Master's Thesis

Optimization of SPARQL Queries Processing Using PRoST

Guilherme Schievelbein

Examiner:	Prof. Dr. Georg Lausen
Co-examiner:	Prof. Dr. Peter Thiemann
Adviser:	Patrick Philipp

University of Freiburg
Faculty of Engineering
Department of Computer Science

March 20, 2019

Examiner

Prof. Dr. Georg Lausen

Co-examiner

Prof. Dr. Peter Thiemann

Adviser

Patrick Philipp

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

Resource Description Framework (RDF) is currently the standard model for the description of semantic data. RDF systems usually persist RDF datasets in the relational model, hence taking advantage of the years of research already done in databases technology. Due to the huge size of RDF datasets, such systems calls for the use of distributed data storage and parallel processing approaches.

Partitioned RDF on SPARK Tables (PRoST) is a distributed RDF system based on Apache Spark that aims to improve query processing time by replicating the RDF datasets with multiple RDF storage models. PRoST analyses input queries and creates a query execution plan that may use any of the data models implemented in the system. This is advantageous due to the heterogeneous query types applied to real life datasets. PRoST improves a query processing time by splitting it into multiple subqueries and selecting the most appropriate data model to be used for each subquery. The results of each subquery are then joined to obtain the final result of the query.

In this work, we introduce two new data models to PRoST: joined wide property table and dynamic extended vertical partitioning.

A joined wide property table is an extension of the two property table data models already implemented in PRoST. The goal of the joined wide property table is to further reduce the number of join operations required to process a query since these operations can be very costly in distributed systems.

The second data model, dynamic extended vertical partitioning, is inspired by the data model first introduced by the S2RDF system. S2RDF utilizes precomputed semi-join reductions of vertically partitioned tables to minimize the amount of data shuffled between nodes of a computer cluster when computing join operations. This comes, naturally, at a big preprocessing cost, as all possible join operations between the existing vertically partitioned tables must be computed when the RDF dataset is loaded. Therefore, we propose the persistence of semi-join reductions extracted from intermediate results obtained during the execution of queries. Thus, we create a dynamic database of semi-join reduction tables that can be used to decrease the processing time when multiple similar queries are executed.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

The Resource Description Framework (RDF) is the standard data model for the Semantic Web. The flexibility of RDF enables it to describe a wide range of structurally distinct datasets. It is not uncommon for those datasets to be extremely big, possibly consisting of billions of triples [1] [2]. This creates the need for distributed data management systems capable of efficiently processing heterogeneous query types and workloads. One such system, Partitioned RDF on Spark Tables (PRoST) achieves that by replicating and partitioning the RDF dataset using multiple data models [3]. When executing a query, PRoST splits it into multiple subqueries, each taking advantage of one available partitioning model expected to provide the fastest query execution time for that subquery structure. In this work, we extend PRoST's data partitioning models with the goal of improving PRoST's current query processing performance. More specifically, we implement a new type of property table and add support for the dynamic creation of semi-join tables during the processing of queries.

1.1. Monograph Organization

Chapter 2 covers the background concepts of this monograph. It describes the RDF, SPARQL, Apache Spark, and various data representation models.

In chapter 3, the PRoST system, as well as some of its functionalities, is introduced. More specifically, we introduce the current data partitioning models available in PRoST, how an RDF dataset is loaded, replicated, and partitioned, and how, given a query, PRoST creates a query execution plan using the available partitioning models.

Chapters 4 and 5 describes the new data partitioning models added to PRoST, details about their implementation, and what we expect to achieve with their introduction. Chapter 4 pertains to the joined wide property table (JWPT), an extension of PRoST's two existing property table implementations. Chapter 5 regards to the dynamic extended vertical partitioning, a query execution-time strategy for the creation of semi-join reduced vertically partitioned tables. This work was inspired by the relational partitioning technique introduced in the S2RDF engine [4].

The dataset and queries used for the evaluation of the implemented data models, as well as details of the computer cluster used for the benchmark, are described in chapter 6. The benchmark results and discussion of the joined wide property table, as well as of the dynamic extended vertical partitioning, are provided in chapters 7 and 8, respectively.

Finally, in chapter 9, we conclude this monograph by discussing what was achieved with this work and possible future research topics.

2. Background Concepts

In this chapter, we introduce the frameworks and data models utilized in the development of this work. We describe the RDF, which is the standard data modeling framework for the Semantic Web, and SPARQL, the standard query language for querying and manipulating RDF datasets. Furthermore, we describe Spark, an open-source framework for the distributed computing of very large RDF datasets. Lastly, we introduce some of the state-of-the-art relational data models used for the persistence of RDF datasets.

2.1. Resource Description Framework (RDF)

The Resource Description Framework is the W3C recommended model for the description of arbitrary facts about resources in the Semantic Web. RDF provides a standardized and versatile schema-free data model capable of describing arbitrary information about resources [5].

The basic elements of an RDF dataset are subject-predicate-object triples, where subject and object are resources in the dataset and predicate is a property. Each triple <s, p, o> in a dataset models an statement such as: the resource *s* has a property *p* with the value *o*. A predicate can be interpreted, therefore, as a directed labeled edge from an arbitrary resource (the subject) to another arbitrary resource (the object). For example, in a social network dataset, it is possible to describe the friendship of two users with the triple <User1, friendOf, User2>, as shown graphically in figure 1.

An RDF dataset is composed of an undefined number of interlinked triples, forming

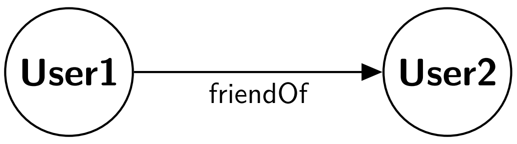


Figure 1.: RDF triple describing that *User1* is a friend of *User2* in a social network dataset

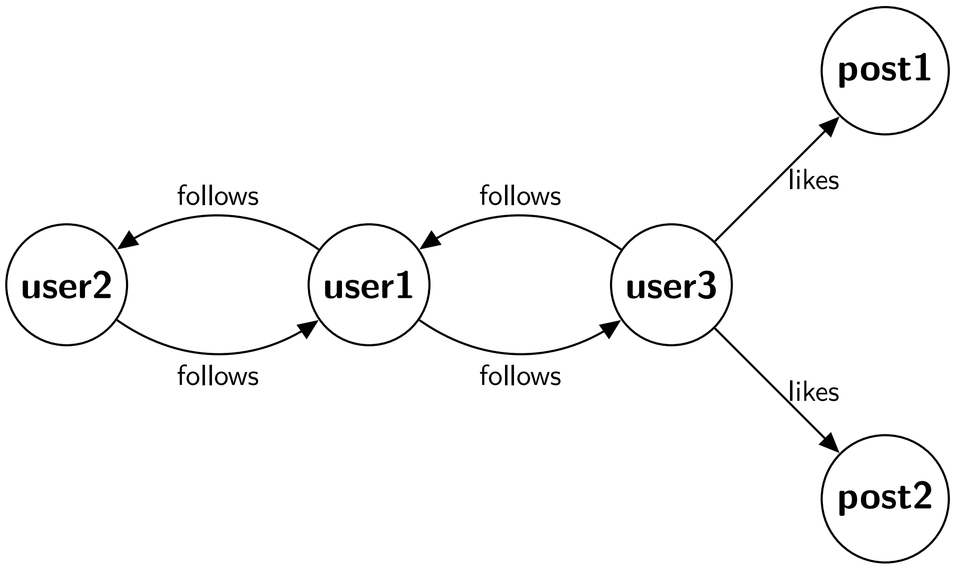


Figure 2.: Example of an RDF graph of a social network dataset

a so-called RDF graph. Figure 2 shows the RDF graph that will be used as the basis for the examples in this introductory chapter. The flexibility of RDF allows for the description of a huge variety of semantic data with arbitrary structure.

2.2. SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL is the standard query language for RDF datasets [6]. In its most basic form, a SPARQL query is a set of triple patterns that represent conditions that must be satisfied in the RDF dataset. The values from the variables of a SPARQL query may either be bound to an existing RDF term (bound variables) or have no set RDF term (unbound variables). The goal of a query is, in its essence, to find the possible values of the unbound variables in a SPARQL query that satisfies the conditions set by its triple patterns.

In the code listing 2.1 we show an example of a basic SPARQL query. The result of the query consists of all values of the variable *?p* that satisfies the restrictions enforced by the triple patterns in lines 3 and 4.

```
1 SELECT ?p
2 WHERE {
3           <user1> <follows> ?u .
4           ?u      <likes>   ?p
5 }
```

Listing 2.1: Example of a SPARQL query

The first pattern restricts the values of the subject and predicate of triples to *user1* and *follows*, respectively. Therefore *?u* can only assume the values of objects present in triples where the subject is equal to *user1* and the predicate is equal to *follows*. Similarly, in the second pattern, *?p* can be any object value where the subject is a possible value of *?u* and the predicate is equal to *likes*.

Line 1 contains all the variables that are returned by this query. Consequently, this query returns all values for *?p* that satisfies the restrictions from the triple patterns in lines 3 and 4. Basically, given the example dataset from 2, this query returns all posts liked by someone followed by *user1*.

A SPARQL query consisting only of triple patterns forms what is called a Basic Graph Pattern (BGP). Although more complex graph patterns may be formed by combining BGPs in different ways, in this monograph we use only queries composed of a single BGP, as that is enough to express what is equivalent to a natural join operation in a relational SQL query.

2.3. Apache Spark

The growing size of RDF datasets emphasizes the need for parallel processing and distributed storage solutions. Single-machine RDF systems do not scale to the processing of queries in large RDF datasets such as DBPedia and Bio2RDF [7]. Distributed computing systems provide a platform for the efficient processing of large scale RDF data. Unfortunately, those systems also impose new challenges, such as properly managing disk I/O and the shuffling of data between the nodes of the computer cluster.

Apache Spark is a framework designed for the processing of large RDF graphs in a distributed environment [8]. It provides scheduling, fault tolerance, and load balancing of computations in a computer cluster. The foundation of Spark is the Resilient Distributed Dataset (RDD). An RDD is a collection of elements that can be processed in parallel. The RDDs are kept in the memory of the cluster nodes. This provides a huge advantage over similar systems that utilize mainly in-disk computations instead of in-memory ones.

Spark introduces a high-level module for the processing of structured and semi-structured data, called Spark SQL. Spark SQL provides a data model, called DataFrame, equivalent to the relational model of Database Management Systems. With Spark SQL, SPARQL queries are translated into SQL queries, which are then processed using the DataFrames. Since DataFrames are equivalent to tables in the relational model, Spark can take advantage of all the state-of-the-art optimization research from that domain.

2.4. RDF Data Representation Models

RDF graphs are usually persisted with the relational database model. Spark SQL provides the means to process structured data with SQL queries, therefore using Apache Spark to process RDF data stored in a relational model is straightforward. Unfortunately, there is not a single standard schema for storing RDF data. There are many ways of storing RDF data in a relational model, each with their advantages and disadvantages. In this section, we discuss some of the state-of-the-art storage models for RDF data.

2.4.1. Triplestore

The triplestore model stores an RDF dataset in a single table, containing three columns: subject, predicate, and object. These columns correspond to the RDF triples subject, predicate, and object elements. Each triple from the RDF graph consists of a single row in the triplestore table. Table 1 shows how the RDF graph from figure 2 is stored with the triplestore model.

This approach, although extremely simple and flexible, also has many shortcomings. Queries require that the entire table be fully scanned at least once, or, with the presence of indexes, they require at least one index table lookup. This can potentially be very costly, considering that this single table contains the entire dataset and may be extremely big. Furthermore, complex queries would potentially require many self-joins on this table, which is inefficient with big tables. Lastly, due to the size of this table, it is likely not to fit entirely in memory, requiring further I/O operations to scan it. Since the entire dataset in contained in a single table, query optimization and selectivity estimation are also difficult with this type of storage model.

Triplestore		
subject	**predicate**	**object**
user1	follows	user2
user1	follows	user3
user2	follows	user1
user3	follows	user1
user3	likes	post1
user3	likes	post2

Table 1.: Example of a triplestore table

2.4.2. Property Table

The Property Table data model aims to improve the performance of queries when compared to the triplestore model by reducing the number of required self-joins operations [9]. A property table contains one column for subject resources and multiple columns for properties, named after predicate values from the RDF dataset. Each row of the property table contains a distinct subject, and the object values connected to each subject are stored in the appropriate property column. Predicates are not stored in the table but are part of the table metadata (the column name) instead. This way, multiple RDF triples with a common subject resource are persisted in a single row of the property table. This data model not only reduces storage requirements but provides faster processing time for some types of queries. More specifically, it reduces the number of self-joins required when running queries whose triple patterns have a common subject variable, as no self-join is needed anymore for join operations of the type *subject-subject*.

The property table of an RDF graph, as introduced in the Apache Jena framework, does not contain all data of the dataset [9]. Instead, a clustering algorithm is used to either group predicates that often appear together and could be united in a single property table, or group similar subjects that also could be stored in a single common property table. Therefore, multiple property tables are used with a single RDF dataset, and data that is not persisted in any property tables would still be stored using the triplestore storage model.

This approach has some issues. Since not all properties are defined for all subjects of the RDF graph, a property table may become sparse, with many NULL values for undefined property values. Similarly, a subject may also have multiple values defined for a single property, which hinders the modeling of the property table schema, since multiple values cannot be put in a single column of the table. Furthermore, there is

Property Table				
subject	**follows0**	**follows1**	**likes0**	**likes1**
user1	user2	user3	NULL	NULL
user2	user1	NULL	NULL	NULL
user3	user1	NULL	post1	post2

Table 2.: Example of a property table

the added complexity of running a clustering algorithm in the RDF data to create the property tables.

Table 2 shows a property table that stores all data from the RDF graph from figure 2. In this example, all properties are present in the Property Table. In this example, we handle multi-valued properties by duplicating those columns, so that all possible values of each property can be represented.

2.4.3. Vertical Partitioning

Vertical partitioning (VP) is a data partitioning and storage strategy for RDF datasets introduced in the SW-Store DBMS [10]. It aims to address some of the limitations of triplestore and property tables. With vertical partitioning, a table is created for every distinct property of the RDF dataset. Each table has two columns, one for subject values, and another for object values. Each row in a vertically partitioned table contains subject and objects values that are linked in the RDF graph through that table's respective property. In table 3 we show how the RDF graph from figure 2 is stored with vertically partitioned tables.

The main advantage of vertical partitioning when compared to the triplestore storage model is that, although a big number of join operations is still necessary for complex queries, the number of tuples in each joined table is smaller. Consequently, the amount of data that must be scanned and shuffled is reduced.

Furthermore, when compared to property tables, there is no sparsity in vertically

follows	
subject	**object**
user1	user2
user1	user3
user2	user1
user3	user1

likes	
subject	**object**
user3	post1
user3	post2

Table 3.: Example of vertically partitioned tables

partitioned tables and, since each triple from the RDF dataset is present in one row of a vertically partitioned table, the entire RDF dataset can be persisted using only vertical partitioning. This means that persisting a triplestore table becomes redundant. Additionally, multi-valued properties are easily handled, considering that they are simply stored in multiple rows of their respective vertically partitioned table. Lastly, vertical partitioning also avoids the added complexity of running a clustering algorithm in the dataset, as would be necessary to create property tables.

Naturally, vertical partitioning also has some disadvantages. When executing complex queries, even though the joined tables are smaller than a triplestore table, a larger number of joins is required than would be necessary if property tables were used instead. Consequently, querying on very large RDF datasets can still be very costly.

2.4.4. Extended Vertical Partitioning

Extended vertical partitioning (ExtVP) is a relational partitioning model used in the S2RDF system [4]. An ExtVP table is a materialized semi-join reduction of two vertically partitioned tables. A semi-join operation selects the tuples from one table only if they have a matching tuple in the second table, given their join columns. The ExtVP model effectively reduces the size of the tables that must be accessed when computing join operations by preemptively removing dangling tuples that would result from these operations.

Table 4 shows the computation of possible ExtVP tables for the vertically partitioned tables shown in table 3. We show the common join values in bold. In the example, *SS* and *OS* indicate on which columns the tables are joined. More specifically, *OS* indicates that the first table object column must be matched with the values from the second table subject column. We omitted the computations of ExtVP tables that are empty or equal to the original vertically partitioned tables, as these tables do need to be stored. All combinations of join columns (*SS*, *SO*, *OS*, *OO*) and tables (*follows*|*follows*, *follows*|*likes*, *likes*|*follows*, *likes*|*likes*) need to be checked to create all possible ExtVP tables. Given the ExtVP tables calculated for this example RDF graph, the execution of any query that would require a join operation between the VP tables *follows* and *likes* could instead compute a join between the appropriate smaller ExtVP table and the *likes* VP table.

