Client-side querying of media fragments using a low-cost server interface

Brecht Hendrickx

Supervisor: Prof. dr. ir. Ruben Verborgh
Counsellors: Dr. ir. Miel Vander Sande, Joachim Van Herwegen, Ir. Ruben Taelman

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Koen De Bosschere
Faculty of Engineering and Architecture
Academic year 2017-2018
Client-side querying of media fragments using a low-cost server interface

Brecht Hendrickx

Supervisor: Prof. dr. ir. Ruben Verborgh
Counsellors: Dr. ir. Miel Vander Sande, Joachim Van Herwegen, Ir. Ruben Taelman

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Koen De Bosschere
Faculty of Engineering and Architecture
Academic year 2017-2018
Preface

In the following 90 pages or so you will find the thesis “Client-side querying of Media Fragments using a low-cost server interface” which is a study of how querying for media files on the (Semantic) Web can be achieved in a scalable and less expensive way. It has been written to fulfil the graduation requirements of the Master of Science in Computer Science Engineering at Ghent university. I was engaged writing this thesis from September 2017 to August 2018.

Writing this thesis has thought me how doing scientific research can be incredibly satisfying while at the same time demanding a lot of energy and sacrifices from the researcher. While this seems unfair at the time, the feeling of being able to contribute to the knowledge in the world eventually surpasses all the negative feelings one might deem to be so demanding at the time. I would therefore like to thank my supervisors Joachim Ruben and Miel for their help and friendliness. They provided me with excellent feedback, always made time when I had questions and always had my back. It is by being in contact with your supervisors and by writing your thesis that you really get to grasp how the academic world is a wondrous place in which people come together to push the boundaries of knowledge.

With this thesis, I conclude a chapter that has been going on for six years. People always say that being a student is the best time of your life. Even though I still have a long life to live, I do feel like they might be right. During your student years, you not only get to learn theoretical knowledge but also life lessons. You learn to make sacrifices but also how to enjoy life and make time for fun. You learn that life can be disappointing at some times but that at the end, everything will always be alright as long as you have people you can rely on. I would therefore like to thank my family and especially my parents for supporting me during every step of the way. They always believed in me and provided me with the warmth, kindness and love from which I got the strength to push through even when things got rough. I therefore will always be grateful for the opportunity they gave me.

I hope you enjoy your reading.

Brecht Hendrickx

Gooik, 2018
Copyright notice

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

Ghent, August 9, 2018
Abstract

In this thesis, a media querying system is implemented which will provide expressive search queries that can define spatial and temporal relations of objects inside media files. This by using Semantic Web technologies. Even though some systems, like SPARQL-MM, already provide this functionality, they are all centralised and rely on the server to fully solve the queries. This leads to high CPU usage of the server when multiple clients are searching at the same time and will cause for some scalability issues. Therefore, the proposed system will try to solve this issue and lower the CPU usage of the media hosting server compared to current centralised approaches by moving CPU intensive tasks to the clients. In order to do this, the Triple Pattern Fragments framework has been extended to provide the same media querying functionalities as SPARQL-MM. Next to this, we will also investigate how we can use the Media Fragments URI structure which is used to represent the objects in the media files to enhance the Triple Pattern Fragments framework’s performance when querying for media files. More specifically, two optimisations are proposed which will allow for faster query execution times and/or lower bandwidth usage in specific cases. The first optimisation will perform query rewriting before the start of the execution in order to ensure an optimal query execution order. The second optimisation will use the structure imposed by the Media Fragments URI standard to solve triple patterns locally where possible. Eventually, the performance will be compared between the unoptimised and optimised Triple Pattern Fragments framework followed by a comparison of both implementations with the centralised SPARQL-MM implementation in Apache Marmotta.

Keywords: Triple Pattern Fragments, SPARQL-MM, Media Fragments URI, client-side querying, Semantic Web
Client-side querying of media fragments using a low-cost server interface

Brecht Hendrickx
Supervisors: Joachim Van Herwegen, ir. Ruben Taelman, dr. ir. Miel Vander Sande

Abstract—This paper will investigate how we can create a media querying system that will allow users to search for media files on the Semantic Web with great expressiveness while at the same time lowering the CPU usage of the media hosting server. In order to do this, the system will extend the Triple Pattern Fragments framework to enable it with all the media querying functionality SPARQL-MM provides as an extension to current centralized Sesame triple stores. Next to this, we will also investigate how we can make use of the Media Fragments URI standard to improve the performance of the Triple Pattern Fragments framework while searching for media files on the Semantic Web. The implementation of the optimizations and SPARQL-MM functionality is finally tested and the performance is compared to the centralized implementation of SPARQL-MM in Apache Marmotta.

keywords—Semantic Web, SPARQL-MM, Media Fragments URI, Triple Pattern Fragments, client-side media querying

I. INTRODUCTION

Creating images, videos and audio files has never been easier. With the rise of smartphones and mobile internet, users are massively uploading media files onto social media sites like Facebook, Twitter, Instagram etc. Even though these are widely used, they do have some shortcomings. There have been a lot of cases where which have proven that social media sites do not take the privacy of their users much into consideration when trying to increase their profit. We thus need a way to take control of our own data again and create a social media platform in which you have full control over who sees what information. For this, the Semantic Web can be used as a base platform as one of the main purposes of this technology is to create a fully decentralized web. Since media sharing and searching is a big part of current Social Media sites, we therefore need a way to search for media on the Semantic Web. SPARQL-MM currently already provides this functionality and allows users to search for media files with great expressiveness which exceeds the capabilities of current social media sites. It is however only implemented as an extension for centralized Sesame triple store solutions meaning that the search for media files needs to be handled by the server itself. This is very CPU intensive and could become problematic when a large amount of users are searching simultaneously. We therefore need to alleviate the server from CPU intensive tasks by moving most of the media searching functionality to the client. This will enable us to facilitate a large amount of users at the same time which is essential for sites with huge amounts of users like social media sites. In this paper we will investigate how we can create such a system and identify the benefits and drawbacks it will have.

The fact that the system works with Semantic Web technologies does however not mean that it can not be used by current systems. It provides much more powerful searching capabilities compared to the ones provided by the current social media sites while at the same time diminishing the CPU load of the server. This does however come at a cost of slower query execution speeds which will be explained later on.

In the following section we will discuss the relevant related work. After this, the proposed solution will be elaborated upon together with an investigation on how we can use URI structures to further improve our solution. Next, the results will be represented and explained. Finally, some final remarks are made together with a listing of some possible future work.

II. RELATED WORK

A. Semantic Web

The Semantic Web [1] is a collection of technologies which will allow machines to interpret data and create a world-wide decentralized database over which they will be able to reason and solve complex problems. In order to do so, we need some standardization as otherwise different machines could not interpret each others information. Therefore, the Linked Data [2] initiative created a standardized stack of Semantic Web technologies. It uses RDF[3] as the main data model to represent information. This is a graph based model which uses triples to denote information about resources on the Semantic Web. These resources are all identified by a unique URI. By creating triples that define relationships between two URIs, resources on the Semantic Web can be linked together in a standardized manner, thus creating graphs. Machines can then use these links to reason over the data provided on the Semantic Web.

B. SPARQL and SPARQL-MM

While RDF allows us to model data in a standardized way, we need an efficient way to consume this data and find the information we need on the Semantic Web. Linked Data therefore defined SPARQL [5], which is a query language and protocol, in the stack of Semantic Web technologies. RDF datasets will provide SPARQL endpoints in which users can input SPARQL formatted queries which are in essence graph patterns. The endpoint will then perform pattern matching on the RDF data and provide the user with the desired results. The SPARQL query language standard already is quite extensive and allows for a highly expressive queries but it is not tailored to querying media files. Therefore, Kurz et al. have proposed SPARQL-MM [6] as an extension on the SPARQL query language to enable users to query for media files on the Semantic Web in a semantically easy understandable way while still providing a lot of functionality.
enabled endpoints are able to search not only for media files but also for fragments defined in the media files themselves. For this, it uses the structure imposed by the Media Fragments URI [7] standard to perform reasoning over the spatial and/or temporal fragments that are defined by it. SPARQL-MM will provide the users with spatial and temporal relational functions (a left beside b, a after b, a overlaps b, etc.), accessor functions (width of fragment a, start time of fragment a, etc.) and aggregation functions which can combine two fragments into one fragment. By doing this, the clients can create media searching queries that have a complexity that can not be met by most current search engines incorporated in social media sites.

C. Media Fragments URI

The standardized stack with Semantic Web technologies has defined URIs [8] as a standard to identify resources on the Semantic Web. These URIs are used in RDF triples to define properties about the resource identified by the URI or even to defined relations between two or multiple resources. The normal URI standard therefore provides us with a lot of functionality but it has a big flaw when it comes to media resources and more specifically media fragments. It only allows for the whole media resource to be identified and not the fragments in them. In order to solve this problem, the Media Fragments URI standard [7] has been created. This will allow clients and servers to interpret URIs that define fragments in media resources. The Media Fragments URI standard allows for temporal fragments to be defined which will indicate which time frame of the source video or audio file the client wants to retrieve. Next to this, the standard also allows for fragments to be defined in the spatial dimension meaning that the URI can indicate which specific pixels the client wants to retrieve from the source image of video file. The URI https://www.example.org/image.jpg?xywh=5,5,10,10 for example indicates that we want the rectangular sub-image with top-left coordinates (5,5) and bottom-right coordinates (15,15). All the fragment identifiers follow the same convention and are preceded by a number sign (#) or question mark (?), meaning that the information that the URI contains is twofold. It contains information about which source media file we are identifying while at the same time also containing information about which fragment of the media file we are identifying.

D. Triple Pattern Fragments

Public SPARQL endpoints have a tendency to have very bad availability which makes using the Semantic Web mostly a hit-or-miss affair when searching for information. This is due to the unlimited complexity of the queries clients can send to them. This together with the public availability means that most public SPARQL endpoints get bottlenecked very often due to overloaded server CPUs, consequently denying service for many users. The Triple Pattern Fragments framework [9] (TPF) aims to solve this problem by moving part of query solving functionality to the clients, which will effectively lower the CPU load of the server. The downside to this approach however is a higher data usage and noticeably slower query execution speeds when filters are used in the SPARQL queries. The server will only accept triple patterns to be queried and not full SPARQL queries as these can fairly easily be solved by the server with minimal CPU usage. It is therefore the responsibility of the client to divide the entire SPARQL query into triple patterns and plan the best order in which it will query the server. For this, it will rely on the metadata returned by the server which will indicate the estimated amount of matches for the given triple pattern. The client will retrieve this metadata for each triple pattern that remains in the execution and will always select the one with the least estimated amount of results first. It will thus use a greedy algorithm. Except from some specific cases, this will yield the fastest query execution order.

III. SOLUTION

As already explained in the introduction, SPARQL-MM does provide us with the needed functionality to be able to query for media files with great expressiveness and granularity. In order to do this, it assumes that the Media Fragments URI structure is used to define fragments in the media files. Based on the information inside these URIs, the SPARQL endpoint can then extract spatial and/or temporal information about the resources in its knowledge graph and reason on them to solve the client’s query. The main problem is that the query needs to be solved by the server itself. This problem is inherent to SPARQL endpoints but as the SPARQL-MM filters and functions are relatively more complex compared to native SPARQL ones, this problem will be even bigger. Our solution will therefore move the CPU intensive query solving functionality to the client by using the TPF framework. We will extend this framework to support SPARQL-MM queries, thus allowing clients to use them without alteration. The TPF framework however performs much slower when filter statements are used in the queries. Unfortunately, SPARQL-MM heavily relies on these filter statements to provide the user with useful functionality like defining spatial and temporal relations between two fragments in a media file. We therefore need to try and compensate this flaw as much as possible. In order to do this, we will make use of the URI structure defined by the Media Fragments URI standard to make the TPF client more intelligent and speed up the SPARQL-MM queries wherever possible.
A. Implementing SPARQL-MM in Triple Pattern Fragments

Since in TPF, all query solving functionality resides in the client, we only need to alter the client implementation and can leave the server implementation unaltered. In order to provide the same functionality as SPARQL-MM in the TPF client, we need to be able to reason with spatial fragments and temporal fragments of media files. This requires for mathematical models to be chosen which will uniquely determine how fragments are related to each other. After considering different models for both temporal and spatial reasoning, we have eventually opted to use the same models as SPARQL-MM to do our calculations in the TPF implementation. This because they are the easiest models to use and will result in exactly the same behaviour as regular SPARQL-MM enabled endpoints. If different models would have been used and a users would switch to TPF, the same query could yield different results which would make interoperability impossible.

For the topological spatial relation calculation ("intersects", 'equal to' etc.), the DE-9IM model is used which will define ten topological relations between two spatial objects. For the spatial directional calculations, the rectangular algebra [10] model is used. This model will only work for rectangular spatial entities. The latest version of the Media Fragments URI standard only however allows for rectangular spatial fragments to be defined so this is no issue. The model defines nine different directional relations ('north', 'east', 'southeast' etc.) which all coincide with the directions humans naturally use in their daily life. For the temporal relational functions, Allens interval algebra [11] is used which is the most widely used model for defining temporal relations. It defines thirteen possible relations between two temporal fragments, each being provided to the end user by filter functions.

SPARQL-MM however also provides some accessor functions to retrieve information about a fragment like the height, start time etc. and aggregation functions which enables users to combine two fragments into one. These functions are simple and do not need any mathematical models. Just like the relational functions, all accessor and aggregation functions defined in the centralized implementation of SPARQL-MM are also available in the TPF implementation. A list of all SPARQL-MM relational, accessor and aggregation functions that are also supported by the extended TPF framework can be found on the SPARQL-MM GitHub page.

By using the same mathematical models and same function identifiers, the TPF based solution can now fully support SPARQL-MM enabled queries without any need for alterations. An example of such a query can be found in listing 1.

B. Deriving bindings locally

In order to cope with the degraded performance of the filters in TPF, we will try to speed up the execution and diminish the bandwidth usage by solving triple patterns locally without having to contact the server wherever possible. This can be done because of the inherent structure the Media Fragments URI standard imposes on the data in the knowledge graph. When solving a query, the TPF client will use an iterative execution where it will check the estimated amount of matches for each triple pattern in the Basic Graph Pattern (BGP) of the query. It will then select the triple pattern with the least amount of matches and retrieve all solution mappings for the unbound variable in this triple pattern. After this, it will remove the triple pattern from the BGP. Then it will apply each possible solution for the unbound variable to the BGP, thus filling in the value of the variable in the remaining triple patterns after which it will again start the same procedure. This will repeat until a result for the query is found or until we have a triple pattern that has zero results, meaning that there is no solution for the current solution mapping. This process is done in iteratively and allows for results to be returned in a streaming fashion. In listing 1, $tp_3$ will for example be resolved first which can result in the following solution mapping for $?f1$: $\lambda_1 = \{ ?f1 \rightarrow http://www.example.org/image#xywh=percent:20,20,5,5 \}$. When this value is used, the regular TPF client will now retrieve $tp_1$ from the server as this will be the one with the least number of matches (being 1). This request to the server can however be omitted as the Media Fragments URI that is found for $?f1$ already contains the information from which source image it originates. The TPF client can therefore immediately update the solution mapping to also contain the URI of the unbound variable $?image$. Since this

---

Listing 1: Example SPARQL-MM query finding a picture where a person is to the left of a car

```sparql
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?f1 ?f2 WHERE {
  ?image ma:hasFragment ?f1; ma:hasFragment ?f2. # ± 40,000 matches (tp1)

  ?f1 dc:description "person". # ± 5,000 matches (tp3)
  ?f2 dc:description "car". # ± 6,000 matches (tp4)

  FILTER mm:leftBeside(?f1,?f2)
}
```

---

1 https://github.com/tkurz/sparql-mm/blob/master/ns/2.0.0/function/index.md
request can be spared for each of the 5 000 matches of $\text{?f1}$, we can effectively spare 5 000 requests to the server. When the latency between the client and server is high, this can cause for some noticeable speed improvements. On top of that, the less amount of requests also means less bandwidth usage and a lower CPU usage of the server. Thus further improving the scalability of the system.

The optimized TPF client will therefore check for every solution mapping that is retrieved from the server if it can use this mapping to create a binding for an unbound variable locally. If so, it will update the solution mapping and remove the corresponding triple pattern from the BGP, thus effectively solving two triple patterns of the BGP with only one request to the server.

C. Query rewriting

Another way of using the structure of Media Fragments URI is to perform query rewriting before the execution starts. Even though it does not seem this way, the queries in listings 1 and 2 will yield exactly the same result when being executed. The filter functions implemented to support SPARQL-MM will after all check for themselves if both fragments that are given as argument have the same source media file. The first two lines of the select clause in listing 1 therefore seem unnecessary. On top of that, the query in listing 2 might seem more logical to most people. We just need to find the two fragments that follow the left beside relation, nothing more nothing less. There is however a big difference in how the latter query will be executed.

Just like in the previous section, the TPF client will start by retrieving all solutions for $\text{tp}_3$ as this has the least number of estimated results. However after this is done, the TPF client has no other choice than to check each of the 5 000 values of $\text{?f1}$ with each of the 6 000 possible values of $\text{?f2}$ as $\text{tp}_4$ is the only other triple pattern in the BGP. This means that the client has to execute the filter for each of these combinations which is very computationally intensive and will be very slow. The query execution order in 1 will therefore be much more optimal. The first and second line in the SELECT clause will essentially implement the part of the filter that will check whether both fragments are from the same source media file. If not, they will yield no result and the filter will not have to be executed. This query will therefore essentially perform early filtering of the results which will lead to much faster execution times and less CPU load on the client. The number of requests to the server will however increase as more triple patterns need to be solved.

Therefore, before executing the server, the query, the optimized TPF client will check if SPARQL-MM filters are used in the query. If so, it will check which unbound variables are given as arguments. After this, it will traverse the BGP checking if we have triple patterns denoting that both fragments have the same source media file by looking for the predicate $\text{ma:hasFragment}$ followed by the unbound variable. If not, the client will insert these triple patterns into the BGP after which it will start the execution.

IV. Results

We have performed tests for deriving the performance of our solution compared to the centralized SPARQL-MM implementation in Apache Marmotta. For this, a dataset of more than 40 000 images and over 200 000 media fragments is used. The experiment will simulate an increasing amount of simultaneous clients querying for media files in order to test the scalability and performance under heavy load. The experiment consists of six identical machines hosted on the Virtual Wall \(^2\) of the University of Ghent. Each machine has a 2x Quad core Intel E5520 (2.2GHz) CPU, 12GB RAM and a 2 - 4 gigabits NIC and are all connected to each other through a local network with a latency of around 0.2 ms. One machine acts as a dedicated server which will only run the Apache Marmotta or TPF server behind an NGINX caching proxy. The other machines are all used to create clients which will simultaneously query the server for media files. One of these will also act as a monitoring entity and will collect statistics. This setup allows us to create up to 200 simultaneous clients without bottlenecking any of them. The same queries are used in the same order for the Apache Marmotta solution, the unoptimised SPARQL-MM enabled TPF solution and the optimised TPF solution implementing local binding updates and query rewriting. The results of this experiment can be found in figure 2.

From graph 2a we can clearly see that the Marmotta implementation at first has a far faster execution speed compared to the TPF solutions. The execution times however rise much faster and when 45 clients are querying simultaneously, even the unoptimized TPF solution has already caught up. After this the Marmotta implementation starts to become very unstable and after 65 simultaneous clients, every single client started to be rejected. This is also what we can see in graph 2g. Even from as little as 10 simultaneous clients, the Marmotta server already starts to reject some clients. This only gets worse and after a while no single client is served anymore. This is also why the graph for the execution of Marmotta stops at 65 simultaneous clients. From the execution time graph we can also see that the unoptimized TPF solution has very slow query execution times of more than 125 seconds in every case. This is due to the fact that there are also suboptimal queries like the one in listing 2 in the test set. We can clearly see that by performing query rewriting in combination with the local binding updates, we can drastically improve the performance of the TPF solution with now an average execution time under 60 seconds. Even though the Apache Marmotta server outperforms the TPF solutions at the first run, it very quickly starts to bottleneck which causes for slower execution speeds and lots of clients being rejected, causing the TPF solution to have much better scalability and allowing for more clients to be served with the same hardware. We can also see this in graphs 2d - 2f. The CPU usage of the Marmotta server rises very fast and even for as little as 5 simultaneous clients it is using as much CPU as it can. The NGINX proxy server can not help the Marmotta server as each query is unique meaning

Listing 2: Example suboptimal SPARQL-MM query finding a picture where a person is to the left of a car

```
PREFIX ma: <http://www.w3.org/ns/ma-ontology#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?f1 ?f2 WHERE {
  ?f1 dc:description "person".
  ?f2 dc:description "car".
}
```

that caching is not very effective for SPARQL endpoints. The TPF solutions however have a much lower CPU usage and tend to increase more slowly with an increasing amount of simultaneous clients. From both TPF solutions, the optimized one seems to perform the best in terms of combined CPU usage. This can be explained as follows: due to the query rewriting, more triple patterns need to be requested by each client. This means that there is a higher chance of having a cached version of the response in the NGINX proxy server as triple patterns are not as unique as SPARQL queries. We can also see this in the graphs. Even though the regular TPF solution already greatly benefits from the NGINX caching server, the rewritten queries will result in even more cache hits, causing the NGINX server to use more CPU. The caching server can however return the results much more efficiently compared to the TPF server, causing an overall lower CPU usage. The TPF solution thus indeed allows for more clients to be served simultaneously with the same hardware. It however comes at a cost which is an increased number of requests and an increased bandwidth usage which can be seen in graphs 2b and 2c. This is inherent to the query solving method of the TPF framework. In stead of having to send one request to the server containing the query, the TPF client needs to send multiple requests with triple patterns in order to solve the query. This number of requests also depends on the number of matches there are for the previous triple pattern as each solution mapping will cause for a different subsequent query execution with each of the triple patterns that need to be requested. The optimized TPF solution does seem to perform worst when it comes to bandwidth usage. This is due to the fact that the rewritten queries will cause for far more triple patterns to be requested compared to the suboptimal ones. In example listing 2, the TPF client for example only needs to retrieve all results for two triple patterns, causing for less requests to be done and subsequently less bandwidth to be used. In listing 1 however, each of the possible solutions of \(?f1\) will cause multiple triple patterns to be requested. Even though the local binding updates will lower this amount a bit, the added requests by the extra triple patterns can not all be derived locally, thus resulting in more requests over all compared to the unoptimized TPF solution which will not perform query rewriting or local binding updates.

V. Conclusion

In this paper we have shown how it is possible to provide SPARQL-MM functionality in the Triple Pattern Fragments framework and how it results in a more scalable solution for querying media on the Semantic Web. It allows for far more clients to simultaneously search for media files with the same hardware, thus effectively making operation costs cheaper...
for companies as less expensive hardware is needed. The execution times are however slower but since results are shown in a streaming fashion, the end user does not have to wait for all results to be found before being presented with solutions. On top of that, we have shown how information embedded in URIs themselves can be used to make the TPF framework more intelligent and aid it in (dynamically) optimizing its query execution. As has been proven, this can lead to some quite significant improvements.

VI. FUTURE WORK

For now, the system relies on users to define SPARQL-MM queries to find their media. As this is not very usable for most people, a natural language processor could be added to translate natural sentences into correct SPARQL-MM queries. The current solution also will simply return the URI of the found media files. Users would still have to click or copy-paste the URI in their browser before being able to see the results. Therefore, one could create a more dynamic and visually pleasing interface which will retrieve and show the resulting media files in place as they are being found. Next to this, we have also proven how the structural information of Media Fragments URI can be used to make the TPF client more intelligent when searching for media files. Further research could investigate how we can use the concept of using information embedded in URIs to further improve the TPF framework both in general cases or to counteract specific bottlenecks.

