UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Farid Valiyev

Query Workload-Driven Schema Optimization For Processing Large RDF Datasets

Master's Thesis (30 ECTS)

Supervisor(s): Dr. Mohamed Ragab , WAIS Lab, Southampton University, UK
Ass.Prof. Riccardo Tommasini, LIRIS Lab, INSA de Lyon, France
Ass.Prof. Alexander Nolte, University of Tartu, Estonia

Tartu 2023

Query Workload-Driven Schema Optimization For Processing Large RDF Datasets

Abstract:
In the world we live in, data are not only increasing in volume, but they are also becoming more and more interconnected and linked. In many areas of our daily lives, such as social media, computational biology and protein networks, telecommunications, and many others, graph data models are the most natural, easy-to-understand, and versatile data abstraction to represent the world's structured knowledge. In fact, the information retrieved via natural language processing and computer vision is currently being represented mostly by Knowledge Graphs (KGs). KGs are efficient means to represent, integrate and connect data from several heterogeneous data sources. Those applications led to a surge in the popularity of KGs. However, on the other side, this brings computational challenges because KGs are growing in massive volumes. Specifically, several applications have used the standard Resource Description Framework (RDF) graph data model to represent, share, and integrate pieces of data on the web.

Therefore, the Semantic Web (SW) community's central problem for managing scalable RDF KGs is now in demand. The native graph databases (e.g., Apache Jena, RDF-3X, and Virtuoso) fall short of managing and processing large RDF datasets due to their centralized computational paradigm, i.e., they cannot scale out. Thus, the SW community has started to investigate relational Big Data (BD) frameworks harnessing their scalability and efficiency. Relational systems get a lot of their efficient performance from sophisticated optimizers that leverage relational model, relational algebra simplicity, and maturity. Despite the flexibility of the relational solutions, the flexible (i.e., schema-less) structure of RDF graphs brings challenges to store and manage RDF graphs in relational schemas. The state-of-the-art shows that there is no "One-Size-Fits-All" RDF relational schema that can fit all the query workloads. In particular, there is a different winner of RDF relational schema by a large margin for each query type, and the winner in one query family may unexpectedly perform the worst in another.

In this thesis, we argue that combining multiple RDF relational schemas to attain a hybrid one provides better performance for the BD system while querying large KGs. Nevertheless, designing hybrid schema solutions for schema-less KGs require huge data engineering efforts and tailored solutions. To this end, this thesis proposes algorithms that automatically design a hybrid RDF relational schema that adapts to the query workload covering a wide range of query types, without ignoring the loading times, as well as the storage overheads. In particular, we approach this goal via data profiling along with query profiling seeking better data localization andcombining relevant data that frequently queried together on the same relations. Our approach reaches to an optimal hybrid schema that consider both the underlying data relationships, as well as the query

workloads.

Keywords: Large RDF Graphs, SPARQL, Spark-SQL, RDF Relational Schema, Workload Driven

CERCS: P170 - Computer Science, numerical analysis, systems, control

Päringukoormusele suunatud skeemi optimeerimine suurte RDF-andmekogumite töötlemiseks

Lühikokkuvõte:
Maailmas, kus me elame, ei suurene andmete maht mitte ainult, vaid need on ka üha enam omavahel seotud ja lingitud. Paljudes meie igapäevaelu valdkondades, nagu sotsiaalmeedia, arvutusbioloogia ja valguvõrgud, telekommunikatsioon ja paljud teised, on graafikute andmemudelid kõige loomulikum, hõlpsamini mõistetav ja mitmekülgsem andmeabstraktsioon, mis esindab maailma struktureeritud teadmisi. Tegelikult esindavad loomuliku keele töötlemise ja arvutinägemise kaudu hangitud teavet praegu peamiselt teadmiste graafikud (KG-d).

KG-d on tõhusad vahendid mitmest heterogeensest andmeallikast pärit andmete esitamiseks, integreerimiseks ja ühendamiseks. Need rakendused tõid kaasa KG-de populaarsuse tõusu. Kuid teisest küljest toob see kaasa arvutuslikke väljakutseid, kuna KG-de maht kasvab tohutult. Täpsemalt on mitmed rakendused kasutanud standardset Resource Description Framework (RDF) graafiku andmemudelit, et esitada, jagada ja integreerida veebis olevaid andmeid.

Seetõttu on nüüd nõutud semantilise veebi (SW) kogukonna keskne probleem skaleeritavate RDF-i KG-de haldamisel. Natiivsed graafikuandmebaasid (nt Apache Jena, RDF-3X ja Virtuoso) ei suuda oma tsentraliseeritud arvutusparadigma tõttu suuri RDF-andmekogumeid hallata ja töödelda, st neid ei saa skaleerida. Seega hakkab SW kogukond uurima relatsioonilisi suurandmete (BD) raamistikke, kasutades nende mastaapsust ja tõhusust. Relatsioonisüsteemid saavad suure osa oma tõhusast jõudlusest tänu keerukatele optimeerijatele, mis kasutavad relatsioonimudelit, relatsioonialgebra lihtsust ja küpsust. Vaatamata relatsioonilahenduste paindlikkusele, pakub RDF-graafiku paindlik (st skeemivaba) struktuur väljakutseid RDF-graafikute salvestamisel ja haldamisel relatsiooniskeemides. Kaasaegne tehnika näitab, et pole olemas ühtset RDF-i relatsiooniskeemi, mis sobiks kõigile päringukoormustele. Eelkõige on iga päringutüübi puhul erinev RDF-i relatsiooniskeemi võitja ja ühe päringuperekonna võitja võib ootamatult teises osas kõige halvemini toimida.

Selles lõputöös väidame, et mitme RDF-i relatsiooniskeemi kombineerimine hübriidskeemi saamiseks tagab BD-süsteemi parema jõudluse suurte KG-de päringute tegemisel. Sellegipoolest nõuab skeemita KG-de jaoks hübriidskeemilahenduste kavandamine to-

hutuid andmetehnilisi jõupingutusi ja kohandatud lahendusi. Sel eesmärgil pakub see lõputöö välja algoritme, mis kujundavad automaatselt hübriidse RDF-i relatsiooniskeemi, mis kohandub päringu töökoormusega, hõlmates paljusid päringutüüpe, jätmata tähelepanuta laadimisaegu ja salvestuskulusid. Eelkõige läheneme sellele eesmärgile andmeprofiilide koostamise ja päringute profileerimisega, et otsida paremat andmete lokaliseerimist, kombineerides asjakohaseid andmeid, mida sageli samade seoste kohta päritakse. Meie lähenemisviis ulatub optimaalse hübriidskeemini, mis võtab arvesse nii aluseks olevaid andmesuhteid kui ka päringu töökoormust.

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Graph-based data models such as RDF (Resource Description Framework) have shown great potential for managing interconnected data[Los22]. In particular RDF, which stands for Resource Descriptive Framework, is a technology for modeling and representing data on the web. RDF uses a very simple structure to represent each relation using a triple-store "<subject, property, object>".

Triple stores is a type of database system that is specifically designed to store and manage RDF data[Ont22]. The architecture of triple-stores is optimized for processing graph data. However,triple-stores are not performing well when processing large RDF datasets due to their centralized architecture which can make it difficult to distribute the workload across multiple nodes[GHH14]. To address this issue, the Semantic Web community has started to consider the relational model as the underlying schema for RDF datasets. By leveraging the benefits of relational databases, such as advanced query optimization and indexing, this approach can improve the performance of RDF data processing. However, storing and managing RDF data in a relational database presents several challenges due to the differences in the structures and semantics of RDF and relational data.

As the volume of RDF data continues to grow, the need for efficient and scalable solutions for storing and querying RDF data in relational databases becomes increasingly important. The use of RDF data in applications such as knowledge graphs, linked data, and semantic web applications make it necessary to have a solution that can handle the complexity and scale of RDF data while preserving its relationships and semantics[Hor08].

Without a proper solution for storing RDF data in a relational database, organizations may resort to using specialized RDF stores or NoSQL databases, which can lead to increased costs and complexity. This can result in increased maintenance activity after development. Therefore, there is a pressing need for a solution that can bridge the gap between RDF data and relational databases and provide a scalable, efficient, and flexible way to store and manage RDF datasets in a relational database.

Different types of RDF relational schemas have been proposed to query RDF datasets, but state of the art shows that there is no "One-Size-Fits-All" solution[KZJ+19]. Depending on the type of query pattern, one solution may outperform another.

In this thesis, we propose a hybrid schema solution that combines different types of RDF relational schemas using both data and query profiling techniques to provide better performance. Furthermore, we create an auto query mapping solution that allows queries to adapt to new versions while the schema is changing, further improving the performance and adaptability of RDF data processing.

1.1 Problem Statement

Storing RDF graphs in a relational model presents several challenges that need to be addressed. RDF data is inherently complex and has a flexible structure, making it difficult to store it in a relational database. Because graph databases are schema-free, while relational databases require schema upfront. The goal of this thesis is to design and implement a solution that can efficiently store and manage RDF datasets in a relational databasewhile preserving the semantics and relationships of the data.

Challenges:

There are already too many ways to represent RDF graphs in relational models (schemas). There is no one schema that can have constant and ever-well-performing behavior with all query shapes. Each schema outperforms the others with specific types of query shapes. For example, the WPT schema is well-suited for star-shaped queries, on the other hand, for linear queries, performance is worse.

Performing schema tuning (changing the schema to adapt to the query workload), requires changes in the query workload (that would not work anymore with the changes in the schema). Therefore, our thesis also aims to provide an automatic translation solution for the newly generated schema.

The solution needs to overcome these challenges and provide a scalable, efficient, and flexible way to store RDF datasets in a relational database while preserving the relationships and semantic meaning of the data.

1.2 Research Questions

The research questions of the thesis are followings :

RQ1: How to automatically design a relational schema for efficiently processing large RDF knowledge graphs?

RQ2: How to automatically adapt query workload with schema changes?

1.3 Contribution

Our main contributions are the followings:

Tailored Schema Optimization Algorithm : The first contribution of the thesis is the development of a tailored schema optimization algorithm for SPARQL. The algorithm takes into account both data and query profiling in order to create an optimized schema that can efficiently handle complex interconnected data. By using data profiling, the algorithm identifies the most frequently used properties and entities in the dataset, while query profiling helps to identify the most common query patterns. By combining these

two approaches, the algorithm creates a tailored schema tailored to the specific dataset and query workload, leading to better query performance.

Replication of Most Used Properties : The second contribution of the thesis is the replication of the most frequently used properties while generating the schema. This is a novel approach that has not been explored in previous research. By replicating the most used properties, in our thesis, we demonstrate that the performance of the system can be greatly improved, particularly in cases where the data is heavily interconnected.

Query Translation Algorithm : The third contribution of the thesis is the development of a query translation algorithm that can convert old queries to newly adapted schema queries . This algorithm is particularly useful when the schema is updated and old queries need to be adapted to the new schema. The query translation algorithm helps to reduce the manual effort required to update old queries and ensures that the new queries are optimized for the new schema.

1.4 Outline of the Thesis

The thesis is structured as follows :

Chapter 2 provides an introduction to various concepts and technologies that are relevant to the topic of the thesis. Knowledge graphs, semantic web, RDF , SPARQL (SPARQL Protocol and RDF Query Language) and RDF relational schemas are all covered in this chapter. The goal of this chapter is to provide the reader with the necessary background and context to better understand the rest of the thesis.

In Chapter 3, the algorithm designed for optimizing an existing WPT (Wide Property Table) schema is presented. The chapter describes the steps involved in the algorithm, including schema normalization and the creation of new sets of tables based on query workloads. The chapter also discusses the process of adapting existing queries to the new schema and the introduction of a new algorithm for automatically transforming old schema queries to new ones.

Chapter 4 provides an overview of different solutions that have been proposed to address the same issue of optimizing RDF schemas. The chapter discusses the strengths and weaknesses of each solution and provides a comparison with the approach proposed in the thesis.

Chapter 5 presents the results of the experiments conducted to evaluate the proposed algorithm against state-of-the-art solutions. The experiments are performed on WATDIV datasets with 1M, 10M, and 100M data sizes. The chapter provides detailed analyses of the experimental results and shows the effectiveness of the proposed algorithm.