The ExtVP model has all characteristics of vertical partitioning, with the further advantage of requiring even smaller tables when computing join operations. This comes with the added cost introduced by having to compute all possible semi-join

follows	
subject	**object**
user1	user2
user1	user3
user2	user1
user3	user1

$\ltimes$

likes	
subject	**object**
user3	post1
user3	post2

$\Longrightarrow$

ExtVP_SS follows_likes	
subject	**object**
user3	user1

follows	
subject	**object**
user1	user2
user1	**user3**
user2	user1
user3	user1

$\ltimes$

likes	
subject	**object**
user3	post1
user3	post2

$\Longrightarrow$

ExtVP_OS follows_likes	
subject	**object**
user1	user3

Table 4.: Computation example of ExtVP tables)

reductions beforehand, which is extremely costly with big datasets.

3. Partitioned RDF on Spark Tables (PRoST)

With the growing size of RDF graphs comes the demand for distributed RDF systems capable of efficiently storing and querying those large datasets. A distributed RDF system partitions the RDF data among a computer cluster's nodes and executes queries in a distributed way. The distributed execution of queries is often accomplished by splitting the main query into multiple smaller independent subqueries. Since the data is partitioned among many nodes, there may be a big communication cost for the exchange of intermediate results between the nodes before reaching the final result of the queries. To tackle that, these systems must employ data partitioning strategies with the goal of improving data locality and reducing the shuffling of data between the cluster's nodes, thus improving querying performance. Unfortunately, these partitioning strategies are often designed with a specific type of query in mind, and are therefore unable to maintain good performance when subjected to varied types of queries.

PRoST is an RDF system based on Apache Spark, available at `https://github.com/tf-dbis-uni-freiburg/PRoST/`, that aims to achieve fast processing times for a wide range of query types [3]. It achieves that by simultaneously storing the RDF dataset with multiple data storage models, each designed to attain a good querying performance with different query shapes. To process a query, PRoST splits it into multiple subqueries, each to be executed using one of the available data models. PRoST applies a heuristic to choose the data model expected to provide the best performance for each subquery. The results of each subquery are then joined to obtain the final result.

In the following sections, we delve into more details about the PRoST system. In section 3.1 we describe PRoST's supported data models. Section 3.2 outlines the steps taken by PRoST to load and collect statistics from an RDF dataset. Finally, section 3.3 shows how a query is split and executed by PRoST. Furthermore, it describes join trees, PRoST's structured way of representing a query execution plan.

3.1. Supported Data Models

PRoST replicates RDF datasets with 4 distinct data models: triplestore, vertical partitioning, and two variations of property tables called wide property table and inverse wide Property table.

PRoST's triplestore and vertically partitioned tables are created as described in sections 2.4.1 and 2.4.3. When a dataset is loaded with PRoST, it is first stored in a triplestore table. Due to its low performance, the triplestore table is not used when executing queries but rather as a basis to load the vertical partitioning tables and wide property tables.

Both of PRoST's variations of property tables adopt features provided by Apache Spark to avoid some of the drawbacks discussed in section 2.4.2.

Apache Spark natively supports the storage of complex data structures. Thus, the schema creation of property tables with multivalued properties becomes trivial, as all possible values of a property can be stored in a single column using an array data type.

Moreover, Apache Spark also supports the storage of tables with the Parquet format [11]. Parquet is a columnar storage format that provides better compression when compared with the traditional row-oriented format of other DBMS systems. With Parquet, all data of a column is stored consecutively, and NULL values are not physically stored, thus avoiding excessive disk space wasted storing sparse tables as it would occur with traditional storage formats. Parquet also decompose a table's schema in a way that only columns required by a query need to be accessed. Therefore it provides good performance even with extremely wide tables, considering that even if a table contains hundreds of columns, only the columns needed by the queries are read from disk.

A wide property table (WPT) is the first variation of a property table used in PRoST. Table 5 shows how a wide property table persists the graph from figure 2. Since the parquet format can efficiently read data from tables containing a large number of columns, every property from the dataset is loaded into a single wide property table, thus no clustering algorithm is needed to create the table's schema. Furthermore, in traditional data storage systems, multi-valued data would require special treatment, as not all values can be stored in a single cell of the table. This overhead is avoided in PRoST by storing array-type values, which is a supported complex data type with the parquet format. Lastly, the NULL values in the example wide property table from table 5 are not physically stored when using the parquet

Wide Property Table		
subject	**follows**	**likes**
user1	[user2, user3]	NULL
user2	user1	NULL
user3	user1	[post1, post2]

Table 5.: Example of a wide property table

Inverse Wide Property Table		
object	**follows**	**likes**
user1	[user2, user3]	NULL
user2	user1	NULL
user3	user1	NULL
post1	NULL	user3
post2	NULL	user3

Table 6.: Example of an inverse wide property table

format.

The second variation of a property table supported by PRoST is the inverse wide property table (IWPT). An IWPT has an *object* column instead of the WPT's *subject* column, and each property column stores subject values. The inverse wide property table for the RDF graph from figure 2 is shown in table 6.

The idea behind having two variations of wide property tables is that the WPT is used when multiple triple patterns in a SPARQL query have a common subject variable, and, if they have a common object variable instead, the IWPT is used. Using the wide property tables for these cases is advantageous because, since all data associated to a subject or object, depending on the table used, is contained in a single row, no further join operation is necessary to compute these multiple triple patterns. Both wide property tables are horizontally partitioned among the cluster's nodes, thus guaranteeing that rows are fully stored in a single node.

The vertically partitioned tables are used in PRoST to process triple patterns that do not have common subject or object variables. Since in these cases, with the available data models, join operations are unavoidable, the use of the vertically partitioned tables is preferable, considering that VP tables contain considerably fewer tuples than the wide property tables.

All of PRoST's data models contain the entire RDF dataset, therefore, when executing a query, it is possible to choose any desired combination of data models to be enabled for use when processing the query.

In the following two sections, we provide a quick overview of PRoST's RDF dataset loading and query execution phases, respectively.

3.2. Loading Phase

Given the RDF dataset to be loaded, it is first stored in a triplestore table. Although this table is not used during a query execution phase, it is used when creating the WPT, IWPT, and VP tables. The triplestore table is used because it provides a direct mapping from the input RDF dataset to a relational schema. All loaded tables are stored in a Hadoop Distributed File System (HDFS).

A crucial step of the loading phase is the generation of a statistics file. This file contains information such as:

- Names of all properties in the dataset.
- Number of distinct subject values for each property.
- Number of distinct object values for each property.
- Total number of tuples in a vertically partitioned table.

These statistics are used by PRoST during the execution phase to create an efficient query execution plan. During the execution phase, a query is split into smaller subqueries. Each created subquery is a node added to a tree structure, called a join tree. This structure defines the order in which each node is executed and joined. A single query may have many valid join trees. A heuristic is applied to choose, using the collected statistics, the most appropriate data model to be employed by each join tree node. The statistics are also used to set priorities to the nodes. Nodes whose computation is estimated to result in fewer tuples are assigned a higher priority and are preferably pushed to the bottom of the join tree and, as a consequence, executed first. By executing these nodes first we not only reduce the amount of data that needs to be shuffled between cluster nodes when executing join operations but also reduce the number of dangling tuples in these operations.

3.3. Execution Phase

Given a SPARQL query to be processed by PRoST, the query is divided into one or more subqueries, each containing at least one triple pattern from the original query. Each subquery is guaranteed to be processable with a SQL expression without a join operation when using one of PRoST's available data models. To avoid redundancy, each triple pattern is present in at most one subquery.

Join operations are very expensive since they require the shuffling of data across the cluster's nodes. Therefore, when creating the subqueries, PRoST tries to maximize the number of triple patterns contained in each one of them (or, in other words, minimize the total number of subqueries) and, as a consequence, reduce the total number of join operations necessary to process the query.

The first step to create the subqueries is to group every triple pattern by their subject and predicate values. Appropriately, each triple pattern will be present in two groups, one for its subject variable, and another for its object variable, unless they have the same value. Each of these groups is a subquery that can be processed without any join operations by using one of the wide property tables.

Next, since there are repeated triple patterns among the created groups, some of them need to be removed. The groups are ordered by the number of elements they contain in descending order. Patterns present in the larger groups are then removed from the smaller ones and, as a result, the groups have as many elements as possible without any redundancy.

Finally, a join tree node is created for each group. If the patterns in the group have a common subject, a WPT node is created, otherwise, an IWPT node is created. If the group only has one triple pattern, a VP node is created, as there would be no advantage in using a property table. The type of the nodes determines the data model that is used to execute the subquery that they represent. A priority is also assigned to each node, according to the following criteria:

- Nodes containing literals have the highest priority since that is a strong constraint in the resulting number of tuples.
- VP nodes that access vertically partitioned tables with fewer tuples have a higher priority than VP nodes that access larger tables.
- WPT nodes have their priority assigned according to the number of distinct subject values present in each property accessed by that node, as well as the total number of tuples with that property. Again, when smaller resulting tables are expected, higher priorities are assigned.
- IWPT nodes have their priority assigned similarly to WPT nodes, but distinct object values are considered instead.

After all nodes are created, the join tree is generated and the query can be executed.

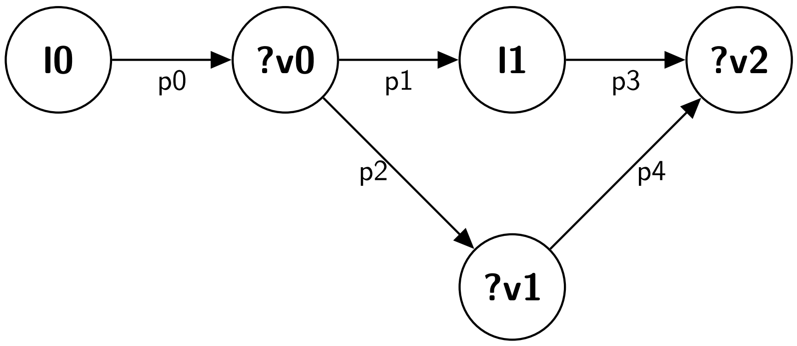


Figure 3.: Triple patterns as a graph

3.3.1. Example of a SPARQL Query Processing with PRoST

In this subsection, we demonstrate the steps taken by PRoST to process the SPARQL query shown in listing 3.1. The query is shown graphically in figure 3.

```
SELECT DISTINCT ?v0, ?v1, ?v2, ?v3
WHERE {
            l0   p0  ?v0 .
            ?v0  p1  l1  .
            ?v0  p2  ?v1 .
            l1   p3  ?v2 .
            ?v1  p4  ?v2
}
```

Listing 3.1: SPARQL query

First, the triple patterns are extracted from the query. For the sake of simplicity, we enumerate them as below:

1. l0 p0 ?v0
2. ?v0 p1 l1
3. ?v0 p2 ?v1
4. l1 p3 ?v2
5. ?v1 p4 ?v2

The triple patterns 2 and 3 have a common subject variable *?v0*, therefore they are both added to a single wide property table node, called WPT[2,3]. Similarly,

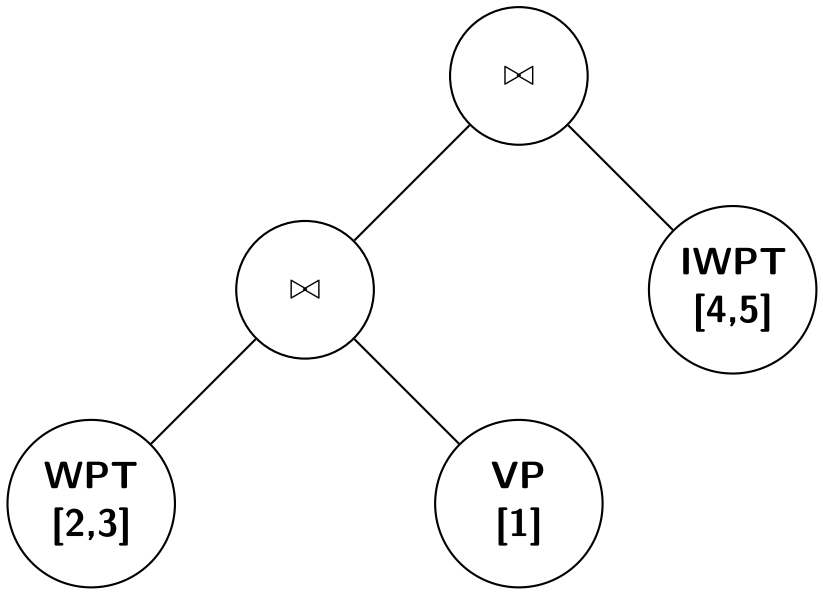


Figure 4.: Example of a Join Tree

triple patterns 4 and 5 have a common object variable and are added to an inverse wide property table node named IWPT[4,5]. The remaining triple pattern is added to the vertical partitioning node VP[1] since it cannot be grouped with any other triple pattern.

After all nodes required to compute the original SPARQL query are created, the query execution plan is generated in the form of a join tree, as shown in figure 4.

All leaf nodes of the join tree can be processed in parallel. Only the join operations need to be executed sequentially due to their dependencies. In listings 3.2, 3.3, and 3.4 we show the SQL expressions used to execute the WPT[2,3], IWPT[4,5], and VP[1] nodes, respectively. The resulting tables from processing these nodes are simply joined on their common variables to obtain the final result of the input SPARQL query.

```
SELECT  subject as v0, p2 as v1
FROM    wide_property_table
WHERE   p1 = "l1" AND p2 IS NOT NULL
```

Listing 3.2: SQL expression that processes a WPT node

```sql
SELECT  object as v2, p4 as v1
FROM    inverse_wide_property_table
WHERE   p3 = "l1" AND p4 IS NOT NULL
```

Listing 3.3: SQL expression that processes an IWPT node

```sql
SELECT  object as v0
FROM    vp_p0
WHERE   subject = "l0"
```

Listing 3.4: SQL query that processes a VP node

The heuristic used to create the join tree tries to reduce the number of join operations necessary to compute the query as much as possible, since these operations require the shuffling of data between the cluster's nodes, and are, therefore, costly. Additionally, in the cases when a join tree node could be joined with multiple other nodes, the order of the join operations also impacts the query computation performance. Therefore, the statistics stored during the loading phase are also used to estimate the expected selectivity of these operations, and joins with lower estimated selectivity are chosen, consequently reducing the size of intermediate tables as early as possible during the join tree execution.

4. Joined Wide Property Table

PRoST has two types of property tables, the wide property table, which reduces the number of subject-subject join operations necessary to process a query, and the inverse wide property table, which reduces the number of object-object joins necessary. This is advantageous because joins in a distributed system are expensive operations that usually require the shuffling of data between the nodes of the cluster.

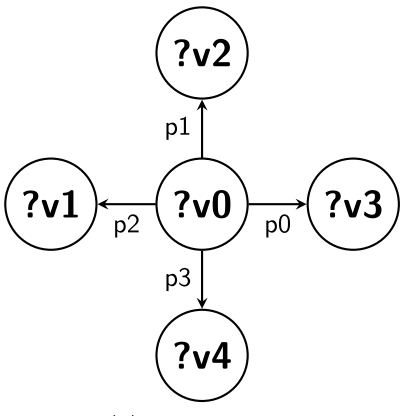
PRoST's wide property table is most effective when processing queries that contain a single common variable in the subject position of its triple patterns. As an example, a SPARQL query containing the below triple patterns (shown graphically in figure 5a), all of which have the same variable $?v_0$ in the subject position, can be processed with a single *SELECT* operation on the wide property table. Therefore PRoST creates a join tree with a single wide property table node. Figure 5b shows the join tree that processes that query.

1. ?v0 p1 ?v1
2. ?v0 p2 ?v2
3. ?v0 p3 ?v3
4. ?v0 p4 ?v4

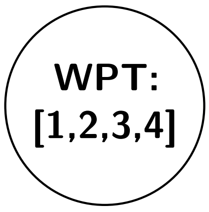
Similarly, a SPARQL query in which all triple patterns have the same object variable, such as the ones shown below (shows graphically in figure 6a), can be processed by PRoST with a single *SELECT* operation on the inverse wide property node, as shown in figure 6b.

