TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Simple querying service for OpenAPI descriptions with Semantic Web extensions

by

Ioannis Apostolakis

A thesis submitted in fulfillment of
the requirements for the degree of
Diploma in Electrical and Computer Engineering

May 2022

THESIS COMMITTEE
Prof. Euripides G.M. Petrakis, *Supervisor*
Prof. Antonios Deligiannakis
Assoc. Prof. Georgios Chalkiadakis

Abstract

This work presents OpenAPI Query Language 2 (OAQL2), a language for querying OpenAPI documents. OpenAPI is a standard format for the description of RESTful services, based on JSON. OAQL2 is designed with syntax similar to SQL and supports querying most of the fields in an OpenAPI document, as well as the semantic annotations proposed for OpenAPI in previous work. A web service capable of executing OAQL2 queries was implemented. This service stores metadata for each description and executes the queries on them. It builds indexes to speed up queries, can handle composite schema objects and uses reasoning to support searching in a semantic model. Compared to the system implemented in previous work, it is shown to be much faster and complete in terms of syntax and compatibility with OpenAPI.

Acknowledgements

I would like to thank my supervisor Prof. Euripides G. M. Petrakis for his tireless support and guidance during every step of this work.

I would also like to express my gratitude to Nikos Mainas for his valuable observations.

Finally, I want to thank my friends and family for their love, support and being always by my side.

Contents

Chapter 1

Introduction

Web services are published on the Web by various software vendors. Each web service is a functional unit that provides services to clients over HTTP. Examples of web services are social media, search engines, online shops, cloud storage and others. A web service is often a combination of other web services that interact with each other to provide the desired functionality. For example, an online shop might consist of a server to handle the requests from users and a database service to store the information.

1.1 Problem definition

OpenAPI Specification provides a standard way to describe RESTful web services. There are thousands of web services in the World Wide Web which makes it difficult for a user to find the service he/she needs. Therefore, it is useful to create a system that can search through OpenAPI documents for those matching the user's criteria. This system should be fast, easy to use and should not require in-depth knowledge of an OpenAPI document's structure.

1.2 Proposed solution

In this thesis, we introduce OAQL2 (OpenAPI Query Language 2) which is a language for searching in OpenAPI documents with syntax similar to SQL. Additionally, we describe a web service that can store OpenAPI documents and execute OAQL2 queries. To optimize performance, we store metadata for each document. OAQL2 queries are translated and executed on the database of metadata. The web service also supports the semantic annotations for OpenAPI proposed in previous work.

1.3 Thesis outline

In chapter 2 we present an introduction to concepts and technologies essential to our work. In chapter 3 we define the syntax rules of OAQL2. Chapter 4 describes the implementation of the service and the algorithms for parsing OpenAPI documents and translating OAQL2. In chapter 5 we analyze the factors affecting the performance of the service, show experimental results and compare our service with other systems. Finally, chapter 6 contains conclusions and ideas for future work.

Chapter 2

Background

The number of web services has been rapidly increasing over the last years. This created the need for a standard description of web services' capabilities that can be understood by both humans and machines. By reading that description, one is able to find a service with the desired functionality and understand how to use its API without needing to know implementation-specific details.

2.1 REST

Representational state transfer (REST) is a software architectural style proposed by Roy Fielding in 2000 [1]. Its purpose is to guide the architecture of large web systems and the interactions between web services in order to achieve better performance, scalability, simplicity, modifiability, visibility, portability and reliability. REST defines the following constraints:

- Client-server architecture: An interaction between client and server can be initiated only by the client with a request and the response is sent by the server as a reaction to the request.
- Statelessness: The server does not keep any information about previous interactions with clients. Each request must contain all the necessary information to be understood by the server on its own.
- Cacheability: Responses must define themselves as cacheable or non-cacheable to reduce network traffic.
- Layered system: The interaction between client and server should not be affected by whether they are connected directly or through intermediary servers.
- Uniform interface: Resource identification in requests, resource manipulation through representations, self-descriptive messages and HATEOAS (hypermedia as the engine of application state).
- Code on demand (optional): The client can request executable code from the server.

A web service that satisfies the above constraints is said to be RESTful.

2.2 OpenAPI Specification

The OpenAPI Specification is a standard, language-agnostic description of RESTful web services. The latest version is 3.1.0, published in February 2021 [2]. An OpenAPI document is a JSON object and can be represented in either JSON or YAML format.

OpenAPI Specification defines many objects contained in an OpenAPI document. The *Paths*, *Path Item*, *Operation* and *Request Body* objects provide information about the requests supported by a service while the *Responses* and *Response* object describe the possible responses to a request. The *Parameter* object describes the parameters that may be passed in a request and the *Header* object describes the headers that will be returned with the response. The *Schema* object provides information about the type of the parameters or headers and the format of the request or response payload. The *Server* object describes the servers of the web service. There are also many other objects providing additional information (eg external documentation).

Among other tools, OpenAPI community offers an online platform for designing APIs, called SwaggerHub[1]. Users can create their own OpenAPI documents using a friendly user interface and make them publicly available.

2.3 Semantic Web

The Semantic Web, sometimes known as Web 3.0, is an extension of the World Wide Web through standards set by the World Wide Web Consortium (W3C) [3]. It aims to give machines the ability to understand the Internet by encoding the data from web services with concepts from a semantic model. This is achieved by using technologies such as RDF (Resource Description Framework) and OWL (Web Ontology Language).

RDF is a W3C standard and can describe a directed graph with triple statements [4]. A triple statement is represented by a node as the *subject*, an arc going from the subject to an object as the *predicate* and a node as the *object*. The subject and the predicate must be a URI (uniform resource identifier) while the object may also be a literal value.

RDF specification defines a vocabulary with some URIs. For example, it defines the resource *rdfs:Class*[2] which is the class of all classes and the property *rdf:type*[3] which states that a resource is an instance of a class. Thus, the triple *A rdf:type rdfs:Class* means that the resource *A* is a class.

Other defined properties are *rdfs:subClassOf* and *rdfs:subPropertyOf*. These properties are:

- reflexive: a class or property is always a subclass or subproperty respectively of itself
- transitive: if A is a subclass of B and B is a subclass of C then A is a subclass of C as well

Usually, not all valid triples are explicitly stated but they can be obtained by using a reasoner.

[1] `https://app.swaggerhub.com`

[2] The prefix *rdfs:* is short for *http://www.w3.org/2000/01/rdf-schema#*

[3] The prefix *rdf:* is short for *http://www.w3.org/1999/02/22-rdf-syntax-ns#*

2.4 Semantic annotations in OpenAPI

A human reading an OpenAPI document will understand in most cases the semantics of the various elements by their name, description or other provided information. However a machine needs an explicit declaration of the elements' semantics. The extension properties in the following table were introduced in older work [5] to provide this information.

Property	Applies to	Meaning
x-refersTo	Schema Object	The concept in a semantic model that describes an OAS element.
x-kindOf	Schema Object	A specialization between an OAS element and a concept in a semantic model.
x-mapsTo	Schema Object	An OAS element which is semantically similar with another OAS element.
x-collectionOn	Schema Object	A model describes a collection over a specific property.
x-onResource	Tag Object	The specific Tag object refers to a resource described by a Schema object.
x-operationType	Operation Object	Clarifies the type of operation.

The value of *x-refersTo*, *x-kindOf* and *x-operationType* is the URI of a concept in a semantic model. The value of *x-mapsTo* and *x-onResource* is a reference to a *Schema* object in the OpenAPI description. Finally, the value of *x-collectionOn* is the name of the *Schema* object's property that is an array holding a collection of items.

2.5 Related work

A similar system was designed and implemented in previous work [6]. However, that work is only an initial approach to the subject and has some weaknesses:

- it is slow because it searches on the original OpenAPI documents instead of metadata and does not use indexing
- it does not support searching in composite schema objects
- it supports searching on a very limited number of OpenAPI fields ignoring the rest
- it does not support the semantic annotations proposed for OpenAPI

OAQL2 is a redesign and extension of the *OpenAPI QL* language defined in that previous work.

Besides this, there are many databases that support querying JSON data and could be used for searching in OpenAPI documents. Some examples are MongoDB, Couchbase, CouchDB, DocumentDB, RethinkDB. However these are query languages for JSON and are not particularly designed for OpenAPI documents (JSON is a very generic format). This is mainly due to the large size and complexity of OpenAPI information. Queries searching for REST services should particularly address specific information pertaining a REST service (i.e. on operations, security, purpose). This requires that the user be familiar with the peculiarity of REST architectural style. Query expressions using query languages for JSON (rather than for OpenAPI) results in complicated and long expressions involving properties of a REST service. To demonstrate this, we present below a MongoDB query to find services accepting requests with JSON in the request body:

```
aggregate([
  {
    "$project": {
      "service": "$$ROOT",
      "tmp": {
        "$objectToArray": "$paths"
      }
    }
  },
  {
    "$unwind": "$tmp"
  },
  {
    "$project": {
      "service": 1,
      "tmp": {
        "$objectToArray": "$tmp.v"
      }
    }
  },
  {
    "$match": {
      "tmp.v.requestBody.content.application/json": {
        "$exists": true
      }
    }
  },
  {
    "$replaceRoot": {
      "newRoot": "$service"
    }
  }
])
```

Removing the complexity of queries on OpenAPI documents is exactly the problem this work is dealing with.

JSONPath [7] and JSONQuery [8] are other query languages designed to find and extract information from JSON objects. Jaql [9] is a data processing and querying language for JSON used on big data. JSONiq [10] is another query language for JSON. However, all these and any other languages for JSON still require the user to know the structure of OpenAPI documents in detail and compose long complicated queries.

Another system working with OpenAPI descriptions is OpenAPI-to-GraphQL[4]. This system aims to create a wrapper service for a web service, based on the OpenAPI description of the web service. The wrapper service accepts GraphQL queries from the clients and determines the HTTP requests that will provide the required information. Then, it executes these requests on the web service and returns the results in GraphQL format to the client. OpenAPI-to-GraphQL executes queries on a single web service's resources rather than searching through multiple OpenAPI documents for web services matching specific criteria. Thus, it does not fulfill our requirements.

Finally, there is a tool[5] named OpenAPI, made by Broadcom, which can execute queries to extract data related to the company's products. This tool is completely unrelated to OpenAPI Specification and simply happened to share the same name.

[4]`https://github.com/IBM/openapi-to-graphql`

[5]`https://techdocs.broadcom.com/us/en/ca-enterprise-software/it-operations-management/dx-netops/20-2/Performance-Monitoring-with-DX-Performance-Management/apis/openapi.html`

Chapter 3

Tables in OAQL2

In this work we introduce OAQL2 (OpenAPI Query Language 2) which is an SQL-like query language for searching in OpenAPI descriptions. Similarly to executing SQL queries on tables in a relational database, in our system a user can execute OAQL2 queries on the tables defined in this chapter. The fields (columns) of these tables are mapped to fields defined in OpenAPI Specification v3.1.0.

The fields of the tables in our system can belong to one of *string*, *number*, *boolean* and *any* data types. The data type *any* represents any kind of value, including the three other types, JSON objects and arrays. We denote that a table field is an array of items by appending square brackets [ ] to its data type.

Besides the defined fields, OpenAPI Specification allows in most objects the definition of any additional field whose name begins with the prefix "x-". Our system supports querying on these extension properties and allows queries in all tables on any field whose name begins with "x-".

Unlike tables in relational databases, we will not define any primary or foreign keys in the tables in our system. The reason for this will be explained later, in section 4.2.

It is worth noting that some table fields contain information considered less useful in a search. For example, there is probably no use case in which one would need to search by the URL of external documentation. The purpose of these table fields is to be included in the results of the query to provide additional information.

3.1 Service

The fields of the *Service* table correspond to fields in *OpenAPI*, *Info*, *Contact*, *License* objects and *External Documentation* objects. These objects contain general information about the web service, such as its title or description. An example is below:

```
{
  "title": "Example service",
  "description": "This is a description of the service.",
  "termsOfService": "https://example.com/terms/",
  "contact": {
    "name": "API Support",
    "url": "https://www.example.com/support",
    "email": "support@example.com"
  },
  "license": {
    "name": "Apache 2.0",
    "url": "https://www.apache.org/licenses/LICENSE-2.0.html"
  },
  "version": "1.0.1"
}
```

The fields of *Service* table along with the corresponding OpenAPI fields are shown below:

Service	**Corresponding OpenAPI field**
contactEmail : *string*	*email* in *Contact* object
contactName : *string*	*name* in *Contact* object
contactUrl : *string*	*url* in *Contact* object
description : *string*	*description* in *Info* object
extDocsDescription : *string*	*description* in *External Documentation* object
extDocsUrl : *string*	*url* in *External Documentation* object
id : *string*	(described below)
jsonSchemaDialect : *string*	*jsonSchemaDialect* in *OpenAPI* object
licenseName : *string*	*name* in *License* object
licenseUrl : *string*	*url* in *License* object
openapiVersion : *string*	*openapi* in *OpenAPI* object
summary : *string*	*summary* in *Info* object
termsOfService : *string*	*termsOfService* in *Info* object
title : *string*	*title* in *Info* object
version : *string*	*version* in *Info* object

The *id* field is an automatically generated id, unique for each OpenAPI description that was inserted in the system. This id can be used to retrieve the original OpenAPI document as described in a later chapter.

We add one entry to *Service* table for each OpenAPI description inserted into the system. The entry for the example above would be the following:

Table fields	**Entry values**
contactEmail	"support@example.com"
contactName	"API Support"
contactUrl	"https://www.example.com/support"
description	"This is a description of the service."
id	"625186c88b441f42012f7dfd"
licenseName	"Apache 2.0"
licenseUrl	"https://www.apache.org/licenses/LICENSE-2.0.html"
termsOfService	"https://example.com/terms/"
title	"Example service"
version	"1.0.1"
...	NULL

3.2 Request

The fields of *Request* table correspond to fields in *Paths*, *Path Item*, *Operation* and *Request Body* objects. These objects describe the requests that can be accepted by the web service. Below is an example:

```
{
  "/pets": {
    "get": {
      "description": "GET request"
    },
    "post":{
      "description": "POST request",
      "requestBody": {
        "content": {
          "application/x-www-form-urlencoded": {
            ...
          },
          "application/json": {
            ...
          }
        }
      }
    }
  }
}
```

This example describes an endpoint on the relative path "/pets" which accepts either GET or POST requests. In these POST requests, the media type of the request body can be either "application/x-www-form-urlencoded" or "application/json".

The fields of *Request* table and their corresponding OpenAPI fields are shown below:

Request	**Corresponding OpenAPI field**
bodyDescription : *string*	*description* in *Request Body* object
bodyRequired : *boolean*	*required* in *Request Body* object
contentType : *string*	property name inside *content* in *Request Body* object
deprecated : *boolean*	*deprecated* in *Operation* object
description : *string*	*description* in *Operation* object
extDocsDescription : *string*	*description* in *External Documentation* object
extDocsUrl : *string*	*url* in *External Documentation* object
method : *string*	property name inside *Path Item* object
operationId : *string*	*operationId* in *Operation* object
path : *string*	property name inside *Paths* object
summary : *string*	*summary* in *Operation* object
tags : *string[]*	*tags* in *Operation* object
x-operationType : *string*	proposed semantic annotation *x-operationType*

We add one entry in *Request* table for each given set of path, method and media type of request body. These three values are stored in *path*, *method* and *contentType* fields respectively. The example above would produce the following 3 entries:

Table fields	**Entry 1 values**	**Entry 2 values**	**Entry 3 values**
contentType	NULL	"application/x-www-form-urlencoded"	"application/json"
description	"GET request"	"POST request"	"POST request"
method	"get"	"post"	"post"
path	"/pets"	"/pets"	"/pets"
...	NULL	NULL	NULL

A query on *x-operationType* field will return the table entries in which the value of the field is a subclass of the given value, including the given value.

Note that queries on the *contentType* field in this and all other tables also return values that specify a media type range containing the queried value. For example, a query for "application/json" will also return entries with a value of "application/*" or "*/*".

3.3 Callback

The fields of *Callback* table correspond to fields in *Callback*, *Path Item*, *Operation* and *Request Body* objects. These objects describe callbacks, which are requests initiated by the web service as a reaction to a request made to the service.

Most of the fields in *Request* table are identical to those of *Callback* table and they correspond to the same OpenAPI fields. The only difference is the addition of *name* field which corresponds to the name of the callback.

The fields of *Callback* table and their corresponding OpenAPI fields are shown below:

Callback	**Corresponding OpenAPI field**
bodyDescription : *string*	*description* in *Request Body* object
bodyRequired : *boolean*	*required* in *Request Body* object
contentType : *string*	property name inside *content* in *Request Body* object
deprecated : *boolean*	*deprecated* in *Operation* object
description : *string*	*description* in *Operation* object
extDocsDescription : *string*	*description* in *External Documentation* object
extDocsUrl : *string*	*url* in *External Documentation* object
method : *string*	property name inside *Path Item* object
name : *string*	property name inside *callbacks* in *Operation* object
operationId : *string*	*operationId* in *Operation* object
path : *string*	property name inside *Callback* object
summary : *string*	*summary* in *Operation* object
tags : *string[]*	*tags* in *Operation* object
x-operationType : *string*	proposed semantic annotation *x-operationType*

3.4 Webhook

The fields of *Webhook* table correspond to fields in *OpenAPI*, *Path Item*, *Operation* and *Request Body* objects. These objects describe webhooks, which are requests initiated by the web service. Their difference from callbacks is that webhooks are independent from other API calls, while callbacks are triggered by a request made to the web service.