Chapter 6 concludes the thesis by summarizing the main contributions of the work and discussing the limitations and future directions. The chapter provides a perspective on the potential improvements and extensions of the proposed algorithm and its possible applications in real-world scenarios.

2 Background

In this chapter, we provide background information necessary to contextualize this thesis's subsequent sections for readers unfamiliar with Semantic Web. We start giving information about knowledge graphs and semantic webs; then, we define Resource Descriptive Framework(RDF) and SPARQL query language. In order to understand the structure of the following chapters, we give some information regarding Relational Schemas and Relational Algebra.

2.1 Knowledge Graphs and Semantic Web

The World Wide Web Consortium is the driving force behind the Semantic Web (W3C). The Semantic Web's main purpose is to cause the existing Web to evolve so that users may search, find, share, and combine information with minimal effort. People could use web for different purposes such as (online shopping,searching for some information and etc.). Nonetheless, machines can not do any of these jobs without human help, since web pages are designed to be viewed by humans. Semantic Web is a vision for the future where machines can easily analyze the data, do different types of tiresome jobs related to the data on the web.

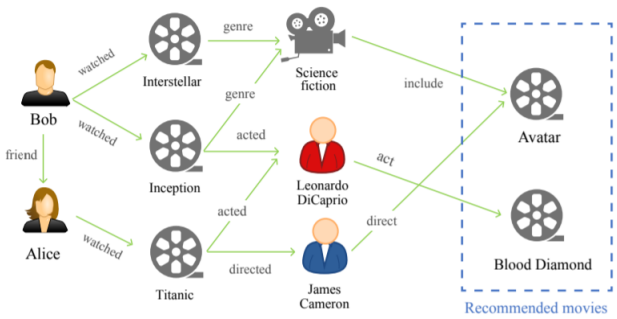


Figure 1. A Guide to the knowledge graphs [May21]

Knowledge graphs are a sort of semantic network that serves as the foundation for network relationships between concepts and entities – links between real-world objects, events, or abstract concepts. There have been different proposals of the knowledge graphs from different big companies with varying complexity and approaches, such as Google's Knowledge Graph, Facebook's Entities Graph, etc.

2.2 RDF and SPARQL

RDF: The standard data model used for the Semantic Web is the Resource Descriptive Framework(RDF). RDF was first released in 1999 as a metadata data model that allows describing anything, including people,animals, and objects. In RDF, Facts is stored in the form of Triples of (<subject,"<predicate">,<object>). The RDF predicate or relationship always connects a subject to an object.

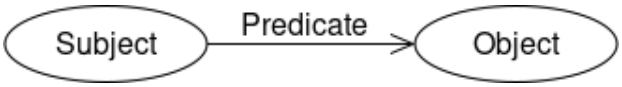


Figure 2. RDF triple relationship

Figure 3 shows a simple example of RDF data that describes...

```
<Student1,Name,Jake>
<Student1,Age,20>
<Student1,Birthyear,2000>
<Student1,Studies,University1>
<Student2,Name,John>
<Student2,Age,23>
<Student2,PhoneNumber,45522244>
<University1,Name,Harvard>
<University1,City,Cambridge>
<University2,Name,MIT>
<University2,City,Massachusetts>
```

Figure 3. An example RDF graph serialized in n-triples format

From this sample, we can see that not only does one statement belongs to one resource, but multiple can also reference the same resource.

SPARQL is the standardized graph query language for RDF data, the same as SQL is for relational databases. The triple pattern is similar to the RDF triple, except that the triple's constituents can be substituted by (unbounded) variables (preceded by the "?" character) and can be referred to in multiple triple patterns across the query. The SELECT clause is used to specify the projected variables. The datasets are specified by the FROM clause, and the filter conditions are defined by the WHERE clause. For instance, the SPARQL Q1 shown in Listing 1 can be described as "return student whose name is Jake".

```
SELECT ?student
WHERE {
        ?student rdf:Name "Jake"
```

```
}
```

Listing 1. An example of a SPARQL query.

2.3 SPARQL Query Shapes

In SPARQL, the concept of query shapes can be related to the structure of the graph patterns used in a query. There are four types of query shapes are used in SPARQL: Linear, Star, Snowflake and Complex[FWF+19].

Linear : SPARQL query that involves a single graph pattern with a single triple pattern. This schema is simple and straightforward, however, it is not optimized for complex queries and can result in large amounts of redundant data.

Star : SPARQL query that involves multiple graph patterns where each pattern has a central node with multiple outgoing edges.

SnowFlake: SPARQL query that involves a complex graph pattern where the central node has many incoming and outgoing edges, and additional nodes are connected to it. This schema reduces data redundancy and saves disk space, but it can also result in more complex queries and slower performance.

Complex : SPARQL query that involves a combination of different types of shapes mentioned above.

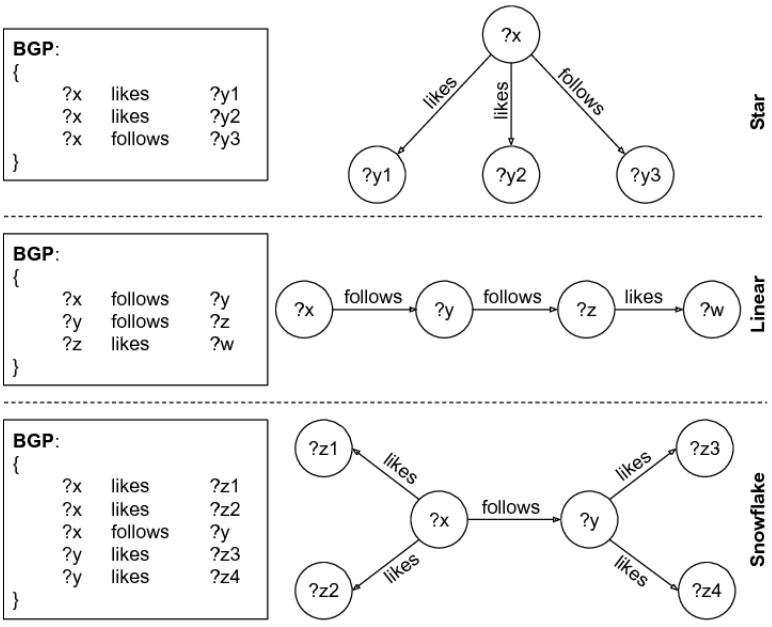


Figure 4. SPARQL query shapes [SPZSL16]

Therefore, the concept of query shapes in RDF and graph databases can be directly applied to SPARQL queries to analyze their structure and optimize query performance. By understanding the different types of query shapes and their characteristics, it is possible to design more efficient RDF and SPARQL query processing systems.

2.4 RDF Relational Schemas

Triplestores are systems that store RDF data in their native graph structures and use the native standard graph query language. RDF queries over mapped databases usually contains star queries that reference multiple columns underlying the table , and generated sql query most likely will contain self-joins. These extra joins are not only costly but also the complexity of query optimization grows exponentially with the number of joins. When compared to relational databases, SPARQL query optimization is quite difficult. SPARQL queries can involve complex patterns of triple patterns. These patterns can be combined with logical operators, optional patterns, and other constructors to form queries that are difficult to optimize. As a result, most queries are executed with poor strategy. RDBMS systems, on the other hand, may be readily optimized utilizing sophisticated techniques such as data splitting, indexing, materialized views, and so on.

The most typical relational schemas for displaying RDF in the relational schema are shown in the next paragraph.

2.4.1 Single Statement Table

Single Statement table schema (ST) stores RDF triples in a single table with three columns (subject, propert, object). It is used in many open-source triple stores, such as RDF4J[dev], Apache Jena[apa], and Virtuoso[lin16]. This schema is very similar to triple store. It has a problem of self-joins if multiple different predicates are used. Therefore its performance on large-scale datasets is poor. In Table 1, we can see the example of the ST table.

2.4.2 Property Table

Property Table (PT) is another relational schema for storing RDF data. In some sources, it is also referred to as an n-ary columns table. It groups all related attributes into one table where column names are predicates, and each cell includes the subject and predicate's object value. This level of grouping can be completed with either a grouping algorithm or a type definition table. Table 2 depicts an example of a Student Table.

This schema does not require any join criteria to obtain all of the student information. This schema, however, has a number of drawbacks. For example, not all subjects will have data for each predicate, thus increasing the sparsity of the data with too many NULL values. If the data comprises an array structure, we must either flatten it or use an

Subject	Property	Object
Student1	Name	Jake
Student1	Age	20
Student1	Birthyear	2000
Student1	Studies	University1
Student2	Name	John
Student1	Age	23
Student1	Phonenumber	45522244
University1	Name	Harvard
University1	City	Cambridge
University2	Name	MIT
University2	City	Massachusetts

Table 1. An example of an RDF graph is represented in the ST schema.

Subject	Name	Age	Birthyear	PhoneNumber	Studies
Student1	Jake	20	2000		University1
Student2	John	23		45522244	

Table 2. PT schema example.

alternative technique. Therefore this schema works best with well-structured data rather than poorly organized data.

2.4.3 Wide Property Table

Wide Property Table (WPT) is a special version of the PT table. It groups all the attributes under one denormalized table called (WPT). The main idea is to decrease the join conditions and try to have one reference table for accessing the data. However, it is also suffering from sparsity and huge size in terms of storage. Table 3 describes an example of a WPT table.

Subject	Name	Age	Birthyear	Phonenumber	Studies	City
Student1	Jake	20	2000		University1	
Student2	John	23		45522244		
University1	Harvard					Cambridge
University2	MIT					Massachusetts

Table 3. WPT schema example

2.4.4 Vertically Partitioned Table

A vertically partitioned table(s) schema, in which each predicate is kept in its own table, is another approach to relationally store RDF data. The table is composed of two columns (i.e., a subject and object) of the triple, with the predicate being the table's name. This schema attempts to circumvent the sparsity problem by deleting null values. However, VP-based methods are inefficient when processing queries with unbounded predicates; in this situation, all tables must be searched and their answers must then be combined to produce the final result. Because this cost grows linearly with the number of various predicates in the dataset[A14], VP is not the ideal solution for encoding datasets with a large number of predicates. This strategy's another drawback is that some partitions may result in a sizable section of the entire graph and generate a lot of I/O [A14]. In table 4, we can see the example of VP table.

Names

Subject	Object
Student1	Jake
Student2	John
University1	Harvard
University2	MIT

Ages

Subject	Object
Student1	20
Student2	23

Phonenumbers

Subject	Object
Student1	45522244

Birthyears

Subject	Object
Student1	2000

Cities

Subject	Object
University1	Cambridge
University2	Massachusetts

Table 4. Vertically Partitioned Tables Example

2.5 FP-Growth Algorithm

The FP-Growth algorithm is a popular algorithm for frequent pattern mining in data mining and machine learning[sof23]. It is an efficient algorithm for mining frequent itemsets from a large dataset. The "FP" in FP-Growth stands for "Frequent Pattern."

The main idea behind the FP-Growth algorithm is to construct a tree-like data structure called the FP-tree that captures the relationships among the frequent itemsets in the dataset. The algorithm first scans the dataset to identify frequent items and then uses these items to build the FP-tree. Once the FP-tree is constructed, frequent itemsets can be extracted from the tree using a recursive mining process.

Here's a brief overview of the steps involved in the FP-Growth algorithm:

1. Scan the dataset to identify frequent items.
2. Build the FP-tree by inserting each transaction into the tree.
3. Mine the FP-tree to generate frequent itemsets.

One of the key advantages of the FP-Growth algorithm is its ability to handle datasets with a large number of items and transactions efficiently. It achieves this by compressing the dataset into a compact FP-tree, which avoids the need to repeatedly scan the original dataset.

Another advantage of the FP-Growth algorithm is that it can be parallelized, which can further improve its performance on large datasets. For example, consider the following scenario described in table 5 below.

Transaction	List of Items
T1	A,B,C
T2	B,C,D
T3	D,E
T4	A,B,D
T5	A,B,C,E
T6	A,B,C,D

Table 5. List of Transactions

We will take the support threshold as 50 percent. According to our example, our min support will be 0.5 * 6=3.

If we calculate a count of each attribute and sort them by count, then we will have the following table.