1. ?v1 p1 ?v0
2. ?v2 p2 ?v0
3. ?v3 p3 ?v0
4. ?v4 p4 ?v0

PRoST can process both these triple pattern groups without executing any join operation. This was only possible because the common variable of these groups was

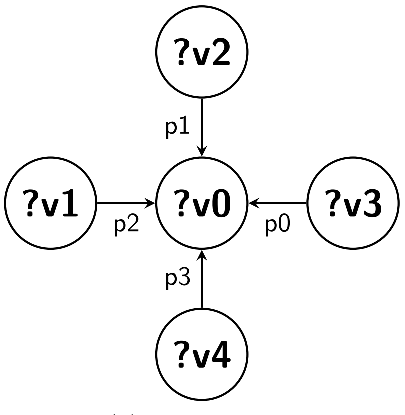


(a) Query graph

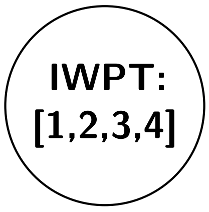


(b) Join Tree

Figure 5.: Star shaped query with a common subject variable in all triple patterns and the join tree that processes it

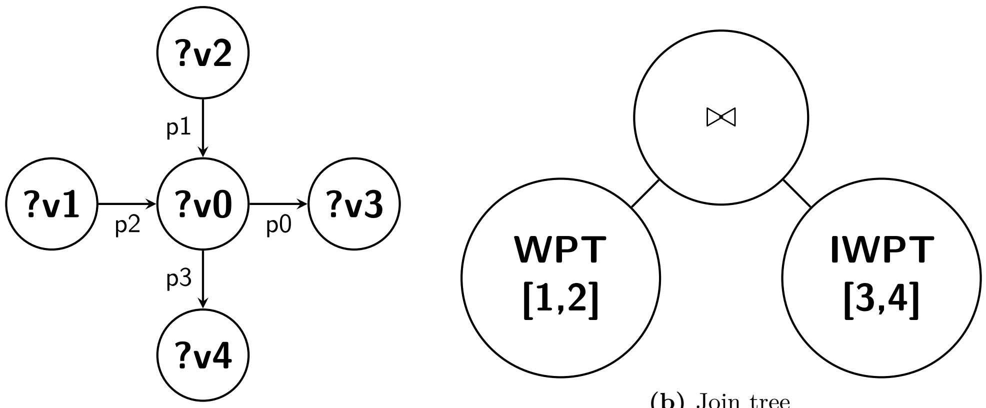


(a) Query graph



(b) Join tree

Figure 6.: Star shaped query with a common object variable in all triple patterns and the join tree that processes it



(a) Query graph

(b) Join tree

Figure 7.: Example of a star shaped query with a common variable in all triple patterns, but in different positions, and the join tree that processes it

always in the same position, that is, they are only a subject, or they are only an object. A SPARQL query with a common variable in all its triple patterns, but in different positions, such as the ones enumerated below and shown graphically in figure 7a, cannot be processed without a join operation when using PRoST's currently available data models.

1. ?v0 p1 ?v1
2. ?v0 p2 ?v2
3. ?v3 p3 ?v0
4. ?v4 p4 ?v0

All triple patterns in that group have the variable *v0* in common, but it is in the subject position of patterns 1 and 2, and object position of patterns 3 and 4. As a result, PRoST computes that query by joining a WPT node containing triples 1 an 2 with an IWPT node containing triples 3 and 4. The join tree generated by PRoST is shown in figure 7b.

In this chapter, we introduce a new type of wide property table, called joined wide property table (JWPT). The goal of this data model is to eliminate the need for join operations when processing queries containing triple patterns with a common variable, regardless if this variable is in the subject or object position of the triples.

A joined wide property table is a single property table that contains all information from both a wide property table and an inverse wide property table. In a JWPT, each property has two columns. One column stores subject values for that property,

Joined Wide Property Table				
likes^{-1}	**follows^{-1}**	**resource**	**follows**	**likes**
NULL	[user2, user3]	user1	[user2, user3]	NULL
NULL	user1	user2	user1	NULL
NULL	user1	user3	user1	[post1, post2]
user3	NULL	post1	NULL	NULL
user3	NULL	post2	NULL	NULL

Table 7.: Example of joined wide property table

similarly to the IWPT, and the other column stores object values, as in the WPT. Additionally, the JWPT has a single-valued column called *resource*. Elements in the *resource* column can be both subject and object values from the RDF dataset, according to which property columns it is associated with.

Basically, for an arbitrary property p, the JWPT contains a column p^{-1} with subject values, and a column p with object values. Elements from p^{-1} have the property p with the value from *resource*, and elements from *resource* have the property p with the values from the column p.

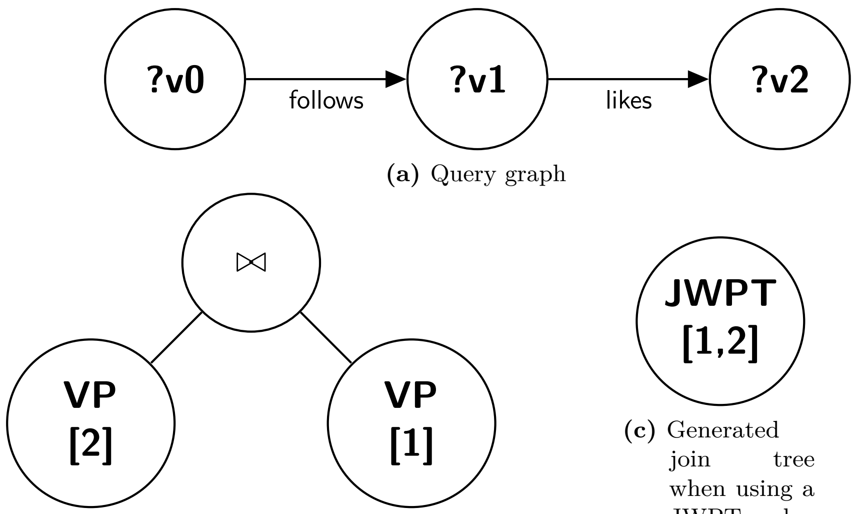
In table 7 we show the joined wide property table that stores the RDF graph from figure 2. Alike to the WPT and IWPT data models, the property columns can be multi-valued, and the NULL values are not stored physically since the table is stored in the parquet format. Moreover, since parquet is a columnar storage format, the increased number of columns of a JWPT when compared to a WPT or IWPT should not impact this model's performance negatively.

4.1. Join Tree Creation Example

As a case example, a SPARQL query containing the triple patterns below can be processed with a single scan of a joined wide property table, since all patterns have the common variable *?v1*. Without the availability of the joined wide property table, this query would require a join between two VP nodes. In figure 8 we show the graphical representation of the query and two possible execution plans, one that uses the JWPT and one that does not. In listing 4.1 we show the pseudo-SQL query that executes that JWPT node.

1. ?v0 follows ?v1

2. ?v1 likes ?v2



(a) Query graph

(b) Join tree generated without using a JWPT node

(c) Generated join tree when using a JWPT node

Figure 8.: Example of a query and two join trees, with and without the use a JWPT, that processes that query

```
SELECT   resource as v1, follows^-1 as v0, likes as v2
FROM     joined_wide_property_table
WHERE    follows^-1 IS NOT NULL AND likes IS NOT NULL
```

Listing 4.1: Pseudo-SQL expression that processes a JWPT node

4.2. Implementation

In the following sections, we show more details about the implementation of the joined wide property table in PRoST. Section 4.2.1 deals with how the JWPT is loaded from an RDF dataset, and section 4.2.2 concerns with the creation and execution of JWPT nodes, as well as the creation of a join tree that uses JWPT nodes.

4.2.1. Loading Phase

The first step, before any SPARQL query can be executed, is to load an RDF dataset with PRoST. A joined wide property table can be simply obtained with a full outer join operation between the data from a WPT and an IWPT. Therefore, the already existing algorithms in PRoST to create these tables can be used. After creating the WPT and IWPT tables, their schema must be changed before executing the outer join operation. The property columns from the IWPT DataFrame are renamed so that these columns are not merged with their counterpart columns in the WPT with the outer join operation. The columns *subject* in the WPT and *object* in the IWPT are also both renamed to *resource*, as the tables will be joined on this column.

The pseudo-code for creating the joined wide property table is shown in algorithm 1. The IWPT and WPT DataFrames are assumed to be already loaded and are passed as arguments to the procedure.

Algorithm 1 Create a joined wide property table

```
1: procedure CREATEJWPT(wptDataFrame, iwptDataFrame, jwptTableName)
2:     $iwptPropertiesNames \leftarrow iwptDataFrame.columns.toList()[1:]$
3:     for each $property \in iwptPropertiesNames$ do
4:         $inversePropertyName \leftarrow getInverseName(property)$
5:         $iwptDataFrame \leftarrow \rho_{inversePropertyName/property}(iwptDataFrame)$
6:     end for
7:     $wptDataFrame \leftarrow \rho_{resource/subject}(wptDataFrame)$
8:     $iwptDataFrame \leftarrow \rho_{resource/object}(iwptDataFrame)$
9:     $jwptDataFrame \leftarrow wptDataFrame ⟗ iwptDataFrame$
10:    $jwptDataFrame.saveAsTable(jwptTableName)$
11: end procedure
```

In figure 4.2.1 we show the loading times in milliseconds of an RDF dataset with around 100 million triples. The dataset contains approximately 5 million distinct subjects and 10 million distinct objects, and exactly 86 distinct properties. In the table, we show the total time necessary to load and persist the RDF dataset with PRoST when using different combinations of created property tables. The first combination shows the times to create and save a triplestore table, vertically partitioned tables, a wide property table, and an inverse wide property table. This is the default combination of tables created by PRoST. The second combination only stores a triplestore table, vertically partitioned tables, and the joined wide property table. The WPT and IWPT are still created, as they are necessary to compute the JWPT, but not persisted. In the last combination, the three types of property tables are persisted. In this case, the time required to compute the JWPT is significantly smaller, since it does not include the time necessary to generate the WPT and IWPT. We show the time necessary to load and store the tables for each data model independently.

We can see in the graph that the added workload necessary for the creation of the JWPT did not have a severely negative impact on the total loading time.

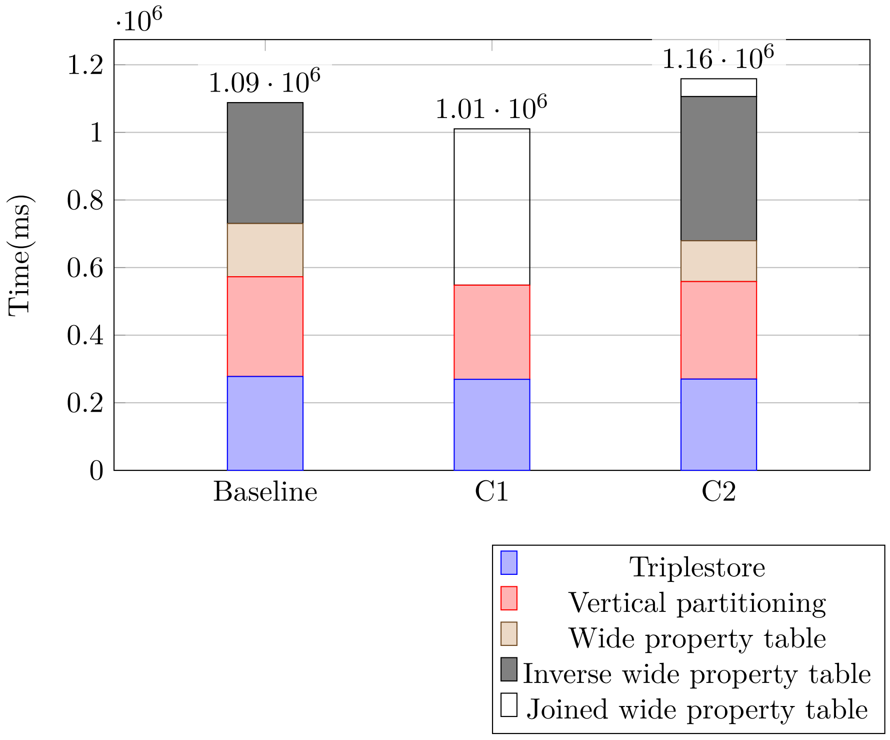


Figure 9.: Loading time of an RDF dataset with approximately 100 million triples

4.2.2. Execution Phase

During the execution phase, an input SPARQL query is executed on a previously loaded RDF dataset. The first step of the algorithm is to, given the set of all triple patterns from the SPARQL query, group them by their subject and object variables. The pseudo-code for that is shown in algorithm 2. A JWPT group is a custom data type that stores two lists of triple patterns, one where the common resource of the triples is a subject, and another where the common resource is an object. The JWPT group data type structure is shown in listing 4.2.

```
struct {
        Triple[] fTriples;
        Triple[] iTriples;
} JWPTGroup;
```

Listing 4.2: Data structure of a JWPT group data type

The algorithm iterates over all triples. Each triple is added to two JWPT groups,

Algorithm 2 Create JWPT groups

```
1: procedure CREATEJWPTGROUPS(triplePatterns)
2:     $jwptGroupsDictionary \leftarrow \emptyset$    ▷ Dictionary of JWPT groups. The common resource of a group is its key.
3:     for each $triple \in triplePatterns$ do
4:         $subject \leftarrow triple.subject$
5:         $object \leftarrow triple.object$
6:         if $subject \notin jwptGroups.keys()$ then
7:             $jwptGroups.add(subject, newJwptGroup(\emptyset, \emptyset))$
8:         end if
9:         $jwptGroups[subject].fTriples \leftarrow jwptGroups[subject].fTriples \cup triple$
10:        if $object \notin jwptNodes.keys()$ then
11:            $jwptGroups.add(object, newJwptGroup(\emptyset, \emptyset))$
12:        end if
13:        $jwptGroups[object].iTriples \leftarrow jwptGroups[object].iTriples \cup triple$
14:    end for
15:    return $jwptGroups$
16: end procedure
```

one for its subject resource, and the other for its object resource. If there is not an existing node for that resource, a new empty JWPT group is created for it.

If a join tree node were to be created for all JWPT groups, each triple pattern would be present in up to two nodes, causing redundancy. Therefore, duplicated triple patterns must be removed before creating the join tree nodes. This is done by iterating over the list of JWPT groups in descending order of the number of elements and removing their triple patterns from the smaller groups. The pseudo-code for that is shown in algorithm 3. The idea is to keep groups as large as possible and, consequently, create less join tree nodes. Having fewer nodes in a join tree means that fewer join operations are necessary to process it.

Finally, join tree nodes are created from the remaining groups with unique triples, as shown in algorithm 4. The algorithm receives as arguments the JWPT groups, booleans informing which data models are available for use, and the minimum number of triple patterns that must be present in a grouped node. By default, the minimum group size for the creation of WPT, IWPT or JWPT nodes in our experiments is equal to two. That means that any group with only one triple pattern is translated to a vertical partitioning node. For consistency, the priority of JWPT nodes is calculated the same way as the already existing WPT and IWPT nodes, as explained in section 3.3. If possible, WPT and IWPT nodes can be created instead of JWPT nodes. WPT and IWPT have fewer tuples than JWPT, and this might have an impact

Algorithm 3 Remove duplicated triples from JWPT groups

```
procedure RemoveDuplicateTriples(jwptGroups)
    $uniqueGroups \leftarrow \emptyset$
    while $jwptGroups \neq \emptyset$ do
        $largestGroup \leftarrow jwptGroups.getLargestElement()$
        $jwptGroups \leftarrow jwptGroups \setminus largestGroup$
        $uniqueGroups \leftarrow uniqueGroups \cup largestGroup$
        for each $triple \in largestGroup.fTriples \cup largestGroup.iTriples$ do
            for each $group \in jwptGroups$ do
                $group.fTriples \leftarrow group.fTriples \setminus triple$
                $group.iTriples \leftarrow group.iTriples \setminus triple$
                if $group.fTriples == \emptyset \land group.iTriples == \emptyset$ then
                    $jwptGroups \leftarrow jwptGroups \setminus group$
                end if
            end for
        end for
    end while
    return $uniqueGroups$
end procedure
```

on query processing performance. This difference in the number of tuples occurs because resources that only appear as a subject in the RDF graph are represented as a row in the WPT, but not in the IWPT. Similarly, resources that only appear as objects in the graph are a row in the IWPT, but not in the WPT. In the JWPT, rows for both these types of resources will be present. In chapter 6 we evaluate the execution time difference of using only JWPT nodes and using WPT and IWPT nodes when possible.

PRoST's algorithm to create the join tree receives as an argument the resulting *priority queue* containing all the nodes necessary to process the original SPARQL query. This algorithm works with any node type, as it only needs to access a node's priority and included triple patterns to generate the final join tree. No changes are necessary in this algorithm, therefore we omit it in this work.

Algorithm 4 Create join tree nodes from JWPT groups

```
1: procedure CreateNodes(jwptGroups, useWPT, useIWPT, minimumGroupSize)
2:     $priorityQueue \leftarrow \emptyset$
3:     for each $group \in jwptGroups$ do
4:         if $group.size() < minimumGroupSize$ then
5:             $priorityQueue \leftarrow priorityQueue \cup createVPNode(group)$
6:         else if $useWPT \wedge group.iTriples == \emptyset$ then
7:             $priorityQueue \leftarrow priorityQueue \cup createWPTNode(group)$
8:         else if $useIWPT \wedge group.fTriples == \emptyset$ then
9:             $priorityQueue \leftarrow priorityQueue \cup createIWPTNode(group)$
10:        else
11:            $priorityQueue \leftarrow priorityQueue \cup createJWPTNode(group)$
12:        end if
13:    end for
14:    return $priorityQueue$
15: end procedure
```

5. Dynamic Extended Vertical Partitioning Database

When executing a query, PRoST splits it into multiple smaller subqueries that can be executed independently and computes join operation on their common variables to retrieve the final result of the query. During the execution of a batch of queries in PRoST, often similar join operations between the same tables appear on multiple queries. Since intermediate results are discarded after the computation of query, these similar join operations between subqueries must be recomputed. If the information of previous query executions were not discarded, fewer resources would be recomputing these operations.