Most of the fields in *Webhook* table are identical to those of *Request* table and they correspond to the same OpenAPI fields. The only difference is the replacing of *path* with *name* which corresponds to the name of the webhook.

The fields of *Webhook* table and their corresponding OpenAPI fields are shown below:

Webhook	**Corresponding OpenAPI field**
bodyDescription : *string*	*description* in *Request Body* object
bodyRequired : *boolean*	*required* in *Request Body* object
contentType : *string*	property name inside *content* in *Request Body* object
deprecated : *boolean*	*deprecated* in *Operation* object
description : *string*	*description* in *Operation* object
extDocsDescription : *string*	*description* in *External Documentation* object
extDocsUrl : *string*	*url* in *External Documentation* object
method : *string*	property name inside *Path Item* object
name : *string*	property name inside *webhooks* in *OpenAPI* object
operationId : *string*	*operationId* in *Operation* object
summary : *string*	*summary* in *Operation* object
tags : *string[]*	*tags* in *Operation* object
x-operationType : *string*	proposed semantic annotation *x-operationType*

3.5 Tag

The fields of *Tag* table correspond to fields in *Tag* and *External Documentation* objects. These objects describe tags that can be used to group operations.

A tag is represented mainly by its name and an optional description. The example below defines a tag named "pet":

```
{
  ... ,
  "tags": [
    {
      "name": "pet",
      "description": "operations about pets"
    }
  ]
}
```

The fields of *Tag* table and their corresponding OpenAPI fields are shown below:

Tag	**Corresponding OpenAPI field**
description : *string*	*description* in *Tag* object
extDocsDescription : *string*	*description* in *External Documentation* object
extDocsUrl : *string*	*url* in *External Documentation* object
name : *string*	*name* in *Tag* object

We add one entry to *Tag* table for each *Tag* object given. The entry for the example above would be:

Table fields	**Entry values**
description	"operations about pets"
name	"pet"
...	NULL

3.6 Response

The fields of *Response* table correspond to fields in *Responses* and *Response* objects. These objects describe the possible responses to a request and the expected responses to a callback or webhook. An example is shown below:

```
{
  "200": {
    "description": "OK",
    "content": {
      "application/json": {
        ...
      },
      "application/xml": {

      }
    }
  },
  "400": {
    "description": "Bad request",
    "content": {
      "text/html": {
        ...
      }
    }
  }
}
```

This example describes two possible responses, one with status code 200 and one with status code 400. The media type of the former response's body is either "application/json" or "application/xml" and the latter response's body is "text/html".

The fields of *Response* table and their corresponding OpenAPI fields are shown below:

Response	Corresponding OpenAPI field
contentType : *string*	property name inside *content* in *Response* object
description : *string*	*description* in *Response* object
statusCode : *string/number*	property name inside *Responses* object

We add one entry to *Response* table for each given set of status code and media type of body. These two values are stored at *statusCode* and *contentType* fields of *Response* table respectively. The example above would produce the following 3 entries:

Table fields	Entry 1 values	Entry 2 values	Entry 3 values
contentType	"application/json"	"application/xml"	"text/html"
description	"OK"	"OK"	"Bad request"
statusCode	200	200	400

The *statusCode* field can be an integer in the interval [100, 599], the string "default" or a string representing a range of HTTP response status codes (one of "1XX", "2XX", "3XX", "4XX", "5XX"). A query about a numeric value of *statusCode* will also return the entries with code ranges which contain at least one code satisfying the query. Entries with "default" value will only be returned if the user queries about that value explicitly.

3.7 Parameter

The fields of *Parameter* table correspond to fields in *Parameter* object. This object describes parameters that can be passed as part of a request, callback or webhook.

The *name* field specifies the parameter's name and the *in* field specifies whether the parameter is a part of the path, a header, a cookie or appended to the URL. There are also other fields specifying serialization rules, whether the parameter is required or deprecated and if it can be empty. An example is:

```
{
  "name": "token",
  "in": "query",
  "description": "token to be given to server",
  "required": true
}
```

This example describes a required parameter named "token" that is appended to the URL of the request.

The fields of *Parameter* table and their corresponding OpenAPI fields are shown below:

Parameter	Corresponding OpenAPI field
allowEmptyValue : *boolean*	*allowEmptyValue* in *Parameter* object
allowReserved : *boolean*	*allowReserved* in *Parameter* object

contentType : *string*	property name inside *content* in *Parameter* object
deprecated : *boolean*	*deprecated* in *Parameter* object
description : *string*	*description* in *Parameter* object
explode : *boolean*	*explode* in *Parameter* object
in : *string*	*in* in *Parameter* object
name : *string*	*name* in *Parameter* object
required : *boolean*	*required* in *Parameter* object
style : *string*	*style* in *Parameter* object

We add one entry to *Parameter* table for each *Parameter* object given. The example above would produce the following entry:

Table fields	**Entry values**
description	"token to be given to server"
in	"query"
name	"token"
required	true
...	NULL

3.8 Header

The fields of *Header* table correspond to fields in *Header* object. This object describes headers that are returned with a response or headers that refer to a schema property when the media type of the payload is a *multipart*[1]. Note that request headers are considered parameters and are described by *Parameter* objects instead of *Header* objects. An example is:

```
{
  "headers": {
    "X-Rate-Limit": {
      "description": "The number of allowed requests",
      "required": true
    }
  },
  ...
}
```

In this example a required header is defined with the name "X-Rate-Limit".

The fields of *Header* table and their corresponding OpenAPI fields are shown below:

Header	**Corresponding OpenAPI field**
allowEmptyValue : *boolean*	*allowEmptyValue* in *Header* object
contentType : *string*	property name inside *content* in *Header* object
deprecated : *boolean*	*deprecated* in *Header* object

[1]A multipart payload represents a composite document, separated into sections, with each section having its own internal headers

description : *string*	*description* in *Header* object
explode : *boolean*	*explode* in *Header* object
name : *string*	*name* in *Header* object
style : *string*	*style* in *Header* object

We add one entry to *Header* table for each *Header* object given. The example above would produce the following entry:

Table fields	**Entry values**
description	"The number of allowed requests"
name	"X-Rate-Limit"
required	true
...	NULL

3.9 Schema

The fields of *Schema* table correspond to fields in *Schema* and *External Documentation* objects. As declared in OpenAPI Specification, the fields of *Schema* object include all fields defined in JSON Schema draft 2020-12.

Schema objects describe the data model (format and data type) of request or response payloads, parameters and headers. These can be a single value, an array or a JSON object. In the case of an array, the format of its items is described by other objects, stored in *Item* table which is defined later. In the case of a JSON object, the properties it contains (key-value pairs) are described by other objects, stored in *Property* table which is also defined later. An example is:

```
{
  "content": {
    "text/plain": {
      "schema": {
        "type": "string",
        "minLength": 2
      }
    }
  }
}
```

The *Schema* object in this example describes a string value with at least two characters.

The fields of *Schema* table and their corresponding OpenAPI fields are shown below:

Schema	**Corresponding OpenAPI field**
const : *any*	*const* in JSON Schema
contentEncoding : *string*	*contentEncoding* in JSON Schema
contentMediaType : *string*	*contentMediaType* in JSON Schema
default : *any*	*default* in JSON Schema

deprecated : *boolean*	*deprecated* in JSON Schema
description : *string*	*description* in JSON Schema
enum : *any[]*	*enum* in JSON Schema
examples : *any*	*examples* in JSON Schema
exclusiveMaximum : *number*	*exclusiveMaximum* in JSON Schema
exclusiveMinimum : *number*	*exclusiveMinimum* in JSON Schema
extDocsDescription : *string*	*description* in *External Documentation* object
extDocsUrl : *string*	*url* in *External Documentation* object
format : *string*	*format* in JSON Schema
maxItems : *number*	*maxItems* in JSON Schema
maxLength : *number*	*maxLength* in JSON Schema
maxProperties : *number*	*maxProperties* in JSON Schema
maximum : *number*	*maximum* in JSON Schema
minItems : *number*	*minItems* in JSON Schema
minLength : *number*	*minLength* in JSON Schema
minProperties : *number*	*minProperties* in JSON Schema
minimum : *number*	*minimum* in JSON Schema
multipleOf : *number*	*multipleOf* in JSON Schema
pattern : *string*	*pattern* in JSON Schema
readOnly : *boolean*	*readOnly* in JSON Schema
required : *string*	*required* in JSON Schema
title : *string*	*title* in JSON Schema
type : *string*	*type* in JSON Schema
uniqueItems : *boolean*	*uniqueItems* in JSON Schema
writeOnly : *boolean*	*writeOnly* in JSON Schema
x-collectionOn : *string*	proposed semantic annotation *x-collectionOn*
x-kindOf : *string*	proposed semantic annotation *x-kindOf*
x-refersTo : *string*	proposed semantic annotation *x-refersTo*

The *x-refersTo* and *x-kindOf* fields correspond to the semantic annotations proposed for OpenAPI that apply to *Schema* objects. A query on *x-refersTo* will return table entries in which *x-refersTo* or *x-kindOf* are a subclass of the given value, including the given value. A query on *x-kindOf* will return the same entries as a query on *x-refersTo*, excluding entries in which *x-refersTo* equals the given value, because *x-kindOf* denotes a specialization of the concept and not the concept itself.

According to OpenAPI Specification, extension properties in *Schema* objects may omit the "x-" prefix from their name. To support this, our system allows the user to query fields with any name in *Schema* table.

We generally add one entry to *Schema* table for each *Schema* object defined. However, for complex *Schema* objects that are a composition of other *Schema* objects, we may add more than one entries to *Schema* table. The exact process for this will be described in a later chapter.

The table entry produced by the example above is the following:

Table fields	**Entry values**
minLength	2
type	"string"

...	NULL

Another example of a *Schema* object is the following:

```
{
  "content": {
    "application/json": {
      "schema": {
        "type": "object",
        "properties":{
          "temperature": {
            "type": "number"
          }
        }
      }
    }
  }
}
```

This example describes a value which is a JSON object containing one property. This property is named "temperature" and is a number. The entry that would be produced by this example is the following:

Table fields	**Entry values**
type	"number"
...	NULL

The information inside "properties" keyword in this example would be stored in an entry in *Property* table which will be described below.

3.10 Property

As mentioned previously, when a *Schema* object describes a JSON object, it also describes the properties of that object. This is done with the keywords "properties", "patternProperties", "unevaluatedProperties" and "additionalProperties" which are defined in JSON Schema. An example is below:

```
{
  "type": "object",
  "properties":{
    "firstName": {
      "type": "string"
    },
    "lastName":{
      "type": "string"
    },
    "age":{
      "type": "number"
    }
  }
}
```

This example describes a JSON object with 3 properties: "firstName", "lastName" and "age". Each property is described by another *Schema* object. These other *Schema* objects declare that the "firstName" and "lastName" are strings and the "age" is a number. Most fields of *Property* table correspond to fields in these *Schema* objects that describe properties.

There are also some fields in *Property* table that correspond to fields in *Encoding* and *XML* objects. These OpenAPI objects provide additional information about the encoding and the XML representation of some properties.

The fields of *Property* table and their corresponding OpenAPI fields are shown below:

Property	**Corresponding OpenAPI field**
allowReserved : *boolean*	*allowReserved* in *Encoding* object
const : *any*	*const* in JSON Schema
contentEncoding : *string*	*contentEncoding* in JSON Schema
contentMediaType : *string*	*contentMediaType* in JSON Schema
contentType : *string*	*contentType* in *Encoding* object
default : *any*	*default* in JSON Schema
deprecated : *boolean*	*deprecated* in JSON Schema
description : *string*	*description* in JSON Schema
enum : *any[]*	*enum* in JSON Schema
examples : *any*	*examples* in JSON Schema
exclusiveMaximum : *number*	*exclusiveMaximum* in JSON Schema
exclusiveMinimum : *number*	*exclusiveMinimum* in JSON Schema
explode : *boolean*	*explode* in *Encoding* object
format : *string*	*format* in JSON Schema
maxItems : *number*	*maxItems* in JSON Schema
maxLength : *number*	*maxLength* in JSON Schema
maxProperties : *number*	*maxProperties* in JSON Schema
maximum : *number*	*maximum* in JSON Schema
minItems : *number*	*minItems* in JSON Schema
minLength : *number*	*minLength* in JSON Schema
minProperties : *number*	*minProperties* in JSON Schema
minimum : *number*	*minimum* in JSON Schema
multipleOf : *number*	*multipleOf* in JSON Schema
name : *string*	property names inside *properties*, *patternProperties* and *unevaluatedProperties*
pattern : *string*	*pattern* in JSON Schema
readOnly : *boolean*	*readOnly* in JSON Schema
required : *string*	*required* in JSON Schema
style : *string*	*style* in *Encoding* object
title : *string*	*title* in JSON Schema
type : *string*	*type* in JSON Schema
uniqueItems : *boolean*	*uniqueItems* in JSON Schema
writeOnly : *boolean*	*writeOnly* in JSON Schema
x-collectionOn : *string*	proposed semantic annotation *x-collectionOn*

x-kindOf : *string*	proposed semantic annotation *x-kindOf*
x-refersTo : *string*	proposed semantic annotation *x-refersTo*
xmlAttribute : *boolean*	*attribute* in *XML* object
xmlName : *string*	*name* in *XML* object
xmlNamespace : *string*	*namespace* in *XML* object
xmlPrefix : *string*	*prefix* in *XML* object
xmlWrapped : *boolean*	*wrapped* in *XML* object

The *x-refersTo* and *x-kindOf* fields correspond to the semantic annotations proposed for OpenAPI. Queries on them behave exactly like queries on the respective fields with the same name in *Schema* table.

Similarly with *Schema* table, our system allows the user to query fields with any name in *Property* table.

The example above would produce the following 3 entries in *Property* table:

Table fields	**Entry 1 values**	**Entry 2 values**	**Entry 3 values**
name	"firstName"	"lastName"	"age"
type	"string"	"string"	"number"
...	NULL	NULL	NULL

3.11 Item

When a *Schema* object describes an array, it also describes the format of the array's items. This is done using the keywords "items", "prefixItems", "unevaluatedItems" and "contains" which are defined in JSON Schema. An example of a *Schema* object is below:

```
{
  "type": "array",
  "items":{
    "type": "number",
    "minimum": 0
  }
}
```

This example describes an array and the items of the array are described by another *Schema* object. In this case, each item is specified to be a number greater than or equal to 0. The fields of *Item* table correspond to fields in these *Schema* objects that describe items.

The fields of *Item* table and their corresponding OpenAPI fields are shown below:

Item	**Corresponding OpenAPI field**
const : *any*	*const* in JSON Schema
contentEncoding : *string*	*contentEncoding* in JSON Schema
contentMediaType : *string*	*contentMediaType* in JSON Schema
default : *any*	*default* in JSON Schema
deprecated : *boolean*	*deprecated* in JSON Schema

description : *string*	*description* in JSON Schema
enum : *any[]*	*enum* in JSON Schema
examples : *any*	*examples* in JSON Schema
exclusiveMaximum : *number*	*exclusiveMaximum* in JSON Schema
exclusiveMinimum : *number*	*exclusiveMinimum* in JSON Schema
format : *string*	*format* in JSON Schema
maxItems : *number*	*maxItems* in JSON Schema
maxLength : *number*	*maxLength* in JSON Schema
maxProperties : *number*	*maxProperties* in JSON Schema
maximum : *number*	*maximum* in JSON Schema
minItems : *number*	*minItems* in JSON Schema
minLength : *number*	*minLength* in JSON Schema
minProperties : *number*	*minProperties* in JSON Schema
minimum : *number*	*minimum* in JSON Schema
multipleOf : *number*	*multipleOf* in JSON Schema
pattern : *string*	*pattern* in JSON Schema
readOnly : *boolean*	*readOnly* in JSON Schema
required : *string*	*required* in JSON Schema
title : *string*	*title* in JSON Schema
type : *string*	*type* in JSON Schema
uniqueItems : *boolean*	*uniqueItems* in JSON Schema
writeOnly : *boolean*	*writeOnly* in JSON Schema
x-collectionOn : *string*	proposed semantic annotation *x-collectionOn*
x-kindOf : *string*	proposed semantic annotation *x-kindOf*
x-refersTo : *string*	proposed semantic annotation *x-refersTo*

Same as with *Schema* and *Property* tables, the *x-refersTo* and *x-kindOf* fields correspond to the semantic annotations proposed for OpenAPI and queries on them behave as described previously.

Our system allows the user to query fields with any name in *Item* table, same as for *Schema* and *Property*.

The example above would produce the following entry in *Item* table:

Table fields	**Entry values**
minimum	0
type	"number"
...	NULL

3.12 Security

The fields of *Security* table correspond to fields in *Security Requirement*, *Security Scheme*, *Oauth Flows* and *Oauth Flow* objects. These objects contain general information about the security mechanisms defined for a web service. Each request, callback and webhook can require its own security mechanisms.