Item	Count
B	5
A	4
C	4
D	4

Table 6. Count of each item ordered by count

As we can see that we have not included E because it is less than minimum support threshold.

The next stage is building FP-tree, which is the final tree described below.

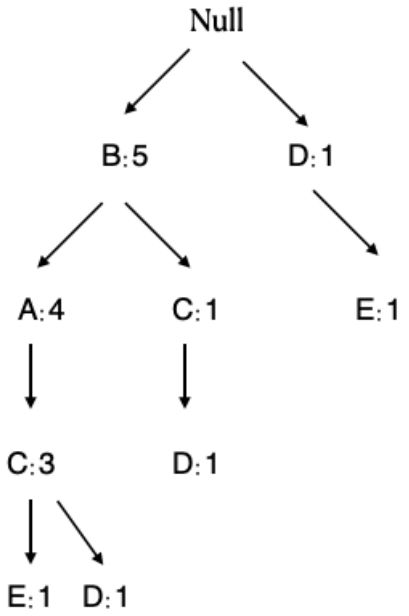


Figure 5. Final FP-tree

The steps for building FP-tree are below :

1. The root node is considered null.
2. During the first scan of Transaction T1 containing items A, B, and C, B is linked as a child to the root node, A is linked to B, and C is linked to A
3. In Transaction T2 containing items B, C, and D, B is already linked to the root node, so its count is incremented by 1. C is linked as a child to B, and D is linked as a child to C
4. In Transaction T3 containing items D and E, a new branch with E is linked to D as a child
5. In Transaction T4 containing items A, B, and D, B is already linked to the root node, so its count is incremented by 1. A is also incremented by 1 as it is already linked with B in T1
6. n Transaction T5 containing items A, B, C, and E, the sequence will be B, A, C, and E, resulting in counts of (B:4), (A:3), (C:2), and (E:1)

7. Finally, in Transaction T6 containing items A, B, C, and D, the sequence will be B, A, C, and D, resulting in counts of (B:5), (A:4), (C:3), and (D:1).

The next stage is a mining of FP-tree which is summarized below.

Item	Conditional Pattern Base	Conditional FP-tree	Frequent Pattern Generated
D	{B,A,C:1},{B,C:1}	{B:2},{C:2}	{B,D:2},{C,D:2},{B,C,D:2}
C	{B,A:3},{B:1}	{B:4},{A:3}	{B,C:4},{A,B,C:3},{B,A,C:3}
A	{B:4}	{B:4}	{B,A:4}

Table 7. FP-tree mining

1. The smallest node item, E, is not considered because it doesn't meet the minimum support count requirement and is therefore removed.

2. The next smallest node, D, is present in two branches: (B, A, C, D4:1) and (B, C, D:1). Using D as a suffix, we get the prefix paths (B, A, C:1) and (B, C:1), which form the conditional pattern base.

3. We then create an FP-tree using the conditional pattern base, which contains (B:2, C:2). A is not included since it doesn't meet the minimum support count.

4. This path generates all combinations of frequent patterns: (B, D:2), (C, D:2), (B, C, D:2). For C, the prefix path is (B, A:3) and (B:1), resulting in a two-node FP-tree (B:4, A:3) and frequent patterns: (B, C:4), (A, C:3), (B, A, C:3).

5. For A, the prefix path is (B:4), generating a single-node FP-tree (B:4) and frequent patterns: (B, A:4).

Overall, the FP-Growth algorithm is a powerful tool for discovering frequent patterns in large datasets and is widely used in a variety of applications, such as market basket analysis, web log analysis, and bioinformatics.

3 Design

The main idea behind the H_2O is to provide a custom structure for each RDF dataset based on its specific characteristics and query workload. The algorithm recognizes that different RDF datasets may have different levels of structure and therefore require different schema designs to optimize their storage and query processing performance.

For instance, suppose we have a dataset for social media posts, where for each post, we have properties such as schema:publish date, author, message and etc. In this scenario, since we have one to one relationship with post and its properties in this scenario, using the WPT table is the best-case scenario. After some time, we have added new property which is mentions (showing who has mentioned your post). The repeated mentions property may result in duplicate data and storage inefficiencies.

The schema generation algorithm aims to find a solution that can handle different types of RDF datasets and query workloads, providing the most efficient storage and query execution performance. H_2O considers various factors, such as the structure and relationships of the data, the frequency and type of queries, and the performance requirements to determine the optimal schema design.

Figure 6. The workflow of the algorithm [May21]

Grouping, Partitioning, and Stars Discovering are the three steps of our approach. The first step, Grouping, is designed to group attributes (or properties) in the RDF data that meet a specified support threshold parameter. The algorithm employs the association rule algorithm known as FPGrowth to identify similar properties and group them together. The idea behind this step is that properties that are frequently used with each other can be stored in the same table, reducing the number of join operations required for query processing.

The result of the Grouping step includes final tables as well as clustered properties. These tables are built as VT tables, as they contain attributes that are not listed in any clusters.

The second step, Partitioning, takes the groups from the previous step and attempts to find a unique set of attributes for each group while keeping the amount of null storage for each group below a specified null threshold. The final tables are the result of the Grouping and Partitioning steps.

The final step, Stars Discovering, attempts to discover stars (a group of properties with a central property) from a specified query workload. New table candidates are formed by merging similar stars. The pseudocode for the entire algorithm is described in Algorithm 1. The three-step approach is designed to provide a flexible and efficient

Algorithm 1: Main algorithm for the H20

Input: The support threshold, null threshold, data workload and query workload
Output: The final optimal schema

```
1 forall support ∈ Support do
2     forall nullthreshold ∈ NullThreshold do
3         Groups, FinalTables ← GetGroups(support, nullthreshold);
4         FinalTables.append(GetPartitions(Groups);
5 FinalTables.append(StarsDiscovering(QueryWorkload);
```

solution for storing RDF data in a relational database optimized for the specific needs of each RDF dataset and query workload.

3.1 Data Structures

Our algorithm takes the RDF graph as input with two different thresholds: (1) Null threshold and (2) Support Threshold. Support threshold is the value to measure how frequently properties appear together for the same subject in the RDF graph. If sets of properties meet the condition then they are considered to be in the same n-ary table. Null Threshold is the percentage of the acceptable null values in the table.

We have defined two main data structures for this algorithm:
(1) Property Usage List. This is a list structure where all the properties of the RDF data is storing counts of the subject that property has. If we have built the data structure based on the Figure 3 from the background section, then our data structure will be like the below figure.
(2) Subject Property Basket. This is the list of the subject with its associated properties. It is shown in the form of SubjId -> prop1,prop2. . . propn) where SubjectId is the URI of the given subject and its property basket is the list of all properties defined for that subject. As an example from, for the sample data in Figure 3, we could have the following table.

3.2 Algorithm Design

The goal of this three-phase clustering process is to group together attributes in a way that maximizes the data saved in n-ary tables. The focus is on identifying maximum-sized clusters that often occur in the data and include the most properties. By doing so, the

Property	Usage
Name	4
Age	2
City	2
Birthyear	1
Studies	1
Phonenumber	1

Table 8. Usage of each property

Subject	Properties
Student1	[Name,Age,Birthyear,Studies]
Student2	[Name,Age,Phonenumber]
University1	[Name,City]
University2	[Name,City]

Table 9. The subject property basket list

approach aims to optimize the storage and querying of the data. The full lifecycle of the algorithm is described in Figure 7.

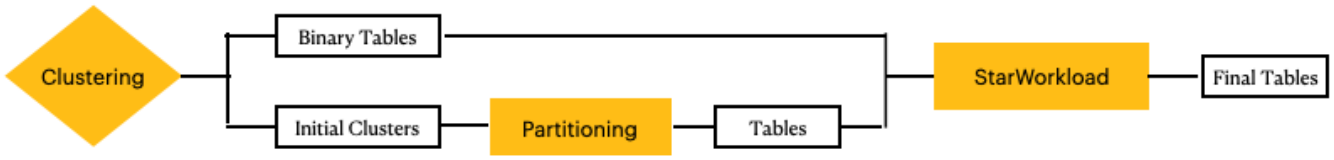


Figure 7. The algorithm design

3.2.1 Grouping Phase

This phrase describes a two-step grouping process for grouping together attributes in a dataset.

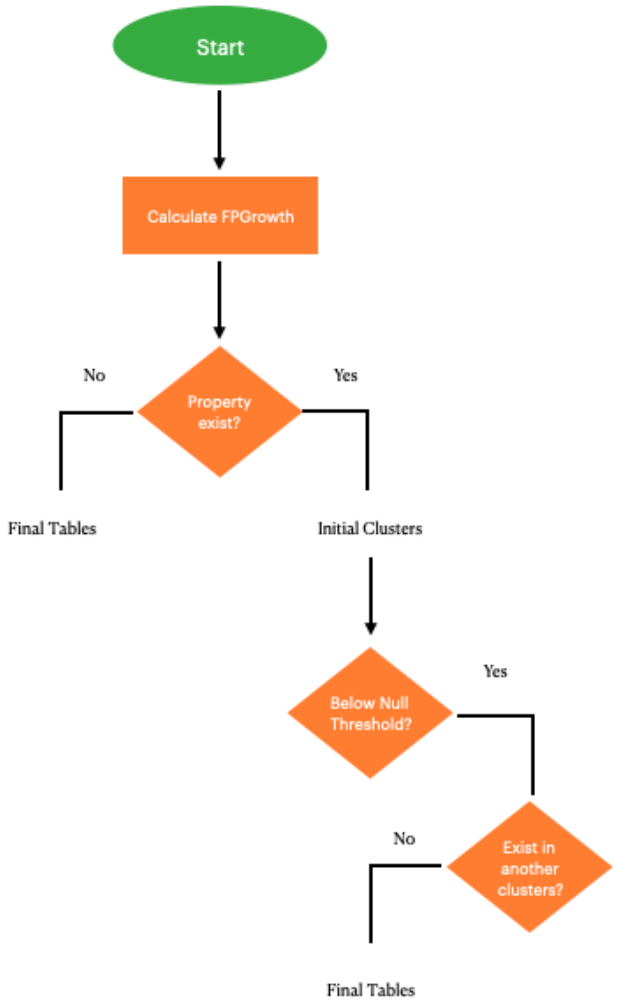


Figure 8. The algorithm design

In the first step, frequent itemset discovery is used to identify sets of highly correlated groups. A frequent itemset is a set of items (in this case, attributes) that frequently occur together in the data. The support of a frequent itemset is defined as the frequency with which the attributes are found together in the same group. A high support indicates that these attributes are highly correlated with one another.

To discover the groups, a support threshold is set, and property sets are considered to be groups only if they are higher than or equal to this threshold. The optimal support threshold is determined based on the size and correlation of the resulting groups. A high support threshold can result in a large number of small, strongly connected groups, while

a low support threshold can result in a smaller number of larger, less connected groups. In the second step of the grouping phase, an initial set of final tables is created. This set contains properties that are not listed in the clusters and are thus saved as a VT Table, as well as properties that fulfill the null threshold (i.e., properties that do not contain any properties from other clusters). Any group that is added to the final table phase is deleted from the list of groups.

Pseudocode for the grouping phase of an algorithm (Algorithm 2) that uses the inputs of a Subject Property Basket, Support Threshold, and Null Threshold to group together attributes in a dataset. The algorithm first detects groups and orders them by

Algorithm 2: Get Groups for initial phase

Input: The Subject Property Basket, Property Usage List, Null Threshold and Support Threshold

Output: Groups and Final Binary Tables

```
1  Groups ← FPGrowthList(Basket, PropertyUsage, SupportThreshold);
2  FinalTables ← Properties which are not in any groups
3  forall c1 ∈ Groups do
4      Flag ← true;
5      if findNullThreshold(c1) < NullThreshold then
6          forall c2 ∈ Groups do
7              if c1 ∈ c2 and c1 ≠ c2 then
8                  Flag ← false
9          if Flag = True then
10             FinalTables.insert(c1);
11             Groups.remove(c1) ;
```

their supportthreshold, which is a value that indicates how frequently the attributes in the group occur together in the dataset. The groups are saved in a variable called Groups[1].

Next, the algorithm creates the first set of VT tables based on the attributes that are not part of any groups. These attributes are saved in a VT table[2].