REFERENCES


Source code

The source code for the SPARQL-MM enabled TPF client with optimisations can be found at https://github.com/bhendric/TPF_SPARQLMM. The code for executing the performance tests can be found at https://github.com/bhendric/TPF_SPARQLMM_RESULTS which also contains the results of the experiments performed in this thesis.
Contents

Source code xii

List of Figures xvii

List of Tables xviii

1 Introduction 1

2 The Semantic Web 6

2.1 General view of the Semantic Web 6

2.2 Main principles of the Semantic Web 7

2.2.1 Principle 1: Everything can be identified by URIs 7

2.2.2 Principle 2: Resources and links can have types 8

2.2.3 Principle 3: Parallel information 9

2.2.4 Principle 4: Freedom of data 10

2.2.5 Principle 5: Support for evolution 10

2.2.6 Principle 6: Minimalistic design 11

2.3 The Semantic web stack and technologies 11

2.3.1 URI and Character Set: Unicode 12

2.3.2 Resource Description Framework (RDF) 13

2.3.3 SPARQL 16

3 Media Fragments URI 17

3.1 The Media Fragments URI standard 17

3.1.1 URI fragments vs URI queries 18

3.2 Media fragments Syntax 19

3.2.1 Temporal dimension 20

3.2.2 Spatial dimension 21

3.2.3 Combining multiple identifiers 22
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td><strong>SPARQL and SPARQL-MM</strong></td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>SPARQL and the Semantic Web</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>SPARQL syntax</td>
<td>24</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Some basic examples</td>
<td>24</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Adding constraints</td>
<td>27</td>
</tr>
<tr>
<td>4.2.3</td>
<td>General working of SPARQL</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>SPARQL-MM</td>
<td>29</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Topological functions</td>
<td>30</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Directional functions</td>
<td>31</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Spatial access functions</td>
<td>31</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Temporal functions</td>
<td>32</td>
</tr>
<tr>
<td>4.3.5</td>
<td>SPARQL-MM implementation</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td><strong>Triple Pattern Fragments</strong></td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Triple Patterns and main concept</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Query Execution</td>
<td>36</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Conceptual workflow</td>
<td>36</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Iterative implementation of query execution</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Limitations of Triple Pattern Fragments</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td><strong>Research question and hypotheses</strong></td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td><strong>Client-side media querying</strong></td>
<td>45</td>
</tr>
<tr>
<td>7.1</td>
<td>Providing SPARQL-MM functionality in Triple Pattern Fragments</td>
<td>45</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Spatial accessor, aggregation and relational functions</td>
<td>45</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Temporal accessor, aggregation and relational functions</td>
<td>54</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Using the SPARQL-MM spatial functions with Triple Pattern Fragments</td>
<td>58</td>
</tr>
<tr>
<td>7.2</td>
<td>Triple Pattern Fragments client optimisations</td>
<td>60</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Deriving bindings locally</td>
<td>60</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Query rewriting</td>
<td>61</td>
</tr>
<tr>
<td>7.3</td>
<td>Implementation details</td>
<td>62</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Implementing SPARQL-MM functions</td>
<td>63</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Number of smartphone users worldwide from 2014 to 2020 (in billions), Graph from Statista [3]</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Most famous social network sites worldwide as of January 2018, Graph from Statista [4]</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Search results of popular image hosting sites</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of URIs used as identifiers for physical resources</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of human interpretable web pages</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Resources and links as used in Web versus Semantic Web implementation</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Semantic Web support for evolution demonstrated in time</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Semantic Web stack, Figure from Wikipedia [14]</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>An example of a URI, Figure from Wikipedia [16]</td>
<td>13</td>
</tr>
<tr>
<td>2.7</td>
<td>Visual representation of the notion “Grass is green”</td>
<td>14</td>
</tr>
<tr>
<td>2.8</td>
<td>Example of a RDF graph</td>
<td>15</td>
</tr>
<tr>
<td>2.9</td>
<td>Example use of a blank node</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Finding name of employee</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Multiple matches</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Matching literals with SPARQL queries</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of Linked Data access methods in function of client cost, server cost, and request specificity, Figure from Linked Data Fragments [30]</td>
<td>35</td>
</tr>
<tr>
<td>7.1</td>
<td>Using the centroid model to perform topological calculus</td>
<td>46</td>
</tr>
<tr>
<td>7.2</td>
<td>Graphical representation of the topological relations defined in RCC-8, figure from “Region Connection Calculus: Its models and composition table” [33]</td>
<td>47</td>
</tr>
<tr>
<td>7.3</td>
<td>Implementation of RCC-8 in GeoSparql, figure from the RCC Wikipedia page [34]</td>
<td>48</td>
</tr>
<tr>
<td>7.4</td>
<td>Graphical representation of the DE-9IM matrix, figure from the DE-9IM Wikipedia page [35]</td>
<td>49</td>
</tr>
<tr>
<td>7.5</td>
<td>Cone and projection reference frame of defined by Frank</td>
<td>50</td>
</tr>
<tr>
<td>7.6</td>
<td>Reference frame and directional relations defined by Rectangular Algebra</td>
<td>51</td>
</tr>
</tbody>
</table>
7.7 Visual representation of the thirteen temporal relations defined be Allen's interval algebra 55
7.8 Example queries ................................................................. 59

8.1 Distribution of classes in Open Images Validation Dataset, figure from Open Images Dataset [39] .................................................. 69
8.2 Graphs of the query rewrite experiment .................................. 76
8.3 Graphs of the local binding update experiment .......................... 78
8.4 Graphs of the final experiment comparing regular TPF, optimised TPF and Apache Marmotta 84
List of Tables

7.1  Topological spatial functions implemented in the Triple Pattern Fragments client . . . . . 49
7.2  Directional spatial functions implemented in the Triple Pattern Fragments client . . . . . 52
7.3  Accessor functions for spatial fragments implemented in the Triple Pattern Fragments client . . . 53
7.4  Temporal relation functions implemented in the Triple Pattern Fragments client . . . . . 56
7.5  Accessor functions for temporal fragments implemented in the Triple Pattern Fragments client . . . 57
7.6  Aggregation functions for temporal fragments implemented in the Triple Pattern Fragments client . . . 57
8.1  Number of images and bounding boxes for the Open Images Dataset . . . . . . . . . . . . . 70
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>BGP</td>
<td>Basic Graph Pattern</td>
</tr>
<tr>
<td>TPF</td>
<td>Triple Pattern Fragments</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>RCC8</td>
<td>Region Connected Calculus</td>
</tr>
<tr>
<td>DE-9IM</td>
<td>Dimensionally Extended nine-Intersection Model</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Creating photos and videos has never been easier. According to Statista (graph 1.1), 2.53 billion people are using a smartphone and this number is expected to keep increasing to 2.87 billion in 2020. With a current world population of 7.6 billion people, this means that over a third of the people in the world have the resources to take pictures and videos and put them on the web. The most popular way to do this is by using social network sites or applications like Facebook, Instagram, Twitter, Snapchat etc. As can be seen in graph 1.2, they indeed have a vast amount of users. Facebook currently has over 2.167 billion monthly active users, followed by YouTube with 1.5 billion active users and WhatsApp with 800 million monthly active users. This causes for a massive amount of media being published on the web. For example, in 2013, Facebook released a white paper [1, p. 6] stating that 235 million photos are uploaded on Facebook every day while in January 2017, Instagram was reported [2] to have 95 million photos uploaded daily to the platform.

Graph 1.1: Number of smartphone users worldwide from 2014 to 2020 (in billions). Graph from Statista [3]

Even though these social media sites are very popular, they have some serious flaws. First of all, there are some serious privacy issues. When uploading photos and videos to Facebook, the media is automatically analysed using machine learning. This algorithm will detect different segments in pictures and categorize them. Facebook is currently using this technology to create automatic captions for all the photos that
CHAPTER 1. INTRODUCTION

Graph 1.2: Most famous social network sites worldwide as of January 2018, Graph from Statista

are uploaded. These captions are then used to aid blind people as the automated caption is spoken out loud when a picture is displayed on the screen. Next to this, the search functionality has also been expanded and allows you to search for photos which contain certain objects in them. This functionality is however still very limited and often results in inaccurate matches and a very low expressiveness. As can be seen in figure 1.3, object recognition does work correctly but any sort of spatial relation between different objects in a picture can not be searched for. We have performed a few queries searching for images that have objects with a certain relation between them. In the query “cat right beside dog”, Facebook was able to detect the dog in the pictures but there are no cats shown, indicating that spatial reasoning is not present. For the queries “person left beside car” and “person left beside tree”, which are shown to the right of the previous query, Facebook did not return any results at all. Flickr is also unable to correctly return images for the query “cat left beside dog” and Instagram does not even support searching for images based on the subjects inside them. When you have a huge collection of images and videos, which is the case on social media sites, it would be very beneficial to be able to express these spatial and temporal relations. This would enable you to search for images where your father stands left beside your mother for example. Even though the object recognition technology provides the users some with added functionality (it was
able to detect dogs in images), Facebook also declared its intentions to use the same recognition algorithm for commercial and health analysis purposes \[\text{[5]}\]. This is another step in the privacy issues that Facebook has been having the last couple of years. Not only Facebook, but also Instagram has some big privacy issues. According to their terms of service \[\text{[6]}\], third parties are able to pay Instagram for which in return they can use your username and photos, together with its metadata, to use in paid or sponsored content without any compensation to the user. This means that third parties and Instagram itself can use your own photos in advertisements without you knowing or having to give an additional permission. These are some serious problems which the users are not aware of when they create an account for these social media sites.

![Search results of popular image hosting sites](image)

**Figure 1.3: Search results of popular image hosting sites**

Privacy is thus a very big problem in our society which is based on sharing and online presence. More and more people are getting aware of this fact and are causing a slow evolution which recently got a faster pace after the latest Facebook privacy scandal in which personally identifiable data of users has been shared with Cambridge Analytica which allegedly used this data to influence the behaviour of voters in favour of the politicians who hired them. When this came to light, Mark Zuckerberg, the CEO of Facebook, was summoned to appear for the American senate in which he was questioned about the privacy policies and behaviour of Facebook towards its users. In this hearing, the senators stated that change is needed. Senator Amy Klobuchar made the following statement during the hearing: “We’re going to have to do privacy legislation now”, indicating that the American government is working on ways to regularise the data usage of users. In Europe, the same movement has started which resulted in the GDPR which will go into effect on May 25, 2018. This new European law requires companies to report which data is gathered, how it is used by the company, for which purposes etc. On top of that, some additional rules have been
incorporated in the law. Companies will now have to explicitly ask the users if they can gather their data. The data can also only be stored for a limited time and should be processed in such extent to obtain the preset goals of the service you are using. If a company states that their goal is to create a profile of you and show you recommendations for new purchases for example, they are allowed to do so, provided they asked permission and clearly stated their intentions to the end user. More information about the GDPR Law can be found in the official regulation document [7].

Even though regularisation is a big step in the right direction, it will not completely solve the above issues. Users are now made more aware of what is done with their data but all gathered and uploaded data is still property of the company. On top of that, if you decide not to allow the data gathering or if you do not agree with some part of the processing that will be done with your data, it is possible that you can not use the application. Suppose you want to use a social media website for sharing and talking to people but the terms and conditions also include that your pictures get analysed for marketing purposes and are property of the company once they are uploaded. If you agree with everything but the analysis of your pictures and the change of ownership, it is possible that you can not use the social media site even though you agree with everything else.

Therefore, the best way of solving privacy and data ownership issues is to decouple the data from the applications. In a social media application, this would for instance mean that users would be able to chat and share pictures with each other without having to upload their data to the company. Thus effectively not having to give up the ownership of their own data and media. This is the main concept of a so-called “distributed social network” which is described by Sebastian Tramp [8]. In this distributed network, everyone is the sole owner of his or her information, has full control over what is shared with who, and allows for a more computer interpretable data representation compared to the current centralised solutions.

Privacy is thus a big driving force for creating such a distributed network based on the Semantic Web but there are some other benefits to this approach. Since users are hosting the information themselves, every computer becomes a small server. Companies would thus not need large server farms for hosting all the data of their users. Only computational and functional parts of the application would have to be run on the servers which would cut down electricity use and costs for the companies. Users could also remove and add information and media at will without having to follow long and elaborate procedures to remove their data from the company servers. The Semantic Web would also allow users to link their photos or videos to certain topics or people which can then be accessed by following the link, thus providing the same functionality as tagging in Facebook.

One step of creating such a distributed social network is a versatile and scalable media hosting solution as uploading and sharing pictures and videos is a huge part of current social media platforms. The versatility of such a media hosting solution comes from the expressiveness of the searches one can perform on the vast amount of media that is hosted on these (distributed) social media sites. For example, it would be nice to be able to search for pictures where your father is standing to the left of your mother or where your dog is sitting to the right of your cat. As already shown, these types of search queries have little
to no support in the current centralised social media applications and thus leave an opportunity for the distributed social network to outshine the current solutions when it comes to media consumption and ease of use.

SPARQL-MM currently offers a way to enable such detailed and expressive search queries over media using basic Semantic Web technologies. There is however one big problem with this solution. SPARQL-MM is currently only implemented in such a way that it uses a single server to evaluate the queries of all clients that connect to it simultaneously. This means that in order to accommodate all the simultaneous clients, some very powerful servers would need to be used as the expressive queries require more computing power than normal queries. Given that social media sites have an enormous amount of users (see graph 1.2), incorporating this technology in a (distributed) social media platform would be very costly.

Even though the ultimate goal is to use SPARQL-MM like functionality in a completely distributed social network, a lot of research still has to be done before such a network can be achieved. This does however not mean that the functionality and performance aspects of SPARQL-MM are irrelevant for current centralised systems. Current social media or media hosting systems could use SPARQL-MM as a backend query engine to provide the users with more powerful search queries. This however introduces the same downsides as explained in the previous paragraph. If we could make it so that less powerful servers are needed to accommodate the same amount of users, companies would gain some benefits. This either by being able to server more clients with the same server infrastructure and a less frequent need for additional server capacity (which diminishes costs over a long period) or being able to use less powerful servers which use less electricity while still being able to serve the same amount of users. There is thus a big financial and ecological benefit for the company if we have more scalable solutions that can still provide the same functionality.

In this thesis, the scalability issues of SPARQL-MM are investigated. This in function of how many simultaneous clients can be served at the same time without rejecting any one of them. A new solution will also be proposed in order to resolve these scalability issues without compromising in the power and expressiveness of the search queries.
CHAPTER 2

The Semantic Web

2.1 General view of the Semantic Web

In 1989, Tim Berners Lee proposed a system to create an information management system based on hypertext, known as the World Wide Web. This system has been adopted worldwide and has become essential in day-to-day tasks. For example, if you want to find out how to make a certain meal, you use Google to find a web page with the recipe. If you want to find information for your current science project in High school, you can use websites like Wikipedia and ResearchGate to find information about the topics of your interest. If you are planning a vacation you can use the web to find hotels and flight tickets and book them instantaneous. Even though the World Wide Web has become vital in our day-to-day activities, there are some problems with the concept. Let us take a look back at the example of booking a vacation, but more specifically, booking a hotel. When booking a hotel, most people want the cheapest possible price for a room. You can find this by accessing the web pages of all the hotel booking companies or travel agencies and see how much the same room costs for each and one of them. You thus have to access a lot of different websites, each with a different style, purpose and possibly different language. Then, you have to mentally integrate the information of all the different sites yourself to achieve your goal of finding the cheapest possible price of your room. It is easy to see that this can be a long and tedious process. Even though there currently exists sites that will aggregate the pricing of different hotels in one place like Booking.com for example, they aren’t perfect. Current aggregation sites can only aggregate a static subset of hotel sites or travel agency sites. The site will scrape information about pricing and availability but only for the hotels that are programmed to be scraped. This selection is mostly determined by contractual agreements between the hotels or travel agencies and the aggregation sites. Next to this, hotels can pay the aggregation sites to promote their hotel more than others, thus also creating an unfair competition. We thus only get a limited set of all available information and therefore can miss out on better solutions. Next to this, the web pages that you can see in your browser only show the tip of the iceberg. There is no immediate access to the data itself but only to a textual representation that is created by the web developer of the company. The real data is mainly stored in databases but only a part is represented in the HTML (HyperText Markup Language) files that are shown in your web browser.

It would therefore be much better if we could access the data directly in the same way as we do with web pages today. This means, being able to link data to each other just like we can create links to other web
pages while adding some extra classification information about the data and links, access the data directly with a unique identifier like an URL (Uniform Resource Locator) of a web page and most importantly, enable machines and programs to interpret the data in order to create more intelligent user agents which can greatly reduce the execution time and tediousness of certain tasks. These are the building blocks of a so called “web of data” and it is also the vision that Tim Berners-Lee expressed when defining the Semantic Web roadmap \[11\]. In this document, he wrote the following argument for the need of a Semantic Web:

"The Web was designed as an information space, with the goal that it should be useful not only for human-human communication, but also that machines would be able to participate and help. One of the major obstacles to this has been the fact that most information on the Web is designed for human consumption, and even if it was derived from a database with well defined meanings (in at least some terms) for its columns, that the structure of the data is not evident to a robot browsing the web. Leaving aside the artificial intelligence problem of training machines to behave like people, the Semantic Web approach instead develops languages for expressing information in a machine processable form."

However, in order to make this web of data universally useable, all these elements and technologies need to be standardised. This is what Linked Data \[12\] was created for. It defines a collection of standard technologies to use in order to create a web of data which is machine interpretable, thus in effect creating a global database where the user agents can navigate over and represent data in such a way that they are able to reason about the information this data represents.

### 2.2 Main principles of the Semantic Web

As written in section \[2.1\] Linked Data standardises technologies in order to enable a web of data which is also known as the Semantic Web. This does not mean that it is intended to completely replace the World Wide Web. On the contrary, it is designed to create an extra layer to the current World Wide Web which will allow machines to interpret the information much better than they currently can. In order to provide this compatibility, Linked Data, or more general, the Semantic Web still works on the same concepts as the current World Wide Web albeit with some minor changes and additions. These are all gathered in six basic principles \[13\] on which the Semantic Web is built.

#### 2.2.1 Principle 1: Everything can be identified by URIs

In the Semantic Web, every thing that exists in the physical world like people, places, objects, etc. can be referred to with identifiers. For example, if one would like to be able to identify a certain city like Ghent, you can refer to the URI of a page which contains information about the city. Figure \[2.1\] shows
CHAPTER 2. THE SEMANTIC WEB

a graphical representation of this concept. This is a very rudimentary way of linking an entity or concept to an object. Later on, technologies will be introduced which enable us to define a more specific relation between the real city and the URI. For example, we would be able to say that http://www.stad.gent is the official homepage of the city in a machine interpretable way. We can also indirectly refer to physical resources, as can also be seen in figure 2.1. Here, the person called “Marja” is identified by the URI of her mailbox. Once we have this identifier, we can refer to her. Let us for example take a look at the following sentence: “The person whose email address is mailto:marja@example.org works at W3C”. Indeed, once we have the mail address of this person, we can uniquely identify her and talk about certain properties. Next to identifying things, we can also identify concepts like properties and relationships in the Semantic Web using URIs. This is what ontologies are used for. The friend of a friend (FOAF) ontology for example defines different concepts regarding people by defining different URIs for each concept. http://xmlns.com/foaf/0.1/knows for example identifies the concept of “knowing someone”.

Figure 2.1: Example of URIs used as identifiers for physical resources

2.2.2 Principle 2: Resources and links can have types

In the way the current Web is defined, there are 2 types of components: resources and links. The resources are the HTML documents which are targeted for humans and very rarely exhibit metadata explaining what they contain, what they are used for or what their relation is to other HTML documents on the web. This lack of information does not pose a big problem for humans as we are very good at interpreting context. For example, it is very clear for humans that the left page in figure 2.2 contains an invoice while the right page contains a news article. Machines however do not have the capability to do this. Therefore, we need specific typing saying that the left page is an invoice and the right page is a news article. This way, machines can act differently depending on which type of document they are currently viewing. Next to being able to easily detect the type of documents, humans also are very good at noticing the relation between different resources. A person can for instance check the text around a certain link to understand the context in which this link is used. This will then enable him to determine the relation between the current web page and the web page identified by the link. This is also something that computers are unable to do. Therefore, links should also be typed and define more precisely what the relation between the current resource and the linked resource is. Figure 2.3 shows an example of how the Semantic Web adds types to resources.
and links. The links for example indicate that the program generates documentation which includes a user manual and api documentation.

Figure 2.2: Example of human interpretable web pages

![Resource flowchart](image)

Figure 2.3: Resources and links as used in Web versus Semantic Web implementation

### 2.2.3 Principle 3: Partial information

In the Web, links are unidirectional. This means that if resource A has a link to resource B, there is no automatic reference added from resource B to resource A. Resource B doesn’t even know that there is a link from resource A to itself which can cause some problems. Suppose for example that resource B has moved. Since it has no notion of the fact that resource A links to him, it cannot let resource A know that it has moved. If one then wants to follow the link from resource A to resource B, he will stumble on to a 404 error which indicates that the resource with the given URI is no longer found. The World Wide Web does however allow this behaviour in order to accommodate a great scalability. Just like the World Wide Web allows for unidirectional links to be used by anyone, so does the Semantic Web. Anyone can say anything about any resource and can create different types of links between different resources. These will again be unidirectional. Resources on the Semantic Web can also be moved or removed and thus Semantic Web tools need to be able to cope with this behaviour.
2.2.4 Principle 4: Freedom of data

Not everything on the Web is true and people are entitled to their opinion. This is also the case for the Semantic Web. There is no need for absolute truth when you publish data. The applications that consume data on the Semantic Web have to decide which data they will trust and which they will omit.

2.2.5 Principle 5: Support for evolution

Principle three and four indicated that everyone is able to publish data on the Semantic Web. This means that there is a possibility that two or more different people have published data about the same concept albeit on different locations. Say for example that Alice, who is curator of the Louvre in Paris, has published some data about the Mona Lisa and identified this resource with the URI \texttt{https://www.example.org/Mona_lisa}. Bob, who is an art enthusiast, has also published data about the Mona Lisa and identified it with the URI \texttt{http://www.example.org/MonaLisa}. Two different people are thus talking about the same concept but both are using a different URI to identify it. On top of that, as Alice is a curator of the Louvre, she has more more information about the subject and thus it would be helpful if Bob could indicate that he is talking about the same thing as Alice and that more information can be found at her URI. This is why "seeAlso" links are defined in the Semantic Web. They indicate that a certain URI also contains information about the same topic. Bob could thus create a "seeAlso" link to the URI of Alice which will tell user agents that the same (or even more) information about this concept can be found by following the URI \texttt{https://www.example.org/Mona_lisa}.

Next to having two different locations for the same concept, the same concept could also be defined in different moments of time. Let us therefore take a look at figure 2.4 to illustrate the support for evolution in time. Suppose John Doe was working for IBM until 2016, after which he decided to quit and join W3C. When he worked at IBM, a URI was created to identify John Doe and some information about him was defined using a certain ontology. However, since he now is working at W3C, a new URI was created and a new ontology was used in order to comply to the standards of his new employer. Since the old URI is still available, a link can be created between the new URI of John Doe and the old one telling that the previous identity of the current resource, being John Doe, can be found at URI 3. We can also see that previously the property “employedBy” was used where with the current ontology, “worksAt” is defined. The Semantic Web therefore allows users to define a transformation from one ontology to another. The new employer can thus define a transformation in which is defined that “worksAt” and “employedBy” describe the same concept.
2.2.6 Principle 6: Minimalistic design

As said in the Semantic Web Activity[13] article: “The Semantic Web makes the simple things simple, and the complex things possible”. This means that there will be as little as possible standardisation in order to keep the technology as open and simple as can be. It will only standardise the components needed for interoperability and scalability just like the World Wide Web.

2.3 The Semantic web stack and technologies

By now we know that the Semantic Web has its foundation in six basic principles and aims to create a web of data which is an extension of the current World Wide Web. In order to do so, W3C has created a layered architecture which defines the different components for such a web of data. Each layer in the stack uses the capabilities of the layer below, thus gradually building a complex system. This architecture can be found in figure 2.5. All layers up until OWL (Web Ontology Language) are currently standardised while there is still uncertainty about how the top of the stack is going to be standardised. In the following sections, the most important layers of the architecture will be explained in detail.
2.3.1 URI and Character Set: Unicode

This layer contains the same standard technologies as the Web and accommodates the first principle of the Semantic Web, namely being able to uniquely identify every possible resource. URIs are short strings that identify a resource on the Web and the syntax is standardised in RFC (Request For Comments) 3986 in January 2005. The syntax is defined as follows:

```
scheme: //[user[:password]@[host[:port]][/path]?[query][#fragment]
```

Each URI has a scheme which defines what protocol is used to transfer the resource you want to access. It is followed by a colon (:) and two slashes (//). Next, there is an authority section which exists of the following elements:

- An optional authentication section which contains a username and password, separated by a colon and followed with an at (@) symbol
- A mandatory host section which is either a registered name like W3C.org or an IP address.
- An optional port number, separated from the host with a colon.