An example of a *Security Scheme* object is below:

```
{
  "exampleScheme":{
      "type": "oauth2",
      "flows": {
        "implicit": {
          "authorizationUrl": "https://example.com/api/oauth/dialog",
          "scopes": {
            "write:pets": "modify pets in your account",
            "read:pets": "read your pets"
          }
        }
      }
    }
  }
```

This defines a security mechanism named "exampleScheme" which uses an Oauth Implicit flow with "write:pets" and "read:pets" scopes. An operation can then use a *Security Requirement* object to require the security mechanism with specific scopes as shown in the example below:

```
{
  "security": [
    {
      "exampleScheme": [
        "read:pets"
      ]
    }
  ]
}
```

In the table below are the fields of *Security* table with the corresponding fields in OpenAPI:

Security	**Corresponding OpenAPI field**
apiKeyIn : *string*	*in* in *Security Scheme* object
apiKeyName : *string*	*name* in *Security Scheme* object
description : *string*	*description* in *Security Scheme* object
httpBearerFormat : *string*	*bearerFormat* in *Security Scheme* object
httpScheme : *string*	*scheme* in *Security Scheme* object
name : *string*	property name inside *Security Requirement* object
oauth2ClientCredRefreshUrl : *string*	*flows.clientCredentials.refreshUrl* in *Security Scheme* object
oauth2ClientCredTokenUrl : *string*	*flows.clientCredentials.tokenUrl* in *Security Scheme* object
oauth2CodeAuthUrl : *string*	*flows.authorizationCode.authorizationUrl* in *Security Scheme* object
oauth2CodeRefreshUrl : *string*	*flows.authorizationCode.refreshUrl* in *Security Scheme* object
oauth2CodeTokenUrl : *string*	*flows.authorizationCode.tokenUrl* in *Security Scheme* object
oauth2ImplAuthUrl : *string*	*flows.implicit.authorizationUrl* in *Security Scheme* object

oauth2ImplRefreshUrl : *string*	*flows.implicit.refreshUrl* in *Security Scheme* object
oauth2PassRefreshUrl : *string*	*flows.password.refreshUrl* in *Security Scheme* object
oauth2PassTokenUrl : *string*	*flows.password.tokenUrl* in *Security Scheme* object
openIdConnectUrl : *string*	*openIdConnectUrl* in *Security Scheme* object
type : *string*	*type* in *Security Scheme* object

We add one entry to *Security* table for each security mechanism required by an operation. The example above would produce the following entry:

Table fields	**Entry values**
name	"exampleScheme"
oauth2ImplAuthUrl	"https://example.com/api/oauth/dialog"
type	"oauth2"
...	NULL

3.13 SecurityScope

When a request, callback or webhook requires a security mechanism, it can also require specific scopes of the mechanism. The fields of *SecurityScope* table correspond to the names and descriptions of the defined scopes that are located in *Oauth Flow* objects.

In the table below are the fields of *SecurityScope* with the corresponding fields in OpenAPI:

SecurityScope	**Corresponding OpenAPI field**
description : *string*	property value inside *scopes* in *Oauth Flow* object
name : *string*	property name inside *scopes* in *Oauth Flow* object

We add one entry to *SecurityScope* for each scope required by an operation. The example shown in section 3.12 would produce the following entry:

Table fields	**Entry values**
description	"read your pets"
name	"read:pets"

3.14 Link

The fields of *Link* table correspond to fields in *Link*, *Response* and *Server* objects. These objects describe links contained in a response. These links show a relationship between the response and another operation but they are not mandatory for the client to follow. An example is:

```
{
  "links": {
    "address": {
      "operationId": "getUserAddressByUUID",
      "parameters": {
        "userUuid": "$response.body#/uuid"
      }
    }
  }
}
```

This example defines a link named "address", referencing the operation with id "getUserAddressByUUID", with a parameter named "userUuid".

In the table below are the fields of *Link* with the corresponding fields in OpenAPI:

Link	**Corresponding OpenAPI field**
description : *string*	*description* in *Link* object
name : *string*	property names inside *links* in *Response* object
operationId : *string*	*operationId* in *Link* object
operationRef : *string*	*operationRef* in *Link* object
requestBody : *string*	*requestBody* in *Link* object
serverDescription : *string*	*description* in *Server* object
url : *string*	*url* in *Server* object

We add one entry to *Link* table for each link defined in a response. The example above would produce the following entry:

Table fields	**Entry values**
name	"address"
operationId	"getUserAddressByUUID"
...	NULL

3.15 LinkParameter

A link contained in a response can specify parameters to be passed when it is followed. The fields of *LinkParameter* table correspond to the name and value of these parameters located in *Link* object.

In the table below are the fields of *LinkParameter* with the corresponding fields in OpenAPI:

LinkParameter	**Corresponding OpenAPI field**
name : *string*	property names inside *parameter* in *Link* object
value : *any*	property values inside *value* in *Link* object

We add one entry to *LinkParameter* for each parameter defined in a link. The example in section 3.14 would produce the following entry:

Table fields	Entry values
name	"userUuid"
value	"$response.body#/uuid"

3.16 Server

The fields of *Server* table correspond to fields in *Server* object. Each request, webhook or callback can define its own servers which are described by *Server* objects. An example is:

```
{
  "url": "https://development.gigantic-server.com/v1",
  "description": "Development server"
}
```

In the table below are the fields of *Server* with the corresponding fields in OpenAPI:

Server	Corresponding OpenAPI field
description : *string*	*description* in *Server* object
url : *string*	*url* in *Server* object

We add one entry to *Server* table for each server in an operation. The example above would produce the following entry:

Table fields	Entry values
description	"Development server"
url	"https://development.gigantic-server.com/v1"

3.17 ServerVariable

The fields of *ServerVariable* table correspond to fields in *Server Variable* and *Server* objects. These objects describe variables in the server's URL that can have different values. An example is:

```
{
  "url": "https://gigantic-server.com:{port}/",
  "description": "The production API server",
  "variables": {
    "port": {
      "enum": [
        "8443",
        "443"
      ],
      "default": "8443"
    }
  }
}
```

This example defines a server with a variable named "port" in its URL which can be either 8443 or 443.

In the table below are the fields of *ServerVariable* with the corresponding fields in OpenAPI:

ServerVariable	**Corresponding OpenAPI field**
default : *any*	*default* in *Server Variable* object
description : *string*	*description* in *Server Variable* object
enum : *any[]*	*enum* in *Server Variable* object
name : *string*	property names inside *variables* in *Server* object

We add one entry to *ServerVariable* table for each variable defined in a server. The example above would produce the following entry:

Table fields	**Entry values**
default	"8443"
description	NULL
enum	["8443", "443"]
name	"port"

3.18 Example

The fields of *Example* table correspond to fields in *Example*, *Media Type*, *Parameter* and *Header* objects. These objects provide examples of an operation's payload, a response's payload, a parameter or a header. An example is:

```
{
  "examples": {
    "foo": {
      "summary": "A foo example",
      "value": 35
    }
  }
}
```

This defines an example named "foo" with a value of 35.

In the table below are the fields of *Example* with the corresponding fields in OpenAPI:

Example	**Corresponding OpenAPI field**
description : *string*	*description* in *Example* object
externalValue : *any*	*externalValue* in *Example* object
name : *string*	property names inside *examples* in *Media Type*, *Parameter* or *Header* objects
summary : *string*	*summary* in *Example* object
value : *any*	*value* in *Example* object

We add one entry to *Example* table for each example provided. The example above would produce the following entry:

Table fields	**Entry values**
name	"foo"
summary	"A foo example"
value	35
...	NULL

The *Media Type*, *Parameter* and *Header* objects in OpenAPI also provide the field *example* whose value is a single example instead of an *Example* object. To support this field, we add an entry in *Example* table with *value* set to the field's value and all other table fields NULL.

Chapter 4

OpenAPI Query Language 2

OAQL2 is a language for searching in OpenAPI descriptions. Its syntax is designed to be very similar to SQL so that a user who already knows SQL can easily use OAQL2.

OAQL2 queries are executed on the tables defined previously and function like SQL queries. Each query consists of the following parts:

1. *SELECT* clause: determines which table fields should be returned to the user
2. *FROM* clause: specifies the tables to be joined
3. *WHERE* clause (optional): specifies conditions that the entries of the resulting table need to satisfy
4. *ORDER BY* clause (optional): specifies the table fields to sort the results by

4.1 SELECT clause

In *SELECT* clause, the user specifies the table fields that will be returned in the result. Each field is denoted by the name or alias of the table it belongs to, followed by a dot and then by the field name. The user can optionally give an alias to some table fields.

Instead of a table field, the user can write the name or alias of a table followed by the suffix ".*". In this case, all fields of the specified table will be returned, including any fields corresponding to extension properties.

Instead of specifying table fields in *SELECT* clause, the user can write "SELECT *". This will return all fields from all tables in *FROM* clause, just like in SQL.

Additionally, the user can include the keyword DISTINCT immediately after the SELECT keyword. This will ensure that the result will not contain two table entries with identical values in all of their fields.

Some examples are:

Example query	**Description**

SELECT s.id AS service_id, s.title
FROM Service s
	Returns *Service.id* field renamed to "service_id" and *Service.title* field renamed to "s.title"

SELECT s.*
FROM Service s
	Returns all fields from *Service* table

SELECT *
FROM Service s
	Returns all fields from all tables in *FROM* clause (in this case only *Service*)

SELECT DISTINCT Request.method
FROM Request
	Returns each different value in *Request.method* field once

4.2 FROM clause

In *FROM* clause, the user specifies tables that will be joined to form a single large table, similarly with SQL queries. Everything else specified in an OAQL2 query (filtering table entries, sorting table entries and selecting table fields) will happen on that table.

In OAQL2, the relationships between two tables are one-to-many, meaning that each entry of one table can be joined with 0 or more entries of the other table. This happens because the tables correspond to OpenAPI objects and an object can contain 0 or more other objects. For example, each request can contain 0 or more responses and each response can contain 0 or more headers. Each join corresponds to a parent-child relationship between the OpenAPI objects that the tables correspond to.

Below is a list with all possible joins between two tables:

Callback - Example	Property - Header	Schema - Property
Callback - Parameter	Property - Property	Security - SecurityScope
Callback - Response	Request - Callback	Server - ServerVariable
Callback - Schema	Request - Example	Service - Request
Callback - Security	Request - Parameter	Service - Tag
Callback - Server	Request - Response	Service - Webhook
Header - Example	Request - Schema	Tag - Schema
Header - Schema	Request - Security	Webhook - Example
Item - Item	Request - Server	Webhook - Parameter
Item - Property	Response - Example	Webhook - Response
Link - LinkParameter	Response - Header	Webhook - Schema
Link - ServerVariable	Response - Link	Webhook - Security
Parameter - Example	Response - Schema	Webhook - Server
Parameter - Schema	Schema - Item	

In the pairs of tables above, each entry of the first table can be joined with 0 or more entries of the second table. The only exception is the pair of *Item* and *Property* tables, for which the reverse is also valid.

Note that the join between *Tag* and *Schema* tables is allowed because of the *x-onResource* semantic annotation which links a tag to a schema object.

In a relational database, all tables have a primary key which is a table field (or set of fields) uniquely identifying each entry in the table. To join two tables, one of them must have a foreign key, which is a field referencing the primary key of the other table. The primary and the foreign key used are specified in the join condition for each join in the *FROM* clause of the SQL query. This is required because SQL needs to support querying in any relational database schema.

In OAQL2, the join condition for each possible join between two tables is predefined. The system can determine each join condition from the names of the tables that are being joined. Therefore, the user does not need to write join conditions in OAQL2. This eliminates the need to define primary and foreign keys in OAQL2 tables because they would never be used. This is the reason why we did not define any primary or foreign keys in the tables presented in the previous chapter.

A *FROM* clause in OAQL2 has the following form:

```
FROM <table1>
JOIN <table2> ON <table1>
JOIN ...
```

For example, the following query returns the path of each request along with the title of the web service it is defined in:

```
SELECT Service.title, Request.path
FROM Service
JOIN Request ON Service
```

As we can see, the join condition, which would normally be following the "ON" keyword, is omitted. This happens for simplicity because, as mentioned, the join condition between *Service* and *Request* tables is predefined and there is no need to be written by the user. The only information the "ON" keyword needs to provide is that the join will happen with *Service* table. This is needed for queries with many joins, as shown later.

The user can optionally specify aliases for some tables and then refer to them by their alias. The query above will have the same results as the following:

```
SELECT s.title, r.path
FROM Service s
JOIN Request r ON s
```

The only difference will be the name of the table fields in the results.

The following query joins the *Request* table with *Response* and *Schema* and returns all fields of the result:

```
SELECT *
FROM Request r
JOIN Response res ON r
JOIN Schema s ON r
```

The first "ON" keyword specifies that the *Response* table will be joined with *Request* and the second "ON" keyword specifies that the *Schema* table will be joined with *Request*. Note that, without the "ON" keyword, it would be ambiguous whether the *Schema* table will be joined with *Request* or *Response*.

Below is an example in which it is necessary to specify aliases for the *Schema* tables:

```
SELECT s1.type, s2.type
FROM Request r
JOIN Response res
JOIN Schema s1 ON r
JOIN Schema s2 ON res
```

This query returns the type of the request and response payload for each pair of request and response defined. The aliases are needed to specify which of the two instances of *Schema* table we refer to.

Note that the order in which the tables are written in *FROM* clause does not matter. For example, the following two queries will both return the request path and response code for each pair of request and response defined:

```
SELECT r.path, res.statusCode
FROM Request r
JOIN Response res ON r
```

```
SELECT r.path, res.statusCode
FROM Response res
JOIN Request r ON res
```

As mentioned before, each join corresponds to a parent-child relationship between OpenAPI objects. OAQL2 finds which table corresponds to the parent object (*Request*) and which to the child (*Response*) and joins them. If an entry of the "parent" table cannot be joined with any entry of the "child" table (in this example this happens when a request defines no responses), it is included in the results with NULL values in the fields of the "child" table. This is the equivalent of a LEFT JOIN in SQL between the "parent" and the "child" table.

There is one case that it is not clear which table corresponds to the parent OpenAPI object and which to the child object. This case is when we join a *Property* or *Item* table with another *Property* or *Item* because each property or item might be a JSON object containing other properties or an array containing other items. To make the distinction between parent and child, we use a different join condition. This is demonstrated in the examples below, where we show all 4 possible combinations of joining a *Property* or *Item* with another *Property* or *Item*.

The *Schema* object in the following example defines a JSON object containing a property named "details". This property is also a JSON object containing another property "address".

```
{
  "type": "object",
  "properties":{
    "details":{
      "type": "object",
      "properties":{
        "address":{
          "type": "string"
        }
      }
    }
  }
}
```

To make a query about this example, we would need to join two *Property* tables like in the query below:

```
SELECT *
FROM Property parentProp
JOIN Property childProp ON parentProp.property = childProp
```

The table with the alias "parentProp" refers to the parent property ("details") that is a JSON object, while the table with the alias "childProp" refers to the child property ("address") contained inside the JSON object of the parent property.

The *Schema* object in the following example defines a JSON object containing a property named "list". This property is an array of numbers.

```
{
  "type": "object",
  "properties":{
    "list":{
      "type": "array",
      "items":{
        "type": "number"
      }
    }
  }
}
```

To make a query about this example, we would need to join a *Property* with an *Item* table like in the query below:

```
SELECT *
FROM Property parentProp
JOIN Item childItem ON parentProp.item = childItem
```

The table with the alias "parentProp" refers to the parent property ("list") that is an array, while the table with the alias "childItem" refers to the items (numbers) inside the array.

The *Schema* object in the following example defines an array of JSON objects. Each JSON object contains a property named "age".

```
{
  "type": "array",
  "items":{
    "type": "object",
    "properties":{
      "age":{
        "type": "number"
      }
    }
  }
}
```

To make a query about this example, we would need to join an *Item* with a *Property* table like in the query below:

```
SELECT *
FROM Item parentItem
JOIN Property childProp ON parentItem.property = childProp
```

The table with the alias "parentItem" refers to the items of the array (the JSON objects), while the table with the alias "childProp" refers to the properties of these JSON objects ("age").

The *Schema* object in the following example defines an array containing other arrays which contain numbers.

```
{
  "type": "array",
  "items":{
    "type": "array",
    "items":{
      "type": "number"
    }
  }
}
```

To make a query about this example, we would need to join an *Item* with another *Item* table like in the query below:

```
SELECT *
FROM Item parentItem
JOIN Item childItem ON parentItem.item = childItem
```

The table with the alias "parentItem" refers to the items of the array (the child arrays), while the table with the alias "childItem" refers to the items of the child arrays (the numbers).

The condition following the ON keyword determines which table corresponds to the parent object and which to the child. More generally, its form is either

"`ON <parent_table>.property = <child_table>`"

or

"`ON <parent_table>.item = <child_table>`"

depending on whether the child table is *Property* or *Item* respectively.

4.3 WHERE clause

In *WHERE* clause, the user can specify various conditions and any entry that does not satisfy these conditions is discarded. OAQL2 supports some of the standard SQL operators:

- $=$, $<>$, $>$, $<$, $>=$, $<=$: note that, when these operators are used with an array, the condition is true if at least one element of the array satisfies it.
- IS NULL and IS NOT NULL: a table field is null when the corresponding field in the OpenAPI description has no value
- BETWEEN: checks if a table field's value belongs in a range of values

- LIKE: searches for a pattern in a table field containing a string value

Parentheses can be used to prioritize evaluation of a condition and two or more conditions can be combined by the logical operators AND, OR, XOR.