The algorithm then looks for groups that are less than the null threshold (a value that determines how many properties a group can have before it is considered "null") and not part of any other groups. If such a group exists, it is added to the final table and removed from the Groups variable[4-11].

The property usage list is used to calculate the null threshold. The formula used for null threshold is:

$$NullThreshold = \frac{\sum(PU_{max}(c) - PU_{count}(c_i))}{(c+1) * PU_{max}(c)}$$

where |c| is the number of properties in the group and PU.maxcount(p) is the maximum property count.

Finally, the algorithm returns the first set of initial tables and the remaining groups that will be sent to the next phase of the algorithm. The initial tables are the VT tables created in the second step of the algorithm, and the remaining groups are those that were not added to the final table. These groups will be further processed in the next phase of the algorithm.

Example. We will use the following WPT table to explain algorithm steps. For our case, we will take a support threshold of 20 percent and a null threshold of 10 percent. Our initial table is described below.

Subject	Name	Age	Working	Likes	Teaches	Supervises	Website	City	Population
Person1	Farid	25	Data Engineer	Table Tennis					
Person2	Parviz	23	Software Engineer	Gym					
Person3	Murad	23	Data Analyst						
Person4	Ragab	23			Data Engineering	Farid			
Person5	Riccardo	28			Data Engineering	Ragab			
University1							ut.ee	Tartu	
University2							taltech.ee	Tallinn	
City1									100k

Table 10. An example of a WPT for the running example.

Firstly, we will calculate our initial data structures (PropertyUsage List and Subject Property Baskets) from the above example.

Property	Usage
Name	5
Age	5
Working	3
Likes	2
Teaches	2
Supervises	2
Website	2
City	2
Population	1

Table 11. Property Usage List.

Subject	Properties
Person1	[Name,Age,Working,Likes]
Person2	[Name,Age,Working,Likes]
Person3	[Name,Age,Working]
Person4	[Name,Age,Teaches,Supervises]
Person5	[Name,Age,Teaches,Supervises]
University1	[Website,City]
University2	[Website,City]
City1	[Population]

Table 12. Subject Property Basket.

Later, we will calculate support and null thresholds based on those data structures. For example, if we want to calculate the support percentage of the [Name, Age, Working,

Likes], we need to find out how many subjects are simultaneously containing these properties out of all subjects. Then in our scenario, it will be

$$support = \frac{2 * 100}{8} = 25\%$$

If we want to calculate the null threshold for the same group, then

$$NullThreshold = \frac{((5-5)+(5-5)+(5-3)+(5-2)) * 100}{5 * 5} = 20\%$$

We will calculate the support percentage for all possible group property combinations. The following tables will be generated.

Step 1

Clusters	Support
Name	62%
Age	62%
Working	37%
Likes	25%
Teaches	25%
Supervises	25%
City	25%
Website	25%
Population	12%

Step 2

Clusters	Support
Name,Age	62%
Name,Working	37%
Age, Working	37%
Name,Likes	25%
Name,Working	25%
Name,Teaches	25%
Name,Supervises	25%
Age,Likes	25%
Age,Working	25%
Age,Teaches	25%
Age,Supervises	25%
Working,Likes	25%
Teaches,Supervises	25%
Website,City	25%

Step 3

Clusters	Support
Name,Age,Working	37%
Name,Age,Likes	25%
Name,Age,Teaches	25%
Name,Age,Supervises	25%
Age,Working,Likes	25%
Age,Supervises,Teaches	25%

Step 4

Clusters	Support
Name,Age,Working,Likes	25%
Name,Age,Teaches,Supervises	25%

Figure 9. Support Percentages

First FPGrowth Function will remove all the groups below the support threshold, such as [Population], which has only 12 percent. Then the algorithm will consider a maximum number of properties that can be grouped and remove all their subsets. For

Example, [Name, Working] is a subset of [Name, Age, Working, Likes], so we will not consider subsets. In that scenario, from the first function, we will get (Name, Age, Working, Likes), (Name, Age, Teaches, Supervises),(Website, City)[1]. Based on those clusters, we now calculate a null percentage.

Cluster	Null Percentage
Name, Age, Working, Likes	20%
Name, Age, Teaches, Supervises	24%
Website, City	0%

Table 13. Null Percentage by Cluster

Since the Population property is not part of any group, then we will create this property as a VT table.[2].

In the next stage, we will loop through all the groups to find the final table candidates[3-11]. First, we check whether the group is below the null threshold to decide whether we can use this group as the final table[5]. Among generated groups, [Name, Age, Working, Likes] and [Name, Age, Teaches, Supervises], which have 20 percent and 24 percent null storage, respectively, do not meet this condition. We will forward these groups to the partitioning phase. However, [Website, City] is less than the null threshold, so we will look into whether other groups contain any properties of this group. Since there is no intersection with other groups, we will consider [Website, City] as the final table.

The result of this phase is in Figure 10:

Final Tables	Clusters
Population	Name,Age,Working,Likes
Website,City	Name,Age,Teaches,Supervises

Figure 10. Grouping algorithm final result

3.2.2 Partitioning Phase

The second phase, which is the focus of this paragraph, is to further refine the grouping by dividing the groups into non-overlapping sets and ensuring that each set is below a certain null threshold.

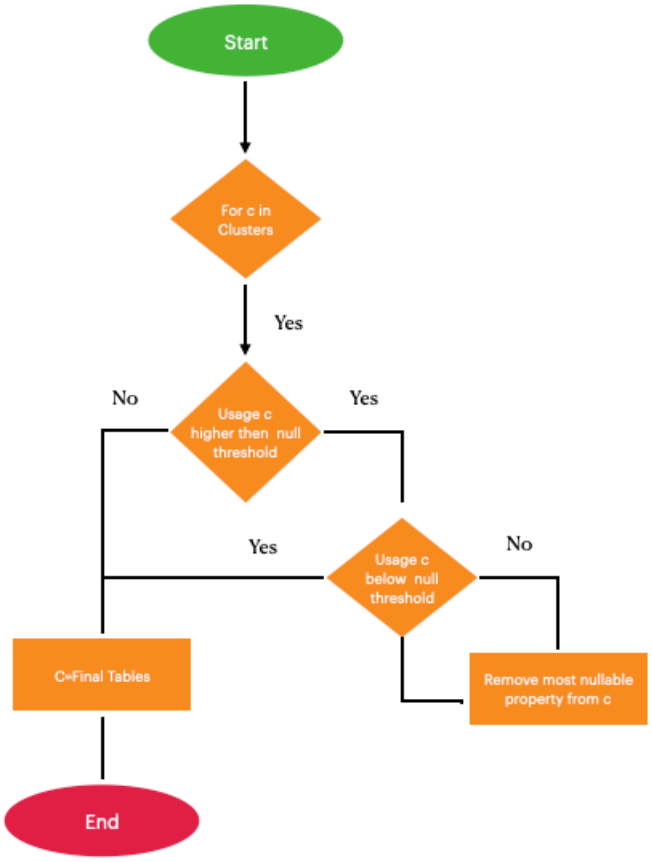


Figure 11. The algorithm design

The null threshold refers to a minimum support level for each cluster, which is the frequency at which a particular attribute or set of attributes appears in the data. The

goal is to ensure that each partitioned group is below the null threshold, so that it can be efficiently queried without the need for costly join or union operations.

The process for discovering final tables involves a greedy approach that attempts to increase the support threshold for each group until a final set of non-overlapping groups is identified. In some cases, a group may meet the null threshold and can be considered a final table. In this scenario, the process attempts to trim any extraneous attributes from the group to further optimize the query workload.

In other cases, a group may not meet the null threshold and the process attempts to prune attributes until the group does meet the threshold. If a pruned attribute is present in another group with a lower support threshold, it may be merged into a larger n-ary table. If the pruned attribute is not present in any other group, it must be retained as a VT table.

Overall, the goal of this process is to create a set of non-overlapping groups that meet the null threshold, which can then be efficiently queried without the need for costly join or union operations.

Algorithm 3 shows the steps of partitioning the groups produced in the first phase of the process and identifying the final tables that meet the null threshold.

Algorithm 3: Get Partitions for the second phase

Input: Groups from the first phase, Property Usage List, Null Threshold
Output: Final Tables from data workload

```
1  forall c ∈ Groups do
2      if PropertyUsage(c1) ≥ NullThreshold then
3          while PropertyUsage(c) ≤ NullThreshold do
4              p1 ← Most Nullable Property from clusters;
5              c1 ← Remove p1 from the group
6              if p1 ∉ Other Groups then
7                  FinalTables.Insert(p1);
8      FinalTables.Insert(c1);
9      forall c2 ∈ Groups do
10         if c2 ∉ c1 then
11             c2.remove(c2 ∩ c1);
```

The input for Algorithm 3 includes the Groups from the first phase, a PropertyUsage List, and the NullThreshold. The algorithm begins by looping over all of the groups and checking whether each group meets the NullThreshold. If a group meets the NullThreshold, it is directly entered into the final table list.

If a group does not meet the NullThreshold, the algorithm attempts to delete a property from the group, which results in more null values in the group. The algorithm

then checks to see whether the deleted property exists in other groups before deciding whether to include it as a distinct final table.

If the final table list does not already exist, the algorithm creates it and adds the group to the list. If the final table list already exists, the algorithm checks whether the group can be merged with an existing final table that has the same set of properties. If a merge is possible, the algorithm merges the groups into a single final table.

If a merge is not possible, the algorithm only removes the property from the group until the null threshold is met. This process continues until all groups have been checked and partitioned into non-overlapping sets that meet the null threshold.

For example, let's say we have three groups, C1, C2, and C3, with the following property usage lists and support levels:

C1: p1, p2, p3, p4, support(C1) = 0.8

C2: p1, p3, p5, p6, support(C2) = 0.6

C3: p3, p5, support(C3) = 0.4

Suppose the null threshold is set to 0.5, Algorithm 3 will iterate over each group and check whether the group meets the null threshold. Since C1 and C2 meet the null threshold, they would be directly inserted into the final table list. However, C3 does not meet the null threshold, so the algorithm would attempt to delete a property from the group. In this case, the algorithm would delete p5 from C3, which results in a null value.

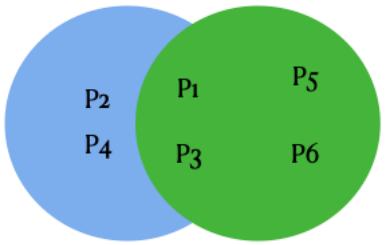


Figure 12. The final table sets

The algorithm would then check whether p5 exists in any other groups. Since p5 is present in C2, the algorithm would not create a distinct final table for C3, but rather merge it with C2. The result would be two final tables: p1, p2, p3, p4 and p1, p3, p5, p6.

Example. Using the example from the grouping phase section, there are two groups that need to be refined into final tables. The first group is [Name, Age, Working, Likes],

and its null storage is higher than the null threshold. To satisfy the condition, the property with the most null values, i.e., "Likes", is removed from this group. Then the null threshold is recalculated for the remaining properties [Name, Age, Working], which is now 10 percent and satisfies the condition. As a result, one VT table, Likes, and one PT table, [Name, Age, Working] are created for this cluster, and these properties are removed from other groups.

In the second group [Name, Age, Teaches, Supervises], Name and Age properties are removed, leaving [Teaches, Supervises]. The null threshold for this group is 0 percent now, which meets the condition and can be generated as a final table.

The final tables from this phase are following :

Tables
Name, Age, Working
Likes
Website, City

Table 14. The final table sets from partitioning phase

3.2.3 Stars Discovering

The algorithm developed for this phase aims to optimize the schema further by creating new sets of final tables and making adjustments to the already created final tables based on the provided query workload using the BGP pattern. Overall, this phase is critical in ensuring that the schema is optimized not just for the data provided but also for the query workload. By identifying stars in the query workload (i.e., properties that are queried together for the same Subject) and optimizing already created tables, the algorithm can ensure that the schema is efficient and can handle queries with minimal join or union operations, thereby improving query performance.

The main goal of this phase is twofold. Firstly, the algorithm tries to identify star query shapes from the query workload and merge them if they have at least one common property.