Following the authority section is the path specifier. This specifier tells where a certain piece of data can be found. The data is mostly organised in an hierarchical fashion which is indicated by the use of
slashes (/). The scheme, authority section and path specifier are mandatory for each URI. Next to these mandatory elements, there are also some optional elements which can be used to further specify what specific part of the resource one wants to retrieve. The first optional element is the **query**. A query follows the path specifier and starts with a question mark followed by a query string. Even though the syntax of a query is not strictly defined, a convention is used where a query string exists of a sequence of attribute-value pairs separated by a delimiter which is mostly an ampersand (&). Another optional element is the **fragment identifier**. This element starts with a number sign (#) followed by an identifier. This identifier will indicate a certain part of the resource that is identified by the path. Section 3.1 will go more in depth about fragment identifiers and their respective standard specifications. An example of a URI can be found in figure 2.6.

![Figure 2.6: An example of a URI, Figure from Wikipedia](image)

As one may have noticed, URIs are all defined in a textual, user readable format. In order to display and write these textual URIs, Unicode is used. It is a standard for the consistent encoding, representation and handling of text expressed in most of the world’s computers. The standard was first defined in January 1991 [17] and has been expanded ever since. It defines encoding methods and a set of standard character encodings so that every computer system can correctly process and display the entered characters.

Using the combination of URIs and Unicode, a user can thus uniquely identify a resource on the (Semantic) Web in such a way that every system can interpret the identifier. This is the most basic and important part of the (Semantic) Web.

### 2.3.2 Resource Description Framework (RDF)

**General concept of RDF**

In order to create a Semantic Web, we need to be able to represent information in such a way that machines can interpret and in some extent reason about the information that is given. This is what the Resource Description Framework [18] was created for. The main idea of the framework is as follows: each single bit of information about a resource is represented as a triple, consisting of a subject, a predicate and an object. The **subject** denotes the resource for which we want to express something, the **predicate** denotes a certain property about the resource and expresses a binary relationship between the resources that are denoted by the **subject** and the **object**. If we for example want to express the following statement “Grass is green” in RDF, we can do this by creating a triple with a subject denoting “grass”, a predicate
denoting "has the colour" and an object denoting the colour "green". We can visually represent this statement by using a graph where the node "grass" is connected to the node "green" with a directed arc which denotes "has the colour". This can be seen in figure 2.7. We can now create different statements about grass which each is represented by a triple. For example, "Grass is a plant" can be represented by the triple (grass, hasType, plant) or "Grass contains water" can be represented by the triple (grass, contains, water). These examples all have grass defined as their subject. So more specifically, we define and talk about properties of grass itself. However, we can also go the other way around and tell that grass is part of something else in which case "grass" will be an object of a triple. For example, we could say that a lawn contains grass. We can represent this with the triple (lawn, contains, grass). If we now combine all these triples and visualise them, we end up with a directed graph consisting of different nodes which represent a certain resource or piece of information and arches which define the relation between the two nodes it connects. This is why a set of triples is called an RDF graph. An example for the earlier statements can be found in figure 2.8. In the previous example we talked about "grass", "water", "plant", "lawn" and "green" as concepts in order to denote something in the real world. These are all categorised as resources in RDF. In figure 2.8, most object nodes were represented by a string. This is called a literal value in the Resource Description Framework and each literal has a certain datatype. In our example each literal was of the type "string". Other examples of literal types are numbers, dates etc.

Figure 2.7: Visual representation of the notion "Grass is green"

Literals are important in the Resource Description Framework as it is a way of representing an object that does not have a URI. The RDF framework does however not allow literals to be used as a predicate or subject. Next to literals, the Resource Description Framework also allows objects to be represented by URIs, just like subjects and predicates. This makes it possible to uniquely identify the resource you are talking about together with the relationship between the two resources. For example, instead of talking about "grass" as a string, we define grass as a resource on the web. This implies defining a URI like http://www.example.org/grass. If we now evaluate our RDF graph, we know which specific resource we are talking about and we can even dereference the URI to get more information about this resource, provided that the resource still exists. Similarly, I might have created a URI about my front lawn http://www.example.org/myLawn and want to express that my specific front lawn has grass. Therefore, instead of using the literal "lawn", the URI http://www.example.org/myLawn is used as value for that node. Subjects, predicates and objects can thus be represented by URIs while only objects can also be represented by literals. There is however another possibility to represent subjects and objects which are called blank nodes. Unlike URIs and literals, blank nodes do not identify a specific resource. Statements using a blank node say that something with the given relationship exists without explicitly naming it. For example, in figure 2.9 a blank node is used to express that both Alice and Bob have a brother but without revealing
who this is. A summation of all possible values for the subject, predicate and object of a triple can be found in the following list:

- subject: URI or blank node
- predicate: URI
- object: URI, blank node or literal

Using these URI and blank nodes allows anyone to create links to other resources and thus creating a huge interconnected graph of data. This is also why the Semantic Web is sometimes called a Giant Global Graph.

By using this simple data model and the ability for the model to represent abstract concepts which can even be real-world things, it is also a very powerful and important part in knowledge management systems that are unrelated to the Semantic Web. In conclusion, RDF can be seen as a natural extension on the linking structure of the Web. Besides telling you you can can go from resource A to resource B, RDF provides extra functionality. By using predicates, the relationship of the two resources connected by it can be uniquely identified as the predicate is required to be defined by a URI. This enables the user agents to extract extra information about the link between the two resources.
RDF documents and syntaxes

RDF is mostly a high-level concept. It only defines that information about resources should be represented in form of triples by using literals, URI and blank nodes but it does not impose any form of syntax on how these triples should be defined in documents. Therefore various serialisation formats, or so called syntaxes, have been developed. Each with their own representation of resources and triples. Some of the most popular RDF syntaxes are Turtle [19], RDFa [20], JSON-LD [21] and RDF/XML [22], which is the first one ever created.

One thing that all syntaxes, except for N-Triples, have in common is the use of prefixes which enable for a more simple and human readable document to be constructed. In figure 2.9 we can see the following value for both predicates: http://www.example.org/:hasBrother. When we have to use this predicate very often in our dataset, the document defining our RDF graph will become unreadable. Therefore, we can use the following textual notation to express the same predicate ex:hasBrother. This is a shortened notation of the full URI where ex: is called the prefix. A prefix is always defined in the beginning of an RDF document and defines a certain base URI. The RDF enabled program will then automatically construct the full URI from the prefix when it encounters one. We can for example define the following prefix in the beginning of an RDF document:

    ex: http://www.example.org/

When the RDF enabled program now encounters the predicate ex:hasBrother, it will automatically construct the full URI by appending the string behind the colon to the base URI. The resulting URI will thus be http://www.example.org/hasBrother. Note that there are no rules requiring the use of prefixes. They do however are used very often in order to create more readable documents.

2.3.3 SPARQL

SPARQL is an RDF query language. This means that it is a semantic query language for RDF-based data. It is able to retrieve and manipulate data stored in RDF format. The fact that users can write queries against RDF data makes it very useful for Semantic Web applications. It provides a syntax that is similar to SQL, is very extensible and does for RDF data what SQL does for relational databases. In chapter 4.1 the syntax and main concepts of SPARQL will be explained in more detail.
CHAPTER 3

Media Fragments URI

3.1 The Media Fragments URI standard

In section 2.3.1 it was shown how URI can be used to identify general resources on the (Semantic) Web. These resources often take the form of a structured textual document like HTML or RDF documents. They are widely used and can easily be read and interpreted by different user agents. There are however some other resources on the web which are not textual but binary. These are called media resources. Media resources like audio and video have always been treated as “foreign” objects on the World Wide Web. They could only be accessed using a plugin that is capable of decoding and interacting with the media source like Adobe Flash Player, Microsoft Silverlight etc. This problem was recognised by W3C and resolved with the newest iteration of the HTML standard, called HTML 5 \[23\]. In this version, video and audio elements were added to the syntax which allows Web browsers that support the HTML 5 syntax and have a built in video/audio player to display the media resource that is linked in this element. This is done by means of a src attribute, thus removing the need for third party plugins and incorporating these media objects in the Web stack.

Even though this solves one problem, there is also another issue when dealing with media on the Web. By using the standard URI syntax, one could only indicate a media resource like a video or picture as a whole. However, there are some use-cases where you do not want to retrieve or display the whole resource but just a small part. Suppose you have a video of the latest news report from CNN hosted on your server and identified it with the URI \[http://www.example.org/news_report\]. This video contains the entire broadcast of the news which discusses various topics from national and international news to the latest sports results and the weather, causing the video to be 30 minutes long. If you would want to only see the weather, you would have to access \[http://www.example.org/news_report\], download the whole video and scrub through the video manually in order to get to the weather at the end. This is an inconvenient way of doing things and causes wasted bandwidth as you have to download the whole video to only see a part of it. This is what Media Fragments URIs are used for. They provide a standardised way of defining parts of media resources like time offsets, spatial offsets etc. on the Web using URIs.

When talking about Media Fragments URI, we have three main concepts that need to be defined:
• URI fragment: This component of a URI is started by a number sign (#) and continues until the end of the URI

• URI query: The query component is indicated by the first question mark (?) and terminated by a number sign (#) character or by the end of the URI

• Media fragment URI: a URI addressing part of a media resource which could be using URI queries or URI fragments. Listing 3.1 shows a schematic of such a URI.

Listing 3.1: Schematic representation of a URI

```
<scheme name> : <hierarchical part> [ ? <query> ] [ # <fragment> ]
```

3.1.1 URI fragments vs URI queries

The definition of a Media Fragments URI indicates that both URI fragments and URI queries can be used in order to address parts of media resources. There is however a difference between both methods when it comes to resolving the URI. When resolving a URI, the URI query will produce a new resource and the URI fragment will produce a secondary resource that has a relationship to the primary resource. This is due to the way the media fragment is extracted and in which location this is done.

URI fragments are resolved from the primary media resource without an additional retrieval action. The extraction of the secondary resource is thus done at the user agent. This also means that the user agent is able to resolve a URI fragment on a resource it has already received without having to request extra data from the server. A benefit of this approach is that when multiple fragments from a same resource are needed, the primary resource only needs to be downloaded once and all the fragments can be extracted at the user agent, thus lowering the bandwidth usage. This specification also introduces a constraint: the media types of the extracted fragment should be the same as the media type of the primary resource. So when a media fragment points to a single frame of a video, the type of the resulting resource will be a video containing one frame, not a still image. Since URI fragments are resolved by the user agent, not every media resource is compatible. There are three categories of media resources in relation to fragment URIs:

• “Fit” media resources: A media fragment can be extracted from these resources by the user agent without having to modify the syntax elements or having to transcode the bitstream.

• “Conditionally fit” media resources: A media fragment can be extracted from these resources but an additional retrieval is required to retrieve the modified syntax elements of the resource.

• “Unfit” media resources: These media resources can not be addressed with a URI fragment as they require transcoding.
When using fragment URI to address Media Fragments, the user thus has to know whether the media fragment can be produced from the primary media resource without transcoding activities. If this is not the case, the user must use URI queries instead of URI fragments.

In contrast with URI fragments, URI queries are resolved at the server and will produce a completely new resource. This brings some added complications when displaying the media fragment. Take for example the following URI: `http://www.example.org/video.ogv?t=50,60`. Here, a URI query is used to indicate that only a video of 10 seconds is needed. The server will thus create a new resource extracting the correct frames from the original video file, called `video.ogv`, and return this media fragment. We thus have a completely new resource at the client which starts at time index 0 and is 10 seconds long. The context of the original source is therefore completely lost and this can be a problem. A video player in a web browser might want to indicate that the media fragment starts at the 50 second mark of the original resource instead of having the time indicator start at time index 0. In order to facilitate this, there are two possible mechanisms. Either the media fragment itself contains some extra information defining that it is an extract from another primary resource starting at a certain offset (Ogg files are an example of such media files) or the user agent is told through the retrieval action which primary resource the retrieved media fragment is related to. The user agent then has to perform an extra retrieval in order to gather information about the original resource.

It is also possible to combine both query URI and fragment URI. When doing so, a combination of the two is performed. The query URI will instruct the server to return a completely new resource which is derived from the primary resource indicated by the hierarchical part of the full URI and the user agent will then resolve the fragment URI on top of the newly created resource. For example, the URI `http://www.example.org/video.ogv?t=50,120#t=30` will result to a 30 second fragment offset applied to the new media resource starting at 50 going to 120. The reply of this URI is thus a media resource that is 70 seconds long and whose playback starts at an offset of 30 seconds.

### 3.2 Media fragments Syntax

Since Media Fragments can be identified either by URI fragments, URI queries or a combination of fragments and queries, both are standardized by the Media Fragment specification [24]. The authors have chosen to make the syntax identical for both of them, effectively making the syntax more simple to use and more unified. Media fragments support addressing the media along four different dimensions: temporal, spatial, track and id. The temporal dimension denotes a specific time range in the original media like “starting at second s and stopping at second t” while the spatial dimension denotes a certain range of pixels from the original media such as “a rectangle with size (w,h) and top left coordinate (x,y)”. The last two dimensions are only defined in the advanced version of the Media Fragment specification and are not able to be resolved by clients implementing the non-advanced Media Fragment specification. The track dimension indicates which specific track is denoted like “the English spoken track or the French spoken
track of a video”. Note that not all media types define tracks. The track dimension can thus only be used on a limited set of media files. The last dimension is the id dimension. This one denotes a named temporal fragment within the original media like “chapter 2”. Again, only a limited set of media standards support this functionality. All dimensions are logically independent and can be combined freely. In what follows, the syntax of the temporal and spatial dimension is explained in more detail. Because of the limited support, the latter two dimensions will not be elaborated upon in this chapter. More information about them can be found in the Media Fragments 1.0 specification [24].

3.2.1 Temporal dimension

The first dimension that can be addressed by Media Fragments is the temporal dimension. Addressing this dimension is also known as “temporal clipping” as extracting a temporal fragment from a video or audio file can be seen as creating a clip from the original file. The temporal clipping is denoted by the name \( t \) in the media fragment identifier, followed by a start time and end time. Either one or both time indicators are allowed to be omitted, with the default start time set to 0 and default end time set to the duration of the original media. When a time indicator is omitted, the default value will automatically be filled in by the user agent or server. It is important to notice that the time interval is set to be half open. This means that the start time is considered to be part of the extracted media fragment while the end time indicates the first time point that is not part of the fragment anymore. Some examples of temporal identifiers can be found in listing 3.2. Note that the same syntax is used for both URI fragments and URI queries.

\[
\text{http://www.example.org/video.ogv#t=10,30:} \\
\Rightarrow \text{Results in the time interval } [10, 30[ \\
\text{http://www.example.org/video.ogv?t=,15:} \\
\Rightarrow \text{Results in the time interval } [0, 15[ \\
\text{http://www.example.org/video.ogv#t=60:} \\
\Rightarrow \text{Results in the time interval } [60, \text{end}[ \\
\]

Listing 3.2: Example of valid temporal identifiers according the the Media Fragments URI specification

The Media Fragments URI specification has also defined some rules that indicate which (combination of) time indicators are valid fragment identifiers. In order to define the rules, the following notation is introduced:

- \( s \): The start point of the source media. This is always zero.
- \( e \): the end point of the source media. This will always be larger than zero.
- \( a \): a positive integer, \( a \geq 0 \)
- \( b \): a positive integer, \( b \geq 0 \)
Using these parameters, a list of rules are defined which dictate the behaviour when the Media Fragments URI is resolved. These rules can be found in the following list, created by W3C[24]:

- \( t=a \) with \( a < e \): media playback start at time \( a \) and stops at end \( e \).
- \( t=a \) with \( b \leq e \): media is played from start to time \( b \).
- \( t=a \) with \( e < b \): whole media is played.
- \( t=a,b \) with \( a = 0, b = e \): whole media is played.
- \( t=a,b \) with \( a < b, a < e \) and \( e \leq b \): media is played from time \( a \) to time \( b \).
- \( t=a,b \) with \( a < b, a < e \) and \( e < b \): media is played from \( a \) to end.

Note that the Media Fragments URI \texttt{http://www.example.org/video.ogv?t=5,5} is not a valid syntax to indicate a single frame in the video. This because time intervals are defined to be half open. If we would resolve the URI, we would end up with the following impossible interval \([5, 5]\). Therefore, the end time always needs to be larger than the start time.

### 3.2.2 Spatial dimension

Spatial clipping selects a range of pixels in the original visual media source like videos or images. The current version of the Media Fragment specification only allows for rectangular selections to be done. The rectangle can be identified both by real pixel coordinates or as percentual values. The rectangular selection box is denoted by the name \texttt{xywh} and the value contains an optional format identifier which can be \texttt{pixel:} or \texttt{percent:}. If no format is defined, the pixel format will be used as default. After the format identifier follows four comma-separated integers. These integers denote the \( x \) and \( y \) position of the upper left corner of the rectangle with \( x = 0 \) and \( y = 0 \) being the upper left corner of the image, followed by the width and the height of the rectangle. If percent is used as format, \( x \) and width are interpreted as a percentage of the width of the source image. \( y \) and height will similarly be interpreted as a percentage of the height of the source image. Some examples of valid spatial identifiers can be found in listing 3.3.

\begin{verbatim}
http://www.example.org/image.jpg#xywh=160,145,30,45:
\rightarrow Results in a 30x45 box at x=160 and y=145

http://www.example.org/image.jpg?xywh=pixel:12,12,320,240:
\rightarrow Results in a 320x240 box at x=12 and y=12

http://www.example.org/image.jpg#xywh=percent:25,25,50,50:
\rightarrow Results in a 50%x50% box at x=25% and y=25%
\end{verbatim}

Listing 3.3: Example of valid temporal identifiers according the the Media Fragments URI specification
CHAPTER 3. MEDIA FRAGMENTS URI

Just like the temporal dimension, some rules are defined indicating which values are valid when identifying spatial fragments. These are different based on whether the pixel or percent format is used. Again, a notation is introduced in order to define the rules:

- a: the x-coordinate of the spatial region, \( a \geq 0 \)
- b: the y coordinate of the spatial region, \( b \geq 0 \)
- c: the width of the spatial region, \( c > 0 \)
- d: the height of the spatial region, \( d > 0 \)
- w: the width of the source media, \( w > 0 \)
- h: the height of the source media, \( h > 0 \)

Using this notation, the rules are defined as follows:

- \( \text{xywh}=a,b,c,d: a+c \leq w \) and \( b+d \leq h \): defines spatial fragment with coordinates (xywh format) \( x = a, y = b, w = c, h = d \).
- \( \text{xywh}=a,b,c,d: a+c > w, a < w, \) and \( b+d < h \): defines spatial fragment with coordinates (xywh format) \( x = a, y = b, w = w-a, h = d \).
- \( \text{xywh}=a,b,c,d \) with \( a+c < w, b+d > h, \) and \( b < h \): defines spatial fragment with coordinates (xywh format) \( x = a, y = b, w = c, h = h-d \).
- \( \text{xywh}=a,b,c,d \) with \( a+c > w, a < w, b+d > h, \) and \( b < h \): defines spatial fragment with coordinates (xywh format) \( x = a, y = b, w = c, h = h-d \).
- \( \text{xywh}=\text{pixel}:a,b,c,d \) with \( a+c \leq 100 \), \( b+d \leq 100 \): defines spatial fragment with coordinates (xywh format) \( x = \text{floor}(a/w*100), y = \text{floor}(b/h*100), w = \text{ceil}(c/w*100), h = \text{ceil}(d/h*100) \) (the normal pixel case).
- \( \text{xywh}=\text{percent}:a,b,c,d \) with \( a+c \leq 100 \), \( b+d \leq 100 \): defines spatial fragment with coordinates (xywh format) \( x = \text{floor}(a/w*100), y = \text{floor}(b/h*100), w = \text{ceil}(c/w*100), h = \text{ceil}(d/h*100) \) (the normal percent case).

3.2.3 Combining multiple identifiers

Previous sections defined how one can identify a temporal or spatial fragment of a source media file. However, one could also want to denote a certain pixel area of a small time frame of a video. In order to do this, temporal and spatial identifiers can be combined into one URI by appending them to each other with an ampersand (&) character when both are defined in a URI fragment. As already explained in chapter 3.1.1, one could also combine URI fragments and URI queries with each of them defining a different dimension. Some examples can be found in listing 3.4.
http://www.example.org/video.mp4#t=50,61&xywh=160,145,30,45:

=> Results in a 30x45 box at x=160 and y=145 in a 10 second clip starting at time 50

http://www.example.org/video.mp4?t=200,300&xywh=pixel:12,12,320,240:

=> The query will create a new resource which is 100 seconds long and starts at time 200 of the original source. In this newly created source, a 320x240 box is selected at position x = 12, y = 12

http://www.example.org/video.mp4?t=,30&xywh=percent:10,10,25,40:

=> Results in a new source created by the server which is a 30 second clip starting at the beginning of the source media file and has a 25% x 40% box at position x = 10%, y = 10%

Listing 3.4: Example of combined temporal and spatial fragment identifiers
CHAPTER 4

SPARQL and SPARQL-MM

4.1 SPARQL and the Semantic Web

As already explained in section 2.3.3, SPARQL is a query language which enables users to search for information on the Semantic Web. It thus provides the same functionality as traditional search engines on the Web albeit in a fundamentally different way. In the traditional Web, all data is stored in Web pages and in textual format which means that machines need to scrap these textual documents and do string matching to find the right information. In the Semantic Web however, all data is stored in RDF format which is a directed and labeled graph format for data. The data on the Semantic Web can thus be interpreted and reasoned with by machines. So instead of having to scrap textual documents, SPARQL will do pattern matching on directed graphs and this way extract the information needed by the end user. Note that there is one exception, being RDFa. This RDF standard incorporates RDF data in HTML documents and thus still requires scraping before we can query its information. In what follows, the syntax and capabilities of SPARQL are explained, followed by a short introduction in graph patterns and graph pattern matching in order to fully understand the working of SPARQL on RDF data. In the last section, the added functionality of SPARQL-MM is explained.

4.2 SPARQL syntax

In order to fully explain the syntax of SPARQL, we will first go over some simple examples and their results which are also shown and explained in the SPARQL Query Language for RDF recommendation by W3C [25].