OAQL2 also supports checking for equality of two table fields in an entry with the "=" operator. The main purpose of this is to facilitate querying the *x-collectionOn* field in *Schema*, *Item* or *Property* tables. By checking if *x-collectionOn* is equal to *Property.name* the user can find the property of an object that holds a collection of items.

Additionally, in OAQL2 we define the operator IN but it functions differently from the SQL operator IN. Its syntax is "`fieldA IN fieldB`" where fieldB is an array. This operator returns true if the value of fieldA equals an element of the array in fieldB. The main purpose of this operator is to allow the user to check if the value of *Tag.name* is in the array in *Request.tags*.

4.4 ORDER BY clause

The *ORDER BY* clause sorts the table entries in the result by a field and is identical to ORDER BY in SQL. It sorts in either ascending or descending order, depending on whether the ASC or DESC keyword was specified respectively and defaults to ascending order if no keyword was specified.

If more than one table fields are specified, the sorting is initially done using the first field. For entries with the same value in that field, the next field is used and so on.

4.5 Example queries

In this section, we present some examples of OAQL2 queries involving the semantic annotations proposed for OpenAPI.

The query below finds services with a request whose payload is annotated to the "Person" concept:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Schema sc ON r
WHERE sc.x-refersTo = "https://schema.org/Person"
```

The query below finds services with a response whose payload is annotated to a specialization of the "Store" concept:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Response res ON r
JOIN Schema sc ON res
WHERE sc.x-kindOf = "https://schema.org/Store"
```

The query below finds services with a request that is annotated to the "SearchAction" concept:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.x-operationType = "https://schema.org/SearchAction"
```

The query below finds services whose request payload is a collection of items. This collection is provided in a separate property and this property refers to the "ProductCollection" concept:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Schema sc ON r
JOIN Property p ON sc
WHERE sc.x-collectionOn = p.name
AND p.x-refersTo = "https://schema.org/ProductCollection"
```

The query below finds the path and method of requests grouped by a tag. This tag must be linked to a *Schema* object annotated to the "Movie" concept:

```
SELECT r.path, r.method
FROM Service s
JOIN Request r ON s
JOIN Tag t ON s
JOIN Schema sc ON t
WHERE t.name IN r.tags
AND sc.x-refersTo = "https://schema.org/Movie"
```

4.6 Syntax

Below we present the BNF (Backus-Naur form) of OAQL2:

```
⟨query⟩ ::= ⟨fromClause⟩ ⟨selectClauseFinal⟩ ⟨whereClause⟩ ⟨orderByClause⟩
  | ⟨fromClause⟩ ⟨selectClauseFinal⟩ ⟨orderByClause⟩
  | ⟨fromClause⟩ ⟨selectClauseFinal⟩ ⟨whereClause⟩
  | ⟨fromClause⟩ ⟨selectClauseFinal⟩

⟨selectClauseFinal⟩ ::= ⟨selectClause⟩
  | ⟨selectKeyword⟩ *

⟨selectClause⟩ ::= ⟨selectClause⟩ , ⟨field⟩ ⟨alias⟩
  | ⟨selectClause⟩ , ⟨field⟩
  | ⟨selectClause⟩ , ⟨identifier⟩ .*
  | ⟨selectKeyword⟩ ⟨field⟩ ⟨alias⟩
  | ⟨selectKeyword⟩ ⟨field⟩
  | ⟨selectKeyword⟩ ⟨identifier⟩ .*

⟨selectKeyword⟩ ::= SELECT DISTINCT
  | SELECT

⟨alias⟩ ::= AS ⟨identifier⟩
  | ⟨identifier⟩
```

```
⟨orderByClause⟩ ::= ⟨orderByClause⟩ , ⟨field⟩ ASC
  | ⟨orderByClause⟩ , ⟨field⟩ DESC
  | ⟨orderByClause⟩ , ⟨field⟩
  | ORDER BY ⟨field⟩ ASC
  | ORDER BY ⟨field⟩ DESC
  | ORDER BY ⟨field⟩

⟨whereClause⟩ ::= WHERE ⟨condition⟩

⟨condition⟩ ::= ⟨field⟩ ⟨operator⟩ ⟨value⟩
  | ⟨field⟩ ⟨operator⟩ ⟨field⟩
  | ⟨field⟩ IN ⟨field⟩
  | ⟨field⟩ BETWEEN ⟨value⟩ AND ⟨value⟩
  | ⟨field⟩ NOT BETWEEN ⟨value⟩ AND ⟨value⟩
  | ⟨field⟩ IS NULL
  | ⟨field⟩ IS NOT NULL
  | ⟨field⟩ LIKE ⟨value⟩
  | ( ⟨condition⟩ )
  | ⟨condition⟩ AND ⟨condition⟩
  | ⟨condition⟩ OR ⟨condition⟩
  | ⟨condition⟩ XOR ⟨condition⟩

⟨field⟩ ::= ⟨identifier⟩ . ⟨identifier⟩

⟨fromClause⟩ ::= ⟨fromClause⟩ JOIN ⟨identifier⟩ ⟨alias⟩ ON ⟨field⟩ = ⟨identifier⟩
  | ⟨fromClause⟩ JOIN ⟨identifier⟩ ⟨alias⟩ ON ⟨identifier⟩ = ⟨field⟩
  | ⟨fromClause⟩ JOIN ⟨identifier⟩ ON ⟨field⟩ = ⟨identifier⟩
  | ⟨fromClause⟩ JOIN ⟨identifier⟩ ON ⟨identifier⟩ = ⟨field⟩
  | ⟨fromClause⟩ JOIN ⟨identifier⟩ ⟨alias⟩ ON ⟨identifier⟩
  | ⟨fromClause⟩ JOIN ⟨identifier⟩ ON ⟨identifier⟩
  | FROM ⟨identifier⟩ ⟨alias⟩
  | FROM ⟨identifier⟩

⟨operator⟩ ::= = | <> | < | > | <= | >= | LIKE

⟨value⟩ ::= ” ” | ” ⟨literal⟩ ” | ⟨literal⟩ | true | false

⟨identifier⟩ ::= ⟨identifier⟩ ⟨letter⟩
  | ⟨identifier⟩ ⟨digit⟩
  | ⟨identifier⟩ -
  | ⟨identifier⟩ _
  | ⟨letter⟩

⟨literal⟩ ::= ⟨character⟩ | ⟨literal⟩ ⟨character⟩

⟨character⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨symbol⟩

⟨letter⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U
    | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
    r | s | t | u | v | w | x | y | z
```

$\langle digit \rangle$ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

$\langle symbol \rangle$ ::= `|` | | `!` | `#` | `$` | `%` | `␣&` | `(` | `)` | `*` | `+` | `,` | `-` | `.` | `/` | `:` | `;` | `>` | `=` | `<`
| `?` | `@` | `[` | `]` | `^` | `_` | `‘` | `{` | `}` | `~` | `’` | `\` | `\”`

Chapter 5

Implementation

Our service is composed of two other services: a Java server that handles the requests from the clients and a MongoDB database for storing and querying OpenAPI descriptions.

5.1 Choice of tools

Despite OAQL2 being very similar to SQL, we use a NoSQL database to store the data. The reasons for that are:

- to avoid the performance cost of joins: The information in OAQL2 tables has the same hierarchical structure as an OpenAPI description since the tables correspond to OpenAPI objects. By storing it as a JSON document in a NoSQL database we keep this structure intact. Otherwise, we would separate the data and group them into tables which then would have to be joined for each query.

- to support not strictly defined data types and sizes: A relational database requires that table fields declare their data type and their size. However, some fields in an OpenAPI document can have values of different types and there is no limit to their size. A workaround would be to use the JSON data type (in relational databases that support it) but that would lead to slower queries.

In a 2016 publication[11], the performance of five NoSQL databases is compared (Redis, MongoDB, Couchbase, Cassandra, HBase). Redis is shown to be the fastest, with MongoDB in the second place. Redis, however, is a key-value database and it is not meant to be used with JSON objects. Another consideration is the graph database Neo4j, which is a relatively new system, but a 2020 publication[12], which compares it to MongoDB and PostgreSQL, shows that Neo4j is slower than MongoDB. For these reasons we chose MongoDB to be used in our service.

In order to parse OAQL2 queries, we use a lexical analyzer and a parser that were generated using *JFlex* and *Java CUP* respectively. These two tools are the Java equivalent of the traditional *lex* and *yacc* tools.

As mentioned in the previous chapter, when querying one of *x-refersTo*, *x-kindOf* or *x-operationType* fields, we also want to return table entries whose value is a subclass of the given value. We need a reasoner to find these subclasses because, for briefness, a semantic model usually omits relationships between concepts that can be inferred from other relationships. *Apache Jena*[1] is a Java framework for building Semantic Web applications and it provides various reasoners. We use the *TransitiveReasoner* which is a lightweight reasoner supporting only the transitive and reflexive properties of *rdfs:subClassOf* and *rdfs:subPropertyOf*. In this thesis we use the semantic model from *Schema.org*[2], which the service loads during startup.

5.2 Description of service

An OpenAPI document contains some of its information as property names in its objects instead of values. For example, the following defines an endpoint on path "/pets" which accepts POST requests:

```
{
  "/pets": {
    "post": {
      "description": "...",
      "requestBody": {
        ...
      }
    }
  }
}
```

Both the path and the method are property names with their values being other OpenAPI objects. This makes querying difficult since the path to other fields depends on these names. It also makes indexing impossible because indexes are built on fields located on a predefined path.

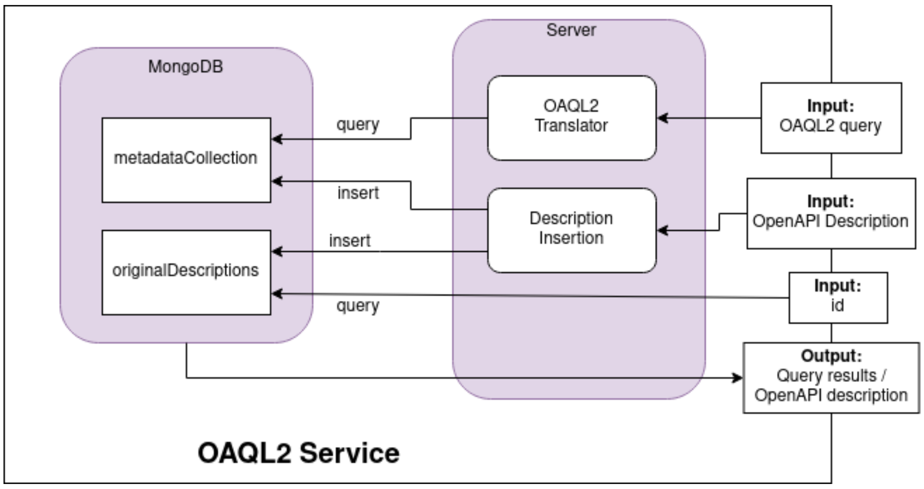
In addition, some of the information may be located in one of a few different places in an OpenAPI description. Thus, querying would require searching in all of these places and deciding which values are overwriting the rest in each case. For example, the servers for a request can be declared in the *OpenAPI* object, the *Path Item* object or the *Operation* object.

To solve these problems, we create and store one metadata object for each OpenAPI description in the service. Each metadata object is a JSON object containing all necessary information in a form that can be queried and indexed easily. All OAQL2 queries are executed on these objects.

The following figure shows the architecture of the service:

[1] `https://jena.apache.org/`

[2] `https://schema.org/version/latest/schemaorg-current-https.nt`



5.2.1 Server

The server handles each HTTP request on a separate thread and accepts them at the following paths:

Path	Accepts	Returns
/	GET Request.	HTML document providing a user interface for inserting or retrieving OpenAPI descriptions and executing OAQL2 queries
/insertDescription	POST request. Request body must be a valid OpenAPI description	204 code with no response body
/query	POST request. Request body must be a valid OAQL2 query	200 code with the results of the query in the response body
/description/<id>	GET request. <id> must be an id, 24 characters long, found in the *Service.id* field and it corresponds to a single OpenAPI description	200 code with the requested OpenAPI description in the response body or 404 code if there is no OpenAPI description with that id

If the server encounters an error, it will respond with a status code of 400 and an error message in the response body.

The results of a query are returned in JSON format. The response body is an array of JSON objects. Each object represents a table entry of the results and each key-value pair in these objects corresponds to a table field with its value. For example, the following query returns the path and summary of each request defined:

```
SELECT r.method, r.summary
FROM Request r
```

The results will have the following form:

```
[
  {
    "r.method": "...",
    "r.summary": "...",
  },
  {
    "r.method": "...",
    "r.summary": "...",
  },
  ...
]
```

If a table field is NULL in an entry, the key-value pair for the field in that object is omitted. For example, one of the JSON objects in the array above might look like the following:

```
{
  "r.method": "..."
}
```

This means that the value of *Request.summary* field in that table entry is NULL.

If all fields in a table entry are NULL, the JSON object corresponding to that entry is omitted from the response body. Otherwise, the result would include some empty JSON objects (without any key-value pairs) which would provide no information.

5.2.2 Database

The MongoDB service contains one database, named *openapiDB*, with two collections:

- *metadataCollection*: contains the metadata created from the OpenAPI descriptions in the service. We search in this collection when executing an OAQL2 query.
- *originalDescriptions*: contains the original OpenAPI descriptions. We search in this collection when a request to the server's */description/<id>* path is made.

5.3 Metadata format

As mentioned previously, for each OpenAPI description inserted into the service we also store a JSON object with the description's information in a different form for faster querying. The structure of these metadata objects is shown in figure 5.1. Nodes with the same name have identical subtrees in figure 5.1.

Each node in figure 5.1 represents an array of JSON objects in the metadata object, as described below. The top level of a metadata object has the following form:

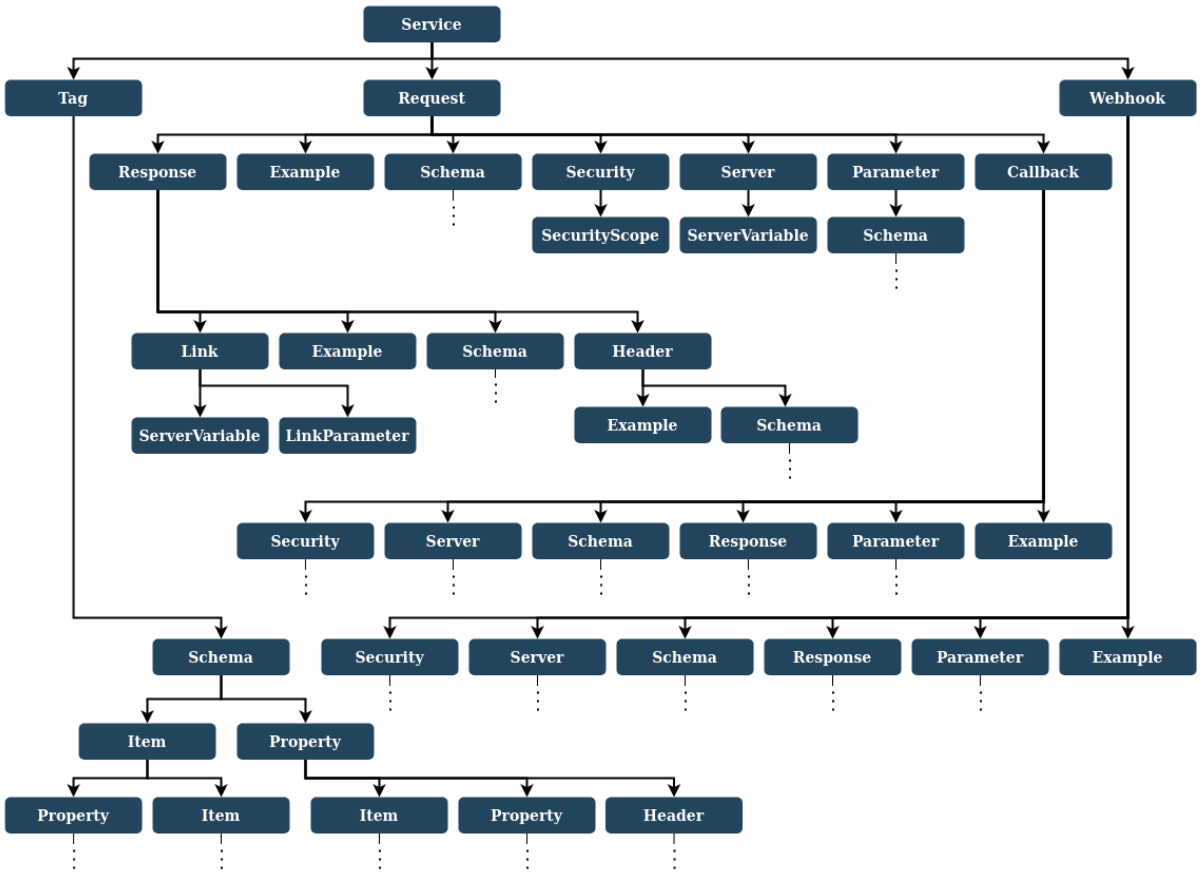


Figure 5.1: Structure of a metadata object

```
{
  "Service":[
    {
      "contactEmail": <value>,
      "contactName": <value>,
      "contactUrl": <value>,
      "description": <value>,
      "extDocsDescription": <value>,
      "extDocsUrl": <value>,
      "id": <value>,
      "jsonSchemaDialect": <value>,
      "licenseName": <value>,
      "licenseUrl": <value>,
      "openapiVersion": <value>,
      "termsOfService": <value>,
      "title": <value>,
      "version": <value>,
      "Request": [ ... ],
      "Tag": [ ... ],
      "Webhook": [ ... ]
    }
  ]
}
```

A metadata object contains an array named "Service" which is represented in figure 5.1 by the top node. The JSON object inside "Service" array contains 3 arrays with the names "Request", "Tag" and "Webhook", which are represented in figure 5.1 by the children of the root node. The JSON object inside "Service" array also contains a key-value pair for each field of *Service* table in OAQL2.