Secondly, the algorithm tries to optimize already created tables from above phases either by joining them with new stars or by keeping them as separate tables. In this scenario, the following conditions need to be considered. Firstly, if the table is a subset of any star, then algorithm need to remove this table from the schema. Secondly, if only some properties exist in both tables, we need to consider the support threshold of each property. If the support threshold is 2 times higher than the support threshold, we keep this property in both tables; otherwise, we remove it from the previous table but keep it in the star tables.

As an example, consider the following scenario: T1=p1,p2,p3, T2=p4,p5, S1=p1,p4,p5. We can see that T2 is a subset of S1, so we will remove T2 and keep S1 only. On the other hand, T1 and S1 have a common property, which is p1. Here we need to decide whether to keep it in both tables or remove it from T1. This will be determined based on the support percentage of p1. If it is less than support threshold, then algorithm will remove this property from T1 and keep it only in S1; otherwise, algorithm will keep it in both tables.

The second most important part of this algorithm is finding out join tables (which is third table used for storing many to many relationships). This step is very crucial for removing redundancy in the data and at the same time improving query performance.

Algorithm 4 aims to find the common stars from the given query workload and change the existing calculated final tables structure.

Algorithm 4: Discovering Stars from given query workload

Input: Query Workload and Final Tables from the partitioning phase
Output: Final Tables from query workload

```
1  FinalStars, JoinTables ← FindFinalStars(QueryWorkload);
2  forall table ∈ FinalTables do
3      if len(table) > 1 then
4          forall property ∈ table do
5              forall star ∈ FinalStars do
6                  if property ∈ star & PropertyUsageList(property) >
                     2 * SupportThreshold then
7                      FinalTable ← Remove property from Final Table
8              forall jointable ∈ JoinTables do
9                  if property ∈ jointable & PropertyUsageList(property) >
                     2 * SupportThreshold then
10                     FinalTable ← Remove property from Final Table
```

The algorithm takes as input a query workload and the final tables from the partitioning phase. It then uses the FindFinalStars function to identify the final stars and connector tables.

For each table in the final tables, it checks if the length of the table is greater than 1. If so, it loops over each property in the table and then over each star in the final stars. If the property is in the star and its usage count exceeds the support threshold, the property is removed from the final table. The output of the algorithm is the updated final tables.

Example. If we continue our running example from the above section with the below query workload described in BGP Pattern and query pattern.

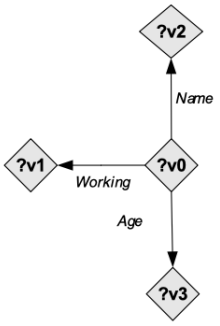
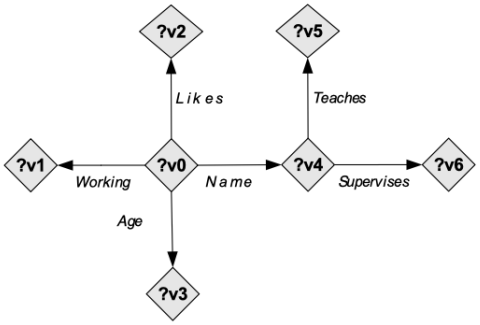
BGP Pattern	Query Graph
{ ?V0 Working ?V1 ?V0 Name ?V2 ?V0 Age ?V3 }	
{ ?V0 Working ?V1 ?V0 Likes ?V2 ?V0 Age ?V3 ?V0 Name ?V4 ?V4 Teaches ?V5 ?V4 Supervises ?V6 }	

Figure 13. The query workload for our algorithm input

Based on the query workload provided, we can identify stars, which are sets of properties that frequently co-occur together in the queries. In this case, we have three stars: [Name, Age, Working], [Name, Age, Working, Likes],[Teaches,Supervises]. Since the [Name, Age, Working] star is a subset of the [Name, Age, Working, Likes] star, we can combine these two sets of properties into one table, which includes all the properties of both stars: [Name, Age, Working, Likes].

If we take final tables from the table 14 into consideration as well we will have following tables:

Name, Age, Working
Likes
Website, City
Name, Age, Working, Likes
Teaches, Supervises

Table 15. Final Tables

Now if we look into the tables generated from the 3 phases we could see that [Name,Age,Working] is a subset of [Name,Age,Working,Likes] so we will remove [Name,Age,Working] table from final tables list. Same for the the Likes table as well. Now updated final tables will be below.

Name, Age, Working, Likes
Website, City
Teaches, Supervises

Table 16. Final Tables

3.2.4 Auto Query Mapping

After creating a new schema, existing queries need to be adapted to the new schema since properties no longer exist in the same tables. To address this issue, a new algorithm was introduced that automatically transforms old schema queries to new ones. For the preprocessing step, a PostgreSQL database was used to store initial data structures, but the choice of database is not mandatory and can be easily replaced with others.

The algorithm relies on two main tables: Global Mapping and Metrics. Global Mapping is used as metadata for keeping track of property location, and Metrics stores information about each property, including the column name, count, and null threshold. The benefit of the Metrics table is that it eliminates the need to calculate metrics each time the queries are run.

The main purpose of the algorithm is to find the best query plan while generating the new schema. When replication of properties is added, the algorithm goes through the following steps to find the appropriate tables:

1. If one table covers the most properties of the selected query, then that table is picked. For example, if we have T1=p1,p2,p3 and T2=p1,p4,p5,p6, and our query contains p1,p2, then we will use T1 since most of the properties are coming from there.

2. If properties from both tables are equally used, then the algorithm uses the Metrics table to calculate the size of each table and picks the one with less size.

The algorithm effectively solves the issue of adapting old schema queries to the new schema, making the transition to the new schema seamless.

3.2.5 AutoQueryMapping Algorithm

Algorithm 5: Find best query plan for existing WPT query

Input: WPT query
Output: Final converted query plan

```
1  TablesDict ← FindDistinctTables(Query);
2  PTTablesDict ← EmptyList;
3  Columns ← TablesDict[table];
4  forall table ∈ TablesDict.keys() do
5      ReplicatedColumns ← FindReplicatedColumns(TablesDict[table]);
6      if Len(ReplicatedColumns) > 0 then
7          BestTable ← FindBestTable(ReplicatedColumns);
8          CoveredColumns ← FindColumns(BestTable);
9          PTTablesDict.Insert(BestTable, CoveredColumns);
10         Columns.Remove(CoveredColumns);
11     PTTablesDict.Insert(FindTables(Columns));
12     JoinCondition ← CreateJoinStatement(PTTablesDict);
13     Subquery ← CreateSubQuery(JoinCondition)
14     Query ← Query.Replace('WPTV0', Subquery)
```

Example Lets consider following WPT query in listing 2.

```
SELECT V0.Name,
       V0.Age,
       V0.Website,
       V0.Teaches
FROM WPT V0;
```

Listing 2. Example query for WPT table

If we continue our running example here, we must rewrite this query to be compatible with our new schema. Our algorithm starts with extrapolating our tables from this query, WPT aliased as V0. And we begin creating a tables dictionary with the table being the key and columns being values[1]. The dictionary described in listing 3.

```
{V0:[Name,Age,Website,Teaches]}
```

Listing 3. TablesDictionary in our sample

Later algorithm iterates through all the tables dictionary to form a subquery for each table[4:11]. First, we find out whether any columns are replicated across multiple tables and store them in the ReplicatedColumns variable.[5]. For finding out replication, we are using the Global Mapping table. In our running example, we do not have any replication. So ReplicatedColumns will be Empty. In this scenario, our algorithm will calculate the list of pt tables with their columns used in the join condition[11]. The following dictionary described in listing 4 will be generated in our example.

```
{T1:[Name,Age],
 T2:[Website],
 T3:[Teaches]}
```

Listing 4. PTTablesDictionary in our sample

Now we will create a join condition based on the following dictionary. According to our running example, it will be like the below in listing 5.

```
T1 FULL JOIN T2 ON T1.Subject=T2.Subject
   FULL JOIN T3 ON T1.Subject=T3.Subject
```

Listing 5. Join condition for WPT V0 table

Later we will generate a subquery considering columns projected from this table. And our final subquery will be like the below in listing 6.

```
(SELECT T1.Name,
        T1.Age,
        T2.Website,
        T3.Teaches
FROM T1 FULL JOIN T2 ON T1.Subject=T2.Subject
         FULL JOIN T3 ON T1.Subject=T3.Subject) V0
```

Listing 6. Final subquery for WPT table

If we replace WPT with our subquery, it will be like the one below in listing 7.

```
SELECT V0.Name,
        V0.Age,
        V0.Website,
        V0.Teaches
FROM (SELECT T1.Name,
        T1.Age,
        T2.Website,
```

```
        T3.Teaches
FROM T1 FULL JOIN T2 ON T1.Subject=T2.Subject
        FULL JOIN T3 ON T1.Subject=T3.Subject) V0;
```

Listing 7. Final subquery for WPT table

3.3 Algorithm Implementation

For the proof of concept of this thesis, we have implemented following algorithms using Python and Postgresql. Python is a popular general-purpose programming language with a wide range of libraries and frameworks available, making it a popular choice for data processing and analysis tasks[Wor22]. PostgreSQL is a powerful open-source relational database system that provides advanced features such as transaction support, data integrity, and concurrency control [Pos]. To connect our Python code to the PostgreSQL database, we used psycopg2, which is a popular database adapter for Python[psy]. It provides an easy-to-use interface for interacting with a PostgreSQL database and can be used to change the database system easily whenever needed.Overall, the combination of Python and PostgreSQL with psycopg2 as a database adapter was a suitable choice for our project, which involved data processing and analysis using a relational database management system.

Project is currently taking WPT schema represented in csv as input file and creates corresponding tables. In the next stage, these tables are converted into csv in order to be deployed to other systems if needed.

4 Related Work

Previous attempts to RDF storage were divided into three types. (1) The triple-store . Oracle[ora23], Sesame[BKvH02], 3-Store[dat], R-Star [MSP+04] and Redland[Bec01] are examples of relational systems that employ a triple-store as their main storage method. (2) The property table. The Property Table schema alleviates the issue of the low performance of Single Statement schema due to excessive self-joins. Jena Semantic Web Toolkit[Wil06]is an example of an architecture that uses property tables as its major storage strategy. Oracle also use property tables as supplementary structures, referred to as materialized join views (MJVs).(3) The decomposed storage model was recently presented as an RDF storage approach , and it has been demonstrated to scale effectively on column-oriented databases with mixed results on row-stores.

One of the attempts was from Peter Boncz[PPEB15], who suggested creating emergent relational schemas by means of semantically and structurally optimizing existing schemas using knowledge graphs.

Figure 14. The workflow of emergent relational schema

The algorithm in figure 14 starts with discovering the Characteristics Set(The group of properties that can be combined)—then tries to find human-understandable labels and relations to the CS with the Labeling Step. In the next step, they merge structurally and semantically similar CS to make the schema more compact. The next step is removing low frequency and high sparse properties from the schema. In the last step, the algorithm filters out some rows to make properties more homogeneous (keeping them in the same type) and individual triples, which make multi-valued attributes. However, in this

solution, only some triples could be described in an automated solution(even in the best-case scenario, 90 % of the data).

Automated relational schema design has primarily been studied using query workload statistics. Techniques have been proposed for index and materialized view creation, horizontal and vertical partitioning, and partitioning for large scientific workloads. However, Mokbel, in his paper, introduced a data-centric approach where the structure of the data is more important than the query workload[LM09]. This work aims to create hybrid schemas using data correlation meaning that putting them in the same cluster if they have used together a lot using null threshold. The approach of this work consists of two phases clustering and partitioning. In the clustering phase initial version of the table and clusters are found. Next phase is further dividing clusters to final tables. However, this approach is very time-consuming especially with large RDF datasets..

Our approach introduced hybrid-solution through query and data-driven approaches while keeping each approach's benefits in mind. Second, we found that none of these approaches are considered property replication in multiple tables. Third, we have introduced a new way of not only schema generation but also automated SQL generation if the data structure changes.

5 Results

This section assesses the effectiveness of our approach to creating RDF schemas in comparison to the currently available RDF storage methods.

5.1 Experimental Setup

In order to evaluate the performance of the proposed schema optimization technique, we test our experiments for querying RDF datasets using Apache Spark-SQL data engine.