4.2.1 Some basic examples

Finding the name of a person

The first query can be found in example query 4.1 in which the data in the RDF graph, SPARQL query and result are shown. The RDF graph contains only one triple, stating that the employee that is identi-
Data stored in RDF graph

```
<http://example.org/employees/employee1> <http://xmlns.com/foaf/0.1/name> "John".
```

Example query 4.1: Finding name of employee

`SELECT ?name
WHERE {
}
```

The query will try to find the name of an employee from the given RDF graph and consists of two parts: the `SELECT` clause identifies which variables need to appear in the query results, and the `WHERE:` clause indicates the start of the Basic Graph Pattern which needs to be matched against the provided RDF graph. A Basic Graph Pattern is simply a collection of triple patterns indicating which variables need to be bound in order to find a solution. The Basic Graph Pattern in this example consists of one triple pattern with a single variable (`?name`) as the object of the pattern. This query will thus effectively search for triples in the RDF graph where the subject equals `http://example.org/employees/employee1` and the predicate is equal to `http://xmlns.com/foaf/0.1/name`. The query did not define a URI, blank node or literal but a variable (indicated by the preceding question mark) as object for the triple pattern. These variables do not have a preset value and are dynamically filled in by the SPARQL engine while executing the query. If this variable is also defined in the `SELECT` clause, the dynamically bound value will be returned as a result. So in this case, the RDF graph will return the triple `<http://example.org/employees/employee1> <http://xmlns.com/foaf/0.1/name> "John"`. The SPARQL engine will then bind the literal "John" to the variable `?name` and return this value.

Getting multiple matches

Example query 4.2 has an RDF graph with more than one triple and indicates how a SPARQL query can have multiple matches. In this example, there are three blank nodes `_:a`, `_:b` and `_:c`. `_:a` and `_:b` both have a name and an age, while `_:c` only has an age and no name defined in the graph. The query will return the values bound to the variables `?name` and `?age`. However, the Basic Graph Pattern also has an additional variable `?x` which will not be returned as it is not defined in the `SELECT` statement. The Basic Graph Pattern is also a bit more elaborate compared to the previous example. We have a subject variable `?x` which has a name and an age. The SPARQL query engine will now try to find all possible bindings for the variables `?x`, `?name` and `?age` so that the query pattern matches the data. The result set will then give all possible solutions for the variables. In the example, two matches were found. Notice that
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "John" .
_:a foaf:age 43 .
_:b foaf:name "Richard" .
_:b foaf:age 20 .
_:c foaf:age 21 .

Data stored in RDF graph

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;John&quot;</td>
<td>43</td>
</tr>
<tr>
<td>&quot;Richard&quot;</td>
<td>20</td>
</tr>
</tbody>
</table>

Query

SELECT ?name ?age
WHERE {
  ?x foaf:name ?name .
}

Result

Example query 4.2: Multiple matches

the age of blank node _:c was not returned. This because the blank node did not have a name defined in the RDF graph and thus did not match all triple patterns in the Basic Graph Pattern of the query. This is a very important notion. In order to have a Basic Graph Pattern match, all variables in the Basic Graph Pattern need to be matched. Another remark that can be made is that, like the RDF syntax, the SPARQL syntax also allows users to define prefixes in order to improve readability.

Matching Literals

As already explained in chapter 2, RDF allows for literals, blank nodes and URI to be defined in a triple. Therefore, SPARQL also needs support for matching literals when querying the data. Example queries can be found in figure 4.3. Since literals can have explicit datatypes beside the default "String" type, caution is needed when querying them. The example contains data in Turtle syntax. In this syntax, "cat"@en is an RDF literal with a lexical form "cat" and a language identifier en. "42"^^xsd:integer on its turn is a typed literal with the datatype http://www.w3.org/2001/XMLSchema#integer. While "abc"^^dt:specialDatatype on the other hand, is a typed literal with a datatype defined by the URI http://example.org/datatype#specialDatatype. When querying the dataset, the types of the requested literals also need to match. Query 1 in figure 4.3, for example, has no returned values. This because the language tag @en was missing in the query. Therefore, the SPARQL engine was searching for a literal with lexical from "cat", without a language identifier. The second query in the figure has included this query tag and as a result does have a return value. The third query in figure 4.3 will try to match a literal of a numeric type and which has value 42. Here, we can see an interesting aspect of the SPARQL query language; integers in the query will automatically be interpreted as a typed literal with datatype http://www.w3.org/2001/XMLSchema#integer. Therefore, it is not necessary to manually defined the datatype in this query. Similarly, decimal numbers are also automatically typed by the SPARQL query language. The last query in figure 4.3 will match a literal with an arbitrary datatype. Since this datatype is not standardised by W3C, SPARQL will not be able to automatically type this literal. Therefore, in order to retrieve the right results, the datatype of the literal needs to be defined in
Data stored in RDF graph

```
SELECT ?v
WHERE {
}
```

Query and result 1: literal without language tag

```
(SELECT ?v
WHERE {
})
```

Query and result 2: literal with language tag

```
(SELECT ?v
WHERE {
})
```

Query and result 3: literal with numeric type

Query and result 4: literal with arbitrary datatype

Figure 4.3: Matching literals with SPARQL queries

the Basic Graph Pattern. The SPARQL query processor does not have to have an understanding of the
datatype. The query processor will match the literal if both the lexical form ("abc") and datatype URI
<http://example.org/datatype#specialDatatype> match. The query language thus enables querying
literals with own defined datatypes without having to understand the datatype itself. This provides a lot
of flexibility and compatibility with user defined data.

4.2.2 Adding constraints

Up until now, all possible values for the variables defined in the Basic Graph Pattern are returned as a
valid result. This however does not provide enough granularity as one often wants to “filter out” some
unneeded results that do match the Basic Graph Pattern but do not have an added value for the user.
For example, one could create the query in listing 4.1 which selects the names of all employees. This
query has however no possibility to express that only names which have the letter “a” should be returned.
Therefore, filters have been added to the SPARQL specification, allowing to filter all matching bindings of
variables with added constraints that can not be expressed using Basic Graph Patterns.
CHAPTER 4. SPARQL AND SPARQL-MM

```
PREFIX ex: <http://www.example.org>
SELECT ?name WHERE {
    ?name a ex:employee.
}
```

Listing 4.1: Example query without filter

Listing 4.2 shows how a filter can be used in a SPARQL query to only select the results where the value of the name variable has an “a” in it. A filter in SPARQL starts with the `FILTER` keyword, followed by a filter function. In the example, the regex filter function is used which has two obligatory arguments and one optional argument. The first argument is the text pattern on which a regex needs to be evaluated. The second argument is for entering the regex that needs to be applied to the first argument and the optional argument is used for defining “flags” which can make the regex case-insensitive for example. In the example, the text on which the regex needs to be applied is the value bound to the variable `?name` and the regex will tell that this value needs to have an “a” in it. The regex filter function is implemented as a standard in the SPARQL specification, just like some additional filter functions like `BOUND(variable var)` which will return true if the variable `var` is bound to a value, `isIRI(RDF_term term)` which will return true if the given RDF term is an IRI, `LANG(literal ltrl)` which returns the language tag of the variable `ltrl` if it has one. These are only a few of the standardised filter functions defined in the SPARQL specification. More can be found in the SPARQL specification document [25].

```
PREFIX ex: <http://www.example.org>
SELECT ?name WHERE {
    ?name a ex:employee.
}FILTER regex(?title, "a")
```

Listing 4.2: Example query with filter

4.2.3 General working of SPARQL

In the previous sections we explained the syntax and how to compose queries which can be processed by SPARQL. But how do you execute these SPARQL queries and which data will be queried when executing? As already explained, all information on the Semantic Web is expressed in RDF format which basically is a subject-predicate-object based language. Just like HTML files need to be stored on the hard drive of a traditional Web server, so does the RDF data. In order to do so, different RDF stores were created. These will store all the data and create an RDF knowledge graph which is a directed and labeled graph representation of all data stored on this particular server or domain. It will then return all or parts of the graph to the client requesting information from this particular server or domain. These RDF frameworks, as they are called, also often have a SPARQL processor implemented and exposed as an endpoint. This enables users to execute SPARQL queries on the data provided by this particular domain. The SPARQL queries are thus only evaluated on the knowledge graph of this particular domain. For example,
that the University of Ghent has a knowledge graph containing information about all its departments, employees and students. In order to make this data queryable by end-users, the university could set up a SPARQL query processor which will query all their data and expose this as a public endpoint, identified by a certain URI. Similarly, DBpedia, the Semantic Web version of Wikipedia, also provides a publicly accessible SPARQL endpoint that end users can utilise to find information about various topics. This endpoint would however not be able to execute queries over the knowledge graph of the University of Ghent as DBpedia is not the author of this knowledge graph. The end user thus has to know which knowledge graph will be likely to have the information it needs and make use of the SPARQL processor of this knowledge graph if this is publicly available. Some companies or organisations might also have SPARQL query engines but for various reasons choose not to make it publicly available. In this case, the end user is out of luck and is not able to perform queries over the knowledge created by this organisation.

4.3 SPARQL-MM

In the previous section, the SPARQL specification was explained by illustrating some examples. In these examples, we came to the conclusion that filters are needed in order to be able to refine the results that are returned by the SPARQL processor as not everything can be expressed using Basic Graph Patterns. We also discovered that the SPARQL specification already defined some standard filter functions like string matching using the \regex function, type checking by using the \isIRI function etc. Even though these functions come in very handy for most RDF data, they are of little use when searching for particular media files and more particularly, Media Fragments on the Semantic Web. Finding a picture where subject A stands to the left of subject B is for example impossible to query in the standard SPARQL specification. One way to solve this would be to query for images that have both subjects and use SPARQL to get all these images. After this, you then have to check yourself in which pictures subject A is left to subject B. This can be very cumbersome and with a large result set, can take a lot of time.

Therefore, Kurz et al. have created an extension on the standard SPARQL specification called SPARQL-MM. This extension defines new filters and media specific concepts that can be used when querying media on the Semantic Web. More specifically, SPARQL-MM will assume that all media on the Semantic Web follows the Media Fragment URI standard. It will then use these Media Fragment URIs to enable users to define filters on the temporal domain, spatial domain or even a combination of both. This while also defining some aggregator functions which can be used in the \select statement of the query. More specifically, SPARQL-MM will define three types of filter functions: topological, directional and temporal functions.
4.3.1 Topological functions

The topological functions will all be applicable for Media Fragment URIs which make use of the spatial domain. As one may assume, these functions will enable users to filter the results according to the relative relationship of different spatial objects in the media source. One example of a topological function is whether one fragment overlaps with another fragment in the source. This can be checked using the x,y,w,h values of the Media Fragment URIs of both objects. In order to define the different topological relationships, the Dimensionally Extended nine-Intersection Model (DE-9im) is used as this is a widely used standard in mathematics. It allows for ten different spatial relations to be defined and calculated based on a standard euclidian coordinate system.

Since SPARQL-MM uses this model for topological reasoning, it has defined ten different topological functions which all coincide with the ten predicates defined by the Intersection Model. The list below contains all ten topological functions. These can all be used as a filter when creating a SPARQL query. Since the names of the functions are self-explanatory, no further detail will be given about the functionality of each of them. If one is interested in how these functions relate to the intersection model, a clear explanation can be found in "A small set of formal topological relationships suitable for end-user interaction" by Clementini et al.

- spatialEquals
- disjoint
- touches
- spatialContains
- covers
- intersects
- within
- coveredBy
- crosses
- spatialOverlaps

Next to the topological query functions, SPARQL-MM also defined two aggregation functions which can be used in the SELECT statement or Basic Graph Pattern of the query and which can be applied to both spatial and temporal fragments. Both functions will return a new spatial or temporal fragment, depending on the input. The first function is the boundingBox function which takes two or more fragments and returns a bounding box containing both fragments. In the case of spatial fragments, a new spatial fragment will be returned with the x, y, w and h values set so it contains all given spatial fragments. When temporal fragments are passed to the boundingBox function, a new temporal fragment will be returned in which the starting point and duration is set to include all passed temporal fragments. The second aggregation function is the intersection function which will return a new fragment which contains the intersection of the passed fragments.
4.3.2 Directional functions

Next to topological functions, directional functions also make use of the spatial domain in Media Fragment URIs. However, instead of defining the topological relative relationship between two fragments or objects, the directional functions will express a certain directional relation between two fragments like fragment A is right beside fragment B etc. Just like the topological functions, the directional functions also make use of a mathematical model to define them. In this case, the projection-based model is used which defines eight possible directions in which two objects can relate to each other. This model was chosen because it is easy to understand for the users, allows for efficient calculations and allows to use intuitive and self-explanatory predicate names. The projection-based model is originally created for comparing the directional relation between two points in a coordinate system. This model can however easily be adapted to also work with objects containing multiple points by taking the centroid of the objects that are compared with each other. Since spatial Media Fragments can only be rectangular, the centroid can easily be calculated by dividing the width and height by two and adding this value to the x and y coordinate respectively. SPARQL-MM will again define a spatial directional function for each spatial predicate defined in projection-based model. These are again available as filters and can be found in the following list.

- leftBeside
- rightBeside
- above
- below
- leftAbove
- rightAbove
- leftBelow
- rightBelow

4.3.3 Spatial access functions

Up until now, the provided functions, except for the two aggregation functions, were all relational functions. Meaning that two spatial fragments are taken as input and their relation to each other is checked. These functions are all implemented as filters. It would however also be handy if we could access some properties of the matched Media Fragments like the width, height, position etc. Therefore, SPARQL-MM has also defined and implemented some accessor functions which can be called in the `SELECT` clause of the SPARQL queries. An example query using the width accessor function can be found in listing 4.3. In conclusion, all the accessor functions of spatial Media Fragments which are implemented in SPARQL-MM can be found in the following list:

- area
- center
- height
- width
- spatialFragment
- hasSpatialFragment
- xy
CHAPTER 4. SPARQL AND SPARQL-MM

```
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?f1 ?f2 (mm:width(?f1) as ?width) WHERE {
  ?image ma:hasFragment ?f1;
  ma:hasFragment ?f2.
  ?f1 dc:description "car".
  ?f2 dc:description "person".
}
```

Listing 4.3: Example query using width accessor function

More information about the projection-based model can be found in a paper written by Skiadopoulos et al. [27], which contains a survey about the different directional relation models for extended objects.

4.3.4 Temporal functions

Up until now, only spatial information has been taken into account when creating functions for media fragment querying. Since the Media Fragment URI standard also has a temporal domain defined, SPARQL-MM also has defined some functions to enable the users to filter according to temporal relations. The mathematical model used for these functions is Allen’s interval algebra for temporal reasoning [28]. This model defines thirteen different possible temporal relations between two time intervals. In this model, a point in time t will be interpreted as an interval with a start time t and duration zero. SPARQL-MM will again make all thirteen temporal predicates available as filter functions, enabling the users to perform fine-grained searches over media data in the Semantic Web. All temporal relations functions can be found in the following list.

- precedes
- meets
- overlaps
- finishedBy
- contains
- starts
- equals
- startedBy
- during
- finishes
- overlapedBy
- metBy
- precededBy

4.3.5 SPARQL-MM implementation

SPARQL-MM is an extension on the standard SPARQL query language and makes use of real filters. This means that the functions are all applied at the end of the SPARQL process. This enables the functions to be used or implemented in any SPARQL evaluator. Currently, SPARQL-MM has been implemented
using the OpenRDF Sesame API and can be added to any Sesame Triplestore using Java class Loader Technology. Currently, Apache Marmotta Triplestore is the main supported Sesame Triple store. This is an RDF framework that enables users to store all their RDF data in a PostgreSQL or MySQL database which then can be accessed using the built in SPARQL processor.

1 http://www.openrdf.org/
2 http://marmotta.apache.org/kivi/triplestore.html
In the previous chapter we have talked about SPARQL and in which way it enables you to query RDF data stored in a triple store. This approach however has a downside. The SPARQL query is fully processed on the server and only the solution mapping or needed RDF triples are returned to the requesting client. Next to this it provides a great expressiveness to the clients which is a good thing. Each of them can create their own unique SPARQL queries containing very selective and specific Basic Graph Patterns catered to the needs of the requesting client. This expressiveness however, comes at a cost which is a high CPU usage when a lot of clients are requesting simultaneously with very specific queries. Availability is therefore a big problem for publicly available SPARQL endpoints. In 2015, Buil-Aranda et al. performed a study monitoring the availability of the most popular publicly available SPARQL endpoints. They did this every hour for a duration of 27 months\textsuperscript{29}. From their study it was shown that only 32.2\% of the publicly available endpoints have a two-nine availability, meaning that the endpoint is available for 99\% - 100\% of the time. All other endpoints had an availability of under 99\% which means that they are unavailable for more than 3.65 days a year. Public SPARQL endpoints are therefore not a sustainable way of querying information on the Semantic Web and can not be used in critical applications where the functionality has to work all the time.

Because of these problems, a lot of organisations opt to not provide public SPARQL endpoints but only make a data dump available of their whole knowledge graph. This approach is the complete opposite of SPARQL when it comes to availability and functionality. When a user wants to retrieve some information from a certain knowledge graph, it has to download the data dump and process the information locally. The client therefore has to have more functionality and must be more powerful compared to the SPARQL approach. This technique has also it’s downsides. First of all there is a lot of wasted bandwidth. Suppose that a client only needs a very small subset of triples from the knowledge graph to fulfil its query. The client now has to download the whole knowledge graph only to access a very small subset of it. Next to the wasted bandwidth, there is also a problem with updating information. Suppose that a user has downloaded the data dump of a knowledge graph and is using this for over a year. It is highly probable that this information has been updated in the coarse of this year and thus the client is using outdated information. It therefore has to periodically download the data dump in order to have the latest possible information.
The third possibility to access Linked Data on the web is by using Linked Data documents. In this solution, the whole knowledge graph is divided into different documents. Each of the documents contain all triples that contain information about a certain topic in the graph. DBpedia for example has provided Linked Data documents of each of their topics. If one would for example access the following URI http://dbpedia.org/page/Belgium, a document will be returned containing all triples in the knowledge graph where the given URI is the subject or object. Because the Linked Data document contains triples which also contain other URIs, the clients can follow these and this way traverse the knowledge graph. Because of the use of separate documents for each subject, the client needs to be aware of the URIs of the different topics it needs for answering a certain query. For this it can use some different techniques like creating a predetermined list of pages available from this knowledge graph and using this to improve the query execution speed. Another possibility is to do a real-time traversal while executing the query. In both cases, the client needs to do all the work just like the case of data dumps. The benefit however is that the queried data is live and thus can not be outdated.

We can clearly see that SPARQL and data dumps are somewhat of each others opposite. SPARQL will enable clients to evaluate very expressive queries on live data without the need for a performant client. Data dumps on the other hand are not live, do not allow for expressive queries and need more powerful clients to store and query the data from the knowledge graph. They however do not have the problem of poor availability which is the big issue with public SPARQL endpoints. Therefore, Triple Pattern Fragments (TPF) was created in order to strike a balance between the two. It will allow to query live data just like SPARQL endpoints but in such a way that the server load is diminished. This will allow for better availability compared to public SPARQL endpoints while still having some of its benefits, being the expressive queries and querying of live data. Figure 5.1 shows a schematic of how the three proposed solutions compare to each other in terms of client complexity, query expressiveness and availability.

![Figure 5.1: Comparison of Linked Data access methods in function of client cost, server cost, and request specificity. Figure from Linked Data Fragments](30)

5.1 Triple Patterns and main concept

In order to access Linked Data in such a manner that the processing load is balanced between the client and server, the Triple Pattern Fragments framework has to define a server interface which enables querying RDF data in such a way that the CPU load is minimised. Therefore, Triple Pattern Fragments has opted for a server interface which enables you to query for a single triple pattern at a time. The server will then select all triples that match the given triple pattern and return them. Note that triples are returned and
not solution mappings. This because solution mappings are more specific compared to triples which could negatively impact the performance of caching servers. The client could for example send the following triple pattern to the Triple Pattern server (\texttt{?subject, ex:hasBirthPlace, ex:Belgium}). The server will then select all triples that match with the predicate and object of the given triple pattern and return all triples. The triple (\texttt{ex:Rembrandt, ex:hasBirthPlace, ex:Belgium}) could for example be one of them.

There has been chosen to use this interface for two reasons. First of all, triple patterns are the most basic elements of a SPARQL query as they all combined form the Basic Graph Pattern. Next to this, there exists a very efficient way to select all triples in an RDF store that match a single triple pattern. This will aid in the lower CPU usage of the server and thus the higher availability. Next to the matching triples, the Triple Pattern fragments server will also return some additional metadata which will aid the client in its execution of a query. This metadata will contain an estimate of the cardinality of the results set of the triple pattern. In other words, it will return an estimate of how many matches it has in its knowledge graph for the given triple pattern. This is done because of query execution planning and pagination. The server will not return all results all at once but paginate the results. This is useful for data savings as the client only has to receive a small amount of triples at a time. Because of the way queries are executed at the client, this will drastically decrease the bandwidth usage. In the following section, a detailed explanation is given on how the client executes SPARQL queries.

5.2 Query Execution

5.2.1 Conceptual workflow

In the previous section, the Server interface has been defined. Now, we will explain how to client can use this server interface to solve a SPARQL query on the live knowledge graph it represents. In order to execute the query, the client must be provided two arguments: the URI of a page of a triple fragment provided by the TPF server and the SPARQL query that needs to be evaluated. From the URI, the client can request hypermedia controls from the server which will inform the client that the server accepts queries in the form of triples and that it will return matching triples which are paginated and combined with some additional metadata. Once the hypermedia controls are extracted, the client can start evaluating the SPARQL query.

First, the BGP $B$ is split into connected sub-Basic Graph Patterns (sub-BGPs), denoted by $B_i$. If the query contains two or more of these sub-BGP’s, the client can evaluate them in parallel, thus making the evaluation faster. If we for example take a look at listing 5.1, we can see that the triple (\texttt{?city, eg:locatedIn, eg:France}) has no common variables with the other two triples in the BGP. This triple can thus be resolved completely independent from the other two without interference. Note that this approach is based on the divide-and-conquer philosophy which will also be implemented later on in the query execution algorithm.
CHAPTER 5. TRIPLE PATTERN FRAGMENTS

Listing 5.1: Example of SPARQL query with two sub-BGP’s

Once the BGP is split into sub-BGP’s $B_i$, the same mechanism is used to find solution mappings for each $B_i$. To do this, the client will send a request to the server for each triple in the (sub-)BGP and only retrieve the first page which will contain a limited set of matching triples together with the count metadata which will indicate the estimated number of results there are for the request triple. If there is at least one triple pattern in the sub-BGP that has no matches, then the result of the sub-BGP is empty. The result of the first retrieval of an example query can be found in listing 5.2. We can see that the first two triples will result in a lot of matches as these are a lot of people who live in New York and as everyone has a name. There are however far less authors which is shown by the smaller number of matches returned by the server. Note that this example has only one sub-BGP, being the whole BGP.

Listing 5.2: Example query searching for all authors living in New York

After collecting the count metadata, the client will continue the divide-and-conquer strategy and select the triple pattern $t_{pc}$ with the smallest number of matches which in our case is $t_{p3}$. This will minimise the number of recursive calls in the further execution of the query and thus also the number of HTTP requests, effectively lowering the bandwidth and CPU usage of the server. Once the triple pattern with the lowest number of matches is determined, the client will fetch all pages of the TPF for the triple pattern $t_{pc}$ and use this information to create a set of solution mappings $\Omega^c$ for $t_{pc}$. For listing 5.2, $t_{pc} = t_{p3}$ and a possible solution mapping in $\Omega^c$ could be $\lambda_1 = \{\text{?person} \mapsto \text{eg:Hemingway}\}$ or $\lambda_2 = \{\text{?person} \mapsto \text{eg:Fitzgerald}\}$.

Once the set of mappings $\Omega^c$ has been determined, the selected triple pattern $t_{pc}$ is removed from the sub-BGP $B_i$ and each solution mapping in $\Omega^c$ is applied to the resulting sub-BGP, effectively creating a new sub-BGP $B_i'$ with less variables. For each $B_i'$ the same procedure as above will be repeated to retrieve other solution mappings for the remaining variables. Listing 5.3 shows the resulting sub-BGP $B_i'$ when we apply the solution mapping $\lambda_1$ to the BGP. The client will again start by retrieving the number of matches for each of the two remaining triple patterns. However, since Hemmingway never lived in New York, this triple pattern will have zero results. The sub-BGP will thus not have any results for the current solution mapping $\lambda_1$ and the execution for this solution mapping stops.
CHAPTER 5. TRIPLE PATTERN FRAGMENTS

1 SELECT ?name WHERE{
2   eg:Hemingway eg:livesIn eg:New_York.   # ± 0 matches (tp1)
3   eg:Hemingway eg:hasName ?name.         # ± 1 match (tp2)
4 }

Listing 5.3: Example query searching for all authors living in New York after first solution mapping is applied

When the second solution mapping is applied however, both triple patterns in $B'_1$ have one match each. The client can then choose to evaluate $tp_2$ and create the set of solution mappings $\Omega^i$ for the variable $?\text{name}$ and combine each of these solution mappings with their corresponding solution mapping of $\Omega^c$. Since we only have one match for $tp_2$, $\Omega^i$ will only contain one solution mapping $\lambda_3 = \{?\text{name} \mapsto \text{"Scott Fitzgerald"}\}$. This partial solution mapping is then combined with $\lambda_2$ to create the full solution mapping $\{?\text{person} \mapsto \text{eg:Fitzgerald}, ?\text{name} \mapsto \text{"Scott Fitzgerald"}\}$ which is a solution for the given query. This process will repeat recursively for each possible solution mapping in the set $\Omega^c$ which will result in all possible names of authors who have lived in New York.

1 SELECT ?name WHERE{
2   eg:Fitzgerald eg:livesIn eg:New_York. # ± 1 match (tp1)
3   eg:Fitzgerald eg:hasName ?name.      # ± 1 match (tp2)
4 }

Listing 5.4: Example query searching for all authors living in New York after second solution mapping is applied

5.2.2 Iterative implementation of query execution

One approach to implement the previous workflow would be to implement it using a recursive function. There are however some disadvantages when using this approach. First of all, it could only return all solution mappings at once as each function call has to finish before a return can be made. This means that intermediary solution mappings need to be kept in memory until all mappings are found. When a lot of solution mappings exists, this could lead to out-of-memory issues especially for clients with a limited amount of memory. Secondly, a lot of needless work is done when only a limited amount of the solution mappings need to be returned (for example when a LIMIT is used in the SPARQL query). Therefore, the actual implementation of the client follows the iterator model. This model defines the concept of an iterator which is an operator that allows intermediary results to be returned individually and in an iterative manner. An iterator defines three functions: open, close and getNext. The open function will perform the initial preparations like setting up the needed data structures and allocating resources to start the iteration. Similarly, the close function will stop the iteration and release the allocated resources. The getNext function allows for the caller to retrieve the next result of the operation that is defined in the iterator. Using this iterator model, the queries are all executed as a tree of iterators where the parent iterators will request the next result of their children and perform an operation with the values returned
by the children which on its turn is returned to its parent. The TPF client has defined three different types of iterators which all combined will be able to dynamically create a tree of iterators which all work together to incrementally solve the query and return results in a non-blocking way.

The first iterator that is defined is called the rootIterator. This iterator is, as the name suggest, a simple iterator for which the getNext function does not take any input and which returns an empty mapping \( \emptyset \). The pseudocode of the getNext function for the root iterator can be found in listing 5.5. Since this iterator does not take any other iterators as input and only returns an empty mapping, it can be used as a starting point for a TriplePatternIterator or BasicGraphPatternIterator.

The pseudocode for the next two iterators can be found in listing 5.6 and 5.7. The TriplePatternIterator will take the solution mappings of a source iterator \( I_s \) as input together with a triple pattern \( \{tp\} \). It will then gather possible mappings for variables in the triple pattern, taking the solution mappings of \( I_s \) into account. For example, suppose the TriplePatternIterator was given the triple pattern \( {?person, eg:hasName, ?name} \) and the source iterator \( I_s \) has provided a solution mapping \( \lambda_s = \{?person \mapsto \text{eg:Fitzgerald}\} \). The TriplePatternIterator will then request the following triple from the server \( \langle\text{eg:Fitzgerald, eg:hasName, ?name}\rangle \) which on its turn will return a triple \( \langle\text{eg:Fitzgerald, eg:hasName, "Scott Fitzgerald"}\rangle \). The Triple PatternIterator will combine this mapping for the \?name variable \( \lambda \) with the given solution mapping \( \lambda_s \) and return this combined solution mapping \( \lambda \cup \lambda_s = \{?person \mapsto \text{eg:Fitzgerald, ?name} \mapsto \text{"Scott Fitzgerald"}\} \).
CHAPTER 5. TRIPLE PATTERN FRAGMENTS

Listing 5.6: Pseudocode for the getNext function of the 'TriplePatternIterator', pseudocode from Verborgh et al. [30]

