UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CACHING OF SPARQL QUERIES

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS DE LA COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

GABRIEL IGNACIO CHANDÍA GONZÁLEZ

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
ANDRÉS ABELIUK KIMELMAN
CARLOS BUIL ARANDA
CLAUDIO GUTIÉRREZ GALLARDO

SANTIAGO DE CHILE
2023

Caching de Queries para SPARQL

La Web Semántica es el área de estudio que se dedica a investigar cómo hacer que la estructuración de los datos en la web sean entendibles por máquinas, mediante la definición de una estructura semántica (RDF) y propiedades ontológicas (OWL). RDF permite estructurar los datos de la web en grafos de triplas, donde un sujeto y un objeto están conectados a través de una propiedad.

SPARQL es el lenguaje definido por la W3C (World Wide Web Consortium) como el estándar para consultas de datos en RDF. En el presente, existe una gran demanda por servicios que usan este lenguaje, de la cual nace la necesidad de reducir los tiempos de respuesta promedio con tal de optimizar su uso. La propuesta de esta tesis consiste en el diseño e implementación de un caché, el cual permite guardar en memoria los resultados de las consultas más frecuentes con tal de poder acceder a ellos de manera más eficiente. En este trabajo se proponen diversas maneras de poder guardar y eliminar elementos del caché diseñado, y se realizaron experimentos para probar la política de uso más eficiente.

Entre las políticas de uso estudiadas se propuso una ideada en este propio trabajo, que toma en consideración el distinto tamaño que pueden poseer los resultados de las consultas (cosa que no hacen necesariamente las otras políticas estudiadas); y los experimentos hechos demuestran que esta es la más eficiente en permitir reducir la mediana de los tiempos de ejecución de las consultas.

El trabajo concluye que es posible reducir los tiempos medianos y de cuartiles superiores mediante el uso de un sistema de caché para SPARQL. Además, se demuestra que mediante otros ajustes tales como la selección apropiada de consultas a guardar en el caché y el uso de un algoritmo de canonicalización, ideado por Salas y Hogan, es posible obtener aun mejores resultados con respecto a los tiempos que toma ejecutar las consultas.

Abstract

The Semantic Web is a research area dedicated to studying the question of how to make the web's data structures comprehensible for machines through the definition of a standard data model (RDF) and ontological properties (OWL). RDF allows for structuring data on the web as graphs of triples, where a subject and an object are linked through a property.

SPARQL is a query language defined by the W3C (World Wide Web Consortium) as the standard for querying data in RDF. Presently, there exists a huge demand for services which use this language, from which a necessity arises to reduce their average response time in order to optimize their use. This thesis proposes the design and implementation of a cache, which enables keeping in memory the results of the most frequent queries so that they can be accessed in a faster way. This work proposes different ways to cache and delete elements, and different experiments were designed to determine which policy is the most efficient.

Among the policies we studied, we propose one created in this work which takes into consideration the different sizes query responses may have (something that the other policies we consider don't necessarily take into account); and the experiments show that this is the most efficient which allows to reduce the median of the execution times for the queries.

The work concludes that it is possible to reduce median and upper quartiles times by using a caching system for SPARQL. Furthermore, we prove that with other adjustments such as selecting the appropiate queries to store in the cache and the use of a canonicalisation algorithm, created by Salas and Hogan, it is possible to get even better results for the times that it takes to execute the queries.

Para Osvaldo, Marcela y Antonia.

Agradecimientos

Quisiera agradecerle a mis papás por brindarme apoyo incondicional, a mi hermana por ser el miembro más cercano de mi familia en quien más puedo confiar, y a mis abuelos por sus cuidados que me han brindado toda la vida.

Le agradezco a mis amigos, con quienes he podido compartir y quienes me han entendido en mis momentos más complicados; a mis mascotas Yuri, Kiryu, Guatoncita, Mitsuki y Chiri; a mis terapeutas sin las cuales no podría haber llevado a cabo este proceso; y finalmente le agradezco a Aidan mi profesor guía quien me brindó toda la ayuda necesaria para poder completar este trabajo.

Table of Content

List of Figures

Chapter 1

Introduction

The Semantic Web is an extension of the current World Wide Web, promoted by the W3C (the World Wide Web Consortium) to be the new standard for how data can be shared over the Web. As it was, sites on the Web would give no more information than what's in their HTML code. However, using certain standards like the Resource Description Framework (more commonly known as RDF) it is possible to share more easily content describing what a website or application is about, allowing machines to do more complex searching or querying tasks. In a way, it could be said that the goal of doing research on the Semantic Web field is to turn the World Wide Web into a giant global Database.

The RDF model [2] allows for data to be described as triples composed by a subject, a predicate and an object. Alongside this model there exist RDF Schema [3] (known as RDFS) and the Web Ontology Language (or OWL), both of which help define properties and traits different entities have and share. Since websites may publish vasts amounts of diverse data using this triple-based model, a language to query over these triples is necessary; and such a language exists in the form of SPARQL [4], which has also been standarized by the W3C as the default querying language for data described as RDF triples.

Currently there exist several sites which store data as RDF triples, the most prominent ones being DBpedia [13] and Wikidata [9]. They also host SPARQL query services, called endpoints, which are known to be vastly used everyday, as is described by Saleem et al.'s work [20], which describes a dataset for queries taken from endpoints such as DBpedia's and others, and Malyshev et al.'s work [15] on Wikidata's query logs, from which anonymized dumps are generated weekly. This latter work goes into detail analyzing how queries over Wikidata change over time, using different combined features, and describes how many organic (or non robotic) queries are being evaluated daily, concluding that this number grew considerably from 2017 (640,419 queries) to 2018 (2,339,735 queries), meaning that SPARQL implementations are able to respond to real world queries at large scale.

While SPARQL engines implement various optimizations, these endpoints often still have performance problems [6], usually returning incomplete results or timeouts. Given this issue, some researchers in this field have dedicated time in trying to find better ways to reduce average response times for SPARQL, as well as trying to minimize the amount of unnecesary timeouts [5]. One such possible solution to these issues is implementing caching.

The idea of caching for SPARQL was first mentioned in the work of Martin et al. [16]. This work introduced a meta-data relational database, which was responsible for caching query results. This system improves query performance overall, however, it does not consider equivalent queries, meaning it can be improved. After this, Papailiou et al. [17] presented a work which partially considers this issue of equivalence, and offers a more complete caching system that adapts to different amounts of workload.

The work by Papailiou et al. is the most complete and up to date in terms of SPARQL caching, to the best of our knowledge; however, it is not as complete as it could be.

First, improvements can be made on Papailiou et al.'s work related to the benchmarks defined to be used for their experiments. They used a syntactic dataset called LUBM [1], which contains data related to universities (including information about teachers, students, courses and so forth). We propose more complex benchmarks to be used for experiments, containing data about unrelated entities, in order to better approach a more realistic scenario.

Second, query languages such as SPARQL give rise to equivalent queries that while expressed differently, return the same results. In the context of caching, relying on the input query syntax alone would imply that the cache will miss relevant available data for equivalent queries. While one of the main novelty's of Papailiou et al.'s work was the addition of methods to canonically label query variables, this only covers query isomorphism, a property two queries share when they only differ in variable labels. However, per the work of Salas and Hogan [19], more complete methods for query canonicalization now exist that can capture further equivalences between queries modulo variable names, and can thus detect more cached queries, since their work is also able to eliminate redundancy, as well as covering algebraic operations such as union.

Example 1. Consider the following queries Q_1 and Q_2:

```
Q1
SELECT DISTINCT ?a
WHERE {
  ?b :name ?a.
  {?b :teaches :Calculus.} UNION
  {?b :teaches :Algebra}}
```

```
Q2
SELECT DISTINCT ?x
WHERE {
  {?b :teaches :Calculus; :name ?x.}
  UNION
  {?b :teaches :Algebra; :name ?x.}}
```

Both queries perform a projection of different variable names yet will return the same results for any RDF dataset; namely, both queries return the names of teachers of Calculus or Algebra. Salas and Hogan's work presented an algorithm which applies to SPARQL monotone queries[1] that performs syntactic canonicalization.

Ultimately, since this algorithm is able to efficiently canonicalize real-world queries, it can be used for SPARQL query caching as a way to capture the largest possible amount of profitable queries, without the need to spend cache space on redundant patterns. □

SPARQL queries work for both bag and set semantics, which means it allows to return or remove duplicate results. In SPARQL, the *distinct* feature allows for the use of set semantics,

[1]Monotone queries are the ones that use the project, select, union and join features such as seen in Example 1.

as can be seen in Example 1. Salas and Hogan's work also covers both kinds of semantics, ultimately meaning their algorithm is sound and complete for all monotone queries. It is also sound for more complex queries, using features such as negation, etc; meaning that given two such queries, if the output canonical form is the same, then the two input queries are equivalent modulo variable names; however, for non-monotone queries it is not complete, meaning that not all queries that are equivalent modulo variable names will generate the same canonical form.

The methods of Salas and Hogan [19], however, can be improved even more. In particular, in the context of a caching system, rather than caching entire queries (which may refer to very specific criteria unlikely to be requested again) we can rather try to identify and canonicalize frequent subqueries for caching.

Example 2. Consider the following simple query Q_3:

```
Q3
SELECT DISTINCT ?movies ?title ?director
WHERE {?movies :title ?title;
       :genre :Drama;
       :director ?director .
       ?director :gender :Female;
       :education :UChile .
}
```

Q_3 describes a select query that will return information about movies, their titles if they are of the drama genre, and their directors if they are women and educated in Universidad de Chile. This query forms many subqueries, two of which are very easy to see: one describes women who were educated in Universidad de Chile, while the other is of movies of the drama genre. It is of our interest to study how profitable it could be to cache these subqueries instead of whole queries, thus the question arises: is it possible to reduce response times for SPARQL by caching only the most profitable subqueries? □

Considering all of these issues, this work proposes to improve on previous work by presenting a novel caching system which is able to decrease SPARQL query response time by using Salas and Hogan's canonicalization algorithm, and that is able to efficiently find profitable subqueries to cache, working in an environment where real world queries are evaluated such as Wikidata or DBpedia.

1.1. Hypothesis

We believe we can propose a more complete caching system that can work better in a more real-world setting by defining proper metrics to cache profitable queries in an environment such as Wikidata or DBpedia. SPARQL caching systems have already been implemented succesfully [16, 17], and since we will iterate over an already existing system, we propose that this work will help improve the performance of SPARQL endpoints for real-world workloads.

1.2. Research Questions

The main goal of this work is to present a novel caching system to improve SPARQL query response times on SPARQL endpoints, hence why we try to replicate a real-world query load to compare against a normal system that evaluates queries without using a cache. We aim to answer the following questions:

- Is it possible to create a caching system that improves on query performance for SPARQL, being able to decrease overall response times in a static data environment?
- Can we define metrics to determine profitable subqueries?
- Is it possible to compare this new system against the state of the art caching systems for SPARQL, in terms of the number of profitable queries and average response time? If so, does it fare well?

1.3. Structure

This work will present its findings by first revising the related work that has been done in terms of state of the art systems for SPARQL query caching, as well as all the work done for this system to be able to be developed; then we will propose our caching system, going into its implementation and then presenting its results in comparison to state of the art systems. In the end we will present our conclusions, as well as propose future work.

Chapter 2

Related Work

In this chapter we describe concepts related to this work, as well as the state of the art for SPARQL query caching and caching policies.

2.1. Semantic Web

The Semantic Web is an extension of the web driven by the World Wide Web Consortium. The end goal of this extension is to transform the web into a global database, and the main method to reach this is by defining protocols such as RDF, OWL and a query language to accompany them, such as SPARQL, which all allow for more complex searches and tasks to be evaluated automatically over the Web.

To see how everything in the Semantic Web connects we present the graphic shown in Figure 2.1 [10].

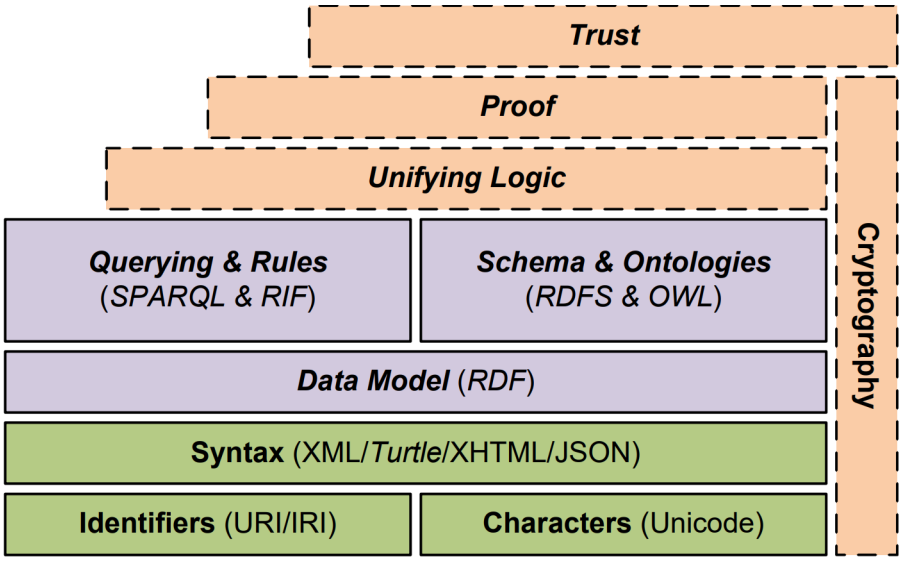

Figure 2.1. Layers of the Semantic Web

As we can see in Figure 2.1, RDF allows for modelling the data which follows certain semantics, and SPARQL along with RDF Schema allow to extract data and give logic to what we're modelling, by setting semantics via RDF Schema and OWL, and allowing the users to query over the data using SPARQL.

In this work we will be using RDF and SPARQL and hence we focus on them in this chapter.

2.1.1 RDF

The RDF model [2] allows for data to be described as triples. These triples are composed by a subject, a predicate and an object. This means there are two nodes or labels connected by a property, which is the predicate.

These nodes and properties are usually identified by an IRI, or Internationalized Resource Identifier, which can be understood as a URL that describes an entity (rather than just a document) on the web. Objects can also be literals, which are datatyped constants, such as a string, a date, a time, an integer, etc. Finally, subjects and objects can also be blank nodes, which refer to the existence of (unidentified) entities.

Let's check the example in Figure 2.2.

```
<http://ex.org/Chile>
<http://ex.org/capital>
<http://ex.org/Santiago> .

<http://ex.org/Chile>
<http://ex.org/population>
_:p .

_:p
<http://ex.org/value>
"19458000"^^<http://www.w3.org/2001/XMLSchema#integer> .

_:p
<http://ex.org/date>
"2021"^^<http://www.w3.org/2001/XMLSchema#gYear> .

<http://ex.org/Santiago>
<http://ex.org/population>
_:q .

_:q
<http://ex.org/value>
"6257516"^^<http://www.w3.org/2001/XMLSchema#integer> .

_:q
<http://ex.org/date>
"2017"^^<http://www.w3.org/2001/XMLSchema#gYear> .
```

Figure 2.2. Example 1

This example follows a format known as N-Triples. They explicitly show each triple forming a RDF graph, as can be seen in Figure 2.3.