In our experiments, we have loaded the generated tables from the WPT schema into Spark and performed a set of benchmark tests on different query shapes and optimization techniques.

The purpose of these experiments is to measure the performance of each created schema and compare it with the performance of the original un-optimized schema. We aim to show that our proposed optimization techniques can significantly improve the performance of SPARQL queries on large RDF datasets.

In our experiments, we opt for the WatDiv RDF benchmark with three different dataset sizes used: 1 million, 10 million, and 100 million.

The support threshold for the experiments was set at 5%, meaning that properties occurring in less than 5% of the triples were not considered significant and were pruned during schema creation. The null threshold was set to 15%, indicating that if a property had more than 15% null values, it was considered insignificant and was also pruned.

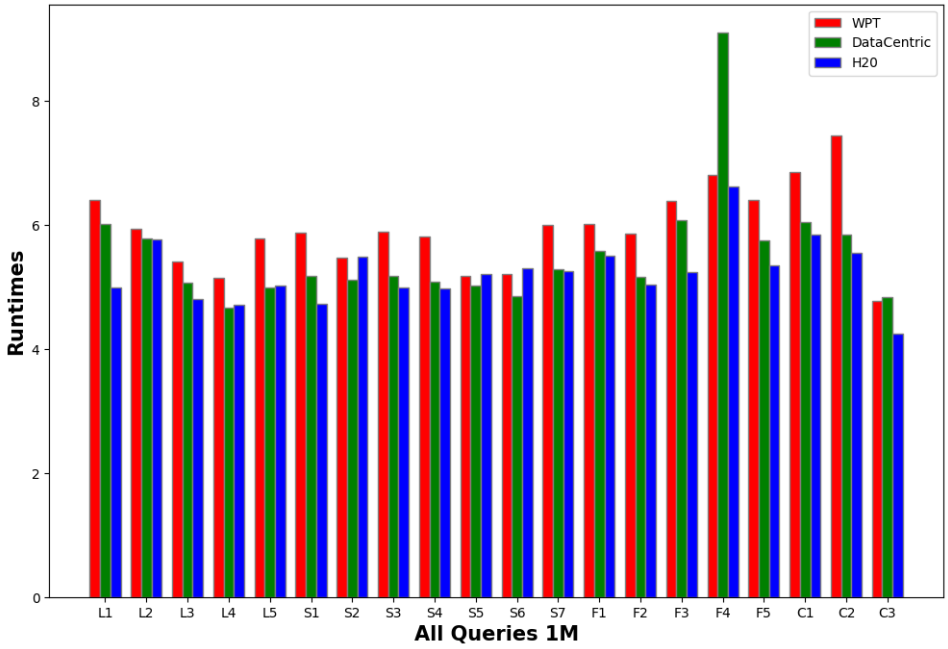
Hardware and Software Configurations. Our experiments have been executed on a Macbook Air M1 running a Mac-OS system with 8GB of unified memory. The schema generation module was built using Python and integrated with Postgresql, although the database choice can easily be changed.

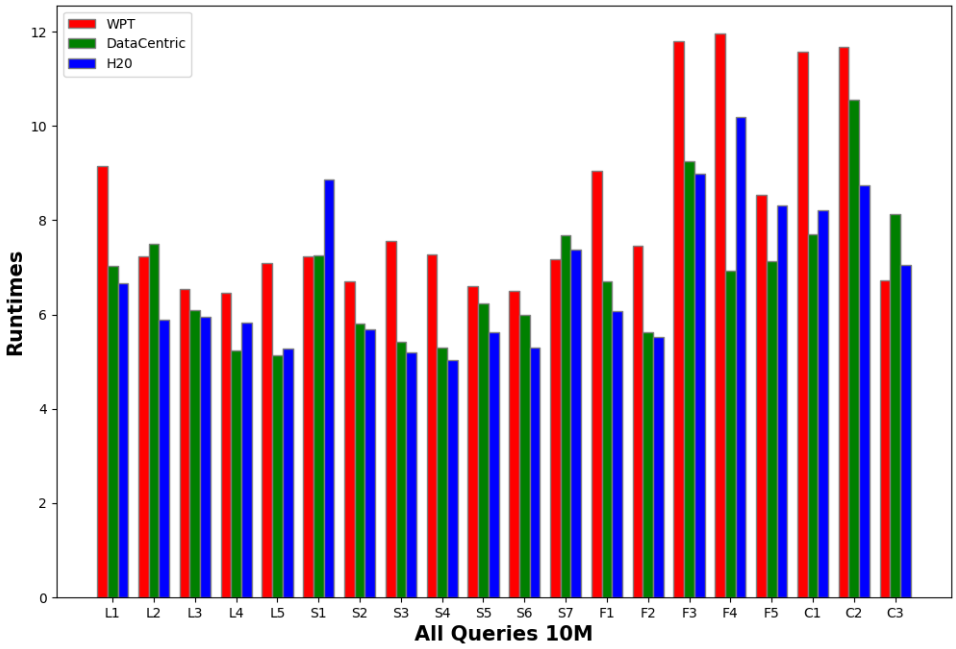
We compare our solution to existing approaches, i.e., the Wide Property Table and Data-Centeric approaches[LM09].

5.2 Experimental Evaluation

A Sample query template was picked to evaluate the performance of a solution for storing RDF data in a relational database. The template includes a collection of 20 predefined queries using the WatDiv benchmark[1]. These query templates are categorized into four groups based on their shapes: star (S), linear (L), snowflake (F), and complex (C).

[1]Query templates can be found here : https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml


WPT
DataCentric
H20
Runtimes
0
2
4
6
8
L1
L2
L3
L4
L5
S1
S2
S3
S4
S5
S6
S7
F1
F2
F3
F4
F5
C1
C2
C3
All Queries 1M


WPT
DataCentric
H20
Runtimes
0
2
4
6
8
10
12
L1
L2
L3
L4
L5
S1
S2
S3
S4
S5
S6
S7
F1
F2
F3
F4
F5
C1
C2
C3
All Queries 10M

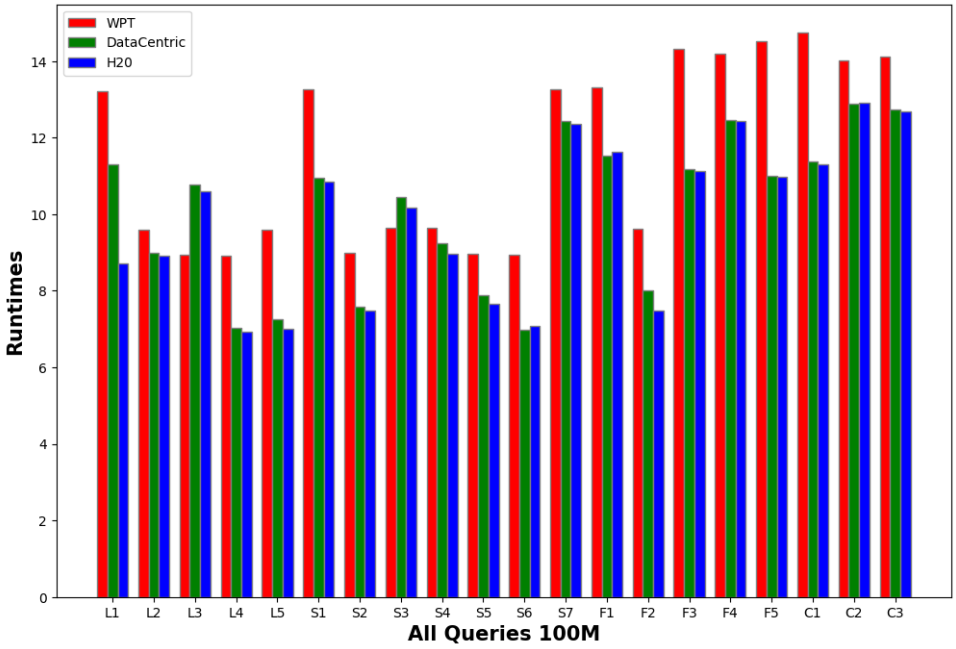


Figure 15. WatDiv benchmark queries' runtimes with hot loading

To measure the performance of the solution, the predefined queries are executed multiple times, and the average runtime is calculated. The resulting data is then plotted on a graph for comparison against another solutions.

The graph shows that the solution being tested outperforms WPT by several orders of magnitude for all categories of query shapes,meanwhile outperforms most of the data-centric schemas as well. However, to deeply analyze the reasons behind this performance difference, it is necessary to look at each of the shapes separately and provide input on the results. This would involve a more detailed analysis of the query execution plan and the specific optimizations that are being applied to each query shape.

Overall, the testing use case provides a rigorous evaluation of the solution's performance and highlights its superiority over another solution. It also demonstrates the importance of considering the shape of queries when optimizing the storage and querying of RDF data in a relational database.

1M WATDIV				10M WATDIV			100M WATDIV		
Query	WPT	DataCentric	H20	WPT	DataCentric	H20	WPT	DataCentric	H20
1	609	409	147	9443	1124	790	554212	81001	6120
2	379	326	321	1380	1808	361	14585	8148	7453
3	224	160	122	696	442	381	7626	47654	40232
4	172	107	111	637	189	337	7524	1144	1023
5	325	147	152	1206	171	195	14581	1405	1102
6	355	177	114	1380	1423	7077	579518	56990	51432
7	240	166	242	823	334	293	8079	1987	1765
8	362	179	147	1919	227	180	15600	34886	26232
9	336	162	146	1453	199	153	15407	10454	7897
10	177	153	182	738	507	279	7839	2663	2100
11	183	128	202	665	398	199	7712	1080	1198
12	403	198	193	1291	2168	1590	579843	252112	232112
13	409	266	248	8454	819	433	607531	100878	112878
14	354	175	154	1748	274	248	15267	3029	1802
15	592	437	190	133453	10355	8000	1640708	71144	67523
16	908	9012	757	157050	1018	26728	1477128	258995	251675
17	608	317	212	5120	1256	4097	2028960	60587	58242
18	945	423	346	106506	2194	3703	2542342	87015	81653
19	1704	346	257	117362	38590	6300	1232334	391403	410232
20	119	127	70	840	3412	1142	1342321	341458	323121
ALL	9404	13415	4313	552164	66098	62486	12699117	1814033	1685742

Figure 16. Query runtimes in ms

5.2.1 Linear queries

In linear queries, we see that The hybrid schema generated by our approach significantly outperforms the WPT schema performance . This difference comes from the input size of each joined table. In our algorithm, we try to decreasing the sparsity of the table (reducing null values). By means of reducing input size of the WPT table we are also improving the scan time as well. But there is an interesting aspect while comparing the linear queries of L1 with others. L1 performs better than others while at the same time being the subpattern for them. The reason is in the underlying data. Thus in the first query, we have used org: caption property as the second join condition, which has only 0.0002 selectivity. In our solution, since this predicate has been used as a separate VP table, we have significantly reduced the size of the second table. But in all other cases joined table does not have such low selectivity.