```
1 Function TriplePatternIterator.getNext()
2   Data: A source iterator Is; a triple pattern tp: controls
3   Output: The next mapping \( \lambda' \) such that when \( \lambda' \) is filled in the triple pattern tp, the tp is present in the knowledge graph, or nil when no mappings are left
4   self.\( \emptyset \) \( \leftarrow \) an empty page without triples or controls if self.\( \emptyset \) had not been assigned to previously;
5   while self.\( \emptyset \) does not contain unread triples do
6      if self.\( \emptyset \) has a control to a next page with URL \( u_{\emptyset'} \) then
7         self.\( \emptyset \) \( \leftarrow \) GET \( u_{\emptyset'} \);
8      else
9         self.\( \lambda_0 \) \( \leftarrow \) Is.getNext();
10        return nil if self.\( \lambda_0 \) = nil;
11        self.\( \emptyset \) \( \leftarrow \) GET page 1 of the TPF for self.\( \lambda_0 \)[tp];
12     end
13    end
14    t \( \leftarrow \) an unread triple from self.\( \emptyset \);
15    \( \lambda \) \( \leftarrow \) a solution mapping \( \lambda' \) such that dom\( \lambda' \) = vars\( \)tp\) and \( \lambda'[tp] = t \);
16    return \( \lambda \cup \) self.\( \lambda_0 \);
17 end

Listing 5.7: Pseudocode for the getNext function of the 'BasicGraphPatternIterator', pseudocode from Verborgh et al. [30]

```