S2RDF is a Spark-based system that reduces the potential size of shuffled data during the computation of a join operation by storing semi-join reductions of vertically partitioned tables [4]. These reduced tables are called ExtVP tables. S2RDF precomputes all possible ExtVP tables for every available vertically partitioned table from an RDF dataset. These ExtVP tables are used instead of VP tables when a join operation is required by a query, due to their smaller number of tuples. ExtVP tables remove dangling tuples that would result from join operations, thus minimizing the amount of data shuffled between the cluster's nodes. Although these semi-join reductions can significantly improve the RDF system performance when executing the queries when compared to the use of only vertically partitioned tables, it comes with a huge preprocessing overhead and storage cost, since all ExtVP tables must be computed and stored beforehand.

Due to the performance increased that ensues from the use of ExtVP tables, we propose their inclusion to PRoST. Instead of precomputing ExtVP tables during the loading of a dataset, we propose the use of intermediate results obtained during the execution phase of a query to extract and persist these tables. Consequently, not only we avoid increasing the loading time of RDF datasets, but the storage cost of ExtVP tables in PRoST is also smaller than in the S2RDF system since not all possible ExtVP tables for the RDF dataset are created. After a join operation between two VP tables is calculated in PRoST, this intermediate result is used to store the ExtVP table of that join pattern. Subsequently executed queries containing the same join pattern can then use the persisted ExtVP tables to improve their execution performance.

In section 5.1 we give an overview of how we dynamically maintain a database of ExtVP tables during the execution phase of PRoST. The implementation details of the proposed ExtVp tables creation strategy is provided in section 5.2.

5.1. Approach

In this section, we show how we added support for ExtVP tables in PRoST by demonstrating the steps necessary to process an example SPARQL query. The SPARQL query used for this demonstration has the following triple patterns:

1. ?v0 p0 ?v3
2. ?v0 p1 l0
3. ?v3 p2 ?v4
4. ?v4 p3 l1

To execute that query, the triple patterns are assigned to triple pattern groups. Each group is a subquery represented as a join tree node in PRoST. The join tree created by PRoST represents an execution plan for the query, where each node uses one of the available data models to process its respective subquery. In the query from this example, triple patterns 1 and 2 have the common variable *v0*, therefore they both can be assigned to single WPT node. The remaining triple patterns, assuming the ExtVP database is empty, are each assigning to a distinct VP node. A possible Join Tree created by PRoST to process that query is shown in figure 10.

The further execution of a similar query, such as one where only the values of constants (*l0* and *l1* in the example) is changed, would result is the same join tree. Consequently, the entire vertically partitioned tables for properties *p2* and *p3* would need to be used again. The triple patterns in these VP nodes have a common variable *?v4*, therefore, they could instead use appropriate ExtVP tables, if they are available. The use of ExtVP tables would result in less dangling tuples after the computation of the join operations.

ExtVP tables can be extracted from the intermediate result obtained after any join operation between VP nodes. To create those tables, before the creation of a join tree, we search for pairs of VP nodes with common variables and create a new

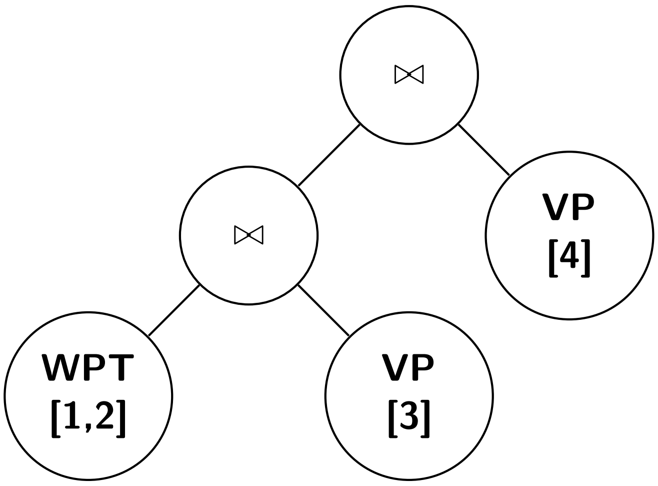


Figure 10.: Join tree created by PRoST when no ExtVP tables are available

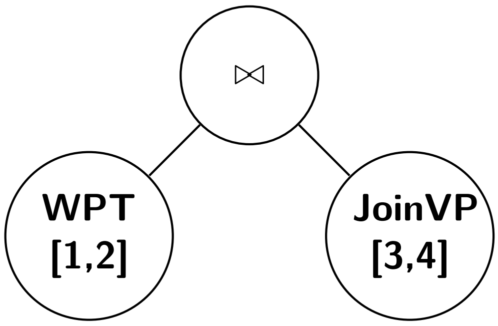


Figure 11.: Join tree with VP nodes merged in a single JoinVP node

node, called JoinVP, for each pair. If more than one pair is possible for a single triple pattern, the statistics collected during the loading-phase are used to estimate which pair would result in fewer tuples when joined. The ensuing join tree created for the example query is shown in figure 11.

This JoinVP node abstracts the execution of the join operation between two VP nodes, as well as the extraction and persistence of ExtVP tables. Unfortunately, the ExtVP tables cannot be created directly from the resulting join between VP nodes. Any bound variable in a triple pattern leads to a selection operation in its SQL query. Since ExtVP tables persist the semi-join reduction of tables without any selection, these variables must be bound only after the ExtVP tables are created.

The node VP[4] contains a triple pattern with a bound variable. This pattern is changed to the following pattern, where the constant value was substituted by a temporary variable: **[4*] ?v4 p3 ?temp_o_q4**. This variable is only bound to

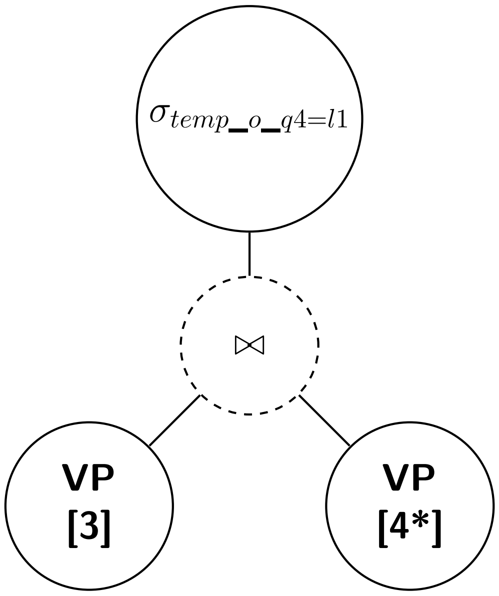


Figure 12.: Execution plan of the JoinVP[3,4]

the original constant value after the creation of the ExtVP tables. The execution of the JoinVP[3,4] will, therefore, execute the following steps:

1. Compute the join operation between the nodes VP[3] and VP[4*].
2. Extract and save the ExtVP tables from the current intermediate result.
3. Bound any temporary variable in the intermediate table to their original constant value.

The execution plan of JoinVP[3,4] is shown in figure 12. The ExtVP tables are created from the resulting table of the join operation shown as a dashed circle in the figure. Statistics such as the number of tuples, distinct objects, and distinct subjects in the ExtVP tables are also collected and saved. More details about the creation of the ExtVP tables from this intermediate result are discussed in section 5.2.

Any subsequent query executed will use previously created ExtVP tables, instead of VP tables, when possible. ExtVP nodes are created for them. These nodes can be executed exactly as VP nodes since their tables have the same schema. The statistics collected during the creation of the ExtVP table are used to determine its priority using the same heuristic used to calculate VP nodes priorities. In figure 13 we show the join tree created for the example query when it is possible to use previously generated ExtVP tables.

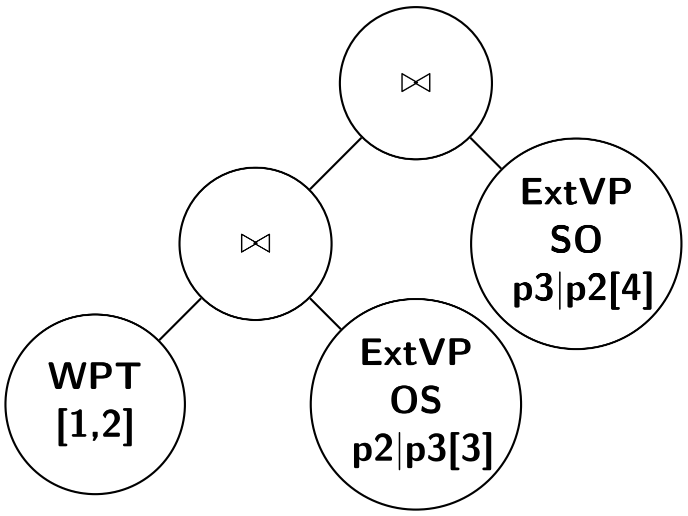


Figure 13.: Join tree with ExtVP nodes

With this strategy for the dynamic creation of ExtVP tables, the execution of queries whose join trees have JoinVP nodes is expected to be slower than the execution of the same queries using VP nodes only, since there is the overhead caused by having to save the extracted ExtVP. As the database of ExtVP tables grows over time and using those tables provides better performance than using vertically partitioned tables, we expect the performance for the execution of queries to gradually improve over time.

5.2. Implementation

In this section, we show the implementation details of the proposed strategy for maintaining a dynamic database of ExtVp tables. The first step to process a query in PRoST is to split it into smaller subqueries and create their respective join tree nodes.

The pseudo-code in algorithm 5 shows how, given an input query, the join tree nodes required to process that query are created. The procedure receives as arguments the list of triple patterns in the query and the statistics file. The nodes are added to a priority queue that is afterward used to generate the join tree. Since the use of property tables is preferable, when possible, to vertically partitioned tables, those nodes are created first. We maintain a list of patterns that are not included in a node yet. We iteratively remove from this list pairs of patterns that can be joining together, and create for them ExtVP nodes, if there exist equivalent ExtVP tables, or JoinVP nodes. We repeat that until no more pairs can be removed. Any pattern still in this list is translated to a VP node. When multiple pairs are possible, the one expected to result in the intermediate result with fewest tuples is chosen. This is done by using the already existing algorithm in PRoST to calculate the priority of VP nodes. The priority of ExtVP and JoinVP nodes are also calculated using these same algorithm.

Algorithm 5 Create join tree nodes with the dynamic ExtVP database

```
1: procedure CREATENODES(patterns, statistics)
2:     $queue \leftarrow createPTNodes(patterns)$
3:     $remainingPatterns \leftarrow patterns$
4:     for each $node \in queue$ do
5:         for each $pattern \in node$ do
6:             $remainingPatterns \leftarrow remainingPatterns \setminus pattern$
7:         end for
8:     end for
9:     while $TRUE$ do
10:        $patternA, patternB \leftarrow statistics.getBestJoin(remainingPatterns)$
11:        if $patternA \neq \emptyset \wedge patternB \neq \emptyset$ then
12:            if $statistics.existExtVpTables(patternA, patternB)$ then
13:                $queue \leftarrow queue \cup createExtVpNodes(patternA, patternB)$
14:            else
15:                $queue \leftarrow queue \cup createJoinVpNodes(patternA, patternB)$
16:            end if
17:            $remainingPatterns \leftarrow remainingPatterns \setminus (patternA \cup patternB)$
18:        else
19:            $queue \leftarrow queue \cup createVpNodes(remainingPatterns)$
20:            return $queue$
21:        end if
22:    end while
23: end procedure
```

After all nodes are created, the join tree is constructed. Similarly to our previous implementation of JWPT, no changes are needed in PRoST's existing algorithm for the creation of join trees, as this algorithm works with any arbitrary node structure, as long as they implement methods to retrieve their priorities and the variables included in their triple patterns. All join tree node types must also implement a procedure that executes an SQL query that process its triple patterns.

The execution of an ExtVP node is trivial, as it is simply a select operation in a single table that has the same schema of the vertically partitioned tables. As discussed in the previous section, the execution of a JoinVP node is more complex.

We shown the pseudo-code for that in algorithm 6. A JoinVP node contains two VP nodes. There are no bound variables in the triples of these VP nodes. Any bound variable from the original triples are removed, and that information is stored in a variable called *constantsMapping*. To execute a JoinVp node, first the VP nodes are executed and joined. The two ExtVP tables are extracted from that intermediate data and stored in a database. Additionally, the statistics for the created tables are also computed and saved. Only after the ExtVP tables are saved the temporary variables are bound to their correct values and the final result of that node is returned.

Algorithm 6 Execute a JoinVPNode

```
1: procedure EXECUTENODE(joinNode)
2:     $df1 \leftarrow executeVPNode(joinNode.vpNode1)$
3:     $df2 \leftarrow executeVPNode(joinNode.vpNode2)$
4:     $joinDf \leftarrow df1.join(df2)$
5:     $saveExtVpTables(vpNode1.pattern, vpNode2.pattern, joinDf, statistics)$
6:     for each $(c, v) \in contantsMapping$ do
7:         $joinDf \leftarrow joinDf.where(col(v).eq(c)).drop(v))$
8:     end for
9:     return $joinDf$
10: end procedure
```

In algorithm 7 we show the pseudo-code for the creation of ExtVP tables from the intermediate results that are obtained during the execution of previously discussed procedure. These tables can be created by merely extracting distinct values of the appropriate columns from that intermediate table. As an example, given an intermediate result as shown in table 8, obtained from the join of the tables *follows* and *likes*, the ExtVP table for *follows* contains distinct rows from the selection of the columns *v0* and *v1*. Similarly, the ExtVP table for *likes* contains the distinct rows from the selection of columns *v1* and *v2*. The procedure *createExtVPTableName* called by the algorithm generates a name for the ExtVP table, according to the position (subject or object) of the common variables on the original vertically partitioned tables.

Lastly, as an optional step, when the database of ExtVP tables starts using too much storage space, the statistics collected can be used to remove tables with high selectivity or that are rarely used. ExtVP tables with high selectivity are those that still contains a large number of tuples from the original VP table, and therefore do not provide a good performance improvement when used.

To wrap things up, in algorithm 8 we show the method calls taken by PRoST to

Algorithm 7 Save ExtVP tables

```
procedure saveExtVpTables(pattern1, pattern2, df, statistics)
    $t1 \leftarrow df.select(col(t1.subject).alias("s"), col(t1.object).alias("o"))$
    $t2 \leftarrow df.select(col(t2.subject).alias("s"), col(t2.object).alias("o"))$
    $tableName1 \leftarrow createExtVPTableName(pattern1, pattern2)$
    $tableName2 \leftarrow createExtVPTableName(pattern2, pattern1)$
    $saveDf(t1, tableName1)$
    $saveDf(t2, tableName2)$
    $updateStatistics(statistics, t1, tableName1))$
    $updateStatistics(statistics, t2, tableName2))$
end procedure
```

VP__follows	
v0	**v1**
user1	user2
user1	user3
user2	user1
user3	user1

$\bowtie$

VP__likes	
v1	**v2**
user3	post1
user3	post2

$\Longrightarrow$

Intermediate Result		
v0	**v1**	**v2**
user1	user3	post1
user1	user3	post2

Table 8.: Example of a join operation between VP nodes)

process a SPARQL query, given their triple patterns and the database statistics. The method that processes a join tree is shown in algorithm 9.

Algorithm 8 Execute a SPARQL query

```
procedure execute(patterns, statistics)
    $priorityQueue \leftarrow createNodes(triples, statistics)$
    $joinTree \leftarrow createJoinTree(priorityQueue)$
    $result \leftarrow executeTree(joinTree)$
end procedure
```

Algorithm 9 Process a join tree

```
1: procedure EXECUTETREE(joinTree)
2:     if $joinTree.hasChildren()$ then
3:         return $executeTree(joinTree.left) \bowtie executeTree(joinTree.right)$
4:     else
5:         return $joinTree.executeNode(JoinTree)$
6:     end if
7: end procedure
```

6. Evaluation Environment

The evaluation experiments for the developed data models were executed in a computer cluster of 10 machines, one master machine, and 9 workers, connected via Gigabit Ethernet protocol. Each machine has a 6 Core Intel Xeon E5-2420 processor, 32GB of RAM, and 4TB of disk space. The machines are running Cloudera CDH 5.11.0 and Spark 2.2.0 on an Ubuntu 14.04 operating system.

We use a dataset and queryset generated with the Waterloo SPARQL Diversity Test Suite (WatDiv) [12]. WatDiv creates synthetic RDF datasets and SPARQL queries with a wide range of shapes with the objective of providing stress-testing tools for RDF systems and evaluating how those systems perform with varying types of queries.

The position of the variables in the triple patterns of a SPARQL query affects that query's shape, which in turn can impact the query processing performance, since the strategy chosen to execute a query may favor one specific shape of query. Therefore, we discuss the results obtained for each query type independently.