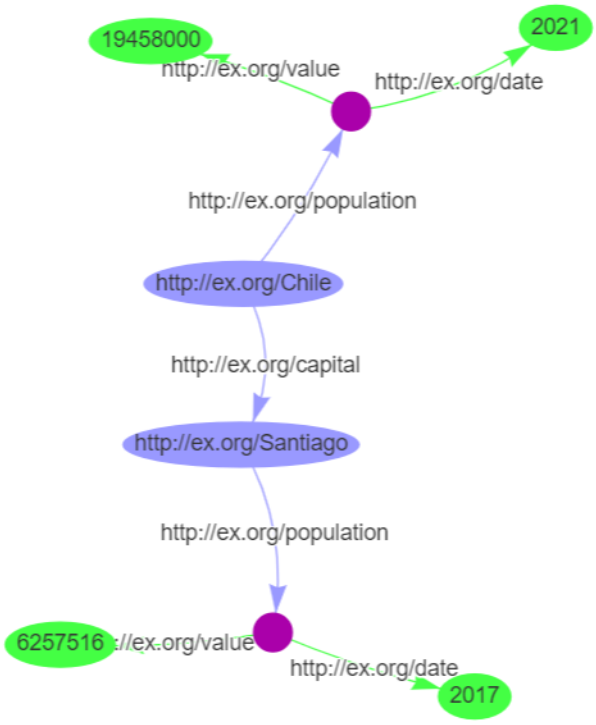

Figure 2.3. RDF Graph for Example 1

However, there are more concise ways to represent triples forming a RDF graph. Take for example the Turtle syntax in Figure 2.4, which can represent the same graph seen in Figure 2.3.

```
@prefix ex: <http://ex.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Chile ex:capital ex:Santiago .
ex:Chile ex:population [ ex:value 19458000 ; ex:date "2021"^^xsd:gYear ] .
ex:Santiago ex:population [ ex:value 6257516 ; ex:date "2017"^^xsd:gYear ] .
```

Figure 2.4. Example 2: Example 1 with Turtle syntax

In Turtle syntax, IRIs can be shortened by using prefixes. Blank nodes also help describing shared properties, and these can be seen in Turtle using brackets.

Finally, we can describe triples by using their direct URL links on the Web, like in the N-Triples example shown in Figure 2.5.

```
    http://www.wikidata.org/entity/Q298
    http://www.wikidata.org/prop/direct/P36
    http://www.wikidata.org/entity/Q2887 .
```

Figure 2.5. Example 3

The first IRI, the subject: `http://www.wikidata.org/entity/Q298` is the Wikidata identifier for the country of Chile. Other Wikidata identifiers are used for the predicate: `http://www.wikidata.org/prop/direct/P36` which describes the property of a capital, and the object: `http://www.wikidata.org/entity/Q2887` which describes the city of Santiago (Chile).

2.1.2 SPARQL

SPARQL is the standard query language used to query over RDF triples of any given graph. It is a pattern-matching language, and this is how it works.

First, any triple pattern in a SPARQL query is considered a query pattern. Let's say we're looking for the following:

```
?x ex:name ?y
```

This query pattern is conformed by, as any triple, a subject a predicate and an object. In this case, the subject and the object are `?x` and `?y`, which are known as *variables*, and can be mapped to anything in the graph. A `SELECT` query with this query pattern is asking for any two entities connected by the property `ex:name`, so it'll look for any triples matching this pattern and return the respective results.

An example SPARQL query, which we call Q_4, is as follows:

```
Q4
SELECT ?place ?value
WHERE {
    ?place ex:population ?population .
    ?population ex:value ?value .
}
```

This example presents a natural join. This natural join is specified though a set of triple patterns, known as a basic graph pattern (or BGP for short). As queries get extended and have more than one triple pattern, the results will be joined as one would see in other query languages, like SQL. The results for this query over the RDF graph of Figure 2.3 will be presented as so:

Another more complex example can be seen below:

?place	?value
ex:Chile	19458000
ex:Santiago	6257516

```
Q5
SELECT ?value
WHERE {
    {ex:Santiago ex:population ?population .
     ?population ex:value ?value .}
    UNION
    {ex:Chile ex:population ?population .
     ?population ex:value ?value .}
}
```

Q_5 presents a `UNION` clause, and will unify both results for the population of Chile and the population of Santiago, as can be seen below:

?value
6257516
19458000

2.1.3 Wikidata

Wikidata [9] is an open knowledge-base which keeps its data in RDF. It allows its users to edit, upload new data and query over it using a SPARQL engine.

Wikidata is meant to be the Wikipedia for data, meaning that it stores the data Wikipedia has while also hosting an endpoint to allow for human users and machines alike to query over its data.

As seen in the RDF section above, Wikidata entities are represented by their URLs. For example the entity of Chile is `http://www.wikidata.org/entity/Q298`, and when that site is visited it will list all of the properties this country has stored in Wikidata.

Currently there are over 100 million data items on Wikidata that anyone is free to edit. Wikidata has over 20 thousand active users, and it receives several million SPARQL queries on their endpoints per day [7].

2.2. Cache Policies

Caching query results so we can re-use them later allows for potentially faster SPARQL response times, which is what we're trying to achieve in this work. A cache memory usually has limited capacity; this is why in order to create a caching framework we need to decide which items we're going to prioritize for caching and which ones we'll be removing when necessary.

A cache policy defines a ruleset for how we're going to decide which items we get to keep in our cache, depending on when they were accessed, how often they were accessed, and so forth.

Below we describe the policies we implemented in this work.

2.2.1 Least Recently Used (LRU)

The least recently used policy, or LRU policy, follows the rule of deleting the least recently used items first from the cache. That is, as soon as the cache fills up, it deletes the item that hasn't been accessed in the longest time. Notice that by accessing in this work we mean either inserting an item or retrieving it, which means the item that is deleted hasn't been inserted or retrieved for the longest time in comparison to others in the cache.

2.2.2 Least Frequently Used (LFU)

The least frequently used policy, or LFU policy, keeps count of how many times each item was accessed, so that in the case that the cache gets filled, it deletes the item that has been least frequently accessed.

2.2.3 Random Replacement (RR)

The random replacement policy, or RR policy, follows a quite simple rule. Each time our cache is filled we look up a random item from it and delete it.

2.2.4 Low Inter-reference Recency Set (LIRS)

The LIRS policy has been defined by Jiang and Zhang [11], and it is of particular interest to us as it has been used in prominent database systems, such as MySQL[1], InnoDB, Infinispan, and an adaptation to this algorithm is used in NetBSD. This policy is important to our work given its usage in large database systems, its complexity and how we can use this to develop our own custom policy described in later sections.

It follows many rules for insertion, retrieval, and deletion of items in the cache. It uses two structures: a stack S and a list Q, each one containing blocks that have relevant information. Each block is described by two Boolean properties: LIR and *residency*. An LIR block is one that has an item with low inter-reference recency (or IRR).

The IRR is calculated by counting how many different items have been accessed between two accesses of one item. That is, if item A is accessed, then B, C and A again, two different items (B and C) are accessed between two A accesses, which means A has an IRR of 2. It

[1] `http://www.iskm.org/mysql56/pgman_8hpp_source.html`

doesn't matter if B or C are accessed multiple times between both accesses, the IRR will continue being 2. The block containing A will be considered an LIR block so long as A's IRR is low in comparison to the IRR properties of other blocks. A block is also considered *resident* if its respective item is inside the cache.

The stack S contains all kinds of blocks, *resident* and non-*resident* ones, and also blocks that are considered LIR and the opposite of it: HIR (high inter-reference recency); while the list Q only contains HIR *resident* blocks, i.e., blocks in the cache that could soon be removed if needed. As defined by the work of Jiang and Zhang [11], 99% of the blocks inserted will enter the stack and cache, and once that threshold is reached the list will begin to be filled up.

Whenever an item has to be deleted, the first item from the queue Q is removed from it and the cache.

To better understand this algorithm we can see the following example.

Let's assume our cache has a capacity of 4, our stack S has a length of 3, and our list Q can hold up to 1 item. This doesn't follow the 99% rule but it's only for the sake of this example. Let's begin the algorithm by filling up the stack and the cache, per Figure 2.6.

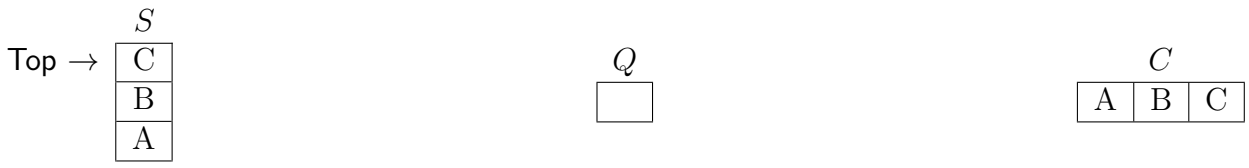

Figure 2.6. Stack S, list Q and cache C after accessing blocks A, B and C

Then we will access block D. When accessing block D the algorithm checks if we have surpassed the length of the stack, and if we have, we insert the block to the list Q[2]. We will also be inserting this block to the stack (surpassing its initial length) and it will be cached too, as shown in Figure 2.7.

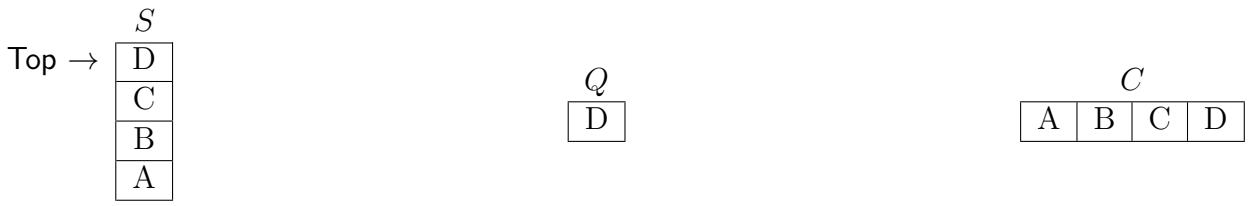

Figure 2.7. Stack S, list Q and cache C after accessing block D

From this step there are many things that can happen according to the algorithm, depending on whether we're accessing a block that already exists in the cache or not, and depending also on whether the block accessed is in the list or not. We'll use Figure 2.7 as the base step for the following examples.

If we access a new block E after having filled both the stack S and the list Q, we'll need to delete an item. For this the algorithm first checks if E is in the cache, and when it finds

[2]Note that, intuitively, we may think block A should go to the list Q, since it's the next block to be deleted. However, the paper also states that new unseen blocks can be considered HIR [11].

it's not, it checks if it belongs to S. As it doesn't, the algorithm will proceed to remove the first item from Q, remove said item from the cache C, and insert block E to Q, as well as adding it to the C, and to the stack.

Figure 2.8 depicts how the stack, the list and the cache look like after inserting block E:

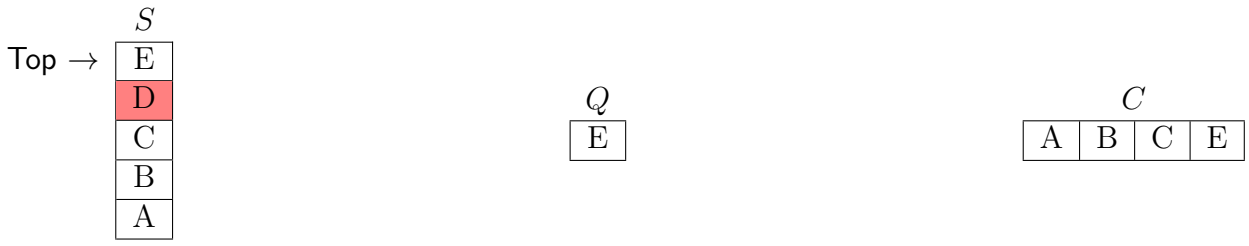

Figure 2.8. Stack S, list Q and cache C after accessing block E

The color red in the stack here represents that the block is not resident in the cache anymore, and it will be removed from the stack as soon as it reaches the bottom. A key observation here is that D's position in the stack represents its best-case IRR, i.e., its IRR if D is accessed again next. Thus by removing the lowest non-resident block in the stack S, we remove the non-resident block with the highest IRR. If block D is accessed again, it will be promoted in the stack, re-gaining its resident status.

Now let's see what happens if a block already in the cache is accessed. There are 3 cases here:

- The block belongs to both S and Q.
- The block *only* belongs to S.
- The block *only* belongs to Q.

We'll check for the first two cases what the algorithm does continuing from Figure 2.7 (forgetting block E). Let's see what happens if we access blocks A and D in that order. When accessing block A, it will be promoted, which means it is moved to the top of the stack. Then, if there are any non-resident blocks at the bottom of S, they're removed.

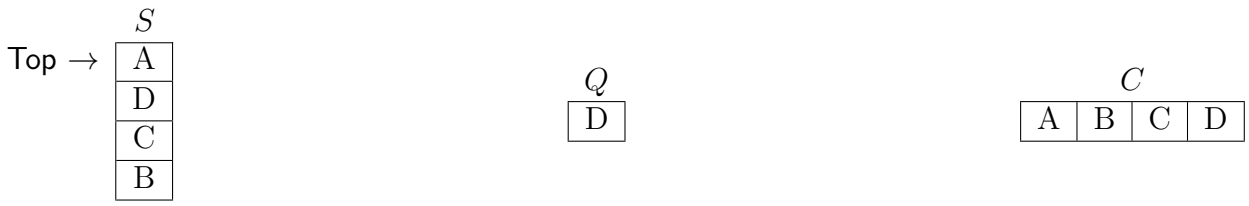

Figure 2.9. Stack S, list Q and cache C after re-accessing block A after Figure 2.7

Then from here, when accessing block D, the algorithm will check if it belongs to both S and Q. When it sees it belongs to both, it will promote D to the top of the stack S, it will

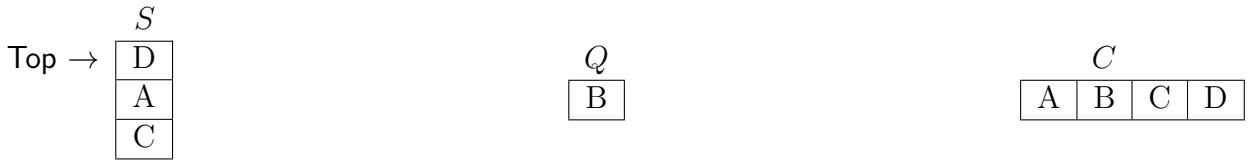

Figure 2.10. Stack S, list Q and cache C after re-accessing block D

remove D from list Q, then proceed to remove the block at the bottom of the stack and put it on the list. The end product can be seen in Figure 2.10.

In the third case, if a block were to be inserted that belongs to Q and not to S, it would be inserted to the stack and be promoted in Q, of course in the case the list could hold more than one item.

For the full LIRS algorithm one may see Appendix A.

2.2.5 Custom Policy

Finally we implemented a custom caching policy which takes into consideration the size of the results, something other policies assume is fixed. We took inspiration from the LIRS policy described in Section 2.2.4 and adapted it to consider the time the system takes to evaluate a subquery. We get into the details of this implementation in Chapter 5.