```
1 Function BasicGraphPatternIterator.getNext()
2   Data: A source iterator Is; a BGP B with number of triple patterns \( > 2 \);
3   Output: The next mapping \( \lambda' \) such that \( \lambda' \in [[B]]F \), or nil when no such mappings are left, controls
4   \( \lambda \) \( \leftarrow \) nil;
5   self.Ip \( \leftarrow \) nil if self.Ip has not been assigned previously;
6   while \( \lambda = \) nil do
7      while self.Ip = nil do
8         self.\( \lambda_\emptyset \) \( \leftarrow \) Is.getNext();
9      return nil if self.\( \lambda_\emptyset \) = nil;
10     foreach triple pattern tpj \( \in B \) do
11        cntj \( \leftarrow \) GET count from TPF server
12     end
13     if \( \forall j : cnt_j > 0 \) then
14        \( c \) \( \leftarrow j \) such that \( cnt_j \leq \) cntk \( \forall tp_k \in B \);
15        Is \( \leftarrow \) TriplePatternIterator(RootIterator(), self.\( \lambda_\)tpv, c);
16        self.Ip \( \leftarrow \) BasicGraphPatternIterator(Is, self.\( \lambda_\)B \ \{tpv\}, c);
17     end
18   end
19   \( \lambda \) \( \leftarrow \) self.Ip.getNext();
20   self.Ip \( \leftarrow \) nil if \( \lambda = \) nil;
21 end
22 return \( \lambda \cup \) self.\( \lambda_\emptyset \);
23 end
```
CHAPTER 5. TRIPLE PATTERN FRAGMENTS

The BasicGraphPatternIterator will take a source iterator and BGP as input and will create a new iterator pipeline for each incoming solution mapping. The iterator will read the incoming solution mappings and will determine, given the mapping, which triple pattern in the BGP has the lowest number of matches (lines 10 - 12). After this, a new iterator pipeline (self.ip) is created by defining a new TriplePatternIterator \( I_p \) and passing the current solution mapping and selected triple pattern \( tp \) to it. This TriplePatternIterator is then given as source for the new BasicGraphPatternIterator self.ip which is also passed an updated BGP in which the currently selected triple pattern \( tp \) has been removed. The mappings that are returned by this pipeline are then combined with the input mapping after which they are returned. The BasicGraphPatternIterator will thus always select the triple pattern with the smallest number of matches based on the current solution mapping that is provided to it and create a new branch in the iterator tree to solve the BGP further with the current solution mapping. By doing this, the iterator thus dynamically selects the best strategy while executing the query and no query execution strategy needs to be calculated or determined prior to executing the query. It is clear that using this iterator approach and creation of iterator pipelines, a tree-like structure emerges where the depth and number of the branches depends on the number of solutions there are in the knowledge graph for the given query. The iterator structure will also allow the client to present results in a streaming fashion, meaning that one branch of the tree could terminate and return a result (if there exists one for this particular branch) while other branches are still being evaluated and expanded.

5.3 Limitations of Triple Pattern Fragments

The Triple Pattern Fragments framework can support all SPARQL query constructs. Some of them do however present some drawbacks due to the combination of two factors: 1) the way the client query execution algorithm is defined and 2) the inherent nature of the query construct which can make it so that the iterator scheme can not return results in an iterative manner while the query is running. The LIMIT and OFFSET query modifiers can both be implemented in such a way that the results are returned to the user in a streaming fashion. The LIMIT modifier will however stop the execution when a certain amount of results are returned while the OFFSET modifier will suppress the first results and start returning the results when the offset is met. The DISTINCT modifier can also return the results in a streaming fashion while the query is executing and thus does not negatively affect the execution time/responsivity. When a FILTER is used, the filter function can be executed as soon as the variable(s) in the filter have a solution mapping. This seems as a big benefit but because of the way the BasicGraphPatternIterator is implemented, the query execution planning will not take the variables in the filter into account. For filters with a low selectivity, this does not pose a big deterioration in the performance of the client. When the filter would however be very specific, it would be very beneficial to plan this filter as early as possible in the query execution as applying this filter would eliminate a lot of iterator pipelines and thus improve the overall bandwidth usage, CPU usage of server and client and query execution time. For example, the filter 
\begin{verbatim}
FILTER(\text{REGEX}(\text{name}, \text{"Richardson"}, \text{"i"}))
\end{verbatim}
is a specific filter as there are not a lot of people who have
the string "Richardson" in their name. If this filter could be applied early in the query execution, it would
dramatically decrease the number of iterator pipelines. Unfortunately because of the implementation in
listing 5.7, all people and their names need to be retrieved first from the server as we need a binding for
the ?name variable before we can apply the filter. This makes it so that the execution time of queries that
use filters becomes a lot larger compared to queries without filters.
CHAPTER 6

Research question and hypotheses

In the previous chapters, all the technologies regarding this thesis have been explained in detail. In chapter 4.3, we have introduced SPARQL-MM which will enable users to query media files which are hosted on the Semantic Web and which follow the Media Fragments URI standard to define fragments inside the media sources. This makes the queries far more powerful when searching for media files. There is however a problem with this technology: the poor scalability when a lot of simultaneous clients connect to the server. Since the SPARQL-MM filters are all processed server-side, the CPU usage of the server will increase drastically when a lot of clients simultaneously start to query media files. Since popular media hosting solutions and social media sites have a massive amount of users, a very powerful and costly server infrastructure is needed in order to cope with all the simultaneous requests. Therefore, it would be beneficial if we could provide the same functionality while reducing the CPU load for the server, thus effectively increasing the scalability when it comes to the number of simultaneous clients. In order to do this, TPF could be used and expanded upon to provide the SPARQL-MM functionality in a more scalable way. On top of that, an investigation is done whether the URI structure that is imposed in the Media Fragments URI standard can aid the TPF client in faster query execution and better execution planning. In the following, we define the research questions and hypotheses that will be answered in this thesis.

Research question 1

**Question:** Can we improve the scalability of SPARQL-MM enabled systems in terms of the number of simultaneous clients that can be served by moving the SPARQL-MM functionality to the client?

**Hypothesis 1.1:** Moving the spatial and temporal reasoning to the client will negatively impact the query execution speed for a single machine executing a single SPARQL-MM query compared to complete server-side reasoning.

**Hypothesis 1.2:** By moving spatial and temporal reasoning to the client, the SPARQL-MM query execution time will not increase as fast with an increasing number of clients accessing the server simultaneously compared to complete server-side reasoning.
CHAPTER 6. RESEARCH QUESTION AND HYPOTHESES

**Hypothesis 1.3:** The CPU usage of the server is smaller and will increase more slowly with an increasing number of clients accessing the server simultaneously when the spatial and temporal reasoning is moved to the clients.

**Hypothesis 1.4:** The bandwidth usage will be bigger when implementing client-side spatial and temporal reasoning compared to complete server-side reasoning.

**Research question 2**

**Question:** Can we improve the performance of the Triple Pattern Fragment’s execution of SPARQL-MM queries by using the Media Fragments URI standard?

**Hypothesis 2.1:** By making use of the URI structure defined in the Media Fragments URI standard, the execution time of SPARQL-MM queries will be lower or at least as fast compared to the standard Triple Pattern Fragment SPARQL-MM query execution.

**Hypothesis 2.2:** By making use of the URI structure defined in the Media Fragments URI standard, the bandwidth usage will be smaller compared to the standard Triple Pattern Fragment SPARQL-MM query execution.

**Hypothesis 2.3:** By making use of the URI structure defined in the Media Fragments URI standard, the number of requests to the TPF servers will be smaller compared to the standard Triple Pattern Fragment SPARQL-MM query execution.

**Hypothesis 2.4:** By making use of the URI structure defined in the Media Fragments URI standard, the CPU usage of the server will be lower compared to the standard Triple Pattern Fragments SPARQL-MM query execution.
CHAPTER 7

Client-side media querying

As already described in the previous sections, we will try to make a media query system which provides the user with fine-grained search functionalities while reducing the server load in order to improve the scalability compared to the currently available SPARQL-MM implementations. The Triple Pattern Fragments framework is the perfect starting base for this as it already achieves a better balance between server and client workload. In order to allow users to query for media (fragments) in a fine-grained manner, the Triple Pattern Fragments framework will be altered to also support SPARQL-MM queries. On top of that, we will try to optimise the query execution speeds of the Triple Pattern Framework for SPARQL-MM queries by using the URI structure of the Media Fragments URI standard. This by using two possible improvements: client-side binding updates and query rewriting.

7.1 Providing SPARQL-MM functionality in Triple Pattern Fragments

Since the client itself is responsible for planning and executing the SPARQL query execution in the Triple Pattern Fragments framework, we have no need to alter the server implementation. We thus only need to add the needed functions and supporting classes to the client in order to provide the functionality of SPARQL-MM. Therefore we will implement functions that will allow the users to use accessor, aggregation and relational functions both in the temporal and spatial domain just like SPARQL-MM does. Since we only have found a suitable dataset for spatial Media Fragments, we will mainly focus on the implementation of this domain and test the correctness of the functions using the created dataset. The temporal relational, accessor and aggregation functions are also added to the client, albeit not tested extensively. As the goal of this thesis is to investigate what the performance of queries are when using Triple Pattern Fragments and whether we can use the Media Fragments URI standard to improve the execution speed, it suffices to have only one domain fully tested and used for executing performance evaluations.

7.1.1 Spatial accessor, aggregation and relational functions

Section 4.3 already explained how SPARQL-MM extension provides different types of spatial functions: accessor, aggregation and relational functions. The relational functions are further divided into two subcategories: the directional relation functions and topological relation functions. These will all take two
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

Figure 7.1: Using the centroid model to perform topological calculus

Media Fragments URIs as input and return a boolean value depending on whether the directional or topological relation between the two is met or not. Because of this functionality, SPARQL-MM implemented these as filter functions and so does the altered Triple Pattern Fragments client.

Selection of the spatial fragment representation

For implementing the spatial reasoning in the Triple Pattern Fragments, we first need to determine which model we want to use to base our reasoning upon. As already noted in section 3.2.2, the Media Fragments URI standard only allows for rectangular spatial fragments. This means that each fragment in the media file is automatically identified by a bounding box. A bounding box model would therefore seem the most obvious model to use for spatial reasoning. This model will take the outer most coordinates of the bounding box into consideration when calculating the spatial relation between two objects. There is however one more model that is commonly used in mathematics called the centroid model. In this model, the centroid of the spatial object is taken to base the calculations on. Even though this model has some benefits like only having to take one single point into consideration when using topological or directional calculus, it is not suitable to use for the purpose of Media Fragments as it can cause for some counterintuitive results. If we look at figure 7.1, we can see that using the centroid model and Frank’s cone-shaped directional calculus \[31\] lead to object B being classified as being to the west of object A. However, if we look at both objects, a seemingly more correct interpretation would be that object B is located above object A. Therefore, the centroid model is not used as a base for the directional and topological calculi. Instead, we will be using bounding boxes to represent the spatial Media Fragments.

Once we have the base model to represent the Media Fragments, we need to select a calculus to calculate the topological and directional relations between two objects. Even though there are some calculi which combine both topological and directional relations in one calculus like the one created by Sun & Li \[32\], they in essence combine a separate topological and directional calculus into a very detailed calculus which is too extensive for our purposes. Therefore, we chose to use a topological and directional calculus
separately as a combination of the two will provide enough flexibility and preciseness to accommodate the most widely used spatial relations when searching for media and Media Fragments.

Topological relation calculus

When selecting a topological model for 2D geometrics, which Media Fragments are, we need to take into account how the model allows us to define the most common and easy to understand spatial relations between two object. The most common predicates like “intersects”, “is equal to” etc. need to be defined and calculable in order to provide the user with the correct Media Fragments. GeoSPARQL opted to use the region connection calculus (RCC-8) [33] which defines the eight topological relations of which a graphical representation can be found in figure 7.2.

- disconnected (DC)
- externally connected (EC)
- equal (EQ)
- partially overlapping (PO)
- tangential proper part (TPP)
- non-tangential proper part (NTPP)
- tangential proper part inverse (TPPi)
- non-tangential proper part inverse (NTPPi)

![Graphical representation of the topological relations defined in RCC-8](image)

Figure 7.2: Graphical representation of the topological relations defined in RCC-8, figure from "Region Connection Calculus: Its models and composition table" [33]

When we look at the relations defined in this calculus, we can see that it defines some relations which do not add any functionality for querying Media Fragments. It is for example very unlikely for a user to search for a Media Fragment that covers another fragment and for which the bounds of both fragments coincide. This is also reflected in the implementation of the standard in GeoSPARQL in which TPPi and NTTPPi are both mapped to the “contains” property while TPP and NTTP are both being mapped to the “within” property as can be seen in figure 7.3. Therefore, we need an alternative topological calculus which provides us with more (easily) useable topological relations. The standard model used for this is the Dimensionally Extended nine-Intersection Model (DE-9IM), created by Clementi et al. [26] which is also used by SPARQL-MM itself. This model also provides the best balance between usability and expressiveness.
The model defines 521 possible topological relations of which 10 have a name that coincides with their logical meaning like “intersects”, “overlaps” etc. It are these 10 relations which we humans also use most frequently and therefore, these are the ones implemented to provide the user with topological reasoning. DE-9IM defines a 3x3 matrix which can be found below and which forms the base to define the different topological relations. The \( \text{dim}(\cdot) \) function defines the maximum number of dimensions of the intersection of the two arguments. \( I(a) \) denotes the interior of a 2D geometry \( a \), \( B(a) \) indicates the boundary of the geometry \( a \) and \( E(a) \) is the exterior of the geometry \( a \).

\[
DE - 9IM(A, B) = \begin{bmatrix}
\text{dim}(I(A) \cap I(b)) & \text{dim}(I(A) \cap B(b)) & \text{dim}(I(A) \cap E(B)) \\
\text{dim}(B(A) \cap I(b)) & \text{dim}(B(A) \cap B(b)) & \text{dim}(B(A) \cap E(B)) \\
\text{dim}(E(A) \cap I(b)) & \text{dim}(E(A) \cap B(b)) & \text{dim}(E(A) \cap E(B))
\end{bmatrix}
\]

A graphical representation of the DE-9IM matrix can be found in figure 7.4 where 0 means that the intersection between the two geometries is a point, 1 indicates that the intersection is a line (1-dimensional) and 2 indicating that the intersection of the two geometries is an area (2-dimensional). Using these definitions, DE-9IM allows us to calculate ten spatial relations that are both widely used and easy to understand.

Since SPARQL-MM also uses this model and the 10 topological relations it describes to define the functions in their ontology, we will refrain from creating our own ontology and use the one defined by Kurz et al. to identify the topological relation functions. In table 7.1, the 10 topological relations, their logical meaning, definition and their URI using the SPARQL-MM ontology are enlisted. These functions are all supported by the Triple Pattern Fragments client and are implemented according to the DE-9IM model.
### Table 7.1: Topological spatial functions implemented in the Triple Pattern Fragments client

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning according to DE-9IM</th>
<th>Relation URI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>equals(A,B)</strong></td>
<td>&quot;fragment A and fragment B are equal if their interiors intersect and no part of the interior or boundary of one geometry intersects the exterior of the other&quot; [26]</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialEquals">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialEquals</a></td>
</tr>
<tr>
<td><strong>disjoint(A,B)</strong></td>
<td>fragment A and fragment B are equal if they have no point in common</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialDisjoint">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialDisjoint</a></td>
</tr>
<tr>
<td><strong>touches(A,B)</strong></td>
<td>fragment A touches fragment B if they have at least one boundary point in common</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#touches">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#touches</a></td>
</tr>
<tr>
<td><strong>contains(A,B)</strong></td>
<td>&quot;fragment A contains fragment B iff no points of B lie in the exterior of A and at least one point of the interior of B lies in the interior of a&quot; [36]</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#touches">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#touches</a></td>
</tr>
<tr>
<td><strong>covers(A,B)</strong></td>
<td>fragment A contains fragment B if no points of B lie in the exterior of A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#covers">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#covers</a></td>
</tr>
<tr>
<td><strong>intersects(A,B)</strong></td>
<td>fragment A intersects fragment B if they have at least one point in common</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#intersects">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#intersects</a></td>
</tr>
<tr>
<td><strong>within(A,B)</strong></td>
<td>fragment A is within fragment B if A lies in the interior of B. This is equal to Contains(B,A)</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#within">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#within</a></td>
</tr>
<tr>
<td><strong>coveredBy(A,B)</strong></td>
<td>fragment A is covered by fragment B if every point of A is a point of B and the interiors of the two have at least one point in common. This is equal to Covers(B,A)</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#coveredBy">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#coveredBy</a></td>
</tr>
<tr>
<td><strong>crosses(A,B)</strong></td>
<td>fragment A crosses fragment B if they have some but not all interior points in common. Further more, the dimension of the intersection is less than that of at least one of them. In case of Media Fragments, this would mean that A crosses B if they have a single row or column of pixels in common.</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#crosses">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#crosses</a></td>
</tr>
<tr>
<td><strong>overlaps(A,B)</strong></td>
<td>fragment A overlaps fragment B if they have some but not all points in common.</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialOverlaps">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialOverlaps</a></td>
</tr>
</tbody>
</table>

![Graphical representation of the DE-9IM matrix, figure from the DE-9IM Wikipedia page](image)
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

**Directional relation calculus**

Next to the topological relations we also need a calculus to denote directional relations between two spatial fragments. The directions defined by this calculus should be easy to understand and provide the user with enough expressiveness to determine the most common directional relations like “above”, “below”, “left above” etc. Nearly all directional calculi define two types of geometric entities. The first entity is called the reference object and is used as a reference to determine the relation to another object in the 2-dimensional space. The second entity is the object for which we want to determine its directional relation to the reference object. This entity is called the target entity. A directional relation is then interpreted as a binary relation between these two entities and is determined by the reference frame used. This reference frame will map different areas in the (2-dimensional) space to symbols which indicate a certain relation. An example of two reference frames using a centroid model to represent the two spatial entities can be found in figure 7.5. There are many different calculi, each using a different reference frame and a different model to represent the spatial entity. In figure 7.5 Frank’s cone-shaped and projection-based models [31] are depicted which use a centroid representation of the spatial entities. These however are not suitable for our use case as already explained in section 7.1.1.

![Figure 7.5: Cone and projection reference frame of defined by Frank](image)

We therefore need a calculus that uses a bounding box model to represent the spatial entities. This is exactly what Rectangle Algebra [37] allows us to do. It assumes that each spatial entity is represented by a rectangle of which the sides are parallel to the x-axis and y-axis of a Cartesian space. The bounding boxes used in Media Fragments cohere exactly to this definition and thus we can use this calculus to determine the directional relation of two Media Fragments. Rectangular Algebra uses a projection based reference frame, meaning that the 2-dimensional space is divided into different segments using horizontal and vertical lines parallel to the x- and y-axis. The calculus defines nine different segments, one of which is the bounding box of the reference object itself. A graphical representation of the nine segments and their symbolic meaning can be found in figure 7.6. We can see that this calculus does provide us with the most common directional relations used in our daily life. On top of that, the calculus also coincides with our natural interpretation of directional relations as it takes the sides of the bounding boxes into account.
In figure 7.6 we have for example an object B which is classified as being to the "northwest" of the reference object A as the right side of object B is to the left of object A and because the bottom side of object B is higher than the top of object A. If the right side of object B would have been further to the right than the left side of object A, it would have been classified as "north" instead of "northwest" which is also intuitively more correct. The object would overlap in the x-direction and thus appear to be above object A. We can conclude that the Rectangular Algebra calculus does provide us with the most commonly used directions and also closely relates to our intuitive interpretation of the directional relations. We therefore use this calculus to implement the directions relation functions which can be used as a filter. We will however not use the geographic terms like "north", "northwest" etc. but opt to use the more general and commonly used predicates like "above", "left above" etc to indicate the directional relation functions. These are also used in the ontology defined by Kurz et al. and we can thus also use this ontology to identify the directional relationship we want to test in the filter. The table 7.2 contains all eight (basic) directional relations defined in the Rectangular Algebra together with their logical meaning and the according URI used to indicate this relation.

**Spatial accessor and aggregation functions**

Next to spatial relation functions which allows users to check for a certain directional or topological relation between two spatial fragments, SPARQL-MM also defined multiple accessor and aggregation functions which have a slightly different mechanic compared to the relational functions. The spatial accessor functions take only one spatial fragment as input and will return the most common proper\-\es of a bounding box. These can be found in table 7.3. All these functions are basic mathematical equations and each of them is implemented in the altered Triple Pattern Fragments client. The functions are again all identified using the SPARQL-MM ontology. Next to these basic accessor functions, there are two additional accessor functions defined in SPARQL-MM: [http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialFragment](http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialFragment) and [http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#](http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#).
### Chapter 7. Client-Side Media Querying

<table>
<thead>
<tr>
<th>Name</th>
<th>Logical meaning</th>
<th>Relation URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>North(A,B)</td>
<td>fragment A is to the north of fragment B if the top of bounding box B is lower than the bottom of bounding box A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#above">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#above</a></td>
</tr>
<tr>
<td>east(A,B)</td>
<td>fragment A is to the east of fragment B if the left side of bounding box A is to the right of the right side of bounding box B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#rightBeside">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#rightBeside</a></td>
</tr>
<tr>
<td>south(A,B)</td>
<td>fragment A is to the south of fragment B if the top of bounding box A is lower than the bottom of bounding box B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#below">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#below</a></td>
</tr>
<tr>
<td>west(A,B)</td>
<td>fragment A is to the east of fragment B if the right side of bounding box A is to the left of the left side of bounding box B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#leftBeside">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#leftBeside</a></td>
</tr>
<tr>
<td>northEast(A,B)</td>
<td>fragment A is to the northeast of fragment B if the left side of bounding box A is to the right of the right side of bounding box B and the top of bounding box A is lower than the bottom of bounding box B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#rightAbove">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#rightAbove</a></td>
</tr>
<tr>
<td>southEast(A,B)</td>
<td>fragment A is to the southeast of fragment B if the left side of bounding box A is to the right of the right side of bounding box B and the top of bounding box A is lower than the bottom of bounding box B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#rightBelow">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#rightBelow</a></td>
</tr>
<tr>
<td>southWest(A,B)</td>
<td>fragment A is to the southwest of fragment B if the right side of bounding box A is to the left of the left side of bounding box B and the top of bounding box A is lower than the bottom of bounding box B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#leftBelow">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#leftBelow</a></td>
</tr>
<tr>
<td>northWest(A,B)</td>
<td>fragment A is to the northwest of fragment B if the right side of bounding box A is to the left of the left side of bounding box B and the top of bounding box B is lower than the bottom of bounding box A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#leftAbove">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#leftAbove</a></td>
</tr>
</tbody>
</table>

Table 7.2: Directional spatial functions implemented in the Triple Pattern Fragments client
### CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

<table>
<thead>
<tr>
<th>Name</th>
<th>Logical meaning</th>
<th>Accessor URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>area(A)</td>
<td>returns the area of the bounding box defining spatial fragment A. This will be percentual or in pixel values depending on the formatting used in the Media Fragments URI</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#area">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#area</a></td>
</tr>
<tr>
<td>center(A)</td>
<td>returns the center of the bounding box defining spatial fragment A. This will be percentual or in pixel values depending on the formatting used in the Media Fragments URI</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#center">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#center</a></td>
</tr>
<tr>
<td>height(A)</td>
<td>returns the height of the bounding box defining spatial fragment. This will be percentual or in pixel values depending on the formatting used in the Media Fragments URI</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#height">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#height</a></td>
</tr>
<tr>
<td>width(A)</td>
<td>returns the width of the bounding box defining spatial fragment. This will be percentual or in pixel values depending on the formatting used in the Media Fragments URI</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#width">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#width</a></td>
</tr>
<tr>
<td>xy(A)</td>
<td>returns (x,y) coordinate of upper left corner of the bounding box defining spatial fragment as a string. This will be percentual or in pixel values depending on the formatting used in the Media Fragments URI</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#xy">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#xy</a></td>
</tr>
</tbody>
</table>

Table 7.3: Accessor functions for spatial fragments implemented in the Triple Pattern Fragments client

The `hasSpatialFragment` function. The first function will take a spatial fragment URI and return a string representation of this fragment containing the (x,y) position of the upper left corner and the width and height of the fragment. This function can be used in the SELECT statement of a SPARQL query in order to simplify the result and get a nice string representation compared to a full URI. The second function will return true or false depending on whether the given variable contains a spatial fragment. Since these two functions are useful additions to the already existing functionality, we have opted to also implement these in the Triple Pattern Fragments client.

The aggregation functions take two spatial fragments as input and will return a newly created Media Fragments URI depending on the semantics of the function. The `http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialBoundingBox` function takes two spatial fragments as input and returns a new Media Fragment with the same base URI and with a bounding box that entails the bounding boxes of both given fragments, effectively creating one big fragment that encapsulates the two given fragments. The `http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#spatialIntersection` function on the other hand returns a new Media Fragment of which the bounding box is equal to intersection of the bounding boxes of both given fragments. They thus both allow the user to create new Media Fragments through a combination or intersection of two given Media Fragments. These are both also implemented in the Triple Pattern Fragments client.
7.1.2 Temporal accessor, aggregation and relational functions

Just like the spatial fragments, SPARQL-MM also defined accessor, aggregation and relational functions which apply to temporal Media Fragments. The relational functions will again take two (temporal) fragments as input and return a boolean value whether both adhere to a certain relation. These functions are again implemented as a filter. The aggregation and accessor functions are however not implemented as filter functions. The importance of this will be explained in section 7.1.3.

Temporal relational functions

Opposed to the spatial functions, we do not need to find another model to represent temporal Media Fragments as time is a universally used notion for temporal reasoning. We do however again need a calculus which will enable us to calculate temporal relations between two temporal Media Fragments. This model again needs to provide us with the most commonly known temporal relations like “comes after”, “comes before”, etc. in an easily calculable manner. Allen’s interval algebra [28] is the de facto standard used for this kind of reasoning. It introduces an interval-based temporal logic in which thirteen basic relations between time intervals are defined. These thirteen relations are also the only possible relations that two definite time intervals can have to each other. A visual representation of the relational functions for temporal intervals can be found in figure 7.7.
This calculus is also used by SPARQL-MM and each of the thirteen relations have been defined in the SPARQL-MM ontology. We therefore again refrain from defining our own ontology and use the function identifiers from the ontology defined by Kurz et al. Table 7.4 contains the thirteen temporal relations, together with their URI and a textual explanation of their logical meaning.

### Temporal accessor and aggregation functions

Just like the spatial fragments, the temporal accessor and aggregation functions are not defined as filters but as functions in the `SELECT` clause of the SPARQL-MM queries. The accessor functions will take one temporal Media Fragment as input and return the most common properties of temporal intervals like the start, end, duration etc. All these functions are the result of simple mathematical equations and each of them are again defined using the SPARQL-MM ontology defined by Kurz et al. Table 7.5 shows a list of all temporal accessor functions supported by the altered Triple Pattern Fragments client.
<table>
<thead>
<tr>
<th>Name</th>
<th>Logical meaning</th>
<th>Relation URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>precedes(A,B)</td>
<td>temporal Media Fragment A precedes temporal Media Fragment B if temporal Media Fragment A starts and ends before the start of temporal Media Fragment B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#precedes">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#precedes</a></td>
</tr>
<tr>
<td>meets(A,B)</td>
<td>temporal Media Fragment A meets temporal Media Fragment B if the end time of temporal Media Fragment A is equal to the start time of temporal Media Fragment B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalMeets">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalMeets</a></td>
</tr>
<tr>
<td>overlaps(A,B)</td>
<td>temporal Media Fragment A overlaps temporal Media Fragment B if A starts before B and ends after the start of B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalOverlaps">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalOverlaps</a></td>
</tr>
<tr>
<td>finishedBy(A,B)</td>
<td>temporal Media Fragment A is finished by temporal Media Fragment B if A and B end at the same time and A starts before B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#finishedBy">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#finishedBy</a></td>
</tr>
<tr>
<td>contains(A,B)</td>
<td>temporal Media Fragment A contains temporal Media Fragment B if it starts before B and ends after B</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalContains">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalContains</a></td>
</tr>
<tr>
<td>starts(A,B)</td>
<td>temporal Media Fragment A starts temporal Media Fragment B if A starts at the same time as B and ends before B ends</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#starts">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#starts</a></td>
</tr>
<tr>
<td>equals(A,B)</td>
<td>temporal Media Fragment A equals temporal Media Fragment B if A and B start and stop at the same time</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalEquals">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#temporalEquals</a></td>
</tr>
<tr>
<td>started by(A,B)</td>
<td>temporal Media Fragment A is started by temporal Media Fragment B if A and B start at the same time and A ends after B ends</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#startedBy">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#startedBy</a></td>
</tr>
<tr>
<td>during(A,B)</td>
<td>temporal Media Fragment A is during temporal Media Fragment B if B starts before A starts and B ends after A ends</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#during">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#during</a></td>
</tr>
<tr>
<td>finishes(A,B)</td>
<td>temporal Media Fragment A finishes temporal Media Fragment B if A and B stop at the same time and A started after B started</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#finishes">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#finishes</a></td>
</tr>
<tr>
<td>overlapped by(A,B)</td>
<td>temporal Media Fragment A is overlapped by temporal Media Fragment B if B starts before A and B ends after A starts and before A ends</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#overlappedBy">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#overlappedBy</a></td>
</tr>
<tr>
<td>met by(A,B)</td>
<td>temporal Media Fragment A is met by temporal Media Fragment B if the end time of temporal Media Fragment B is equal to the start time of temporal Media Fragment A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#metBy">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#metBy</a></td>
</tr>
<tr>
<td>preceded by(A,B)</td>
<td>temporal Media Fragment A is preceded by temporal Media Fragment B if temporal Media Fragment B starts and ends before the start of temporal Media Fragment A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#precededBy">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#precededBy</a></td>
</tr>
</tbody>
</table>

Table 7.4: Temporal relation functions implemented in the Triple Pattern Fragments client
### CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

<table>
<thead>
<tr>
<th>Name</th>
<th>Logical meaning</th>
<th>Accessor URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration(A)</td>
<td>returns the duration the temporal interval given by temporal Media Fragment A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionduration">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionduration</a></td>
</tr>
<tr>
<td>end(A)</td>
<td>returns the end time of the temporal interval given by temporal Media Fragment A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionend">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionend</a></td>
</tr>
<tr>
<td>start(A)</td>
<td>returns the start time of the temporal interval given by temporal Media Fragment A</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionstart">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionstart</a></td>
</tr>
<tr>
<td>temporalFragment(A)</td>
<td>returns a string representation of the temporal Media Fragment A (t=start,end)</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalFragment">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalFragment</a></td>
</tr>
<tr>
<td>hasTemporalFragment(A)</td>
<td>return a boolean value indicating whether the Media Fragment A contains a temporal Media Fragment or not.</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionhastemporalFragment">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functionhastemporalFragment</a></td>
</tr>
</tbody>
</table>

**Table 7.5: Accessor functions for temporal fragments implemented in the Triple Pattern Fragments client**

<table>
<thead>
<tr>
<th>Name</th>
<th>Logical meaning</th>
<th>Aggregation URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalIntermediate(A,B)</td>
<td>creates a new Media Fragments URI with temporal fragment t=min(A.end, B.end), max(A.start, B.start) if there is no intersection between A and B. Otherwise null is returned</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalIntermediate">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalIntermediate</a></td>
</tr>
<tr>
<td>temporalBoundingBox(A,B)</td>
<td>returns a new Media Fragments URI with temporal fragment t=min(A.start, B.start), max(A.end, B.end)</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalBoundingBox">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalBoundingBox</a></td>
</tr>
<tr>
<td>temporalIntersection(A,B)</td>
<td>returns a new Media Fragments URI with temporal fragment t=max(A.start, B.start), min(A.end, B.end) if intersection exists. Otherwise null is returned.</td>
<td><a href="http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalIntersection">http://linkedmultimedia.org/sparql-mm/ns/2.0.0/functiontemporalIntersection</a></td>
</tr>
</tbody>
</table>

**Table 7.6: Aggregation functions for temporal fragments implemented in the Triple Pattern Fragments client**

Next to the accessor functions, SPARQL-MM also defined three temporal aggregation functions which will take two temporal Media Fragments as input and create a new temporal fragment out of them. Each of these functions, together with their logical meaning and URI are enlisted in table 7.6.

We thus have implemented all temporal and spatial functions of SPARQL-MM in the Triple Pattern Fragments client. One remark has however need to be made. Because the performance evaluation is performed on spatial fragments and no temporal fragments, the temporal functions have been tested less extensively compared to the spatial functions.

57
7.1.3 Using the SPARQL-MM spatial functions with Triple Pattern Fragments

The previous sections have shown that we have provided the Triple Pattern Fragments server with nearly the same functionality as the SPARQL-MM extension for Sesame Triple Stores. For now, the spatial functions will only be supported for fragments that have the same representation format as there is currently no support for converting pixel coordinates to percentual coordinates or vice versa. This is done because of time constraints, the lack of standardisation for representing the dimensions of image or video files and the lack of dimensional information in the used dataset. We therefore have opted to not implement this functionality at this time and focus on creating optimisations for the Triple Pattern Fragments client as this will provide more value to our solution. By using the same ontology for defining the functions, we have achieved interoperability and SPARQL-MM queries can thus integrally be used with the altered Triple Pattern Fragments client, provided the previous limitation. The spatial and temporal relational functions are all implemented as a filter. However, as filters cause for a performance degradation in the Triple Pattern Fragments client (section 5.3), these functions will also cause for a diminished query execution speed. In the following section we will try to mitigate this by using the URI structure imposed by the Media Fragments URI standard. The accessor and aggregation functions both reside in the SELECT clause of the query and take one or more variables as input. This because the functions have a different purpose compared to the relational functions. Instead of checking for a relation which can influence the results while executing the query, the accessor and aggregation functions will access certain properties of the fragments once the query has been fully executed and bindings have been found for the given variables. In order to facilitate this functionality, the Triple Pattern Fragments client has been extended to allow functions to be defined in the SELECT clause of a query. The client will first fully execute the query like normal and search for solutions of the variable(s) defined in the function. Once the bindings are found, they are written out to the user. It is at this point that the client will check whether a function is defined in the SELECT clause and if so, the function will be called and the resulting value will be written as result. The functions can thus be implemented in a non-blocking way. This works due to the fact that the aggregation and accessor functions both rely on the current binding of the variable to compute its result without the need for any other possible bindings of the variable. The accessor and aggregation functions will thus have no negative effect on the execution speed of the queries and thus have less of an impact compared to the relational functions. To conclude this section, some examples of possible SPARQL-MM queries using spatial reasoning are given in figure 7.8 to illustrate the working of the spatial accessor, aggregation and relational functions.
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?subject WHERE {
  ?subject ma:hasFragment ?f1;
  ma:hasFragment ?f2.

  ?f1 dc:description "Dog".
  ?f2 dc:description "Person".

  FILTER mm:leftBeside(?f1, ?f2)
}

Finding images where a dog is standing to the left of a person

PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?subject WHERE {
  ?subject ma:hasFragment ?f1;
  ma:hasFragment ?f2;
  ma:hasFragment ?f3.

  ?f1 dc:description "Person".
  ?f2 dc:description "Car".
  ?f3 dc:description "Tree".

  FILTER mm:leftBeside(?f1, ?f2)
  FILTER mm:rightBeside(?2, ?f3)
}

Finding images where a car is standing between a person and a tree

PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?subject ( mm:area (?f1) as ?area ) WHERE {
  ?subject ma:hasFragment ?f1.
  ?f1 dc:description "sky".
}

Finding images that have a sky and returning the area of the sky in the picture

Figure 7.8: Example queries
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

7.2 Triple Pattern Fragments client optimisations

In the previous section we have considered the different possible calculi for implementing the SPARQL-MM functionality in the Triple Pattern Fragments client while using the same function identifiers. This way, we can transparently use SPARQL-MM queries in the Triple Pattern Fragments framework. We do however have concluded that a lot of the functionality provided by SPARQL-MM has to be implemented as filters. This is a disadvantage as filters are a bottleneck for the Triple Pattern Fragments framework. Therefore, we should try to counter this and improve the execution speed of the SPARQL-MM queries where ever possible. In order to do so, we can make use of the Media Fragments URI standard to provide the client with some additional logic while solving the queries.

7.2.1 Deriving bindings locally

The Media Fragments URI standard has the property that a fragment of a certain source media file always needs to have the same base URI as its source media file. This means that when the client receives a solution mapping for a fragment, it can extract the base URI and create a binding for the base media file without having to consult the Triple Pattern Fragments server. This effectively diminishes the number of requests to the server which on its turn will lower its CPU usage. On top of that, the solution will also provide a mitigation for lower connection speeds between the client and the server. The client will be able to solve some triple patterns locally which is faster than having to send a request to the server and waiting for a response before being able to create the solution mapping. The slower the internet connection between the client and server, the bigger the benefit will become. If we for example look at the query in listing 7.1 the client will start by evaluating \texttt{tp}_3 as this triple pattern has the least estimated number of matches. Now, for every possible solution mapping for \texttt{?f1}, the client can automatically extract the base URI from the Media Fragments URI given by the solution mapping. After this, the client checks whether there is a triple pattern where an (unbound) subject has the variable \texttt{?f1} as object and the predicate \texttt{ma:hasFragment} to define the relation between the two. If this is the case, the client knows that the subject must be equal to the base URI of the solution mapping and it can simply create a binding for the subject variable locally before resuming the normal execution. If we for example receive the following solution mapping: 
\[
\lambda_1 = \{ \texttt{?f1} \rightarrow \texttt{http://www.example.org/image#xywh=percent:20,20,5,5} \},
\]
the client can automatically extract the solution mapping \[
\lambda_2 = \{ \texttt{?f1} \rightarrow \texttt{http://www.example.org/image#xywh=percent:20,20,5,5, ?image \rightarrow \texttt{http://www.example.org/image}} \}
\]
as \texttt{?f1} is a fragment of \texttt{?image}. The client can thus immediately remove two triple patterns (\texttt{tp}_1 and \texttt{tp}_3) from its sub-BGP before passing it on to the next BasicGraphPattern Iterator instance.
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?f1 ?f2 WHERE {
  ?image ma:hasFragment ?f1; ma:hasFragment ?f2. # ± 40,000 matches (tp1)
  ?f1 dc:description "person". # ± 5,000 matches (tp3)
  ?f2 dc:description "car". # ± 6,000 matches (tp4)

  FILTER mm:leftBeside(?f1,?f2)
}

Listing 7.1: Example query for finding images with two fragments

7.2.2 Query rewriting

Creating solution mappings locally using the structure of the Media Fragments URI standard is one aspect we can use to optimise the query execution speed. There is however one other aspect where the structure of the Media Fragments URI standard can aid us in an optimal query execution. This time in the form of query rewriting. Notice that up until now, nearly all example queries contained triple patterns denoting that a certain image contains certain media fragments, like $tp_1$ and $tp_2$ in listing 7.1. This is done purposefully as will be explained by the following example. If we take a look at listing 7.2, the query will provide us with the same results as the query in listing 7.1. We will get the URI of fragments where a person stands to the left of a car. This query will also seem more logical to most users as it only describes the needed constraints: “We need to find fragments in which a person is standing to the left of a car. Nothing more, nothing less”. The added triple patterns in listing 7.1 will therefore seem to not add any functionality as the filter already will check if both fragments are from the same media file. The difference however is the way the query will be executed by the client. In listing 7.2, the client has to try every possible combination of the 5,000 fragments of a person and 6,000 fragments containing a car and apply the filter on each pair. Needless to say, this results in a huge amount of solution mapping pairs and filter executions which is far from optimal.

PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?f1 ?f2 WHERE {
  ?f1 dc:description "person". # ± 5,000 matches (tp1)
  ?f2 dc:description "car". # ± 6,000 matches (tp2)

  FILTER mm:leftBeside(?f1,?f2)
}

Listing 7.2: Example query for finding images with two fragments with less specificity
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

If we however look at the query execution of the query in listing 7.1, we can see a clear benefit by adding the extra triple patterns $tp_1$ and $tp_2$. The execution will again start the same. The Triple Pattern Fragments client will start by selecting $tp_3$, remove the triple pattern from the sub-BGP and create iterators for each possible solution mapping. If we for example have the solution mapping $\lambda_1 = \{?f1 \mapsto \text{http://www.example.org/image#xywh=percent:10,10,20,20}\}$, the resulting sub-BGP will be as shown in listing 7.3. We now have a triple pattern with only one match ($tp_1$). This one will be selected by the Triple Pattern Fragments client and the solution mapping will be applied. Now, since media files typically have a limited amount of fragments defined, there will be only a small amount of solution mappings for ?f2 that have the base URI http://www.example.org/image. Each of these mappings will then be checked whether it contains the right subject and only when this is this case, will the filter be executed. We can clearly see that this execution order is far more optimal compared to the execution for listing 7.2. By adding the extra triple patterns, we essentially implement the part of the filter that checks whether both fragments are from the same media file. Thus making the query more selective. This combined with the fact that each media fragment can only have one base media file from which it originates, means that we essentially do an early filtering on the possible combinations of fragments.