We generated a WatDiv dataset with approximately 100 million RDF triples. The dataset contains around 5 million distinct subjects, 80 predicates, and 9 million objects. The benchmark queries can be classified into four query types: linear, star, snowflake, and complex shaped queries. Multiple queries were used in each category. There are 88 distinct queries, distributed as follows: 5 linear-shaped query groups with 5 queries each; 7 star-shaped query groups with 5 queries each; 5 snow-shaped query groups with 5 queries each; and 3 complex-shaped queries. The difference between the queries in a group is only the value of their constants. One sample of each query group is available in the appendix A.

In the following section, we provide a description of the shapes of the queries used. The evaluations of the JWPT data model and dynamic ExtVP database are available in chapters 7 and 8, respectively. In these chapters, we show the execution time of the query with and without the implemented strategies and discuss whether their differences are statistically significant. The queryset is executed 50 times to obtain the average execution times and standard deviations.

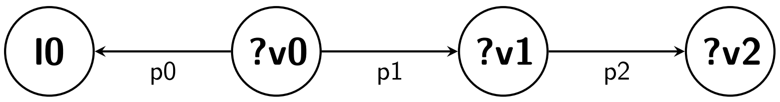


Figure 14.: Example of a linear-shaped query

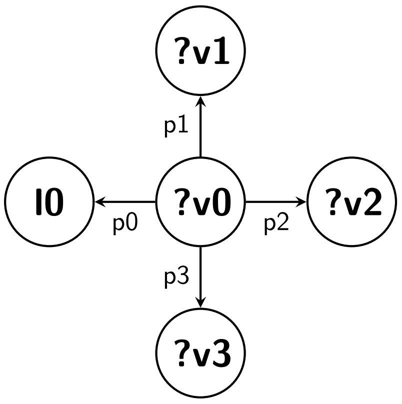


Figure 15.: Example of a star-shaped query

6.1. Query Shapes

Linear-shaped queries consist mostly of triple patterns with distinct subject and object variables, that is, the join operations are mainly of the type subject-object, as visualized in figure 14. This type of query usually has a worse performance in PRoST when compared with the other types, as it usually requires the most join operations. The query plan created by PRoST for this type of query uses mostly vertically partitioned tables instead of property tables.

Star-shaped queries contain a single central variable, present in all triple patterns of the SPARQL query, as shown in figure 15. Most RDF systems are optimized for this type of query. Property tables are especially effective when used with this type of query.

Snowflake-shaped queries are formed by multiple star-shaped queries, as in figure 16. Again, property tables are suited for this type of query, although it requires more join operations that star-shaped queries.

The last shape of queries used in the benchmark is the complex-shape. They consist of the combination of multiple of the previously discussed query shapes. In figure 17 we display an example of such query.

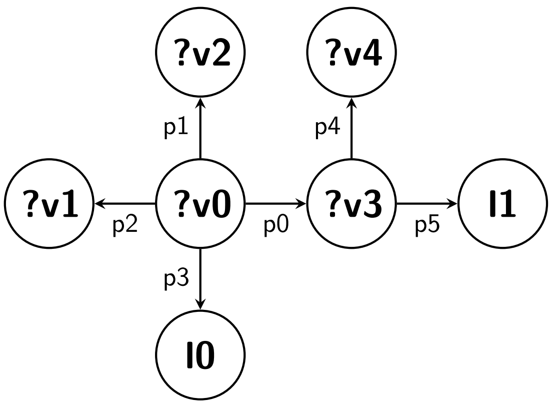


Figure 16.: Example of a snow-shaped query

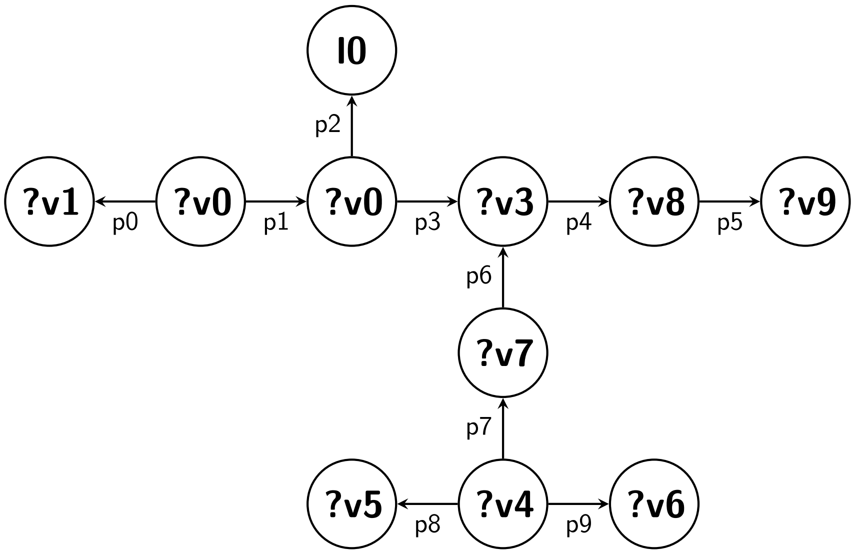


Figure 17.: Example of a complex-shaped query

7. Joined Wide Property Table Evaluation

In this chapter, we discuss the performance of PRoST when executing the benchmark queryset with and without using the joined wide property table. We compare the execution time of queries with three different combinations of enabled data models:

- **VP + (I)WPT:** The default strategy used by PRoST. Enables the use of WPT, IWPT, and VP nodes. Used as the baseline case to be compared with the results obtained when using the JWPT.

- **VP + JWPT:** In this mode only JWPT and VP nodes are enabled.

- **VP+ (I/J)IWPT:** In this mode, the VP tables and all property tables are used. The difference between using all available property tables and using only the JWPT is that when all common variables from a grouped node are in the same position (subject or object) the WPT or IWPT is used instead of the JWPT. Since the WPT and IWPT have fewer tuples than the JWPT, we want to verify if this causes a significant impact on the execution time of the queries.

For each query shape, we show a table displaying the average processing time of all individual queries with the baseline strategy, and the time difference of the other two strategies when compared to that baseline. Negative values mean faster processing times, and positive values, slower. The queryset was executed a total of 50 times. We highlight in the table the cases that were statistically significant with $\alpha = 0.05$ (blue when it is a faster execution time, red otherwise). We also provide a bar graph displaying the average processing time of the queries. For clarity, similar queries (where the only differences between them are the values of the constants) are grouped. The graph uses a logarithm scale.

	VP + (I)WPT	VP + JWPT	VP + (I/J)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
L1-1	3238	-1658	-1625
L1-2	3108	-1630	-1566
L1-3	3214	-1718	-1627
L1-4	3245	-1569	-1599
L1-5	3047	-1600	-1476
L2-1	848	183	-5
L2-2	841	162	23
L2-3	843	195	9
L2-4	838	179	-7
L2-5	1446	238	4
L3-1	707	147	963
L3-2	624	166	14
L3-3	657	195	30
L3-4	633	177	16
L3-5	632	215	25
L4-1	561	200	4
L4-2	518	216	5
L4-3	502	226	15
L4-4	509	194	16
L4-5	511	1163	13
L5-1	631	814	676
L5-2	632	846	635
L5-3	582	782	532
L5-4	606	811	628
L5-5	659	859	708

Table 9.: Execution time of linear-shaped queries

7.1. Linear-Shaped Queries

We show the execution times of the linear-shaped queries in table 9. Although the query group L1 could be processed faster when using the JWPT, we did not manage to obtain the same improvement with the other query groups.

By analyzing the join trees created for each query group, we could observe that all JWPT nodes created for groups L2, L3, L4, and L5 could be substituted by a WPT node since the common variable from their triples were always located in the subject position. We can observe that when enabling the use of all available property tables, there is not a significant difference in the execution times with and without

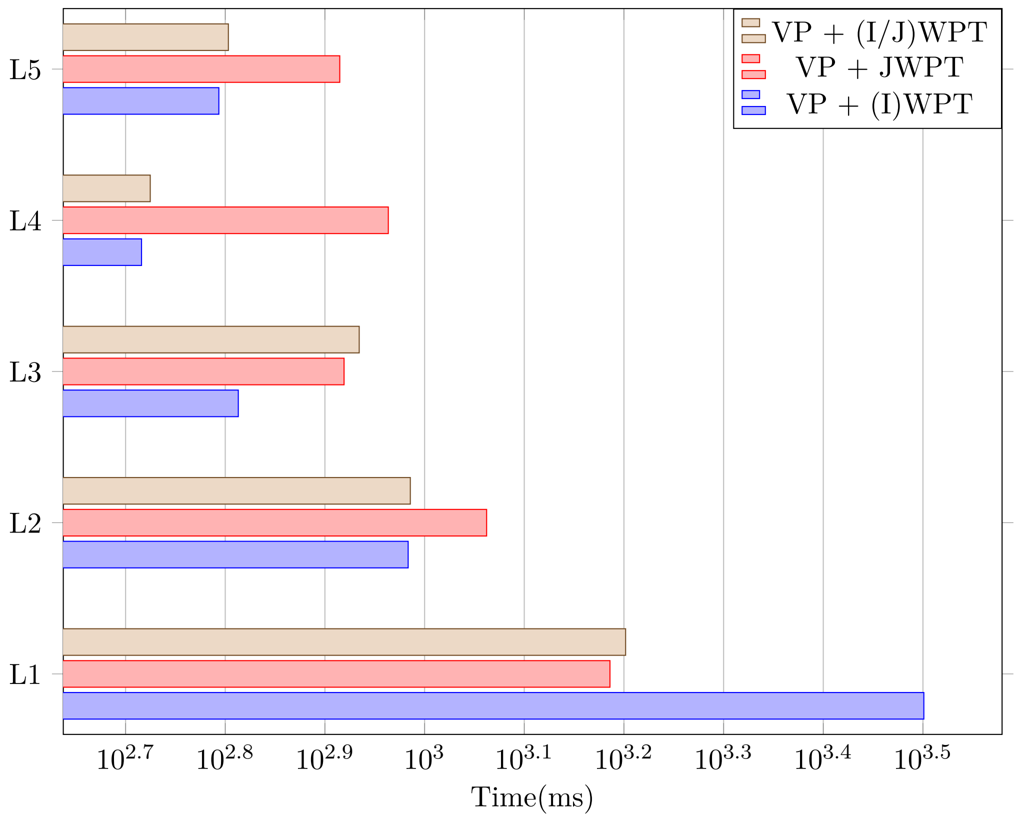


Figure 18.: Execution time of linear-shaped queries

the JWPT. This happens because in these cases the created join tree is the same.

We can conclude from these results that, although we can obtain faster results with a JWPT, a WPT is desirable if possible, since the same triple patterns are processed slightly slower in the JWPT than in the WPT. This most likely occurs because, since it contains more tuples, the JWPT is distributed among more partitions in the HIVE database.

The graph with the average execution time of each linear-shaped query is available in figure 18.

7.2. Star-Shaped Queries

We can observe a similar behavior to linear-shaped queries in the processing of star-shaped queries (table 10).

Most star-shaped queries from the benchmark could be executed with a single WPT node, and when only the use of the JWPT is enabled, the execution time was

	VP + (I)WPT	VP + JWPT	VP + (I/J)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
S1-1	1510	-42	-41
S1-2	1591	-130	-134
S1-3	1677	-119	-162
S1-4	1558	-95	-82
S1-5	1453	49	-3
S2-1	732	198	2
S2-2	635	215	2
S2-3	660	229	7
S2-4	655	206	-2
S2-5	680	219	-5
S3-1	656	218	1
S3-2	643	205	3
S3-3	676	206	-10
S3-4	624	226	17
S3-5	705	180	-3
S4-1	2547	-1715	-1675
S4-2	2688	-1789	-1808
S4-3	2040	-1131	-1166
S4-4	2743	-1849	-1863
S4-5	2696	-1792	-1817
S5-1	757	212	18
S5-2	732	146	-29
S5-3	775	280	12
S5-4	705	214	-3
S5-5	763	258	-9
S6-1	587	190	-10
S6-2	614	266	49
S6-3	585	193	-16
S6-4	561	319	8
S6-5	588	192	89
S7-1	983	-92	-102
S7-2	1002	-117	-116
S7-3	1108	-176	-216
S7-4	1051	-160	-157
S7-5	1007	-81	-107

Table 10.: Execution time of star-shaped queries

again slightly slower. The only exceptions were the queries from the query groups S1, S4, and S7. Without enabling the JWPT, those queries are processed by joining a WPT node with a VP node ($WPT \bowtie VP$), whereas they can be executed without any join operation with a JWPT node.

The average execution time of each star-shaped query group is displayed in the graph from figure 19.

7.3. Snow-Shaped Queries

We obtained the most meaningful results in the execution time of snow-shaped queries (table 11).

The snow-shaped queries from the benchmark consist of two connect star-shaped queries, therefore we can expect them to require at least one join operation between WPT nodes, such as $WPT \bowtie WPT$. The query groups F1 and F2 are processed with one such operation. When enabling only the JWPT, the operation executed is of the type $JWPT \bowtie JWPT$ and required on average one more second to be executed. When all property tables are enabled, again a join tree such as $WPT \bowtie WPT$ is used, and therefore required about the same time to be executed as in the baseline strategy.

The other query groups (F3, F4, and F5) required a join tree with the following form: $WPT \bowtie (WPT \bowtie VP)$. When the JWPT is enabled, a join tree such as $JWPT \bowtie JWPT$ is created instead, therefore needing one less join operation to be executed. This had a very significant impact on the execution time, as it was on average almost six seconds faster.

We show the graph with the average execution time of the snow-shaped queries in figure 20.

7.4. Complex-Shaped Queries

The average execution time of complex-shaped is shown in table 12.

To more easily discuss the execution times obtained for those queries, we show the join trees created for queries C1 and C2 with and without using the JWPT in figures 21 and 22. We omit the join tree for the query C3 since, as can be seen in its SPARQL query available in appendix A.4, it is simply a very big star-shaped query that can be executed with a single JWPT or WPT node.

Both queries C1 and C2 were processed slower when using the JWPT. When using

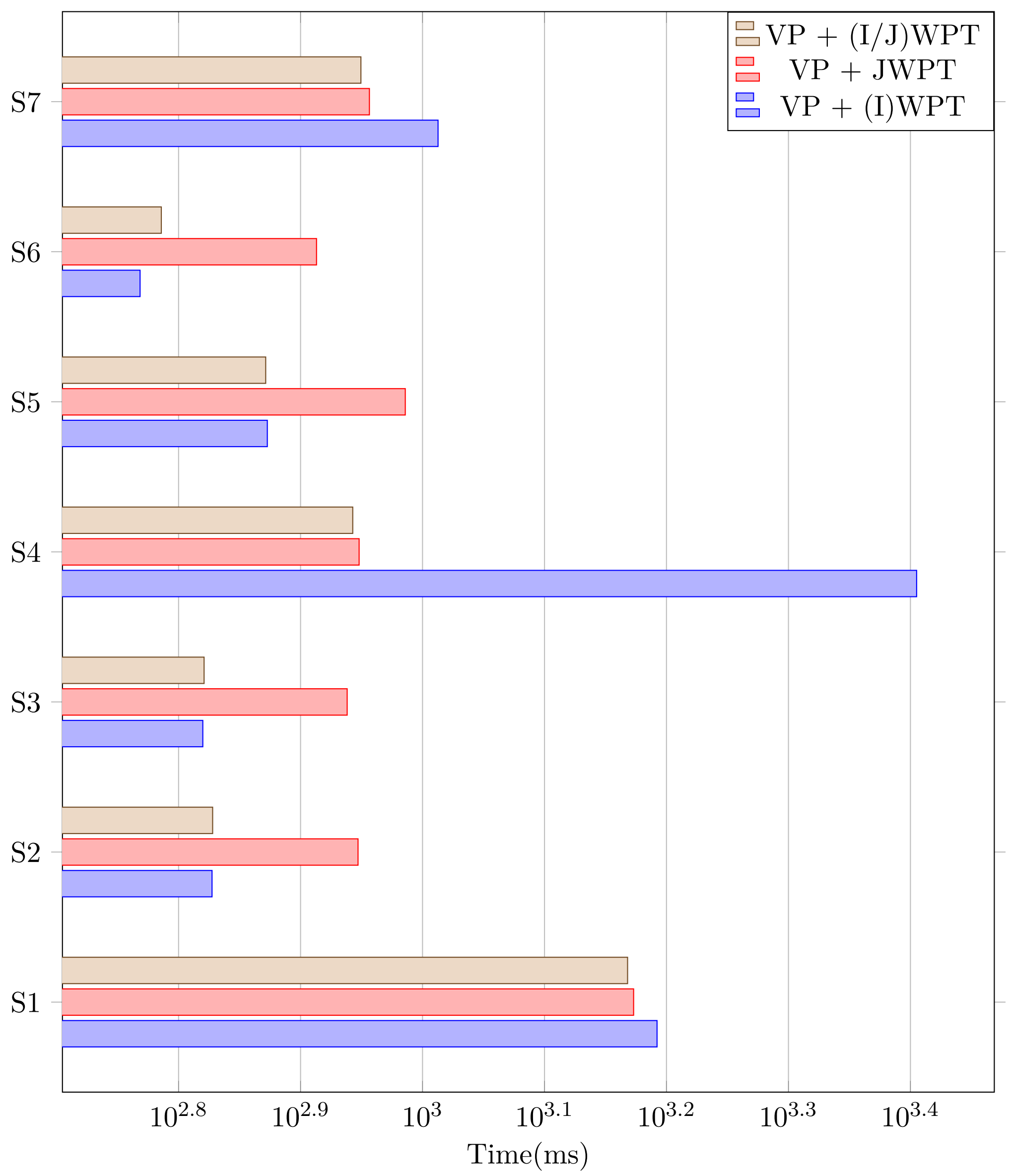


Figure 19.: Execution time of star-shaped queries

	VP+(I)WPT	VP+JWPT	VP+(I/J)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
F1-1	1603	1203	13
F1-2	1513	1270	33
F1-3	1473	1166	20
F1-4	1463	1164	5
F1-5	1447	1171	35
F2-1	1394	981	3
F2-2	1390	1014	-9
F2-3	1363	1055	21
F2-4	1382	1054	21
F2-5	1317	1084	54
F3-1	8624	-6253	-6276
F3-2	8265	-5874	-5960
F3-3	8880	-6570	-6552
F3-4	8224	-5896	-5946
F3-5	9415	-7010	-6962
F4-1	7952	-4948	-5897
F4-2	7921	-4926	-5796
F4-3	7904	-4896	-5856
F4-4	7838	-4847	-5727
F4-5	7919	-4912	-5821
F5-1	8865	-6017	-6930
F5-2	10890	-8139	-8918
F5-3	8171	-5363	-6233
F5-4	9399	-6551	-7510
F5-5	9306	-6480	-7389

