novel query language specifically tailored towards querying TOSCA files in a concise and flexible manner. It facilitates searching for and retrieving values from templates and repositories. We highlighted several important query languages, detailing their syntax and use cases. Drawing inspiration from XPath and Cypher, we introduced a new query language called Queries4TOSCA (Q4T). It features a path syntax to retrieve elements from a TOSCA template, with the ability to add filters at any step. Additionally, a syntax for constructing return objects is included, which gives users precise control over the content and shape of the query output. To tackle the challenge of searching for specific patterns in the architecture of the application topology, a pattern matching syntax based on Cypher is added.

Implemented as part of the larger project OpenTOSCA Vintner, the prototypical query client supports all the language features described in the previous chapter. To retrieve instance and template data, it uses a plugin-based architecture that can be expanded in the future.

6.2 Future Work

One potential area for further research is the ability to use Q4T with older TOSCA templates using the profile version 1.0. Since Paths4TOSCA uses a generic path syntax to access elements of a template by their name, it would work without any modifications. The same applies to `MATCH` statements. Since they are an abstract representation of patterns in the topology template, only the resolver would need to be adjusted, not the syntax itself. However, since support for XML profiles has already been discontinued by OASIS, it might be a better solution to convert older applications to the new YAML-based specification. One such approach is described in [Kle17].

Many query languages support all CRUD operations. For example, SQL has the commands INSERT (create), SELECT (retrieve), UPDATE (update), and DELETE (delete). Q4T, in its current version, only supports retrieval, however, it could be extended in the future to support other operations as well. Since Paths4TOSCA already provides a way to browse to a specific element of the template, it would be possible to create a syntax that adds an assignment to the end of a path expression to set an element at that location.

Subqueries are another possible addition that would make Q4T more versatile. Subqueries are queries that are nested inside a larger query. This can be useful in cases where a user wants to run a query based on the result of another query.

Bibliography

[Ang12] R. Angles. "A comparison of current graph database models". In: *Proc. - 2012 IEEE 28th Int. Conf. Data Eng. Work. ICDEW 2012*. IEEE. 2012, pp. 171–177. ISBN: 9780769547480. DOI: 10.1109/ICDEW.2012.31 (cit. on p. 29).

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. *TOSCA: Portable automated deployment and management of cloud applications*. Vol. 9781461475. 2014, pp. 527–549. ISBN: 9781461475354. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on pp. 17, 22, 24).

[BBLS12] T. Binz, G. Breiter, F. Leyman, T. Spatzier. "Portable cloud services using TOSCA". In: *IEEE Internet Comput.* 16.3 (2012), pp. 80–85. ISSN: 10897801. DOI: 10.1109/MIC.2012.43 (cit. on p. 22).

[BEI09] O. Ben-Kiki, C. Evans, B. Ingerson. "Yaml ain't markup language (yaml™) version 1.1". In: *Work. Draft 2008* 5 (2009), p. 11 (cit. on p. 24).

[BEK+16] U. Breitenbücher, C. Endres, K. Kepes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. "The OpenTOSCA Ecosystem - Concepts & Tools". In: *EPS*. 2016 (cit. on p. 23).

[Byr15] L. Byron. *GraphQL: A data query language*. 2015. URL: https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/ (cit. on p. 37).

[CFR+03] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, M. Stefanescu. "XQuery: A query language for XML". In: *SIGMOD Conf.* Vol. 682. 2003, p. 50 (cit. on p. 35).

[Eur21] Eurostat. *Cloud computing - statistics on the use by enterprises*. 2021. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises (visited on 09/12/2022) (cit. on p. 17).

[FGG+18] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor. "Cypher: An evolving query language for property graphs". In: *Proc. ACM SIGMOD Int. Conf. Manag. Data*. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. ISBN: 9781450317436. DOI: 10.1145/3183713.3190657. URL: https://doi.org/10.1145/3183713.3190657 (cit. on p. 36).

[For04] B. Ford. "Parsing expression grammars: A recognition-based syntactic foundation". In: *Conf. Rec. Annu. ACM Symp. Princ. Program. Lang.* Vol. 31. 2004, pp. 111–122 (cit. on pp. 26, 27).

[GWO02] J. R. Groff, P. N. Weinberg, A. J. Oppel. *SQL: the complete reference*. Vol. 2. McGraw-Hill/Osborne, 2002 (cit. on p. 33).

[IBHS18] E. T. Irawan, D. Bappedyanto, D. Hariyadi, A. Syawqi. "BIZNET QUERY LANGUAGE PADA INFRASTRUCTURE AS CODE". In: *Teknomatika* 11.1 (2018). DOI: 10.13140/RG.2.2.29978.70086. URL: https://www.researchgate.net/publication/327282938 (cit. on p. 33).