The array named "Request" has the following form:

```
[
  {
    "bodyDescription": <value>,
    "bodyRequired": <value>,
    "contentType": <value>,
    "deprecated": <value>,
    "description": <value>,
    "extDocsDescription": <value>,
    "extDocsUrl": <value>,
    "method": <value>,
    "operationId": <value>,
    "path": <value>,
    "summary": <value>,
    "tags": <value>,
    "x-operationType": <value>,
    "Callback": [ ... ],
    "Example": [ ... ],
    "Parameter": [ ... ],
    "Response": [ ... ],
    "Schema": [ ... ],
    "Security": [ ... ],
    "Server": [ ... ]
  },
  ...
]
```

Each JSON object inside "Request" array contains 7 arrays with the names "Callback", "Example", "Parameter", "Response", "Schema", "Security" and "Server", which are represented in figure 5.1 by the children of "Request" node. Each JSON object inside "Request" array also contains a key-value pair for each field of *Request* table in OAQL2.

The rest of the arrays represented by nodes in figure 5.1 are similarly formed. They contain JSON objects and each JSON object contains other arrays as shown in figure 5.1. Each JSON object also contains one key-value pair for each field of the OAQL2 table with the same name as the array, like shown above. The full format of a metadata object is presented in Appendix A.

Each of the arrays described above corresponds to the OAQL2 table with the same name as the array. Each JSON object inside one of these arrays corresponds to one entry of that table. For example, the JSON object in "Service" array corresponds to one entry in *Service* table and each JSON object in "Request" array corresponds to an entry in *Request* table. The values of the table fields are stored in the properties inside the JSON object that have the same name with the table fields. Below we show an example OpenAPI description and the metadata object that would be produced:

OpenAPI description

```
{
  "openapi": "3.1.0",
  "info": {
    "title": "Example",
    "version": "v1"
  },
  "paths": {
    "/req1": {
      "get": {}
    }
  }
}
```

Metadata object created

```
{
  "Service" : [
    {
      "id" : "624defc8152952076c7b3af6",
      "title" : "Example",
      "openapiVersion" : "3.1.0",
      "version" : "v1",
      "Request" : [
        {
          "path" : "/req1",
          "method" : "get",
          "deprecated" : false
        }
      ]
    }
  ]
}
```

This OpenAPI description defines only a GET request. The metadata object was created by simply copying the information from the OpenAPI description to the corresponding fields in the metadata object. It also contains the automatically generated "id" field and the "deprecated" field.

If a field is not provided in an OpenAPI description, we store its default value in the metadata object (the "deprecated" field in the example above). If there is no default value, we do not insert the field in the metadata object. We also omit empty arrays from the metadata object.

5.4 Algorithm for inserting OpenAPI descriptions

This algorithm is executed each time a user wants to insert an OpenAPI description into the service. It consists of the following steps:

1. create metadata object from new OpenAPI description
2. store new OpenAPI description into *originalDescriptions* collection in MongoDB
3. store metadata object into *metadataCollection* collection in MongoDB

To create the metadata object we mostly copy the information from the fields of the OpenAPI description. However, some additional processing is needed for *Reference* and *Schema* objects.

5.4.1 Parsing Reference objects

Reference objects are used to reference reusable objects defined somewhere else in an OpenAPI description. For example, the following part of an OpenAPI description defines a response with status code 200 and references a *Response* object:

```
{
  "responses":{
    "200":{
      "$ref":"#/components/responses/okResponse"
    }
  }
}
```

In this example, the referenced *Response* object provides only a description and is shown below:

```
{
  ... ,
  "components":{
    "responses":{
      "okResponse":{
        "description": "OK"
      }
    }
  }
}
```

When encountering a *Reference* object we replace it with the referenced object and we continue parsing normally. The example above would be turned into the following:

```
{
  "responses":{
    "200":{
      "description": "OK"
    }
  }
}
```

5.4.2 Parsing Schema objects

The *Schema* object in OpenAPI Specification uses the vocabulary defined in JSON Schema draft 2020-12[3]. The JSON Schema is used to validate a JSON object (called *instance*) against a schema object. The schema specifies rules about the contents and structure of the instance.

A schema that describes an instance which is an object uses other schema objects to describe the properties of the object. These schema objects may be located under the keywords "properties", "patternProperties", "additional-Properties" or "unevaluatedProperties". For simplification, we parse all of the

[3]https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00#section-8

objects under these keys and store them into the "Property" array in the metadata object, which corresponds to the *Property* table. An example is below:

```
{
  "type": "object",
  "properties":{
    "weight": {
      "type": "number"
    }
  },
  "additionalProperties":{
    "type": "string"
  }
}
```

The above example describes a JSON object containing a property named "weight" which is a number. This JSON object may also contain other properties that must be strings.

Similarly, a schema that describes an array uses other schema objects to describe the items of the array. These objects may be located under "items", "prefixItems", "unevaluatedItems" or "contains". We parse and store these objects into the "Item" array in the metadata object, which corresponds to *Item* table. The example below describes an array of strings:

```
{
  "type": "array",
  "items":{
    "type": "string"
  }
}
```

A schema object may be a composition of other schema objects which makes searching difficult and slow. To solve this problem, we convert each schema into an array of simple objects which are stored in the metadata object. Each of these simple objects is one of the possible schemas that an instance can be validated against.

The "allOf" keyword provides an array of schema objects that must be all valid. To simplify it, we merge all key-value pairs of all the subschemas into the parent schema. An example is shown below:

Original	Simplified
`{`	`{`
`  "allOf": [`	`  "multipleOf": [`
`    {`	`    4,`
`      "multipleOf": 4,`	`    5`
`      "type": "number",`	`  ],`
`      "default": 20`	`  "type": "number",`
`    },`	`  "default": 20,`
`    {`	`  "description": "an example"`
`      "multipleOf": 5,`	`}`
`      "type": "number"`	
`    }`	
`  ],`	
`  "description": "an example"`	
`}`	

The "oneOf" keyword provides an array of schema objects, exactly one of which must be valid. To simplify it, we create one object for each schema inside the array under "oneOf". An example is shown below:

Original	Simplified
`{`	`[`
`  "type": "number",`	`  {`
`  "oneOf": [`	`    "multipleOf": 5,`
`    {`	`    "type": "number"`
`      "multipleOf": 5`	`  },`
`    },`	`  {`
`    {`	`    "multipleOf": 3,`
`      "multipleOf": 3`	`    "type": "number"`
`    }`	`  }`
`  ]`	`]`
`}`	

The "anyOf" keyword provides an array of schema objects and at least one of them must be valid. Creating all possible simplified objects would mean creating one object for each possible combination of schemas inside "anyOf" array. We show below an example of an object using the "anyOf" keyword and all possible simplified objects, which are being a multiple of 3, 5 or both:

Original schema	All possible simplified objects
`{ "anyOf": [ { "multipleOf": 5 }, { "multipleOf": 3 } ] }`	`[ { "multipleOf": 3, }, { "multipleOf": 5, }, { "multipleOf": [ 3, 5 ] } ]`

This can lead to creating a very large number of objects from a small number of schemas given in an "anyOf" array. For example, 10 schemas would produce 1023 objects. To avoid this, we treat "anyOf" exactly like "allOf". Queries generally check the existence of specific values in an object without caring if the object contains other additional values, so this simplification will not alter the results in most cases.

The "not" keyword provides a schema against which the instance must not validate. The example below is valid for values that are not a multiple of 4:

```
{
  "not":{
    "multipleOf": 4
  }
}
```

When querying schema objects, our service simply checks if the specified key-value pairs are inside the objects. A query about multiples of 4 will return only schema objects containing the keyword "multipleOf" with a value of 4. For that reason we simplify the schema object by omitting the "not" keyword.

The "if", "then", "else" keywords validate the instance against their schemas conditionally. To simplify them, we convert them into their equivalent that is shown below and we treat it as described previously.

if, then, else	Equivalent

`{`
`  "if": {<schemaA>},`
`  "then": {<schemaB>},`
`  "else": {<schemaC>}`
`}`
	`{`
	`  "oneOf":[`
	`  {`
	`    "allOf":[`
	`      {<schemaA>},`
	`      {<schemaB>}`
	`    ]`
	`  },`
	`  {`
	`    "allOf":[`
	`      {"not": {<schemaA>}},`
	`      {<schemaC>}`
	`    ]`
	`  }`
	`  ]`
	`}`

The "dependentSchemas" keyword defines a schema that the instance must validate against if a property is present in the instance. For simplification, we merge the key-value pairs of the given schema into the parent schema, just like we would simplify "allOf" containing a single schema.

The "x-mapsTo" semantic annotation that applies to *Schema* objects references an object with similar semantics. For example, below we show a schema that references another schema with "x-mapsTo":

```
{
  "x-mapsTo": "#/components/schemas/personSchema",
  "type": "object"
}
```

And below is the referenced object:

```
{
  ... ,
  "components":{
    "schemas":{
      "personSchema":{
        "x-refersTo": "http://schema.org/Person",
        "description": "a person"
      }
    }
  }
}
```

To simplify this relationship between objects, we copy the "x-refersTo" and "x-kindOf" fields from the referenced object to the object containing the "x-mapsTo" field. The simplified schema from the example above would be the following:

```
{
  "x-refersTo": "http://schema.org/Person",
  "type": "object"
}
```

The OpenAPI Specification defines the keyword "externalDocs" which holds an *External Documentation* object. We add its fields to the parent object renamed as "extDocsDescription" and "extDocsUrl".

The OpenAPI Specification also defines the *Encoding* and *XML* objects which provide additional information about some properties defined in a schema. We add this information to the item inside the "Property" array that corresponds to the respective property.

We ignore the *Discriminator* object because it provides information useful only to validation tools.

All other fields of a *Schema* object that were not mentioned above, including the *x-collectionOn* proposed semantic annotation, are copied to the metadata object as they are.

5.5 Query translation algorithm

This algorithm accepts an OAQL2 query and produces a MongoDB query. First we will present the general form of a MongoDB query and then we will explain the algorithm to produce it.

We use the "aggregation pipeline" querying functionality of MongoDB. A pipeline consists of various stages. Each stage accepts documents (JSON objects) as input, performs an operation on them and passes the output to the next stage. A stage is represented by a JSON object and, thus, a pipeline is represented by an array of JSON objects.

The pipeline stages that will be used are:

- match: outputs only documents that satisfy the specified conditions, discards the rest
- group: groups input documents by specified fields and outputs one document for each distinct set of values in these fields
- unwind: deconstructs an array field from the input documents and outputs one document for each element. Each output document is the input document with the value of the array field replaced by the element. For example, below is the input and output of an unwind stage that operates on the field "myArray":

Input of unwind	**Output of unwind**
`{ "myArray": [2, 4, 6] }`	`{ "myArray": 2 }, { "myArray": 4 }, { "myArray": 6 }`

- project: hides/renames/copies fields in the document
- sort: sorts documents by specified fields
- addFields: adds new fields to document
- replaceRoot: replace document with specified document

The algorithm to produce a pipeline from an OAQL2 query is the following:

1. translate FROM clause into a series of project, unwind and addFields stages and put them in the pipeline
2. translate WHERE clause into a match stage and append it to the pipeline
3. create another match stage and put it in the beginning of the pipeline
4. translate SELECT and ORDER BY clauses into a set of group, project and sort stages and append them to the pipeline

Below we describe in detail each of these steps and we show the stages produced from the following OAQL2 query:

```
SELECT s.id AS serviceID, r1.path, r2.path
FROM Service s
JOIN Request r1 ON s
JOIN Request r2 ON s
WHERE r1.method="post"
AND r2.method="put"
ORDER BY s.title ASC, s.description DESC
```

This query finds each service with a POST and a PUT request and returns the id of the service, as well as the paths of these two requests. It also sorts the results by the title and description of the service.

5.5.1 Step 1: translating FROM clause

In this step, we translate the *FROM* clause of an OAQL2 query into a series of project, unwind and addFields stages. The purpose of these stages is to convert the metadata objects into objects corresponding to the entries of the table described by the *FROM* clause. Each object in the output of these stages corresponds to one entry and has the following form:

```
{
  "<table1>": {
    "<field1>": <value> ,
    "<field2>": <value> ,
    ...
  },
  "<table2>": {
    ...
  },
  ...
}
```

First, we create a tree with one node for each instance of table specified in *FROM* clause and one edge for each join between two tables. Root of the tree is the table with the least nesting level in the metadata object and the structure of the tree follows the hierarchy of a metadata object. In the example query, root of the tree is the *Service* table and has two children, one for each instance of *Request*.

Note that a node in the tree cannot have more than one parent because the query would have no result. For example consider the following query which joins the same instance of *Schema* table with both *Request* and *Response*:

```
SELECT *
FROM Request r
JOIN Schema s ON r
JOIN Response res ON s
```

There is no schema that belongs to both a request and response. Even if a request and a response reference the same schema in an OpenAPI description, we will store separate copies of the schema in the metadata object. Our service will return an error if given a query with a *FROM* clause such as this.

Once we create the tree, we add to the pipeline the stages shown below:

```
{
  "$project": {
    "<tableAlias>":{
      "$concatArrays":[
        "$<path1>",
        ...
      ]
    }
  }
},
{
  "$unwind":{
    "path": "$<tableAlias>"
  }
}
```

The function of these stages is to concatenate all arrays corresponding to the root table in each metadata object, store the result under the alias of the root table and unwind it.

Note that we need to repeat the unwind stage a number of times equal to the maximum nesting level of the arrays we concatenated.

In the example query, the root table is *Service* and the pipeline stages are:

```
{
  "$project": {
    "s":{
      "$concatArrays":[
        "$Service"
      ]
    }
  }
},
{
  "$unwind": "$s"
}
```

We used one unwind stage because the "Service" array is at the top level of the document and therefore has a nesting level of 1. If the root table was *Request*, we would add a second unwind stage identical to the first because the "Request" array is inside the "Service" array and therefore has a nesting level of 2.

There is a special case when the root table is *Property* or *Item* because these tables can be joined with themselves and therefore the arrays corresponding to them might be located at any depth inside the arrays corresponding to *Schema* tables. To solve this, we need to execute a Javascript function that concatenates these arrays from each document. First, we add the project and unwind stages that concatenate the arrays corresponding to *Schema* table as described above. Then we add the following stages:

```
{
  "$project": {
    "<tableAlias>": {
      "$function": {
        "body": "<JS function>",
        "args": [
          "$<tableAlias>"
        ],
        "lang": "js"
      }
    }
  }
},
{
  "$unwind": "$<tableAlias>"
}
```

The JS function is a function that finds all arrays named "Property" or "Item" inside the document depending on which of the two is the root table, concatenates them and returns the result.

The output of these stages is one document for each entry of the root table. After this, we follow a recursive process for each node of the tree, beginning with the root. For each child of the current node, we add the following pipeline stages:

```
{
  "$addFields":{
    "<child1Alias>": "$<currentTableAlias>.<child1>",
    ...
  }
},
{
  "$unwind":{
    "path": "$<child1Alias>",
    "preserveNullAndEmptyArrays": true
  }
},
...
```

Then, we repeat this process for each child node. The first of the above stages adds for each child node a field with a copy of the array corresponding to the child table. The next stages unwind all newly added fields.

The stages for this process in the example query are:

```
{
  "$addFields": {
    "r2": "$s.Request",
    "r1": "$s.Request"
  }
},
{
  "$unwind": {
    "path": "$r1",
    "preserveNullAndEmptyArrays": true
  }
},
{
  "$unwind": {
    "path": "$r2",
    "preserveNullAndEmptyArrays": true
  }
}
```

The output of all the pipeline stages described until now will be one document for each entry of the table formed in *FROM* clause.

5.5.2 Step 2: translating WHERE clause

In this step, we translate the *WHERE* clause of an OAQL2 query into a match stage. This stage discards documents (table entries) that do not satisfy the conditions specified in *WHERE* clause. If there is no *WHERE* clause in a query, this stage is omitted.