Table 11.: Execution time of snow-shaped queries

	VP + (I)WPT	VP + JWPT	VP + (I/J)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
C1	2806	9025	7974
C2	23898	467554	464573
C3	2350	10	-16

Table 12.: Execution time of complex-shaped queries

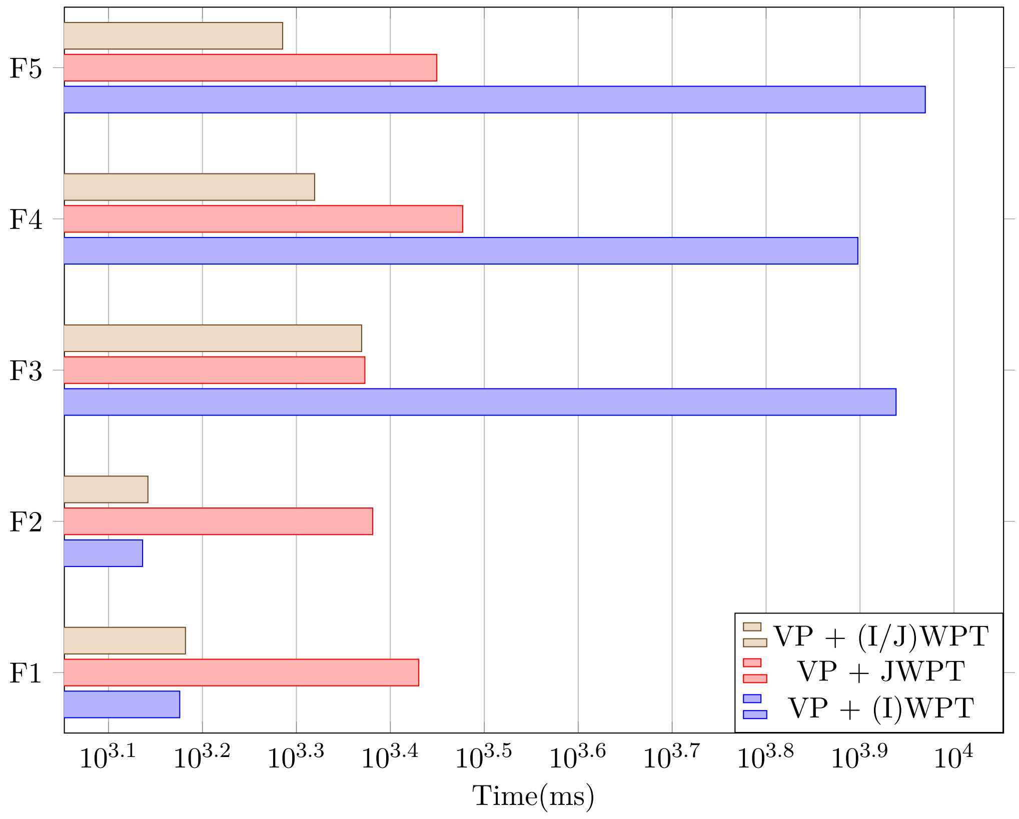
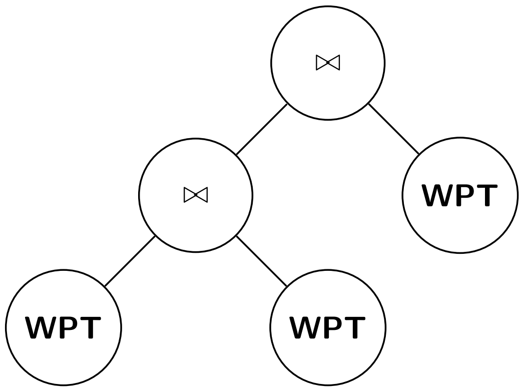
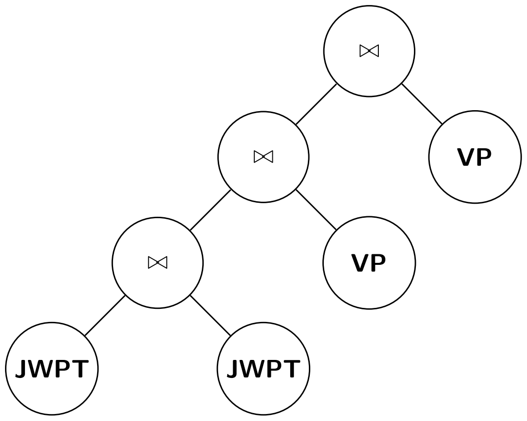


Figure 20.: Execution time of snow-shaped queries



(a) Join tree with the baseline strategy



(b) Join tree with the JWPT enabled

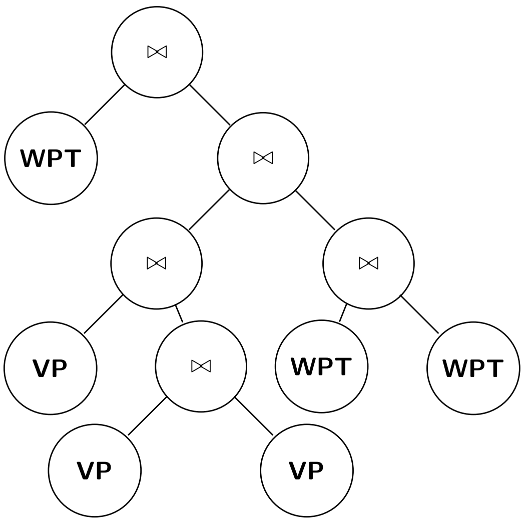
Figure 21.: Join trees created for the complex query C1 with and without using the JWPT

the JWPT, we guarantee that no triple pattern from the SPARQL query appears in more than one node. Our heuristic for the creation of JWPT tries to keep the biggest number of patterns in each node as possible. When there are multiple possible nodes with the same size, we use the available statistics to choose which node to keep and remove its triple patterns from other nodes. This heuristic not necessarily reduces the number of nodes created.

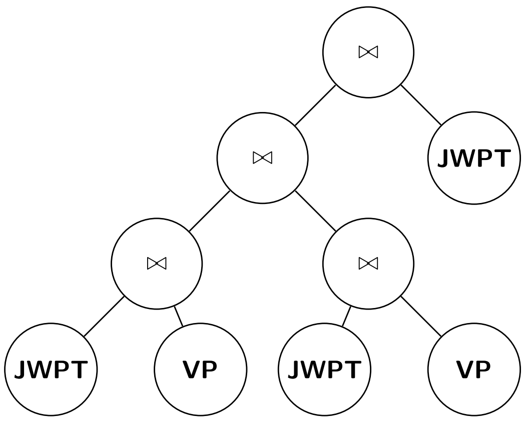
As seen in figure 21, when using the JWPT, the created join tree for query C1 ended up having more nodes, and, therefore, requires more join operations. This might have caused the slower execution. Nevertheless, the join tree created for the query C2 (figure 22) with the JWPT enabled requires one less join operation but needs on average more than seven minutes to be processed than with the baseline strategy. We can conclude from these observations that reducing the number of join operations not necessary reduces the execution time of the SPARQL, and, therefore, there are still some issues with the priority calculation and join construction in PRoST that needs to be dealt with.

The query C3 is executed with a single WPT node with the baseline strategy. Unlike what happened with the previously discussed query shapes, there was not much executing time difference when processing the query with a single JWPT node instead.

Figure 23 shows the graph with average executing time of the complex-shaped queries.



(a) Join tree with the baseline strategy



(b) Join tree with the JWPT enabled

Figure 22.: Join trees created for the complex query C2 with and without using the JWPT

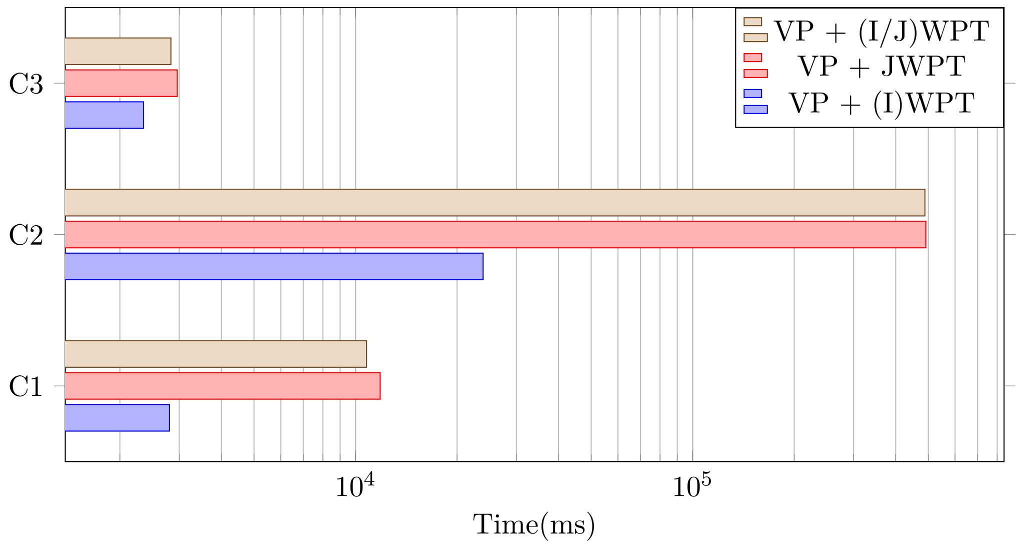


Figure 23.: Execution time of complex-shaped queries

7.5. Final Remarks

When using the JWPT makes it possible to execute a SPARQL query with fewer join operations, we managed to obtain faster processing times, with a few exceptions. Unfortunately, due to the higher number of tuples in a JWPT than in a WPT, it was not as efficient as PRoST's default mixed strategy when the number of required join operations is not reduced.

Although using only the JWPT was demonstrated to not always provide faster processing of queries, we showed that, for most cases, when using all three available property tables, we can expect to process the queries at least as efficiently as with the baseline strategy. This, naturally, comes at an increased storage cost due to the necessity of persisting three property tables instead of only one.

8. Dynamic Extended Vertical Partitioning Database Evaluation

The queryset is executed 50 times, and the times used for the evaluation are the average execution time of the queries. During each run of the benchmark, the queries are executed in random order. The dynamic ExtVP database and collected statistics are cleared after each execution of the queryset, meaning that the ExtVP tables must be created again for each run of the benchmark.

We evaluate PRoST with six different strategies, described below:

- **VP:** In this mode only VP nodes are used.
- **(Ext)VP:** Uses the dynamic ExtVp database. The dynamic ExtVP database and any statistics collected are cleared after each execution of the queryset, therefore the data collected includes the time necessary to create the ExtVP tables for each run of the queryset.
- **(Ext)VP creation:** The ExtVP tables are created and stored, but the dynamic ExtVP database is cleared after each query is processed. This means that there is never an available ExtVP table in the database to be used in a query.
- **VP + (I)WPT:** This is PRoST's default strategy, where WPT, IWPT, and VP nodes are used.
- **(Ext)VP + (I)WPT:** Uses WPT and IWPT, and dynamically creates ExtVP tables from VP nodes when possible. Again, after each execution of the queryset, the ExtVP database and its statistics are cleared.
- **(Ext)VP creation + (I)WPT:** The WPT, IWPT, and dynamic ExtVP database are used. Created ExtVP tables are deleted after each query is executed, therefore there is never an available ExtVP table in the database from previously processed queries.

The *(Ext)VP creation* mode is used to evaluate the extra time necessary to extract and save the ExtVP tables in the database. This mode also reflects the worst case scenario where there are never available ExtVP tables to be used to process a query.

With these chosen strategies we want to analyze how effective the use of the dynamic ExtVP database is when compared with the use of only VP nodes, as well as the mixed strategy (VP and (I)WPT nodes). Furthermore, we also evaluate how costly the computation and storing of the ExtVP tables is.

For each query shape, we show two tables, one for the cases where only vertically partitioned tables and the dynamic ExtVp database are used, and another for the cases where the property tables are also used. The modes that do not use the ExtVP database are considered the baseline, and we display the execution time difference of the other modes compared to those baselines. Furthermore, if that difference is statistically significant, with $\alpha = 0.05$, we display it in a colored cell (blue if the query was processed faster, and red otherwise).

We also display a graph where the execution time of each mode can be more easily visualized. The graph is in a logarithm scale. For simplicity, in this graph, we group similar queries and display their average execution time. The difference between queries in the same group is only the constant values in their triple patterns.

8.1. Linear-Shaped Queries

In table 13 we show the execution times of linear-shaped queries when using only vertically partitioned tables and the dynamic ExtVP database.

As can be observed, the execution time when using the dynamic ExtVP database was consistently faster than when using only VP nodes (on average around 5 seconds faster). Not only that, but there was not any noticeable increase in the execution time for queries when the ExtVP tables still need to be extracted and persisted. This can be justified by the small number of join operations necessary to process these queries, as they have between two and three triple patterns only, and consequently, the small number of ExtVP tables created during the execution of each query.

Overall, we can conclude that there is no disadvantage in using the dynamic ExtVP databases with this query-shape, since there is a significant performance increase when ExtVP tables are used, and even if the created ExtVP tables would not be used again, their computation does not cause a significant performance hit.

In table 14 we show the execution time for the cases where the IWPT and WPT nodes are enabled. Unfortunately, there is not a noticeable performance increase

	VP	(Ext)VP	(Ext)VP creation
Query	Average (ms)	Difference (ms)	Difference
L1-1	16574	-13918	-324
L1-2	16426	-13631	-1246
L1-3	15423	-13808	-1126
L1-4	13113	-186	-182
L1-5	15511	-13557	-1036
L2-1	3962	-1800	-320
L2-2	3766	-1468	-284
L2-3	3740	-1307	-57
L2-4	3827	-1394	-213
L2-5	4121	-576	1681
L3-1	7324	-5355	-1000
L3-2	7565	-6852	-1221
L3-3	3062	-1216	1032
L3-4	3112	-1174	1726
L3-5	3111	-1012	101
L4-1	3295	-2285	606
L4-2	3324	-2259	524
L4-3	3708	-2279	-252
L4-4	3275	-335	92
L4-5	3446	-2396	-128
L5-1	10552	-8370	510
L5-2	11935	-9687	-1119
L5-3	10263	-7682	-723
L5-4	9945	-6219	1316
L5-5	10344	-7956	936

Table 13.: Execution time of linear-shaped queries with VP nodes and dynamic ExtVP

between these cases. As can be seen in the linear-shaped SPARQL queries available in the appendix A.1, most queries with this shape can be processed with a single WPT node or a join between a WPT node and a VP node, therefore the ExtVP database is not used for these queries when the (I)WPT nodes are enabled.

In figure 24 we show the graph with the average execution times of each linear-shaped query group.

8.2. Star-Shaped Queries

Table 15 shows the processing time of star-shaped queries when using only VP nodes and the dynamic ExtVP database.

Again, using the dynamic ExtVP database was consistently faster than using only VP nodes, being on average approximately six seconds faster. Since more ExtVP tables are created during the execution of these queries than for linear-shaped queries, there was an increase in the overall time necessary to extract and store ExtVP tables, although that increase was not statistically significant for all queries. On average, there was an increase of one second in the execution time when ExtVP tables are created.