2.3. SPARQL Query Caching

Here we describe some of the work done on SPARQL query caching.

Back in 2004 Stuckenschmidt proposed a caching method based on the similarity of queries [21]. While he did not present a caching architecture, he did discuss the possibility of doing similarity-based caching, studying the cost of solving one query given another. This was done before SPARQL became the standard query language for RDF syntax.

The first idea of SPARQL query caching, as far as we acknowledge, comes from the work of Martin et al. [16] done back in 2010. Their work describes caching of query results graph patterns as entries for a database. Our work is inspired in a way by theirs, with the exception that we use only in memory caching.

In 2011 we find the work of Lampo et al. [12] proposing caching of star joins, which strikes a good balance in terms of reducing costs. The same year, Williams and Weaver [22] studied a way to cache at the HTTP level.

In 2012 Wu and Yang studied the performance of SPARQL queries doing algebraic tree caching [23], by keeping in cache an expression tree for queries (AET or Algebraic Expression Tree). Their work is inspired by the work of Martin et al. [16] and they worked with benchmarks such as LUBM [1] and BSBM [8], concluding ultimately that they could get

a reasonable amount of cache hits for benchmarks with more uniform data, but couldn't reproduce similar results for a more complex benchmark such as BSBM.

In 2013 Lorey and Naumann presented a work based on predicting what a user might ask for based on their initial queries [14]. Their aim was to modify a query such that they could retrieve important results for subsequent query sessions and concluded that they were able to predict to a certain degree what non-organic users might ask for in future sessions; however more work was needed to determine what human users might ask for in different query sessions. Being able to predict what a user would ask for next allows for some sort of pre-fetching and caching of results, but this approach is different to ours since we do not assume information about which user specifes which query.

In 2015, Zhang et al. [24] proposed a way to cache frequently used triples, instead of studying how subqueries behave when cached. These are known as "hot triples", and although innovative, this approach incurs the cost of recomputing joins. Caching data rather than query results is an interesting approach; however, most query engines already implement a buffer manager that caches raw data in memory, in which case having another such cache can be redundant. Also, it would still be necessary to compute joins over the cached data. Caching query solutions can occupy more space, but can help to reduce time even more, especially for repetitive queries.

In the same year we find the inspiration for this work: Papailiou et al.'s caching system [17]. Their work describes a complex architecture for a cache, using ideas like canonical labelling of query variables, which weren't considered previously. Furthermore, since their work runs over a system of their own creation (see [18]), we believe we improved upon their work by creating a more accessible caching system that operates over Apache Jena: a popular Open Source library for Semantic Web development. We also identify potential ways in which the caching method of Papailiou et al. [17] can potentially be improved: by adding query canonicalization to capture queries that are equivalent modulo variable names rather than just isomorphic; to compare and adapt well-known caching policies; and by creating a custom caching policy that can improve on said caching policies. Since this is the closest caching system for SPARQL that accomplishes a similar goal than ours, we believed this was the best point of comparison to prove if our work fares well against the state of the art on caching systems. However, we tried to no avail to compare these systems. Further discussion on this point is made in Chapter 7.

2.4. Query Canonicalization

Finding out when two queries are equivalent is a very hard problem [19]; it may be NP-complete or even undecidable, depending on the type of queries considered. An even harder problem is to find a canonical version for a query, that is, a generalized version that allows one to find when two queries are equivalent[3] by checking if both share the same canonical form. Salas and Hogan's work [19] creates an algorithm to canonicalize SPARQL queries,

[3]Henceforth, when we refer to equivalence of queries, we mean equivalence modulo variable names, i.e., the queries return the same results over any database modulo a one-to-one rewriting of variables

which is sound (that is, every canonical form is equivalent to its input query) for all features in version 1.1 of SPARQL, and also complete (that is, all queries that are equivalent share the same canonical form) for monotone queries, that is, queries that contain join and union features, as well as projections and the distinct keyword.

This canonicalization algorithm, in theory, may allow for a better caching engine since we don't have to worry about two different equivalent queries having different representations in the cache. As long as we cache the canonical form of the query, we can save space by not caching two or more queries that may be equivalent, while at the same time increasing the cache hit rate. More specifically, we should be able to save space by caching the results for syntactically distinct but equivalent queries only once, and we should be able to increase the cache hit rate by finding results for queries that are syntanctically distinct from all queries in the cache, but equivalent to some query.

Chapter 3

SPARQL Query Caching

In this chapter we provide the architecture for our SPARQL query caching system, as well as a high-level description for how each part works. Subsequent chapters will describe in more detail the operation of each component.

3.1. System Proposal

For our work we propose a simple design, where an input (query) is processed in a blackbox, or rather, our caching system, and proceeds to return an output, which in this case is the results for the query, while in the process keeping in memory information we can re-use in the future.

The overall architecture for this system can be seen in Figure 3.1:

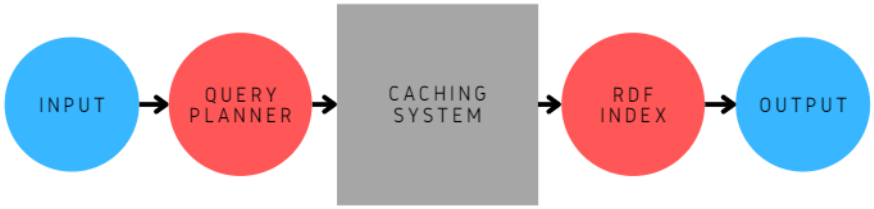

Figure 3.1. System Proposal

The input is a SPARQL query and the output are its results; the caching system is for now a black box.

As to what the box holds inside, it contains a caching query planner, which generates subqueries and interacts with the cache, checking if a new subquery is to be cached; and the subquery cache, which stores a subquery along with its results, following a caching policy.

The architecture of the caching system can be seen in Figure 3.2.

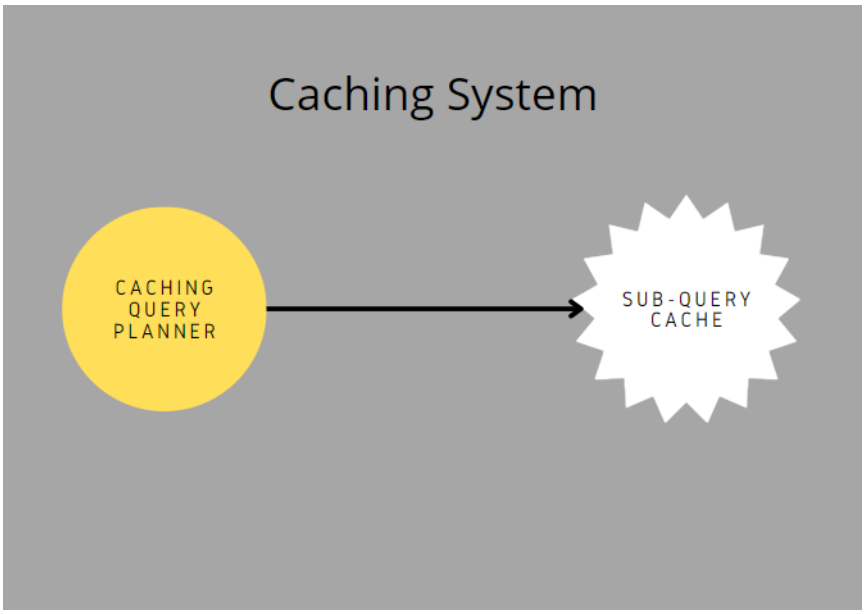

Figure 3.2. Caching System

In the following subsections we'll provide an explanation on what each part does, as well as give an example for a SPARQL query and how it is processed by every part of the system.

3.1.1 Input

As said above, the system receives as an input any SPARQL query. For this work the queries will come in a string format. Let's say the system receives the following query Q_6 for example:

```
Q6
SELECT ?country ?capital ?population
WHERE {
    ?country ex:capital ?capital .
    ?capital ex:population ?population .
}
```

The query will ask for all countries, with its respective capital cities and their populations. The following step is to pass this to the query planner.

3.1.2 Query Planner

The query planner used for this work is Apache Jena's standard query planner. Here the Jena library will optimize the query to search for the most efficient way to respond to it. If we reproduce an output for this the query would look something like this:

```
(project (?country ?capital ?population)
  (bgp
    (triple ?country <http://ex.org/capital> ?capital)
    (triple ?capital <http://ex.org/population> ?population)
  )
)
```

Something of interest here is that a projection is used for the variables, and that the query pattern forms a BGP. We will be working with BGPs in the next steps; by extracting them we can check if they are cached and then proceed to cache a BGP or remove it when necessary. Let's take a look at the caching system.

3.1.3 Caching Query Planner

This is our own query planner, and its purpose is to interact with the cache by giving it subqueries that are to be cached, and replace a subquery with its cached results if found. The caching query planner will be described in more detail in Chapter 4. Here we illustrate its operation via an example.

This planner generates subqueries of the input. The way we define a subquery is each query contained inside the bigger query by mixing its triple patterns is a subquery; more formally, a BGP is a set of triple patterns, and any non-empty subset of those triple patterns is a subquery of that BGP, so the subqueries for our example query would be the following:

```
Q7
SELECT *
WHERE {
    ?country ex:capital ?capital .
    ?capital ex:population ?population .
}
```

```
Q8
SELECT *
WHERE {
    ?country ex:capital ?capital .
}
```

```
Q9
SELECT *
WHERE {
    ?capital ex:population ?population .
}
```

Note that in this work we use `SELECT *` queries a lot, so we will be reducing the text boxes so they just contain the query's triple patterns.

Now we have to select which subqueries we will check to potentially cache. One option is to cache the powerset of triple patterns for a BGP (i.e., all possible non-empty subsets of

the BGP). However, this would give rise to $2^n - 1$ possible subqueries to check. Also some possible subqueries may not be connected via variables, giving rise to Cartesian products not present in the original query, and to potentially very large result sets. Specifically, consider a simple graph where nodes are triple patterns and edges denote that the two triple patterns share at least one variable. Given a BGP, we call it *connected* if one can reach any other triple pattern of the BGP traversing via their shared variables. Otherwise, we call it disconnected, and a Cartesian product arises across the connected components. We then only consider connected subqueries of BGPs. Still, this might lead to many BGPs, and create subqueries that generate many results. Later we will explore a variant where we will only be using subqueries that use the first triple pattern in the query after optimizing it, as it is deemed to be the most selective. So in this case, we would be using the first two queries from top to bottom, following the optimization order suggested by the Jena query planner.

Now let's make the assumption that we have saved in our cache the second subquery. The planner will check for this then replace it in the query with the subquery results, as we can see in the following:

```
(project (?country ?capital ?population)
    (sequence
        (table (vars ?var1 ?var2)
            (row [?var1 <http://www.wikidata.org/entity/Q298>] [?var2 <http://www.
                wikidata.org/entity/Q2887>])
        )
        ...
        (bgp
            (triple ?capital <http://ex.org/population> ?population)
        )
    )
)
```

The Table operator here indicates that there is a solution of two variables, and for each row each variable will be mapped to an entity. Each row of the table will then contain one of the results for the query:

```
Q8
?country ex:capital ?capital .
```

In this case, the first row is an example result containing the Wikidata entity for Chile and the entity for its capital, Santiago.

Notice that the function to retrieve the solution of this query doesn't actually retrieve the variables shown in this example, rather it will transform the variables from the canonical labels to the variables used in the original query. So the query would look more like this:

```
(project (?country ?capital ?population)
    (sequence
        (table (vars ?country ?capital)
            (row [?country <http://www.wikidata.org/entity/Q298>] [?capital <http://www
                .wikidata.org/entity/Q2887>])
        )
        ...
        (bgp
            (triple ?capital <http://ex.org/population> ?population)
        )
    )
)
```

Where ?*var*1 and ?*var*2 will be replaced by ?*country* and ?*capital*.

The other part of the caching system would be the subquery cache, which is what receives as input the subqueries.

3.1.4 Subquery Cache

The purpose of the caching portion of the system is to receive subqueries and keep their solutions in memory following a set of rules. These rules may vary, depending on what we are experimenting with. The subquery cache will be described in more detail in Chapter 5. Here we illustrate its operation with an example.

To continue our previous example, the cache may see an input of the form:

```
Q8
?country ex:capital ?capital .
```

Where this is a subquery of the query we have been working with. Then, the system would solve the query, checking if the results can be cached or not. If it is possible, depending on some conditions that will be explained later, the system will return a true Boolean value, saving the values of the results in a map. In the case that the results values already exist in the cache, or if they can't fit, the system will proceed to return a false Boolean.

Whenever we have a cache hit, which is the case when we find the same results cached already, the cache system proceeds to modify values, according to the policy it follows. For example, it may increase by 1 the value of the number of times the results have been hit, so that we know they're a frequently used result. The cache also has a removal system; when the cache is full space needs to be freed, and according to its policy, it will remove the least frequently hit results. We discuss specific cache replacement policies in Chapter 5.

The caching system is also in charge of keeping record of the constants of the queries, which are the values of the query that are not variables or blank nodes. For example, in the previous subquery, the property `ex:capital` would be a constant. To keep them in our memory, we will hash the constant using *SHA-256* encoding. This is done so we can filter triple patterns that have constants in the cache, and keep out the ones that don't, making

it easier for the query planner to canonicalize subqueries. More details will be explained in Chapter 5.

3.1.5 RDF Index

After everything has gone through the caching portion of the system, it will then proceed to return the results for the query. In the case of this system, the RDF index will be handled by the Jena library itself when solving the query, potentially joining results from the index with cached results from a subquery.

3.1.6 Output

Finally, the output of the system will give the results of the query in the form of an iterable object (the object being ResultSet, a set of results). In the case of our example, as we iterate over our ResultSet the results would look something like this:

`?country`	`?capital`	`?population`
ex:Chile	ex:Santiago	6257516
ex:Perú	ex:Lima	9943800
...	...	...

Chapter 4

Caching Query Planner

In this chapter we explore in detail the implementation for the caching query planner of our system. We will begin by looking at how we separate our query into chunks of subqueries, proceeding to handle said subqueries, filtering the ones we will use by feeding them to the caching portion of the system, ending by receiving a signal from the cache to then solve the queries. We will also look at how the system uses the cache to its benefit to answer queries faster.

The input for our query planner will be produced after processing the query through the standard query planner, which compiles the input query text then optimizes it, proceeded by an extraction of the query BGPs. To extract them we utilize the getBgps function.

4.1. Extracting BGPs

There are two instances of this function; one receives the optimized version of the query which creates a list and then recursively calls the other instance of the function, which receives as input the optimized version of the query and the list. We will be explaining this with pseudocode, expanding upon its details as we explain how this algorithm functions. We refer to this algorithm as Algorithm 1.

Algorithm 1 Pseudocode to extract BGPs

```
ArrayList<OpBGP> bgps
procedure getBgps(Op op)
    init bgps
    getBgps(op, bgps)
    return bgps
end procedure
procedure getBgps(Op op, ArrayList<OpBGP> bgps)
    ...
end procedure
```

The second version of getBgps checks if the first operation of the query is an instance of an operator that works over one instance (an operator accepting one BGP, such as filter), two instances (a binary operator over BGPs, such as unions, negation, etc.) or N instances (an n-ary operator, used for optimized BGPs where the order of triple patterns matters).