The translation rules to produce the match stage are given at the table below:

OAQL2	MongoDB
`WHERE <condition>`	`{"$match": <condition> }`
`<condition1> AND <condition2>`	`{"$and": [<condition1>, <condition2>] }`
`<condition1> OR <condition2>`	`{"$or": [<condition1>, <condition2>] }`
`<field> <operator> <value>`	`{"<field>": {"<operator>": <value>} }`
`<field> IS NULL`	`{"<field>": {"$exists": false} }`
`<field> IS NOT NULL`	`{"<field>": {"$exists": true} }`
`<field> LIKE <value>`	`{ "<field>": { "$regex": <value>, "$options": "s" } }`
`<field1> <operator> <field2>`	`{ "$expr": { "<operator>": [ "<field1>", "<field2>" ] } }`

`<field1> IN <field2>`	`{ "$expr":{ "$in":[ "<field1>", { "$cond":[ { "$isArray":"<field2>" }, "<field2>", [] ] } ] } }`

We translate the operators as shown below:

OAQL2 operator	**MongoDB operator**
=	$eq
<>	$ne
>	$gt
>=	$gte
<	$lt
<=	$lte

When applying the rule for the translation of `<field> LIKE <value>`, we make some changes to the value in order to fit MongoDB's regular expression rules. We add the characters "^" and "$" in the beginning and the end of the value respectively. We also replace "%" with ".*" and "_" with ".".

When we translate a condition that compares the *Response.statusCode* field with a numeric value, we also combine with OR the conditions comparing the field with the code ranges that contain codes satisfying the initial condition. For example, the condition:

```
Response.statusCode < 201
```

will be translated into the equivalent of the following:

```
Response.statusCode < 201
OR Response.statusCode = "2XX"
OR Response.statusCode = "1XX"
```

When we translate a condition that compares the *contentType* field with a value, we also combine with OR the conditions comparing the field with the media type ranges the value belongs in. For example, the condition:

```
Request.contentType ="application/json"
```

will be translated into the equivalent of the following:

```
Request.contentType = "application/json"
OR Request.contentType = "application/*"
OR Request.contentType = "*/*"
```

When we translate a condition that checks if the *x-refersTo* field of *Schema*, *Property* or *Item* table is equal to a value, we also want to return results about a subclass of that value. First, we execute the following SPARQL query on the reasoner:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?c
WHERE { ?c rdfs:subClassOf|rdfs:subPropertyOf <value> }
```

The above query will return all subclasses (or subproperties) of the value, including itself since rdfs:subClassOf and rdfs:subPropertyOf are reflexive. Then, the condition is translated to the following:

```
{
  "$or": [
    {
      "<tableAlias>.x-refersTo": {
        "$in": [
          "<subclass1>",
          ...
        ]
      }
    },
    {
      "<tableAlias>.x-kindOf": {
        "$in": [
          "<subClass1>",
          ...
        ]
      }
    }
  ]
}
```

The above condition checks if the *x-refersTo* or *x-kindOf* field in the document is equal to one of the subclasses returned by the reasoner.

When we translate a condition that checks if the *x-kindOf* field of *Schema*, *Property* or *Item* table is equal to a value, we follow a process similar to when translating a query on *x-refersTo*. We execute the same SPARQL query and we produce almost the same condition. The only difference is that we exclude the given value from the array of subclasses that we query *x-refersTo* for. This is because we want only table entries that refer to a specialization of the concept and not the concept itself.

When translating a condition that checks if the *x-operationType* field of *Request*, *Callback* or *Webhook* table is equal to a value, we execute the SPARQL query mentioned previously and the translated condition has the following form:

```
{
  "<tableAlias>.x-operationType": {
    "$in": [
      "<subClass1>",
      ...
    ]
  }
}
```

The above condition checks if the *x-operationType* field is equal to one of the subclasses returned by the reasoner.

The match stage for the example query is:

```
{
  "$match": {
    "$and": [
      {
        "r1.method": {
          "$eq": "post"
        }
      },
      {
        "r2.method": {
          "$eq": "put"
        }
      }
    ]
  }
}
```

5.5.3 Step 3: creating another match stage

In this step we create another match stage, different from the one created in the previous step, which will be placed in the beginning of the pipeline.

This stage is produced by following the same rules defined in the previous step with one exception. This stage cannot check conditions with table fields as both operands. To avoid losing documents, it assumes that these conditions are always true for all documents.

We also need to specify the full path in the metadata object of any field we want to query. For example, we translate the *Request.method* field to "Service.Request.method". This becomes more challenging when the array corresponding to the root table is located in more than one places in a metadata object. In this case, we create a separate condition for each possible path and we combine them with the OR operator.

The match stage for the example query is:

```
{
  "$match": {
    "$and": [
      {
        "Service.Request.method": {
          "$eq": "post"
        }
      },
      {
        "Service.Request.method": {
          "$eq": "put"
        }
      }
    ]
  }
}
```

The purpose of this match stage is to reduce the number of documents that pass through the following stages of the pipeline. It discards metadata objects that are certain to not contain information satisfying the conditions in *WHERE* clause. If the query has no *WHERE* clause, this stage is omitted.

By being the first stage in the pipeline, it searches directly on the collection of metadata objects and, therefore, can use any existing indexes to speed up the process.

5.5.4 Step 4: translating SELECT and ORDER BY clauses

In this step, we translate the *SELECT* and *ORDER BY* clauses of an OAQL2 query into a set of project, group and sort stages.

The main purpose of these stages is to bring each document to the following form, showing only the table fields specified in *SELECT* clause:

```
{
  "<field1>": <value>,
  ...
}
```

This is achieved by using a project stage like the following:

```
{
  "$project":{
    "_id": 0,
    "<field1Alias>": "$<field1>",
    ...
  }
}
```

If the keyword "DISTINCT" is given in the query, we use a group and a replaceRoot stage instead of the project stage described. These two stages group the documents by the values in the specified fields and return one document for each distinct set of values. They have the following form:

```
{
  "$group":{
    "_id":{
      "<field1Alias>": "$<field1>",
      ...
    }
  }
},
{
  "$replaceRoot":{
    "newRoot": "$_id"
  }
}
```

If the query has an *ORDER BY* clause, we also add a sort stage in the pipeline. The sort stage has the following form:

```
{
  "$sort": {
    "<field1Name>": 1/-1,
    ...
  }
}
```

Value of 1 means ascending order and -1 means descending. If the "DISTINCT" keyword is not given in the query, the sort stage is placed before the project stage described previously so that it is able to sort the documents by fields that might not be included in *SELECT* clause. Otherwise, the sort stage is placed after the replaceRoot stage.

Note that MongoDB does not allow dots in a key. For this reason we replace any dots with the character '@' in the table field aliases and the server fixes it before forwarding the result from the database to the client.

Apart from the stages described for this part of the pipeline, we need to add another project stage in the beginning of this part if the query uses the wildcard "*". The wildcard means that all table fields should be returned, including any extension properties. However, we do not know the names of the extension properties and thus we cannot write them explicitly to be included like the rest of the table fields.

To solve this, we note that each object corresponding to a table entry contains only the fields corresponding to the table fields as well as some arrays corresponding to entries from other tables. The keys of these arrays are known and we can specify them to be excluded. This is done with the extra project stage that has the following form:

```
{
  "$project":{
    "<table1Alias>":{
      "<array1Key>":0,
      ...
    },
    ...
  }
}
```

Then, the following project or group stage keeps these objects with no change. They need only to be flattened so that they match the format of the other fields, which is done by the server before forwarding the result to the client.

The pipeline stages of this part for the example query are:

```
{
  "$sort": {
    "s.title": 1,
    "s.description": -1
  }
},
{
  "$project": {
    "_id": 0,
    "r2@path": "$r2.path",
    "r1@path": "$r1.path",
    "serviceID": "$s.id"
  }
}
```

5.6 Equivalence of OAQL2 and translated queries

In this section we will show that each MongoDB pipeline produced by the process described previously can be translated into an OAQL2 query that will always return the same results with the original OAQL2 query, despite some small differences that they may have in the way they are written. We will show this by describing the process to translate a pipeline into an OAQL2 query.

To produce the *FROM* clause of the OAQL2 query from a pipeline, we need first to construct the tree described previously. We find the root table from the first project stage (or stages if root table is *Property* or *Item*) in the pipeline. Then, we find its children from the addFields stage that follows, each child's children from the next addFields stages and so on. Once we create the tree, we can easily write a *FROM* clause that produces this tree. The only difference it may have from the original query's *FROM* clause is the order in which the tables are written but their functionality will be the same.

To produce the *WHERE* clause of the OAQL2 query from a pipeline, we simply apply the reverse of the translation rules described for the production of the match stages. For conditions querying *x-refersTo*, *x-kindOf* or *x-operationType* we will need to execute SPARQL queries on the reasoner to find which value is the superclass of all the others. The resulting *WHERE* clause will have the same functionality with the one from the original query and they will be almost identical. The only difference is the

```
<field> BETWEEN <value1> AND <value2>
```

condition which might be translated as

```
<field> >= <value1> AND <field> <= <value2>
```

or vice versa.

We produce the *SELECT* clause from the project or group stage at the end of the pipeline. We determine if the "DISTINCT" keyword is used by whether there is a group stage or not and we find the table fields with their aliases inside the project or group stage. The produced *SELECT* clause will have identical functionality to the original one. Their only difference might be that a "SELECT *" clause may be translated as