When the use of WPT and IWPT nodes is enabled the dynamic ExtVP database had no impact on the processing time of queries, as can be seen in table 16. Just as it happened with the linear-shaped queries, the star-shaped queries can all be executed with either a single WPT node or a join tree that consists of a join between one WPT node and one VP node.

The graph with the execution time of star-shaped queries with all strategies is available in figure 25.

8.3. Snow-Shaped Queries

As can be observed in the sample SPARQL snow-shaped queries available in the appendix A.3, queries with this shape have a higher number of triple patterns than the previously discussed types of queries. Consequently, these queries require more join operations to be processed and benefited the most from using the dynamic ExtVp database, being processed on average around 8.5 seconds faster. Naturally, since more ExtVP tables are created during the execution of a single query, there is also an increased execution time of almost 4 seconds on average when new ExtVP tables need to be created. The average processing times using only VP nodes and

	VP + (I)WPT	(Ext)VP + (I)WPT	(Ext)VP creation + (I)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
L1-1	3238	241	333
L1-2	3108	398	292
L1-3	3214	299	87
L1-4	3245	-142	-226
L1-5	3047	470	249
L2-1	848	-6	6
L2-2	841	-10	-14
L2-3	843	-99	68
L2-4	838	0	30
L2-5	1446	-140	-203
L3-1	707	20	35
L3-2	624	29	37
L3-3	657	-32	-35
L3-4	633	-6	-16
L3-5	632	-25	-37
L4-1	561	-22	27
L4-2	518	-4	-1
L4-3	502	19	6
L4-4	509	-3	3
L4-5	511	3	10
L5-1	631	26	17
L5-2	632	10	13
L5-3	582	-2	-18
L5-4	606	4	13
L5-5	659	2	-39

Table 14.: Execution time of linear-shaped queries with mixed strategy and dynamic ExtVP

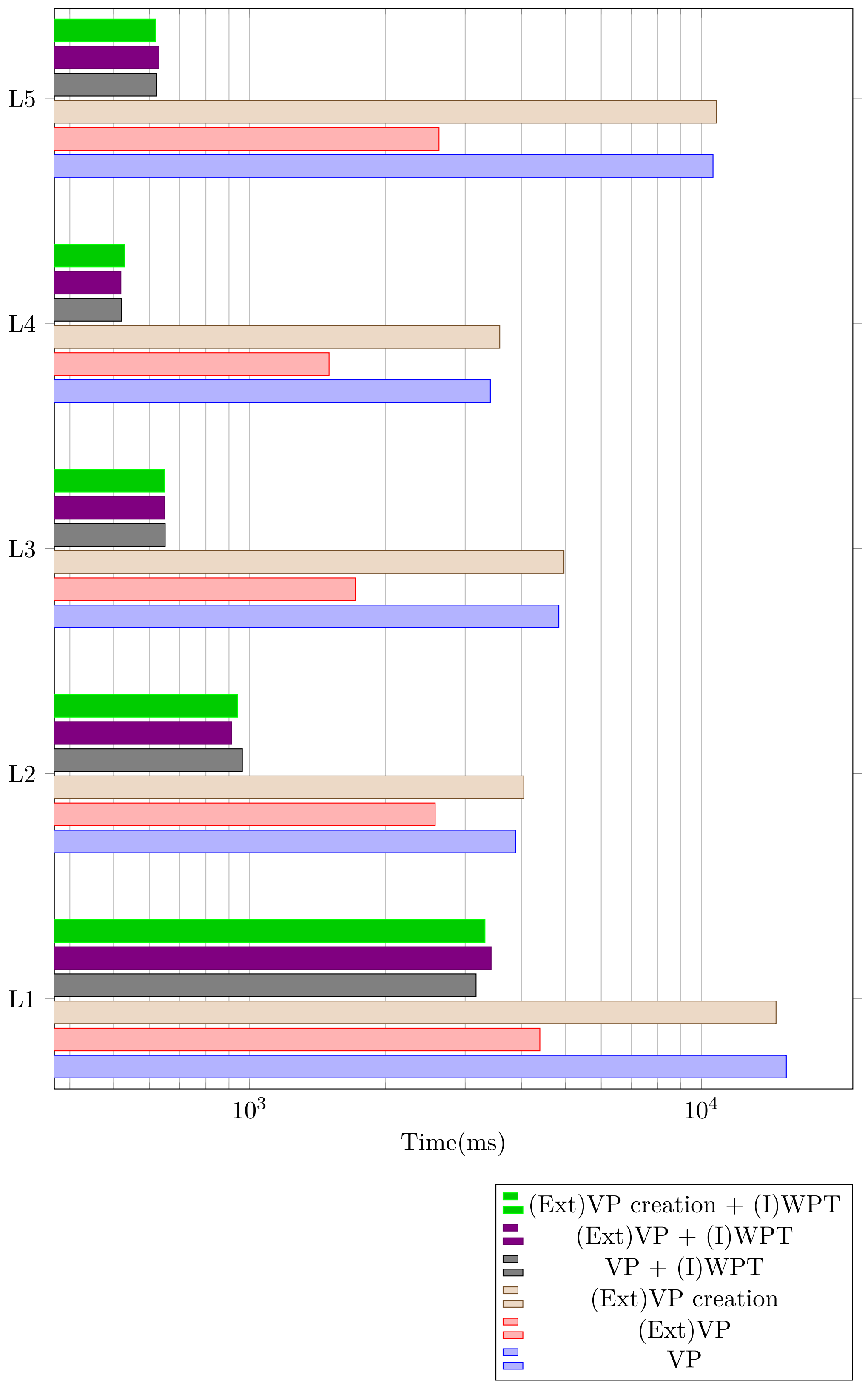


Figure 24.: Execution time of linear-shaped queries

	VP	(Ext)VP	(Ext)VP creation
Query	Average (ms)	Difference (ms)	Difference
S1-1	25694	-9346	1378
S1-2	25821	-9439	606
S1-3	25408	-9306	770
S1-4	27672	3376	-774
S1-5	26443	-9149	181
S2-1	14328	-11455	-1494
S2-2	13267	-10069	-1247
S2-3	14133	-10855	-2571
S2-4	13479	-3819	-942
S2-5	13414	-10278	-725
S3-1	9924	-7954	1972
S3-2	11768	-9848	81
S3-3	9959	-8048	1452
S3-4	11019	2691	1146
S3-5	10917	-8445	880
S4-1	9416	-6549	1
S4-2	9800	-7632	-30
S4-3	8715	-1478	82
S4-4	9431	-6958	-1033
S4-5	9424	-7236	-759
S5-1	8442	3624	2248
S5-2	9628	-6606	938
S5-3	8632	-5536	2048
S5-4	9441	-6846	1534
S5-5	9163	-5920	1446
S6-1	7387	-4969	2065
S6-2	7890	-5335	1958
S6-3	8097	-5576	1002
S6-4	7507	-5124	2073
S6-5	7985	4589	2735
S7-1	9285	5101	2476
S7-2	11584	-7438	766
S7-3	10170	-6627	3245
S7-4	10637	-5625	2031
S7-5	10213	-5838	1558

Table 15.: Execution time of star-shaped queries with VP nodes and dynamic ExtVP

	VP + (I)WPT	(Ext)VP + (I)WPT	(Ext)VP creation + (I)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
S1-1	1510	23	-19
S1-2	1591	-40	6
S1-3	1677	-23	-5
S1-4	1558	-48	4
S1-5	1453	39	28
S2-1	732	6	1
S2-2	635	16	-6
S2-3	660	19	11
S2-4	655	19	4
S2-5	680	18	-24
S3-1	656	12	-20
S3-2	643	13	-13
S3-3	676	10	-26
S3-4	624	28	-25
S3-5	705	-6	-24
S4-1	2547	-261	240
S4-2	2688	1394	29
S4-3	2040	224	292
S4-4	2743	-379	-157
S4-5	2696	1179	79
S5-1	757	1	-3
S5-2	732	1	-16
S5-3	775	-38	-30
S5-4	705	11	11
S5-5	763	-1	-18
S6-1	587	24	49
S6-2	614	-6	-57
S6-3	585	-7	-21
S6-4	561	35	-2
S6-5	588	4	-27
S7-1	983	10	29
S7-2	1002	-8	45
S7-3	1108	-33	-41
S7-4	1051	-73	8
S7-5	1007	-50	14

Table 16.: Execution time of star-shaped queries with mixed strategy and dynamic ExtVP

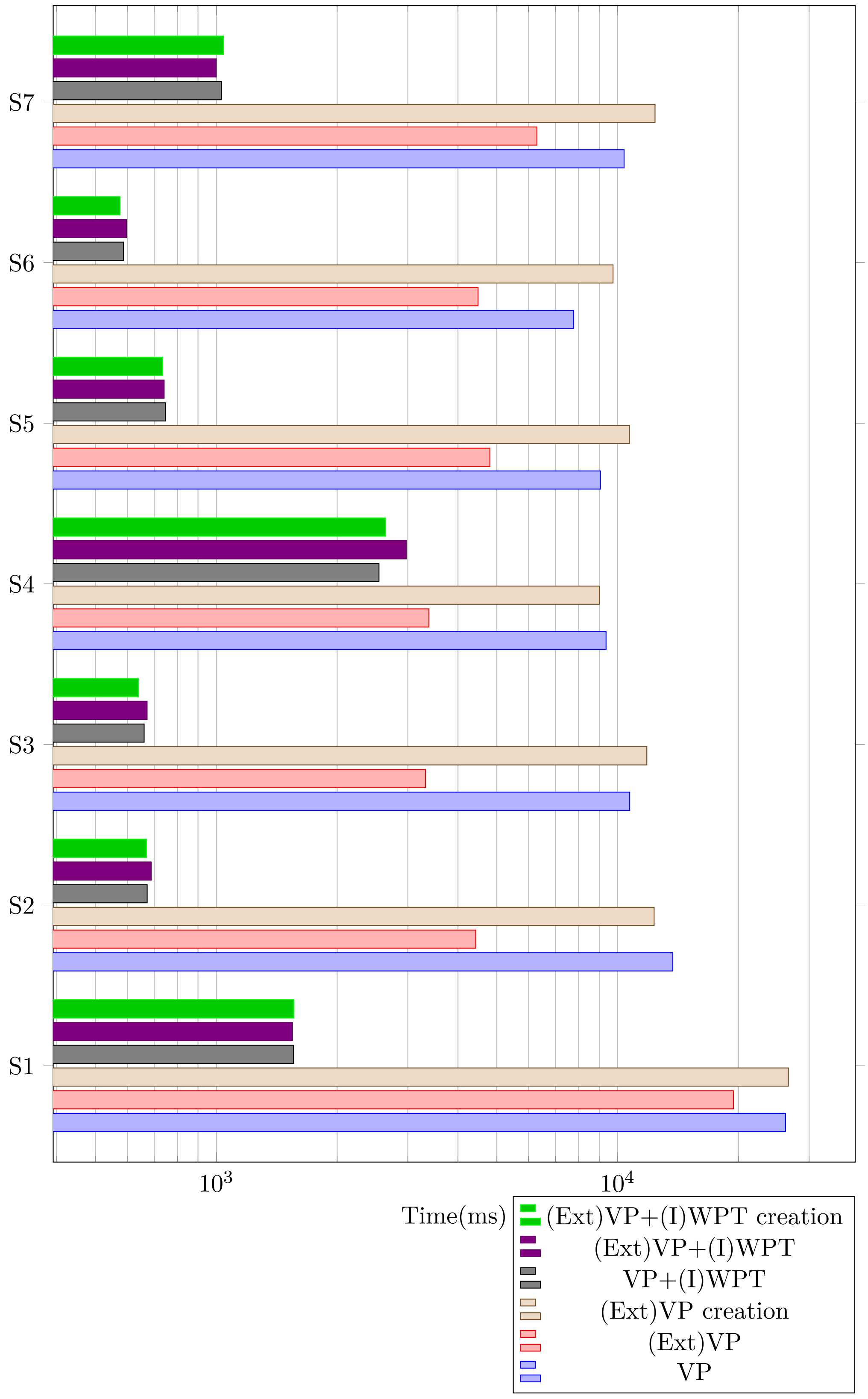


Figure 25.: Execution time of star-shaped queries

the dynamic ExtVP database are shown in table 17.

The execution times when the use of property tables is enabled is shown in table 18. Similarly to the previous cases, there is not a significant difference between the baseline and using the dynamic ExtVP database, since the join trees created to process these queries do not contain joins between VP nodes.

The graph with the average execution time of the snow-shaped query groups is shown in figure 26.

8.4. Complex-Shaped Queries

The last queries analyzed are complex-shaped queries. The shape of these queries can be seen as the combination of multiple subqueries with the previously discussed query shapes. The execution times without using property tables is shown in table 19.

The query C3 showed the most promising result, being processed on average over 47 seconds faster. Similarly to snow-shaped queries, there is a big number of possible ExtVP tables for each query, and accordingly there is a bigger cost to execute these queries when we still need to extract and store these tables.

Table 20 shows the executing time of the complex-shaped queries when using property tables. From the three complex-shaped queries present in the queryset, only the query C2 will result in the creation of ExtVP tables. The two ExtVp tables that are used to process that query were already enough to speed its processing time by an average of more than 4 seconds.

Figure 27 shows the graph with average execution time of these queries.

8.5. Final Remarks

When compared to the S2RDF system, the dynamic ExtVP database has a clear advantage when considering the loading time necessary to initialize an RDF dataset. The benchmark dataset (containing around 100 million triples) is loaded in approximately 30 minutes by PRoST, whereas S2RDF required over 16 hours. This happens because S2RDF computes all possible ExtVP tables during the loading phase.

S2RDF computed a total of 22 102 ExtVP tables, whereas PRoST computed only 72 tables with the dynamic ExtVP database. Naturally, computing ExtVP during the processing of queries comes at a cost, but this cost is negligible when executing smaller queries, and it only happens once for each possible ExtVP table.

	VP	(Ext)VP	(Ext)VP creation
Query	Average (ms)	Difference (ms)	Difference
F1-1	15129	-10823	1143
F1-2	16724	-12826	192
F1-3	13111	4569	1825
F1-4	14741	-9868	1273
F1-5	14907	-10471	-1338
F2-1	22439	-12559	5002
F2-2	22417	-12160	5263
F2-3	20648	13226	5498
F2-4	21023	-12093	2986
F2-5	22271	-14716	3723
F3-1	25379	-9402	2771
F3-2	26681	-6247	2671
F3-3	23956	-13458	5125
F3-4	24281	8818	5806
F3-5	26987	-8649	2167
F4-1	22048	-12260	3581
F4-2	20810	-6388	4966
F4-3	22235	-7382	5228
F4-4	21505	-7388	5402
F4-5	22361	-8172	4599
F5-1	23939	-1915	4838
F5-2	24630	-15829	5806
F5-3	24460	-15872	5068
F5-4	25098	-15544	5682
F5-5	23615	-13146	5679

Table 17.: Execution time of snow-shaped queries with VP nodes and dynamic ExtVP

	VP + (I)WPT	(Ext)VP + (I)WPT	(Ext)VP creation + (I)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
F1-1	1603	45	-2
F1-2	1513	37	-22
F1-3	1473	-7	-14
F1-4	1463	28	-11
F1-5	1447	15	-22
F2-1	1394	-33	-53
F2-2	1390	-39	51
F2-3	1363	33	11
F2-4	1382	-28	-37
F2-5	1317	3	-3
F3-1	8624	694	89
F3-2	8265	570	-111
F3-3	8880	-300	-232
F3-4	8224	436	79
F3-5	9415	109	-592
F4-1	7952	-207	-206
F4-2	7921	17	-167
F4-3	7904	-5	-29
F4-4	7838	109	87
F4-5	7919	-40	-102
F5-1	8865	-1309	-1103
F5-2	10890	-1581	-1457
F5-3	8171	61	375
F5-4	9399	-208	-158
F5-5	9306	-647	-438

Table 18.: Execution time of snow-shaped queries with mixed strategy and dynamic ExtVP

	VP	(Ext)VP	(Ext)VP creation
Query	Average (ms)	Difference (ms)	Difference (ms)
C1	26531	-1538	7891
C2	34124	-2486	10067
C3	84718	-47502	20918

Table 19.: Execution time of complex-shaped queries with VP nodes and dynamic ExtVP

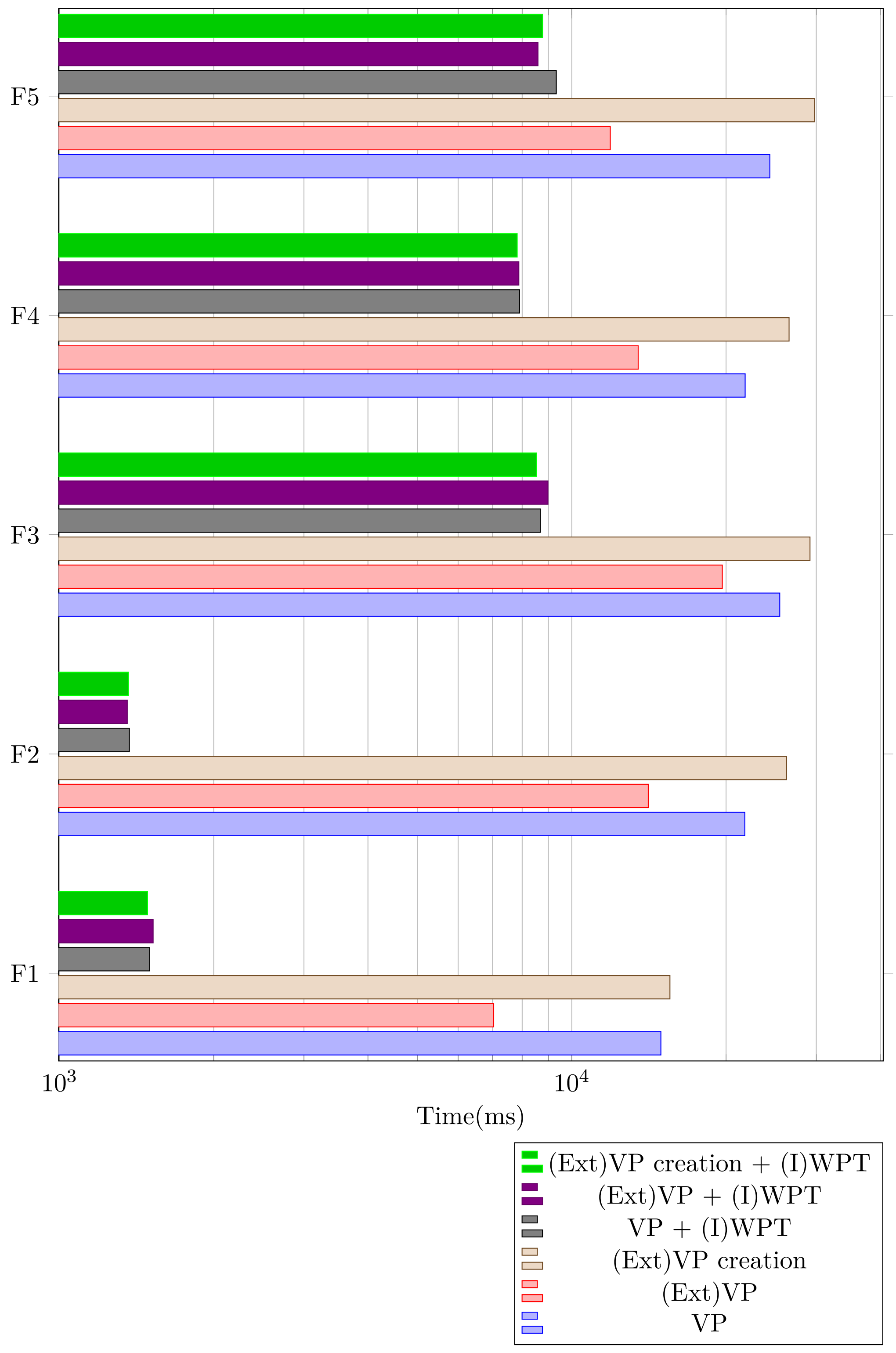


Figure 26.: Execution time of snow-shaped queries

	VP + (I)WPT	(Ext)VP + (I)WPT	(Ext)VP creation + (I)WPT
Query	Average (ms)	Difference (ms)	Difference (ms)
C1	2806	-23	-237
C2	23898	-4575	11325
C3	2350	-49	62

Table 20.: Execution time of complex-shaped queries with mixed strategy and dynamic ExtVP

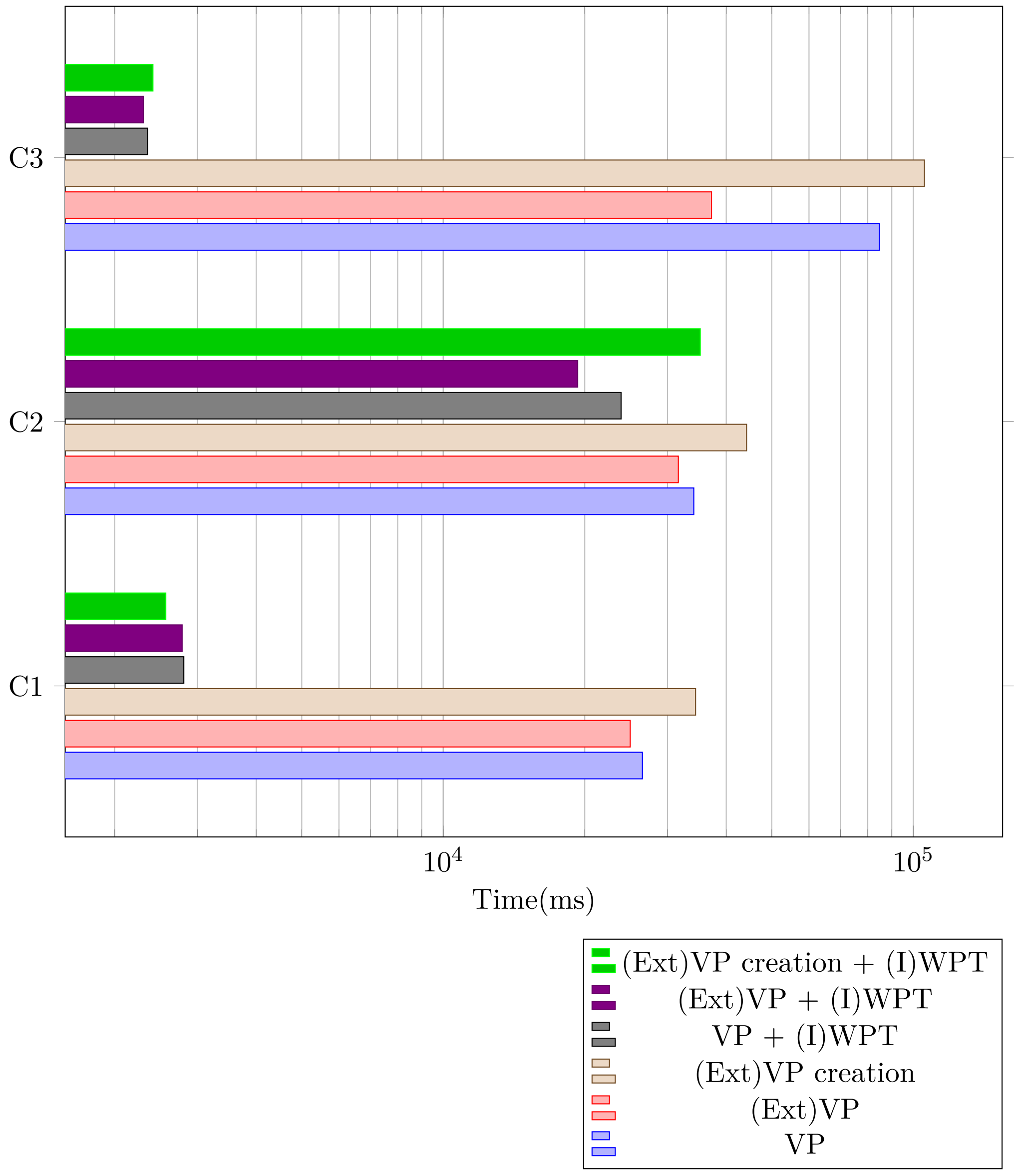


Figure 27.: Execution time of complex-shaped queries

Furthermore, there is also the storage space cost to consider, as many of the ExtVP tables created and stored by S2RDF might never be used to process a query.

Unfortunately, the use of the dynamic ExtVP database did not provide any performance increase to the already existing mixed strategy in PRoST (using both vertically partitioned tables and wide property tables), although we showed that there is a significant performance increase when compared to the use of only vertically partitioned tables. We can expect to be able to obtain similar performance increase if we dynamically extract and persists join reduction tables obtained after join operations between the other node types used in PRoST.

9. Conclusion and Future Work

PRoST is an RDF system that is capable of using multiple supported data models to process SPARQL queries. This is achieved by representing queries in the form of a tree structure, called a join tree. Each leaf node of a join tree is a smaller part of the query that can be processed using one of the available data models in PRoST, independently of the other leaf nodes. An inner node represents a join operation between its children. We added support in PRoST for two new data models.

PRoST previously offered support for two types of property tables. The first, called a wide property table, is suitable to process subqueries with triple patterns containing a common subject resource. Similarly, the second property table, called an inverse wide property table, is suitable to process subqueries with triple patterns containing a common object resource.

We added support to a new type of property table, called joined wide property table, that is obtained by joining both the WPT and IWPT on their common resources. With this property table, it becomes possible to process subqueries where the common resource of their triple patterns can be a subject and an object. Since, by using a joined wide property table, a higher number of triple patterns from a SPARQL query can be processed without executing a join operation, this data model can potentially reduce the total number of join operations required to process a SPARQL query.

Unfortunately, although we managed to obtain faster query processing times in PRoST when the use of the joined wide property tabled reduced the number join operations required to execute a SPARQL query, the joined wide property table had a lower performance than the wide property table when executing queries where the common resource is only a subject resource of its triple patterns. This most likely occurs because a JWPT has a higher number of tuples. This occurs because the joined wide property table was implemented so that it can be the only property table needed by PRoST. As a future improvement to PRoST, the joined wide property table could be modified to only include the data of resources that are both subject and objects in the RDF graph. This table would then only be used when the triple patterns of the queries require a join of the type subject-object. Join operations of the type subject-subject and object-object would demand the original WPT and IWPT.

We also added support to the Extended Vertical Partitioning data model. This model was first introduced in the S2RDF system and showed a big improvement when compared to the use of only vertically partitioned tables. In the S2RDF system, the ExtVP tables are created during the loading of an RDF dataset, which is a time-consuming process due to the potentially high number of ExtVP tables that need to be computed. Furthermore, many of these precomputed ExtVP tables are discarded or might never be used to process queries. In our implementation of ExtVP, we create the ExtVP tables dynamically from intermediate results obtained during the normal execution of queries in PRoST.

Naturally, creating ExtVP tables during query processing time introduces overhead to its execution. We demonstrated that this overhead is rather small, and it is easily compensated by the faster processing of queries acquired when using ExtVP tables. We showed how using our dynamic ExtVp database provided better performance when executing all queries from our benchmark when compared to using only the static vertically partitioned tables.

Regrettably, we did not obtain a noticeable improvement when compared to the use of a mixed strategy, that is, the use of both the property tables and the vertically partitioned tables. As further development to be made in PRoST, we propose the persistence of semi-join reductions obtained from the intermediate results from join operations between any of PRoST's join tree node types. Since we demonstrated that the dynamic extraction and persistence of semi-join reductions from VP nodes give a noticeable improvement to the query execution time when compared to the static case, and the tables creation and storage is not costly, we expect that such improvements can also be achievable with any node type.

Bibliography

[1] "Dbpedia." https://wiki.dbpedia.org/. Accessed: 2019-02-24.

[2] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette, "Bio2rdf: towards a mashup to build bioinformatics knowledge systems," *Journal of biomedical informatics*, vol. 41, no. 5, pp. 706–716, 2008.

[3] M. Cossu, M. Färber, and G. Lausen, "Prost: Distributed execution of sparql queries using mixed partitioning strategies," *arXiv preprint arXiv:1802.05898*, 2018.

[4] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2rdf: Rdf querying with sparql on spark," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 804–815, 2016.

[5] "Rdf 1.1." https://www.w3.org/TR/rdf11-concepts/. Accessed: 2019-02-24.

[6] "Sparql 1.1." https://www.w3.org/TR/sparql11-overview/. Accessed: 2019-02-24.

[7] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[8] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[9] K. Wilkinson and K. Wilkinson, "Jena property table implementation," 2006.

[10] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Sw-store: a vertically partitioned dbms for semantic web data management," *The VLDB Journal*, vol. 18, no. 2, pp. 385–406, 2009.

[11] "Apache parquet." `https://parquet.apache.org/`. Accessed: 2019-02-24.

[12] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of rdf data management systems," in *International Semantic Web Conference*, pp. 197–212, Springer, 2014.

[13] A. Madkour, A. M. Aly, and W. G. Aref, "Worq: Workload-driven rdf query processing," in *International Semantic Web Conference*, pp. 583–599, Springer, 2018.

A. WatDiv Queries

A.1. Linear-Shaped Queries

L1:

```
SELECT ?v0 ?v2 ?v3 WHERE {
        ?v0     wsdbm:subscribes     wsdbm:Website7355     .
        ?v2     sorg:caption         ?v3                   .
        ?v0     wsdbm:likes          ?v2                   .
}
```

L2:

```
SELECT ?v1 ?v2 WHERE {
        wsdbm:City70    gn:parentCountry     ?v1               .
        ?v2             wsdbm:likes          wsdbm:Product0    .
        ?v2             sorg:nationality     ?v1               .
}
```

L3:

```
SELECT ?v0 ?v1 WHERE {
        ?v0     wsdbm:subscribes     wsdbm:Website43164    .
        ?v0     wsdbm:likes          ?v1                   .
}
```

L4:

```
SELECT ?v0 ?v2 WHERE {
        ?v0     og:tag          wsdbm:Topic142    .
        ?v0     sorg:caption    ?v2               .
}
```

L5:

```
SELECT ?v0 ?v1 ?v3 WHERE {
```

```
          ?v0               sorg:jobTitle          ?v1  .
          wsdbm:City40      gn:parentCountry       ?v3  .
          ?v0               sorg:nationality       ?v3  .
}
```

A.2. Star-Shaped Queries

S1:

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE {
          ?v0                        gr:includes            ?v1  .
          wsdbm:Retailer8535         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  .
}
```

S2:

```
SELECT ?v0 ?v1 ?v3 WHERE {
          ?v0      dc:Location              ?v1                .
          ?v0      sorg:nationality         wsdbm:Country4     .
          ?v0      wsdbm:gender             ?v3                .
          ?v0      rdf:type                 wsdbm:Role2        .
}
```

S3:

```
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
          ?v0      rdf:type          wsdbm:ProductCategory1    .
          ?v0      sorg:caption      ?v2                       .
          ?v0      wsdbm:hasGenre    ?v3                       .
          ?v0      sorg:publisher    ?v4                       .
}
```

S4:

```
SELECT ?v0 ?v2 ?v3 WHERE {
            ?v0      foaf:age                  wsdbm:AgeGroup5  .
            ?v0      foaf:familyName           ?v2              .
            ?v3      mo:artist                 ?v0              .
            ?v0      sorg:nationality          wsdbm:Country1   .
}
```

S5:

```
SELECT ?v0 ?v2 ?v3 WHERE {
            ?v0      rdf:type                  wsdbm:ProductCategory7  .
            ?v0      sorg:description          ?v2                     .
            ?v0      sorg:keywords             ?v3                     .
            ?v0      sorg:language             wsdbm:Language0         .
}
```

S6:

```
SELECT ?v0 ?v1 ?v2 WHERE {
            ?v0      mo:conductor     ?v1                       .
            ?v0      rdf:type         ?v2                       .
            ?v0      wsdbm:hasGenre   wsdbm:SubGenre130         .
}
```

S7:

```
SELECT ?v0 ?v1 ?v2 WHERE {
            ?v0               rdf:type        ?v1  .
            ?v0               sorg:text       ?v2  .
            wsdbm:User54768   wsdbm:likes     ?v0  .
}
```

A.3. Snowflake-Shaped Queries

F1:

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
            ?v0      og:tag           wsdbm:Topic49      .
            ?v0      rdf:type         ?v2                .
            ?v3      sorg:trailer     ?v4                .
            ?v3      sorg:keywords    ?v5                .
```

```
        ?v3     wsdbm:hasGenre   ?v0                      .
        ?v3     rdf:type         wsdbm:ProductCategory2   .
}
```

F2:

```
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       wsdbm:SubGenre117       .
}
```

F3:

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
        ?v0     sorg:contentRating   ?v1                     .
        ?v0     sorg:contentSize     ?v2                     .
        ?v0     wsdbm:hasGenre       wsdbm:SubGenre111       .
        ?v4     wsdbm:makesPurchase  ?v5                     .
        ?v5     wsdbm:purchaseDate   ?v6                     .
        ?v5     wsdbm:purchaseFor    ?v0                     .
}
```

F4:

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
        ?v0     foaf:homepage        ?v1               .
        ?v2     gr:includes          ?v0               .
        ?v0     og:tag               wsdbm:Topic122    .
        ?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               .
}
```

F5:

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
        ?v0     gr:includes                          ?v1 .
        wsdbm:Retailer9885      gr:offers            ?v0 .
        ?v0     gr:price                             ?v3 .
        ?v0     gr:validThrough                      ?v4 .
        ?v1     og:title                             ?v5 .
        ?v1     rdf:type                             ?v6 .
}
```

A.4. Complex-Shaped Queries

C1:

```
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 .
}
```

C2:

```
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       .
}
```

C3:

```
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  .
}
```