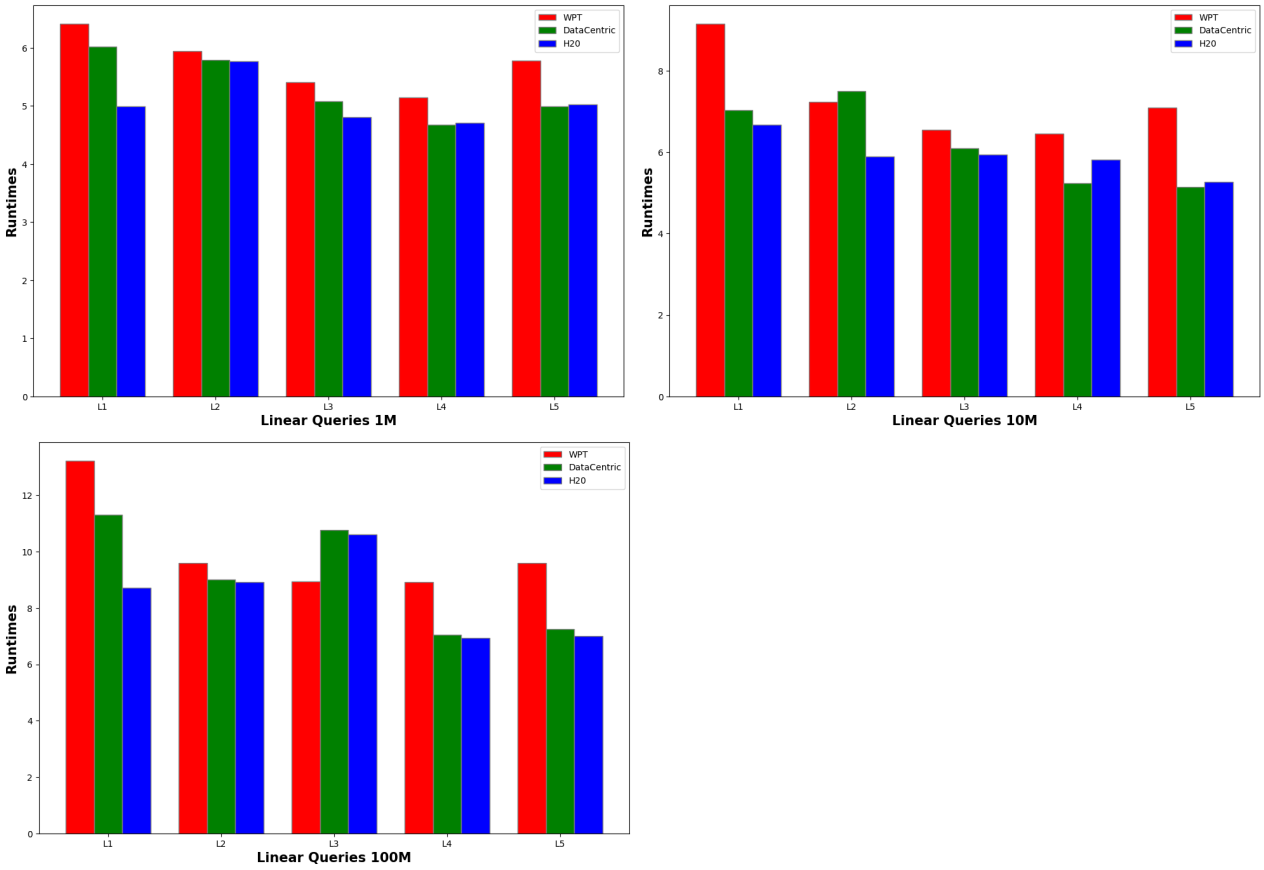


Figure 17. Linear queries runtimes with different scales

5.2.2 Star queries

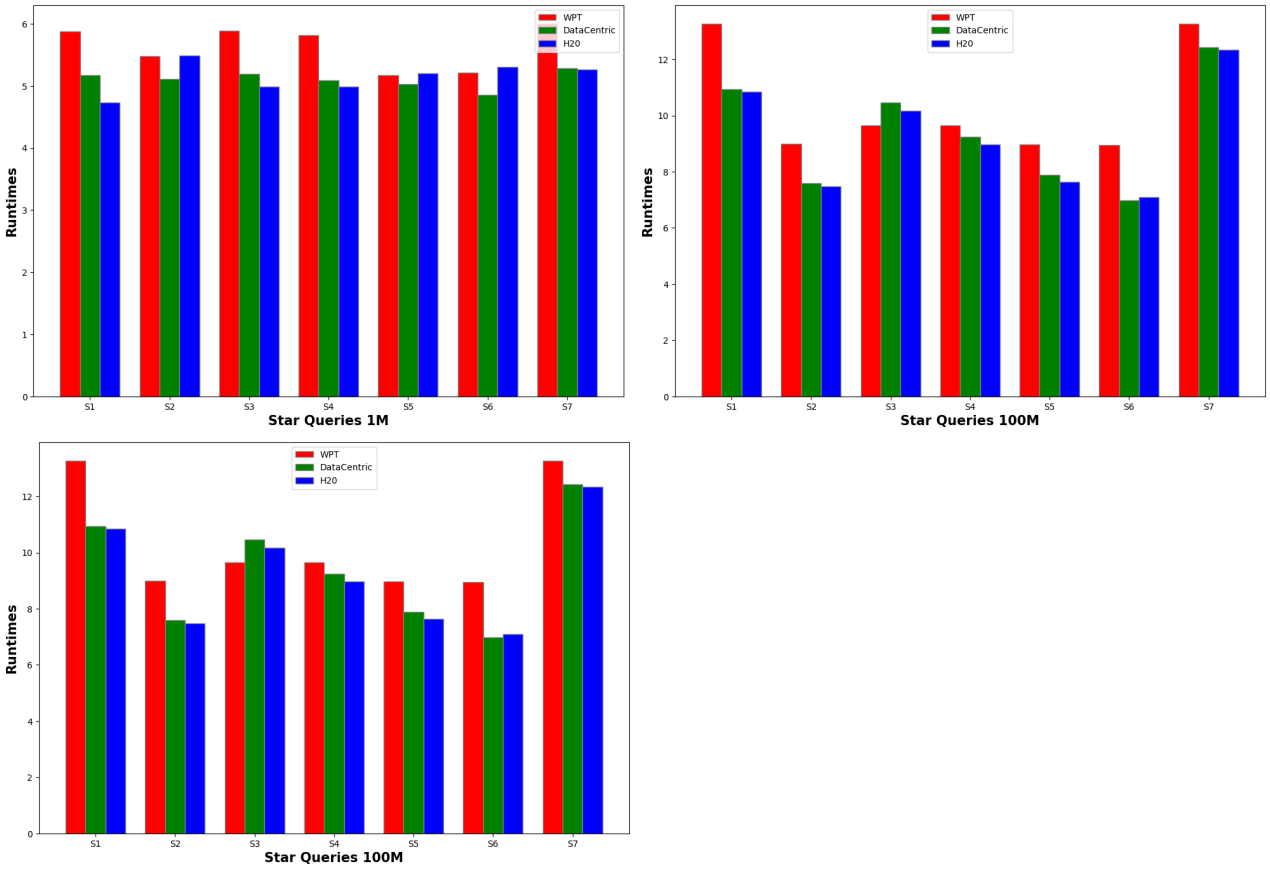


Figure 18. Star queries runtimes with different scales

In star queries, we see that our solution is not the best-performing. WPT tables are optimized primarily for star queries, and to retrieve stars, you don't need to do any join. However, our optimized schema needs to have multiple joins in order to retrieve the same result. This figure shows that our solution is still outperforms in 5 out of 7 cases. The underlying reason is that the size of the generated schema tables (especially star tables) is not so big that splitting the predicate into the different tables and keeping them in the same table does not have a huge difference.

5.2.3 Snowflake queries

Since snowflake is a more complex version of the star and linear queries, that is why we need to consider different scenarios. Our sample can easily show that the WPT schema

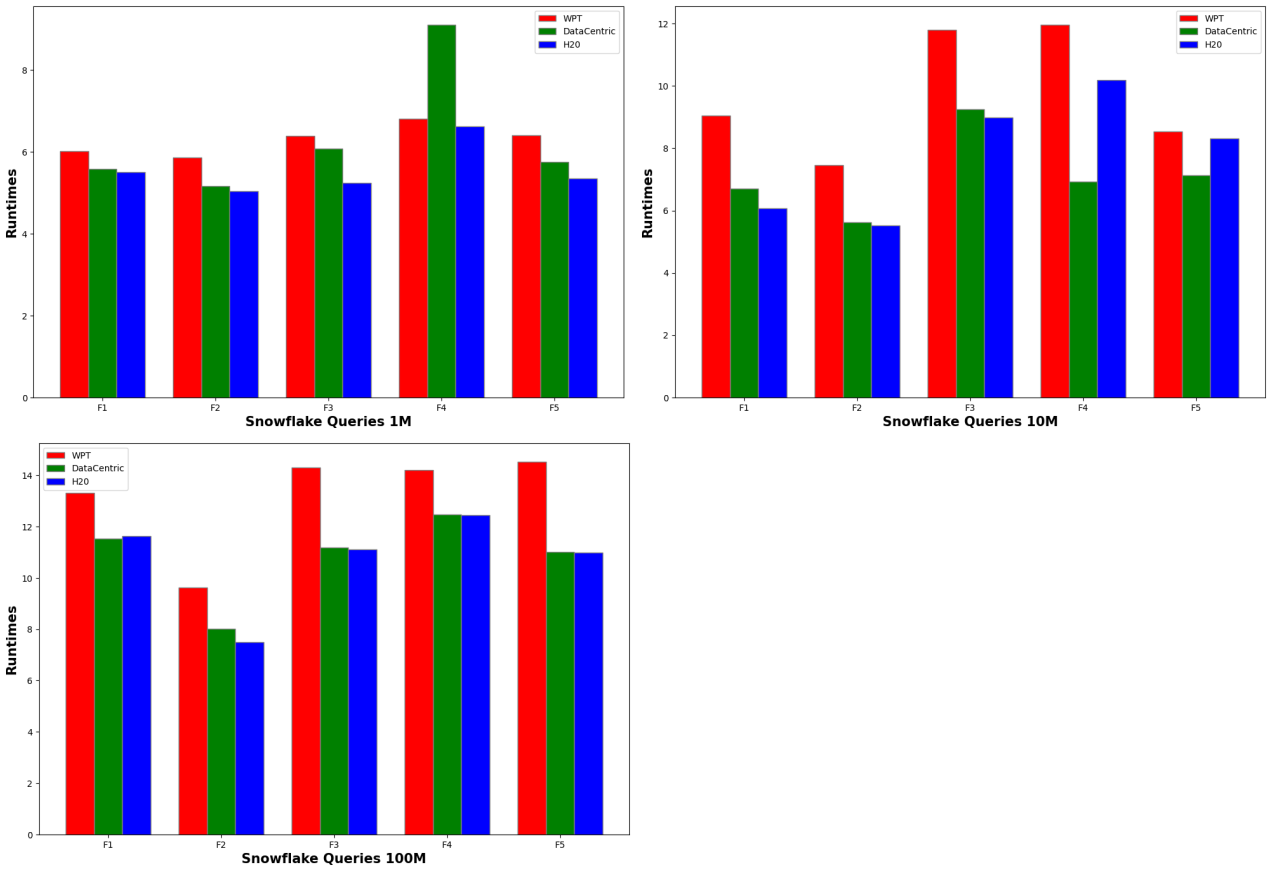


Figure 19. Snowflake queries runtimes with different scales

is not performing well against our schema. If we look at some edge cases, such as F3 and F4, we can see that there is a subquery and filter that has been used against a huge wide property table which creates a bottleneck in the execution time.

5.2.4 Complex queries

A significant difference comes when the query is complex. It means a lot of joins and filter conditions that can be overwhelmed the system. In this graph, we can see a huge difference in terms of the runtime. The main difference is coming from using self-joined in the WPT table, which is quite a big table. If we look at the graph, we can see one anomaly, which, although queries C1 and C2 have significant improvement, however, C3 is almost at the same level. This huge difference is the structure of the C3 which is entirely close to the Star schema. Since it retrieves huge property values, it is considered a complex query.