```
SELECT <table1>.*, <table2>.*, ...
```

or vice versa.

The *ORDER BY* clause can be produced from the sort stage towards the end of the pipeline. In this stage are specified the table fields to sort by and the sorting order. The resulting clause will be identical to the original one.

5.7 Indexing

As mentioned previously, the indexes are used by the match stage in the beginning of the pipeline. The role of that stage is to reduce the number of metadata objects that will pass through the next stages of the pipeline.

Using an index will only save time if it can be used to discard a large number of documents without searching them. For example, most OpenAPI descriptions contain at least one GET request, so the existence of an index will not help when querying about GET requests. However, it will help when searching for a PATCH request since it is used in less descriptions.

MongoDB sets a limit of up to 64 indexes in a collection. Considering that the information of some table fields is contained in many different locations in a metadata object, we build indexes for the following fields only on the specified paths of the metadata object:

Table field	**Indexed paths**
Request.method	Service.Request.method
Request.path	Service.Request.path
Request.contentType	Service.Request.contentType
Request.x-operationType	Service.Request.x-operationType
Parameter.name	Service.Request.Parameter.name
Response.statusCode	Service.Request.Response.statusCode
Response.contentType	Service.Request.Response.contentType
Header.name	Service.Request.Response.Header.name
Security.type	Service.Request.Security.type
Schema.x-refersTo	Service.Request.Schema.x-refersTo Service.Request.Parameter.Schema.x-refersTo Service.Request.Response.Schema.x-refersTo Service.Request.Response.Header.Schema.x-refersTo
Schema.x-kindOf	Service.Request.Schema.x-kindOf Service.Request.Parameter.Schema.x-kindOf Service.Request.Response.Schema.x-kindOf Service.Request.Response.Header.Schema.x-kindOf
Schema.type	Service.Request.Schema.type Service.Request.Parameter.Schema.type Service.Request.Response.Schema.type Service.Request.Response.Header.Schema.type
Property.name	Service.Request.Schema.Property.name Service.Request.Parameter.Schema.Property.name Service.Request.Response.Schema.Property.name Service.Request.Response.Header.Schema.Property.name

Property.x-refersTo	Service.Request.Schema.Property.x-refersTo Service.Request.Parameter.Schema.Property.x-refersTo Service.Request.Response.Schema.Property.x-refersTo Service.Request.Response.Header.Schema.Property.x-refersTo
Property.x-kindOf	Service.Request.Schema.Property.x-kindOf Service.Request.Parameter.Schema.Property.x-kindOf Service.Request.Response.Schema.Property.x-kindOf Service.Request.Response.Header.Schema.Property.x-kindOf
Property.type	Service.Request.Schema.Property.type Service.Request.Parameter.Schema.Property.type Service.Request.Response.Schema.Property.type Service.Request.Response.Header.Schema.Property.type
Item.x-refersTo	Service.Request.Schema.Item.x-refersTo Service.Request.Parameter.Schema.Item.x-refersTo Service.Request.Response.Schema.Item.x-refersTo Service.Request.Response.Header.Schema.Item.x-refersTo
Item.x-kindOf	Service.Request.Schema.Item.x-kindOf Service.Request.Parameter.Schema.Item.x-kindOf Service.Request.Response.Schema.Item.x-kindOf Service.Request.Response.Header.Schema.Item.x-kindOf
Item.type	Service.Request.Schema.Item.type Service.Request.Parameter.Schema.Item.type Service.Request.Response.Schema.Item.type Service.Request.Response.Header.Schema.Item.type

Chapter 6

Results and comparisons

We describe the elements of a query that influence the time needed for its execution in our system, providing also example queries to show these differences. In addition, we compare the execution time of some queries on different systems.

6.1 Performance analysis

In this section, we describe the factors that affect the execution time of a query in our system and we execute some queries to show the difference between their execution times.

The queries are executed on 1000 OpenAPI descriptions taken from Swaggerhub, with a total size of 25.9MB. MongoDB needs 7.6MB of memory to store them, using compression. It also uses another 13.4MB to store the metadata objects and an additional 2.9MB to store the indexes.

The execution time is counted from the moment our system receives an HTTP request with an OAQL2 query until the moment it finishes sending to the client the results of the query. This means that it includes both the translation time from OAQL2 to MongoDB query and the time for searching in MongoDB. The translation time is insignificant (less than 2 ms) except for cases in which we use the reasoner and which will be explained later.

In Appendix C we show the translation to MongoDB queries of some of the OAQL2 queries below in order to demonstrate the difference in simplicity of syntax.

The following factors affect the speed of a query:

- Existence of index on queried table fields
- Number of documents passing through the pipeline
- Sorting of results
- Querying the fields *x-refersTo*, *x-kindOf* or *x-operationType*

There are also two special cases:

- *FROM* clause contains only *Property* and *Item* tables
- Conditions in *WHERE* clause with fields as both operands

6.1.1 Existence of index

Queries on indexed fields are faster than queries on fields without index since the index can help reduce quickly the number of documents that will be examined.

The first of the queries below finds services containing a property named "firstName" or "lastName" in the request payload and the second query finds services that contain a response with status code 404. We execute these queries before and after we build an index on fields *Property.name* and *Response.statusCode*:

Query	Without index	With index
SELECT s.id FROM Service s JOIN Request r ON s JOIN Schema sc ON r JOIN Property p ON sc WHERE p.name = "firstName" OR p.name = "lastName"	95ms	74ms

SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Response res ON r
WHERE res.statusCode = 404
	208ms	189ms

We can see that the queries are faster when an index exists on the queried fields.

6.1.2 Number of documents passing through the pipeline

The more documents that pass through the pipeline, the slower a query will be since MongoDB executes operations on each document. The number of documents depends on the number of metadata objects that contain information satisfying the conditions in *WHERE* clause of the query. These metadata objects will be split to create the documents corresponding to table entries while the rest will be discarded at the first stage of the pipeline.

The first of the queries below finds services that contain a PATCH method and the second finds services containing a GET method. The PATCH method is used in less OpenAPI descriptions than the GET method, so we expect the first query to be faster.

Query	Execution time	Number of entries in result

SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.method = "patch"
	90ms	511

SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.method = "get"
	179ms	6936

We also execute the queries shown in tables B.1 and B.3 in Apppendix B. The queries in these two tables are similar, with their only difference being the conditions in *WHERE* clause. The queries in table B.1 request less common values while the queries in table B.3 request values contained in most OpenAPI descriptions. Therefore, the queries in table B.1 will be faster.

Below we present the average execution time needed per query and the average number of table entries in the result per query for each of the tables B.1 and B.3:

	Average execution time	**Average number of entries in result**
Table B.1	96ms	566
Table B.3	273ms	14803

As expected, the average time for table B.1 is less than for table B.3.

Another factor affecting the number of documents passing through the pipeline is the number of tables joined in *FROM* clause. This number determines the size of the final table that will be created and whose number of entries equals the number of documents that will pass through the pipeline.

Both requests below find services containing a response with status code 201. However, the second query includes two instances of *Response* table in its *FROM* clause. This will lead to forming a bigger table, which means more documents in the pipeline and, thus, a slower query.

Query	**Execution time**	**Number of entries in result**

SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Response res ON r
WHERE res.statusCode = 201
	128ms	1225

SELECT s.id FROM Service s JOIN Request r ON s JOIN Response res ON r JOIN Response res2 ON r JOIN Response res3 ON r WHERE res.statusCode = 201	186ms	15372

We also execute the queries shown in table B.5 in Apppendix B. The queries in this table have the same functionality with those in table B.1. However, the queries in table B.5 include more joins in *FROM* clause which makes them slower.

Below we present the average execution time needed per query and the average number of entries in the result per query for each of the tables B.1 and B.5:

	Average execution time	**Average number of entries in result**
Table B.1	96ms	566
Table B.5	127ms	7607

We can see that the average execution time for table B.1 is less than for table B.5.

6.1.3 Sorting

When an OAQL2 query uses the *ORDER BY* clause, it requests sorting of the entries in the result. This sorting causes an additional delay during the execution of the query. The execution of the queries below shows this difference in performance:

Query	Execution time	Number of results

SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.method = "patch"
	90ms	511

SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.method = "patch"
ORDER BY r.path DESC
	4.3sec	511

Both of the above queries return the titles of services containing a PATCH request. We can see that the second query is much slower because it also sorts the request paths before returning them.

We execute the queries shown in table B.7 and compare the average execution time per query with that of the queries in table B.1. The queries in table B.7 are almost identical to those of table B.1. The only difference is the addition of an *ORDER BY* clause which makes them slower.

	Average execution time	Average number of entries in result
Table B.1	96ms	566
Table B.7	3.5sec	566

6.1.4 Querying x-refersTo, x-kindOf, x-operationType

Querying *x-refersTo*, *x-kindOf* or *x-operationType* will cause the additional delay of using the reasoner to find the subclasses or subproperties of the given value.

We demonstrate this with the queries below. The first query finds services containing a PATCH request. The second query specifies an additional condition on *x-refersTo* field.

Query	Execution time	Number of results

SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Schema sc ON r
WHERE r.method = "patch"
	71ms	538

SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Schema sc ON r
WHERE r.method = "patch"
OR sc.x-refersTo = "https://schema.org/Store"
	106ms	538

The second query is slower than the first because it searches for *Schema* objects annotated to the "Store" concept. Therefore, the system will request the subclasses of "Store" from the reasoner, which causes the additional delay. Then, the query will return all table entries in which *x-refersTo* or *x-kindOf* is equal to one of those subclasses.

6.1.5 Special cases

When the *FROM* clause contains only *Property* and *Item* tables, the query is slower because we need to execute a Javascript function for each document as described previously. However, such queries will be rare because the user will usually want information from other tables as well. This delay is shown by the queries below:

Query	Execution time	Number of results

SELECT p.type
FROM Property p
JOIN Schema s ON p
WHERE p.name = "products"
	512ms	16

SELECT p.type
FROM Property p
WHERE p.name = "products"
	12.5sec	38

The first query of the above finds all schemas containing a property named "products" and returns the type of that property. The second query returns the type of all properties named "products" which can be a property in a schema, a property of another property and so on. This makes the second query slower.

As mentioned, conditions with fields as both operands are assumed to be satisfied by all documents at the match stage in the first part of the pipeline. This leads to a larger table in 'FROM' clause and therefore to slower execution as shown by the queries below:

Query	Execution time	Number of results

SELECT DISTINCT s.id
FROM Tag t
JOIN Service s ON t
JOIN Request r ON s
WHERE t.name IN r.tags
AND r.method = "put"
	733ms	234

SELECT DISTINCT s.id
FROM Tag t
JOIN Service s ON t
JOIN Request r ON s
WHERE t.name IN r.tags
OR r.method = "put"
	954ms	721

The first query finds services containing a PUT request with a tag, while the second query finds services that either define tags for their requests or define a PUT request. In the first query, the first stage of the pipeline can discard all descriptions not containing a PUT method. However, in the second query, assuming that the first condition is true makes the whole condition true so the first stage of the pipeline cannot discard any descriptions and the query is slower.

6.2 Performance comparisons

In this chapter, we compare the performance of the following three systems:

- the system we implemented
- a system which stores OpenAPI descriptions in a MongoDB and searches directly on them instead of creating and querying metadata objects
- the previous system [6]

We will show that our system has better performance than the other two.

First, we compare our system with the previous one. We loaded into each system 100 OpenAPI descriptions taken from Swaggerhub with a total size of 2.1MB. We did not use all of the 1000 OpenAPI descriptions that we used in the previous section because the queries in the previous system timed out after 75 seconds and were cancelled before returning a result. Below we show some

OAQL2 queries with the equivalent queries in OpenAPI QL (the language of the previous system) and the execution time in each system:

	Query	Execution time

OAQL2	SELECT s.id AS service_id
	FROM Service s
	JOIN Request req ON s
	WHERE req.method = "get"
		51ms

OpenAPI QL	SELECT s.id AS service_id
	FROM Service s,
	Request req
	WHERE s.request = req
	AND req.method = "get"
		464ms

OAQL2	SELECT s.id AS service_id
	FROM Service s
	JOIN Request req ON s
	WHERE req.contentType = "application/json"
		54ms

OpenAPI QL	SELECT s.id AS service_id
	FROM Service s,
	Request req
	WHERE s.request = req
	AND req.media_type = "application/json"
		6.7sec

OAQL2	SELECT s.id AS service_id
	FROM Service s
	JOIN Request req ON s
	JOIN Parameter p ON req
	WHERE p.name = "limit"
	AND p.in = "query"
		36ms

OpenAPI QL	SELECT s.id AS service_id FROM Service s, Request req, QueryParam qp WHERE s.request = req AND req.queryparam = qp AND qp.name = "limit"	9.6sec

OAQL2	SELECT s.id AS service_id
	FROM Service s
	JOIN Request req ON s
	JOIN Security sec ON req
	WHERE sec.apiKeyName = "api_key"
	AND sec.apiKeyIn = "header"
		30ms

OpenAPI QL	SELECT s.id AS service_id FROM Service s, Request req, ApiKeyAuth ap WHERE s.request = req AND req.apikeyauth = ap AND ap.name = "api_key" AND ap.in = "header"	13.9sec
OAQL2	SELECT s.id AS service_id FROM Service s JOIN Request req ON s JOIN Response resp ON req JOIN Schema sc ON req JOIN Property p ON sc WHERE req.contentType = "application/json" AND p.name = "id" AND resp.statusCode BETWEEN 199 AND 209	37ms

OpenAPI QL	SELECT s.id AS service_id FROM Service s, Request req, Response resp, Schema sc WHERE s.request = req AND req.media_type = "application/json" AND req.schema = sc AND sc.property = "id" AND req.response = resp AND resp.status_code BETWEEN 199 AND 209	9.7sec
OAQL2	SELECT s.id AS service_id FROM Service s JOIN Request req ON s JOIN Response resp ON req JOIN Schema sc1 ON req JOIN Property p1 ON sc1 JOIN Schema sc2 ON resp JOIN Property p2 ON sc2 WHERE (req.method = "post" AND req.contentType = "application/json" AND p1.name = "payload") OR p2.name = "payload"	31ms
OpenAPI QL	SELECT s.id AS service_id FROM Service s, Request req, Response resp, Schema sc WHERE s.request = req AND req.method = "post" AND req.media_type = "application/json" AND req.schema = sc AND sc.property = "payload" OR req.response = resp AND resp.schema = sc AND sc.property = "payload"	3.4sec

We can see that our system has much better performance than the previous one. The main reason for this is that the previous system uses a Couchbase as its database. Queries on OpenAPI descriptions are very complex and Couchbase does not handle them as efficiently as MongoDB. Another reason is the lack of indexes in the previous system.

In the following table we compare our system with a system that searches directly on the OpenAPI descriptions in a MongoDB without creating metadata objects. We use the 1000 OpenAPI descriptions with a total size of 25.9MB that we used for the queries in the previous section. We execute the same queries as above:

Query	**Our system**	**System without metadata**

SELECT s.id AS service_id
FROM Service s
JOIN Request req ON s
WHERE req.method = "get"
	179ms	280ms

SELECT s.id AS service_id
FROM Service s
JOIN Request req ON s
WHERE req.contentType = "application/json"
	168ms	375ms

SELECT s.id AS service_id
FROM Service s
JOIN Request req ON s
JOIN Parameter p ON req
WHERE p.name = "limit"
AND p.in = "query"
	67ms	291ms

SELECT s.id AS service_id
FROM Service s
JOIN Request req ON s
JOIN Security sec ON req
WHERE sec.apiKeyName = "api_key"
AND sec.apiKeyIn = "header"
	49ms	357ms

SELECT s.id AS service_id FROM Service s JOIN Request req ON s JOIN Response resp ON req JOIN Schema sc ON req JOIN Property p ON sc WHERE req.contentType = "application/json" AND resp.statusCode BETWEEN 199 AND 209 AND p.name = "id"	135ms	498ms
SELECT s.id AS service_id FROM Service s JOIN Request req ON s JOIN Response resp ON req JOIN Schema sc1 ON req JOIN Property p1 ON sc1 JOIN Schema sc2 ON resp JOIN Property p2 ON sc2 WHERE (req.method = "post" AND req.contentType = "application/json" AND p1.name = "payload") OR p2.name = "payload"	405ms	845ms

We can see that our system is faster. This happens mainly because our system uses indexes to speed up the search.

Chapter 7

Conclusions and future work

In this thesis, we defined OAQL2 (OpenAPI Query Language 2) which is a language for finding web services by querying their OpenAPI descriptions. This language is very similar to SQL and allows the user to query or view information from almost all objects and fields in an OpenAPI description. We also implemented a web service that is able to execute OAQL2 queries and consists of a Java server and a MongoDB database.

7.1 Conclusions

As mentioned, the syntax of OAQL2 is by design similar to the syntax of SQL. A user who knows SQL and is familiar with the concepts of REST architecture should be able to understand and use OAQL2 without any knowledge of the OpenAPI Specification. However, a user who is not familiar with the OpenAPI Specification may not understand the purpose and meaning of many fields and tables.

OAQL2 is designed as an improved version of OpenAPI QL. Specifically:

- it supports more operators, such as LIKE, IS NULL, IN
- it allows the user to query most fields of an OpenAPI document as well as the proposed semantic annotations
- it allows the user to retrieve any defined table fields instead of only *Service.id*

Besides fully supporting OAQL2, the implementation described in this thesis also has the following advantages compared to the system implemented in previous work:

- it is much faster
- it supports searching in composite schema objects
- it uses reasoning when querying *x-refersTo*, *x-kindOf* and *x-operationType* fields

Our system needs more than double the memory of the previous system since, besides the OpenAPI descriptions, it also stores a metadata object for each description and keeps indexes. However, we are more interested in the performance of the system than the space requirements.

7.2 Future work

An idea for future work is that, along with the results returned by our system, queries about a schema object should also return schema objects that are annotated to the same semantic concepts with the results. This would require to execute the query normally and get the results, use reasoning to find all subclasses of all concepts the schema objects in the results are annotated to and, lastly, execute another query to find schema objects that are annotated to the concepts returned by the reasoner.

Another idea is to try to create a semantic model for schema objects that are not annotated to a semantic concept. These objects will be automatically annotated to a concept by the system based on the values of their fields and similarities with other objects.

References

[1] *Representational state transfer.* https://en.wikipedia.org/wiki/Representational_state_transfer.

[2] *OpenAPI Specification v3.1.0.* https://spec.openapis.org/oas/v3.1.0.

[3] *Semantic Web.* https://en.wikipedia.org/wiki/Semantic_Web.

[4] *Resource Description Framework.* https://en.wikipedia.org/wiki/Resource_Description_Framework.

[5] N. Mainas and E. Petrakis. "Soas 3.0: Semantically enriched openapi 3.0 descriptions and ontology for rest services". In: *IEEE Intern. Conf. on Semantic Computing (ICSC 2020).* San Diego, California, 2020, pp. 207–210. URL: http://www.intelligence.tuc.gr/~petrakis/publications/ICSC2020.pdf.

[6] I.M. Stergiou. "Searching in REST service catalogues with OpenAPI descriptions". Diploma Thesis. School of Electrical and Computer Engineering, Technical University of Crete, Oct. 2021.

[7] *JSONPath - XPath for JSON.* https://goessner.net/articles/JsonPath/.

[8] *JSONQuery: Data Querying Beyond JSONPath.* https://www.sitepen.com/blog/jsonquery-data-querying-beyond-jsonpath.

[9] *Jaql.* https://en.wikipedia.org/wiki/Jaql.

[10] *JSONiq.* https://www.jsoniq.org/.

[11] Enqing Tang and Yushun Fan. "Performance Comparison between Five NoSQL Databases". In: *2016 7th International Conference on Cloud Computing and Big Data (CCBD).* 2016, pp. 105–109. DOI: 10.1109/CCBD.2016.030.

[12] Linggis Galih Wiseso, Mahmud Imrona, and Andry Alamsyah. "Performance Analysis of Neo4j, MongoDB, and PostgreSQL on 2019 National Election Big Data Management Database". In: *2020 6th International Conference on Science in Information Technology (ICSITech).* 2020, pp. 91–96. DOI: 10.1109/ICSITech49800.2020.9392041.

Appendix A

Format of metadata object

Below we show the format of a metadata object with all of its fields. We abbreviate the contents of fields that are repeated in different locations to three dots.