Algorithm 1 Pseudocode to extract BGPs

```
procedure getBgps(Op op, ArrayList<OpBGP> bgps)
    if (op instanceof Op1)
        ...
    end if
    else if (op instanceof Op2)
        ...
    end if
    else if (op instanceof OpN)
        ...
    end if
end procedure
```

In the first case, we call the function over the instance which is being operated. In the second case, we call the function over both instances; and in the last case, we iterate over multiple instances.

Algorithm 1 Pseudocode to extract BGPs

```
procedure getBgps(Op op, ArrayList<OpBGP> bgps)
    if (op instanceof Op1)
        getBgps(((Op1)op).getSubOp(), bgps)
    end if
    else if (op instanceof Op2)
        getBgps(((Op2)op).getLeft(), bgps)
        getBgps(((Op2)op).getRight(), bgps);
    end if
    else if (op instanceof OpN)
        OpN opn = (OpN) op
        for (Op sop : opn.getElements())
            getBgps(sop, bgps)
        end for
    end if
end procedure
```

There are two important cases to check in the getBgps function. First, there is the case we find a BGP operator. In this case, we simply add the BGP to our list and end the recursive call.

Algorithm 1 Pseudocode to extract BGPs

```
procedure getBgps(Op op, ArrayList<OpBGP> bgps)
    if (op instanceof OpBGP)
        bgps.add((OpBGP)op)
    end if
    else if (op instanceof Op1)
        getBgps(((Op1)op).getSubOp(), bgps)
    end if
    else if (op instanceof Op2)
        getBgps(((Op2)op).getLeft(), bgps)
        getBgps(((Op2)op).getRight(), bgps);
    end if
    else if (op instanceof OpN)
        OpN opn = (OpN) op
        for (Op sop : opn.getElements())
            getBgps(sop, bgps)
        end for
    end if
end procedure
```

On the other hand, we may find path operators. These represent property paths, and we have decided to deal with them by creating a BGP instance, composed by the subject of the path, its object, and the property path itself encoded. We encode it by adding a prefix and hashing the string that represents the property path itself. For hashing this, we use SHA256. We do not consider equivalences of path expressions, but rather only capture caching for syntactically identical paths.

Algorithm 1 Pseudocode to extract BGPs

```
procedure getBgps(Op op, ArrayList<OpBGP> bgps)
    if (op instanceof OpBGP)
        bgps.add((OpBGP)op)
    end if
    else if (op instanceof OpPath)
        TriplePath path = ((OpPath)op).getTriplePath();
        BasicPattern bp = new BasicPattern()
        Triple nt = Triple.create(path.getSubject(),
                                  encodeAsUri(path.getPath()),
                                  path.getObject());
        bp.add(nt)
        bgps.add(new OpBGP(bp))
    end if
    else if (op instanceof Op1)
        getBgps(((Op1)op).getSubOp(), bgps)
    end if
    else if (op instanceof Op2)
        getBgps(((Op2)op).getLeft(), bgps)
        getBgps(((Op2)op).getRight(), bgps);
    end if
    else if (op instanceof OpN)
        OpN opn = (OpN) op
        for (Op sop : opn.getElements())
            getBgps(sop, bgps)
        end for
    end if
end procedure
```

4.2. Extraction of Subqueries

Once the input is processed, we proceed to extract our subqueries. For our work, as was justified in Section 3.1.3, we decided to get each subquery that involves a contiguous subsequence containing the first triple pattern once the query has been optimized. We do this according to the pseudocode shown in Algorithm 2. Note that the input is a list of BGPs as each input query can contain multiple BGPs.

Algorithm 2 Pseudocode to extract subqueries

```
int n, int y, ArrayList<OpBGP> output, BasicPattern bp
procedure getSubQueries(ArrayList<OpBGP> input)
    n = input.size()
    init output
    for (x = 0; x < n; x++)
        y = input.get(x).size()
        init bp
        for (i = 0; i < y; i++)
            bp.add(input.get(x).get(i))
            output.add(bp.copy())
        end for
    end for
    return output
end procedure
```

Following the previous code, for example the following query, Q_{10}:

```
Q10
?country ex:name ?name .
?country ex:language ?language .
?country ex:capital ?capital .
?capital ex:population ?population .
```

Would have the next four subqueries:

```
Q11
?country ex:name ?name .
```

```
Q12
?country ex:name ?name .
?country ex:language ?language .
```

```
Q13
?country ex:name ?name .
?country ex:language ?language .
?country ex:capital ?capital .
```

```
Q14
?country ex:name ?name .
?country ex:language ?language .
?country ex:capital ?capital .
?capital ex:population ?population .
```

Afterwards, we sort our list of subqueries from the biggest one to the smallest one. This way, when we check which subquery to cache, if we cache one of the bigger ones, there is no

need to check any further since the bigger subquery will give us more information on what is efficient to keep in the cache.

Before doing this check, we have to remove any subquery that is disconnected via variables. That is to say, any triple patterns that don't share at least one variable have to be removed, since that would imply caching a disconnected sub-graph from the original query.

Since this query was connected, that is, all triple patterns generate a Cartesian product between themselves, we don't have to worry about this in our example. However, say the query is disconnected; it might cause the subqueries to be disconnected as well.

Taking some of the triple patterns we had before, imagine we had the query:

```
Q15
?country ex:name ?name .
?capital ex:population ?population .
```

This would be a disconnected query. As we can see, both triple patterns don't share any variables, making this a disconnected BGP. For our work, we won't be submitting this subquery to check if it is cachable, since it would produce many results. We will, however, consider a subquery with a single triple pattern (the first triple pattern to be evaluated after optimization of the BGP).

4.3. Canonicalising subqueries

Take one of the previous subqueries, Q_{12}, and another congruent query as follows:

```
Q12
?country ex:name ?name .
?country ex:language ?language .
```

```
Q16
?c ex:language ?l .
?c ex:name ?n .
```

Both will have the same results modulo variable names, but the cache could miss this since the variable names are different. When we use Salas and Hogan's canonicalisation algorithm, we can use their feature *SingleQuery* which delivers a canonical form of the query. For this example, the function returns the following for both of the aforementioned queries:

```
Q17
SELECT DISTINCT  ?v2 ?v0 ?v1
WHERE {
    ?v1  <http://dbpedia.org/language>  ?v2 ;
    <http://dbpedia.org/name>  ?v0
}
```

Taking our example from the previous section, we proceed to canonicalize this list of subqueries. For our example, they would end up looking something like this in BGP form:

```
(bgp (triple ?v0 <http://example.org/name> ?v1))
```

```
(bgp
  (triple ?v1 <http://dbpedia.org/language> ?v2)
  (triple ?v1 <http://dbpedia.org/name> ?v0)
)
```

```
(bgp
  (triple ?v3 <http://dbpedia.org/capital> ?v1)
  (triple ?v3 <http://dbpedia.org/language> ?v0)
  (triple ?v3 <http://dbpedia.org/name> ?v2)
)
```

```
(bgp
  (triple ?v1 <http://dbpedia.org/capital> ?v2)
  (triple ?v1 <http://dbpedia.org/language> ?v0)
  (triple ?v1 <http://dbpedia.org/name> ?v3)
  (triple ?v2 <http://dbpedia.org/population> ?v4)
)
```

After this, the system starts checking which of these canonicalised subqueries may enter the cache. Starting with the biggest (latter) one, each subquery will enter the caching portion of the system which will return if it is a hit or a miss. In either case, the query will be then evaluated, possibly replacing a subquery with its cached results in the case of a cache hit, and its results will be returned. We will discuss this further in Chapter 5.

4.4. Query Evaluation

When evaluating the query, if we find any subquery that is in the cache we have to replace it with its corresponding results; that is, the final query to be evaluated will be a mix of the results of the cached subqueries and the triple patterns which are not present in the cache. The query planner interacts with the cache by asking if there are any subqueries from its input query that may be replaced with its solutions. For this, we use a *transform* class, which will allow us to convert our query into something that benefits us to answer using a visitor pattern that allows us to transform the query's algebra.

The system begins by taking the optimized form of the query and passing it through the transform class, which will reduce down the query to chunks until getting to a BGP. Once it gets to a BGP it will be taken by our custom transform class, which manipulates the BGP to replace its contents if necessary.

Our custom transform class begins by filtering triple patterns, so that we don't have to handle triples with only constants or only variables. Once we've filtered our BGP, we'll

extract all of its sub-BGPs, which is basically every subquery of the query that comes with the BGP; then we check for each of these subqueries if one of them is in the cache. If we get a cache hit, we call the *retrieve* function, which basically replaces the subquery with its solutions. Note that before checking if said subquery is in the cache, we have to canonicalize it, and we keep a variable map so once we replace the subquery with its solutions, we can go back to using the original variables the query used.

Once this process is done we join these BGPs with the ones that had constants and return it, which is proceeded to be transformed into a query, optimized, and answered using the Apache Jena library.

To exemplify this process let's take Q_{12}:

```
Q12
?country ex:name ?name .
?country ex:language ?language .
```

Say in the cache we have saved the following list entry:

```
((bgp (triple ?v0 <http://example.org/name> ?v1)),
(table (vars ?v0 ?v1) (row [?v0 <http://www.wikidata.org/wiki/entity/Q298>] [?v1 "Chile
    "])
(row [?v0 <http://www.wikidata.org/wiki/entity/Q419>] [?v1 "Peru"])...)
```

Basically what this algorithm does is replace the subquery Q_{11}:

```
Q11
?country ex:name ?name .
```

With the results from the cache. In the end, the final operation to be evaluated will look something like this:

```
(sequence
    (table (vars ?v0 ?v1)
        (row [?v0 <http://www.wikidata.org/wiki/entity/Q298>] [?v1 "Chile"])
        (row [?v0 <http://www.wikidata.org/wiki/entity/Q419>] [?v1 "Peru"])
        ...
    )
    (bgp
        (triple ?v0 <http://dbpedia.org/language> ?v2)
    )
)
```

Which is then transformed into a query, Q_{18}, and then we obtain its results by using the corresponding Jena library features. The algebra previously shown is equivalent to the following query in concrete SPARQL syntax using the `VALUES` feature.

```
Q18
SELECT *
WHERE {
    ?country ex:language ?language.
    VALUES (?country ?name)
    {
        (<http://www.wikidata.org/wiki/entity/Q298> "Chile")
        (<http://www.wikidata.org/wiki/entity/Q298> "Peru")
    }
}
```

In the next chapter, we will proceed to explain the caching portion of the system in detail.

Chapter 5

Cache Implementation

The caching portion of the system has the role of keeping profitable subquery solutions in memory, as well as communicating with the query planner whenever a query has been succesfully cached. It also has to stay at a reasonable size, so it's also in charge of deleting unnecessary items from its memory.

In this chapter we will describe how the query planner sends repeated subqueries to the cache, which is followed by a description of the implementation of the cache, as well as how it deletes its items so as to keep a reasonable size by following what are known as *cache policies*.

5.1. Query planner-Cache Interaction

Once we have got our subqueries, the system starts the process of checking which of these canonicalized subqueries may enter the cache. We begin by initializing some lists for each query we check: one is called *checkedBgpSubQueries* and it will contain the BGPs we have already checked; and the other one will contain the BGPs that we have cached, known as *cachedBgpSubQueries*. There will also be a list called *seenOnceBgpSubQueries* which is initialized as soon as the class is instantiated, and it will contain BGPs that we have seen once in one query, but not two times in two different queries; this list has a fixed capacity that we will assume here to be $10,000$.

The idea of the algorithm to follow is that we will attempt to cache canonicalized subqueries we have seen twice in two different queries in order to avoid the overhead of filling the caching system with the results for every subquery encountered during warm-up (note, for example, that all caching policies fill their buffers during a warm-up phase). This decision is arbitrary, as we can experiment with caching subqueries we have seen more times in different queries, but we believe we can get a reasonable number of cache hits by checking if we see the same subquery twice without getting too excessive. We will proceed to explain the algorithm to search for repeated subqueries. Its pseudocode is shown in Algorithm 3.

Algorithm 3 Pseudocode to check for repeated subqueries

```
list seenOnceBgpSubQueries, list checkedBgpSubQueries, list cachedBgpSubQueries
procedure INIT()
    init seenOnceBgpSubQueries
    init checkedBgpSubQueries
    init cachedBgpSubQueries
end procedure
procedure checkBgps(list bgps)
    init checkedBgpSubQueries
    init cachedBgpSubQueries
    for (OpBGP bgp : bgps)
        checkBgpQueue(bgp)
    end for
end procedure
procedure checkBgpQueue(OpBGP bgp)
    if checkedBgpSubQueries.contains(bgp) then
        end procedure
    end if
    if seenOnceBgpSubQueries.contains(bgp) then
        cacheBgpQuery(bgp)                          ▷ This attempts to cache bgp
        checkedBgpSubQueries.add(bgp)
        cachedBgpSubQueries.add(bgp)
        seenOnceBgpSubQueries.remove(bgp)
        end procedure
    end if
    checkedBgpSubQueries.add(bgp)
    if seenOnceBgpSubQueries.size() < 10000 then
        seenOnceBgpSubQueries.add(bgp)
    else
        seenOnceBgpSubQueries.remove(0)
        seenOnceBgpSubQueries.add(bgp)
    end if
end procedure
```

Take the following queries:

```
Q15
?country ex:name ?name .
?capital ex:population ?population .
```

```
Q19
?country ex:name ?name .
?person ex:livesIn ?country .
```

Assuming that the optimized version of these bpgs keeps the triple patterns in their input order, we see here that we'll be working with four subqueries, which are the following:

```
Q11
?country ex:name ?name .
```

```
Q15
?country ex:name ?name .
?capital ex:population ?population .
```

```
Q11
?country ex:name ?name .
```

```
Q20
?country ex:name ?name .
?person ex:livesIn ?country .
```

As we can see, there is one subquery that is repeated, that being Q_{11}, which we will attempt to cache.

For each query, we re-initialize the lists *checkedBgpSubQueries* and *cachedBgpSubQueries*.

We begin by checking if the first subquery is contained in the *checkedBgpSubQueries* list. Running the algorithm, this would be the biggest subquery from the first query, which is Q_{15}.

When converted to BGP form, Q_{15} looks like this:

```
(bgp
    (triple ?country <http://example.org/name> ?name)
    (triple ?capital <http://example.org/population> ?population)
)
```

Keep in mind that actually the BGP form will have its canonicalized variables, but to better understand the example we will keep the names this way.

Since this isn't in the mentioned list, we proceed to check if it is contained in the *seenOnceBgpSubQueries* list. Since this is the first subquery we're checking it's not there

either, so it will added to to both lists then the procedure will move on to the next subquery.

For our example case, our next BGP to check is:

```
(bgp
    (triple ?country <http://example.org/name> ?name)
)
```

We check if this BGP has already been seen in the same query. For our example it is clear we haven't. If not, we can move on to check if this BGP is equal to one that has been seen in a previous query, by comparing it to the contents of the *seenOnceBgpSubQueries* list. When we compare our current BGP to the only other one we have checked, we can see they are different just by seeing one has one triple pattern, and the other one has two.

In case we see the same BGP twice, as we already discussed, we proceed to attempt to cache it. Once it's been attempted to be cached, we remove the BGP from *seenOnceBgpSubQueries*, since it's been seen twice and passed to the cache, and we add it to both of our lists *checkedBgpSubQueries* and *cachedBgpSubQueries*, to then end the procedure.

In this part of the procedure, we add the current BGP to *checkedBgpSubQueries*. If we haven't seen this BGP previously, we add it to the *seenOnceBgpSubQueries* list. We can only store a limited amount of BGPs in memory, so we assume for now that we keep a number of ten thousand BGPs at a time. If we exceed that limit, we remove the oldest item in the list, then add the newer one, following an LRU policy. We justify the size of this list in Chapter 7.

In our example, since we haven't seen this BGP previously, we proceed to add it to the *checkedBgpSubQueries* list. Then, since our *seenOnceBgpSubQueries* list has only a size of one so far, we proceed to add our BGP to it as well.

So far, our lists look like this:

```
seenOnceBgpSubQueries: {
(bgp
    (triple ?country <http://example.org/name> ?name)
    (triple ?capital <http://example.org/population> ?population)
),
(bgp
    (triple ?country <http://example.org/name> ?name)
)
}
```

```
checkedBgpSubQueries: {
(bgp
    (triple ?country <http://example.org/name> ?name)
    (triple ?capital <http://example.org/population> ?population)
),
(bgp
    (triple ?country <http://example.org/name> ?name)
)
}
```

```
cachedBgpSubQueries: {}
```

Moving on to the next query, we re-initialize the last two lists as seen above. We check the next BGP:

```
(bgp
    (triple ?country <http://example.org/name> ?name)
    (triple ?person <http://example.org/livesIn> ?country)
)
```

Since *seenOnceBgpSubQueries* is not empty, and *cachedBgpSubQueries* is empty, and our list of checked BGPs is empty, the algorithm skips to the part of adding this BGP to the lists *checkedBgpSubQueries* and *seenOnceBgpSubQueries*.

The last BGP we check is:

```
(bgp
    (triple ?country <http://example.org/name> ?name)
)
```

We repeat the same process, checking if it was previously seen in another query. Since we have, the algorithm then attempts to cache this BGP, then adding it to both lists we have: *checkedBgpSubQueries* and *cachedBgpSubQueries*.

We also remove it from the *seenOnceBgpSubQueries* list. The final lists look like this:

```
seenOnceBgpSubQueries: {
(bgp
    (triple ?country <http://example.org/name> ?name)
    (triple ?capital <http://example.org/population> ?population)
),
(bgp
    (triple ?country <http://example.org/name> ?name)
    (triple ?person <http://example.org/livesIn> ?country)
)
}
```

```
checkedBgpSubQueries: {
(bgp
    (triple ?country <http://example.org/name> ?name)
    (triple ?person <http://example.org/livesIn> ?country)
),
(bgp
    (triple ?country <http://example.org/name> ?name)
)
}
```

```
cachedBgpSubQueries: {
(bgp
    (triple ?country <http://example.org/name> ?name)
)
}
```

The query planner has a function that tells the cache to attempt to save the results of a subquery. First, it builds the query from the BGP that it receives, which is canonicalized, and it will always be a `SELECT` and a project-all variables query. This means that if the query is of the form:

```
Q21
SELECT ?country ?name
WHERE {
    ?country ex:name ?name .
}
```

Its canonicalised subquery will be passed to the cache like this:

```
Q22
SELECT *
WHERE {
    ?v0 ex:name ?v1 .
}
```

It is important to note that whenever we transform to a canonicalized query we will be working with a `SELECT *` type of query. Other queries that are not of the `SELECT` type are not supported by our system right now.

Once we have this query built we will do two things with it. We will extract its one and only BGP, and we will solve it. With these two things, we proceed to ask the cache to attempt to cache the query's BGP with its results. This will return a Boolean result, and if it is true, we will also keep in the cache two things: the query's constants, and an approach for how much time the query takes to solve, which is data we can use later on.

Now, we will describe the cache's implementation.

5.2. The Cache Classes

A basic map for how the system is implemented can be seen in Figure 5.1. Note that the implementation is in Java, more details of this can be read in Chapter 6.

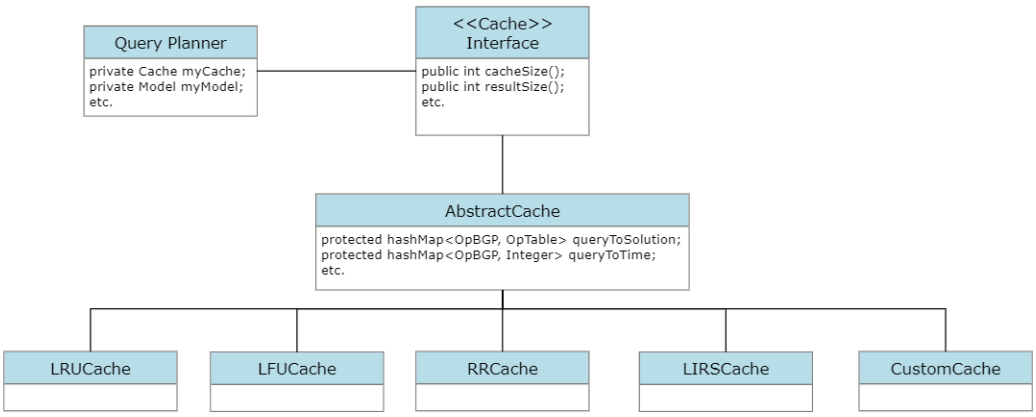

Figure 5.1. UML Graph of the Query Planner and the Caching Portion of the System

There is more to this for the whole system, but this is the core of how the query planner and the Cache interface interact. The cache classes inherit operations from an abstract class called AbstractCache, which implements the Cache interface. We will explain the children classes in the next section; for now we will focus on the interface and the abstract class.

5.2.1 Cache Interface

This interface defines the basic functions we will need to use from the caching classes. There are many of these functions, but most importantly we will be using the cache, cacheConstants, isBgpInCache, and the retrieveCache functions.

5.2.2 The AbstractCache Class

This is the basic implementation of our cache. The cache itself is a map which maps BGPs to its results, which come in the form of OpTable instances. Some important things that are programmed here are a limit to how many items the cache can hold at a time, as well as how many results a table can hold; the maps for the constants that can be subjects, predicates or objects of a query; the query to time map which holds how much time we expect a query to take to be responded; and most importantly, the query to solution map, which holds the results of the subqueries we cache.

We will proceed to describe how the most important functions, named in the previous subsection, are programmed.

The cache Function

The cache function, in combination with another function called addToCache, are in charge of creating a new item for the query to solution map. The addToCache function is related to the children classes which implement caching policies, so we will see it later in Section 5.3. The cache function is in charge of transforming the results set of the incoming subquery into a table, which will be easier to manipulate for other operations.

An important thing to keep in mind here is that we limit the number of results a table can hold up to ten thousand. We tried a threshold of a hundred thousand results but this slows down the system quite a lot. This is necessary since the standard caching policies typically assume blocks (e.g., disk sectors) of equal, fixed size. In our case, the number of solutions we cache for a single query may vary, taking up a variable amount of memory. In fact, the amount of solutions and memory that can be occupied by one caching item (subquery) is, in theory, unbounded. Hence we apply this upper-bound to ensure that available memory is not immediately occupied by a few subqueries with very many results.

Another important aspect is that we also limit the number of total results the cache can hold, as well as the number of entries it can have. For our experiments we usually used one hundred items and one million total results as a threshold; more details can be seen in the experiments chapter later on.

Whenever we call the cache function, and after it creates the table, we call a function called *clean*, which will call the delete function if we are above the limit of these thresholds we defined. More details about how the delete function will be explained in a later section, related to the cache policies implemented.

The cacheConstants Function

The basic functionality of this method is to keep in memory the constant values from the queries; this way when we're trying to retrieve results from the cache we can retrieve them in an easier way, without dealing with BGPs that have constant values.

First, since we work with subqueries that correspond to BGPs, which can be represented as asterisk queries (`SELECT` all queries or `SELECT *`), we convert the input query into one. Then, we extract all the triples of the query as an iterator; when iterated over, we ask if each subject, predicate or object is a variable or a blank node. If they are neither, we map said subject/predicate/object to a set of queries containing them, which are hashed.

The isBgpInCache Function

This is one of the easier methods, which basically asks the cache if there is an input BGP in the queryToSolution map. If it is in there, this function returns a Boolean value of true.

The retrieveCache Function

For the implementation of the cache we use a map instance, which maps items (in the form of instances of *OpBGP*) to their solution (in the form of instances of *OpTable*). For the sake of the experiments we also define a limit to the number of results the cache can hold, and the number of items it can hold. Each time an item is added, how it is added is going to depend on the policy the type of cache follows, but most of them will insert the item with its respective solution to the map, respecting the limit of items and results the cache can hold. To retrieve from the cache, we retrieve the necessary table, and build it from scratch using new variables as the original query which is going to use these results requires.

5.3. Cache Policies

There are several ways to implement a cache, some more optimal than others depending on the environment in which it is going to be used. For this work, we implement five different kinds of cache, which follow different policies to decide which item is going to be deleted once it is filled.

Below are the policies that were implemented.

5.3.1 Least Recently Used (LRU)

For the implementation of this policy, we define a map and a number *LRUHit* which gives an access number to each item that either is inserted to the cache or is retrieved. When we insert an item A, it receives a number 1 which means this is the first item that accessed the cache. Then, if we insert an item B, it receives a 2. Followed by this, if we retrieve item A, we give item A an access number 3, replacing the 1 it had. Assuming we have a cache of size two, if we try to insert an item C the cache has to decide which item it deletes. Since B has the lowest *LRUHit* number, meaning it was the item that hasn't been accessed the longest, the cache will delete item B. This is implemented with ease by keeping a map, as previously mentioned, with each item and its access number, sorted by *LRUHit* from lowest to biggest. Then as we need to delete an item, we delete the first item from the map, which has the lowest access number, taking a time of $O(1)$.

5.3.2 Least Frequently Used (LFU)

To implement this we create a map of items and their respective number of access times. Each time an item is inserted to the cache it creates an element containing the item and an access number 1. Every time the item is attempted to be inserted or is retrieved, its access number grows by 1. So in the case we have four items in a cache of size four, the items being A, B, C and D, with access numbers of 5, 3, 3 and 10 respectively, and we want to insert an item E. We can keep these access times sorted, so when we look up the cache for the

item with the lowest access number we can make this operation be $O(1)$. In the case of a tie, we delete the first item with the lowest access number that we find, so in the case of this example, the cache would decide to delete item B to make space for item E.

5.3.3 Random Replacement (RR)

To implement this we simply create a random number between 0 and the size of the cache minus one, representing the index of the item we will delete, and by keeping an *ArrayList* of the items in cache, we can call the remove function for the *ArrayList*, get the item we need to remove then proceed to remove it from our cache. This operation is of $O(1)$.

5.3.4 Low Inter-reference Recency Set (LIRS)

The algorithm defined by this policy is quite complex, where we refer the reader to Section 2.2.4 for a detailed example and to Appendix A for the full algorithm; the implementation then works as follows:

1. Each time an item is accessed and while the threshold of 99% has not been reached, the item is inserted to the cache and is also stacked in S.
2. Once the threshold has been reached, if the item accessed does not belong to the stack it is stacked and then if the list Q is full, the first item of Q is removed from it and from the cache. We then proceed to add the item to Q and then to the cache.
3. If the item accessed belongs to the stack, we *promote* it, by making it a *resident* LIR block, promoting it to first in the stack and adding it to the cache. Then we run the next procedure to remove an element from the cache: if Q is full, we remove its first item from the list and from the cache. Then, we remove the item at the bottom of the stack, we *prune* the stack (that is, remove all items from the bottom that are not a LIR *resident* block), then move the bottom item from the stack to the end of the list Q.
4. If the item accessed already belongs to the cache, there are three possibilities. One is that the item belongs to S and not to Q. In this case, we *promote* the item then *prune* the stack.
5. If the item accessed belongs to both S and Q, the item is *promoted* and is removed from Q. We then run the procedure described in (3).
6. Lastly, if the item accessed belongs only to Q, the item is stacked and is also *promoted* in Q, this is, moving it to its last spot as to keep it for a longer time as a *resident* block.

For the implementation of this policy, we create a *block* instance, which describes if an item is going to be LIR or not, and if it's going to be *resident* or not. We also implement both stack S and list Q as maps, containing both an item (represented as an *OpBGP* instance),

and its respective block. For S, the bottom of the stack is represented by index 0 of the map, whereas for Q, index 0 represents the oldest item in the list, that is, the ones that are to be removed from the cache.

We follow the algorithm presented above whenever we have to insert an item or retrieve it, and for the removal of items we delete the first item in Q from it and the cache.

5.3.5 Custom Caching Policy

Finally, we create a custom caching policy inspired by LIRS as described above, but which takes into consideration the amount of results the cache deals with. Notably the previous caching policies were defined for scenarios where each element is assumed to be the same size, and to have the same cost to retrieve in the case of a cache hit. However, in our scenario, a SPARQL query may vary in terms of the number of results it generates, and the cost of generating those results in the case of a cache hit. Thus we try to prioritize keeping cached subqueries that do not have many results, but take a long time to generate, which we measure in the following way. We take into consideration a factor of μ/t, where μ equals to a prediction of time the query will take to run, and t equals to the total amount of results the query returns. The factor μ/t thus represents an estimate of how long it takes to generate each result of the subquery, where we give higher priority to keeping queries with a higher such value.

For this occasion predicting the total time a query is going to take before executing it fully takes a longer time, since we need to read each and every result for each query to calculate how much time it takes to evaluate it. We address this issue by only running one solution and extrapolating the total amount of time it is going to take to resolve by multiplying this number by the total amount of results the query has. So μ equals to $\sigma \times t$, where σ equals to the amount of time the query takes to return a single answer, and then we multiply that by $1/t$, which in total is just σ.

Therefore, the only number we have to consider for this policy is the time the query takes to return one result. We use this to determine which item from the cache we have to delete, so we will be removing the one which takes the least amount of time, since that query is probably not as useful to maintain as the ones that take more time; it's more efficient to keep in cache the items that cost more. Using this result may cause an issue: that only reading the first result doesn't actually reflect how much time on average the query takes to evaluate one result. We further justify the use of this in Chapter 7.

Chapter 6

Implementation

In this chapter we give an overview of the code and we talk about the libraries we used.

This work was programmed in Java using Runtime Environment 1.7. Other external libraries we use are Apache Jena version 4.4.0, and Qcan 1.1 created by Salas and Hogan [19]. The decision to implement the Caching System in Java allows easy integration with the existing libraries.

The main packages of this work consist of the following:

1. bgps: This contains the code pertinent to extracting the BGPs of each query.
2. cache: Contains the main cache classes and the cache policies discussed in Chapter 5.
3. parser: Contains the Jena query parser.
4. queries: Contains the main controller of the Caching System, which extracts subqueries, sends the order to cache and then returns the query results.
5. tests: The package we use to run code tests.
6. transform: This contains the Transform classes which allow to manipulate the query algebra. Here the queries interact with the cache when we get a hit.

Other packages such as data_reader and utils contain other helpful files which we used during this project, though they aren't reflected in the functionality of the final work. Below, we can see class diagrams for each of our main packages.[1]

6.1. Bgps

This only uses the `ExtractBgps` class, which was discussed in Chapter 4. In Figure 6.1 we can see its diagram which contains its main methods.

[1]The full code can be found in: `https://github.com/gchandia/JenaCachingMaven`

Figure 6.1. Class Diagram for the bgps package

6.2. Cache

As we discussed further in Chapter 5 the caching classes implement an interface and use a parent abstract class. Figures 6.2 and 6.3 contain an overview of how these classes interact with each other.

Figure 6.2. Class Diagram for the cache package: Cache and AbstractCache

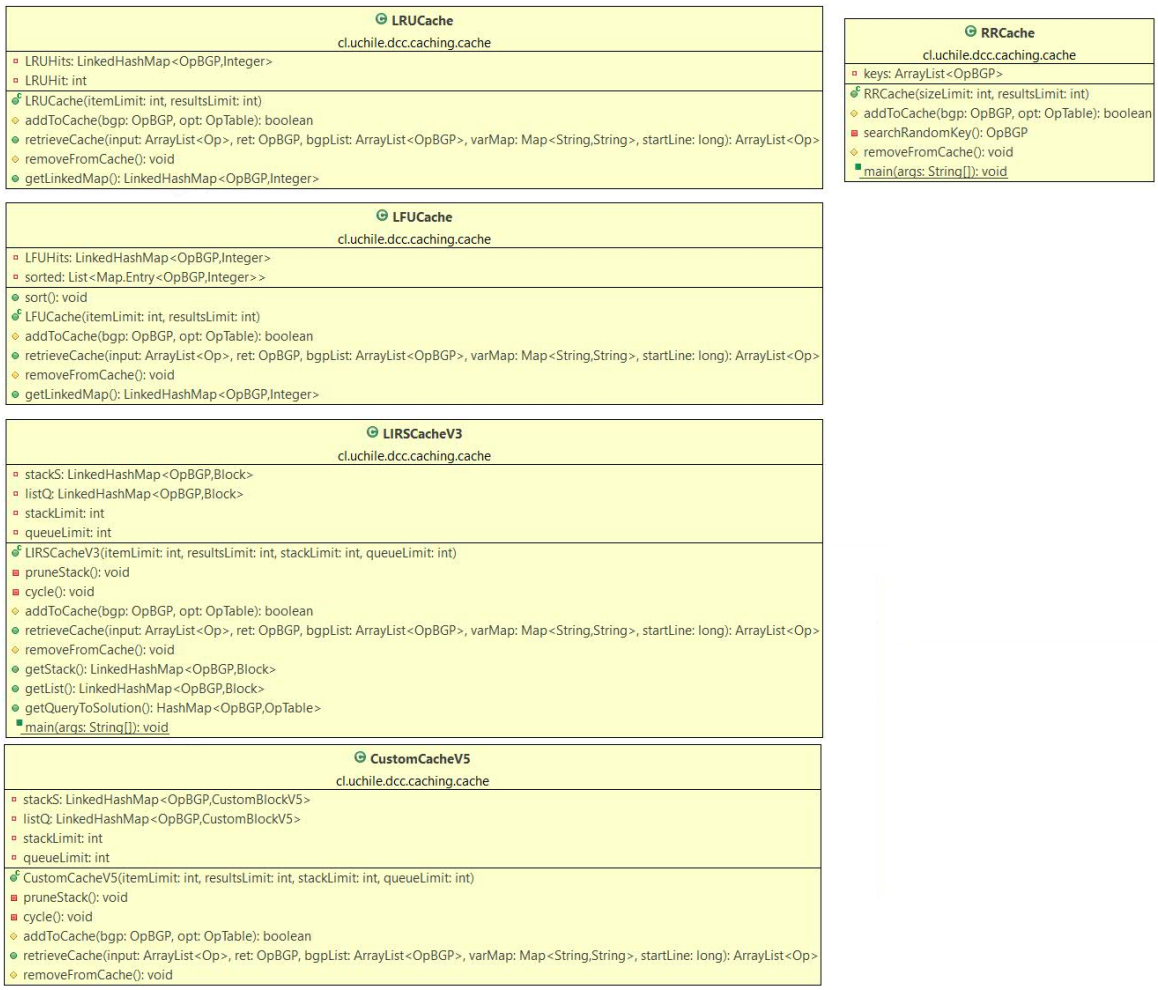

Figure 6.3. Class Diagram for the cache package: Caching policies

Figure 6.3 shows the classes for the different caching policies implemented for this work. They inherit all properties from AbstractCache and implement its abstract functions.

6.3. Parser

This mainly includes the `Parser` class, which helps parse text into SPARQL queries. Figure 6.4 shows its class diagram.

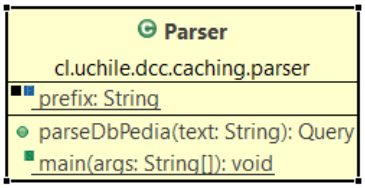

Figure 6.4. Class Diagram for the parser package

6.4. Queries

This package contains the main controller. Its class diagram can be seen in Figure 6.5.

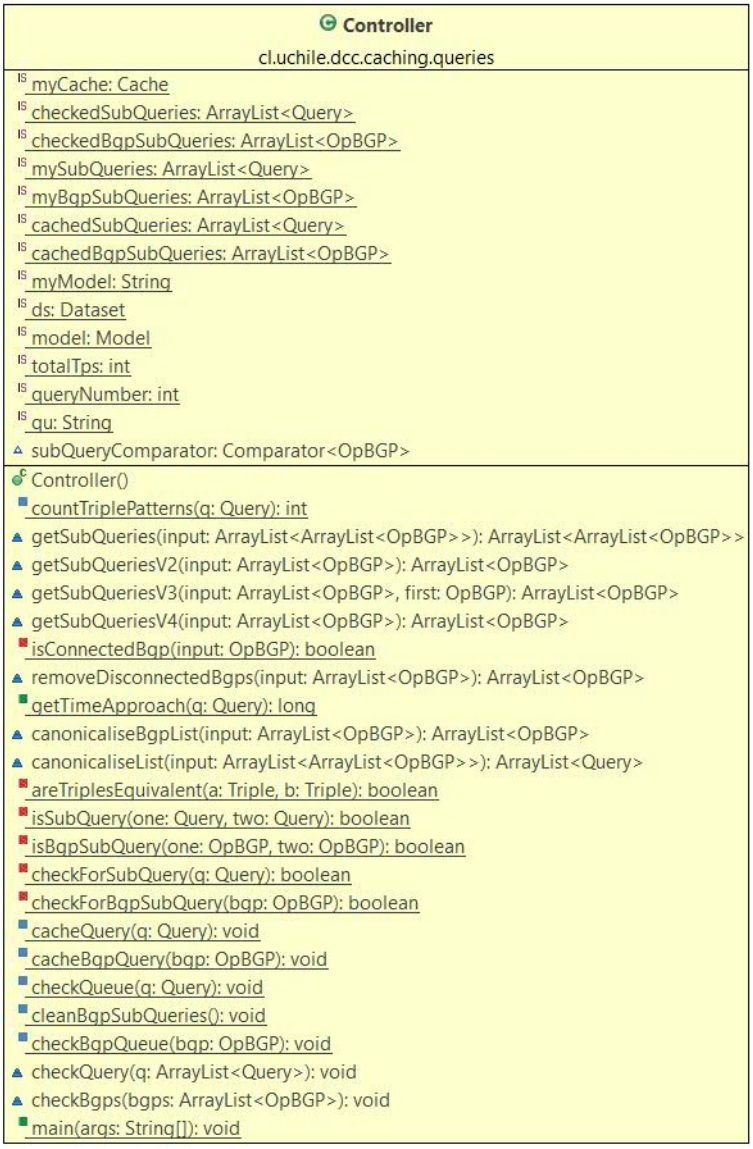

Figure 6.5. Class Diagram for the queries package

6.5. Transform

This package contains all the functions to manipulate query algebra and which allow to interact with the cache. The class diagram can be seen in Figure 6.6.

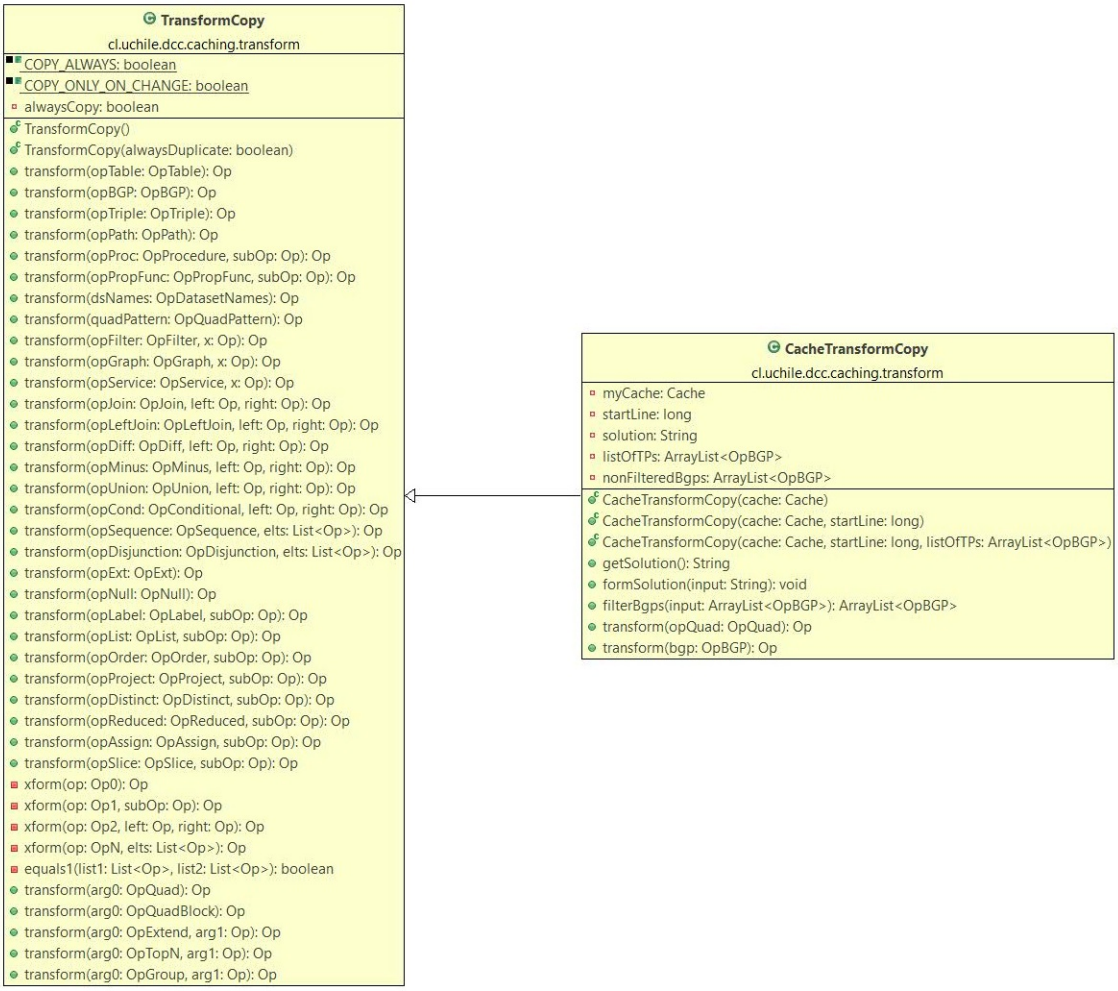

Figure 6.6. Class Diagram for the transform package

Chapter 7

Results

For this work we now address the research questions introduced in Chapter 1. As a reminder, they're the following:

- Is it possible to create a caching system that improves on query performance for SPARQL, being able to decrease overall response times in a static data environment?
- Can we define metrics to determine profitable subqueries?
- Is it possible to compare this new system against the state of the art caching systems for SPARQL, in terms of the number of profitable queries and average response time? If so, does it fare well?

In this chapter we describe how we define experiments to answer these questions, using the Caching System we proposed in the previous chapters.

7.1. Experiments

Since our system reads SPARQL queries we need to use as input logs of queries. For our experiments we considered real-world queries, all made against the Wikidata query service. All of these logs can be taken from `https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en`. In particular, we used interval 2 for these experiments, which contains the queries our system could parse with the fewest amount of errors. Though we could consider all intervals, this would considerably extend the time needed to run experiments, making it costly to evaluate all configurations.

7.1.1 Dataset

For works similar to ours there exist benchmarks such as LUBM [1] or BSBM [8] which allow to generate data graphs and queries to different ends, such as proving if a SPARQL endpoint

works well enough.

For our work, however, we intend to prove our system works in a real-world setting, and thus we need a more realistic approach, such as using queries directly taken from SPARQL endpoints and run them in real time replicating how they were originally executed.

To accomplish that end, we use Wikidata logs which contain a lot of information about how they were run, such as: the query itself, timestamp, source category (whether it was organic or made by a bot) and user agent (whether it comes from a browser-like agent or a bot-like agent). Our parser class shown in Chapter 6 allows us to extract only the information we need, which is the query. This comes in a string format, and the parser allows us to convert it to a format we can use and manipulate. Queries are listed in chronological order, allowing us to replay the queries as they were evaluated on the live service.

7.1.2 Limitations

For our experiments, we took $200,000$ queries which we pre-processed by removing clauses we deemed unnecessary (e.g SERVICE or others that are either non-standard extensions implemented in the Wikidata query service for finding labels, or calls to remote endpoints, and thus out of scope). We also only work with `SELECT` type queries as we indicate in Section 5. Our system is limited by only being able to receive these type of queries as input, and also any canonicalized subquery will be transformed to a `SELECT *` type query.

7.2. Machine Specifications

All experiments were ran on an external machine of 12-core Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz, 32 GB of RAM, with a TOSHIBA N300 4TB SATA hard-disk.

7.3. First Look at Results

For this section we will be showing box and bar plots. Bar plots are basic to understand, they represent quantity; so the bigger the bar is, the bigger their value is. Box plots are more complex, as they represent quartiles for the data set. What's important to note is that the middle horizontal bar represents the median for the data, and the rectangle box represents lower quartiles (the lowest part of the box) and higher quartiles (the highest part of the box) for the data set.

We first show a boxplot containing the distribution of running times for our cache policies, which were discussed further in Chapter 5. An option which answers the queries without using the cache was also included. Figure 7.1 shows the result for this experiment.

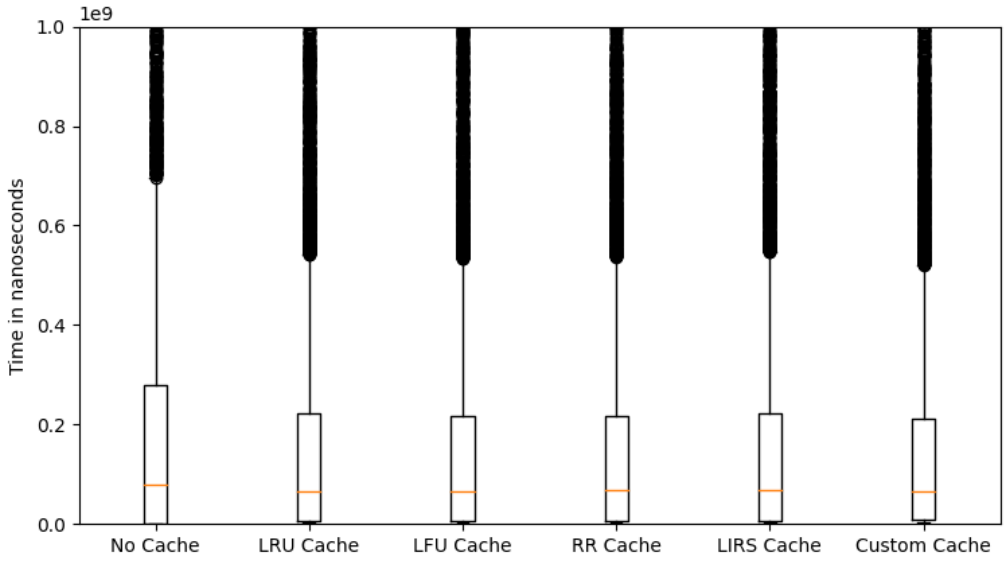

Figure 7.1. Graphic representation of median and quartiles for the runtimes of each caching policy, including no cache

This experiment was done for 10000 queries, applying a 15 second timeout for queries to be run. As we can see, running the experiment without a cache is slower. For the rest there isn't any noticeable difference yet for this small volume of queries. In a zoomed version we can tell some of the medians vary, which we can see in Figure 7.2.

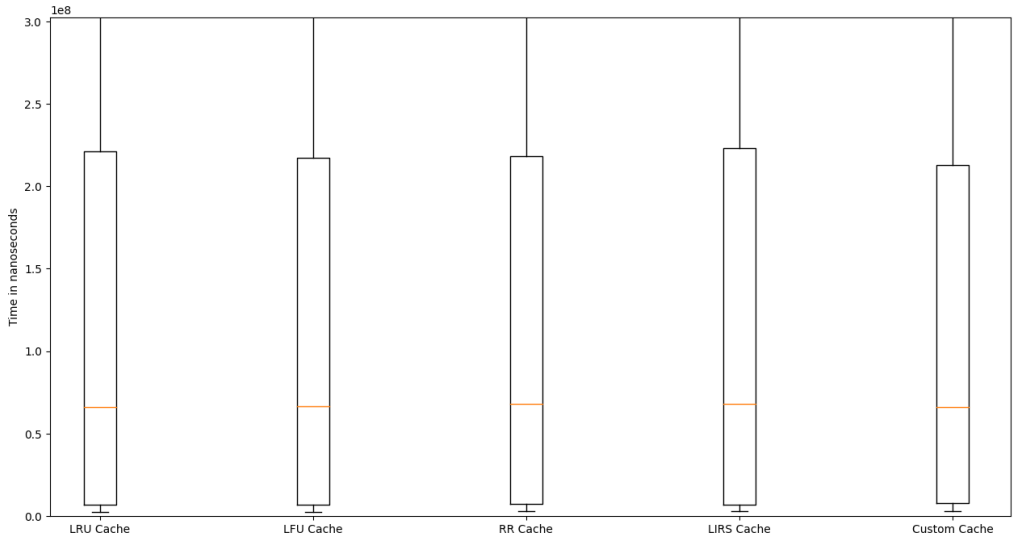

Figure 7.2. Median and quartiles for the runtimes of each caching policy

In Figure 7.2 we can see that the interquartile range for the custom cache is slightly

smaller, which represents that for harder queries to execute, this policy will work better. We will later show in this section how this policy fares against the no cache version of the system for a larger amount of queries.

To justify some of the decisions taken in the process of building our caching system, we provide some additional experiments. Some of these decisions include:

- The size of the buffer that keeps subqueries being 100000.
- Checking only the subsequence of subqueries containing the first triple pattern, the first and the second triple pattern, and so forth.
- Attempting to cache a BGP only after it repeats one time.
- The custom cache only predicting one result.

Buffer Size

To justify the first decision taken we have to see how many subqueries we check to see if they can enter the cache, as well as how much the size of the buffer influences the runtime of the experiment. We tried checking if the buffer should be of size ten thousand or a hundred thousand. In Figure 7.3 we can see how many queries are checked for each of these sizes.

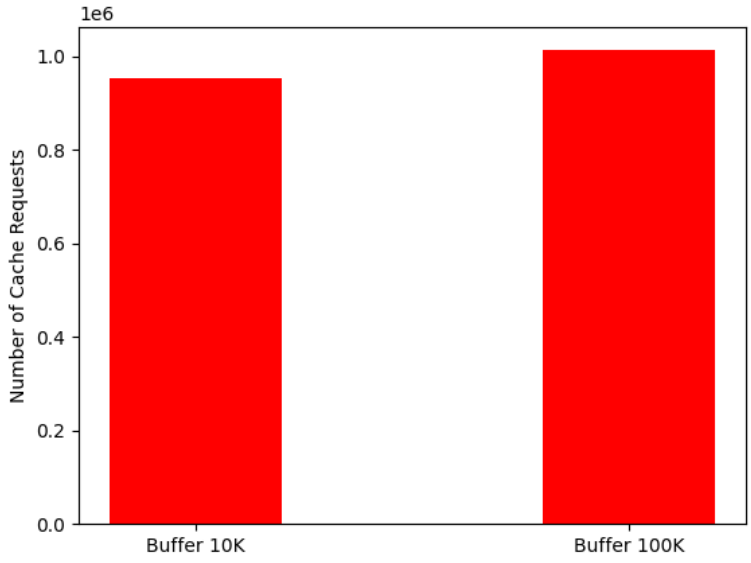

Figure 7.3. Graphic representation for how many cache hits the buffer has depending on its size

For our system, we attempt to send as many requests as possible, while still taking a reasonable time to run experiments. This experiment was run over a log of around 2 million queries. We can see a bigger buffer sends more requests as expected, thus the final decision

was to use a buffer of size 100000. An even bigger buffer would be better in theory, however we decided this number was enough considering we will be running experiments for just a couple of hundreds of thousands queries, explained in a later section.

Subqueries

We measured here the number of cache hits and the time each one takes too, considering extracting all possible subqueries, and only subqueries in optimization order starting from the first triple pattern, this time after running just a thousand queries (given the cost of running the experiment while extracting all possible subqueries).

In Figure 7.4 we see how many cache requests extracting every subquery generates versus how many extracting only the subqueries that follow the optimization order gets with our method. We see that generating and checking all subqueries against the cache generates an approximate 4-fold (3.7 to be exact) increase versus only considering subqueries starting from the first triple pattern after optimization.

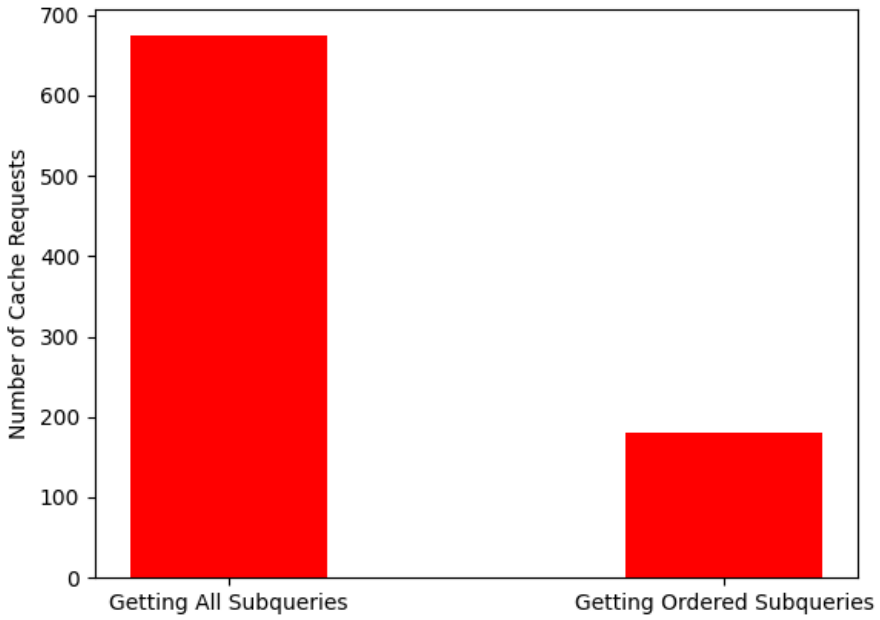

Figure 7.4. Number of cache requests extracting subqueries gets

In Figure 7.5 we see the distribution of runtimes for each configuration. Our custom caching policy was used for this experiment.

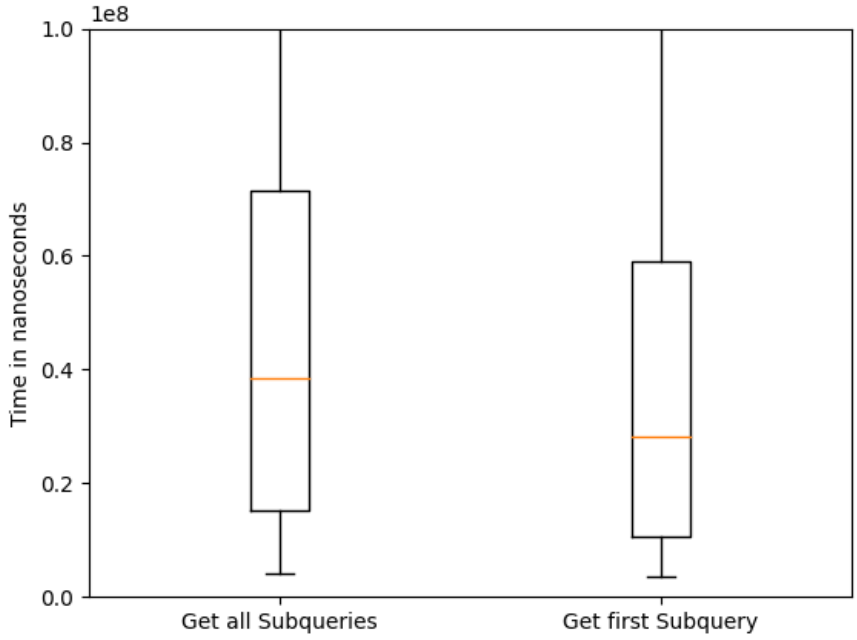

Figure 7.5. Runtimes for extracting subqueries

We see there's a reasonable amount of more cache requests, which would make us consider trying to extract all subqueries for larger experiments. However, for this small experiment of a thousand queries, we already can see there's a considerable time difference between both methods due to the overhead of having to process many more subqueries, and also due to the fact that by inserting cached results for other subqueries, the resulting plan may not follow the optimized join order. To avoid running experiments for too long of a time, we decided to just extract the contiguous subsequence of subqueries which contain the first triple pattern in the optimized join order.

Cache Requests

We predicted we could capture the most amount of BGPs to cache when they repeat once. The core idea behind this is that most subqueries will only appear once, and are thus not worth trying to cache. However, when we see a query twice, we are likely to see it more times. In Figure 7.6 we see how often subqueries repeat when extracting them in optimized order, after running it for around 2 million queries.

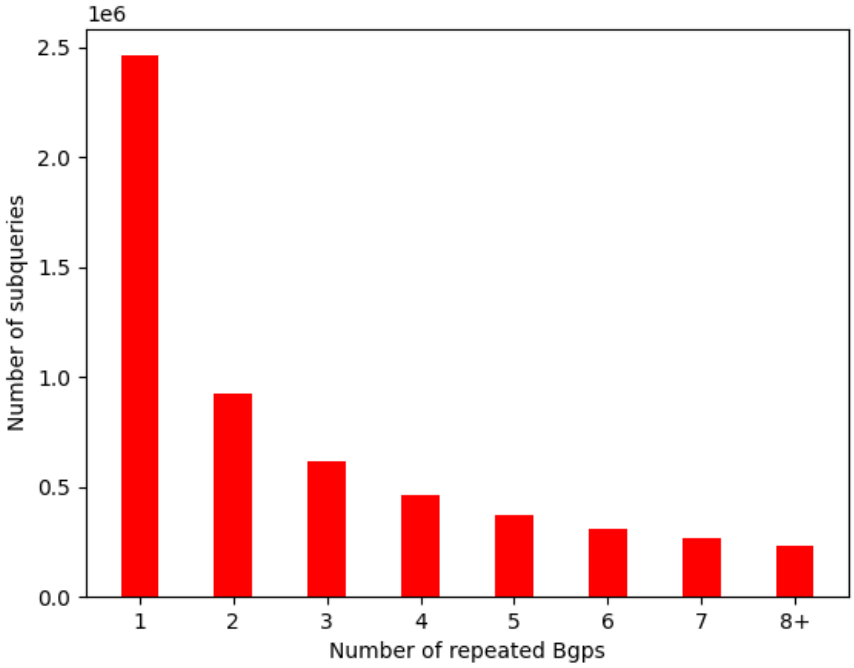

Figure 7.6. Amount of cache hits for each number of repeated BGPs

We see that a little under 2.5 million BGPs only appear once. By requiring that BGPs appear twice before sending them to the cache, we thus save 2.5 million cache requests at the potential cost of 0.9 million cache misses for those BGPs that appear exactly twice. Whenever another BGP repeats that is another cache request, and as the plot shows there are many BGPs which repeat. For example, there are around 0.5 million BGPs which appear three times; this means we get at least 0.5 million more cache requests as we know the BGPs appear at least twice, and we add one more request for each one of these which appear thrice. For the BGPs that appear 4 times, if we use the rule of must appear twice, there's a little bit less of 0.5 million BGPs which appear 4 times, each of which will have up to 2 cache hits.

The exact math that counts how many hits we can potentially get using this rule, assuming the BGP is not replaced in the meantime by the cache policy, is:

$$(n-2) \times f(n)$$

Where n ranges between 3 and ∞ (in this example, we only see up until 8) and $f(n)$ is the Y-axis, that is, the number of subqueries that appear n times. Taking the numbers, the number of potential cache hits we can receive is:

$$1 \times 613,788 + 2 \times 462,177 + 3 \times 370,226 + 4 \times 309,493 + 5 \times 265,560 + 6 \times 232,559 = 6,609,946$$

That means we are getting around 6.6 million potential hits when we consider that a BGP must appear twice before being processed by the cache, which can be compared with

the 0.9 million potential cache hits we lose out on, and the 2.5 million cache requests we avoid having to process.

Custom Cache

In our custom caching policy, we use a metric based on the average cost to produce a single result, prioritizing the caching of subqueries with a higher cost, i.e., with a better potential for future time saving with respect to the cache space used. We proposed to use the time to compute the first result as a heuristic. There is reason to believe a good custom caching policy could measure how much time it takes to run the query associated to the BGP we are about to cache, instead of just predicting how much time it takes to run it based on one result only. This is why we need to justify if this was a good choice by showing if the average runtime per solution is similar to the prediction based on the time to return the first solution. That is to say, we could consider taking the time to compute all solutions and divide that time by the number of solutions to get the average cost to compute each solution. This would provide a more accurate result, but would be potentially (much) more complex to compute.

In Figure 7.7, we show the estimated time to compute a single result using both methods.

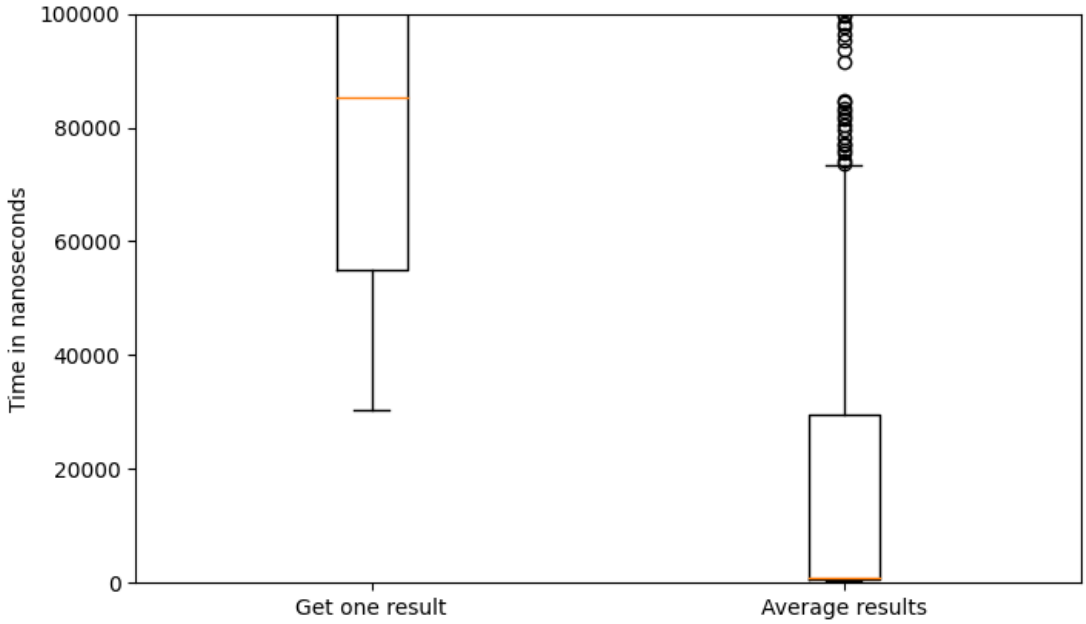

Figure 7.7. Box plot for the time taken to return one result (the first result) versus the average time taken to return each result (the time taken to return all results divided by the number of results)

We can see that, if anything, it would seem that our prediction is inaccurate because it is a larger number on average. However, in terms of prioritizing subqueries to cache, we are more interested in how similar are the ranks induced by the metrics, rather than the absolute

similarity. Along these lines, we measured Kendall's rank correlation coefficient, or τ, whose value is of 0.53 for this case. This value can range from -1 (perfect inverse correlation) to 1 (perfect correlation), with 0 showing no correlation in either direction. When above 0 and closer to 1, this measure means the values we measured are more similar than not. Since our value for τ is closer to 1 than 0, we decided to implement a custom caching policy that predicts just how much time computing the first result takes, instead of measuring the average time per result, since it is more costly to obtain all of the results for a query related to each BGP.

7.3.1 Final Results

For our final experiment we measured for $200,000$ queries how much time it takes to run them without a cache, and with our custom caching policy, taking into consideration all of the adjustments justified previously. Figure 7.8 shows this.

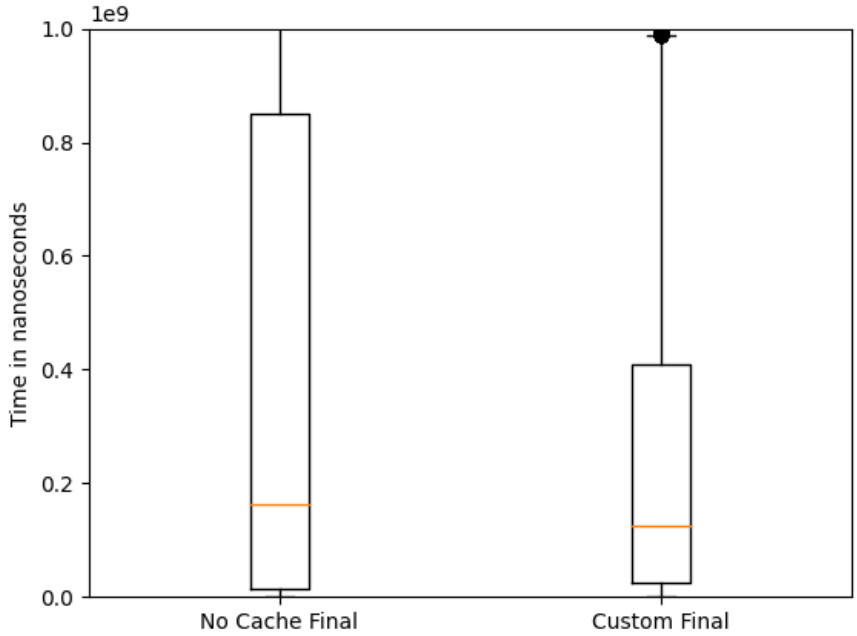

Figure 7.8. Median and quartiles for the runtimes of the experiment without cache versus using the caching system

We can see from this plot that the bottom of the box represents that the no caching experiment is faster for the bottom quartile, probably due to the caching experiment being slower for potentially easier queries to evaluate, but for the other parts of the box the caching portion is faster, indicating the benefits of caching for more expensive queries.

7.3.2 Comparison Against State of the Art

As we discussed in Chapter 2, we believe the best system to compare ours against is Papailiou et al.'s caching system [17]. Their experiments focused on using LUBM [1] to generate artificial datasets and then running queries over these. We tried to replicate these experiments by building and running their code, which can be found at `https://github.com/npapa/h2rdf_caching`. However, we ran into several issues by trying this: the system required a deprecated version of the HBase library, and the code presented a considerable amount of bugs that made it impossible to run in a considerable amount of time. We tried to trim down the code to keep just the caching portion of the system, but since it ran on their own version of H2RDF this was also impossible to achieve. In the end we desisted and narrowed down our goals to compare how different caching policies worked against each other, and if the system was worth using at all as we show in Figure 7.8.

Chapter 8

Conclusion

8.1. Summary

For this work our hypothesis was that a caching system could improve on SPARQL response times by keeping in memory a list of most repeated subqueries with their results which we could access whenever a query had one of these subqueries. Our system was meant to work in a static setting of data, where data would not be edited as time moved on. We also prioritized focusing on running experiments using as close to real world settings as we could get by evaluating queries as they were run in a real setting.

8.2. Conclusions

According to the results shown in Chapter 7, we were able to prove it is possible to create a Caching System which can decrease median and upper quartile response times in a static data environment versus not using a cache. Furthermore, we could test our metrics to determine which subqueries to use were the most profitable. Other adjustments made, such as choosing how our custom caching policy works, how we select our subqueries to be checked for possible cache hits, and the size of the buffer that keeps the subqueries, were also tested to evidence their efficiency.

We introduced a Caching System which can succesfully cache subqueries, and use their results when we find a cache hit. We have also proved we can use Salas and Hogan's query canonicalisation algorithm [19] to get even better results by using canonical labelling for equivalent queries modulo variable names.

However, there is more we can do work on.

8.3. Future Work

Our proposal works for static data environment. However, this work can be improved by experimenting on a dynamic data environment, with data that changes over time, to further prove we can make this work in a real-world setting.

There are also more SPARQL queries we could cover. This work mainly covered `SELECT` type queries, but there are more properties to be worked on.

We propose the parser could be improved to capture a larger number of queries. Better experiments could also be defined to work on closer to real-world settings, potentially taking into consideration other datasets.

In the final results, we see that for the fastest quartile of queries, no caching is more efficient than caching. For future work, it would be interesting to try to define heuristics that can detect these inexpensive cases where the overhead of caching is not worth it.

Bibliography

[1] LUBM queries. http://swat.cse.lehigh.edu/projects/lubm.

[2] RDF: Resource description framework. http://www.w3.org/RDF/.

[3] RDF Schema. http://www.w3.org/TR/rdf-schema/.

[4] SPARQL. http://www.w3.org/TR/rdf-sparql-query/.

[5] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *VLDB J.*, 31(3):1–26, 2022. https://arxiv.org/pdf/2102.13027.pdf.

[6] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, volume 8219 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2013. https://link.springer.com/content/pdf/10.1007/978-3-642-41338-4_18.pdf.

[7] Adrian Bielefeldt, Julius Gonsior, and Markus Krötzsch. Practical linked data access via SPARQL: the case of wikidata. In Tim Berners-Lee, Sarven Capadisli, Stefan Dietze, Aidan Hogan, Krzysztof Janowicz, and Jens Lehmann, editors, *Workshop on Linked Data on the Web co-located with The Web Conference 2018, LDOW@WWW 2018, Lyon, France April 23rd, 2018*, volume 2073 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. https://iccl.inf.tu-dresden.de/w/images/8/85/Wikidata-SPARQL-queries-Bielefeldt-Gonsior-Kroetzsch-LDOW-2018.pdf.

[8] Christian Bizer and Andreas Schultz. The berlin SPARQL benchmark. In Amit P. Sheth, editor, *Semantic Services, Interoperability and Web Applications - Emerging Concepts*, pages 81–103. CRC Press, 2011.

[9] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandecic. Introducing wikidata to the linked data web. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy,*

October 19-23, 2014. Proceedings, Part I, volume 8796 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2014.

[10] Aidan Hogan. Linked data & the semantic web standards. In Andreas Harth, Katja Hose, and Ralf Schenkel, editors, *Linked Data Management*, pages 3–48. Chapman and Hall/CRC, 2014. https://aran.library.nuigalway.ie/bitstream/handle/10379/4386/ldmgmt_local_chapter.pdf?sequence=1.

[11] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In Richard R. Muntz, Margaret Martonosi, and Edmundo de Souza e Silva, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA*, pages 31–42. ACM, 2002. http://web.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-02-6.pdf.

[12] Tomas Lampo, Maria-Esther Vidal, Juan Danilow, and Edna Ruckhaus. To cache or not to cache: The effects of warming cache in complex SPARQL queries. In Robert Meersman, Tharam S. Dillon, Pilar Herrero, Akhil Kumar, Manfred Reichert, Li Qing, Beng Chin Ooi, Ernesto Damiani, Douglas C. Schmidt, Jules White, Manfred Hauswirth, Pascal Hitzler, and Mukesh K. Mohania, editors, *On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part II*, volume 7045 of *Lecture Notes in Computer Science*, pages 716–733. Springer, 2011.

[13] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015. http://svn.aksw.org/papers/2013/SWJ_DBpedia/public.pdf.

[14] Johannes Lorey and Felix Naumann. Caching and prefetching strategies for SPARQL queries. In Philipp Cimiano, Miriam Fernández, Vanessa López, Stefan Schlobach, and Johanna Völker, editors, *The Semantic Web: ESWC 2013 Satellite Events - ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers*, volume 7955 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 2013. https://link.springer.com/content/pdf/10.1007/978-3-642-41242-4_5.pdf.

[15] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, volume 11137 of *Lecture Notes in Computer Science*, pages 376–394. Springer, 2018. https://iccl.inf.tu-dresden.de/w/images/5/5a/Malyshev-et-al-Wikidata-SPARQL-ISWC-2018.pdf.

[16] Michael Martin, Jörg Unbehauen, and Sören Auer. Improving the performance of semantic web applications with SPARQL query caching. In Lora Aroyo, Grigoris

Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II*, volume 6089 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2010. https://link.springer.com/content/pdf/10.1007/978-3-642-13489-0_21.pdf.

[17] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. Graph-aware, workload-adaptive SPARQL query caching. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1777–1792. ACM, 2015. http://www.cslab.ece.ntua.gr/~npapa/sigmod15.pdf.

[18] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. H_2rdf+: an efficient data management system for big RDF graphs. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 909–912. ACM, 2014. https://dl.acm.org/doi/pdf/10.1145/2588555.2594535?casa_token=P1vj3cGFvmIAAAAA:TAJ-WoqaR1bJXCwmHE36xswYeFIrbd0TRxwvVqwpbl1x9B_rQWGJ9cBcZUGQbb_Vng7dAKVlY7xdbA.

[19] Jaime Salas and Aidan Hogan. Canonicalisation of monotone SPARQL queries. In Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I*, volume 11136 of *Lecture Notes in Computer Science*, pages 600–616. Springer, 2018. http://aidanhogan.com/qcan/extended.pdf.

[20] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: the linked SPARQL queries dataset. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, volume 9367 of *Lecture Notes in Computer Science*, pages 261–269. Springer, 2015. https://svn.aksw.org/papers/2015/ISWC_LSQ/public.pdf.

[21] Heiner Stuckenschmidt. Similarity-based query caching. In Henning Christiansen, Mohand-Said Hacid, Troels Andreasen, and Henrik Legind Larsen, editors, *Flexible Query Answering Systems, 6th International Conference, FQAS 2004, Lyon, France, June 24-26, 2004, Proceedings*, volume 3055 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2004. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.6935&rep=rep1&type=pdf.

[22] Gregory Todd Williams and Jesse Weaver. Enabling fine-grained HTTP caching of SPARQL query results. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors,

The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I, volume 7031 of *Lecture Notes in Computer Science*, pages 762–777. Springer, 2011. `http://www.cs.rpi.edu/~weavej3/papers/iswc2011.pdf`.

[23] Gang Wu and Mengdong Yang. Improving SPARQL query performance with algebraic expression tree based caching and entity caching. *J. Zhejiang Univ. Sci. C*, 13(4):281–294, 2012.

[24] Wei Emma Zhang, Quan Z. Sheng, Kerry Taylor, and Yongrui Qin. Identifying and caching hot triples for efficient RDF query processing. In Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema, editors, *Database Systems for Advanced Applications - 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part II*, volume 9050 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2015.

.1. ANNEX

Algorithm 4 Pseudocode for the LIRS algorithm

```
queue S, queue Q, set cache, int l, int h
procedure INIT(l', h')
    l ← l'
    h ← h'
    init S
    init Q
end procedure
procedure LIRS()
    return {x | x ∈ cache and x ∈ S}
end procedure
procedure CYCLE()
    if Q is full then
        x ← dequeue(Q)
        cache.remove(x)
    end if
    y ← dequeue(S)
    prune(S)                ▷ Remove elements from bottom until resident block is found
    Q.enqueue(y)
end procedure
procedure ACCESS(k)
    if k ∉ cache then
        if |S| < l then
            S.enqueue(k)
            cache.add(k)
        else if k ∉ S then
            S.enqueue(k)
            if Q is full then
                x ← dequeue(Q)
                cache.remove(x)
            end if
            Q.enqueue(k)
            cache.add(k)
```

```
        else if k ∈ S then
            S.promote(k)
            CYCLE()
            cache.add(k)
        end if
    else
        if k ∈ S and k ∉ Q then
            S.promote(k)
            prune(S)
        else if k ∈ S and k ∈ Q then
            S.promote(k)
            Q.reomve(k)
            CYCLE()
        else if k ∉ S and k ∈ Q then
            S.enqueue(k)
            Q.promote(k)
        end if
    end if
end procedure
```