```sparql
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?f1 ?f2 WHERE {
?image ma:hasFragment <http://www.example.org/image#xywh=percent:10,10,20,20>;# 1 match
ma:hasFragment ?f2. # 40,000 matches ($tp_2$)

?f2 dc:description "car". # 6,000 matches ($tp_4$)

FILTER mm:leftBeside(?f1,?f2)
}
```

Listing 7.3: Example query for finding images with two fragments with solution mapping for '?f1'

### 7.3 Implementation details

In the previous sections, the main models, concepts and ideas that were implemented in the Triple Pattern Fragments framework were discussed generally. In this section, the implementation details are elaborated upon. First, the implementation of the SPARQL-MM functions is briefly discussed. After this, the alterations that needed to be done to do client-side binding are given, followed by a short explanation of the query rewriting function and its position in the Triple Pattern Fragments client. All alterations are done on the JavaScript implementation of the client as this is the most actively developed implementation. In the following chapters, both approaches will be explained and their effects are discussed for the case of media fragment querying.
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

7.3.1 Implementing SPARQL-MM functions

When working with SPARQL-MM functions we will be reasoning about media fragments and their relation to each other and their source media. Even though we could simply use the Media Fragments URI and string operations to extract the required information, it is far easier and faster to represent media fragments as objects. Therefore, a module called MediaFragmentExtractor.js has been created. This module has two exported methods toObject(URI) and toString(MediaFragment). The toObject() function will take a URI as argument. Before further parsing, it will first check if the given URI is a valid Media Fragments URI. After this, a MediaFragment object will be created which on its turn will be returned to the caller. This object and the extracted information it contains can then be used in further calculations when relational, accessor and aggregation functions are calculated. The MediaFragmentExtractor.js module also provides the client with a toString() function which takes a media fragment object as input and will create a valid Media Fragments URI representing the object. This latter function is used in the spatial and temporal accessor functions.

Next to the module for creating a media fragment object, there are two modules defined for implementing the SPARQL-MM functions. The first module (SpatialFunctions.js) will contain all calculations for the spatial relational, accessor and aggregation functions. The second module (TemporalFunctions.js) will implement all temporal functions that are supported by SPARQL-MM. An important remark that needs to be made is that currently, the spatial functions only work when the same representation is used. This means that we can only calculate relations of fragments that both use a percentual or pixel values to indicate the spatial position. Mixing both will cause for inaccurate results. These modules are both called by the SparqlExpressionEvaluator.js module already present in the standard TPF client. This module takes a filter or other function expression as input and will present the caller with the desired results. We thus expanded the SparqlExpressionEvaluator to also recognise the SPARQL-MM function URIs and call the right functions in one of the previously mentioned modules.

We still have one aspect that needs to be altered before we can fully support SPARQL-MM functions which is the use of functions the SELECT clause of the query. Without this, we could only use the filter functions of SPARQL-MM (which are the temporal and spatial relational functions). The standard TPF client only supports variables to be defined in the SELECT clause and therefore can not support the spatial and temporal accessor and aggregation functions. In order to allow for this support, the SparqlIterator.js module has been updated so that the SELECT projection function first checks whether the variable is a function or not. If so, a new SparqlExpressionEvaluator is created and the function is passed, together with the bindings of its input values. The evaluator will then provide the same functionality as when it is called from a filter context. The result of the function is then converted to a string and displayed to the user.

We thus have provided the TPF client with SPARQL-MM functionality by adding three modules and altering two already existing ones to call the newly implemented functionality. In order to improve the
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

calculation performance, each fragment URI is converted into an object with numeric values as fields. This way, we only have to do a single string parsing operation per fragment instead of parsing every time we need a certain property of the media fragment.

7.3.2 Deriving bindings locally

For deriving the bindings locally, a new version of the BasicGraphPatternIterator has been implemented, called the OptimizedGraphPatternIterator. This iterator is nearly identical to the BasicGraphPatternIterator and will only provide one additional functionality. Every time new bindings are retrieved from the TPF server, the OptimizedGraphPatternIterator will check if the binding contains a Media Fragments URI. If so, it will traverse the current sub-BGP and try to find triple patterns where the variable with the Media Fragments URI as binding appears as an object for the predicate http://www.w3.org/ns/ma-ont#hasFragment. If one is found, the iterator will extract the base URI of the Media Fragments URI and bind this value to the subject of the corresponding triple pattern, thus solving two triple patterns in one step. There has been opted to use a separate BGP iterator as we want the user to be able to manually enable or disable the optimisations. If the client is passed an \texttt{--o} flag, the optimised BGP iterator will be called during the execution, otherwise the regular iterator will be used. The pseudocode of the OptimizedGraphPatternIterator can be found in listing 7.4. The only difference with the pseudocode in listing 5.7 is the call to the updateBindings function on line 9. This function will take the current bindings and BGP and will try to add extra bindings to the mapping. If an extra mapping is found, this triple pattern is also removed from the BGP which means that we need less GET requests to the server. The pseudocode for the updateBindings function can be found in listing 7.5.
7.3.3 Query rewriting

The last alteration that has been done to the TPF client is rewriting the query in order to obtain a more optimal query execution order. The client will first perform query rewriting before starting the query execution. In order to do so, the SparqlIterator will call the insertTriplePatterns function which will take the entire BGP and the filters as input. This function will then insert the needed triple patterns to ensure the faster execution of the query. After this, the execution is started as normal, only with a
more optimal query. The pseudocode for the rewrite can be found in listing 7.6. It will check for each
filter defined in the query whether a SPARQL-MM function is used for this particular filter. This gives
us an indication that both arguments in the function are media fragments and that we can perform the
optimisation for these arguments. Next, the BGP is checked whether the user has already defined the
optimisation himself (in which case the client does not have to do anything) or if at least one of the
arguments of the filter is already present in the BGP in combination with the http://www.w3.org/ns/
ma-ont#hasFragment predicate. If so, the client can extract the corresponding subject variable and add
a new triple pattern with the same subject stating that it also has the second argument of the filter as
media fragment. If none are found, the client will simply create a dummy variable and add two new triple
patterns to the BGP indicating that the dummy variable has two fragments. This rewrite will not alter the
semantics of the query as the filter functions also check if the passed values both originate from the same
media file. Adding these extra triple patterns will thus not lead to less or different results compared to the
original one. For clarification, a sample rewrite is given in listings 7.7 and 7.8.

Listing 7.6: Pseudocode for query rewriting function

```java
Function insertTriplePatterns() {
  Data: basic graph pattern, filters;
  Output: updated basic graph pattern with extra triple patterns
  foreach filter function f_i ∈ filters do
    arguments_i = []
    if f_i SPARQL-MM function then
      arguments_i ← arguments of f_i;
      noneFound = false
      for argument arg_j in arguments_i do
        for triple pattern tp_k in bgp do
          if tp_k has object arg_j and tp_k has predicate http://www.w3.org/ns/ma-ont#hasFragment then
            noneFound = false
          end
        end
      end
      if noneFound then
        for argument arg_j in arguments_i do
          append (?subject, http://www.w3.org/ns/ma-ont#hasFragment, arg_j) to bgp
        end
      else
        subject ← subject of triple tp_k in bgp where arg_j found = true
        for argument arg_j in arguments_i do
          if arg_j not found then append (subject, http://www.w3.org/ns/ma-ont#hasFragment, arg_j) to bgp
        end
      end
    end
  end
}
```
CHAPTER 7. CLIENT-SIDE MEDIA QUERYING

```sparql
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?f1 ?f2 WHERE {
    ?f1 dc:description "person".
    ?f2 dc:description "car".
    ?f3 dc:description "Person".
    FILTER mm:leftBeside(?f1, ?f2)
    FILTER mm:leftBeside(?f2, ?f3)
}
```

Listing 7.7: SPARQL-MM query before rewrite

```sparql
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ma: <http://www.w3.org/ns/ma-ont#>