```
{
  "Service": [
    {
      "contactEmail": <value>,
      "contactName": <value>,
      "contactUrl": <value>,
      "description": <value>,
      "extDocsDescription": <value>,
      "extDocsUrl": <value>,
      "id": <value>,
      "jsonSchemaDialect": <value>,
      "licenseName": <value>,
      "licenseUrl": <value>,
      "openapiVersion": <value>,
      "termsOfService": <value>,
      "title": <value>,
      "version": <value>,
      "Request": [
        {
          "bodyDescription": <value>,
          "bodyRequired": <value>,
          "contentType": <value>,
          "deprecated": <value>,
          "description": <value>,
          "extDocsDescription": <value>,
          "extDocsUrl": <value>,
          "method": <value>,
          "operationId": <value>,
          "path": <value>,
          "summary": <value>,
          "tags": <value>,
          "x-operationType": <value>,
          "Callback": [
            {
              "bodyDescription": <value>,
              "bodyRequired": <value>,
              "contentType": <value>,
              "deprecated": <value>,
              "description": <value>,
              "extDocsDescription": <value>,
              "extDocsUrl": <value>,
              "method": <value>,
              "name": <value>,
              "operationId": <value>,
              "path": <value>,
              "summary": <value>,
              "tags": <value>,
              "x-operationType": <value>,
              "Example": [
                {
                  "description": <value>,
                  "externalValue": <value>,
                  "name": <value>,
                  "summary": <value>,
                  "value": <value>
                },
                ...
              ],
              "Parameter": [
                {
                  "allowEmptyValue": <value>,
                  "allowReserved": <value>,
                  "contentType": <value>,
                  "deprecated": <value>,
                  "description": <value>,
                  "explode": <value>,
                  "in": <value>,
                  "name": <value>,
                  "required": <value>,
                  "style": <value>,
                  "Example": [ ... ],
                  "Schema": [
                    {
                      "const": <value>,
                      "contentEncoding": <value>,
                      "contentMediaType": <value>,
                      "default": <value>,
                      "deprecated": <value>,
                      "description": <value>,
                      "enum": <value>,
                      "examples": <value>,
                      "exclusiveMaximum": <value>,
                      "exclusiveMinimum": <value>,
                      "extDocsDescription": <value>,
                      "extDocsUrl": <value>,
                      "format": <value>,
                      "maxItems": <value>,
                      "maxLength": <value>,
                      "maxProperties": <value>,
                      "maximum": <value>,
                      "minItems": <value>,
                      "minLength": <value>,
                      "minProperties": <value>,
                      "minimum": <value>,
                      "multipleOf": <value>,
                      "pattern": <value>,
                      "readOnly": <value>,
                      "required": <value>,
                      "title": <value>,
                      "type": <value>,
                      "uniqueItems": <value>,
                      "writeOnly": <value>,
                      "x-collectionOn": <value>,
                      "x-kindOf": <value>,
                      "x-refersTo": <value>,
                      "Item": [
                        {
                          "const": <value>,
                          "contentEncoding": <value>,
                          "contentMediaType": <value>,
                          "default": <value>,
                          "deprecated": <value>,
                          "description": <value>,
                          "enum": <value>,
                          "examples": <value>,
                          "exclusiveMaximum": <value>,
                          "exclusiveMinimum": <value>,
                          "format": <value>,
                          "maxItems": <value>,
                          "maxLength": <value>,
                          "maxProperties": <value>,
                          "maximum": <value>,
                          "minItems": <value>,
                          "minLength": <value>,
                          "minProperties": <value>,
                          "minimum": <value>,
                          "multipleOf": <value>,
                          "pattern": <value>,
                          "readOnly": <value>,
                          "required": <value>,
                          "title": <value>,
                          "type": <value>,
                          "uniqueItems": <value>,
                          "writeOnly": <value>,
                          "x-collectionOn": <value>,
                          "x-kindOf": <value>,
                          "x-refersTo": <value>,
                          "Item": [ ... ],
                          "Property": [
                            {
                              "allowReserved": <value>,
                              "const": <value>,
                              "contentEncoding": <value>,
                              "contentMediaType": <value>,
                              "contentType": <value>,
                              "default": <value>,
                              "deprecated": <value>,
                              "description": <value>,
                              "enum": <value>,
                              "examples": <value>,
                              "exclusiveMaximum": <value>,
                              "exclusiveMinimum": <value>,
                              "explode": <value>,
                              "format": <value>,
                              "maxItems": <value>,
                              "maxLength": <value>,
                              "maxProperties": <value>,
                              "maximum": <value>,
                              "minItems": <value>,
                              "minLength": <value>,
                              "minProperties": <value>,
                              "minimum": <value>,
                              "multipleOf": <value>,
                              "name": <value>,
                              "pattern": <value>,
                              "readOnly": <value>,
                              "required": <value>,
                              "style": <value>,
                              "title": <value>,
                              "type": <value>,
                              "uniqueItems": <value>,
                              "writeOnly": <value>,
                              "x-collectionOn": <value>,
                              "x-kindOf": <value>,
                              "x-refersTo": <value>,
                              "xmlAttribute": <value>,
                              "xmlName": <value>,
                              "xmlNamespace": <value>,
                "xmlPrefix": <value>,
                "xmlWrapped": <value>,
                "Header": [
                  {
                    "allowEmptyValue": <value>,
                    "contentType": <value>,
                    "deprecated": <value>,
                    "description": <value>,
                    "explode": <value>,
                    "name": <value>,
                    "required": <value>,
                    "style": <value>,
                    "Example": [ ... ],
                    "Schema": [ ... ]
                  },
                  ...
                ],
                "Item": [ ... ],
                "Property": [ ... ]
              },
              ...
            ]
          },
          ...
        ],
        "Property": [ ... ]
      },
      ...
    ]
  },
  ...
],
"Response": [
  {
    "contentType": <value>,
    "description": <value>,
    "statusCode": <value>,
    "Example": [ ... ],
    "Header": [ ... ],
    "Link": [
      {
        "description": <value>,
        "name": <value>,
        "operationId": <value>,
        "operationRef": <value>,
        "requestBody": <value>,
        "serverDescription": <value>,
        "url": <value>,
        "LinkParameter": [
                      {
                        "name": <value>,
                        "value": <value>
                      },
                      ...
                    ],
                    "ServerVariable": [
                      {
                        "default": <value>,
                        "description": <value>,
                        "enum": <value>,
                        "name": <value>
                      },
                      ...
                    ]
                  },
                  ...
                ],
                "Schema": [ ... ]
              },
              ...
            ],
            "Schema": [ ... ],
            "Security": [
              {
                "apiKeyIn": <value>,
                "apiKeyName": <value>,
                "description": <value>,
                "httpBearerFormat": <value>,
                "httpScheme": <value>,
                "name": <value>,
                "oauth2ClientCredRefreshUrl": <value>,
                "oauth2ClientCredTokenUrl": <value>,
                "oauth2CodeAuthUrl": <value>,
                "oauth2CodeRefreshUrl": <value>,
                "oauth2CodeTokenUrl": <value>,
                "oauth2ImplAuthUrl": <value>,
                "oauth2ImplRefreshUrl": <value>,
                "oauth2PassRefreshUrl": <value>,
                "oauth2PassTokenUrl": <value>,
                "openIdConnectUrl": <value>,
                "type": <value>,
                "SecurityScope": [
                  {
                    "description": <value>,
                    "name": <value>
                  },
                  ...
                ]
              },
              ...
            ],
            "Server": [
              {
                "description": <value>,
                "url": <value>,
                "ServerVariable": [ ... ]
              },
              ...
            ]
          },
          ...
        ],
        "Example": [ ... ],
        "Parameter": [ ... ],
        "Response": [ ... ],
        "Schema": [ ... ],
        "Security": [ ... ],
        "Server": [ ... ]
      },
      ...
    ],
    "Tag": [
      {
        "description": <value>,
        "extDocsDescription": <value>,
        "extDocsUrl": <value>,
        "name": <value>,
        "Schema": [ ... ]
      },
      ...
    ],
    "Webhook": [
      {
        "bodyDescription": <value>,
        "bodyRequired": <value>,
        "contentType": <value>,
        "deprecated": <value>,
        "description": <value>,
        "extDocsDescription": <value>,
        "extDocsUrl": <value>,
        "method": <value>,
        "name": <value>,
        "operationId": <value>,
        "summary": <value>,
        "tags": <value>,
        "x-operationType": <value>,
        "Example": [ ... ],
          "Parameter": [ ... ],
          "Response": [ ... ],
          "Schema": [ ... ],
          "Security": [ ... ],
          "Server": [ ... ]
        },
        ...
      ]
    },
    ...
  ]
}
```

Appendix B

Results

Table B.1: Queries on less common values

Query	**Time**	**Number of rows in result**
SELECT s.id FROM Service s JOIN Request r ON s WHERE r.method = "patch"	90ms	511
SELECT s.id FROM Service s JOIN Request r ON s JOIN Response res ON r WHERE res.statusCode = 201	128ms	1225
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req WHERE res.contentType = "application/octet-stream"	52ms	300
SELECT s.id FROM Service s JOIN Request req ON s JOIN Schema sc ON req WHERE sc.type = "string"	42ms	149
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Schema sc2 ON res WHERE sc.type = "string" OR sc2.type = "string"	224ms	1131
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Property p ON sc WHERE p.name = "latitude" OR res.statusCode = 210	38ms	78

Table B.3: Queries on very common values

Query	**Time**	**Number of rows in result**

SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.method = "get"
	179ms	6936

SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Response res ON r
WHERE res.statusCode = 200
	278ms	13599

SELECT s.id
FROM Service s
JOIN Request req ON s
JOIN Response res ON req
WHERE res.contentType = "application/json"
	301ms	22358

SELECT s.id
FROM Service s
JOIN Request req ON s
JOIN Schema sc ON req
WHERE sc.type = "object"
	167ms	6209

SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Schema sc2 ON res WHERE sc.type = "object" OR sc2.type = "object"	493ms	37976
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Property p ON sc WHERE p.name = "data" OR res.statusCode = 204	217ms	1742

Table B.5: Queries with many joins

Query	Time	Number of rows in result
SELECT s.id FROM Service s JOIN Request r ON s JOIN Response res ON r JOIN Response res2 ON r WHERE r.method = "patch"	96ms	5753
SELECT s.id FROM Service s JOIN Request r ON s JOIN Response res ON r JOIN Response res2 ON r JOIN Response res3 ON r WHERE res.statusCode = 201	186ms	15372
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Request req2 ON s WHERE res.contentType = "application/octet-stream"	115ms	19201
SELECT s.id FROM Service s JOIN Request req ON s JOIN Schema sc ON req JOIN Tag ON s JOIN Response res ON req JOIN Header ON res JOIN Parameter ON req WHERE sc.type = "string"	64ms	1898
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Schema sc2 ON res JOIN Parameter p ON req JOIN Schema sch ON p JOIN Header ON res WHERE sc.type = "string" OR sc2.type = "string"	262ms	3162
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Property p ON sc JOIN Response res2 ON req JOIN Schema sc2 ON req WHERE p.name = "latitude" OR res.statusCode = 210	37ms	256

Table B.7: Queries with ORDER BY clause

Query	Time	Number of rows in result

SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.method = "patch"
ORDER BY r.path DESC
	4.3sec	511

SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Response res ON r
WHERE res.statusCode = 201
ORDER BY s.title, r.path
	8.4sec	1225

SELECT s.id
FROM Service s
JOIN Request req ON s
JOIN Response res ON req
WHERE res.contentType = "application/octet-stream"
ORDER BY req.contentType
	1.7sec	300

SELECT s.id
FROM Service s
JOIN Request req ON s
JOIN Schema sc ON req
WHERE sc.type = "string"
ORDER BY s.description DESC
	51ms	149

SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Schema sc2 ON res WHERE sc.type = "string" OR sc2.type = "string" ORDER BY req.method	6.3sec	1131
SELECT s.id FROM Service s JOIN Request req ON s JOIN Response res ON req JOIN Schema sc ON req JOIN Property p ON sc WHERE p.name = "latitude" OR res.statusCode = 210 ORDER BY req.method, s.title	41ms	78

Appendix C

Translated OAQL2 queries

Below we show some of the OAQL2 queries from section 6.1 and their translation as MongoDB queries.

C.1

OAQL2:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Schema sc ON r
JOIN Property p ON sc
WHERE p.name = "message"
OR p.name = "secret"
```

Translation in MongoDB:

```
aggregate([
  {
    "$match": {
      "$or": [
        {
          "Service.Request.Schema.Property.name": {
            "$eq": "message"
          }
        },
        {
          "Service.Request.Schema.Property.name": {
            "$eq": "secret"
          }
        }
      ]
    }
  },
  {
    "$project": {
      "s": "$Service"
    }
  },
  {
    "$unwind": "$s"
  },
  {
    "$addFields": {
      "r": "$s.Request"
    }
  },
  {
    "$unwind": {
      "path": "$r",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$addFields": {
      "sc": "$r.Schema"
    }
  },
  {
    "$unwind": {
      "path": "$sc",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$addFields": {
      "p": "$sc.Property"
    }
  },
  {
    "$unwind": {
      "path": "$p",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$match": {
      "$or": [
        {
          "p.name": {
            "$eq": "message"
          }
        },
        {
          "p.name": {
            "$eq": "secret"
          }
        }
      ]
    }
  },
  {
    "$project": {
      "_id": 0,
      "s@id": "$s.id"
    }
  }
])
```

C.2

OAQL2:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
WHERE r.method = "patch"
```

Translation in MongoDB:

```
aggregate([
  {
    "$match": {
      "Service.Request.method": {
        "$eq": "patch"
      }
    }
  },
  {
    "$project": {
      "s": "$Service"
    }
  },
  {
    "$unwind": "$s"
  },
  {
    "$addFields": {
      "r": "$s.Request"
    }
  },
  {
    "$unwind": {
      "path": "$r",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$match": {
      "r.method": {
        "$eq": "patch"
      }
    }
  },
  {
    "$project": {
      "_id": 0,
      "s@id": "$s.id"
    }
  }
])
```

C.3

OAQL2:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Response res ON r
JOIN Response res2 ON r
WHERE r.method = "patch"
```

Translation in MongoDB:

```
aggregate([
  {
    "$match": {
      "Service.Request.method": {
        "$eq": "patch"
      }
    }
  },
  {
    "$project": {
      "s": "$Service"
    }
  },
  {
    "$unwind": "$s"
  },
  {
    "$addFields": {
      "r": "$s.Request"
    }
  },
  {
    "$unwind": {
      "path": "$r",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$addFields": {
      "res": "$r.Response",
      "res2": "$r.Response"
    }
  },
  {
    "$unwind": {
      "path": "$res",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$unwind": {
      "path": "$res2",
      "preserveNullAndEmptyArrays": true
    }
  },
  {
    "$match": {
      "r.method": {
        "$eq": "patch"
      }
    }
  },
  {
    "$project": {
      "_id": 0,
      "s@id": "$s.id"
    }
  }
])
```

C.4

OAQL2:

```
SELECT s.title
FROM Service s
ORDER BY s.title
```

Translation in MongoDB:

```
aggregate([
  {
    "$project": {
      "s": "$Service"
    }
  },
  {
    "$unwind": "$s"
  },
  {
    "$sort": {
      "s.title": 1
    }
  },
  {
    "$project": {
      "_id": 0,
      "s@title": "$s.title"
    }
  }
])
```

C.5

OAQL2:

```
SELECT s.id
FROM Service s
JOIN Request r ON s
JOIN Schema sc ON r
WHERE r.method = "patch"
OR sc.x-refersTo = "https://schema.org/Store"
```

Translation in MongoDB:

```
aggregate([
  {
    "$match": {
      "$or": [
        {
          "Service.Request.method": {
            "$eq": "patch"
          }
        },
        {
          "$or": [
            {
              "Service.Request.Schema.x-refersTo": {
                "$in": [
                  "https://schema.org/Store",
                  "https://schema.org/HobbyShop",
                  "https://schema.org/GroceryStore",
                  "https://schema.org/HardwareStore",
                  "https://schema.org/JewelryStore",
                  "https://schema.org/Florist",
                  "https://schema.org/HomeGoodsStore",
                  "https://schema.org/GardenStore",
                  "https://schema.org/ToyStore",
                  "https://schema.org/PetStore",
                  "https://schema.org/SportingGoodsStore",
                  "https://schema.org/ClothingStore",
                  "https://schema.org/TireShop",
                  "https://schema.org/FurnitureStore",
                  "https://schema.org/BikeStore",
                  "https://schema.org/ElectronicsStore",
                  "https://schema.org/LiquorStore",
                  "https://schema.org/ConvenienceStore",
                  "https://schema.org/DepartmentStore",
                  "https://schema.org/PawnShop",
                  "https://schema.org/AutoPartsStore",
                  "https://schema.org/MovieRentalStore",
                  "https://schema.org/MobilePhoneStore",
                  "https://schema.org/ShoeStore",
                  "https://schema.org/WholesaleStore",
                  "https://schema.org/MensClothingStore",
                  "https://schema.org/OutletStore",
                  "https://schema.org/ComputerStore",
                  "https://schema.org/OfficeEquipmentStore",
                  "https://schema.org/BookStore",
                  "https://schema.org/MusicStore"
                ]
              }
            },
            {
              "Service.Request.Schema.x-kindOf": {
                "$in": [
                  "https://schema.org/Store",
                  "https://schema.org/HobbyShop",
                  "https://schema.org/GroceryStore",
                  "https://schema.org/HardwareStore",
                  "https://schema.org/JewelryStore",
                  "https://schema.org/Florist",
                  "https://schema.org/HomeGoodsStore",
                  "https://schema.org/GardenStore",
                  "https://schema.org/ToyStore",
                  "https://schema.org/PetStore",
                  "https://schema.org/SportingGoodsStore",
                "https://schema.org/ClothingStore",
                "https://schema.org/TireShop",
                "https://schema.org/FurnitureStore",
                "https://schema.org/BikeStore",
                "https://schema.org/ElectronicsStore",
                "https://schema.org/LiquorStore",
                "https://schema.org/ConvenienceStore",
                "https://schema.org/DepartmentStore",
                "https://schema.org/PawnShop",
                "https://schema.org/AutoPartsStore",
                "https://schema.org/MovieRentalStore",
                "https://schema.org/MobilePhoneStore",
                "https://schema.org/ShoeStore",
                "https://schema.org/WholesaleStore",
                "https://schema.org/MensClothingStore",
                "https://schema.org/OutletStore",
                "https://schema.org/ComputerStore",
                "https://schema.org/OfficeEquipmentStore",
                "https://schema.org/BookStore",
                "https://schema.org/MusicStore"
              ]
            }
          }
        ]
      }
    ]
  }
},
{
  "$project": {
    "s": "$Service"
  }
},
{
  "$unwind": "$s"
},
{
  "$addFields": {
    "r": "$s.Request"
  }
},
{
  "$unwind": {
    "path": "$r",
    "preserveNullAndEmptyArrays": true
  }
},
{
  "$addFields": {
    "sc": "$r.Schema"
  }
},
{
  "$unwind": {
    "path": "$sc",
    "preserveNullAndEmptyArrays": true
  }
},
{
  "$match": {
    "$or": [
      {
          "r.method": {
            "$eq": "patch"
          }
        },
        {
          "$or": [
            {
              "sc.x-refersTo": {
                "$in": [
                  "https://schema.org/Store",
                  "https://schema.org/HobbyShop",
                  "https://schema.org/GroceryStore",
                  "https://schema.org/HardwareStore",
                  "https://schema.org/JewelryStore",
                  "https://schema.org/Florist",
                  "https://schema.org/HomeGoodsStore",
                  "https://schema.org/GardenStore",
                  "https://schema.org/ToyStore",
                  "https://schema.org/PetStore",
                  "https://schema.org/SportingGoodsStore",
                  "https://schema.org/ClothingStore",
                  "https://schema.org/TireShop",
                  "https://schema.org/FurnitureStore",
                  "https://schema.org/BikeStore",
                  "https://schema.org/ElectronicsStore",
                  "https://schema.org/LiquorStore",
                  "https://schema.org/ConvenienceStore",
                  "https://schema.org/DepartmentStore",
                  "https://schema.org/PawnShop",
                  "https://schema.org/AutoPartsStore",
                  "https://schema.org/MovieRentalStore",
                  "https://schema.org/MobilePhoneStore",
                  "https://schema.org/ShoeStore",
                  "https://schema.org/WholesaleStore",
                  "https://schema.org/MensClothingStore",
                  "https://schema.org/OutletStore",
                  "https://schema.org/ComputerStore",
                  "https://schema.org/OfficeEquipmentStore",
                  "https://schema.org/BookStore",
                  "https://schema.org/MusicStore"
                ]
              }
            },
            {
              "sc.x-kindOf": {
                "$in": [
                  "https://schema.org/Store",
                  "https://schema.org/HobbyShop",
                  "https://schema.org/GroceryStore",
                  "https://schema.org/HardwareStore",
                  "https://schema.org/JewelryStore",
                  "https://schema.org/Florist",
                  "https://schema.org/HomeGoodsStore",
                  "https://schema.org/GardenStore",
                  "https://schema.org/ToyStore",
                  "https://schema.org/PetStore",
                  "https://schema.org/SportingGoodsStore",
                  "https://schema.org/ClothingStore",
                  "https://schema.org/TireShop",
                  "https://schema.org/FurnitureStore",
                  "https://schema.org/BikeStore",
                  "https://schema.org/ElectronicsStore",
                  "https://schema.org/LiquorStore",
                  "https://schema.org/ConvenienceStore",
                  "https://schema.org/DepartmentStore",
                  "https://schema.org/PawnShop",
                  "https://schema.org/AutoPartsStore",
                  "https://schema.org/MovieRentalStore",
                  "https://schema.org/MobilePhoneStore",
                  "https://schema.org/ShoeStore",
                  "https://schema.org/WholesaleStore",
                  "https://schema.org/MensClothingStore",
                  "https://schema.org/OutletStore",
                  "https://schema.org/ComputerStore",
                  "https://schema.org/OfficeEquipmentStore",
                  "https://schema.org/BookStore",
                  "https://schema.org/MusicStore"
                ]
              }
            }
          ]
        }
      ]
    }
  },
  {
    "$project": {
      "_id": 0,
      "s@id": "$s.id"
    }
  }
])
```