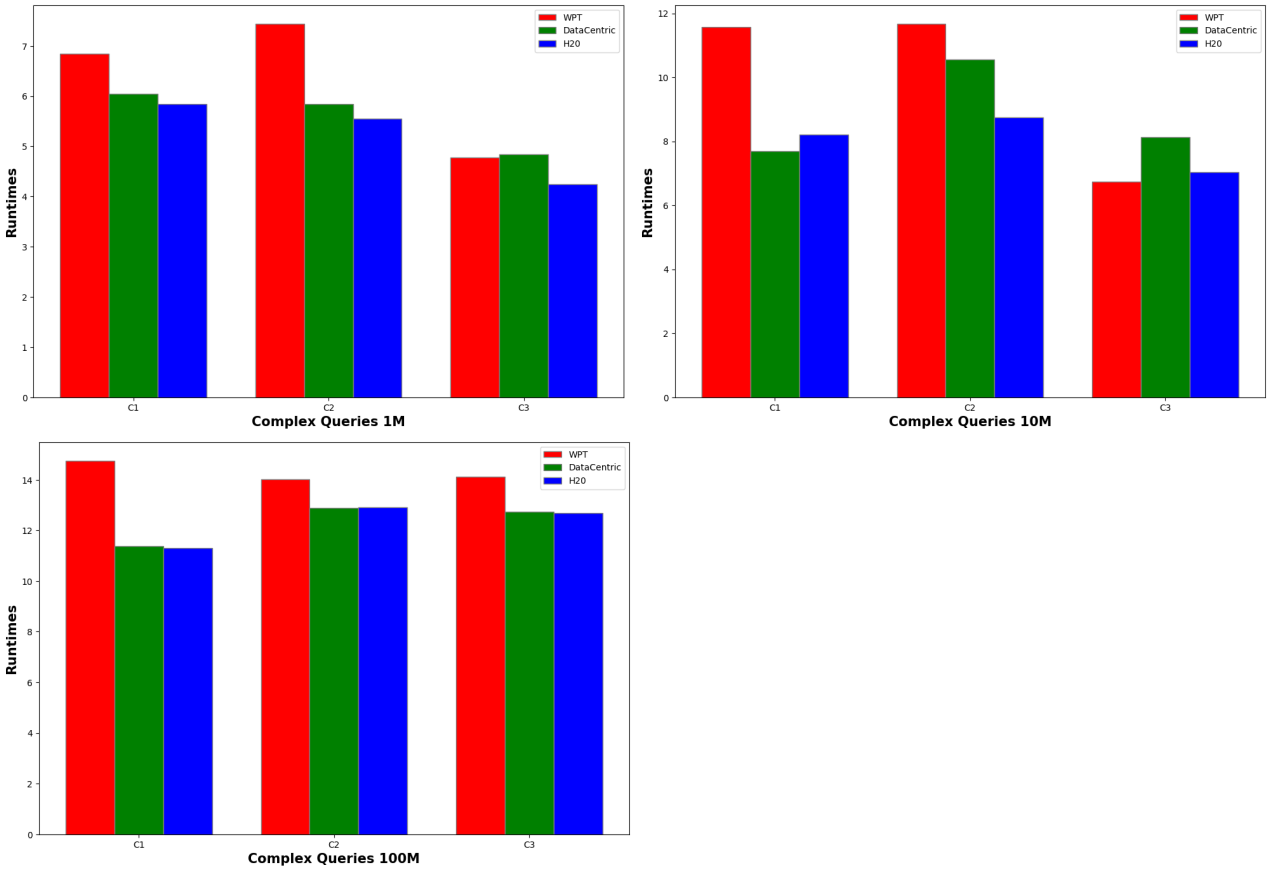


Figure 20. Complex queries runtime with different scales

6 Conclusion & Future Work

The exponential growth and interconnectedness of data in various domains creates a need for the development of efficient and scalable solutions for processing Knowledge Graphs(KGs). The Resource Description Framework(RDF) is one of the popular graph data models for representing KGs. However, it faces significant challenges in storing and managing KGs using relational schemas.

In our thesis, we have introduced a new hybrid schema generation algorithm using data and query workload techniques. Our algorithm has shown great potential for managing RDF data. Meanwhile, we have created an algorithm for smooth query translation from old queries to new ones. It has also opened so many interesting improvements and challenges for future research.

6.1 Exploiting Hybrid Schema in Batch Processing

In our thesis, we have designed a workload-driven approach to manage and optimize the processing of RDF data queries. The approach involves storing frequently-used queries to determine the actual workload of the system. Thus , we can identify the most important queries and optimize the system accordingly to improve the overall performance.

However, the frequency of queries can change over time based on user requirements, and our solution is flexible enough to accommodate these changes. We take snapshots of query statements at various points in time and apply any new changes to the existing schema.

To ensure seamless transitioning of queries from the old schema to the new schema, we leverage the benefits of our auto query mapping solution. This solution adapts queries to the new schema automatically, without any manual intervention needed. By doing so, we ensure that our system remains efficient and effective, even as the schema changes over time.

6.2 Finding a correlation between properties using Machine Learning Algorithms

In our thesis, we used query and data profiling techniques to select the best hybrid schema for a given dataset. It could be interesting to explore the use of machine learning techniques to automate this process even further, potentially leading to more effective schema selection.

6.3 Extension of auto query mapping solution

Our thesis proposes an auto query mapping solution to adapt queries to new versions of the schema. This solution could be extended to handle more complex schema changes, or to include additional features such as automatically generate new queries from native SPARQL itself directly.

References

[apa] Jena tutorials.

[A14] Güneş Aluç, M. Tamer Özsu, and Khuzaima Daudjee. Workload matters. *Proceedings of the VLDB Endowment*, 7(10):837–840, 2014.

[Bec01] David Beckett. The design and implementation of the redland rdf application framework. *Proceedings of the 10th international conference on World Wide Web*, 2001.

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. *The Semantic Web — ISWC 2002*, page 54–68, 2002.

[dat] 3store.

[dev] Eclipse RDF4J developers. Documentation · eclipse rdf4j™: The eclipse foundation.

[FWF+19] Marcin Wylot TU Berlin / Fraunhofer FOKUS, Marcin Wylot, TU Berlin / Fraunhofer FOKUS, Manfred Hauswirth TU Berlin / Fraunhofer FOKUS, Manfred Hauswirth, Philippe Cudré-Mauroux University of Fribourg, Philippe Cudré-Mauroux, University of Fribourg, Sherif Sakr University of Tartu, Sherif Sakr, and et al. Rdf data storage and query processing schemes: A survey: Acm computing surveys: Vol 51, no 4, Jul 2019.

[GHH14] Rong Gu, Wei Hu, and Yihua Huang. Rainbow: A distributed and hierarchical rdf triple store with dynamic scalability. *2014 IEEE International Conference on Big Data (Big Data)*, 2014.

[Hor08] Ian Horrocks. Ontologies and the semantic web. *Communications of the ACM*, 51(12):58–67, 2008.

[KZJ+19] Yasar Khan, Antoine Zimmermann, Alokkumar Jha, Vijay Gadepally, Mathieu D'Aquin, and Ratnesh Sahay. One size does not fit all: Querying web polystores. *IEEE Access*, 7:9598–9617, 2019.

[lin16] Efficient query processing in rdf databases. *Linked Data Management*, page 145–164, 2016.

[LM09] Justin J. Levandoski and Mohamed F. Mokbel. Rdf data-centric storage. *2009 IEEE International Conference on Web Services*, 2009.

[Los22] Peter Loshin. What is rdf (resource description framework)?, Feb 2022.

[May21] Mohit Mayank. A guide to the knowledge graphs, Sep 2021.

[MSP+04] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. Rstar. *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, 2004.

[Ont22] Dec 2022.

[ora23] Oracle database documentation - oracle database, Apr 2023.

[Pos] PostgreSQL. About.

[PPEB15] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter Boncz. Deriving an emergent relational schema from rdf data. *Proceedings of the 24th International Conference on World Wide Web*, 2015.

[psy] Psycopg 2.9.6 documentation.

[sof23] Frequent pattern (fp) growth algorithm in data mining, Mar 2023.

[SPZSL16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2rdf. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.

[Wil06] Kevin Wilkinson. Jena property table implementation. 2006.

[Wor22] Summer Worsley. What is python? - the most versatile programming language, Mar 2022.

Appendix

I. Basic WATDIV query workload

```
# L1:
{
  ?v0 wsdbm:subscribes  %v1% .
  ?v2 sorg:caption  ?v3 .
  ?v0 wsdbm:likes ?v2 .
}

#L2:
{
  %v0%  gn:parentCountry  ?v1 .
```

```
  ?v2 wsdbm:likes wsdbm:Product0 .
  ?v2 sorg:nationality  ?v1 .
}

#L3:
{
  ?v0 wsdbm:likes ?v1 .
  ?v0 wsdbm:subscribes  %v2% .
}

#L4:
{
  ?v0 og:tag  %v1% .
  ?v0 sorg:caption  ?v2 .
}

#L5:
{
  ?v0 sorg:jobTitle ?v1 .
  %v2%  gn:parentCountry  ?v3 .
  ?v0 sorg:nationality  ?v3 .
}

#S1:
{
  ?v0 gr:includes ?v1 .
  %v2%  gr:offers ?v0 .
  ?v0 gr:price  ?v3 .
  ?v0 gr:serialNumber ?v4 .
  ?v0 gr:validFrom  ?v5 .
  ?v0 gr:validThrough ?v6 .
  ?v0 sorg:eligibleQuantity ?v7 .
  ?v0 sorg:eligibleRegion ?v8 .
  ?v0 sorg:priceValidUntil  ?v9 .
}

#S2:
{
  ?v0 dc:Location ?v1 .
  ?v0 sorg:nationality  %v2% .
  ?v0 wsdbm:gender  ?v3 .
  ?v0 rdf:type  wsdbm:Role2 .
}
```

```
#S3:
{
  ?v0 rdf:type  %v1% .
  ?v0 sorg:caption  ?v2 .
  ?v0 wsdbm:hasGenre  ?v3 .
  ?v0 sorg:publisher  ?v4 .
}

#S4:
{
  ?v0 foaf:age  %v1% .
  ?v0 foaf:familyName ?v2 .
  ?v3 mo:artist ?v0 .
  ?v0 sorg:nationality  wsdbm:Country1 .
}

#S5:
{
  ?v0 rdf:type  %v1% .
  ?v0 sorg:description  ?v2 .
  ?v0 sorg:keywords ?v3 .
  ?v0 sorg:language wsdbm:Language0 .
}

#S6:
{
  ?v0 mo:conductor  ?v1 .
  ?v0 rdf:type  ?v2 .
  ?v0 wsdbm:hasGenre  %v3% .
}

#S7:
{
  ?v0 rdf:type  ?v1 .
  ?v0 sorg:text ?v2 .
  %v3%  wsdbm:likes ?v0 .
}

#F1:
{
  ?v0 og:tag  %v1% .
  ?v0 rdf:type  ?v2 .
```

```
  ?v3 sorg:trailer  ?v4 .
  ?v3 sorg:keywords ?v5 .
  ?v3 wsdbm:hasGenre  ?v0 .
  ?v3 rdf:type  wsdbm:ProductCategory2 .
}

#F2:
{
  ?v0 foaf:homepage ?v1 .
  ?v0 og:title  ?v2 .
  ?v0 rdf:type  ?v3 .
  ?v0 sorg:caption  ?v4 .
  ?v0 sorg:description  ?v5 .
  ?v1 sorg:url  ?v6 .
  ?v1 wsdbm:hits  ?v7 .
  ?v0 wsdbm:hasGenre  %v8% .
}


#F3:
 {
  ?v0 sorg:contentRating  ?v1 .
  ?v0 sorg:contentSize  ?v2 .
  ?v0 wsdbm:hasGenre  %v3% .
  ?v4 wsdbm:makesPurchase ?v5 .
  ?v5 wsdbm:purchaseDate  ?v6 .
  ?v5 wsdbm:purchaseFor ?v0 .
}

#F4:
{
  ?v0 foaf:homepage ?v1 .
  ?v2 gr:includes ?v0 .
  ?v0 og:tag  %v3% .
  ?v0 sorg:description  ?v4 .
  ?v0 sorg:contentSize  ?v8 .
  ?v1 sorg:url  ?v5 .
  ?v1 wsdbm:hits  ?v6 .
  ?v1 sorg:language wsdbm:Language0 .
  ?v7 wsdbm:likes ?v0 .
}

#F5:
```

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

#C1:
{
  ?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:
{
  ?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:
{
  ?v0 wsdbm:likes ?v1 .
  ?v0 wsdbm:friendOf  ?v2 .
  ?v0 dc:Location ?v3 .
  ?v0 foaf:age  ?v4 .
  ?v0 wsdbm:gender  ?v5 .
  ?v0 foaf:givenName  ?v6 .
```

```
}
```

II. Access to Code

The code used to obtain the results can be found in this GitHub repository given below:
`https://github.com/faridvaliyev1/Comet`

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Farid Valiyev**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Query Workload-Driven Schema Optimization For Processing Large RDF Datasets,
   (title of thesis)

   supervised by Mohamed Ragab, Riccardo Tommasini, Alexander Nolte.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Farid Valiyev
09/05/2023