[Kim09] W. Kim. "Cloud computing: Today and tomorrow." In: *J. Object Technol.* 8.1 (2009), pp. 65–72 (cit. on p. 17).

[Kle17] C. Kleine. "Backward and Forward Compatibility for TOSCA Simple Profile in YAML Version 1.0: Concept and Modelling Tooling Support". PhD thesis. 2017. DOI: 10.18419/OPUS-9469. URL: http://elib.uni-stuttgart.de/handle/11682/9486 (cit. on p. 69).

[LF09] F. Leymann, D. Fritsch. "Cloud computing: The next revolution in IT". In: *Proc. 52th Photogramm. Week* (2009), pp. 3–12 (cit. on p. 17).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing". In: (2011) (cit. on pp. 21, 22).

[Neo22] Neo4j. *Values and types - Neo4j Cypher Manual*. 2022. URL: https://neo4j.com/docs/cypher-manual/current/syntax/values/#composite-types (visited on 11/29/2022) (cit. on pp. 37, 56).

[OAS] OASIS Standard. *TOSCA Simple Profile in YAML Version 1.3*. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html (visited on 10/11/2022) (cit. on pp. 22–24).

[OAS13] OASIS Standard. "Topology and Orchestration Specification for Cloud Applications Version 1.0". In: *Organ. Advacement Struct. Inf. Stand.* November (2013), pp. 1–114. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 22, 24).

[OAS20] OASIS. *TOSCA Version 2.0*. Tech. rep. 2020. URL: https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd04/TOSCA-v2.0-csd04.html#_Toc106639604 (cit. on pp. 22, 29, 30).

[OCo] C. O'Connor. *The Norway Problem - why StrictYAML refuses to do implicit typing and so should you - HitchDev*. URL: https://hitchdev.com/strictyaml/why/implicit-typing-removed/ (visited on 10/14/2022) (cit. on p. 25).

[One22] OneCommons Co. *Template Processing — Unfurl Documentation*. 2022. URL: https://docs.unfurl.run/processing.html#id10 (visited on 11/15/2022) (cit. on p. 31).

[RCDS11] J. Robie, D. Chamberlin, M. Dyck, J. Snelson. *XQuery 3.0: An XML Query Language*. 2011. URL: https://www.w3.org/TR/xquery-30/%20http://www.w3.org/TR/2011/WD-xquery-30-20111213/ (visited on 11/21/2022) (cit. on p. 35).

[RHK+16] O. van Rest, S. Hong, J. Kim, X. Meng, H. Chafi. "PGQL: a property graph query language". In: *Proc. Fourth Int. Work. Graph Data Manag. Exp. Syst.* 2016, pp. 1–6 (cit. on p. 37).

[Tec16] Technopedia. *Query Language*. 2016. URL: https://www.techopedia.com/definition/3948/query-language (cit. on p. 29).

[VWA+15] A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, J. Partner. *Neo4j in action*. Vol. 22. Manning Shelter Island, 2015 (cit. on p. 36).

All links were last followed on November 29, 2022.

A Grammar Definition

Listing A.1 Parsing Expression Grammar for Q4T

```
Main = (Expression | MatchExpression) end
Expression = FromExpression Select
MatchExpression = FromExpression Match Select

FromExpression = "FROM" ("instances" | "templates") ("/" | ".") (asterisk | filePath)

Select = "SELECT" Path ("," Path)*
Path = (Group | Policy | Step | ".") (ArrayAccess | Map | Filter)* ReturnClause?
Map = "." Step
Filter = PredicateExpression
Step = shortcut? (asterisk | ident)
ArrayAccess = "[" integer "]"
ReturnClause = "{" KeyValuePair ("," KeyValuePair)* "}"
KeyValuePair = Variable ":" Variable --complex
           | Variable              --simple
Group = "GROUP" "(" ident ")"
Policy = "POLICY" "(" ident ")"

PredicateExpression = "[" Predicate "]"
Predicate = Predicate logic Predicate -- multi
        | Condition -- single
Condition = negation Value comparison literal -- comparison
        | negation Value -- existence

Match = "MATCH" Node (Relationship Node)*
Node = "(" ident? PredicateExpression? ")"
Relationship = arrowLeft arrowRight --simple
           | arrowLeft "{" ident? PredicateExpression? Cardinality? "}" arrowRight --cond
Cardinality = asterisk integer ".." integer --range
          | asterisk ".." integer --max
          | asterisk integer ".." --min
          | asterisk integer --exact
          | asterisk --unlimited
Variable = literal | path | ident
Value = shortcut? (path | literal)
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature