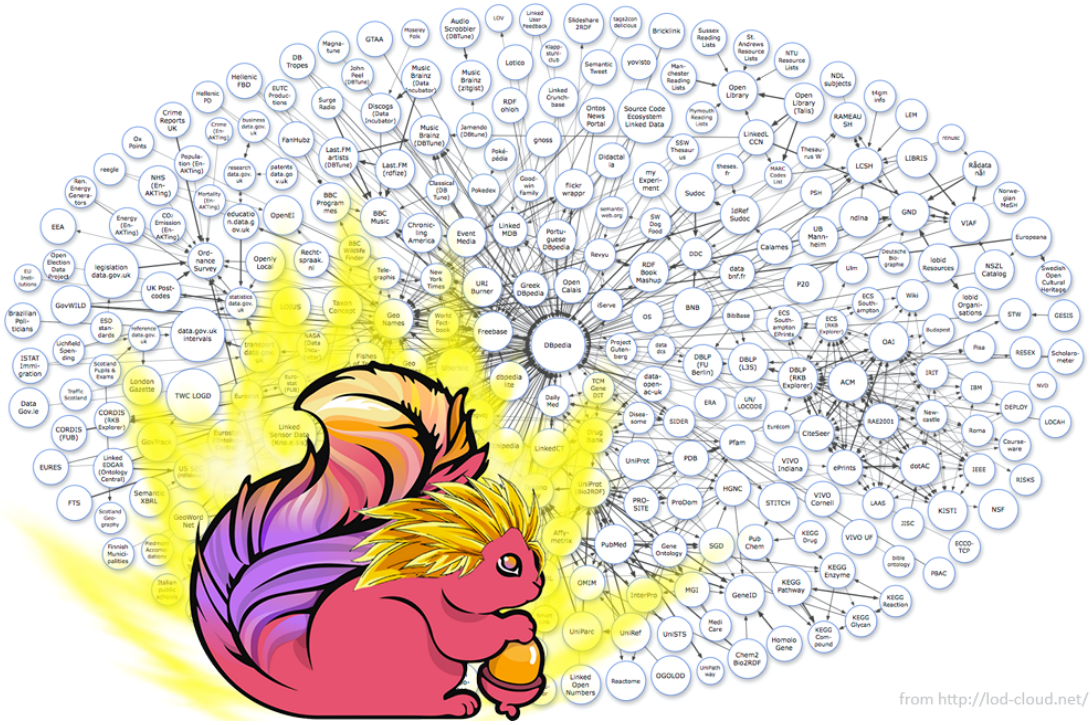
SFRDF+: Join Plans for SPARQL Processing in Apache Flink



Aalborg University
Software – P10 – Spring 2017
DPT106F17

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Software
Selma Lagerlöfsvej 300
9220 Aalborg
www.cs.aau.dk

Abstract:

RDF data is becoming increasingly popular as a model for representing unstructured data on the Web. The data sets therefore reaches web-scale sizes in the form of RDF graphs with billions of triples. In order to handle such large data sets distributed processing systems are needed. SFRDF is one of such systems, which is based on the distributed framework called Apache Flink and the partitioning technique known as ExtVP. SFRDF showed to nearly be competitive with state of the art systems during its creation in the fall of 2016. We propose an improved version of SFRDF, SFRDF+, which implements several improvements to the original system. The improvements include simple changes such as introducing dictionary encoding, but also more advanced features such as introducing join order optimizations in order to generate better query plans. In order to find the best approach for generation of query plans we implement and evaluate different approaches from within the area of RDF processing, i.e. CliqueSquare, and traditional database management systems, i.e. DPCCP and a greedy approach. We evaluate the dictionary encoding and the different approaches for join order optimizations and learn that the only feasible approach for our system is the greedy one. We modify the cost function for the greedy approach to prefer bushy plans to see if these yield better performance. This is not the case for the plans generated by our algorithm, but the standard greedy algorithm shows promising results with an overall reduction to query response times.

Title
SFRDF+: Join Plans for SPARQL Processing in Apache Flink

Theme
Specialization

Project period
P10, Spring semester 2017

Project group
DPT106F17

Jesper Clausen
jclau12@student.aau.dk

Mathias Eriksen Otkjær
meot12@student.aau.dk

Project supervisor(s)
Katja Hose
Stefan Schmid

Number of Pages: 64
Number of Appendix Pages: 11
Published: 2017-06-05

Summary

In this thesis we look at introducing several improvements to an existing RDF and SPARQL processing platform, SFRDF [14], which we developed during our pre-specialization semester. SFRDF is based on the distributed framework Apache Flink, which allows for easily distributing a workload in a cluster environment. The focus of our previous work was to see whether or not it was possible to produce an RDF processing system in Flink, which primarily is a streaming framework, that could compare with other systems implemented in batch-processing frameworks such as Apache Spark. We developed a system, that was close to comparable with another state of the art system, S2RDF [16].

We therefore decided to continue our work with the system to see whether or not we could improve upon it. We do this by implementing several improvements in the system. One of the improvements is introducing dictionary encoding. Dictionary encoding the data set should reduce the size of the data sets significantly, as we, instead of storing strings spanning several characters, can simply store a number referring to that string. In doing this consistently for both query and data set, we are able to execute queries in the system as usual and retrieve results. This is done to reduce the IO in the system as well as the memory footprint, which thereby also should reduce query response times. We test the dictionary encoded approach and learn that the data set does indeed become smaller. However, to our surprise we see, that for smaller data set it actually makes the response times higher, whereas it for larger data sets becomes lower. This indicates that it scales well in regards to the size of the input data. The fact that it is slower for smaller data sets might be attributed to different issues that we encounter throughout the evaluation of the system.

Another improvement that we introduce to the system is join order optimization algorithms. Flink does in itself not change the join order of the provided input query and the joins are therefore executed in a left deep manner as they are presented. Our previous work optimized the join order by utilizing the S2RDF query translator, which used a simple heuristic for using the smallest tables in the first joins. This was therefore a place with room for improvements, and we therefore test different join order optimization algorithms. The first algorithm that we look at is taken from the context of relational databases and is called DPCCP, which is a dynamic programming approach, which efficiently enumerates its subproblems, and based on these finds the best order of joins according to some cost function. We do however quickly come to realize that this approach is not applicable in the context of SPARQL queries as these commonly have a different and more complex structure than standard relational queries.

We therefore seek inspiration in previous research in the area of SPARQL query processing, and utilize an approach called CliqueSquare [5] to generate flat plans. This approach did unfortunately not fit with the Flink framework, as the framework only supports binary joins and the CliqueSquare algorithm generates n-ary ones. We therefore changed our focus towards producing a greedy algorithm with inspiration from [3]. The greedy approach turned out to perform well and we actually saw a noticeable performance gain in our evaluation. The greedy algorithm used a cost function based on [6], which estimates cardinality based on SPARQL query structure. We find that the cost function might not necessarily prefer bushy plans, which we would like as our intuition tells us that this would be better in a distributed context. We therefore modify the cost function such that it prefers bushy plans, but we learn that this variation of the algorithm does not perform better than the original one in most cases.

We evaluate the scalability of the system both in terms of the size of the input data, but also in regards to the number of nodes. We find that our system scales well with data set size, but not so much with number of nodes. We attribute the negative results when scaling the number of nodes to the amount of configuration needed in order to make the framework run efficiently.

In the end of the thesis we conclude on our findings. We come to the understanding that a greedy algorithm can be very effective in the context of RDF and SPARQL processing. We conclude that our system performs well even when processing up to 400 million triples, but that using the Flink framework might entice some issues. These issues include Flink being a JVM based framework with unpredictable garbage collections that makes the query response times fluctuate. Additionally, Flink requires a lot of fine-tuning when running on different amount of nodes and different data set sizes, which makes optimal performance hard to achieve. These issue are the subject of some of the future work that we present in the end, along with several other features and changes that we think may improve our new system, SFRDF+.

Preface

This report is written by two master Software Engineering students from Aalborg University and acts as their master project. The report is written during the spring semester of 2017 between February and June. The report is based on the project we worked on in our pre-specialization semester, namely SFRDF [14]. The purpose of this report is to introduce potential improvements to this system and evaluate the outcome of it.

We would like to thank our supervisors, Katja Hose and Stefan Schmid, for their cooperation during our pre-specialization and specialization semesters, as well as their interest in our projects and eagerness to help us. Furthermore, the S2RDF system [16] has been a cornerstone of our projects during both semesters, so we would like to extend our thanks to the authors behind it, as their system has been a great inspiration.

Contents

1 | Introduction

Efficient processing of large RDF data sets is becoming increasingly relevant in recent years, as it provides a general data model for representing unstructured data, which in turn is becoming increasingly common in the big data communities. The information that can be extracted from all this data is usable in many contexts, and it is therefore relevant to be able to process RDF and SPARQL queries in an efficient manner. Techniques such as Web scraping are responsible for increasing amounts of data, and multiple data sets, e.g. data sets from Data.gov[1], YAGO [17] and more, have already now exceeded sizes beyond one billion triples.

Centralized approaches have been suggested to handle the task of processing SPARQL queries on RDF data sets, however, the size of the largest data sets do not allow for a centralized approach as they are limited by their hardware. Instead more recent research have been focused on processing the data in a distributed manner, which also has shown promising results. S2RDF [16] has for instance used a novel approach for partitioning the data called Extended Vertical Partitioning (ExtVP), and systems such as CliqueSquare [5] have focused on generating flatter query plans in order to reduce the overhead of using the Hadoop MapReduce framework. Another example is our previous work [14], where we propose a solution in the context of distributed SPARQL processing, namely SFRDF (SPARQL on Flink for RDF). SFRDF did however show to perform worse in comparison with the similar system, S2RDF [16], on which it was based, and we have therefore chosen to introduce improvements to SFRDF, in a new system called SFRDF+.

The first improvement that we introduce to SFRDF+ is dictionary encoding, which is done in many other RDF processing systems [1, 12, 15, 19]. The encoding should make the size of the data set significantly smaller, and thereby reduce the storage overhead introduced by using ExtVP, and at the same time reduce time spent on IO and amount of RAM needed for handling large joins. The second improvement that we introduce to the system is join order optimizations. The Flink framework itself does not provide any form of optimizations in this area and simply performs joins "as is" in the query, which usually results in a left deep execution. We therefore attempt to modify the Flink source code with additions of different join order optimization algorithms inspired from both relational database research, i.e. DPCCP [10] and greedy join ordering, as well as algorithms from previous research in the area of SPARQL query processing, i.e. CliqueSquare [5]. Our primary goal in this regard is to introduce bushy plans to the systems in order to better utilize the distributed nature of Flink.

The report will firstly cover some of the basics for this project, namely what RDF and SPARQL is. This is accompanied by a description of our previous work, SFRDF [14], along with a description of the general theory on which we base our improvements upon. All of this is introduced in Chapter 2. Chapter 3 briefly covers how we implement dictionary encoding in our system, but also contains descriptions of the different join optimization algorithms we have utilized in order to achieve lower query response times. This is followed by an evaluation in Chapter 4, where we evaluate how the different optimizations have affected the performance. We conclude on our findings in Chapter 5, and lastly in Chapter 6 we describe some of the open problems we would have liked to work with given more time.

[1] https://data-gov.tw.rpi.edu/wiki/data set_91

2 | Background and Preliminaries

This chapter will describe some of the basics for our project. It will firstly cover what RDF and SPARQL is, and then how we in our previous work implemented SFRDF, a system for processing SPARQL queries on large quantities of RDF data. This is followed by a theoretic overview of the areas we find interesting for further improving the SFRDF system.

2.1 RDF and SPARQL

We will in this section only describe aspects of Resource Description Framework (RDF) and SPARQL Protocol and RDF Query language (SPARQL) that are important to the understanding of the remainder of the report. We invite the reader to read our previous work [14] for an in-depth description of RDF and SPARQL.

2.1.1 Resource Description Framework

RDF is a standard for representing data on the Web recommended by the World Wide Web Consortium. RDF is a very general model, that can be used to represent a wide variety of data and connections between entities in data. RDF is easily interpreted by machines, which normally would have difficulties interpreting links between entities in heterogeneous data sets. An RDF data set, also called an RDF graph, is a collection of triples, consisting of a subject-, predicate-, and an object-part. A single triple describes an entity or the relation between two entities, i.e. the subject has some relation (the predicate) to an object. All of the parts of a triple can be represented by an International Resource Identifier (IRI), and the subject and object part can also be literals; e.g. string, numbers, and dates; or blank nodes, i.e. an entity that does not need an unique identifier. By utilizing IRI's we have an unique identifier for a resource, and can link between different resources by using it. An example of some RDF data in n-triples format is shown in Code Snippet 2.1.

```
1  http://example.com/jesper    http://www.w3.org/1999/02/22-rdf-syntax-ns#type    http://xmlns.com/foaf/0.1/Person
2  http://example.com/jesper    http://xmlns.com/foaf/0.1/name                     "Jesper"
3  http://example.com/jesper    http://xmlns.com/foaf/0.1/age                      24
4  http://example.com/mathias   http://www.w3.org/1999/02/22-rdf-syntax-ns#type    http://xmlns.com/foaf/0.1/Person
5  http://example.com/mathias   http://xmlns.com/foaf/0.1/name                     "Mathias"
6  http://example.com/mathias   http://xmlns.com/foaf/0.1/age                      24
7  http://example.com/rasmus    http://www.w3.org/1999/02/22-rdf-syntax-ns#type    http://xmlns.com/foaf/0.1/Person
8  http://example.com/rasmus    http://xmlns.com/foaf/0.1/name                     "Rasmus"
9  http://example.com/rasmus    http://xmlns.com/foaf/0.1/age                      24
10 http://example.com/rasmus    http://example.com/follows                         http://example.com/jesper
11 http://example.com/jesper    http://example.com/follows                         http://example.com/rasmus
12 http://example.com/mathias   http://example.com/follows                         https://example.com/rasmus
```

Code Snippet 2.1: An example of some RDF data in n-triples format.

2.1.2 SPARQL

We are not only interested in representing data in RDF, but also interested in querying it. For this purpose W3C recommends utilizing SPARQL, which is a query language specifically targeted towards RDF. The main component of a SPARQL query is the triple patterns, which can be seen as a triple, `<subject, predicate, object>`. In a triple pattern any of the three parts can be replaced by a variable. We can in this way define patterns to match against the triples in the RDF graph. Multiple triple

patterns form a basic graph pattern, which enables us to specify joins by utilizing the same variable in multiple triple patterns. Thereby, we have the two main concerns when processing SPARQL queries, namely data access, i.e. matching the patterns and returning the matching triples, and joins, i.e. joining the triples retrieved by data access in order to obtain the result. An example of a SPARQL query is shown in Code Snippet 2.2. The query returns `?a` and `?b` for which it holds that `?a` is a person, `?b` is called "Peter", and `?a` and `?b` follow each other. In the query we can see the basic graph pattern between the curly braces, which consists of multiple triple patterns that each end with a period. There are multiple joins present in the query one of which is the join between the first two triple patterns, which both use the variable `?a` in the subject position, hence the triples found by these patterns should be joined where the subjects are the same.

```
SELECT ?a, ?b WHERE {
   ?a rdf:type foaf:Person .
   ?a example:follows ?b .
   ?b example:follows ?a .
   ?b foaf:name "Peter" .
}
```

Code Snippet 2.2: Example of a SPARQL query.

2.2 SFRDF

This report is based on our previous work [14], where we reimplemented an existing experimental SPARQL processing approach S2RDF [16] in Flink instead of Spark. For an in-depth overview of SFRDF and S2RDF we invite the reader to refer to our previous work [14] and the S2RDF article [16]. We reimplemented S2RDF as it was a state of the art approach in relational RDF processing, and we wanted to test whether or not Apache Flink could outperform Spark in this context. The SFRDF system proved to have worse but comparable performance to S2RDF, and we therefore decided to continue our work on the system.

SFRDF is as mentioned based on Apache Flink, which is big data processing framework that specializes in stream processing, unlike Apache Spark which mainly does batch processing. This might be seen as a disadvantage for our purposes, but Flink also has great batch processing performance, and offers features that are not available in Spark, such as operator chaining, which allows for records to be processed in a pipeline.

The original SFRDF system was, as mentioned above, based on S2RDF, which implemented a novel partitioning scheme of the data called Extended Vertical Partitioning (ExtVP). ExtVP is based on Vertical Partitioning (VP) [1], which is a scheme where the RDF data is divided into multiple partitions based on the predicate of a given triple. All predicates with the same predicate, e.g. `foaf:friendof`, will be stored in the same partition/file. This increases the granularity of data access when working with triple patterns with constant predicates, because we can scan only the triples relevant for the given predicate instead of scanning all triples in the entirety of an RDF graph.

Using VP in itself is great for achieving faster data access, as we reduce the amount of triples that we have to go through, however, it does not affect join speed. For this purpose the authors of [16] propose ExtVP. ExtVP takes the preprocessing of the RDF a step further, and pre-computes a series of semi-joins, the results of which can be

used instead of the standard VP partitions. We can then, given a basic graph pattern, use the best possible table when doing data access, and thereby reduce the size of the intermediate results in the joins, ultimately leading to lower response times.

Given the basic graph pattern in the query shown in Code Snippet 2.2, we can see that the first triple pattern can be joined with the second or the third triple pattern as they share the `?a` variable. This means that there are three different candidates for the data access for the first triple pattern, namely the VP table, `rdf:type_VP`; the subject-subject semi-join table between `rdf:type` and `foaf:follows`, `rdf:type_foaf:follows_SS`; and the subject-object semi-join table between `rdf:type` and `foaf:follows`, `rdf:type_foaf:follows_SO`. Amongst these, the table with the lowest *selectivity factor* is selected, i.e. the table that reduces the size of the original table (`rdf:type_VP`) the most is used. In doing this S2RDF, and by extension SFRDF, significantly reduces the size of the intermediate join results.

2.3 Dictionary Encoding

When operating on, or comparing, large strings, an easy way to achieve a performance increase is to shorten some part the string, which is a standard technique in database systems. This was also the case for the work done by for example S2RDF [16], where the predicate for an RDF IRI will be shortened to another format. In the case of S2RDF, a predicate such as `http://www.w3.org/2000/01/rdf-schema#` will be shortened to `rdfs`. In this case, it reduces a 37 character string to a 4 character string, which both means that you need less memory and disk to store the data, but it also requires less checks to compare this string with something else. We can take this concept further by, instead of encoding the string with a smaller string, just replacing the entire string with an integer, which has already been done in previous work [1, 12, 15, 19]. Integers will, in many programming languages, be implemented with a 32 bit data structure, which means that integers can be used to represent up to 4.294.967.295 different strings. Individual characters used to build string objects are typically implemented using an 8 bit data structure. This means that as long as a string has more than 4 characters, we can represent it more efficiently by using integers instead of strings.

For an example of how efficient this is compared to the full strings, consider the following example: The largest RDF data set is around 1 billion triples with 3 values per triple. The average prefix length for the IRIs that we handle with SFRDF is 30 characters, and it is not unreasonable to consider the IRI value after the prefix to have a length of 15 characters. This means we have:

$$45 \text{ characters} \cdot 8 \text{ bit} \cdot 1.000.000.000 \text{ rows} \cdot 3 \text{ values/row} = 135\text{GB}.$$

If we instead use integers, we can reduce this to:

$$32 \text{ bit} \cdot 1.000.000.000 \text{ rows} \cdot 3 \text{ values/row} = 12\text{GB}.$$

That is more than an order of magnitude less memory / disk space when working on the rows themselves, and that is by considering every character to be 8 bits, which is not always the case when working with Internationalised Resource Identifiers (IRI), that are unicode encoded.

In order to achieve this integer encoding, it is a necessity that all occurrences of the same value have the same number, otherwise it will become impossible to join values together in any meaningful way. Therefore, some kind of dictionary must be kept where

it is possible to look up a value and receive the correct number for your value. The most efficient method for implementing such collections is likely to be a `HashMap` of some kind, in order to achieve constant lookup time, as a regular list would probably take too long to iterate through as the data size increases. Furthermore, due to the integer values having to be converted back to the string values at some point, it is likely necessary to keep another `Map` implementation that contains the reverse mapping. This adds some overhead to the integer encoding approach, but since there are many repetitions of IRIs in the list of RDF triples, it will not take the same amount of space as the full triples list. The speed of the approach is furthermore limited by the size of the result set, since a larger data size simply takes longer to iterate through and translate back to "useful" values in the end. This is mitigated by the previously mentioned constant lookup speed, so the impact will be relatively insignificant. In Chapter 3 we discuss the implementation of the dictionary encoding, and in Chapter 4 we discuss several tests related to it, such as how much time is spent on creating the encoding and how much disk space is saved.

2.4 Query Optimization

When utilizing Flink as the framework for our application we have access to the rule based optimizer of Flink. This allows for several different optimization rules, such as selection and filter pushdown. However, one thing that Flink does not consider in its optimization of queries is join ordering. Instead the framework executes joins in the order of which they are entered in the SQL input. In our previous work [14] we utilized the Query Translator from S2RDF [16] to translate SPARQL queries into SQL, which in turn could be executed by Flink. That of course entailed that S2RDF would decide the join order, as we had no direct influence on the result of the translation. The order of which the join are executed in have a big influence on the size of the intermediate results, and therefore join order optimizations is a logical next step for improvements to SFRDF.

There are two typical join orders to consider here, namely the (left) linear join tree and the bushy join tree. The linear tree has a structure as can be seen in Figure 2.1, where the process starts by joining the two tables at the bottom, indicated with letters A and B, and then the intermediate result of this join is used to join with the next table. Using this method, the operations are linear and you always execute one join at a time.

The linear join tree can utilize pipelining, which means that individual rows from one join can be sent to the next join as soon as it has been computed. This means that hashtables can be created for all tables in the join tree, and thereby speed up processing quite a bit. The main disadvantage is that the joins are calculated sequentially, and that it is not possible to calculate multiple joins in parallel.

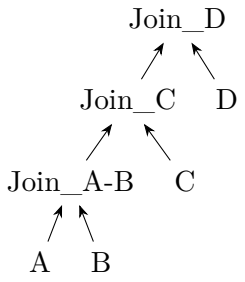


Figure 2.1: Representation of a left-deep join tree

The other type of join tree, the bushy tree, has a structure as seen in Figure 2.2. The difference, when using this type of plan, is that parts of the plan can be executed in parallel. When two data sets are joined together, it is typical to utilize some kinds of join conditions, i.e. specific columns, in order to avoid making expensive cartesian joins, where all rows from one table are joined with all rows in the other table. When multiple data sets are joined together on the same column, it makes sense to keep these joins "close together" in the execution plan. By doing this, it is possible to split parts of the plan that use different join criteria into separate subtrees. This way all subtrees can be computed in parallel, because none of the data sets have to be joined with any of the other sub-trees, until the parent node has been reached where the tree was originally split up.

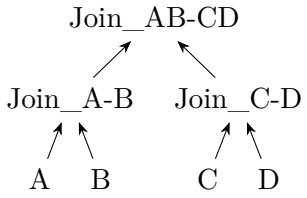


Figure 2.2: Representation of a bushy join tree

There are multiple ways to create join execution plans, but the main difficulty with all these methods is, that the problem is NP-hard once you want to figure out which of the plans is the optimal one [11] in terms of response times. It is therefore common to settle for approximation by using heuristics or probabilistic optimizations. The authors of [11] furthermore concludes that it might be easier to construct a good bushy plan than a left deep plan, while also claiming that there exist several indications that using greedy or dynamic programming approaches can efficiently find optimal or close-to-optimal bushy join orders. It is important to remember that making a good execution plan is a cost/benefit problem, where the overall time from the query entering the system to the result being fetched might actually increase if too much time is spent on making a good plan that does not provide the same reduction to response times as it spends on finding the query plan.

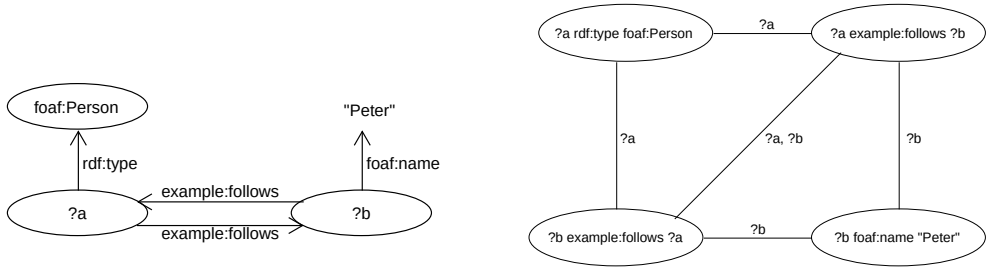
The join ordering problem has many existing solutions when working with standard SQL, such as DPCCP[10] that efficiently enumerates all possible sub-trees and then build up an optimal join tree using a dynamic programming approach. When working with RDF data and SPARQL, the problem is a bit harder, because there are typically a lot more joins present in a query than in SQL. This means that brute-forcing all possible joins can be a costly affair. Instead, other approaches have been made such as CliqueSquare[5], which creates cliques of triple patterns that share a variable in order to create sub-trees in a bushy execution plan. Another system which achieves rather good performance using a left-deep execution plan is S2RDF[16], which simply uses precomputed intermediate join results with low *selectivity factor* for the joins, which S2RDF orders by using the tables with the highest amount of constant fields first, and then by order of the smallest table. Based on the nature of how joins work, starting with the smallest data sets first, keeps the needed memory and computation power as low as possible for as long as possible. By first using the triple patterns with the most constant fields, you significantly reduce the input size, as these constants have potential to greatly limit the amount of rows by filtering the input data.

For our purposes we have decided to try implementing a bushy join order. This choice is based on several factors. First of, we are utilizing a framework designed for massively parallel distributed computing, making the bushy approach more beneficial. We have furthermore observed that query execution times do not increase much, even though we increase the data set size significantly. This might indicate that there is a lot of overhead in setting up tasks, such as joins. By using a bushy strategy this overhead might become less apparent as we can setup multiple tasks at once and thereby allow the overhead to overlap, ultimately reducing the response time of queries. We do not at this point know whether or not it will make a difference in performance, but we think that there is grounds for this approach being better than the previously used linear approach.

2.5 Query Graphs

Many query optimization algorithms, including the ones that we will implement in our efforts to produce a bushy query plan, take a *query graph*, also known as a *join graph*, as their primary input. Therefore, we will in this section quickly outline what a query graph is and how it differs from a *SPARQL graph*.

A query/join graph does, as the name implies, describe the structure of a query just as a SPARQL graph does. However, in the case of a query graph, each node in the graph represents an entire triple pattern, and the edges between the nodes represent joins between these, i.e. the variables shared between the two patterns. In contrast, a SPARQL graph will, as previously mentioned, have each vertex represent either the subject or the object from a triple pattern, while the edges represent the predicates. An example of a SPARQL graph based on the query in Code Snippet 2.2 and the corresponding query graph is shown in Figure 2.3a and Figure 2.3b respectively.



(a) SPARQL graph based on the query in Code Snippet 2.2.

(b) Query graph based on the query in Code Snippet 2.2.

Figure 2.3: A SPARQL and a query graph of the query in Code Snippet 2.2.

Figure 2.3b clearly shows how the joins of the query are structured, which is needed for the join order optimization algorithms. Each edge in the graph is labeled with the join condition between the two relations. However, as we are working with SPARQL, the only possible join condition is an equi join. We can therefore omit the condition and just express which variable the join is based on.

3 | Design and Implementation

In this chapter we cover the implementation of some of the improvements that we have introduced to SFRDF [14], which include reducing the memory footprint and general size overhead by implementing dictionary encoding, as well as utilizing the parallelism in the system better by using bushy plans in the system.

3.1 Dictionary Encoding

As mentioned in Section 2.3 dictionary encoding can reduce the storage overhead, memory footprint, and increase efficiency of joins. It is thereby attractive to try out in our system. The concept of changing to using a dictionary is rather simple:

1. Create the dictionary
2. Translate everything to numbers
3. Compute the partitions and execute queries as we did before
4. Translate results back

While this sounds like a trivial task, there were many complications that we had to deal with during this change.

Firstly, we need to generate the dictionary. Initially we did this by loading all triples into memory, iterating over them, and inserting entries into a `HashMap` along with an incrementing counter if they were not present in the map already. This approach is not very scalable as we need to load all triples into main memory on the `JobManager` in order for it to work, and we therefore encountered obstacles when scaling the data sets beyond 100 thousand triples.

Instead we decided to utilize native Flink functionality, `data setUtils.zipWithIndex`, which enumerates the input it receives in a distributed manner. If it receives a data set containing strings, `data set<String>`, it will return a data set with a tuple containing a integer and the string, `data set<Tuple2<Long, String»`. We use this method to generate our dictionary by passing a `data set<String>` of all distinct string entries as an argument. The data set of distinct strings is generated by utilizing a `flatMap` function on the original triples data set in order to emit all individual entries, i.e. the subject, predicate, and object, of each triple. We then call the `distinct` transformation on the result of the `flatMap` in order to generate a `data set<String>` with all the distinct string entries in the RDF graph.

When using the first way of generating the dictionary, we could of course directly translate the triples when generating the dictionary. This was less desirable with the second approach as we again would need to load all triples into main memory on the `JobManager` in order to do so. Instead, we decided translate the triples in a distributed manner as well. However, in order to make distributed translation, all of the `TaskManagers` would need to have access to the dictionary. There exist two approaches to distribute variables in Flink, namely using variable closures or using broadcast variables. Both of the approaches proved to be ineffective when the size of input data became large enough, hence we needed to find an alternative solution. Therefore, we

looked at the possibility for using joins for this purpose, and join the triple table with the dictionary data set, where the string entry of the dictionary is equal to either the subject, predicate, or object of the triples. In this way we could in the end utilize a projection operator to only select the numeric entries corresponding to the subject, predicate, and object respectively and in this way translate the triples table.

A key concern when working with dictionary encoding is that everything has to be persistent. If the dictionary is lost, the entire data set is useless, because nobody will know what the individual numbers mean. As our preprocessing time is pretty extensive, it would be beneficial not having to generate the data set each time we want to run queries. Flink offers native functionality for outputting data sets, such as the dictionary, to the file system. However, if no precaution is taken Flink will print as many files as it has task slots. This is not an issue if the file access methods from Flink are used, but it is bothersome when working in standard Java. This brings up the question as to how we should translate the query results back into strings, as we identify two ways for doing this. One way is to translate the results back with joins as we did with the initial translation in a distributed fasion, and the other method is to load the dictionary into memory on the `JobManager`, collect the results here, and iterate through them one by one.

By setting the parallelism of an output operation in Flink to one when writing the dictionary to disk, the issue with the many files can be avoided, as it then will only have one task slot available and only one file is produced. It would then be easy to read the file on the `Jobmanager` and perform the translation. It would most likely be more efficient to do for small result sets, as executen it in memory of the `JobManager` should outperform the overhead in regards to splitting up the data, sending it to the taskmanagers and performing a join for every single value in the result. However, translation on the `JobManager` might prove to not be effective when the result size becomes too large, which therefore can result in having to switch to translating the results in a distributed fashion instead. Even though it might cause us to change eventually, we have decided to start out with the local execution, and if any issues arise with the translation, we will have to do change the method.

3.2 Plan Generation

In this section we will cover the different approaches that we have worked with in order to generate bushy query plans. The first approach is DPCCP [10], which selects the optimal plan according to a cost function; the second one is CliqueSquare [5], which utilizes clique decomposition to find flat bushy plans; and the final one is a greedy approach, which is much faster in generating a plan compared to the other approaches.

3.2.1 DPCCP

The DPCCP (Dynamic Programming Connected sub-graph Complement Pairs) algorithm [10] is, as the name implies, a dynamic programming approach for generating an optimal query plan for regular relational queries. As with other dynamic programming approaches it is exhaustive as it explores the entire search space. However, unlike previous approaches DPCCP only considers the bare minimum of sub-problems, making its search space the same size as the theoretical lower bound of dynamic programming approaches for bushy query plan generation. We implemented this algorithm, in order to validate whether it could work for a SPARQL based system as well.

The algorithm works by enumerating all possible pairs of connected sub-graphs (CSG) and their complements, i.e. connected sub-graph complement pairs (csg-cmp-pairs). In order to talk about the algorithm and the enumeration of these pairs, we firstly need to cover some definitions.

Definition 3.1:
Let $G = (V, E)$ be a query graph, where $V = \{v_0, ..., v_{n-1}\}$. A **connected subset/sub-graph**, S, is a subset of V, $S \subseteq V$, such that S is connected, meaning that all the vertices of S should be reachable by all other vertices in S.

Definition 3.2:
Let $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ be two non-empty connected sub-graphs. We call them a **csg-cmp-pair**, iff:

1. $V_1 \cap V_2 = \emptyset$
2. There exists $v_1 \in V_1$ and $v_2 \in V_2$, which has an edge between them

Definition 3.3:
Let $G = (V,E)$ be a connected query graph. For a node, $v \in V$, we define the **neighborhood** of v, $\mathcal{N}(v)$, to be $\{v' | (v,v') \in E\}$. For a subset, $S \subseteq V$ we define the neighborhood of S, $\mathcal{N}(S)$, to be $\bigcup_{v \in S} \mathcal{N}(v) \backslash S$.

Note that due to join being a commutative action if (S_1, S_2) is a csg-cmp-pair then (S_2, S_1) must be as well. If all of the csg-cmp-pairs are enumerated we can iterate over them as sub-problems and build an optimal bushy query plan in a bottom-up fashion by combining them to larger plans as shown in Algorithm 1. The algorithm is very simple and simply builds up a map, *BestPlan*, that maps from a set of relations to a query plans. This makes the order in which the csg-cmp-pairs are enumerated important. The pairs containing the smallest CSGs need to be processed by the algorithm first, such that these are present, when processing pairs consisting of supersets of these small CSGs. By building up the map in this fashion we will in the end have a map containing an entry with the key being set of all relations, and the value being the optimal bushy query plan for joining all of these.

Algorithm 1 The DPCCP algorithm from [10].

Input: A connected query graph $G = (V, E)$, where $V = \{R_0,...,R_{n-1}\}$.
Output: An optimal bushy join tree/plan
for all $R_i \in V$ **do**
    BestPlan($\{R_i\}$) = R_i
end for
for all csg-cmp-pairs, (S_1, S_2), **do**
    $S = S_1 \cup S_2$
    p_1 = BestPlan(S_1)
    p_2 = BestPlan(S_2)
    CurrentPlan = CreateJoinTree(p_1, p_2)
    if cost(BestPlan(S)) > cost(CurrentPlan) **then**
        BestPlan(S) = CurrentPlan
    end if
    CurrentPlan = CreateJoinTree(p_2, p_1)
    if cost(BestPlan(S)) > cost(CurrentPlan) **then**
        BestPlan(S) = CurrentPlan
    end if
end for
return BestPlan($\{R_0,...,R_{n-1}\}$).

The algorithm in itself is as seen a pretty simple dynamic programming algorithm. The more advanced part and the core of the DPCCP algorithm is therefore to enumerate all the pairs in an efficient manner. This is done in two stages, namely generating connected-sub-graphs and then generating all the possible complements for each of these. The key concern in the generation is to avoid producing duplicates. Therefore, the algorithm seen in Algorithm 2 has the requirement that the input query graph has its vertices enumerated by a breadth first search. We can in this way avoid producing duplicates when generating CSGs for a given vertex, v_i, by adhering to the constraint that the generated sub-graphs cannot contain any vertex, v_j, where $j < i$. We therefore define $\mathcal{B}_i = \{v_j | j \leq i\}$ in the algorithm, which is a set of vertices that is excluded in the recursive generation of subsets.

Algorithm 2 EnumerateCsg [10].

Input: A connected query graph $G = (V, E)$, where $V = \{v_0,...,v_{n-1}\}$.
Precondition: Nodes in V are numbered according to a breadth-first search.
Output: All subsets of V that induces a CSG of G
for $i = n - 1$ down to 0 **do**
    emit($\{v_i\}$)
    EnumerateCsgRec($G, \{v_i\}, \mathcal{B}_i$);
end for

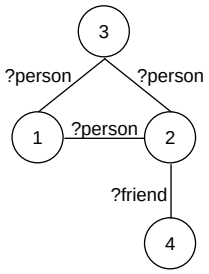
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$
for all $S' \subseteq N, S' \neq \emptyset$, enumerate subsets first **do**
    emit($S \cup S'$)
end for
for all $S' \subseteq N, S' \neq \emptyset$, enumerate subsets first **do**
    EnumerateCsgRec($G, (S \cup S'), (X \cup N)$)
end for

The enumeration of CSGs works by going through all vertices in V and emitting it followed by a call to the recursive part of the enumeration. The recursive function takes three parameters. The graph itself, G, which is required in order to find the neighborhoods; the seed set, S, which initially consists of the one node; and lastly the set of vertices to be excluded, X, which initially is equal to $\mathcal{B}_i$. In the recursive part the first thing that is done is to calculate the neighborhood of the seed set, and subtract the exclude set, X. All the non-empty subsets of the result of this operation are then iterated over in order of ascending size and the union of the seed set and the given subset is emitted to produce another CSG. This is followed by a recursive call for each of these newly generated CSGs to find expansions of them.

An example of the generation of subgraphs is shown in Table 3.1. The example is based on the query and its corresponding query graph in Figure 3.1, where we see a fairly simple query requesting to find persons aged 24 with the name "Jesper Clausen" and the names of their friends. Table 3.1 illustrates the calls made by the algorithm and recursive calls with indentation. Each of the iterations in the outermost call is separated by a horizontal line in order to make a clear division. In the case of $\{v_4\}$ and $\{v_3\}$ we see that they have no nodes in their neighborhood that are not in the exclude set. However, if we look at $\{v_2\}$ it actually has v_3 and v_4 as neighbors not in the exclude set. We can therefore emit all non-empty subsets, $\{v_3\}$, $\{v_4\}$, and $\{v_3, v_4\}$, combined with the original set, $\{v_2\}$, resulting in the emits shown in the right most column. After these have been emitted we can for each of the combined results call the enumeration process recursively, where all the recursive calls end up with no further emits as the exclude set now includes all nodes in V. The process is then continued with $\{v_1\}$, the calls of which you can see in the bottommost row of Table 3.1.

```
SELECT ?person, ?friendName WHERE {
  ?person foaf:name "Jesper Clausen" .// 1
  ?person foaf:knows ?friend .        // 2
  ?person foaf:age 24 .               // 3
  ?friend foaf:name ?friendName .     // 4
}
```

(a) SPARQL query.



(b) Query graph.

Figure 3.1: A simple SPARQL query and the corresponding query graph enumerated breadth first.

Calls	**Emitted**
$emit(\{v_4\})$	$\{v_4\}$
EnumerateCsgRec($S = \{v_4\}, X = \{v_1, ..., v_4\}$)	
$N = \mathcal{N}(\{v_4\})\backslash\{v_1, ..., v_4\} = \emptyset$	
$emit(\{v_3\})$	$\{v_3\}$
EnumerateCsgRec($S = \{v_3\}, X = \{v_1, v_2, v_3\}$)	
$N = \mathcal{N}(\{v_3\})\backslash\{v_1, v_2, v_3\} = \emptyset$	
$emit(\{v_2\})$	$\{v_2\}$
EnumerateCsgRec($S = \{v_2\}, X = \{v_1, v_2\}$)	
$N = \mathcal{N}(\{v_2\})\backslash\{v_1, v_2\} = \{v_3, v_4\}$	
$emit(\{v_2, v_3\})$	$\{v_2, v_3\}$
$emit(\{v_2, v_4\})$	$\{v_2, v_4\}$
$emit(\{v_2, v_3, v_4\})$	$\{v_2, v_3, v_4\}$
EnumerateCsgRec($S = \{v_2, v_3\}, X = \{v_1, v_2, v_3, v_4\}$)	
$N = N(\{v_2, v_3\})\backslash\{v_1,...,v_4\} = \emptyset$	
EnumerateCsgRec($S = \{v_2, v_4\}, X = \{v_1, v_2, v_3, v_4\}$)	
$N = N(\{v_2, v_4\})\backslash\{v_1,...,v_4\} = \emptyset$	
EnumerateCsgRec($S = \{v_2, v_3, v_4\}, X = \{v_1, v_2, v_3, v_4\}$)	
$N = N(\{v_2, v_3, v_4\})\backslash\{v_1,...,v_4\} = \emptyset$	
$emit(\{v_1\})$	$\{v_1\}$
EnumerateCsgRec($S = \{v_1\}, X = \{v_1\}$)	
$N = \mathcal{N}(\{v_1\})\backslash\{v_1\} = \{v_2, v_3\}$	
$emit(\{v_1, v_2\})$	$\{v_1, v_2\}$
$emit(\{v_1, v_3\})$	$\{v_1, v_3\}$
$emit(\{v_1, v_2, v_3\})$	$\{v_1, v_2, v_3\}$
EnumerateCsgRec($S = \{v_1, v_2\}, X = \{v_1, v_2, v_3\}$)	
$N = N(\{v_1, v_2\})\backslash\{v_1, v_2, v_3\} = \{v_4\}$	
$emit(\{v_1, v_2, v_4\})$	$\{v_1, v_2, v_4\}$
EnumerateCsgRec($S = \{v_1, v_2, v_4\}, X = \{v_1, v_2, v_3, v_4\}$)	
$N = N(\{v_1, v_2, v_4\})\backslash\{v_1,...,v_4\} = \emptyset$	
EnumerateCsgRec($S = \{v_1, v_3\}, X = \{v_1, v_2, v_3\}$)	
$N = N(\{v_1, v_3\})\backslash\{v_1, v_2, v_3\} = \emptyset$	
EnumerateCsgRec($S = \{v_1, v_2, v_3\}, X = \{v_1, v_2, v_3\}$)	
$N = N(\{v_1, v_2, v_3\})\backslash\{v_1, v_2, v_3\} = \{v_4\}$	
$emit(\{v_1, v_2, v_3, v_4\})$	$\{v_1, v_2, v_3, v_4\}$
EnumerateCsgRec($S = \{v_1, ..., v_4\}, X = \{v_1, ...v_4\}$)	
$N = N(\{v_1, ..., v_4\})\backslash\{v_1,...,v_4\} = \emptyset$	

Table 3.1: Call stack for DPCCP algorithm over the query shown in Figure 3.1

When the EnumerateCsg routine terminates it should have emitted all of the CSGs of the given graph, which is fed into Algorithm 3, which will commence the second phase of enumerating the csg-cmp-pairs, namely generating the complements for each of the CSGs. We use the following definitions in the algorithm:

> **Definition 3.4:**
> Let $G = (V,E)$ be a connected query graph, where each $v \in V$ is enumerated according to a breadth first search; and S be a non-empty subset of V, $S \subseteq V$. We then define $min(S)$ to be $min(\{i|v_i \in S\})$.

Algorithm 3 EnumerateCmp [10].

1: **Input:** A connected query graph $G = (V, E)$ and a connected subset S_1.
2: **Precondition:** Nodes in V are numbered according to a breadth-first search.
3: **Output:** Emits all complements S_2 for S_1, such that (S_1, S_2) is a csg-cmp-pair
4: $X = \mathcal{B}_{min(S_1)} \cup S_1$
5: $N = \mathcal{N}(S) \setminus X$
6: **for all** $v_i \in N$ by descending i **do**
7: emit($\{v_i\}$)
8: EnumerateCsgRec($G, \{v_i\}, \mathcal{B}_i$);
9: **end for**

The EnumerateCmp algorithm will given a connected subset emit all the complements for the subset, meaning that it will generate multiple pairs for each of the CSGs that we previously generated. The algorithm is very similar to what we do when enumerating the connected sub-graphs. However, in this case the exclude set will contain all the vertices in the connected subset itself, as well as all the vertices, v_j, where j is greater than $min(S_1)$. As seen in Definition 3.4, min for a set is defined to be the smallest i, or in other words, the enumeration of the one that was originally used in the generation of the specific CSG. In this way, we can again avoid duplicates by excluding all of the ones that have been considered before. When the exclude set is calculated, the complement generation uses the exact same approach as the CSG generation.

We implemented most of this approach in our system, only missing an actual cost function and translation from the DPCCP plan back to a Flink plan. However, we quickly ran into issues when testing this version on our queries. On some queries, the algorithm would consume more time than it took actually executing the query. The time consumed for generating a plan for each individual query is shown in Table 3.2. The top row in the table reveals the results of the stanard recursive version of the algorithm.

	C1	**C2**	**C3**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
Recursive	11013.0	26817.0	3951.0	38025.2	913.2	860.4	792.6	0.0	361.2	316.8
Iterative	81.0	310.0	16.0	1823.8	0.8	0.8	2.0	0.0	0.8	0.7
	F1	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**
Recursive	1812.6	15611.6	4468.8	31653.4	4267.2	314.2	323.6	77.0	76.4	324.8
Iterative	5.2	144.0	5.2	517.6	4.8	0.2	1.0	0.0	0.2	0.6

Table 3.2: Average time consumed on generating plans for the different queries in ms.

The result clearly indicate that the recursive approach does not work well as we have plan generation times up to 26 second, which is in most cases an order of magnitude more time than what is required to execute the unoptimized version of the query. As

recursive calls can be detrimental to performance we decided to design and implement an iterative version of the DPCCP enumeration algorithm. The algorithm is seen in Algorithm 4, which reveals that we instead of the recursive calls utilize a queue in order to enumerate the csg-cmp-pairs in the correct order.

Algorithm 4 Our iterative version of EnumerateCsg [10].

Input: A connected query graph $G = (V, E)$, where $V = \{v_0,...,v_{n-1}\}$.
Precondition: Nodes in V are numbered according to a breadth-first search.
Output: All subsets of V that induces a CSG of G
for $i = n - 1$ down to 0 **do**
    emit($\{v_i\}$)
    EnumerateCsgIterative($G, \{v_i\}, \mathcal{B}_i$);
end for

EnumerateCsgIterative(G, S, X)
Let Q be a queue.
Q.enqueue(G,S,X)
while Q is not empty **do**
    current = Q.dequeue()
    $N = \mathcal{N}(current.S) \setminus current.X$
    for all $S' \subseteq N, S' \neq \emptyset$, enumerate subsets first **do**
        emit($S \cup S'$)
        Q.enqueue($G, (S \cup S'), (X \cup N)$)
    end for
end while

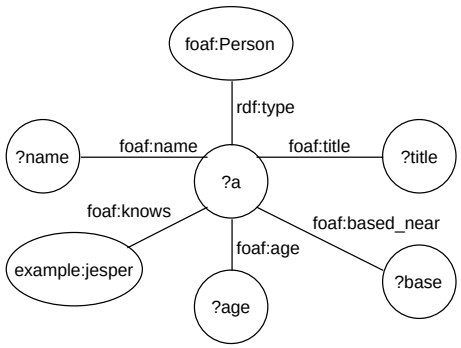
As seen in Table 3.2 the plan generation times have been reduced significantly for every query type. However, several of the query types still take a fair bit of time. Even though the complex queries only consume between 16ms and 310ms on plan generation, we have in our experiments utilized a overly simplified cost function which just return 0, i.e. it introduces no apparent overhead. If we introduce a cost function that would spend just 0.25ms in checking the cost of a plan C1, would have to spend around 1000ms more on checking all of its 2304 csg-cmp-pairs, and that is for one of the better queries. If we were to do the same calculation for S1, we would not even see the plan generation complete within a couple of minutes. So even though the plan generation for most of the query types show reasonably good performance, we still have some reservations about going all in on this approach.

After further investigation we found, that having what the creators of DPCCP refer to as clique queries in SQL, yielded high amounts of subproblems. Clique queries are queries where the query graph contains subgraphs in which all nodes are connected to each other on the same join condition, which is relatively rare in standard SQL. However, clique queries are a very common thing in SPARQL, as this corresponds to what is defined in SPARQL as star queries. As we mentioned in our previous work [14], star queries are the most common query types, and is a base for many of the different query types in the WatDiv test suite [2].

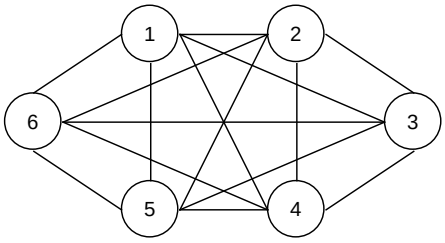
An example of how a star query in SPARQL translates to a clique query graph is shown Figure 3.2, where we show the different representations of the SPARQL star query shown in Figure 3.2a. When looking at the SPARQL graph in Figure 3.2b, it is clear to see why it is called a star query as we have one focal point, `?a`, with outgoing edges. If we were to create a query graph for this specific query we end up with the graph shown in Figure 3.2c, where each node is numbered according to the order of the triple patterns in the original query. The final result clearly shows that we are dealing with a graph with one large clique, i.e. all nodes in the graph are connected to every other node in the graph.

```
SELECT ?name, ?age, ?base, ?title WHERE {
   ?a rdf:type foaf:Person .
   ?a foaf:name ?name .
   ?a foaf:age ?age .
   ?a foaf:based_near ?base .
   ?a foaf:title ?title .
   ?a foaf:knows example:jesper
}
```

(a) Example of a SPARQL star query.



(b) SPARQL graph representation.



(c) Query graph representation.

Figure 3.2: Illustration of how a SPARQL star query directly translates to a query graph containing one large clique.

Having these clique or close-to clique queries being a common thing in SPARQL is also apparent in the query types that we have tested on. The complex queries (CX) along with the star (SX) and snowflake queries (FX) are all highly connected queries. The problems arise when the number of relations involved in these queries increase, as we have an exponential growth of subproblems. This makes this approach unattractive to use in the context of SPARQL, as finding the best plan in some cases would take longer than executing an unoptimized version of a complex query, and our primary purpose is to find a strategy that performs well for every query type. For these reasons, we decided to try some different approaches for bushy plan generation. These approaches are described in the following sections.

3.2.2 CliqueSquare

After our attempt at the DPCCP algorithm which did not prove fruitful, we instead decided to look at other possibilities. The main limitation in DPCCP algorithm used for SPARQL was the high amount of joins coupled with a rather low amount of join variables, which therefore resulted in a lot of large clique joins. DPCCP could not enumerate these efficiently and it therefore resulted in extremely high processing times for some queries. The high amount of cliques in SPARQL query graphs is something that

has previously been exploited in one other system that we know of called CliqueSquare [5], because a clique corresponds to an n-ary star equi-join. We therefore looked into their algorithm and wanted to attempt an implementation for our system such that it could be combined with the Extended Vertical Partitioning approach that our system was focused on last semester.

CliqueSquare works by creating a bushy plan that is as flat as possible, which means that they the authors try to minimize the amount of joins when traversing the query plan from root to leaf. The focus of CliqueSquare is to create n-ary relations, or multi way joins, that are connected by joins. Sadly, this would not work for Flink, since Flink does not have an implementation that allows for multi-way joins in their Table and SQL API. As with many other things in Flink, it is one of the nice features that the developers are aware of, but does not have a prospect for implementation currently. We therefore looked into the CliqueSquare implementation[1], and found a method which could supposedly prune n-ary plans and therefore hopefully give us a binary bushy plan, where the join variables are primarily constricted to individual subtrees. This would, as mentioned in Section 2.4, allow us to compute individual subtrees in parallel, which would hopefully increase our performance. This was one of the documented and guaranteed ways to make bushy plans for SPARQL queries. We therefore decided to proceed with the implementation, even though we knew that it might not work out for us in the end.

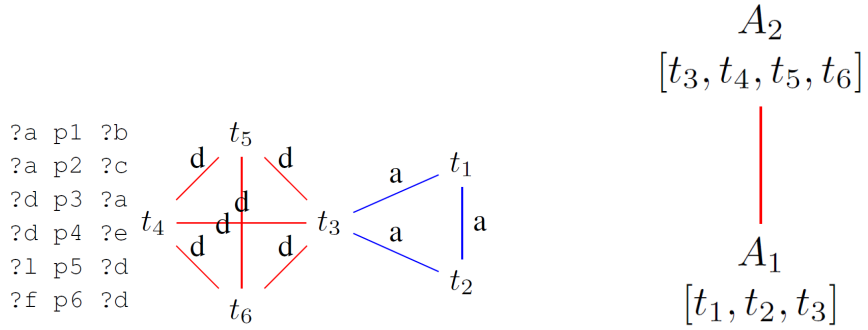
So we went on with an attempt at utilizing as much of the CliqueSquare code as possible, with as few changes as possible during our implementation. The CliqueSquare system additionally has several different modi operandi, but we restricted our attempts for an implementation to a single one, as it would otherwise take way too much time for us, when we had to verify the success/failure of each modus. We decided to go with one of the modes that the authors of CliqueSquare themselves say is one of the better ones, which is called MSC. This abbreviation corresponds to an algorithm that utilizes minimum set covers, simple covers, and the absence of a + at the end means that it also utilizes partial cliques. A simple cover means that a node is allowed to be part of several cliques. A minimum set cover indicates that the plan contains the minimum possible number of cliques, which is desirable as a clique can be directly translated into a join. A Partial clique means that a join can be made that only utilizes part of the clique in the join, even though there are more available nodes sharing the join variable. The algorithm from [5] can be seen in Algorithm 5.

[1] Source available from https://sourceforge.net/projects/cliquesquare/

Algorithm 5 The cliquesquare algorithm from [5].

```
1: Input: Variable graph G; queue of variable graphs states.
2: Output: Set of logical plans QP
3: states = states ∪ {G};
4: if |G| = 1 then
5:     QP ← CREATEQUERYPLANS(states);
6: else
7:     QP ← ∅;
8:     D ← CLIQUEDECOMPOSITIONS(G);
9:     for all d ∈ D do
10:        G′ ← CLIQUEREDUCTION(G, d);
11:        QP ← QP ∪ CLIQUESQUARE(G′, states);
12:    end for
13: end if
14: return QP
```

The way the algorithm works is by first identifying all clique decompositions for the input graph, i.e. all possible sets of cliques that cover all nodes. The cliques in the decomposition can either be partial or maximal where we allow partial cliques as mentioned before, and the nodes can be part of multiple cliques. After finding all the decompositions, they are iterated through, and according to the authors the clique reduction is found. What is referred to as clique reduction here is basically what is commonly referred to as *Edge Contraction*. This means that if one clique consists of nodes *t1*, *t2*, and *t3*, and another clique consists of nodes *t3*, *t4*, *t5*, and *t6*, if these two cliques are connected somehow, we now insert two nodes in the reduced graph called *t1,t2,t3* and *t3,t4,t5,t6*, which are connected by an edge. This example can be seen in Figure 3.3, where the where clause from the query can be seen in Figure 3.3a with its corresponding query graph. The first step of the reduction can be seen in Figure 3.3b. An additional step can be made where the reduction from Figure 3.3b is reduced further until one node remains. The algorithm is then executed recursively on these reduced clique graphs, and once the size of the graph reaches 1, a logical query plan gets created and returned. The output is then a list of logical query plans.



(a) Triple patterns from query and the corresponding variable graph with cliques clearly visible

(b) Step 1 reduction of variable graph from Figure 3.3a

Figure 3.3: Variable graph and its step 1 reduction. Both figures are taken from [5]

Our implementation of the algorithm works fine, and executes rather fast, but we ran into an unforseen issue. The entire foundation for our reasoning to implement the algorithm was the possibility of pruning plans that had bushy n-ary joins instead of a bushy plan with binary joins. It was only after the implementation that we realized that most of the "good" modi operandi for CliqueSquare only gave one single query plan for star joins, which therefore meant that there was only a single plan to choose from in many of our queries, and that plan was one that contained an n-ary join. So after confirming that CliqueSquare approach does indeed work on SPARQL queries, we were unable to progress further with this approach before we had found solution for how we wanted to deal with the multiway joins. This lead us to consider a greedy algorithm approach, which we will cover in Section 3.2.3.

3.2.3 Greedy Algorithm

After multiple unsuccessful attempts in implementing a bushy plan generator we decided to utilize a simpler approach, namely using a greedy algorithm instead, which continuously selects the join with the lowest cost according to some cost function. Greedy algorithms for query plan generation and join reordering is no novel approach when it comes to SQL [3]. However, we have not been able to identify any previous work within the area of SPARQL and RDF processing that have done so. The main advantage of utilizing a greedy approach is the speed of which one can find a close to optimal plan, making it superior to the DPCCP approach as it does not use an order of magnitude more time to find an optimal plan than actually executing the query.

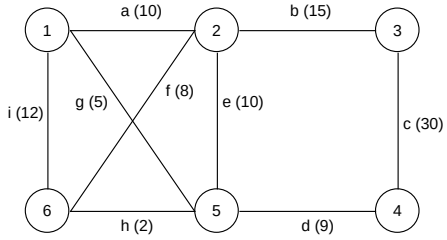
As already mentioned the greedy approach is fairly simple, which also is reflected in Algorithm 6, where the algorithm for our approach is shown. The algorithm is based on the concepts introduced in [3].

Algorithm 6 A simple greedy algorithm for generating binary bushy plans

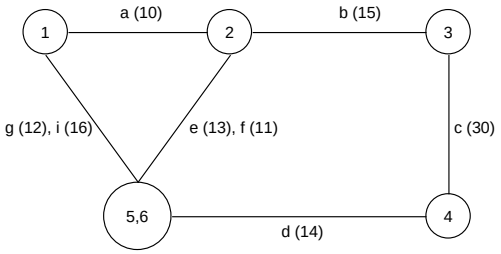
```
1: Input: A connected query graph G = (V, E), where V = {v_new,...,v_n−1}.
2: Output: A binary bushy join tree
3: while |V| > 1 do
4:     // Find best join
5:     for all (v_i, v_j) ∈ E do
6:         if cost(bestJoin) > cost(v_i, v_j) then
7:             bestJoin = (v_i, v_j)
8:         end if
9:     end for
10:    Let the nodes of the best join be called v_i and v_j.
11:    Create v_new.
12:    All edges involving v_i and v_j are moved to v_new.
13:    v_new.ancestors = {v_i, v_j}.
14:    V = V\{v_i, v_j}
15: end while
16: return V
```

The algorithm firstly identifies the best join in the graph according to our cost function. When the best join has been identified we combine it to a single node, v_{new}, and update all edges where the original nodes of the join, v_i and v_j, is involved to the new node, i.e. if some edge, (v, v_i), exists, we replace this edge with (v, v_{new}). Lastly, when all edges are replaced, we add v_i and v_j as ancestors to v_{new} and in this way we build up a join tree. The original nodes are removed from V, and the process is repeated until there is only a single node left, which is the root of our join tree.

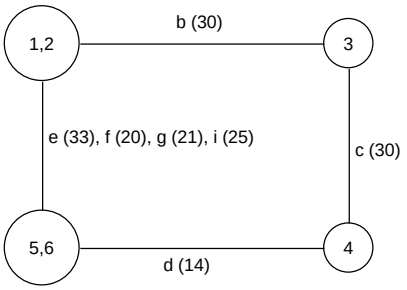
An illustration of how this process transforms a query graph whilst saving its ancestors is shown in Figure 3.4, where we, given a query graph over relations, $\{1, ..., 6\}$, and joins with their cost in parenthesis, $\{a, ..., i\}$, determine the best join order. We start with the query graph in Figure 3.4a, where we estimate the join with the lowest cost to be h. The two nodes of this join, 5 and 6, are then combined to a single node, (5,6), as seen in Figure 3.4c, and all edges involving these nodes are moved to the new node. Notice, that the cost of joins, $\{g, i, e, f\}$, are now revised as the cardinality of the combined node is most likely higher than the cardinality of the individual nodes, thereby making the joins more expensive. This is an aspect that should be handled by the cost function described in Section 3.2.4. For the following iteration, from Figure 3.4b to Figure 3.4c, we estimate a to be the cheapest join, hence 1 and 2 are combined. In repeating this process we reduce the number of nodes in the graph to a single one, which has every single node in the initial graph as ancestors. If we look at the join tree produced by this approach in Figure 3.5 we can see how the combination of the nodes translates to a bushy plan.



(a) The initial query graph.



(b) After the first iteration 5 and 6 are combined.



(c) After the second iteration 1 and 2 are combined.

(d) The result of the final iteration. All nodes have been combined to one single node.

Figure 3.4: The stages of the query graph when using the greedy algorithm.

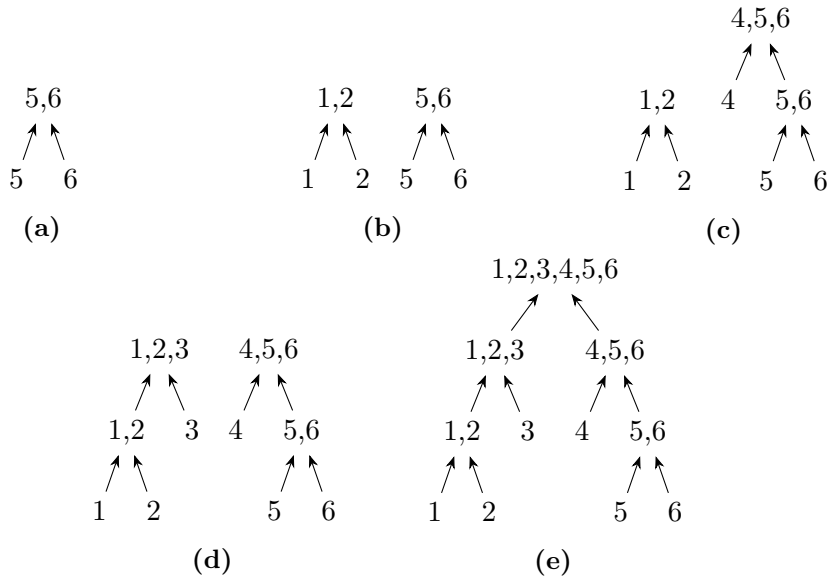


Figure 3.5: Stages of building a join tree with the greedy algorithm.

3.2.4 Cost Function

The primary aspect in Algorithm 6 for determining the structure of the join tree is the cost function. For this aspect we decided to utilize the cardinality estimation model proposed in [6], which in turn is inspired from database literature [18]. The cost model describes formulae for estimating the cardinality of individual triple patterns, i.e. data access and filter, as well as estimation formulae for the cardinality of joins between these. An extract of the estimation formulae can be seen in Table 3.3 and Table 3.4. We have only presented some of the estimation formulae here as they are rather extensive. We invite interested readers to find the formulae in full in tables 1 and 2 in [6].

Triple Pattern	Cardinality
?s ?p ?o	c_t
subjA ?p ?o	$\frac{c_t}{c_s}$
?s predA ?o	$cp_{predA,t}$
?s ?p objA	$\frac{c_t}{c_o}$
subjA predA ?o	$\frac{cp_{predA,t}}{cp_{predA,s}}$
subjA ?p objA	$\frac{c_t}{c_s \cdot c_o}$
?s predA objA	$\frac{cp_{predA,t}}{cp_{predA,o}}$
subjA predA objA	$\frac{cp_{predA,t}}{cp_{predA,s} \cdot cp_{predA,o}}$
...	...

Table 3.3: Cardinality estimation formulae for triple patterns.

In order to utilize this cardinality estimation technique there are some statistics that are needed, most of which we already have computed in our preprocessing phase. We need to have the information available to set the constants in the formulae of the two tables. We know the total amount of triples, c_t, and also the total amount of triples for a given predicate, $cp_{predA,t}$, which corresponds to the size of our vertical partitions. We do however not compute any statistics for c_s and c_o, which are the number of distinct subjects and objects in the data set. Also the similar measures, $cp_{predA,s}$ and $cp_{predA,o}$,

Type	Pattern	Cardinality
SS	?s predA ?o . ?s predB ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{max(cp_{predA,s}, cp_{predB,s})}$
SS	?s predA objA . ?s predB objB	$\frac{card(pat_1) \cdot card(pat_2)}{max(cp_{predA,s} \cdot cp_{predA,o}, cp_{predB,s} \cdot cp_{predB,o})}$
SS	?s predA ?o . ?s predB objB	$\frac{card(pat_1) \cdot card(pat_2)}{max(cp_{predA,s}, cp_{predB,s} \cdot cp_{predB,o})}$
SS	?s predA objA . ?s predB ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{max(cp_{predA,s} \cdot cp_{predA,o}, cp_{predB,s})}$
...	...	...

Table 3.4: Cardinality estimation formulae for joins.

which are the distinct subject and object count for a specific predicate will need to be calculated. These newly needed statistics are simple to calculate and can be done during the creation of vertical partitions.

We have chosen only to utilize the cardinality estimations put forward in the paper as they will provide a solid basis for selecting the optimal join order as it should be heavily influenced by IO and per record processing time, as the sizes of the intermediate results are the main factor in both costs. The implementation of the cost function from [6] was simple as each case that they present for estimating the cardinality of a triple pattern could be represented in code as an if-statement, where the condition is the "triple pattern"-pattern and the body corresponds to the estimation formula. For the cardinality estimation of joins we use the same approach to represent all the permutations of variable or constant subject, predicate, and object. We choose to ignore the special case of `rdf:type` as we have noticed that our primary test data set does not utilize it as a class entity identifier, but as any other property.

After some initial testing of the greedy algorithm with the cost function described above, we noticed that the plans produced were not necessarily bushy. We quickly discovered that this was due to the nature of our cost function. We came to realize that combined nodes where filters were involved would be prioritized higher than the initial relations, and thereby they would in many cases produce linear plans. We therefore tweaked the cost function to yield much higher costs for combined node by multiplying the weights of a combined node with a constant, which ultimately lead to flatter plans.

One thing worth noting is that this estimation approach is based on traditional relational database literature, hence it is, as with many other approaches within this area, developed with focus on queries with fewer joins, than what we often see in SPARQL queries. As more relations, or triple patterns, are involved in a join the harder it becomes to estimate the cardinality correctly. This means that even though the approach might function well in traditional relational databases, it might not be the case for the purpose of RDF and SPARQL. The cost function described above also does not consider is that we are utilizing ExtVP and we do thereby not consider the correct cardinalities for the initial input tables. Furthermore, we have more information available about the joins of two relations, which we might be able to use for the joins on the lowest level. This is an area where we could potentially see improvements in the cardinality estimates, however due to limited time we have chosen not to pursue it any further, but it is definitely an area worth looking into if SFRDF were to be further improved.

3.2.5 Executing Bushy Plans in Flink

The implementation of the greedy algorithm is almost an exact implementation of Algorithm 6. However, some more effort were needed in order to execute the generated graph in Flink. In order to get it to execute in Flink we had to modify the pipeline that existed. Figure 3.6 shows a simplified overview of the SFRDF+ and Flink system. Flink specific items have a red background, whereas our modules from the original SFRDF [14] and additions in SFRDF+ are shown in yellow and green respectively. The parts in blue are Flink components or utilities connected to their streaming API that we currently do not utilize, however, it should be possible to enable our current setup to run SPARQL queries on streams, without too much effort. This is definitely an area that could be worth looking into in the future.

Figure 3.6: Overview of SFRDF+ and Flink.

As seen in Figure 3.6, the original pipeline utilized the query translator implemented in S2RDF, which translated the SPARQL queries to SQL queries. This was convenient as the produced SQL files are directly executable by the Flink SQL and Table API. However, in translating the query to SQL the information needed for our greedy algorithm became harder to obtain as triple patterns was translated to sub-queries. Furthermore, we could not firstly utilize the greedy algorithm and then use the S2RDF translator as the bushy nature of the join order then would be lost. We instead decided to directly translate the queries from SPARQL to the Logical model in Flink. This is done by firstly utilizing the Jena framework to produce an abstract syntax tree, which we in turn generate a query

graph for (see Section 2.5). This graph is then fed to the algorithm which produces a join tree with the help of the statistics made by our data converter (not shown in the figure). The join tree is then finally converted to a Calcite Logical Plan, which is logical model in the Flink Table API. By using this logical model we are still able to benefit from the rule-based optimizer from Flink and allow the framework to generate a plan for execution on the cluster. Our evaluation of the greedy algorithm will be presented in Section 4.4.

4 | Evaluation

This chapter will cover an in-depth evaluation of the changes that we have made to SFRDF. We will before describing the results of our evaluation firstly cover our experimental setup, including the specifications of the machines that we will run our experiments on and how we have fine tuned parameters for Flink in order for it to run more smoothly on high-end hardware. This is followed by descriptions of the different experiments that we have run along with the results of them, and our thought on why we see the results that we do. The experiments include tests of the features that have been described throughout the report, but also an experiment for evaluating how well we scale with both the size of the data sets that we run on and the number of compute nodes.

4.1 Experimental Setup

For the evaluation of our system this semester, we decided to utilize a different cluster than what we used in [14]. For our previous system evaluation we utilized a cluster running on Amazon Web Services (AWS), which had a variety of issues that we implore you to read [14] for an overview of. Our decision to change cluster was, in addition to the issues we had with AWS, partly based on AWS being costly, and partly Abacus providing us with better hardware. Abacus 2.0 is placed at 355th place on TOP500 [9] clusters in the world, with its 14016 cores and its theoretical maximum performance of 766.6 TFlop/s.

4.1.1 Hardware Specifications

The cluster consists of different types of nodes, namely slim nodes, fat nodes, and GPU nodes [4]. The cluster has 448 slim nodes, 64 fat nodes, and 72 GPU nodes. Each of the different types of node shares the same base configuration, which is seen in Table 4.1.

CPU	Intel Xeon E5-2680v3
Number of CPUs	2
Cores per CPU	12
Core Clock Speed	2.5GHz
RAM	64 GB
Storage	200 GB SSD

Table 4.1: Base configuration for Abacus 2.0 nodes.

Except for some of the slim nodes which have 400 GB SSDs, most of them adhere to the base configuration. The fat nodes and the GPU nodes differ in two different ways. The fat nodes are configured with more memory and storage, i.e. 512 GB RAM and 400 GB SSDs instead of 64 GB and 200 GB respectively. On the other hand, the GPU nodes have had two nVidia K40 GPU cards added to their setup. All the nodes in the cluster are connected with Infiniband FDR, which has a max speed of 56 GB/s.

Previously we have been using HDFS for shared storage between the nodes in the AWS cluster, but for this semester the cluster already comes with IBM Spectrum Scale [7], previously known as General Parallel File System (GPFS), which should be at least as efficient as our previous approach. IBM Spectrum allows for a more transparent integration with the native file system of the nodes, and thereby simplifies the process of using the filesystem.

A problem in many high performance systems that utilize many CPUs is that the main bus in such a system might be a bottleneck. Therefore, the nodes in the cluster, as well as many other high performance systems, utilize a Non Uniform Memory Access (NUMA) architecture [8] for its two CPUs. NUMA architecture tries to mitigate the bottleneck by having a dedicated connection to a segment of RAM (local memory) for a group of processors, and furthermore have a shared NUMA bus that allows for access to the memory segments used by the other processors in the system (remote memory). This makes the memory access times non-uniform as it will take longer to access the remote memory than the local one, but in general it allows for faster access to memory.

For our purposes we will be using the Slim nodes with the base configuration together with the GPFS network storage provided in the cluster. When having two CPUs in a NUMA architecture it furthermore makes sense for us to have two `TaskManagers` (slaves) on each node. For example if we were to use three slim nodes, we would have one node dedicated to being the `JobManager` (master), and then the remaining two nodes would run two instances of the `TaskManagers` each. This gives us a total of four `TaskManagers` and one `JobManager`, which will be the most common setup in our experiments.

4.1.2 Slurm Workload Manager

We cannot possibly take advantage of all the hardware in the cluster, and the cluster is shared amongst several users. Therefore, the cluster maintainers have installed Slurm Workload Manager, which is scalable cluster management solution for Linux based clusters. It allows for scheduling and allocating resources, i.e. nodes and switches, for different jobs. The scheduling ensures fairness between the jobs, and tries to utilize as much of the available hardware at all times.

For our purposes we will need to be able to run Flink through the use of the Slurm commands that we have at our disposal, namely `sbatch` and `srun`. The first of the two, `sbatch`, allows us to run batch scripts, which will firstly allocate some resources according to the command line arguments that we pass to the command, and then it will run a shell script passed to the command. This shell script can then utilize the latter of the two commands, `srun`, to start "job steps" within the batch workload. This allows for starting multiple parallel tasks, which in our case is useful for starting the `JobManager` (Master) and `TaskManagers` (Slaves) of Flink. This was how we originally intended to run our workload. However, we encountered issues with this approach that did not allow us to choose which node the different parts should be running on. This entailed that we would start all the `TaskManagers` on the same compute node, and at the same time could not control which node the `JobManager` should run on. Having control over where the `JobManager` is located is crucial for the `TaskManagers` to know in order to register. We were of course not interested in this, so we modified the script to utilize SSH instead of using `srun`, allowing us to have more control over which nodes that ran the different managers.

4.1.3 Flink on Abacus 2.0

Moving to a cluster where Flink was not just a one-click install also entailed issues regarding configuration. We had on the previous cluster experienced that some customization of the framework was needed, however, on the new cluster the configuration was extensive and tiresome. The first issue that we encountered was the number of network buffers that the framework has preconfigured. The framework simply ran out under execution because of the high number of threads, which each needed to communicate with other parts of the framework. We firstly tried to increase the number of network buffers to a number based on an equation in the Flink documentation replicated in Equation 4.1, where #slotsPerTM is the number of slots per `TaskManager`, i.e. how many tasks it can run, and #TMs is the number of `TaskManagers`.

$$\text{Number of buffers} = \#\text{slotsPerTM}^2 \cdot \#\text{TMs} \cdot 4 \tag{4.1}$$

This only allowed our system to run longer before crashing. Therefore, we increased the number of network buffers to an absurd amount, which crashed the system because of memory limitations. Lastly, after many trials and failures we found a number that seems to work, but it might entail problems, when the sizes of the data sets increase.

Another consequence of us moving to a different cluster was having a very high amount of memory. This, apparently, is not altogether a good thing when working with Java and JVMs, due to the garbage collection system in Java, which is not designed for large amounts of memory. The time it takes for the JVM to do garbage collection is proportional to the size of the JVM heap. This caused us to experience a significant downgrade in performance on the new cluster, even though it had better hardware than the previous one. After some investigation and configuration of the framework, we found that adjusting several parameters in conjunction worked the best. The first parameter that we changed was the heap size of the `TaskManager`, called *taskmanager.heap.mb*. Originally we intended this parameter to be as high as possible, however as already mentioned this is not a good idea when working with Java garbage collection. Therefore, we tried to reduce this to a much lower amount, which yielded much better performance, but would crash at bigger queries and data sets due to the memory not being large enough. We started up with some initial values of a few gigabytes, but ended up using 35% of the available memory for every `TaskManager` on each node. This is around 44.8 GB for both `TaskManagers` out of the 64 GB. We are able to set the memory this high due to the next parameter, which is called *taskmanager.memory.size*. This parameter describes the amount of the heap memory that should be managed by Flink and how much should be managed by standard Java for executing user defined functions. This value is normally set to be a percentage of the total heap size, however for our purposes a flat amount seemed better, as this could possibly allow us to scale up the memory managed by Flink without changing how much was left for the JVM. This was desirable due to the bad performance of the JVM garbage collector mentioned previously, and since Flink does not suffer from the same garbage collection overhead, as it manages the garbage collections in a much more efficient manner. This allowed us to increase the total heap to a large amount and only have a low amount for our user defined objects, which was managed by the JVM garbage collector. The higher this parameter is set, the more memory we give to Flink out of the available heap, so for the queries it was possible to execute them with a value of *taskmanager.heap.mb - 1024*. For the data conversion and thereby the creation of our ExtVP we had to increase the value of the variable once we reached data set of 100 million triples and above.

After the configuration of the heap space, we were at a state where we could run the data conversion for the first two scale factors on the cluster. However, when we tried to run the third one we experienced another issue. This time, the size of the messages sent between nodes got too large, meaning that we had to configure another parameter, *akka.framesize*, which describes the maximum allowed size of the messages sent between nodes. Again, we had a parameter that for some data sets can be small and thereby gain performance, but will have to be increased for large data sets. We ended up using a maximum framesize of 512 MB, as this seemed to fit our purpose without allocating too much memory for the messages and making everything crash. The Flink team is also working to resolve issues regarding the framesize by splitting packages larger than the parameter into several smaller messages.

It was a common issue, that configuration parameters were reliant on each other, which made it difficult to pinpoint what caused the different inefficiencies and other issues that we encountered. This included the previously mentioned parameters, such as the amount of memory on each node and what the memory was allocated for on a node, as well as the size of the data set and lastly the amount of nodes. The greatest problem of all of the configuration was the amount of time it took before we could tell if tweaking a parameter worked. It could take up to an hour before the system crashed due to an exception or similar related to the configuration. We could then adjust the parameters and wait again. This caused a lot of headaches and cost us a lot of time that could otherwise have been spent improving the actual system. At some point we decided that we had hurry up and start executing queries, which was currently blocked by ExtVP generation crashing after hours of running, so we made a feature to mitigate the issue with crashes. The system already had features implemented where it was able to receive parameters that defined whether it should generate data sets, execute queries, or both. We changed SFRDF+ such that if the option to generate the data set was passed to the program, it would check the target folder for existence of specific files and data partitions, and then resume the workload by starting over from the furthest possible step from the beginning. This feature helped us to move past the bottleneck we were facing and actually start executing queries. We would then, at a later stage, return to the generation of these data sets in order to properly measure their execution time.

In general, we experienced a lot of issues in moving to the new cluster, which made it a time consuming and tiresome task. We can not know for sure if the fine tuning of all the parameters in the Flink configuration was only due to us moving cluster, as we did not run the data conversion task on the cluster during our previous experiments. It is however certain that we during this setup spent a considerable amount of time that could have been better spent. We were however not able to keep utilizing an Amazon Web Services cluster due to it being too costly, which did, on the positive side, allow us to run our system on better hardware. We will, due to the hardships with configuring the flink framework and the loss of time related to this, only compare SFRDF+ against our system from [14], SFRDF. We simply do not have the time to set up a completely different framework that possibly could entice the same amount of configuration overhead as we have with Flink. Even though we do not compare results directly in this paper, our system can, through the results of [14] and the results of this paper, be compared to S2RDF [16], and thereby to other recent SPARQL processing systems if so desired.

4.1.4 Query Setup

During the evaluation phase we will necessarily have to execute some amount of queries, in order to test the effectiveness of the improvements we have made. We have decided to do this by using the WatDiv queries, because they fit the data sets that we also use from WatDiv [2]. The data set can be generated to the size you want by specifying the scale factor (SF) as a parameter to the data set generator. An SF of 1 or SF1, corresponds to 100 thousand triples, and any other SF is a product of this multiplied by the SF, e.g. the SF10 data set would contain 100000 triples $\cdot$ 10 = 1 million triples. The queries on the other hand are actually query templates, which means that in almost all of the queries, there is a place where some value can be replaced. An example of a query template is seen in Code Snippet 4.1.

```
1 SELECT ?v0 ?v1 WHERE {
2     ?v0 wsdbm:likes ?v1 .
3     ?v0 wsdbm:subscribes    %v2% .
4 }
```

Code Snippet 4.1: An example of a query template.

These query templates will, for all the tests, be instantiated 5 times, where the template variable has been changed every time, which we call query permutations. This means, that for one query we might have a triple pattern such as `?v0 wsdbm:subscribes %v2%`, which can be instantiated as `?v0 wsdbm:subscribes 'Peter'` if *Peter* is a valid value for this triple pattern. One of the other 4 permutations for the template might be `?v0 wsdbm:subscribes Alice`, in order to test the query on different parts of the RDF graph. Each of these 5 permutations will additionally be executed 20 times each, in order to get a better average of each individual permutation. This effectively means that every single query is executed 100 times (5 permutations $\times$ 20 executions), except for the complex queries *C1* through *C3*, as they do not have any constants to make permutations from, and they are therefore simply executed 20 times each. During the experiments, we noticed fluctuations in the response times of the queries. These fluctuations caused outliers to be present in our results, which affected the average response times negatively. The outliers were more prevalent when the JVM managed heap space was high, which indicates that garbage collection was part of the cause. In order to mitigate the effect of the outliers on the average response times, we decided to use only use results within the 5% and 95% fractiles.

4.2 Object Reuse and Optimization

During our evaluation of the system, we discovered two individual features in Flink that are supposed to provide an improvement in performance, so we decided to test them out. These features in Flink are called *object reuse* and *code analysis*, and they are supposed to help in two different ways. The object reuse feature reduces the amount of object instantiations inside the Flink parts of the code, which in turn might decrease the necessity of Java garbage collection. The code analysis makes a pre-interpretation of the user defined functions in order to give the Flink optimizer some insight regarding possible optimizations of the code.

In order to test whether *object reuse* and *code analysis* has any effect on performance, we have decided to create a type of split test (commonly referred to as A-B test) [13]. The general method is to subject certain control groups for different versions of your product, and then selecting one of the versions based on which one receives the most positive feedback. When doing split tests we will use SF1, SF10, and SF100. We do not scale to larger data sizes, because we just want to measure the general tendency when

performing the test, and whether or not it helps us when scaling to more data. In this case, the focus is on which version of our system will have the better performance: the original SFRDF or SFRDF with these optimizations enabled.

We executed the base system, which essentially is our implementation from [14], in addition to a version of the same system where object reuse and code analysis had been enabled, which we will refer to as *Reuse* in the rest of the report. The results of this was rather insignificant, as the average query speed had a mere difference of around 4 milliseconds for both SF1, SF10 and SF100. The query averages can be seen in Table 4.2, and the full query results can be found in Appendix A. The only remarkable thing to notice from these results is, that Reuse goes from being 4 milliseconds slower to being 3.6 milliseconds faster, but this change is simply too small to confirm whether or not this was merely a coincidence.

System	SF1	SF10	SF100
SFRDF	1605.1	1680.6	1719.8
Reuse	1609.1	1684.6	1716.2

Table 4.2: Average query speeds for SFRDF vs SFRDF reuse

Since Reuse does not perform much worse than base SFRDF and it does seem like it might improve performance when scaling up, there is no reason for us to not use the features for the rest of the tests. This means, that for the rest of the evaluation section, *all* tested system variations include these two optimizations.

4.3 Dictionary Encoding

In order to test the dictionary encoding that we have introduced in SFRDF+, we evaluate a split test between our standard system and the system using the dictionary encoding. Even though Section 2.3 mentions the huge benefits on both computation time needed for comparisons and the reduction in necessary memory and disk, it is not certain that all the expected improvements will actually be present when executing the code on our cluster machine. Additionally, we do not know if Flink has an underlying implementation that makes the strings more efficient in some way.

We have devised various experiments using different data sizes in order to test how useful the encoding will be at different workloads, i.e. SF1, SF10, and SF100. We compare the time required for creating the dictionary encoding, creating the partitions, running the queries as per usual, and how much time would be needed in order to translate the results back into a textual format.

4.3.1 Partition Speed and Space

In the case of SF1, seen in Table 4.3 we can see that the time required to make the initial dictionary is rather small. It only takes 6 seconds to create the dictionary, which basically means it has no influence in the total run-time of 4322 seconds. After the dictionary is made, we create VP which is 14 seconds faster for the dictionary encoded version, which means that we already here get the 6 seconds back that we spent on the dictionary. The total difference between the execution is 4322 seconds versus 4566.4 seconds for dictionary and non-dictionary respectively. This result shows that the dictionary encoded version had a 4 minutes faster execution time, which corresponds to 94.7%. Some amount of time can therefore be saved, even on small data sets, which might be promising once we start scaling to much bigger data.

Scale Factor 1

In Table 4.4 we see that the dictionary encoded version uses only 61.4% of the disk space that is used by the string encoded version, which is 16.64 MB versus 27 MB. Saving the dictionary to disk results in needing an additional 0.64 MB disk space, which corresponds 3.85% of the total space used for dictionary encoding. This ensures that the created partitions can be reused in case the system gets turned off and has to be restarted. That the used disk space is so much lower than the normal encoding furthermore means, that significantly less memory must also have been necessary in order to operate the data, which makes the dictionary encoding much more usable for computers with low-end hardware.

	Dict.	VP	SS	OS	SO	Total
Dictionary	6.9	58.2	2558.0	863.1	835.5	4322.0
Normal	N/A	72.2	2817.3	847.9	829.0	4566.4

Table 4.3: Time required to create partitions in seconds for SF1.

	VP	ExtVP	Dictionary	Total
Dict.	9.4	6.6	0.64	16.64
Normal	24	13	N/A	27

Table 4.4: Disk usage of partitions in MB for SF1.

Scale Factor 10

Next we take a look at the data for SF10, which can be seen in Tables 4.5 and 4.6. Here we can see that the time required to make the dictionary for 10 times as much data is not even twice as much as for SF1. Strangely enough, the dictionary encoded version is faster at making standard VP, and almost even for the SS relation type, but falls off quite a bit for the OS and SO types. This results in an advantage for the normal encoding of 8.78 minutes, which is quite a lot. The total execution time is now 81 minutes for the dictionary encoded version, which is 9 minutes slower than SF1. The total execution time for the normal encoding is 76.1 minute for SF1 versus 72.35 minutes for SF10, where it actually has gained 4 minutes. A remarkable increase in speed that we are not entirely sure where comes from, as the system needs to process 10 times as much data. It might be related to inefficient garbage collection, noise on the network from other users during the SF1 execution, or something else entirely.

The disk usage for the dictionary encoded partitions is now 106.7 MB, which is merely 37.43% of the 285 MB used for the normal encoding. A very big reduction in both memory and disk space required for using and storing the partitions. The size of the dictionary is now 5.7 MB, which is 5.34% of the total disk space used for the dictionary encoding. This is a slightly higher percentage compared to SF1, but due to the fact that the dictionary is still fast to make while making the system much more space efficient, there is no big drawback to this slight increase.

	Dict.	VP	SS	OS	SO	Total
Dictionary	11.4	67.4	2781.4	819.4	822.5	4867.8
Normal	N/A	75.0	2739.7	771.1	755.0	4341.0

Table 4.5: Time required to create partitions in seconds for SF10.

	VP	Extvp	Dictionary	Total
Dict.	40	61	5.7	106.7
Normal	129	156	N/A	285

Table 4.6: Disk usage of partitions in MB for SF10.

Scale Factor 100

In Table 4.7 and Table 4.8 we see the time and disk requirements for SF100 respectively. We still only use a very small amount of time to create the dictionary, and VP is still faster for the dictionary encoded version. We now spend 81.6 minutes in total against the 91.7 of the normal encoding. This is a difference of 10 minutes which is rather significant when the dictionary encoded version was around 8 minutes behind for SF10. This could indicate that once the data set becomes large enough, the dictionary encoded version will start to take over. The disk usage for dictionary encoding is 727 MB against the 1368 MB of normal encoding, which resolves to 53.14%. Saving the dictionary now takes up 7.57% of the disk space usage, and we therefore have an indication that for higher data sets, a larger amount of strings will have to be saved to the dictionary. The fact that the dictionary continues to grow could mean that it would be beneficial to find a more efficient way of storing it in the long run.

	Dict.	VP	SS	OS	SO	Total
Dictionary	17.3	81.9	3104.9	854.5	838.9	4897.5
Normal	N/A	113.0	3203.5	1278.8	905.8	5501.1

Table 4.7: Time required for to make partitions in seconds for SF100.

	VP	Extvp	Dictionary	Total
Dict.	285	387	55	727
Normal	523	845	N/A	1368

Table 4.8: Disk usage of partitions in MB for SF100.

It is difficult to pinpoint what causes the negative performance for the dictionary encoding with SF10, as the disk consumption is much lower and most likely also the memory usage. We definitely see worse execution times when making the partitions using the dictionary encoding at SF10, but the improvement at SF100 makes the encoding seem promising to continue using when scaling to larger data sets than we used for this test. The reduced storage consumption is the main advantage in the context of data set conversion, and being able to reduce the sizes this significantly is great, especially when dealing with the massive storage overhead of ExtVP.

4.3.2 Query Response Times

In this section we will present the results of executing the queries presented in Section 4.1.4. The test is executed as a split test with the purpose of determining if our dictionary encoding improves the performance of SFRDF+ compared to Reuse. The results of the test are seen in Table 4.9, which shows the average execution times for each type of query for the two systems on SF1, SF10, and SF100. The query column states whether the execution is for the normal encoding or for the dictionary encoding, which both have object reuse enabled as mentioned in Section 4.2. The two categories are denoted by Reuse and Dict respectively.

If we take a look at the results for SF1 shown in Figure 4.1, we can see that the performance of most queries are comparable. However, some of the results, e.g. for C1, shows to be significantly slower for Dictionary than for Reuse, whereas for other queries, e.g. L4, the improvement have made them faster. In general, dictionary encoding is slower for most query types, which is also reflected in the average response time for SF1, where we have an average query response time for the normal encoding at 1609.2 ms while it is 1687.9 ms for the dictionary encoding. Contrary to what we would expect, the average response time is slightly higher for the dictionary encoded version. A diffence in speed of 78.7 ms for every query is a large enough gap to indicate that the dictionary encoding is slower than the normal encoding. However, we will have to validate the efficiency of the encoding with the larger data sets. Note that query S5 was unable to run for SF1, because no statistics are generated for one of the necessary joins, which therefore makes an invalid query. The thing to note is though, that missing statistics means that the join is invalid, and therefore the query would return 0 rows anyway.

Scale	Query	C1	C2	C3	S1	S2	S3	S4	S5	S6	S7
1	**Reuse**	2044.8	2033.2	1531.3	2770.3	1363.4	1248.6	1289.0	0.0	1123.2	1067.2
	Dict	2522.6	2254.4	1585.5	2702.4	1508.2	1304.3	1408.3	0.0	1133.7	1059.3
10	**Reuse**	2325.9	2904.6	1761.3	2486.8	1471.5	1172.0	1350.6	1312.4	1185.3	1129.1
	Dict	2662.2	2667.7	1942.8	2498.3	1535.7	1198.2	1424.1	1420.9	1242.8	1141.8
100	**Reuse**	2711.1	2986.4	2011.7	2730.6	1283.5	1191.8	1342.0	1277.5	1071.8	1064.5
	Dict	2169.8	2841.4	1893.2	2905.8	1285.3	1197.0	1391.4	1381.3	1021.9	999.9
Scale	**Query**	**F1**	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**
1	**Reuse**	1437.8	2364.9	2008.9	2876.7	1958.3	1091.1	1293.8	975.1	1078.5	1017.9
	Dict	1733.8	2558.8	2086.4	2827.4	2028.2	1182.6	1338.5	922.0	889.3	1023.8
10	**Reuse**	1995.1	2292.1	1770.0	2853.5	2035.8	1118.0	1299.5	1121.6	929.5	1177.7
	Dict	2238.4	2387.3	1973.5	3090.2	2213.7	1109.7	1274.9	1003.0	897.2	1176.6
100	**Reuse**	1954.8	2371.5	2156.0	2730.1	2244.5	1076.8	1251.2	923.4	901.0	1044.4
	Dict	2262.1	2310.6	2055.1	2904.5	2204.5	1096.1	1304.8	894.1	865.6	1022.3

Table 4.9: Table with query results for dictionary and normal encoding for SF1, SF10, and SF100.

When considering the query times for SF10, we can see that, in some cases, e.g. S1 and F2, the query speeds are significantly faster than the results for SF1. This is peculiar, but there are some reasons why this might be the case. The first observation is that the ExtVP schema makes it possible to store more efficient tables that were not stored for smaller data sizes due to the tables falling within the allowed selectivity factor upper bound (see [14] or [16] for clarification regarding this upper bound). Another observation is that SFRDF is running on a JVM, which means that we have close to zero control over the garbage collector. If the garbage collector decided during many of the query executions of SF1 (although heavier garbage collection is strange with a smaller data size), it can severely impact the execution time in a negative manner. Another observation is that our cluster computer does not allow us to have exclusive access to switches, which therefore means, that if someone else is using the switch when we are, we can lose some very valuable milliseconds during this time.

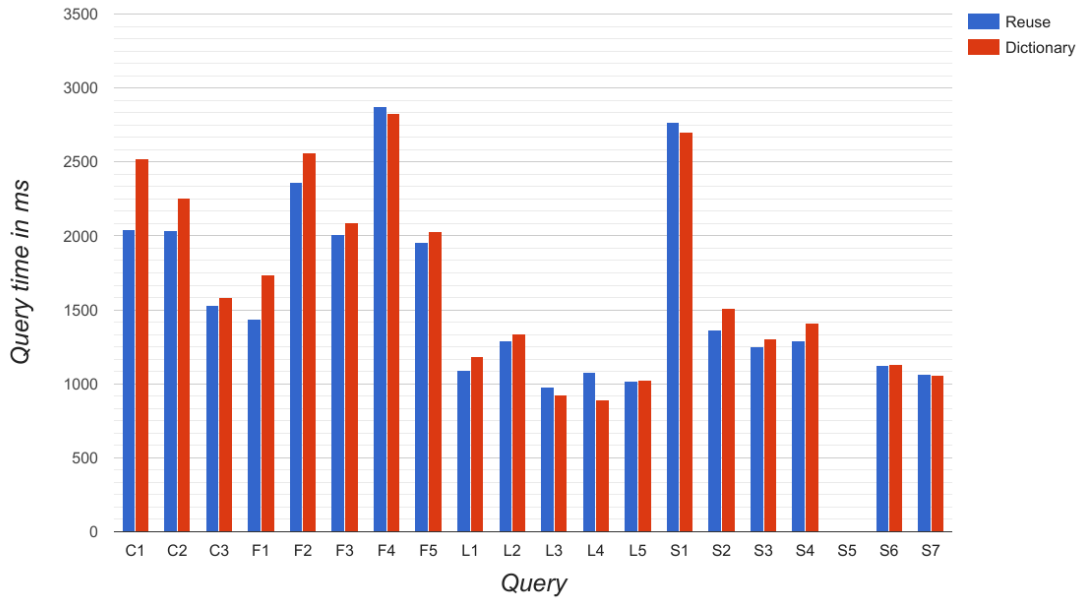


Figure 4.1: Query times for SF1 using dictionary and normal encoding

Looking at the visual representation of the results in Figure 4.2 it is still obvious to see that the general tendency is for the dictionary encoded version to be slower. This is also reflected in the average query response time of 1684.6 ms for the normal encoding and 1755.0 ms for the dictionary encoding. We can here see that the normal encoding actually has an average query speed that is 70.4 milliseconds faster than the dictionary encoding, which again suggests that the dictionary encoded version is slower than the other approach. As before this is not what we expected.

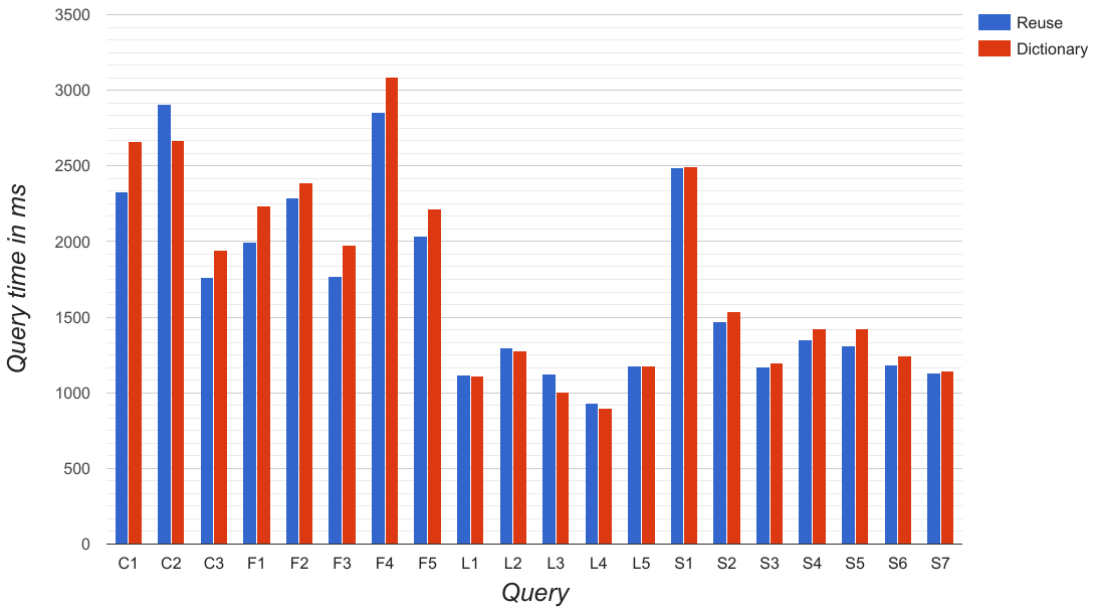


Figure 4.2: Query times for SF10 using dictionary and normal encoding

As mentioned in Section 4.1.4 we experienced a lot of fluctuations during experiments. These could also have influenced the result, even though we tried to mitigate the issue by removing outliers, and made it look like the dictionary encoded version performs worse than the standard one. We experienced that the average can actually fluctuate as much as 300 to 400 milliseconds between each execution of a query. Another possibility is that the underlying Flink framework does not operate better on integers than it does on strings. There are some factors that make this a definite possibility, such as making it easier to make good hash tables with less risk of collisions. This is a possible problem, but we do not think it is responsible for the extent of the results that we see.

For SF100, the dictionary encoding now has the fastest average execution speed of 1700.3 ms versus 1716.2 ms for the normal encoding. When looking at the diagram over the query response for SF100 in Figure 4.3 we see a general tendency of the query performance being close to the same, but with single queries, e.g. C1, being significantly faster for dictionary encoding. We cannot tell for sure why dictionary now has the advantage compared to SF1 and SF10, as we would expect the advantage to be consistent for every execution for whichever mode functions better. This could indicate that there is some significant overhead related to storing the dictionary at lower data sizes, which then disappears when the data set becomes larger. Additionally, the dictionary encoding for SF100 actually has an average execution speed that is 38.8 ms faster than the SF10 version, which could indicate that dictionary encoding scales really well with larger data sets as was also indicated by the tests from Section 4.3.1, whereas the normal encoding consistently loses performance. There does not seem to be some kind of clear consistency between which queries that perform worse for either of the encoding types, so we will have to get better control over the environment or scale to even larger data sets before we can make a clear differentiation between the two types.

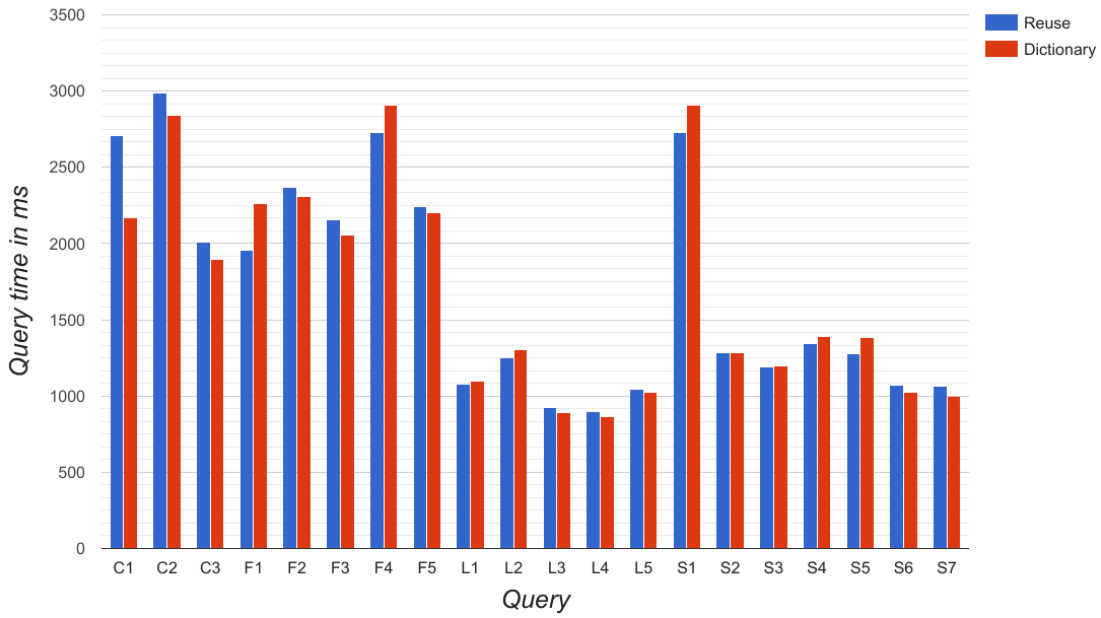


Figure 4.3: Query times for SF100 using dictionary and normal encoding

The goal of our system is to possibly scale to data sets spanning billions of triples, so it is very important that the system we use scales well with the size of the data. Based on the results of our tests, we deem that the dictionary encoded version shows the most promising results, as it already starts showing better results at SF100, which corresponds

to mere 10 million triples, both in terms of making the partitioning scheme and querying the system later on. Additionally, the size on disk is much lower for the dictionary encoded version, but possibly also the memory usage which we did not measure during the execution. We therefore think it much more feasible to scale to very large data sets when using the dictionary encoding. We will therefore base the remainder of our improvements on the dictionary encoded version of our system, and not present any findings for the normally encoded version in the rest of the report.

4.4 Greedy Algorithm

In this section, we cover our test of the greedy algorithm from Section 3.2.3. The test of the greedy algorithm is done against the normal dictionary encoded version that uses regular SQL and left deep plans. As mentioned in Section 3.2.3 one of the key benefits of using a greedy algorithm is the speed with which it can find a query plan. In Table 4.10 we present the milliseconds needed to create the query plans during one of our experiments. Here we can clearly see that the time required for making the most complex plans is extremely low compared to DPCCP covered in Section 3.2.1, as even the slowest plan is created within 19 ms.

C1	C2	C3	S1	S2	S3	S4	S5	S6	S7
18.95	11.20	8.15	18.41	7.35	6.97	7.82	5.34	6.74	5.20
F1	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**
8.35	13.38	9.62	14.93	10.31	3.51	5.29	3.50	3.29	4.39

Table 4.10: Time consumed on generating the plans for greedy SF1000 in milliseconds.

As with the evaluation of the dictionary encoding in Section 4.3 we execute the tests with the format explained in Section 4.1.4. The results of these tests can be found in Table 4.11. Notice that the greedy algorithm seems to scale really well with the size of the data set, as its average response times generally are very similar for each of the scale factors. The table also reveals the general tendency that the execution plan created by our greedy algorithm outperforms the Flink built-in linear execution plan in close to every case.

Scale	Query	C1	C2	C3	S1	S2	S3	S4	S5	S6	S7
1	**Dictionary**	2522.6	2254.4	1585.5	2702.4	1508.2	1304.3	1408.3	0.0	1133.7	1059.3
	Greedy	2024.2	2581.5	1693.2	2378.2	1300.1	1214.7	1285.7	0.0	1051.7	1038.0
10	**Dictionary**	2662.2	2667.7	1942.8	2498.3	1535.7	1198.2	1424.1	1420.9	1242.8	1141.8
	Greedy	2041.5	2556.1	1367.5	2519.6	1337.3	1269.4	1318.0	1248.7	1019.0	1117.3
100	**Dictionary**	2169.8	2841.4	1893.2	2905.8	1285.3	1197.0	1391.4	1381.3	1021.9	999.9
	Greedy	1961.5	2781.6	1634.4	2429.1	1321.5	1213.5	1286.8	1317.9	1010.8	1066.1
Scale	**Query**	**F1**	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**
1	**Dictionary**	1733.8	2558.8	2086.4	2827.4	2028.2	1182.6	1338.5	922.0	889.3	1023.8
	Greedy	1703.5	2264.0	1806.1	2280.1	1806.2	1033.9	1090.2	1054.8	916.9	1150.7
10	**Dictionary**	2238.4	2387.3	1973.5	3090.2	2213.7	1109.7	1274.9	1003.0	897.2	1176.6
	Greedy	1982.5	2271.0	1830.0	2597.5	1804.7	1068.7	1128.5	899.3	882.4	1011.4
100	**Dictionary**	2262.1	2310.6	2055.1	2904.5	2204.5	1096.1	1304.8	894.1	865.6	1022.3
	Greedy	1702.7	2336.2	1877.2	2481.0	2008.0	1049.7	1083.6	870.1	867.9	1002.0

Table 4.11: Table with query results for dictionary and greedy execution for SF1, SF10, and SF100.

While the average time for the dictionary encoding fluctuates quite a bit for the various scale factors, first by rising 67.1 ms and then afterwards falling 54.7 ms, the average time for the greedy approach starts out significantly lower and increases extremely slowly. The greedy approach starts off by saving approximately 7.48% of the execution

time, when going from 1687.9 to 1561.8 ms for SF1. This increases to an advantage of approximately 10.91% when considering SF10 where we compare 1755.0 ms to 1563.5 ms for the dictionary approach and the greedy approach respectively. When looking at the numbers for SF100 the advantage then falls to 7.95% with 1700.3 ms against 1565.1 ms for SF100, due to the, in Section 4.3, mentioned drop in average execution time for that SF. When taking a look at the visual representation of this SF the results it is also clear to see that almost every query performs as well or even better than the left deep approach.

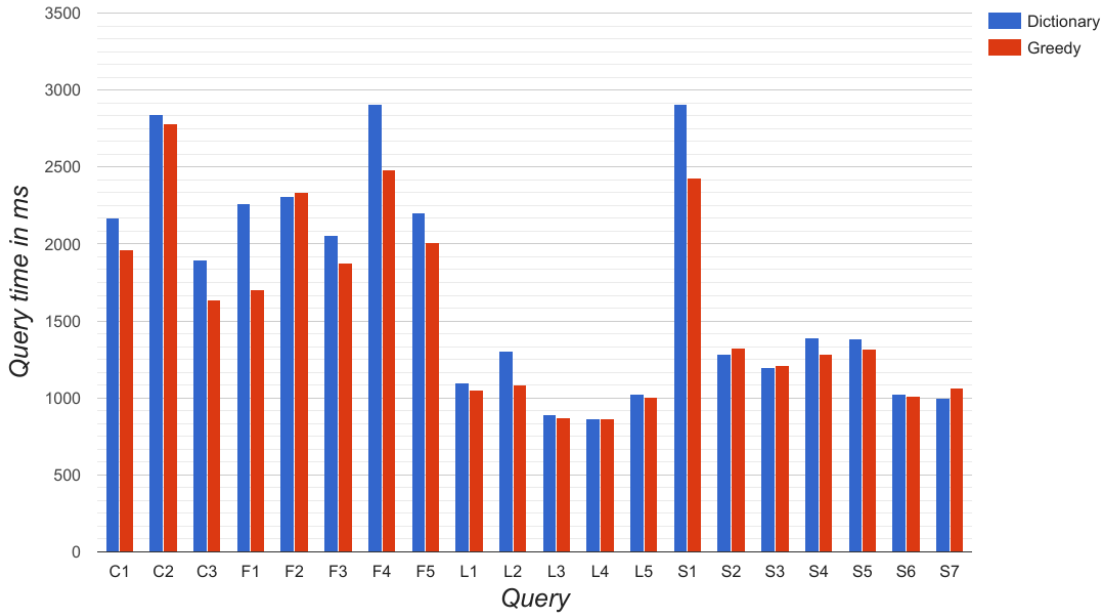


Figure 4.4: Query times for SF100 using normal left deep and greedy approach.

Notice that especially for the complex and longer running queries the greedy query plans perform much better, which clearly indicates that the join order is important in these cases. The diagrams over the execution results for SF1 and SF10 can be found in Appendix B.

The greedy approach is, based on the aforementioned results, better than the standard linear execution plan, which means that effective (possibly bushy) execution plans can be made in very short time for SPARQL queries. Additionally, considering that the query times for the greedy algorithm also includes the time it takes to generate its execution plan/join order, while the execution plan for the dictionary encoding is made by the S2RDF [16] query translator when making the SQL script, one could say that the greedy algorithm is even better, as additional time *should* be added to the dictionary measurements. Of course, there is some work that both of the systems have to do, which are part of the query times already, such as when the execution plan is translated to a plan internal to Flink.

4.4.1 Query Bushiness

As mentioned in Section 3.2.4, the fact that our greedy algorithm is based on the cost function from [6] meant, that we could not guarantee that it would actually make a bushy plan. This was not satisfactory for us, so we decided to modify the cost function in order to make it prefer bushy plans. As previously mentioned, we did this by making a difference when returning the cost for a node in the logical plan, where a combined node would have its cost multiplied by some constant, such that combined nodes are

more expensive to use in the plan than individual nodes would be. We call this modified version of the system *greedy+*. The difference between how these plans look can be seen in figures 4.5, 4.6, and 4.7, which respectively show the logical plans for query F4-1 (see Appendix D) as a standard left-deep plan suggested by the S2RDF QueryTranslator; the plan generated by the standard greedy algorithm; and the plan generated by the bushy-prefering greedy+ algorithm. The nodes in the figures correspond to specific VP or ExtVP tables of the format `predicate1_predicate2_relationType`, for example `foaf:homepage_sorg:contentSize_SS`. The nodes are translated as seen in Table 4.12.

Symbol	Actual Table Name
A	`sorg:language_VP`
B	`foaf:homepage_sorg:contentSize_SS`
C	`og:tag_sorg:contentSize_SS`
D	`sorg:description_sorg:contentSize_SS`
E	`sorg:contentSize_VP`
F	`sorg:url_VP`
G	`wsdbm:hits_VP`
H	`wsdbm:likes_sorg:contentSize_OS`
I	`gr:includes_sorg:contentSize_OS`

Table 4.12: Overview of the table symbols used in the join trees.

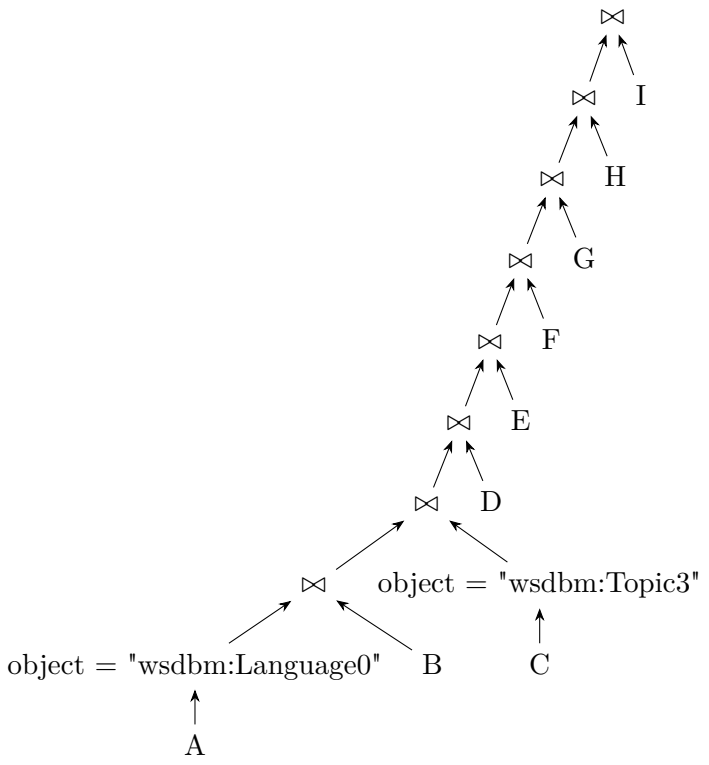


Figure 4.5: Logical plan for execution through Flink's Table API

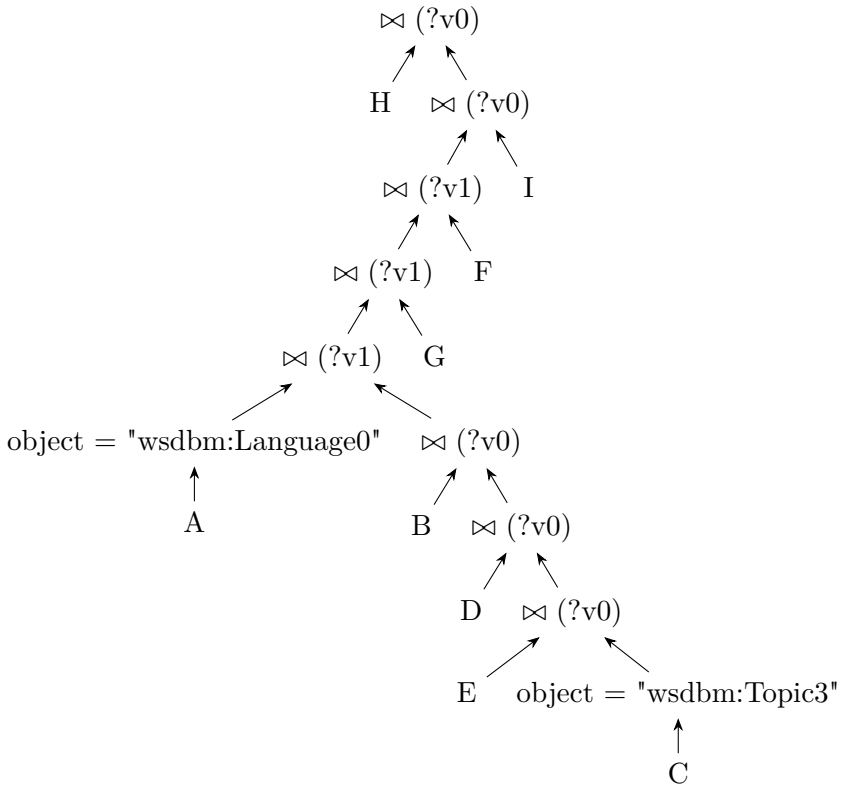


Figure 4.6: Logical plan for execution through the greedy algoritm

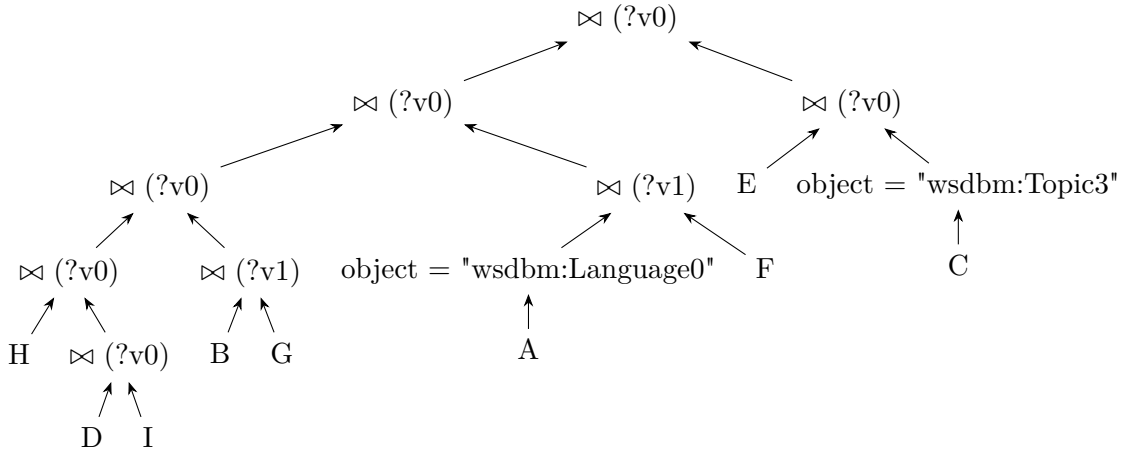


Figure 4.7: Logical plan for execution through the greedy+ algorithm

It should be easy to see that the plan from Figure 4.7 is significantly more bushy, and thereby flatter and more suited for parallelism than the other versions. We additionally ran the same experiments as in the previous section on the new greedy+ version, the results of which can be seen in Table 4.13. As can be seen, deterring from the heuristics regarding the size of the tables in favor of a flatter and bushier plan is not necessarily a good thing, e.g. L2 for SF100 where the average response time for the greedy+ approach is almost 800 milliseconds worse than the original greedy approach. On the other hand we have queries such as F4 for SF100, where the greedy+ approach is almost 1 second faster on average. It is therefore hard to find a general tendency in the results. However, taking a look at the average response times for the different scale factors we see an advantage in average execution time of 1484.3 ms for greedy+ versus the 1561.8 ms of the greedy algorithm. Greedy+ comes out ahead at scale factor 10 as well, where the normal greedy algorithm had an average execution time of 1563.5 ms while greedy+ has a time of 1499.7 ms. When using SF100, the average speed takes a small hit, compared to the difference from SF1 and SF10, but we still have comparable execution speeds between greedy and greedy+. The average for greedy+ is 1586.0 ms compared to the 1565.1 ms for the normal greedy algorithm, which means that greedy now has the fastest response time of the two, which might indicate that the greedy approach scales better than greedy+. Visual representations of all the greedy versus greedy+ response times can be found in Appendix C.

Scale	Query	C1	C2	C3	S1	S2	S3	S4	S5	S6	S7
1	**Greedy**	2024.2	2581.5	1693.2	2378.2	1300.1	1214.7	1285.7	0.0	1051.7	1038.0
	Greedy+	1777.2	2235.3	1422.9	2225.3	1344.0	1170.8	1310.5	0.0	1000.1	1113.6
10	**Greedy**	2041.5	2556.1	1367.5	2519.6	1337.3	1269.4	1318.0	1248.7	1019.0	1117.3
	Greedy+	2025.2	2199.2	1505.4	2143.3	1414.0	1355.2	1311.5	1406.1	1088.7	1084.9
100	**Greedy**	1961.5	2781.6	1634.4	2429.1	1321.5	1213.5	1286.8	1317.9	1010.8	1066.1
	Greedy+	2341.7	1709.3	1387.4	2726.2	641.3	1624.5	884.4	639.0	1151.2	1534.1
Scale	**Query**	**F1**	**F2**	**F3**	**F4**	**F5**	**L1**	**L2**	**L3**	**L4**	**L5**
1	**Greedy**	1703.5	2264.0	1806.1	2280.1	1806.2	1033.9	1090.2	1054.8	916.9	1150.7
	Greedy+	1673.2	1988.7	1911.7	2309.8	1705.0	1033.2	1056.1	881.3	922.6	1121.0
10	**Greedy**	1982.5	2271.0	1830.0	2597.5	1804.7	1068.7	1128.5	899.3	882.4	1011.4
	Greedy+	1695.8	1934.6	1825.6	2258.0	1893.4	1013.0	1080.8	890.1	858.3	1011.3
100	**Greedy**	1702.7	2336.2	1877.2	2481.0	2008.0	1049.7	1083.6	870.1	867.9	1002.0
	Greedy+	1421.9	2322.6	2322.4	1571.5	2459.5	1353.3	1820.7	1207.3	1326.1	1275.4

Table 4.13: Table with query results for greedy and greedy+ execution for SF1, SF10, and SF100.

We had hoped to get a unanimous result that could clearly tell us which of the two results would scale the best, but it does not seem like there is a clear distinction based on the presented query times. It is very possible that greedy+ would work much better than greedy if we had used a cost function that would prefer to place "good" combined nodes, i.e. the joins where an input data set has filters, at the bottom of subtrees in the query plan, but this is no guarantee with the cost function we used for this test. With the version we used for this test, the algorithm will prefer to create the combined nodes that include the filters, but it will not necessarily place them at the bottom of any subtree, which is quite a big flaw. We would have liked to make this change, but we did not have time to implement and test such a version of the system before the deadline, so for now we can only present greedy+ as an algorithm that is on par with greedy when using rather small data sets and 3 nodes in the cluster. Due to the difficulty of singling out the system that works the best, we have decided to continue the evaluation of both systems with respect to data size and amount of nodes used.

4.5 Testing Scalability

There are two aspects of testing the scalability of our system as we see it, namely testing how the system scales with different data set sizes and testing how the performance improves when we introduce more workers. Testing how the system scales with different data sizes have already been partly done as we have tested for SF1, SF10, and SF100, however, we will here use SF1000, SF2000, SF3000, and SF4000 and show sizes correlation to average response times for all the queries.

4.5.1 data set Size

For the purpose of testing how well the different approaches, i.e. Greedy and Greedy+, scales with increasing amounts of data, i.e. SF1000, SF2000, SF3000, and SF4000, we have conducted a test to find out how the response times of these approaches will react when increasing data sizes. As seen in Figure 4.8 we get a quite unexpected result.

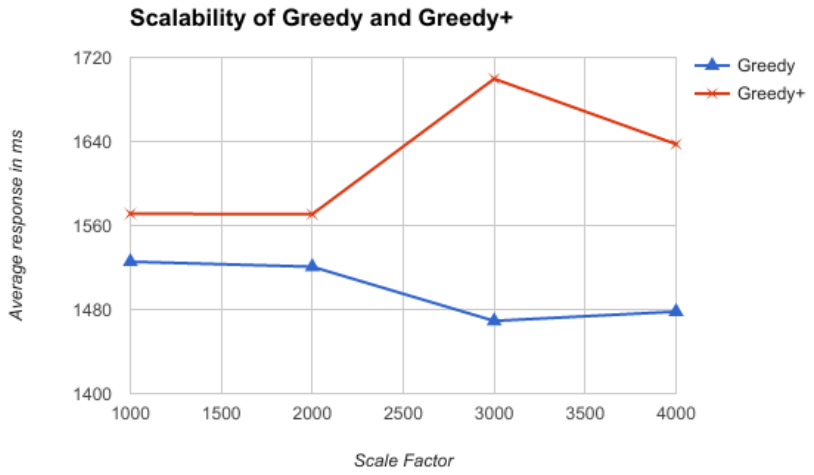


Figure 4.8: The graph shows how our system scales with increasing data set sizes and minimal configuration.

As the graph in the figure illustrates the numbers and the sizes of the data sets does not seem to correlate entirely. In fact we have that the average execution time of the non-bushy version algorithm seems to drop as we increase the data sizes. If we look at the bushy version the average response from SF2000 to SF3000 actually increases as expected, but then drops from SF3000 to SF4000. Notice that the y-axis is only spanning over 320 ms which makes the result seem more impressive than it is. For the greedy approach we have a reduction of 47 ms in average from the highest time (SF1000) to the lowest response time (SF3000). This may just be due to interference from other users on the cluster; the JVM garbage collection starting at the wrong time; or something else entirely. When looking at the to approaches in comparison, we see a result similar to what we predicted earlier, namely that the greedy approach seems to scale better than the greedy+ approach. We still cannot say for certain that this is the case. However, we can be fairly certain that the system scales really well with both approaches in regards to data size, as we at most see a slight increase in execution times when increasing the scale factor.

4.5.2 Number of Nodes

For the purpose of testing how the systems scales when increasing the amount of workers, we have chosen to test the best approach that we have found so far, namely Greedy. However, our intuition tells us that when scaling the amount of nodes, flatter plans might perform better. Therefore, we have chosen to test the Greedy+ approach as well. We have tested the scalability on 3, 5, 7, and 9 nodes for SF1000. It is here important to note that we always have one node dedicated as the master, i.e. the `JobManager`, and the rest of the nodes each run two instances of the workers, i.e. the `TaskManagers`, as we take advantage of the NUMA architecture of the nodes in the system. To exemplify this means that when we run the tests for 9 nodes, we have 1 master and 16 workers. The results of this test are shown in Figure 4.9, which shows a harsh truth about our system.

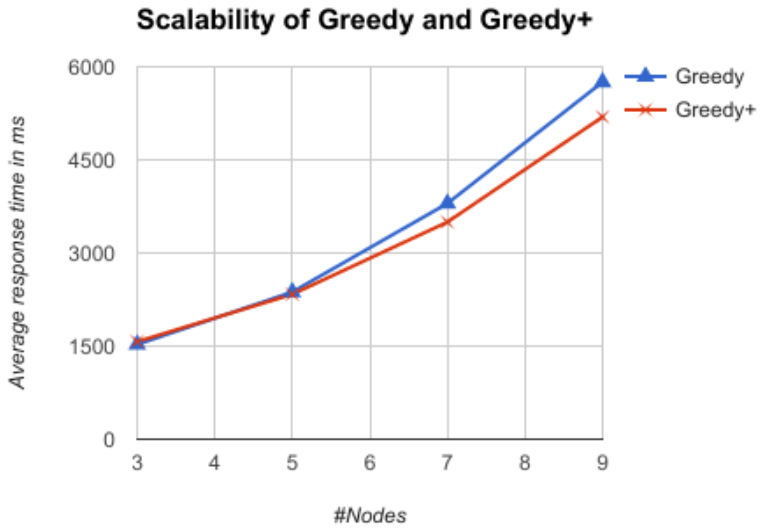


Figure 4.9: The graph shows how our system scales with increasing amount of nodes and minimal configuration.

As seen in Figure 4.9 our system simply does not scale well with increasing amounts of node. In fact mean execution time actually increases when utilizing more nodes. We cannot say for certain what is the cause of this, but it should be said that we for the purpose of these test have spend minimal time on fine-tuning the Flink configuration, which as already covered in Section 4.1.3 is very important for the performance of the Flink framework. Another aspect worth considering in this regard is the communication overhead that might be present in introducing more nodes. We do however not think that this is responsible for the entirety of the problem. One thing that we might take away from this experiment is that as far as we can tell, the Flink framework is not easy to scale, i.e. it requires a lot of customization when increasing datasize and amount of nodes, and the framework might be better suited for long running tasks, i.e. its primary purpose, streaming, where it might be able to achieve a high throughput without the same amount of overhead.

5 | Conclusion

RDF data sets are becoming increasingly larger due to techniques such as Web Scrabing, which is why the area of SPARQL query processing is an interesting area of research. We have as many others created a distributed approach within the area, called SFRDF[14], which almost were as good as state of the art systems at the time. Therefore, we decided to introduce several improvements, i.e. dictionary encoding and join order optimizations, to a new and improved version of our system, namely SFRDF+.

The first major improvement that we introduced to the system was the dictionary encoding, where we by replacing strings with integers, would reduce the memory footprint and disk usage of the system. According to our calculations we should be able to reduce the space consumption by an order of magnitude on average. However, as we did not end up utilizing a binary format for storing the integers, we did not have that big of an reduction. Even though we did not see the reduction that was theoretically possible we still reduced the space consumption significantly. In the worst case, i.e. for the largest data size, we still cut the space requirements in half. As we are saving this much space, it would make sense that the IO of the system and the memory footprint throughout the system should be smaller. This might very well be the case, but we did not see an improvement in execution times on data sets on 1 million triples or below. For the largest of the data sets we tested against we did however see an slight improvement, which indicates that it scales better. Therefore, we decided to built on top of the encoding for the remainder of the implementation of the SFRDF+ system.

The second major change we introduced to SFRDF+, compared to the original system, was query plan optimization algorithms. For this purpose, we introduced different algorithms from database literature, i.e. DPCCP[10] and a greedy approach, and from more recent research from within the area of RDF and SPARQL processing, i.e. CliqueSquare[5]. The DPCCP approach, which was based on dynamic programming, showed sinister results in its recursive form. Therefore, we modified the algorithm to be iterative instead, which significantly reduced the execution time of the algorithm. This was however still not enough to make it feasible to utilize this approach in SPARQL queries as these queries commonly involve more relations, which are more tightly coupled, than what is usual in standard relational queries. This made the execution times too long.

We instead started looking at approaches within the area of RDF and SPARQL processing, as we were guaranteed that these to be usable for RDF and SPARQL purposes. The plans from CliqueSquare were generated in a short amount of time, but did not fit our system, as they involved n-ary joins, and our framework, Flink, only supports binary joins. Therefore, we moved on to a greedy approach, in which we would greedily select the best joins according to a cost function from [6]. We did however learn that the plans produced were not necessarily bushy as we expected, hence a modification of the cost function was needed. We tested the algorithm with both versions of our cost function and learned that they in most cases were comparable in performance. On some queries we saw the bushy preferring version pull ahead, and in other queries the opposite. Our initial tests indicated that the bushy version was better at lower scale factors, but that the greedy version would perform better when used on larger data sets.

We therefore tested the scalability for both types in regards to data sets ranging from 100 million to 400 million triples. These tests revealed our previous prediction, i.e. that greedy scales better. The test also revealed that the average response time actually would drop for larger data sets. Even though the drop was not significant enough to rule out outside influence, we still are able to say that the system scales really well in regards to the size of the input data. Besides testing how the variations scaled with data set sizes, we also tested how they would scale when introducing more compute nodes. Our results were in this case not as promising as with the data set sizes, as it actually showed an increase in the average query response time. We attribute this to missing fine-tuning of the parameters as these have tremendous effect on performance in the Flink framework.

To conclude, we find that Flink being a JVM based framework has certain disadvantages when it comes to making reproducible scientific results as garbage collection seemed to have massive impact on query performance. Furthermore, we have during our work learned that relational approaches, e.g. DPCCP [10], are not always applicable in the context of RDF and SPARQL. The reason for this is that the structure of queries in SPARQL often involve more relations and are more connected than what is common in SQL. We have also learned that making a plan bushier does not necessarily improve performance in Flink as we saw in our evaluation of greedy versus greedy+. Our system, SFRDF+, does however scale well with respect to data size and can handle data sets of at least 400 million triples in size. When increasing the amount of nodes, we did unfortunately see a decrease in performance, due to Flink being a framework which requires extensive configuration. All in all we can conclude that we have introduced improvements to the system both in terms of how well it scales with data size and how we have reduced the average query response times.

6 | Future Work

In this chapter we will describe some of the thoughts and ideas that have come up during our work and we would have liked to work with if we had more time. The thoughts are mainly about how we could have utilized some of the optimization algorithms in our work, but there are also considerations regarding the framework that we use.

6.1 Utilizing DPCCP

As mentioned in Section 3.2.1 the DPCCP algorithm took too long to generate plans for queries with many relations and certain shapes. This was enough for us to change focus and find another algorithm which would be good for every type of query. However, it still could be interesting to see how much better the plans produced by DPCCP are compared to our more efficient greedy approach. This would of course highly depend on the cost function, which should take many things into consideration, such as how parallelism affects joins in a bushy tree, i.e. the children of the join might be computed in parallel and the cost function therefore should assume that the cost of calculating the children is $max(cost(child_1), cost(child_2))$. We might even be able to implement the DPCCP algorithm in a functional system, by firstly doing some static analyzes of the query in order to see whether or not it is worth it to use the algorithm or use another algorithm, such as our greedy approach. The static analysis should, based on the number of relations involved and how connected these are, determine if it is worth to use DPCCP or not. This approach might also open up for some other algorithms from relational database research that might work on the less complex SPARQL queries but not for the complex ones.

6.2 Utilizing CliqueSquare

The CliqueSquare [5] algorithm covered in Section 3.2.2 was as mentioned not able to produce binary bushy plans for star queries or queries containing star query structures. This made the algorithm unusable for our purposes as the Flink framework does not support n-ary joins. We did however consider several options in order to resolve this issue. One of which was to implement n-ary joins ourselves in the Flink framework. We did unfortunately arrive at the conclusion that this would be too time consuming for our project, and we instead decided to focus on other potential solutions. These other solutions included one, where we would utilize our greedy approach in order to translate the n-ary joins into binary ones, but still sustaining the flattened structure of the plan that CliqueSquare introduced. This would also entice other benefits that the Flink system might be able to utilize. The n-ary joins would all be over the same variable, and Flink might therefore be able to pipeline the joins in a more efficient manner. Due to limited time we were not able to make our implementation of this approach run without errors. Hence, we decided to omit it from the report.

6.3 Finetuning Parameters

As shown in Section 4.5.2 our system is not scalable. However, as we mentioned this might be due to the fact that the parameters should be fine-tuned. We cannot fully conclude that the Flink framework is not that scalable in our context before we have

customized the parameters for the amount of nodes involved in each scalability test. As covered in Section 4.1.3 configuring parameters for Flink yielded great improvements, but it was a very time consuming task, and we therefore chose not to pursue the fine-tuning in the scalability test any further. Given more time fine-tuning the parameters had been ideal, as we then firmly could conclude one thing or another. One thing is for certain, it is not easy to scale with Flink in our context.

6.4 Binary Format

As we covered in Section 2.3 all the data that we process have been dictionary encoded, and we do in this way save quite a bit of storage space. A way that the reduction of storage space might be further reduced was to utilize a binary file format instead of saving everything in the system as UTF8 encoded CSV-files. This should decrease the size of the data set significantly as we could represent a number that is 10-digits (10 bytes) long using 32-bit (4 bytes) or 64-bit (8 bytes) instead of the 10 bytes. This could decrease the size of the data set significantly, and might also allow us to utilize run-length encodings and similar in order to compress the data even further.

6.5 Alternative Framework

As we have mentioned throughout the report, the Flink framework has not been easy to work with. The amount of customization needed in order to scale to other data sizes and more nodes is too extensive, and it has been very time consuming in our project. We therefore think changing to another framework that is better suited for the workload might be beneficial. These systems could be other big data frameworks that are specialized around batch processing, which might be systems such as Spark. Another possibility might be to implement a native system as this would grant us more control over how we perform joins, do garbage collection, communicate with other machines, and the like.

6.6 Greedy Cost Function

The cost function covered in Section 3.2.4 was, as mentioned, inspired from [6]. We did however modify the cost function to prefer bushy plans by increasing the cost of combined nodes, which thereby made it beneficial to choose single relations before join subtrees. This did yield slightly better performance on smaller data sets, but our experiments showed that using the original cost function scaled better. We attribute this to the fact that it does not place nodes with filters at the bottom of large subtrees, which should be changed in a future version of our system. As mentioned in Section 3.2.4 we have also considered some additional changes to the cost function, such as utilizing the correct cardinalities for ExtVP tables, instead of basing them on the base VP table for the ExtVP. Additionally we might be able to using the additional information we have gained by having an ExtVP about the joins that will be made, in order to somehow improve the cost function. Our intuition is still that good bushy plans should utilize the resources of the cluster better and thereby also improve the query response times, and we believe that more work on a cost function that somehow prefers flatter plans would be beneficial, and it is therefore an area worth looking into.

6.7 Metrics

A possibility that we chose not to pursue during our implementation and evaluation of our system was the usage of metrics in Flink. These metrics include the throughput of each operator in an execution plan; time spent on garbage collection and amount of times it had run; time spent on IO and the like. We did not look further into getting performance metrics out of the Flink framework as it seemed to be fairly advanced and involve a lot of setup and we felt that enough time had been spent on configuring the framework (see Section 4.1.3) in comparison with time spent on actual improvements. However, having the metrics might have made the configuration easier, as well as give us a clear indication of where we should focus our efforts on improving the system. It might also give us some insights as to why the results seems to fluctuate as much as they do, which we currently attribute to the garbage collection in the JVM.

Bibliography

[1] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. "Scalable semantic web data management using vertical partitioning". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 411–422.

[2] Güne*C*s Alu*C*c, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. "Diversified stress testing of RDF data management systems". In: *International Semantic Web Conference*. Springer. 2014, pp. 197–212.

[3] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. "Optimization of parallel execution for multi-join queries". In: *IEEE Transactions on Knowledge and Data Engineering* 8.3 (1996), pp. 416–428.

[4] DeIC National HPC Centre. *Abacus 2.0 - Hardware*. Last seen: 19-04-2017. 2017. URL: https://abacus.deic.dk/setup/hardware.

[5] Fran*C*cois Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. "CliqueSquare: Flat plans for massively parallel RDF queries". In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 771–782.

[6] Stefan Hagedorn, Katja Hose, Kai-Uwe Sattler, and Jürgen Umbrich. "Resource planning for SPARQL query execution on data sharing platforms". In: *Proceedings of the 5th International Conference on Consuming Linked Data-Volume 1264*. CEUR-WS. org. 2014, pp. 49–60.

[7] IBM Corporation. *IBM Spectrum Scale*. Last seen: 19-04-2017. 2017. URL: https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=SP&infotype=PM&appname=STGE_DC_ZQ_USEN&htmlfid=DCD12374USEN&attachment=DCD12374USEN.PDF#loaded.

[8] Christoph Lameter. "Numa (non-uniform memory access): An overview". In: *Queue* 11.7 (2013), p. 40.

[9] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. *Top 500 - The List*. Last seen: 16-03-2017. 2016. URL: https://www.top500.org/list/2016/11/.

[10] Guido Moerkotte and Thomas Neumann. "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 930–941.

[11] Guido Moerkotte and Wolfgang Scheufele. "Constructing optimal bushy processing trees for join queries is NP-hard". In: *Technical reports* 96 (2004).

[12] Thomas Neumann and Gerhard Weikum. "The RDF-3X engine for scalable management of RDF data". In: *The VLDB Journal* 19.1 (2010), pp. 91–113.

[13] Jakob Nielsen. "Putting A/B Testing in Its Place". In: (2005). Last seen: 19-05-2017. URL: https://www.nngroup.com/articles/putting-ab-testing-in-its-place/#.

[14] Mathias Eriksen Otkjær and Jesper Clausen. *SFRDF: Distributed Processing of RDF Data using Apache Flink*. Tech. rep. 2017.

[15] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. "H_2RDF+: High-performance distributed joins over large-scale RDF graphs". In: *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, pp. 255–263.

[16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. "S2RDF: RDF Querying with SPARQL on Spark". In: *arXiv preprint arXiv:1512.07021* (2015).

[17] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. "Yago: A Core of Semantic Knowledge". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. ACM, 2007, pp. 697–706. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242667. URL: http://doi.acm.org/10.1145/1242572.1242667.

[18] Arun Swami and K Bernhard Schiefer. "On the estimation of join result sizes". In: *International Conference on Extending Database Technology*. Springer. 1994, pp. 287–300.

[19] Kevin Wilkinson. *Jena Property Table Implementation*. Tech. rep. 2006. URL: http://www.hpl.hp.com/techreports/2006/HPL-2006-140.pdf.

Part I

Appendicies

A | SFRDF versus Reuse

Test results of running SFRDF from [14] against the same program with the object reuse and optimization features activated can be seen in Table A.1.

Scale	Query	C1	C2	C3	S1	S2	S3	S4	S5	S6	S7
1	**Nml**	2022.9	2000.8	1506.1	2743.1	1392.6	1240.1	1298.9	0	1120.7	1071.3
	Reuse	2044.7	2033.1	1531.2	2770.2	1363.4	1248.6	1289.0	0	1123.2	1067.1
10	**Nml**	2344.1	2925.3	1727.3	2403.3	1461.4	1162.5	1356.2	1298.6	1171.4	1130.6
	Reuse	2325.9	2904.6	1761.2	2486.7	1471.4	1171.9	1350.6	1312.4	1185.2	1129.0
100	**Nml**	2523.8	3027.2	2250.1	2714.9	1252.7	1178	1338.4	1282.2	1071.5	1076.9
	Reuse	2711.0	2986.4	2011.6	2730.5	1283.4	1191.8	1342	1277.4	1071.7	1064.4

Scale	Query	F1	F2	F3	F4	F5	L1	L2	L3	L4	L5
1	**Nml**	1452.5	2319.5	2005.7	2857.3	2022.3	1082.0	1271.8	1042.1	1041.5	1007.0
	Reuse	1437.8	2364.9	2008.9	2876.7	1958.3	1091.1	1293.8	975.1	1078.5	1017.9
10	**Nml**	1953.9	2380.3	1818.9	2895.1	2090.7	1067.3	1302.3	1063.2	901.5	1158.3
	Reuse	1995.1	2292.1	1770.0	2853.5	2035.8	1118.0	1299.5	1121.6	929.5	1177.7
100	**Nml**	2022.5	2312.4	2117.8	2764.1	2236.0	1090.6	1271.5	908.0	916.1	1041.5
	Reuse	1954.8	2371.5	2156.0	2730.1	2244.5	1076.8	1251.2	923.4	901.0	1044.4

Table A.1: Table with query results for dictionary vs normal encoding for scale factors 1 through 100.

B | Dictionary versus Greedy

This appendix contains the diagrams showing the query results for the experiments of dictionary encoding with linear execution plans versus the greedy algorithm. This appendix only includes the figures that were not present in Section 4.4.

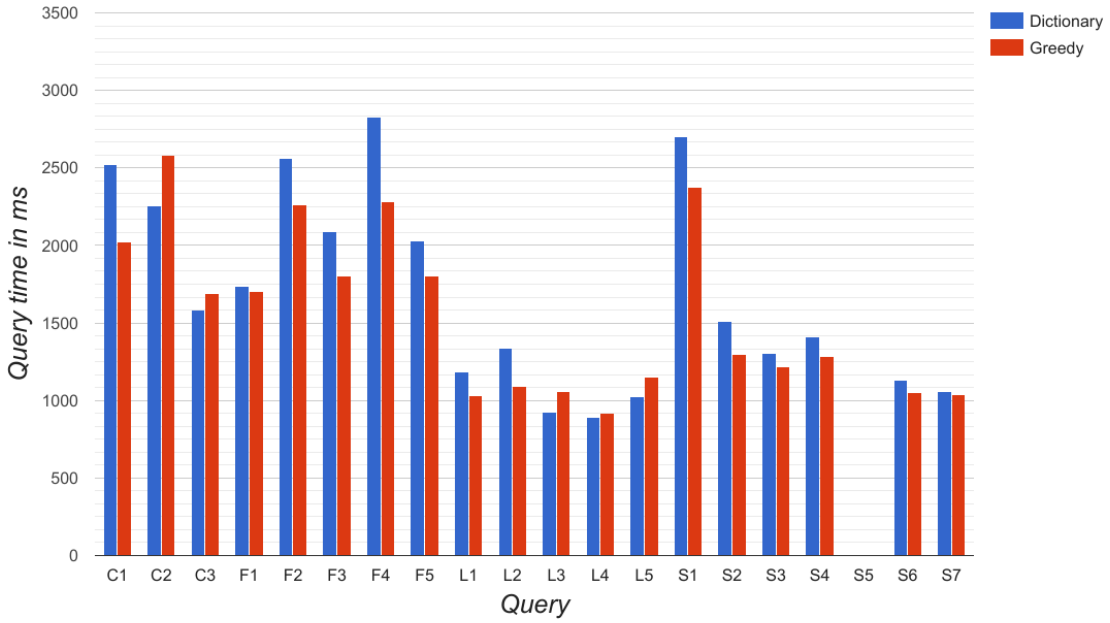


Figure B.1: Query results of dictionary versus greedy for SF1.

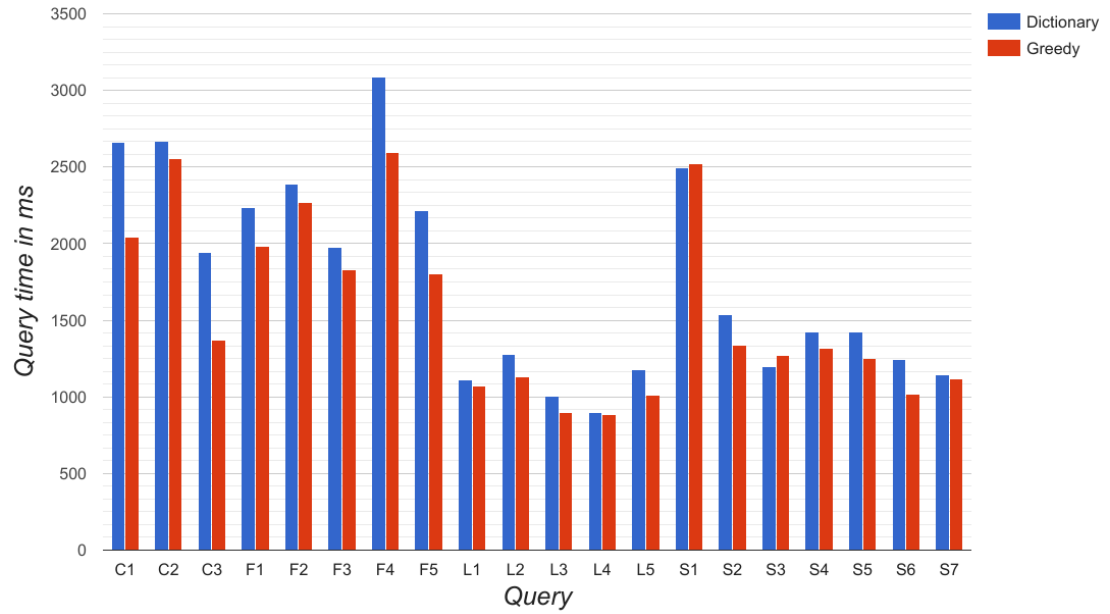


Figure B.2: Query results of dictionary versus greedy for SF10.

C | Greedy versus Greedy+

This appendix contains the diagrams showing the query results for the experiments of the greedy algorithm against the modified version of the same algorihm that prefers bushy plans, greedy+.

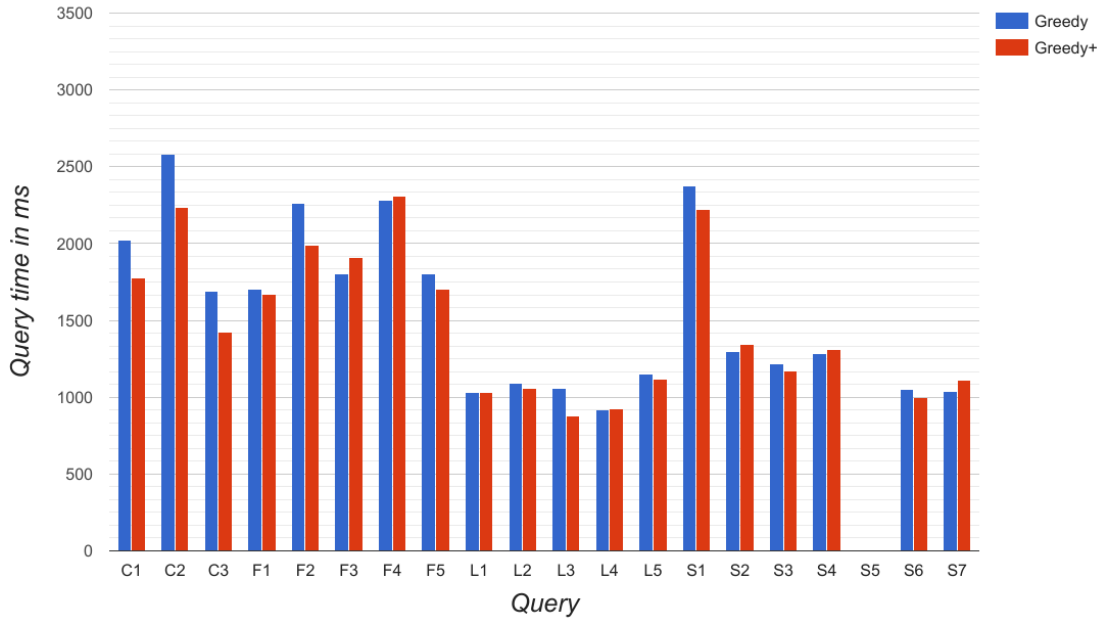


Figure C.1: Query results of greedy versus greedy+ for SF1.

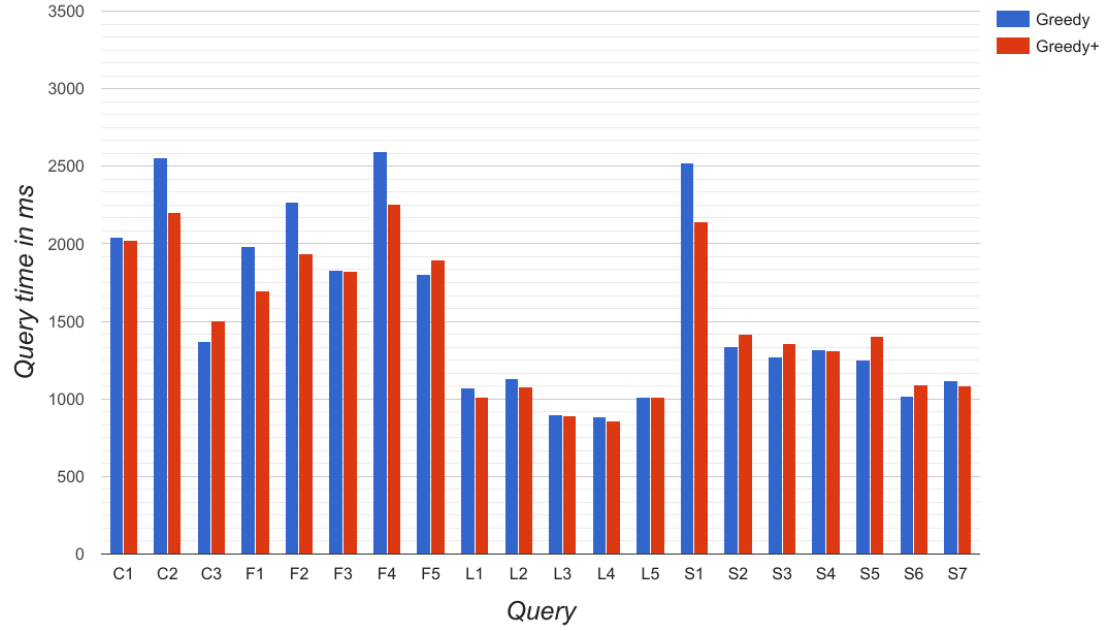


Figure C.2: Query results of greedy versus greedy+ for SF10.

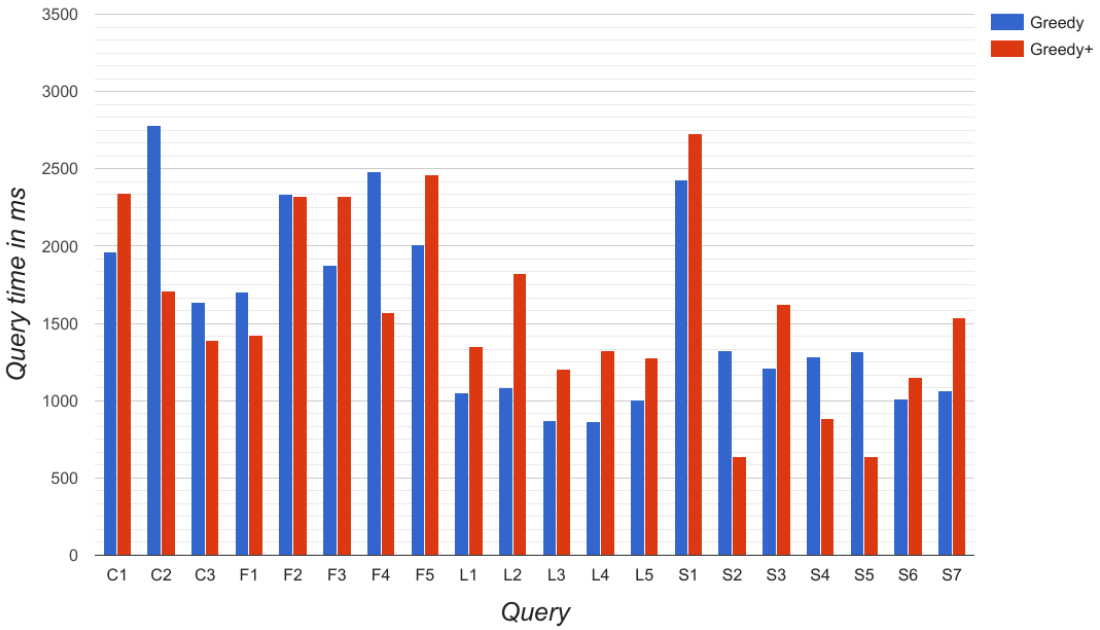


Figure C.3: Query results of greedy versus greedy+ for SF100.

D | WatDiv Queries

Linear Queries

```
SELECT ?v0 ?v2 ?v3 WHERE {
   ?v0   wsdbm:subscribes  %v1% .
   ?v2   sorg:caption   ?v3 .
   ?v0   wsdbm:likes ?v2 .
}
```

Code Snippet D.1: Linear 1 (L1)

```
SELECT ?v1 ?v2 WHERE {
   %v0%  gn:parentCountry  ?v1 .
   ?v2   wsdbm:likes wsdbm:Product0 .
   ?v2   sorg:nationality  ?v1 .
}
```

Code Snippet D.2: Linear 2 (L2)

```
SELECT ?v0 ?v1 WHERE {
   ?v0   wsdbm:likes ?v1 .
   ?v0   wsdbm:subscribes  %v2% .
}
```

Code Snippet D.3: Linear 3 (L3)

```
SELECT ?v0 ?v2 WHERE {
   ?v0   og:tag   %v1% .
   ?v0   sorg:caption   ?v2 .
}
```

Code Snippet D.4: Linear 4 (L4)

```
SELECT ?v0 ?v1 ?v3 WHERE {
   ?v0   sorg:jobTitle  ?v1 .
   %v2%  gn:parentCountry  ?v3 .
   ?v0   sorg:nationality  ?v3 .
}
```

Code Snippet D.5: Linear 5 (L5)

Star Queries

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE {
   ?v0   gr:includes ?v1 .
   %v2%  gr:offers   ?v0 .
   ?v0   gr:price ?v3 .
   ?v0   gr:serialNumber   ?v4 .
   ?v0   gr:validFrom   ?v5 .
   ?v0   gr:validThrough   ?v6 .
   ?v0   sorg:eligibleQuantity   ?v7 .
   ?v0   sorg:eligibleRegion  ?v8 .
   ?v0   sorg:priceValidUntil ?v9 .
}
```

Code Snippet D.6: Star 1 (S1)

```
SELECT ?v0 ?v1 ?v3 WHERE {
   ?v0   dc:Location ?v1 .
   ?v0   sorg:nationality  %v2% .
   ?v0   wsdbm:gender   ?v3 .
   ?v0   rdf:type wsdbm:Role2 .
}
```

Code Snippet D.7: Star 2 (S2)

```
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
   ?v0   rdf:type %v1% .
   ?v0   sorg:caption   ?v2 .
   ?v0   wsdbm:hasGenre ?v3 .
   ?v0   sorg:publisher ?v4 .
}
```

Code Snippet D.8: Star 3 (S3)

```
SELECT ?v0 ?v2 ?v3 WHERE {
   ?v0   foaf:age %v1% .
   ?v0   foaf:familyName   ?v2 .
   ?v3   mo:artist   ?v0 .
   ?v0   sorg:nationality  wsdbm:Country1 .
}
```

Code Snippet D.9: Star 4 (S4)

```
SELECT ?v0 ?v2 ?v3 WHERE {
   ?v0   rdf:type %v1% .
   ?v0   sorg:description  ?v2 .
   ?v0   sorg:keywords  ?v3 .
   ?v0   sorg:language  wsdbm:Language0 .
}
```

Code Snippet D.10: Star 5 (S5)

```
SELECT ?v0 ?v1 ?v2 WHERE {
   ?v0   mo:conductor   ?v1 .
   ?v0   rdf:type ?v2 .
   ?v0   wsdbm:hasGenre %v3% .
}
```

Code Snippet D.11: Star 6 (S6)

```
SELECT ?v0 ?v1 ?v2 WHERE {
   ?v0   rdf:type ?v1 .
   ?v0   sorg:text   ?v2 .
   %v3%  wsdbm:likes ?v0 .
}
```

Code Snippet D.12: Star 7 (S7)

Snowflake Queries

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
   ?v0   og:tag   %v1% .
   ?v0   rdf:type ?v2 .
   ?v3   sorg:trailer   ?v4 .
   ?v3   sorg:keywords  ?v5 .
   ?v3   wsdbm:hasGenre ?v0 .
   ?v3   rdf:type wsdbm:ProductCategory2 .
}
```

Code Snippet D.13: Snowflake 1 (F1)

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 WHERE {
   ?v0   foaf:homepage  ?v1 .
   ?v0   og:title ?v2 .
   ?v0   rdf:type ?v3 .
   ?v0   sorg:caption   ?v4 .
   ?v0   sorg:description  ?v5 .
   ?v1   sorg:url ?v6 .
   ?v1   wsdbm:hits  ?v7 .
   ?v0   wsdbm:hasGenre %v8% .
}
```

Code Snippet D.14: Snowflake 2 (F2)

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
   ?v0   sorg:contentRating   ?v1 .
   ?v0   sorg:contentSize  ?v2 .
   ?v0   wsdbm:hasGenre %v3% .
   ?v4   wsdbm:makesPurchase  ?v5 .
   ?v5   wsdbm:purchaseDate   ?v6 .
   ?v5   wsdbm:purchaseFor ?v0 .
}
```

Code Snippet D.15: Snowflake 3 (F3)

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
   ?v0   foaf:homepage  ?v1 .
   ?v2   gr:includes ?v0 .
   ?v0   og:tag   %v3% .
   ?v0   sorg:description  ?v4 .
   ?v0   sorg:contentSize  ?v8 .
   ?v1   sorg:url ?v5 .
   ?v1   wsdbm:hits  ?v6 .
   ?v1   sorg:language  wsdbm:Language0 .
   ?v7   wsdbm:likes ?v0 .
}
```

Code Snippet D.16: Snowflake 4 (F4)

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
   ?v0   gr:includes ?v1 .
   %v2%  gr:offers   ?v0 .
   ?v0   gr:price ?v3 .
   ?v0   gr:validThrough   ?v4 .
   ?v1   og:title ?v5 .
   ?v1   rdf:type ?v6 .
}
```

Code Snippet D.17: Snowflake 5 (F5)

Complex Queries

```
SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
   ?v0   sorg:caption   ?v1 .
   ?v0   sorg:text   ?v2 .
   ?v0   sorg:contentRating   ?v3 .
   ?v0   rev:hasReview  ?v4 .
   ?v4   rev:title   ?v5 .
   ?v4   rev:reviewer   ?v6 .
   ?v7   sorg:actor  ?v6 .
   ?v7   sorg:language  ?v8 .
}
```

Code Snippet D.18: Complex 1 (C1)

```
SELECT ?v0 ?v3 ?v4 ?v8 WHERE {
   ?v0   sorg:legalName ?v1 .
   ?v0   gr:offers   ?v2 .
   ?v2   sorg:eligibleRegion  wsdbm:Country5 .
   ?v2   gr:includes ?v3 .
   ?v4   sorg:jobTitle  ?v5 .
   ?v4   foaf:homepage  ?v6 .
   ?v4   wsdbm:makesPurchase  ?v7 .
   ?v7   wsdbm:purchaseFor ?v3 .
   ?v3   rev:hasReview  ?v8 .
   ?v8   rev:totalVotes ?v9 .
}
```

Code Snippet D.19: Complex 2 (C2)

```
SELECT ?v0 WHERE {
   ?v0   wsdbm:likes ?v1 .
   ?v0   wsdbm:friendOf ?v2 .
   ?v0   dc:Location ?v3 .
   ?v0   foaf:age ?v4 .
   ?v0   wsdbm:gender   ?v5 .
   ?v0   foaf:givenName ?v6 .
}
```

Code Snippet D.20: Complex 3 (C3)