SELECT ?f1 ?f2 WHERE {
    ?file0 ma:hasFragment ?f1;
    ma:hasFragment ?f2;
    ma:hasFragment ?f3.
    ?f1 dc:description "person".
    ?f2 dc:description "car".
    ?f3 dc:description "Person".
    FILTER mm:leftBeside(?f1, ?f2)
    FILTER mm:leftBeside(?f2, ?f3)
}
```

Listing 7.8: SPARQL-MM query after rewrite

### 7.3.4 Conclusion

We were able to provide the TPF client with nearly all functionality that SPARQL-MM provides. We do however still lack the functionality to convert pixel values to percentile values in the spatial domain. Next to this, the TPF client has also been given two optimisations which will result in a faster query execution for SPARQL-MM queries compared to the standard implementation. In order to give the users the choice whether they want the optimisation, a new optional parameter is added to the JavaScript client (-o). This parameter will cause the client to both rewrite the query where possible and calculate bindings locally during execution. The SPARQL-MM functionality will always be present, even if the new optional parameter is not passed.
CHAPTER 8
Performance evaluations

Now that we have implemented the SPARQL-MM functionality in the TPF client together with some optimisations, we need to investigate how our solution compares to the SPARQL-MM implementation in Apache Marmotta which will evaluate the queries all server-side. Therefore, a big enough dataset is needed to be able to fully test the capabilities of both solutions. In the following section, the choice of dataset will be elaborated upon together with the consequences of using Media Fragments URIs on the RDF data structure. After this, the experiment setup is explained followed by the results of two different experiments which will evaluate the influence of query rewriting and local binding updates compared to the native TPF client. These two experiments give us an insight of which optimisation will create the biggest benefit. After this, a final experiment will compare the execution of SPARQL-MM queries with Apache Marmotta, the unaltered TPF client and the optimised TPF client from which we can prove or disprove our hypotheses defined in section 6.

8.1 Creating the dataset

In order to test the performance of the (optimised) Triple Pattern Fragments implementation of the SPARQL-MM functions, we first have to find a suitable dataset on which we can perform queries. This thesis has two requirements regarding the dataset. First of all, a fine-grained media search functionality (like querying for images where a person stands to the left of a car) will be provided to the Triple Pattern Fragments client. Therefore, the dataset needs to have media files where different subjects and/or objects are identified, possibly together with some information about the objects that are depicted in the media file. Secondly, the Media Fragments URI standard will be used in order to optimise the query execution speed of the TPF client. Therefore, we have to construct a dataset in which the Media Fragments URI structure is used to identify the objects that are depicted inside the media files. There are a lot of datasets containing annotated media files. The majority however only classifies the media files as a whole and will add tags or annotations at the level of the whole image or video. There are only few openly available datasets that provide media files together with object bounding boxes and classifications. The two most promising openly available datasets were YouTube-BoundingBoxes [38] and Open Images Dataset [39]. Both these datasets are mainly used for machine learning competitions. This does not remove the fact that the train and validation sets of these datasets can be used to create a database of media files with media fragments.
CHAPTER 8. PERFORMANCE EVALUATIONS

YouTube-BoundingBoxes however poses a problem as the bounding boxes and objects are annotated at millisecond level. This means that the bounding boxes created are very narrow and in very small time periods. An attempt was made to aggregate the bounding boxes over a longer time period to obtain less time fragments and less strict spatial fragments. This was proven difficult to do as it is infeasible to algorithmically determine the amount of bounding boxes to aggregate. In some videos the objects move quite a lot and aggregating the bounding boxes over a longer timespan would cause the aggregated bounding box to cover nearly half of the pixels, meaning that we can not provide enough precision in the spatial domain to query for spatial relation between objects in the video. On the other hand, if some objects are nearly static and a too short time period is chosen, we end up with too narrow bounding boxes and minor displacements that could influence the querying for fragments in the video that are in the same position but only in a later time period. Therefore, there has been opted to omit this dataset and only use the Open Images Dataset which contains images and bounding boxes for objects in the images. Since this thesis will mainly focus on the performance differences between SPARQL-MM in a centralised implementation and the (optimised) SPARQL-MM enabled TPF, we can focus on only using the spatial domain for identifying media fragments. We thus have no problem omitting the video dataset.

Figure 8.1: Distribution of classes in Open Images Validation Dataset, figure from Open Images Dataset [39]

The Open Images Dataset contains around 9 million images that have image-level annotations and object bounding boxes which identify fragments in the images. Just like any dataset that is used for machine learning, the set is divided into 3 subsets: the training set, the validation set and the test set. Each set has a different amount of images and bounding boxes defined and in each of the images the bounding boxes were created and annotated by humans. This way, we can be sure that the bounding boxes are precise enough and have the correct class recognised. In table 8.1, the distribution of the number of images and bounding boxes can be found. For this thesis, there has been chosen to use the validation set as this already contains around 41 620 images which is sufficient to emulate a medium sized image hosting solution without requiring very powerful server infrastructure to perform the experiments. On top of that, there are 204 621 fragments defined over these 41 620 images so each image has on average 5 fragments defined. This means that we have a good distribution of media fragments and will thus have
CHAPTER 8. PERFORMANCE EVALUATIONS

<table>
<thead>
<tr>
<th></th>
<th>Train</th>
<th>Validation</th>
<th>Test</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Images</td>
<td>1,743,042</td>
<td>41,620</td>
<td>125,436</td>
<td>-</td>
</tr>
<tr>
<td>Boxes</td>
<td>14,610,229</td>
<td>204,621</td>
<td>625,282</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 8.1: Number of images and bounding boxes for the Open Images Dataset

many results when querying for images which is important to test the performance of the system. On
top of that, there are 600 different classes defined and recognised in the dataset, each defined by their id
in the Google Freebase database [40] which allows us to immediately link the fragments to concepts on
the Semantic Web. This will aid us in creating a set of queries with varying specificity and thus creating a
balanced set of queries to emulate normal behaviour of a lot of simultaneous users where one could be
searching for more specific images while the other is using broader search terms. To aid this, the Open
Images Dataset has also published the class distribution for each dataset that shows which classes are
most commonly used which can be found in figure 8.1.

The validation dataset consists of three different csv files. The first one, validation-images.csv con-
tains a list of all images where each image has a unique identifier and an URI to the image. Every image
used in this dataset is hosted on Flickr and has a CC BY 2.0 license, meaning that the images are free to
use for our purposes. Next to the list of images, there is also a validation-annotations-bbox.csv file
enlisting all bounding boxes and annotations in the validation dataset and having next to some other less
important columns, the following columns:

- ImageId: Unique Image Identifier (same as in validation-images.csv)
- Source: Image source
- LabelName: The label of the class depicted in the bounding box which acts as an identifier for Google
  Freebase
- Xmin: X-axis minimum of bounding box
- Xmax: X-axis maximum of bounding box
- Ymin: Y-axis minimum of bounding box
- Ymax: Y-axis maximum of bounding box

The last csv file, class-descriptions.csv, contains textual representations for each labelName in the
bounding boxes csv file. Note that this structure and encoding does not allow us to use SPARQL queries
and we need to convert this information into a format that can be handled by the Apache Marmotta triple
store and by the TPF server. Therefore, the images and bounding boxes are all transformed into an RDF
document which can be imported by both technologies. The structure of the RDF document is as follows:
for every image I in the validation dataset, the following entry is added to the document:
Subsequently, from each bounding box in validation-annotations-bbox.csv, a Media Fragments URI is created using the base URI of the corresponding image and using the X and Y values defined with the bounding box. These values are percentual values and thus the Media Fragments URIs that are created follow the percentual syntax defined for spatial fragments. Each Media Fragments URI is then added to the RDF document and the object that is represented by this fragment is added as a literal. Note that we could have used the Google Freebase URI to indicate the resource depicted in the fragment but since we will not really make use of this extra information, there has been chosen to include the literal value of the subject depicted next to the URI to the Google Freebase entry and use this later on when querying the dataset. Each fragment will result in the following entry in the RDF document:

<URI of media fragment> a http://www.w3.org/ns/ma-ont#MediaFragment;
  http://purl.org/dc/elements/1.1/description "Subject shown in the fragment";
  http://xmlns.com/foaf/0.1/depicts <FreeBase URI>.

As one may have already noticed, the Ontology for Media Resources is used and enables us to define which resources are images and which resources are media fragments. This will be used later on in queries when querying for images with certain fragments in them. The spatial positioning of the fragments inside the images is, as the Media Fragments URI standard dictates, encapsulated inside the URI of the Media Fragment Resource. The RDF document is created using a Python script which will parse all the input csv files and create a file containing all the images and fragments in a valid Turtle Syntax. In the following snippet, a small example is given of two images and their corresponding fragments.

@prefix ns1: <http://purl.org/dc/elements/1.1/> .
@prefix ns2: <http://xmlns.com/foaf/0.1/> .
@prefix ns3: <http://www.w3.org/ns/ma-ont#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<https://c1.staticflickr.com/1/10/11682923_79c2315639_o.jpg> a ns2:Image ;
  ns1:creator "Rocco Lucia" ;
  ns1:title "[What do you expect from Windows?]" ;
  ns3:hasFragment <https://c1.staticflickr.com/1/10/11682923_79c2315639_o.jpg#xywh=percent:1,0,99,100> .

<https://c1.staticflickr.com/1/10/11682923_79c2315639_o.jpg#xywh=percent:1,0,99,100> a ns3 :MediaFragment ;
  ns1:description "Computer monitor" ;
ns2:depicts <http://rdf.freebase.com/ns/m.02522> .

<https://c1.staticflickr.com/1/119/299164606_39e9159460_o.jpg> a ns2:Image ;
ns1:creator "Alex Indigo" ;
ns1:title "Dragon boat" ;
ns3:hasFragment <https://c1.staticflickr.com/1/119/299164606_39e9159460_o.jpg#xywh=percent:12,38,63,30>,
<https://c1.staticflickr.com/1/119/299164606_39e9159460_o.jpg#xywh=percent:5,50,88,30>,

<https://c1.staticflickr.com/1/119/299164606_39e9159460_o.jpg#xywh=percent:12,38,63,30> a
ns3:MediaFragment ;
ns1:description "Person" ;
ns2:depicts <http://rdf.freebase.com/ns/m.01g3l7> .

<https://c1.staticflickr.com/1/119/299164606_39e9159460_o.jpg#xywh=percent:5,50,88,30> a
ns3:MediaFragment ;
ns1:description "Canoe" ;
ns2:depicts <http://rdf.freebase.com/ns/m.0ph39> .

<https://c1.staticflickr.com/1/119/299164606_39e9159460_o.jpg#xywh=percent:8,49,55,21> a
ns3:MediaFragment ;
ns1:description "Paddle" ;
ns2:depicts <http://rdf.freebase.com/ns/m.014y4n> .

We thus have a dataset containing more than 40 000 images and over 200 000 media fragments following
the Media Fragments URI standard. These images and fragments are each annotated with extra information
like creator, title, description etc. as these are the most common elements used when searching for
images on the web. The formatting of the dataset is also directly compatible with SPARQL-MM and thus
can be integrally imported by Apache Marmotta.

8.2 Experiment setup

In order to be fully able to test the capabilities of the Apache Marmotta and TPF solution, we need to
create a lot of clients which will query the server at the same time. It is important that the clients querying
the server do not get bottlenecked by the machine they are running on. In the case of Apache Marmotta,
we would be able to run 100 clients simultaneously on one machine without overloading its CPU since
all the calculation are done on the server. With TPF however, most of the logic and calculation reside
at client side. This means that we have a higher CPU usage at the client machine. Having 100 clients
simultaneously solving a query on one machine could in this case cause for CPU bottlenecks which could
influence our results. Therefore, the experiment exists of six identical machines hosted on the Virtual
Wall of Ghent University. Each machine has a 2x Quad core Intel E5520 (2.2GHz) CPU, 12GB RAM and a 2 - 4 gigabits NIC and are all connected to each other through a local network. One of the machines will be used as a dedicated server which will only run the Apache Marmotta or TPF server behind an NGINX caching proxy server. The other machines will all create clients and query the server at exactly the same time. One of these querying machines will also be responsible for synchronising the different machines, retrieving the results and parsing them. With this setup we can create up to 200 simultaneous clients which will query the server at exactly the same time without introducing bottlenecks at client side, thus simulating a very big user base. The Apache Marmotta server is deployed in Apache Tomcat using the standard configuration and uses a PostgreSQL database to store the dataset. The TPF server will be run with 6 daemon workers.

The testing framework is written in Python and will take a pickle file containing an array of query strings as input. The monitoring machine will create an increasing amount of clients which are all assigned a random query from the pickle file. We do however want that for every execution of the experiment, the same queries are selected in the same order. Therefore, the random selection of queries is given a static seed which will ensure us of the same execution every time.

The queries that are passed to the experiment will all be slightly modified versions of the same queries. This to be able to fully test the influence of the implemented optimisations of the TPF client. More specifically, there are three sets of queries defined. The first set will follow the structure of listing 7.2. They will thus all rely on the filter to determine if the two fragments are from the same source media. This set is used for determining the influence of query rewriting. The second set will only contain optimal queries following the structure of listing 7.1. By using this set, we eliminate the influence of the query rewriting in the optimal client and can solely test the influence of the local binding updates. The final set will contain all the queries from set 1 and set 2. This set is used to do a final comparison between SPARQL-MM in Apache Marmotta, the standard TPF SPARQL-MM query execution and the optimised TPF SPARQL-MM query execution. Each query of each set will always be implementing a filter and each filter function of the spatial relational functions is used at least once in each set. This is done because the filters of SPARQL-MM provide the most frequently used functionality. The core functionality of advanced querying is to be able to find fragments that have a certain relation to each other (x to the left of y). These are all implemented as filters in SPARQL-MM and TPF. On top of that, there is also the fact that filters have a slower execution speed in TPF compared to queries without filters. We therefore focus on the worst case scenario to compare the execution speeds. By this, we thus evaluate the upper bound and each real life combination of queries that are used by the users will behave at least the same or even better in TPF.

When creating the queries, the distribution of classes of our dataset has been consulted. This shows us that “person” is by far the most common class. We will also search for fragments with cars in them as this is a parent of the “auto part” class in figure 8.1 and thus also has a high occurrence in the dataset.
Next to these, fragments with the class “tree” and “animal” are also used in queries as these have a lower occurrence than the previous ones. These will therefore cause for less results and less intensive queries. Each query will however contain at least one fragment with a person and the majority will contain both a fragment with a person and a fragment with a car. The reasoning behind this is that the more popular a class is, the higher the chance is for users to search for fragments with these classes. On top of that, the more common classes will cause for more results and thus more computationally intensive queries, thus again evaluating the worst case scenario.

In conclusion, all of the three sets contain around 47 queries, each with at least one fragment containing a person. Next to this, each spatial filter function defined in SPARQL-MM is used at least once in each set and multiple versions of the same query are also inserted, each with a different limit on the number of results to return. This again because the most common queries will be used by the most number of people simultaneously.

The experiments will all measure the following values: mean execution time of the query, mean CPU usage of the server, mean bandwidth usage, the number of clients that were rejected or were not able to solve the query and the total number of requests done to the server. The mean execution time will measure the time for the client to fully execute the query and return the values. The mean bandwidth will be measured on the NGINX caching proxy server and will contain the bandwidth used between all the clients and the server. The number of requests will also be measured on the NGINX server and gives us an indication how queries are solved in the TPF clients. Lastly, the mean CPU usage of the server will be a combined CPU usage of the NGINX server and the TPF/Marmotta server in order to quantify the effectiveness of the solution as a whole.

8.3 Experiment 1: Influence of query rewriting on the execution of SPARQL-MM queries in TPF

For the first experiment, the influence of query rewriting for SPARQL-MM queries in the TPF client is investigated. Therefore, all queries used in this experiment follow the less optimal syntax and will rely on the the filter to determine whether two fragments are from the same source media file, as already explained in section 7.2.2. The results of the experiment can be found in the graphs in figure 8.2. As one can see, we get some interesting results. First of all, we see a clearly noticeable difference in execution time between the rewritten queries and the unaltered queries. This can be explained by our reasoning in section 7.2.2. Because of the more optimal query execution order, the client can find the right solution mappings more easily. The effect of the rewrite, which is essentially an early filtering of the results, shows how much the TPF client benefits from applying the filter as early as possible.

This can also be seen in graph 8.2g which shows that the regular TPF client had some clients which failed to execute the query opposed to the optimised version where every client was able to correctly execute
the query and return the results. This can be explained when comparing graphs 8.2g and 8.2d. In the
case of the regular TPF client, we see that the TPF server gets bottlenecked at certain points (when 25,
55 and 70 simultaneous clients are querying). This can be explained due to the big amount of pages
with triples that need to be served simultaneously to the clients. It is at these time instances that clients
start to fail their execution. This is clearly shown in graph 8.2g between 25 and 55 simultaneous clients.
The TPF server was bottlenecked when 25 clients were accessing it. At this time, these could still finish
their queries. However, the TPF server needs some time to recuperate and we can see that for the next
four rounds (until 50 simultaneous clients), some clients were rejected. After this, the server was again
able to handle all clients simultaneously. It was however again bottlenecked by this burst, leading to a
rejected client in the next round of the experiment. The TPF server is thus running into its limits which
causes for some instability. As can be seen in figure 8.2f, the NGINX proxy server will mitigate some of
this instability by returning cached results where possible. It can however only do this for a very limited
amount of requests which is shown by the low CPU usage of the NGINX server.

If we now look at the TPF client enabled with query rewriting, we can see a more stable execution. No
clients are rejected, the query execution time is much faster and the overall CPU usage is lower com-
pared to the regular TPF client. This even though more bandwidth and requests are used. This seems
contradictory at first but can easily be explained. As already shown in section 7.2.2, the query execution
in the rewritten case is drastically different compared to the unaltered (suboptimal) query. If we again
look at listing 7.2, the TPF client has only two triple patterns for which it has to retrieve the pages with
results. In listing 7.1 however, all possible solution mappings for ?f1 will result in a different request of
the first page for tp1, after which a request for tp2 will be done to retrieve all other fragments in the source
media file. This on its turn results in multiple possible values for ?f2, each causing in a different request
for tp4. We thus have a far larger number of requests that are done to the server which every time has
to respond with the first page of the TPF corresponding to the requested triple pattern. Because of this
large amount of requests, the NGINX caching server has a bigger chance of having a cached version of
the response for a request. We also clearly see this in graph 8.2e and 8.2f. After a while, nearly every
request can be served by the caching server causing it to consume more CPU power and reducing the
CPU usage of the TPF server. The caching server can however return the response far more efficiently
compared to the TPF server, thus resulting in a lower CPU usage over all. The caching server is thus a big
benefactor for providing a lower total CPU usage. We can also see this in graph 8.2e. At first, when the
caching server does not have many cached responses, the CPU usage of the TPF server is nearly identical
for both solutions. Only when caching starts to happen, will the total CPU usage start to differentiate.
If the caching server would not be in place, the CPU usage would be as much or greater than when the
regular TPF client is used.

The execution of the rewritten query will also have its benefits for the execution on the client. Even
though the optimised execution order will indeed result in more requests, it will also result in far less
pairs of values that need to be checked by the filter as the client can use the count metadata to stop the
execution early if no suitable fragments are found. This will lead to a big benefit as the unaltered query
Figure 8.2: Graphs of the query rewrite experiment
forces the client to apply the filter on every possible pair of values for the two fragments it has as input. Since these are computationally more intensive than fetching a page from the TPF server and checking its (count meta-) data, the pairwise application of the filter will result in a slower execution.

We can thus drastically improve the query execution speed by letting the TPF client automatically rewrite the SPARQL-MM query to its optimal form before execution. This will however go paired with an increased bandwidth and larger amount of requests. Therefore, a caching server is a big benefactor for this solution. Without it, the TPF server itself would have to serve all the request, resulting in a possible higher CPU usage at the server compared to execution of the regular TPF client.

8.4 Experiment 2: Influence of local binding calculation on the execution of SPARQL-MM queries in TPF

In this experiment, the influence of calculating bindings on the client without requesting the server for a certain subset of triple patterns will be evaluated. In order to fully cancel out the influence of query rewriting, a query set of only optimal queries will be used. Therefore, no rewriting will be done in the optimised client and a fair comparison can be made to measure the difference in execution time, CPU usage, bandwidth usage etc. It is also important to notice that these experiments were performed on a 2 gigabit/s connection with a latency of 0.2ms on average as the effect of the optimisation will greatly be determined by the connection speed between the client and server. The results of the experiment can be found in figure 8.3.

We can see that the optimisation clearly creates a benefit compared to the regular SPARQL-MM enabled client. Graph 8.3a shows that, except for some anomalies, the execution time is always lower in the optimised case. The anomalies can be explained by the slight difference in the requests that are made. This means that the caching server will have a slightly different behaviour and due to this, the optimised client sometimes has to rely on the TPF server to respond opposed to the caching sever. We can also see this if we look at the other graphs in figure 8.3. When looking at the results of 125 simultaneous clients, we can see in graph 8.3d that they did not do more requests to the server but when looking at graphs 8.3d and 8.3e, we can see that there is more CPU activity from the TPF server compared to the regular SPARQL-MM enabled TPF client while the CPU usage of the caching server is slightly lower. This confirms our theory. Next to the lower mean execution times, we can also see that the mean execution time increases more slowly for the optimised TPF client. This can be explained as follows: the number of fragments returned by the server will influence the effect of the local binding calculation. Take for example the query of listing 7.1. The TPF client will start by retrieving the solution for $tp_3$ as this has the least number of estimated solutions. For each of these 5 000 results, we will then traverse the BGP and update the binding of the $?image$ variable. For each of these updates, we will save a little bit of time by not doing a request to the server. When more clients are querying the server at the same time, the
Figure 8.3: Graphs of the local binding update experiment

(a) Mean execution time for SPARQL-MM queries

(b) Total bandwidth usage between all clients and server

(c) Total number of requests between all clients and server

(d) CPU usage of server

(e) CPU usage of TPF server

(f) CPU usage of NGINX proxy server
more local binding updates we will do and the bigger the difference will become. In essence, the more fragments are returned, the greater the effect will be. If instead of 5000 fragments containing a person, only one contained a person in our dataset, the effect would be much smaller as this would mean that we only will do 1 less request to the server compared to the regular SPARQL-MM enabled TPF client. The difference might in this case not be as big but it does however still provides a small improvement. Lastly, we will make a final remark concerning the results of the query execution time. We have a mean difference of around 10 seconds over the whole experiment. Considering the fact that all queries are done over a local network with a very low latency and very big bandwidth, this is quite a significant result. In real life, a 0.2ms latency and 2 gigabit/s connection will almost never occur. A connection with a latency of 7ms and speed of 200 megabit/s is considered quite fast for consumer internet connections. The effect would therefore be even bigger on those connections as the difference between calculating a binding or requesting the server to determine a binding will become much bigger than in our experiment.

Next to the decreased execution times, we also notice that indeed less requests are done to the server. This on its turn results in a lower bandwidth usage compared to the regular SPARQL-MM enabled TPF client. We again see that the benefit becomes bigger with an increasing amount of simultaneous clients, albeit less significant compared to the execution time. For this, the same reasoning as previous paragraph applies. Graph 8.3c also indicates this effect. The more clients there are, the bigger the difference in number of requests will become. When looking at 150 simultaneous clients, this difference becomes quite large with the optimised client doing around 400 000 less requests to the server which is an improvement of 12.5%. Since the number of requests and the bandwidth usage are directly correlated to each other, the same improvement can be found in used bandwidth.

Finally, we will examine the influence of the optimisation on the CPU usage of the server. Looking at graph 8.3d we can see that, except from some anomalies which can be explained due to the different behaviour of the cache, the CPU usage of the server will be around 5% lower when the optimised client is used. This because the server simply has to process less requests compared to when the regular SPARQL-MM enabled TPF client is used. What might seem strange at first is that, in contradiction to the execution time and bandwidth, the difference between both solutions does not become bigger for an increasing amount of simultaneous clients. This is because of the NGINX caching server which will be able to respond to nearly all requests after the cache has been warmed up. It can do this so efficiently that the extra amount of requests will have nearly no effect on its CPU usage. We can also see this in graph 8.3f where the CPU usage of NGINX is nearly identical for both situations. Only from 15 to 50 simultaneous clients, the CPU usage seems a little lower. This because the lower amount of requests results in a slower warm up period. When requests however have a cache miss, the TPF server will need to respond. Now, since the caching server has only a limited amount of requests it can cache, the larger amount of requests in the regular SPARQL-MM enabled TPF client will increase the chance of having requests evicted from the cache. Therefore, more requests will be done to the TPF server and a higher CPU usage can be seen. This theory can also be seen in graph 8.3e and 8.3f. Once the cache is warmed up, it will always use 25% of the server CPU, indicating that it is working at peak performance and has a full cache. We do however
CHAPTER 8. PERFORMANCE EVALUATIONS

see peaks of CPU usage of the TPF server even when the cache is working at maximum capacity, thus indicating that the cache starts evicting entries or that requests are made that have never appeared before during the experiment.

We can clearly see that the optimisation indeed has a positive effect on the bandwidth and CPU usage of the server. These effects will always be of the same magnitude, irregardless of the connection between the clients and server. On top of that, even with our very high performance connections between the clients and server, we can see a clear improvement in execution times and slower connection speeds will further improve this difference. Even though the connection between the clients and server play an important role in the achievable improvements, the queries that are used will also determine how much faster we can be. As already explained in the beginning of this section, the number of fragments that are returned by the server will determine how many local updates we can do in the remainder of the query execution. So if the query has only a very limited amount of matching fragments, the effect will be smaller. Notice however that even when only a very small amount of requests can be saved, it still will provide an improvement over the regular SPARQL-MM enabled TPF client. When the connection between the clients and server is very slow, this could still result in some noticeable differences. This optimisation therefore indicates how using the Media Fragments URI structure during execution (in stead of before executing like the query rewriting optimisation) can noticeably impact the performance. On top of that, the more matching fragments, the greater the relative improvement will be. This is a very favourable effect.

8.5 Experiment 3: Comparing regular TPF, optimised TPF and SPARQL-MM in Apache Marmotta

The previous experiments were specifically tailored to evaluate the influence of both optimisations implemented in the optimised version of the TPF client. In order to do so, the queries were carefully constructed, thus creating an artificial situation. In real life however, users are free to construct the queries however they want. This means that there is a high chance of having suboptimal queries being executed in the system. Therefore, in order to simulate this behaviour, the two previous query sets are combined in one big set. Since both sets have, from a semantic standpoint, identically the same (number of) queries, we have a 50% chance of selecting an optimal query (and consequently also a 50% chance of selecting a suboptimal query). This way, we have a balanced set between optimal and suboptimal queries which allows us to simulate the normal behaviour of most systems. This set will be used to compare the performance of the TPF client without optimisations, the optimised TPF client which will perform query rewriting and local solution mapping calculation and the centralised SPARQL-MM implementation of Apache Marmotta. The results of this experiment can be found in figure 8.4. These will enable us to answer the two research questions by evaluating the corresponding hypotheses defined in section 6.
8.5.1 Research question 1

Question: “Can we improve the scalability of SPARQL-MM enabled systems in terms of the number of simultaneous clients that can be served by moving the SPARQL-MM functionality to the client?”

In order to answer this question, we will evaluate all the corresponding hypotheses. This by taking both the regular and optimised TPF client into consideration when comparing results with Apache Marmotta.

Hypothesis 1.1: “Moving the spatial and temporal reasoning to the client will negatively impact the query execution speed for a single machine executing a single SPARQL-MM query compared to complete server-side reasoning.”

When looking at graph 8.4a this is indeed the case. Since we have 5 separate machines querying the server, each machine will only perform one query on the first run of the experiment. We therefore look at these values to evaluate this hypothesis. When looking at the results, it is clear that the the SPARQL-MM implementation of Apache Marmotta indeed is faster than both the regular and optimised TPF client. The difference between the regular client and Apache Marmotta is very big while the optimised client manages to close the gap and have an execution time much closer to the centralised approach. It however is still much slower than the centralised approach. This hypothesis therefore holds.

Hypothesis 1.2: “By moving spatial and temporal reasoning to the client, the SPARQL-MM query execution time will not increase as fast with an increasing number of clients accessing the server simultaneously compared to complete server-side reasoning.”

The data for proving this hypothesis can again be found in graph 8.4a. Here we can clearly see how the mean execution time of clients using Apache Marmotta rises much faster with an increasing number of simultaneous clients compared to the TPF solution. When 15 clients are querying the server, the optimised TPF client already performs better than the centralised approach and from 50 simultaneous clients on, even the regular TPF client starts to outperform the server-side implementation of Apache Marmotta. On close inspection, we can argue that the unoptimised TPF solution will even outperform the Apache Marmotta solution much faster. In graph 8.4a we can see that from as early as 10 simultaneous clients, the Apache Marmotta implementation starts to reject clients. These are mainly timeouts in the beginning of the experiment but as more simultaneous clients start to query these rejections are purely because the server starts to crash and stops accepting requests. At 10 simultaneous clients, 5 are getting rejected. For 15 simultaneous clients, they got all rejected. After this, we have multiple rejections for each run and for 25, 30, 40 and 60 simultaneous clients, they even got all rejected again. This is why the time measurements for these points are missing in graph 8.4a. From 60 simultaneous clients on, the server completely bottlenecks and no single query is accepted anymore. This is why the trendline for the Apache Marmotta implementation also stops at 60 simultaneous clients. The TPF solution is however
CHAPTER 8. PERFORMANCE EVALUATIONS

capable of serving all clients without a problem and no rejections occur. On top of that, the slopes of the mean execution times in graph 8.4a are much shallower compared to the rapid increasing execution times of the centralised approach. Moving the SPARQL-MM reasoning to the client does indeed result in a much slower increasing execution time. This hypothesis is therefore also proven to be correct.

One other noticeable remark that can be made is how even though the unoptimised TPF client also suffers from the inefficient queries and, just like Apache Marmotta, is unable to rewrite these queries, it still is capable to solve the query for all clients without rejecting any of them. This however at the cost of a very slow execution time which can take up to several minutes.

Hypothesis 1.3: “The CPU usage of the server is smaller and will increase more slowly with an increasing number of clients accessing the server simultaneously when the spatial and temporal reasoning is moved to the clients.”

In order to prove this hypothesis, we will look at graph 8.4d. It is clear that there is a big difference between the CPU usage of the TPF solution and the centralised implementation. The Apache Marmotta implementation will consistently be using nearly all available CPU. This even for as little as 10 simultaneous clients. This is mainly due to the presence of the suboptimal queries. They will not only have an impact on the query execution for the TPF client but also the Apache Marmotta implementation will suffer greatly from the inefficiency they introduce. Just like the TPF clients would have to compare each possible combination of two fragments (see section 7.2.1), the Apache Marmotta server will also have to do this for every suboptimal query. The big difference is however that in the case of TPF, the filter executions are all done locally by the client while the Apache Marmotta server has to do the filter executions for all clients on the server itself, thus using nearly all of the CPU of the server. One might wonder why the CPU usage is capped at 81%. This is due to other processes like the NGINX proxy server and CPU usage monitoring scripts which are also running on the server and are using around 20% of the CPU all together. The Apache Marmotta server is thus constantly using all available CPU power. When looking at the TPF solution however, we can see a much better performance with a maximum server CPU usage of around 65%. This is where the benefit of client-side query execution, and more specific, filter execution can been seen. Since the Apache Marmotta server is already using all available CPU power from 10 simultaneous clients on, this solution will be deemed to have a far faster increase of CPU usage compared to the TPF solution. We after all went from nearly 0% usage to 81% usage with only 5 more clients simultaneously querying the server. This hypothesis therefore is also correct.

Hypothesis 1.4: “The bandwidth usage will be bigger when implementing client-side spatial and temporal reasoning compared to complete server-side reasoning.”

Graph 8.4b shows us that both the regular and optimised TPF client use more bandwidth compared to Apache Marmotta. This is inherent to the way the TPF interface is defined. In stead of doing one request
and getting a single response with the result like Apache Marmotta, the client has to do a vast amount of requests to the TPF server which then also has to send a corresponding response in order to solve the query. For 5 simultaneous clients, the Apache Marmotta solution only has to respond with the answer to 5 requests while the TPF server has to serve around 90,000 requests to solve the same queries. We can only compare up until 60 simultaneous clients as after this, the Apache Marmotta server starts to deny all clients. It is however very clear that querying media data using the TPF client will cause for a significantly larger bandwidth usage. This hypothesis is therefore also correct.

Conclusion

Previous hypotheses have shown that by moving the SPARQL-MM functionality to the client, a far better scalability can be achieved compared to the server-side implementation of Apache Marmotta. The CPU usage of the server is lower and the execution times are not increasing as fast for an increasing amount of simultaneous clients. There are however some disadvantages using TPF to query media files. Firstly, the execution time of a query will be much larger compared to using a server-side implementation like Apache Marmotta when only one client is accessing it. Especially the unoptimised TPF client has a limited performance and can take several minutes before all answers are found. We do however have a far smaller chance of getting rejected while executing the query and since the TPF client will return results in a streaming fashion, the user does not have to wait minutes before seeing results. Next to the slower execution times, the TPF solution will also use far more bandwidth compared to Apache Marmotta. This will have an impact both on the server and clients. The TPF server will need a more stable and faster internet connection to prevent bottlenecking. From the experiment we can further more calculate that on average 12MB of data is consumed by each client opposed to Apache Marmotta where only 1MB is needed at most per client. For devices with a mobile network connection and limited data, this could pose a problem. Even though we have some disadvantages when executing queries, their gravity is far less than not being able to execute the queries at all due to unavailable server.

8.5.2 Research question 2

Question: “Can we improve the performance of the Triple Pattern Fragment’s execution of SPARQL-MM queries by using the Media Fragments URI standard?”

In order to answer this question, we will now focus specifically on the differences between the regular SPARQL-MM enabled TPF client and the optimised TPF client which will be performing query rewriting and local solution mapping updates where possible. We will again provide answers to the hypotheses related to this question followed by a final conclusion of whether we can use the Media Fragments URI structure to our benefit.
CHAPTER 8. PERFORMANCE EVALUATIONS

Figure 8.4: Graphs of the final experiment comparing regular TPF, optimised TPF and Apache Marmotta
CHAPTER 8. PERFORMANCE EVALUATIONS

**Hypothesis 2.1**  “By making use of the URI structure defined in the Media Fragments URI standard, the execution time of SPARQL-MM queries will be lower or at least as fast compared to the standard Triple Pattern Fragment SPARQL-MM query execution.”

From graph 8.4a it is clear that using the structure of the Media Fragments URI standard can cause drastically better performance compared to the regular SPARQL-MM enabled TPF client. The regular TPF client has recorded mean query execution times of up to around 450 seconds (= 7.5 minutes) before all results of the query are found. The optimised TPF client is however (in most cases) able to solve the same queries on the same infrastructure in under 50 seconds. If we look at the trend lines we see an average difference of 250 seconds (= 3.6 minutes) which is quite a big improvement. For 10 simultaneous clients, we have the least difference between both solutions which is still around 100 seconds (1.6 minutes). On top of that, the execution times are far more consistent and increase less rapidly compared to the regular SPARQL-MM enabled TPF client. This big difference is mainly the consequence of the query rewriting optimisation. We could also see this in graph 8.2a where the difference between the regular and optimised client are even bigger. The fact that the difference between the regular and optimised client is smaller compared to graph 8.2a can be explained by our query set used during the experiment. The set contains equally as much optimal queries as suboptimal queries. On top of that, the query selection is completely random, meaning that 50 % of the queries that are executed by the regular client already will be optimal. The fact that the difference in graph 8.4a is around half as big as in graph 8.2a confirms this theory. If we would however provide the regular client with only optimal queries, this advantage completely disappears and the difference in execution speed will depend on the time saved by local binding updates. This depends on the quality of the connection between the client and the server and will therefore differentiate from client to client. Even though they might be small, there will always be some improvements made by the optimisations and therefore we can accept this hypothesis.

**Hypothesis 2.2**  “By making use of the URI structure defined in the Media Fragments URI standard, the bandwidth usage will be smaller compared to the standard Triple Pattern Fragment SPARQL-MM query execution.”

In order to discuss this hypothesis, we will look at graph 8.2b, 8.3b and 8.4b. We can see that the hypothesis does indeed hold for the local bindings update optimisation. The client has to do fewer requests to the server and therefore, less bandwidth needs to be used to solve the same queries. This improvement however only holds when the unoptimised client uses optimal queries which is not always the case. The saved bandwidth is also much lower compared to the extra bandwidth usage the query rewriting optimisation introduces. This because the new query will cause for far more requests to be done to the server as already explained in section 7.2.2. Even though the query rewriting will enable the client to perform local binding updates due to the specific triple patterns it introduces, the number of requests that can be saved are only limited. This does however not mean that the local binding updates have no purpose. They will act as a mitigator to limit the extra bandwidth usage introduced by the rewritten queries. Since the overall bandwidth usage in graph 8.4b is larger for the optimised TPF solution, we reject this hypothesis.
CHAPTER 8. PERFORMANCE EVALUATIONS

Hypothesis 2.3 “By making use of the URI structure defined in the Media Fragments URI standard, the number of requests to the TPF servers will be smaller compared to the standard Triple Pattern Fragment SPARQL-MM query execution.”

As can be seen in graphs 8.2d, 8.3d and 8.4d, the same results can be found as when looking at the bandwidth usage for the optimised solution. This because the bandwidth usage and number of requests are directly correlated to each other. The more requests are done, the more bandwidth will be used. We can therefore use the same reasoning as in the previous hypothesis. Since the number of requests will be greater in general, we also reject this hypothesis.

Hypothesis 2.4 “By making use of the URI structure defined in the Media Fragments URI standard, the CPU usage of the server will be lower compared to the standard Triple Pattern Fragments SPARQL-MM query execution.”

When looking at the graph 8.4d, we can see that the optimised TPF client results in an average of 15% to 20% less CPU usage of the server compared to the regular SPARQL-MM enabled client. This is a very interesting result as a lower CPU usage of the server will further improve the scalability of the system. The improvement of CPU usage is due to the combination of both optimisations. As already explained in section 8.3, the query rewriting will result in triple patterns that can be solved much more efficiently by the TPF server, thus diminishing the CPU usage. Next to this, the extra requests that are done to the server will cause for a higher chance of a cache hit. This can also be seen in graph 8.4f. The CPU usage of the NGINX caching server is higher for the optimised clients than when regular clients are used indicating that the caching server will create more responses, thus diminishing the load of the TPF server. These two factors, combined with the benefit of local binding updates on the CPU usage of the server causes for this big improvement. The optimised clients will indeed cause for a lower CPU usage and therefore we can accept this hypothesis.

Conclusion

Even though the optimisations generally result in a bigger bandwidth and bigger amount of requests to the server, we can clearly see that using the Media Fragments URI standard to make the client more intelligent does have its benefits. By using the hierarchical structure of Media Fragments and their source media files, we are able to express constraints that normally appear in a filter in the form of triple patterns, thus effectively doing a partial early filtering. Next to this, the standardised structure of Media Fragments URI could help the client mitigate slow internet connection speeds by calculating parts of the solution mappings locally without requesting the server. Over all, the obtained benefit over the regular SPARQL-MM client will greatly depend on the used query and the connection to the server. The query rewriting will however always ensure an optimal execution speed and thus alleviates the person constructing the
CHAPTER 8. PERFORMANCE EVALUATIONS

query from having to consider the best possible construction. This greatly improves the overall usability of the system and will result in more consistent results overall.
CHAPTER 9

Conclusion

The goal of this thesis was to create a more scalable querying system which allows users to define fine-grained search queries with which they can define spatial and temporal constraints on the object shown in the media files. Next to this, we also wanted to make use of Semantic Web technologies as these are fundamental for creating a completely distributed social net which can solve a lot of problems as described in chapter 1. We however do not have to wait for a social network to be realised as current centralised systems can also use Semantic Web technologies in the backend to provide the end user with search functionality. Currently, SPARQL-MM already provided an extensions on the SPARQL query language which allowed for spatial and temporal reasoning on Media Fragments. This solution however has some scalability issues as the queries all need to be solved server-side. Therefore, in order to solve this scalability issue, the TPF framework has been expanded so it can support SPARQL-MM enabled queries. For this we have considered multiple models for calculating the temporal and spatial relations of fragments and implemented the most optimal ones. As the TPF framework is designed to move the query solving logic to the clients, using this framework will result in less load on the server which will on its turn allow for more clients to be served simultaneously. From section 8.5.1 we can conclude that implementing the SPARQL-MM functionality indeed results in a far more scalable solution compared to the complete server-side evaluation of the query, thus achieving our goal. The query execution time will however be far slower for the TPF solution. Therefore, the structure the Media Fragment URI standard introduces has been used to optimise the TPF client by allowing it to rewrite queries to an optimal form and calculating bindings locally without having to request the server. This has proven to drastically improve the query execution speed while simultaneously lowering the CPU usage of the server. The bandwidth usage will however increase slightly but this increase is far less compared to the improvements we achieved in execution speed and CPU usage. This therefore can not outweigh the benefits of having a much faster execution time. Even though the greatness depends on the used queries and connection between the client and server, we will always create some improvement on the query execution time and CPU usage of the server.
This thesis has proven that we can use the TPF Framework in combination with Media Fragments URI to create a very expressive media querying system that can manage a high number of simultaneous users. However, in order to search for media files, the queries all need to be constructed in the SPARQL query language. Therefore, in order to improve the usability of the system, a natural language processor could be added to translate natural sentences into correct SPARQL-MM queries. The current solution also imposes very strict rules on how the media data needs to be structured and which predicates need to be used to denote the relation between the source media file and its fragments. However, since Media Fragments URI follow a standardised structure, research could be done in how we can use this structure to enable the TPF framework to automatically detect the relation "source media - media fragment". This way, datasets can be created more liberally and a fault tolerance is added when the provided data is missing certain relations. The thesis has also shown how we can use the hierarchical structure encoded in Media Fragments URI themselves to allow for more optimal SPARQL-MM queries and a more intelligent SPARQL-MM query execution. Future research could investigate how we can use this structure to further improve the TPF client and how we can also use it on the server when using it as a media querying system. Further more, as using information encapsulated in URIs itself to help solve SPARQL queries has not been done before in the TPF framework, we have created a new approach of improving the TPF framework which allows for some further research to be done. Can the TPF framework make use of other URI standards to optimise its SPARQL execution algorithm in general? If not, are there other cases in which we can use more specific URI standards like Media Fragments URI to counteract bottlenecks in the current TPF framework? Can we define URI structures ourselves and specifically tailor them to the TPF framework for optimal results? How would these URI structures be compliant with the Semantic Web? These are a few questions that come to mind and need some further research in order to be answered.
References


